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

# BVM: Reformulação da metodologia de verificação funcional VeriSC

**Helder Fernando de Araújo Oliveira**

Campina Grande, Paraíba, Brasil

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

# BVM: Reformulação da metodologia de verificação funcional VeriSC

## Helder Fernando de Araújo Oliveira

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 Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Orientadores

**Elmar Uwe Kurt Melcher**

**Joseana Macêdo Fechine**

Campina Grande, Paraíba, Brasil

**FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG**

O482b Oliveira, Helder Fernando de Araújo.
BVM: reformulação da metodologia de verificação funcional VeriSC / Helder Fernando de Araújo Oliveira.– Campina Grande, 2010.
139f. : il.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientadores: Prof. Dr. Elmar Uwe Kurt Melcher, Profª. Drª. Joseana Macêdo Fechine.

1. Verificação Funcional. 2. Metodologia – Reformulação. 3. BVM – Brazil – IP Verification Methodology. 4. VeriSC. I. Título.

CDU 004.3142.23 (043)

*Dedico estre trabalho às pessoas que mais amo nesta vida: Minha mãe, meu pai e meu irmão.*

# Agradecimentos

Agradeço primeiramente a Deus, por ter me iluminado e guiado em todos os momentos de minha vida.

Agradeço especialmente à minha mãe, por todo seu esforço, trabalho, carinho e abnegação, para que eu pudesse chegar aonde estou. Ao meu pai, por todo seu apoio e incentivo para que eu continuasse sempre lutando por meus objetivos. Ao meu irmão, por sempre ter me motivado a seguir em frente. A minha namorada Rayanna Coelho, por toda sua compreensão, estímulo e amor durante o tempo em que estamos juntos.

Aos professores Elmar e Joseana, por sua dedicação, amizade e por todas as suas contribuições em minha vida acadêmica e pessoal.

A todos os meus familiares, amigos e colegas que sempre me apoiaram ao longo deste caminho.

Às equipes do Brazil-IP, por terem contribuído no andamento deste trabalho, em especial ao colega Jorgeluis.

As funcionárias da Copin, Aninha e Vera, por toda atenção e assistência prestada.

E, finalmente, a todos aqueles que, de alguma forma, contribuíram para que eu pudesse atingir meus objetivos.

# Resumo

O processo de desenvolvimento de um circuito digital complexo pode ser composto por diversas etapas. Uma delas é a verificação funcional. Esta etapa pode ser considerada uma das mais importantes, pois tem como objetivo demonstrar que as funcionalidades do circuito a ser produzido estão em conformidade com a sua especificação. Porém, além de ser uma fase com grande consumo de recursos, a complexidade da verificação funcional cresce diante da complexidade do *hardware* a ser verificado. Desta forma, o uso de uma metodologia de verificação funcional eficiente e de ferramentas que auxiliem o engenheiro de verificação funcional são de grande valia. Neste contexto, este trabalho realiza uma reformulação da metodologia de verificação funcional VeriSC, originando uma nova metodologia, denominada BVM (*Brazil-IP Verification Methodology*). VeriSC é implementada em SystemC e utiliza as bibliotecas SCV (*SystemC Verification Library*) e BVE (*Brazil-IP Verification Extensions*), enquanto BVM é implementada em SystemVerilog e baseada em conceitos e biblioteca de OVM (*Open Verification Methodology).* Além disto, este trabalho visa a adequação da ferramenta de apoio à verificação funcional eTBc *(Easy Testbench Creator)* para suportar BVM. A partir do trabalho realizado, é possível constatar, mediante estudos de caso no âmbito do projeto Brazil-IP, que BVM traz um aumento da produtividade do engenheiro de verificação na realização da verificação funcional, em comparação à VeriSC.

# Abstract

The development process of a complex digital circuit can consist of several stages. One of them is the functional verification. This stage can be considered one of the most important because it aims to demonstrate that a circuit functionality to be produced is in accordance with its specification. However, besides being a stage with large consumption of resources, the complexity of functional verification grows according to the complexity of the hardware to be verified. Thus, the use of an effective functional verification methodology and tools to help engineer the functional verification are of great value. Within this context, this work proposes a reformulation of the functional verification methodology VeriSC, resulting in a new methodology called BVM (*Brazil-IP Verification Methodology*). VeriSC is implemented in SystemC and uses the SCV (SystemC Verification Library) and BVE (*Brazil-IP Verification Extensions*) libraries, while BVM is implemented and based on SystemVerilog and OVM (Open Verification Methodology) concepts and library. Furthermore, this study aims the adequacy of the functional verification tool eTBc (testbench Easy Creator), to support BVM. From this work it can be seen, based on case studies under the Brazil-IP project, that BVM increase the productivity of the engineer in the functional verification stage when compared to VeriSC.

# Sumário

**1 Introdução**.....1
1.1 Motivação.....1
1.2 O Brazil-IP.....3
1.3 Definição do problema.....4
1.4 Objetivos.....5
1.5 Organização.....6
**2 Conceitos fundamentais**.....7
2.1 Verificação.....7
2.1.1 Verificação formal ou estática.....7
2.1.2 Verificação dinâmica ou funcional.....8
2.1.3 Verificação híbrida ou semi-formal.....9
2.2 Plano de verificação.....9
2.3 *Testbench*.....10
2.4 Modelagem em nível de Transação (*Transaction Level Modelling*).....13
2.5 Geração de estímulos.....13
2.6 Cobertura.....14
2.7 SystemVerilog.....15
**3 Análise comparativa entre BVM e VeriSC**.....18
3.1 VeriSC.....18
3.2 Outras metodologias.....19
3.3 BVM.....22
3.3.1*Testbench conception*.....22
3.3.1.1 *Single refmod*.....22
3.3.1.2 *Double refmod*.....23
3.3.1.3 *DUV emulation*.....23
3.3.2 *Hierarchical refmod decomposition*.....25
3.3.2.1 *Refmod decomposition*.....25
3.3.2.2 *Single hierarchical refmods*.....25
3.3.2.3 *Hierarchical refmods verification*.....26
3.3.3 *Hierarchical testbench*.....26
3.3.3.1 *Double hierarchical refmods*.....27
3.3.3.2 *Hierarchical DUV emulation*.....27
3.3.3.3 *Hierarchical DUV*.....28
3.3.4 *Full testbench*.....29
3.4 *Full testbench analysis*.....30

3.5 A ferramenta eTBc....32
3.6 A biblioteca OVM....34
3.7 Diferenças entre BVM e VeriSC....39
**4 Resultados**....45
4.1 Implementação de BVM....45
4.1.1 *Testbench conception*....45
4.1.1.1 *Single refmod*....45
4.1.1.2 *Double refmod*....50
4.1.1.3 *DUV emulation*....53
4.1.2 *Hierarchical refmod decomposition*....54
4.1.2.1 *Refmod decomposition*....54
4.1.2.2 *Single hierarchical refmods*....54
4.1.2.3 *Hierarchical refmods verification*....55
4.1.3 *Hierarchical testbench*....55
4.1.3.1 *Double hierarchical refmods*....55
4.1.3.2 *Hierarchical DUV emulation*....55
4.1.3.3 *Hierarchical DUV*....55
4.1.4 *Full testbench*....56
4.2 *Full testbench analysis*....56
4.3 Estudo comparativo preliminar entre BVM e VeriSC....56
**5 Considerações finais e sugestões para trabalhos futuros**....62
5.1 Sugestões para trabalhos futuros....63
**Glossário**....68
**A. Código fonte dos *template patterns***....72
A.1 *axi_cover*....72
A.2 *checker*....73
A.3 *duv*....75
A.4 *duv_emulation*....76
A.5 *duv_hierarchical*....77
A.6 *gene_clock*....78
A.7 *in_actor*....78
A.8 *Makefile_double_refmod*....80
A.9 *Makefile_duv*....80
A.10 *Makefile_duv_emulation*....81
A.11 *Makefile_full*....81
A.12 *Makefile_full_analysis*....82
A.13 *Makefile_hierarchical_refmod*....82

A.14 *Makefile_single_refmod*.....83
A.15 *out_actor*.....83
A.16 *refmod_caller*.....84
A.17 *sink*.....86
A.18 *source*.....87
A.19 *tb_double_refmod*.....88
A.20 *tb_duv*.....90
A.21 *tb_duv_analysis*.....92
A.22 *tb_hierarchical_refmod*.....95
A.23 *tb_single_refmod*.....97
A.24 *tb_tcl*.....98
A.25 *tdriver*.....99
A.26 *tdriver_duv*.....100
A.27 *tmonitor*.....101
A.28 *tmonitor_duv*.....103
A.29 *top*.....104
A.30 *top_analysis*.....106
A.31 *top_tcl*.....107
A.32 *trans*.....108
**B. *An OVM based Functional Verification Methodology with Testbench Generator***.....110
Abstract.....110
B.1 I*ntroduction*.....110
B.2 *VeriSC methodology*.....111
B.3 *BVM methodology*.....113
B.4 eTBc *tool*.....114
B.5 BVM x VeriSC.....116
B.6 *Results*.....118
B.7 *Conclusion*.....121

# Lista de abreviações

AAC - *Advanced Audio Coding*

AVM - *Advanced Verification Methodology*

BVE-COVER - *Brazil-IP Verification Extension*

BVM - *Brazil-IP Verification Methodology*

CDV - *Coverage Driven Verification*

CLM - *Cycle-level Modelling*

CPU - *Central Processing Unit*

DPI - *Direct Programming Interface*

DUT - *Design Under Test*

DUV - *Design Under Verification*

EDA - *Electronic Design Automation*

eDL - *eTBc Design Language*

ESL- *Electronic System Level*

eTBc - *Easy Testbench Creator*

eTL - *eTBc Template Language*

FIFO - *First In First Out*

HDL - *Hardware Description Language*

HDMI - *High-Definition Multimedia Interface*

HVL - *Hardware Verification Language*

IEEE - *Institute of Electrical and Electronics Engineers*

ISO - *International Organization for Standardization*

IP - *Intellectual Property*

IPCM/URM - *Incisive Plan-to-Closure Universal Reuse Methodology*

IUS - *Incisive Unified Simulator*

MPEG - *Moving Picture Experts Group*

OSCI - *Open SystemC Initiative*

OVC - *OVM verification components*

OVM - *Open Verification Methodology*

RTL - *Register Transfer Level*

RVM - *Reference Verification Methodology*

SBCCI - *Symposium on Integrated Circuits and Systems Design*

SCV - *SystemC Verification Library*

SoC - *System on a chip*

TLM - *Transaction Level Modelling*

TLN - *Transaction Level Netlist*

UNISIM - *UNIted SIMulation environment*

USB - *Universal Serial Bus*

UVC - *Universal Verification Component*

VHDL - *VHSIC Hardware Description Language*

VMM - *Verification Methodology Manual*

# Lista de figuras

1 Fases para o processo de desenvolvimento de *hardware* na ordem proposta por VeriSC..... 1

2 Modelo de reconvergência para *equivalence checking* …..... 7

3 Modelo de reconvergência para *property checking* …..... 8

4 Modelo de reconvergência para verificação dinâmica …..... 8

5 Estrutura genérica de um *testbench* …..... 10

6 Diagrama do *testbench* da metodologia VeriSC …..... 11

7 Representação do subpasso *Single refmod* …..... 23

8 Representação do subpasso *Double refmod* …..... ..23

9 Representação do artificio usado para emular o DUV ..... 24

10 Representação do subpasso *DUV emulation* …..... 24

11 Representação da decomposição do Modelo de Referência …..... 25

12 Representação do subpasso *Single hierarchical refmods* para o Modelo de Referência 1 …. .25

13 Representação do subpasso *Single hierarchical refmods* para o Modelo de Referência 2 …. .26

14 Representação do subpasso *Hierarchical refmods verification* …..... 26

15 Representação do subpasso *Double hierarchical refmods* para o Modelo de Referência 1 … 27

16 Representação do subpasso *Double hierarchical refmods* para o Modelo de Referência 2 … 27

17 Representação do subpasso *Hierarchical DUV emulation* para o Modelo de Referência 1 …. 28

18 Representação do subpasso *Hierarchical DUV emulation* para o Modelo de Referência 2 …. 28

19 Representação do subpasso *Hierarchical DUV* para o Modelo de Referência 1 …..... 28

20 Representação do subpasso *Hierarchical DUV* para o Modelo de Referência 2 …..... 29

21 Representação do passo *Full testbench* …..... 29

22 Protocolo de comunicação do DUV 1 com *Actor* 1 e do DUV 1 com o DUV 2 …..... 30

23 Possível inserção de monitores entre os DUV …..... 30

24 Representação da decomposição hierárquica do DUV …..... 31

25 Inserção de um Monitor para a detecção do erro entre o DUV 1 e DUV 2 …..... 31

26 Representação da ferramenta eTBc …..... 32

27 Ambiente de verificação em OVM …..... 36

28 Hierarquia das classes que compõe a biblioteca OVM …..... 37

29 Representação de um DPCM …..... 47

# Lista de tabelas

1 Respostas sobre a questão 1 ..... 57

2 Respostas sobre a questão 2 ..... 58

3 Respostas sobre a questão 3 ..... 58

4 Respostas sobre a questão 4 ..... 58

5 Respostas sobre a questão 5 ..... 59

6 Respostas sobre a questão 6 ..... 59

7 Respostas sobre a questão 7 ..... 59

# Lista de quadros

1 Comparação entre VMM e OVM …..... 21

2 Questões sobre BVM e VeriSC …..... 57

# Lista de códigos

1 Trecho de código do *template pattern* para gerador de estímulos em VeriSC …..... 33

2 Trecho de código do *template pattern* para gerador de estímulos em BVM …..... 33

3 Código para *testbench element Source* …..... 37

4 Exemplo de Cobertura funcional em VeriSC …..... 39

5 Exemplo de Cobertura funcional em BVM …..... 40

6 Exemplo de Cobertura funcional para valores ilegais em VeriSC …..... 40

7 Exemplo de Cobertura funcional para valores ilegais em BVM …..... 40

8 Exemplo de *Cross Coverage* em BVM …..... 41

9 *Constraints* para gerar estímulos em VeriSC …..... 42

10 *Constraints* para gerar estímulos em BVM …..... 42

11 Trecho de código de módulo responsável por conectar elementos do *testbench* em VeriSC ...42

12 Trecho de código de módulo responsável por conectar elementos do *testbench* em BVM ….. 43

13 Código para *template pattern* do arquivo trans …..... 46

14 Código de uma TLN exemplo …..... 48

15 Código gerado pela ferramenta eTBc utilizando TLN do DPCM e *template* do arquivo trans ...49

16 Código do *template Makefile_double_refmod* …..... 51

17 Código do *template pattern source* …..... 52

18 Código do *template pattern checker* …..... 52

# Capítulo 1

# 1 Introdução

Diante da complexidade de circuitos digitais atuais, verificar as funcionalidades de um projeto de *hardware* (verificação funcional) torna-se uma tarefa cada vez mais difícil. De acordo com algumas empresas da área (CADENCE DESIGN SYSTEMS, 2010; MENTOR GRAPHICS, 2010a) e alguns estudiosos (BERGERON, 2006; MOLINA, 2007), a etapa de verificação funcional é um dos maiores gargalos dentro de um projeto de desenvolvimento de *hardware*. Estima-se que cerca de 70% dos recursos de um projeto de *hardware* são gastos na etapa de verificação funcional (BERGERON, 2003, 2006). Assim sendo, ferramentas e metodologias de verificação funcional eficientes são de grande importância, diminuindo os riscos, tempo e custos necessários para realização desses projetos.

## *1.1 Motivação*

Sistemas complexos podem ser formados por diversos componentes de *hardware* que desempenham funcionalidades específicas. Tais componentes são blocos de *hardware* dedicados, denominados IP *cores* (Intelectível *Property core*). Segundo Dueñas (2004), 65% dos IP *cores* falham em sua primeira prototipação em silício e 70% desses casos ocorrem devido à má elaboração do processo de verificação funcional. 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 (BRUNELLI et al., 2001). Isto implica que esforços extra são necessários na fase de verificação funcional, a fim de evitar prejuízos no decorrer do projeto.

Diante do exposto, em 2007, uma aluna de doutorado* da Universidade Federal de Campina Grande propôs uma metodologia para verificação funcional, denominada VeriSC (SILVA, 2007). A ordem proposta por VeriSC para o processo de desenvolvimento de *hardware* é apresentada na Figura 1.


**Figura 1 - Fases do processo de desenvolvimento de *hardware* na ordem proposta por VeriSC (SILVA, 2007).**

* Karina Rocha Gomes da Silva

A fase de **especificação do *hardware*** consiste no estudo e documentação dos requisitos necessários para a sua construção. Segundo Silva (2007), esta especificação deve possuir 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, e também informações de baixo nível, por exemplo, especificação e descrição de pinos a serem utilizados.

A fase de **especificação da verificação funcional** é a fase na qual se documenta aspectos importantes que devem ser verificados no *hardware*, tais como suas funcionalidades, estímulos que serão utilizados na verificação do projeto, a faixa de valores destes estímulos, variáveis que serão utilizadas, etc.

A fase de **implementação do *testbench*** é voltada para a construção do ambiente de verificação funcional do DUV (*Design Under Verification*), também conhecido como DUT (*Design Under Test*). Este ambiente é conhecido como *testbench*. O DUV é uma descrição do dispositivo (ou parte dele) a ser desenvolvido, codificado em alguma linguagem de descrição de *hardware (HDL - Hardware Description Language)*. Uma diferença significava entre esta e outras linguagens de programação voltadas para *software* é que a sintaxe e a semântica de linguagens HDL incluem notações explícitas que expressam concorrência e tempo. A etapa de implementação do *testbench* apresenta grande impacto na fase de verificação funcional, visto que é neste ambiente que o DUV será inserido e receberá estímulos, gerando saídas que serão comparadas com as saídas desejadas.

A **implementação do DUV** é caracterizada pela implementação do código RTL (*Register Transfer Language*). O DUV é desenvolvido objetivando alcançar a especificação do dispositivo. Este código é escrito em alguma linguagem de descrição de *hardware*, tal como Verilog, ou VHDL por exemplo.

De acordo com Bergeron (2003), **verificação funcional** é um processo usado para demonstrar que o objetivo do projeto é preservado em sua implementação. A verificação funcional constitui umas das etapas do processo de desenvolvimento de *hardware,* a qual é realizada a partir de simulação. Nela dois modelos são comparados: o que está sendo desenvolvido (o DUV) e outro que reflete a especificação. Uma maneira de realizar essa comparação consiste em inserir valores pseudo-aleatórios na entrada do DUV, o qual será comparado com um modelo ideal, chamado de Modelo de Referência. Tal modelo “é uma implementação executável e, por definição, reflete a especificação” (SILVA, 2007, p. 5 ). As saídas produzidas por ambos deverão ser iguais, demonstrando assim, que o DUV apresenta as mesmas funcionalidades do Modelo de Referência. Para fazer tal afirmação, faz-se necessário que os estímulos escolhidos abranjam

todas as funcionalidades especificadas. A cobertura funcional mede o progresso da simulação, a qual indica o nível de confiança do dispositivo simulado.

A **síntese** consiste na conversão do código RTL em uma *netlist*, que pode ser definida como uma representação do circuito, em termos de conexões entre portas lógicas.

A **simulação pós-síntese** assemelha-se com a etapa de verificação funcional, só que nela aspectos como atrasos de portas lógicas, por exemplo, são considerados.

A **fase de prototipação** é caracterizada pela implantação da *netlist* em algum dispositivo de *hardware*, tal como FPGA ou, até mesmo em silício. Ao final de todo processo, caso o protótipo gerado funcione de maneira correta, o IP *core* poderá ser integrado com outros IP para criação de um SoC (*System on a chip*).

Metodologias tradicionais propõem a implementação do DUV seguido da implementação do *testbench* (BERGERON, 2003). Basicamente, o objetivo da metodologia VeriSC (SILVA, 2007) é a criação de uma metodologia de verificação funcional, para verificar componentes digitais síncronos, por meio da comparação do DUV com seu Modelo de Referência, permitindo a criação do *testbench* antes da implementação do DUV, de forma a facilitar a verificação, dando ênfase a esta fase. Assim a metodologia se propõe a minimizar o tempo total de verificação e encontrar erros mais cedo, quando o DUV começa a ser implementado (SILVA, 2007, p. 8). No entanto, a metodologia VeriSC apresenta algumas desvantagens, as quais serão tratadas neste trabalho (*BVM – Brazil-IP Verification Methodology*).

## *1.2 O Brazil-IP*

Em 2001, surgiu um esforço colaborativo entre instituições brasileiras para criar um conjunto de centros de desenvolvimento de circuitos integrados capazes de produzir núcleos de propriedade intelectual (IP *cores*), chamado de *Brazil-IP*. No início, o grupo era formado por oito universidades, UNICAMP, USP, UFPE, UFCG, UFMG, UNB, UFRGS e PUCRS. Até 2006, três IP já tinham sido desenvolvidos no âmbito do *Brazil-IP*: O decodificador de vídeo MPEG-4, o decodificador de MP3 e o microcontrolador 8051. A UFCG (Universidade Federal de Campina Grande) ficou responsável pelo desenvolvimento de um decodificador de vídeo MPEG-4. Tal IP é, até hoje, o *chip* mais complexo desenvolvido por uma instituição brasileira. O artigo "*Silicon Validated IP Cores Designed by the Brazil-IP Network"* (MELCHER et al., 2006), foi ganhador do prêmio *"Best IP/SOC 2006 Design Award"*. Tal artigo apresentou o sucesso da metodologia de verificação funcional VeriSC e do processo de desenvolvimento de IP *cores* ipPROCESS na criação do decodificador MPEG-4, do microcontrolador 8051 e do decodificador de MP3, devido ao funcionamento correto desses IP em sua primeira prototipação em silício.

O sucesso da metodologia VeriSC na verificação funcional dos IP produzidos pelo Brazil-IP pode ser apresentado como principal motivação para sua reformulação sob a denominação BVM. BVM foi implementada na linguagem SystemVerilog (IEEE STANDARDS, 2005), utilizando conceitos e biblioteca de OVM (*Open Verification Methodology*) (OVM, 2010). BVM tem como objetivo aumentar a produtividade do engenheiro na realização no processo verificação funcional, o qual poderá ser agilizado com o uso da ferramenta eTBc (*Easy Testbech Creator*) (PESSOA, 2007). Essa ferramenta tem como objetivo automatizar o processo de construção de ambientes de verificação, ou seja, os *testbenches.* Tal automação reduz consideravelmente o tempo que o engenheiro levaria para construir todo este ambiente. Este, por sua vez, poderá ser descrito em qualquer linguagem, utilizando qualquer metodologia de verificação, desde que a ferramenta tenha sido adequada para dar suporte a esta linguagem, utilizando uma dada metodologia. De acordo com (PESSOA et al., 2007), o uso da ferramenta eTBc na construção semi-automática do ambiente de verificação utilizando a metodologia VeriSC, apresenta um ganho de 83% em relação a um processo manual. Porém, esta ferramenta só suportava a metodologia VeriSC, ou seja, ferramenta eTBc precisaria ser adaptada para suportar BVM.

## *1.3 Definição do problema*

A metodologia VeriSC e sua implementação apresentam alguns pontos que podem ser melhorados. Em relação à metodologia, não existe, em seus *testbench,* um elemento responsável por controlar e monitorar o protocolo de comunicação com o DUV. Também não existe um mecanismo de detecção de erros na fase de integração dos DUV. Em projetos que exigem uma integração de vários módulos, este mecanismo torna-se de grande importância, visto que se pode encontrar erros de comunicação entre eles de maneira mais eficiente

No período de desenvolvimento da metodologia VeriSC (2003 a 2007), optou-se por implementá-la em SystemC (IEEE STANDARDS, 2005 a), pois dentre as outras linguagens existentes na época, esta parecia ser a melhor candidata. SystemC era e ainda é comumente usada como linguagem de descrição no nível de sistema (ESL- *Electronic System Level*), em outras palavras, SystemC tem como foco um nível de abstração mais alto. Porém, em 2005, uma nova linguagem padronizada: SystemVerilog (IEEE STANDARDS, 2005 b). Diferentemente de SystemC, a linguagem SystemVerilog possui características voltadas para desenvolvimento e verificação funcional de projetos de *hardware*. A decisão de implementar VeriSC em SystemC teve grande impacto durante seu desenvolvimento. Um esforço extra foi necessário para viabilizar o uso dessa linguagem para verificação funcional. SystemC, por exemplo, não possuía uma biblioteca para realização da cobertura funcional. Surgiu, então, a necessidade do desenvolvimento de uma biblioteca para suprir tal carência: a biblioteca BVE (*Brazil-IP Verification Extensions*). De maneira oposta, SystemVerilog já possui características intrínsecas voltadas para

cobertura funcional.

A facilidade e os recursos que SystemVerilog oferece em relação a SystemC para verificação funcional tornam essa linguagem mais atraente para engenheiros de verificação. Com o amadurecimento dessa linguagem, grandes empresas na área de desenvolvimento de projetos de *hardware*, tais como Cadence (CADENCE DESIGN SYSTEMS, 2010), Mentor (MENTOR GRAPHICS, 2010a) e Synopsys (SYNOPSYS, 2009) lançaram metodologias e ferramentas de desenvolvimento com ênfase em SystemVerilog.

Visando uma melhoria da metodologia VeriSC e de sua implementação, decidiu-se reformulá-la, como também adaptá-la de SystemC para Systemverilog, utilizando conceitos de OVM (*Open Verification Methodology*). Outra contribuição deste trabalho é a adequação da ferramenta de apoio à verificação funcional eTBc (*Easy Testbench Creator*), para suportar BVM.

## *1.4 Objetivos*

O objetivo geral deste trabalho consiste em reformular a metodologia de verificação funcional VeriSC e adaptar sua implementação de SystemC para SystemVerilog, utilizando conceitos e a biblioteca de verificação funcional da metodologia OVM (*Open Verification Methodology*), como também em adequar e validar a ferramenta de apoio à verificação funcional eTBc (*Easy Testbench Creator*), para suportar tal abordagem. A implementação de VeriSC é baseada em SystemC e utiliza as bibliotecas SCV (*SystemC Verification Library*) e BVE-COVER (*Brazil-IP Verification Extension)* para realização da verificação funcional. Não existe um mapeamento direto das funcionalidades providas pela biblioteca de OVM para estas bibliotecas. Funcionalidades como: criação de estímulos, conexão dos elementos do *testbench* e cobertura funcional precisam ser mapeadas. Em relação aos objetivos específicos, estes podem ser resumidos em:

- Reformular, com base na metodologia OVM, a metodologia de verificação funcional VeriSC, viabilizando o controle e monitoramento do protocolo de comunicação do DUV com o *testbench;*

  - Modificar alguns subpassos da metodologia VeriSC com a inserção do elemento *Actor.*

- Criar um novo passo para garantir que, caso ocorram erros na integração do DUV, estes sejam detectados de maneira eficiente;

  - Criar um mecanismo que viabilize o monitoramento do protocolo de comunicação entre os elementos que compõe o DUV.

- Criar um protocolo de comunicação padrão para os elementos que se comunicam diretamente com o DUV.
- Desenvolver os *Template Patterns* que serão usados como base para a geração de código SystemVerilog pela ferramenta eTBc;
- Testar os *Template Patterns* criados e a ferramenta eTBc;
- Reportar erros e necessidades de adequação do núcleo da ferramenta eTBc, para que esta possa dar suporte à linguagem SystemVerilog e caraterísticas da metodologia BVM;
- Automatizar, em alguns aspectos, o processo de verificação funcional utilizando a ferramenta de simulação da Cadence, a chamada IUS (*Incisive Unified Simulator*);

    - Executar a simulação até que 100% de cobertura seja atingida (parada automática da simulação).

- Realizar um estudo comparativo entre as duas abordagens: VeriSC e BVM, apresentando os aspectos positivos e negativos de cada uma delas;
- Apresentar resultados do uso de BVM na verificação de IP *cores* produzidos pelo projeto *Brazil-IP.*

## *1.5 Organização*

O restante deste documento encontra-se organizado da seguinte maneira:

- **Capítulo 2**: Neste capítulo, são apresentados conceitos fundamentais para o entendimento do trabalho realizado;
- **Capítulo 3:** A metodologia BVM é apresentada de forma comparativa à metodologia VeriSC. Neste capítulo, também são descritos alguns conceitos da metodologia OVM, assim como é introduzida a ferramenta eTBc na geração dos *testbenches* da metodologia BVM;
- **Capítulo 4:** Apresentação dos resultados;
- **Capítulo 5:** São apresentadas as considerações finais e sugestões para trabalhos futuros.

# Capítulo 2

# 2 Conceitos fundamentais

## *2.1 Verificação*

Verificar o funcionamento de um circuíto digital não é uma tarefa trivial. Como já mencionado anteriormente, a fase de verificação consome cerca de 70% de todos os recursos do projeto. Existem várias maneiras de realizá-la em um projeto de implementação *hardware.* O dispositivo a ser verificado pode ser implementado em diferentes níveis de abstração. Porém, neste trabalho adotou-se sua implementação no nível de RTL. Os tipos de verificação mais conhecidos são: *verificação formal ou estática*, *verificação funcional ou dinâmica* e *verificação híbrida.*

### 2.1.1 Verificação formal ou estática

De acordo com Bergeron et al. (2005), verificação formal, também conhecida como verificação estática, pode ser definida como uma comparação matemática de uma implementação contra uma especificação ou requisito para determinar se a implementação viola esta especificação ou requisito. A aplicação da verificação formal recai em uma de duas categorias: (i) por equivalência (*equivalence checking*); e (ii) por propriedade (*property checking*). Em resumo, pode-se dizer que a verificação formal por equivalência consiste em provar matematicamente, por meio de ferramentas, que a transformação de uma representação do circuito digital em outra preserva sua funcionalidade e comportamento original. Por exemplo, provar que uma *netlist* é logicamente equivalente ao seu código RTL, garantindo que a ferramenta de síntese não inseriu erros na geração dessa *netlist*. Um verificação formal por equivalência pode ser realizada entre quaisquer duas seguintes representações: RTL <=> netlist, netlist <=> netlist ou RTL <=> RTL. Na Figura 2, é apresentado um modelo que reflete este tipo de verificação.

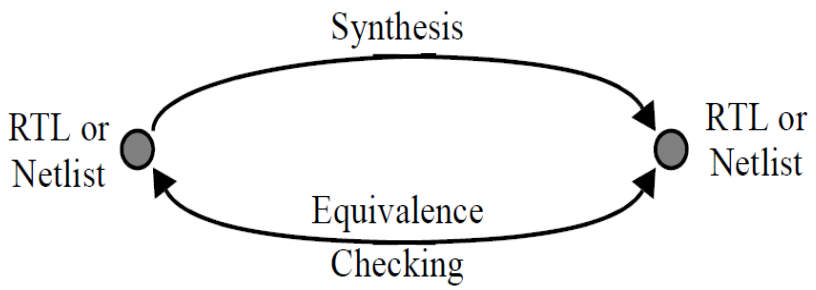


**Figura 2 – Modelo de reconvergência para *equivalence checking* (BERGERON et al, 2005).**

Na verificação formal por propriedade, *assertions*[1] ou características de um RTL são formalmente provadas ou refutadas. Em outras palavras, ela prova matematicamente se o comportamento do RTL está em conformidade com sua especificação, expressa como um conjunto de descrições de propriedades. Para a realização deste tipo de verificação, o engenheiro geralmente descreve tais propriedades na forma de *assertions*. Bergeron (2006) afirma que o maior obstáculo deste tipo de verificação é identificar, a partir da especificação do projeto, quais *assertions* devem ser provadas. Na Figura 3, é apresentado um modelo de reconvergência para este tipo de verificação.

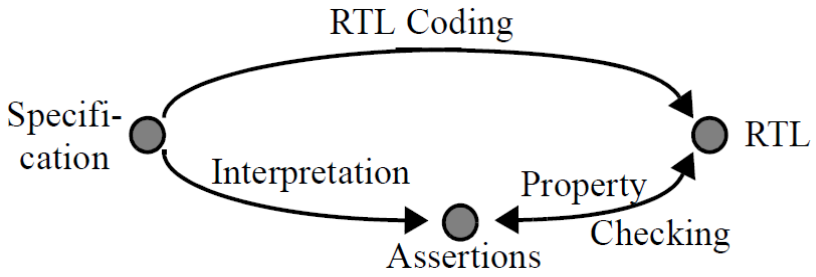


**Figura 3 – Modelo de reconvergência para *property checking* (BERGERON et al, 2005).**

### 2.1.2 Verificação dinâmica ou funcional

Verificação funcional, também conhecida por verificação dinâmica, pode ser definida formalmente como um processo usado para demonstrar que o objetivo do projeto é preservado em sua implementação (BERGERON, 2006). Realizada a partir de simulação, a verificação funcional não pode provar a ausência de erros na implementação do DUV, mas apenas a existência deles. Na Figura 4, é apresentada a relação entre especificação e RTL.

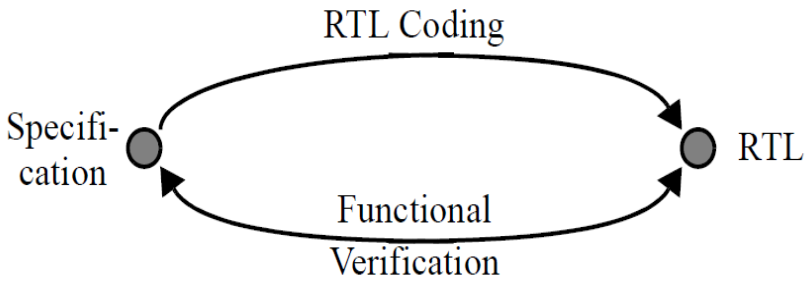


**Figura 4 – Modelo de reconvergência para verificação dinâmica (BERGERON et al, 2005).**

1 De acordo com Bergeron (2006), na perspectiva de verificação, uma a*ssertion* é uma declaração de um comportamento esperado.

Durante a simulação, estímulos são inseridos na entrada do DUV e sua saída é coletada. Os valores de saída são comparados com os valores esperados. Tais estímulos tem como função exercitar as funcionalidades desejadas do DUV. Por este motivo, devem ser escolhidos visando atingir os critérios de cobertura previamente definidos. Caso todas funcionalidades sejam testadas durante este processo, a simulação é interrompida e diz-se que foi atingida uma cobertura de 100%. Em outras palavras, o término da simulação é alcançado quando todos os critérios de cobertura são satisfeitos.

### 2.1.3 Verificação híbrida ou semi-formal

Verificação híbrida ou semi-formal combina verificação dinâmica com verificação formal por propriedade (*property checking*). Em outras palavras, combina simulação com métodos formais de verificação. Este tipo de verificação é usada na tentativa de superar a inviabilidade de verificação de projetos complexos utilizando verificação formal, como também de superar as limitações intrínsecas de verificação baseada em simulação. Este tipo de verificação pode provar que o RTL não viola algumas propriedades dados estímulos de entrada legais[2].

## *2.2 Plano de verificação*

O plano de verificação especifica requisitos de verificação a nível de sistema. Fortemente ligado à especificação, tem como objetivo garantir que a implementação do projeto será testada de forma abrangente, fazendo com que esta atenda todas as funcionalidades especificadas dentro do prazo do projeto. Bergeron et al. (2005) diz que o plano deve ser iniciado o quanto antes. Deve ser idealmente concluído antes da implementação do RTL e antes da implementação do *testbench*, aumentando assim a probabilidade de um RTL livre de falhas.

Este documento descreve quais as funcionalidades que devem ser exercitadas e quais técnicas devem ser usadas para tanto. Em outras palavras, define-se *o que* deve ser verificado e *como* será verificado. Além disso documenta que funcionalidades são prioridades e quais são opcionais. Bergeron e outros autores (BERGERON et al, 2005) afirmam que o plano de verificação detalha, tipicamente, conjuntos de testes individuais que devem ser escritos para verificar um RTL particular. Silva (2007) diz que este plano precisa lidar com várias questões, desde objetivos de alto nível até a identificação e documentação de variáveis. Além disto, de acordo com Bergeron (2006), 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. Da mesma forma, Spear (2006)

2 Estímulos de entrada legais podem ser entendidos como estímulos de entrada válidos para testar uma dada funcionalidade do DUV ou parte dele.

afirma que, com o plano de verificação em mãos, o engenheiro de verificação estará apto para escrever os vetores de estímulos que irão exercitar o DUV. Em resumo, pode-se dizer que o plano de verificação guia a construção do *testbench* para a realização da verificação funcional em um projeto de *hardware.*

## *2.3 Testbench*

Para a realização da simulação, faz-se necessária a criação de um ambiente de verificação, conhecido por *testbench*. Este ambiente é responsável por testar o DUV, inserindo estímulos e comparando respostas entre o DUV e um modelo ideal, denominado Modelo de Referência ou *Golden Model*. Conforme mencionado anteriormente, o DUV visa alcançar a especificação do *hardware* ou parte dela, descrita em código RTL. O Modelo de Referência é uma implementação da especificação, o que significa que conceitualmente é livre de erros. Os estímulos são dados de entrada para do DUV e para o Modelo de Referência. Para a validação do DUV, esses estímulos são inseridos em sua entrada e na entrada do Modelo de Referência. As respostas de ambos serão comparadas, validando ou não no DUV. Tais estímulos devem ser escolhidos de forma cuidadosa, para que executem todas as funcionalidades especificadas. A estrutura genérica de um *testbench* é apresentada na Figura 5, na forma de U invertido.

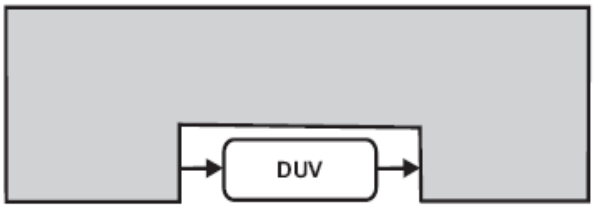


**Figura 5 – Estrutura genérica de um *testbench* (SILVA, 2007).**

Geralmente, entre os estímulos inseridos, aqueles de origem pseudo-aleatória são comumente usados. Para saber quando parar a simulação, deve-se verificar se todas as funcionalidades especificadas foram exercitadas. É a cobertura funcional quem realiza este papel. De acordo com Silva (2007), cobertura funcional pode ser definida como uma técnica usada para medir o progresso da simulação e reportar quais funcionalidades deixaram de ser testadas. Quando todas as funcionalidades especificadas foram exercidas, diz-se que foi atingida uma cobertura de 100%.

Na metodologia VeriSC, o *testbench* é implementado antes do DUV. Para tal, esta metodologia utiliza elementos do próprio *testbench* para simular o DUV; o que significa que não existe a necessidade de código adicional para a construção do ambiente de verificação. A

estrutura de um *testbench,* utilizando a metodologia VeriSC, é apresentada na Figura 6.

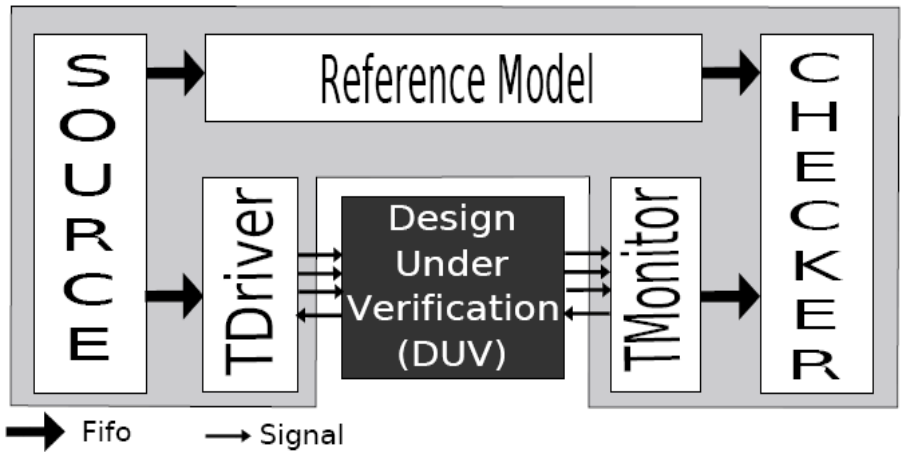


**Figura 6 – Diagrama do *testbench* da metodologia VeriSC (SILVA, 2007).**

O *testbench* é ligado ao DUV por interfaces formadas por sinais, representados na figura por setas finas. As setas mais largas representam a conexão dos elementos do *testbench* através de FIFO (*First IN First Out*). O *testbench* nesta metodologia é composto de cinco elementos básicos: *Source*, Modelo de referência, *Checker*, *TDriver(s)* e *Tmonitor(es),* de acordo com Silva (SILVA, 2007, p. 34), os quais são descritos a seguir.

1. O *testbench* é ligado por FIFO. As FIFO 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 FIFO devem ser retirados individualmente e comparados, de forma que sempre estarão na ordem correta de comparação.

2. 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)[3]. Essa comunicação se faz com a ajuda dos blocos *TDriver(s)* e *TMonitor(es)* que traduzem sinais para transações[4] e vice-versa.

3. O *Source* é responsável por criar estímulos para a simulação. Esses estímulos devem ser cuidadosamente escolhidos para satisfazer os critérios de cobertura especificados. Todos

3 Abordagem de alto nível para modelagem de comunicação entre blocos funcionais ou elementos do *testbench* de um sistema digital.

4 Silva (2007) define transação como uma representação básica para a troca de informações entre dois blocos funcionais, ou seja, 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.

os estímulos criados pelo *Source* são transações.

4. 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, a partir de sinais.

5. 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,* convertendo todos os sinais de saída do DUV para transações, as quais são repassadas para o *Checker*, via FIFO. Além disto, o *TMonitor* também executa um protocolo de comunicação com o DUV, a partir de sinais.

6. O módulo *Checker* é o responsável por comparar as respostas resultantes do *TMonitor* e Modelo de Referência, determinando sua equivalência. O Modelo de Referência contém a implementação ideal (especificada) do sistema. Por isso, ao receber estímulos, ele deve produzir respostas corretas.

Bergeron (2003) afirma que, para um bom *testbench,* são desejáveis as seguintes características:

- Ser dirigido por cobertura, ou seja, os estímulos escolhidos e o tempo de simulação dependem dos critérios de cobertura pré-estabelecidos;
- Possuir randomicidade direcionada. A randomicidade dos estímulos gerados é direcionada para atingir de valores pré-determinados, visando testar as funcionalidades do projeto abrangendo uma cobertura especificada;
- Ser autoverificável. Os resultados esperados são automaticamente comparados com os resultados coletados na saída do DUV;
- Ser baseado em transações. A comunicação entre blocos funcionais ou elementos do *testbench* deve ser implementada em um nível de abstração mais alto, visando uma maior eficiência na realização da simulação;
- Não possuir entradas e saídas;
- Imprimir mensagens e criar histórico de comportamentos inesperados do DUV;
- Ser escrito em alguma linguagem voltada para verificação funcional.

É importante que erros propositais sejam inseridos no conjunto de testes. De acordo como

Bergeron (2006), nunca se deve acreditar que um *testbench* não produz mensagens de erro. A ausência de detecção desses erros indicará que o *testbench* não está realizando seu papel de maneira correta.

## *2.4* Modelagem em nível de Transação *(Transaction Level Modelling)*

Vale lembrar que a comunicação entre os elementos do *testbench* ocorre mediante transações. Em sistemas digitais, uma transação pode ser definida como “uma transferência de dados e controle entre dois subsistemas” (SUTHERLAND, DAVIDMANN, FLAKE, 2006, p. 330). Neste nível, detalhes de protocolo de comunicação no nível de sinais são abstraídos, reduzindo o esforço na modelagem do sistema como também aumentando a velocidade da simulação.

## *2.5 Geração de estímulos*

De acordo com Spear (2006), estímulos randômicos são cruciais para exercitar projetos complexos. Porém não é desejado que estes estímulos sejam totalmente aleatórios, ou seja, é interessante que estes tenham restrições, como por exemplo, quantidade de *bits,* faixa de valores, etc. Tais estímulos com formatos predefinidos são chamados de estímulos aleatórios com restrições (*Constrained-Random Stimulus)* ou simplesmente estímulos de randomicidade direcionada.

Em projetos pequenos, compostos por dois ou três módulos, estímulos podem ser previamente definidos para exercitar o DUV, sendo escritos manualmente, ou até mesmo, retirados de ambientes de funcionamento reais. Esses estímulos são chamados de estímulos direcionados. *Testbenches* que utilizam esses tipos de estímulos são chamados de *testbenches* direcionados. O problema deste tipo de abordagem é a inexistência de escalabilidade. De acordo com Spear (2006), em testes direcionados, quando a complexidade do projeto dobra, este levará o dobro do tempo ou o dobro de pessoas trabalhando para atingir 100% de cobertura. Bergeron (2006) define que, com uma equipe de 10 engenheiros de verificação, utilizando testes direcionados, um projeto que tenha acima de mil conjuntos de testes identificados pode levar mais de um ano para ser completado. Assim, alguma forma de automação é necessária para realizar essa tarefa dentro do prazo determinado. Tal automação pode ser alcançada com verificação dirigida por cobertura e randomicidade direcionada, criando assim *testbenches* randômicos para exercitar funcionalidades específicas.

Enquanto *testbenches* randômicos podem encontrar falhas nunca imaginadas, *testbenches* direcionados encontram apenas falhas previstas. Spear (2006) afirma que testes randômicos frequentemente cobrem um maior espaço do que testes direcionados. Desta maneira

é fácil perceber que utilizar verificação dirigida por cobertura e gerar de estímulos mediante randomicidade direcionada parece ser uma boa opção para projetos que envolvem um maior nível de complexidade.

Os estímulos gerados para teste do DUV devem ter idealmente como objetivo testar todas as suas funcionalidades. Por isto devem cobrir os possíveis valores válidos para o DUV. Bergeron (2006) sugere que, para testar tais funcionalidades, o DUV deve ser simulado com os estímulos citados na especificação, situações críticas, os randômicos e casos de testes reais. Um resumo de alguns trabalhos que visam à criação de testes para estimular o DUV durante a verificação funcional pode ser encontrado no trabalho de Silva (2007).

## *2.6 Cobertura*

Quando se usam estímulos randômicos, faz-se necessária a cobertura funcional para medir o progresso da verificação. A cobertura funcional é uma medida de quais funcionalidades do projeto foram exercitadas durante os testes (SPEAR, 2006, p. 241). As funcionalidades que não são devidamente testadas durante a simulação são denominadas buracos de cobertura. Silva (2007) aponta três causas para que ocorram:

- O simulador precisa de mais tempo para exercitar as funcionalidades desejadas;
- Não foram gerados estímulos suficientes para exercitar todas as funcionalidades;
- Existem erros no dispositivo que não permitem que as funcionalidades sejam testadas.

Caso o buraco de cobertura não seja identificado, a verificação do dispositivo será realizada de forma incompleta, podendo deixar passar erros que causarão grandes prejuízos para o projeto. O primeiro passo para implementar a cobertura funcional é criar um modelo de cobertura. Rodrigues (RODRIGUES et al., 2009) define que o modelo de cobertura é um espaço da cobertura funcional que captura os requisitos de todo comportamento do dispositivo. Ele consiste em isolar e definir os valores de atributos que devem ser analisados na cobertura funcional. Tais atributos e valores são identificados pelo engenheiro de verificação, de acordo com o plano de verificação. Formalmente, pode ser definido como uma lista de atributos e um conjunto de valores desses atributos que se deseja verificar. Além disto, o modelo de cobertura especifica condições ilegais do sistema como também condições que devem ser ignoradas na medição da cobertura. A análise da cobertura é o processo de identificação de áreas não cobertas no modelo de cobertura, identificando, deste modo, o próximo requisito da verificação funcional que será alvo.

A cobertura funcional, quando usada de maneira adequada, torna-se uma especificação

formal do plano de verificação, visto que todas as funcionalidades definidas em tal plano serão exercitadas durante a simulação.

É importante dizer que se o plano de verificação for mal elaborado, a cobertura funcional realizada não verificará todas as funcionalidades desejadas. Por exemplo se o plano de verificação abrange 80% das funcionalidades que deveriam ser testadas, então, mesmo que a cobertura funcional seja realizada de maneira adequada, chegando aos 100% de cobertura, ela estará testando apenas 80% das funcionalidades que deveriam ser testadas. Por este motivo, é importante que o plano de verificação seja revisto e modificado quando necessário, já que ele reflete o estado presente do projeto.

## *2.7 SystemVerilog*

Nos meados de 1990, Verilog tornou-se a linguagem de descrição de *hardware* mais utilizada para simulação e síntese (SPEARS, 2006). Porém, em suas duas versões padronizadas pelo IEEE, esta linguagem oferecia recursos para criar apenas testes simples, não atendendo à parte de verificação para circuitos mais complexos. Assim, surgiram algumas linguagens de verificação de *hardware* (HVL – *Hardware Verification Language*) comerciais como OpenVera e “e”. Porém, algumas empresas preferiam criar ferramentas personalizadas do que utilizar uma destas linguagens (SPEARS, 2006).

Diante dos fatos, um consórcio de companhias de EDA (*Electronic Design Automation*) e usuários uniram-se, com intenção de impulsionar o desenvolvimento e o uso de padrões exigidos pela indústria de semicondutores. Tal união deu origem à organização Accellera (ACCELLERA, 2010), a qual decidiu criar uma nova geração da linguagem Verilog. A doação da linguagem OpenVera, por parte da Synopsys, formou a base para as características de verificação de *hardware* dessa nova linguagem, a qual foi denominada SystemVerilog.

Adotada como padrão IEEE em 2005, SystemVerilog foi a primeira linguagem padrão da indústria a cobrir RTL, *assertions, transaction-level modeling* e *coverage-driven constrained random verification*. Em outras palavras, é uma linguagem unificada que abrange modelagem em nível de sistema, RTL e verificação. Spears (2006) afirma que um dos maiores benefícios de SystemVerilog é permitir a criação de ambientes de verificação confiáveis e repetíveis em uma sintaxe consistente que podem ser usados em vários projetos.

Outra linguagem, também adotada como padrão IEEE em 2005, foi SystemC. Esta linguagem tem sido amplamente usada para escrever representações TLM e modelos de referência. SystemC é normalmente aplicada para modelagem no nível de sistema (ESL-*Electronic System Level*).

Assim como SystemC, a linguagem SystemVerilog incorpora programação orientada a objetos, com algumas diferenças, citadas a seguir:

- SystemC é uma linguagem mais adequada para simulações de *hardware* em alto nível. Diferentemente de SystemVerilog, qualquer projeto de *hardware* baseado em código SystemC será, provavelmente, ineficiente e volumoso. De acordo com Goering (2010), um grupo de projetistas da Motorola desenvolveu um projeto de *hardware* em SystemC, que Segundo o líder do grupo, o código era muito mais difícil de ler do que Verilog, mais “lento e desajeitado” para escrever, difícil de depurar e ineficiente para reuso. Tal fato corrobora o que já foi anteriormente dito, visto que SystemVerilog engloba a linguagem Verilog;

- SystemC estende a capacidade de C ++. Para viabilizar a modelagem de *hardware*, SystemC adiciona conceitos de concorrência, eventos e tipos de dados apropriados para *hardware*, tais como portas, sinais e módulos Desta maneira, para criar modelos (i.e representações de um circuito digital ou parte dele) em SystemC, é necessário que o engenheiro aprenda C++ e tais conceitos. De maneira oposta, SystemVerilog já fornece recursos nativos para *modelagem de hardware*.

- SystemC não possui recursos em sua linguagem para modelar cobertura. De modo contrário, SystemVerilog possui características nativas para medição da cobertura funcional.

- Tanto em SystemC quanto em SystemVerilog é possível criar modelos em nível de transação com eficiência comparável.

- Os simuladores SystemVerilog necessitam de licenças pagas, enquanto os simuladores SystemC não necessitam, caso todos os modelos estejam disponíveis em SystemC.

- SytemVerilog incorpora programação orientada a objetos, porém SystemC tem um suporte mais rico para este tipo de abordagem, já que suporta herança múltipla e sobrecarga de funções.

- SystemVerilog destaca-se na fusão de diferentes tecnologias de verificação funcional: *Assertions*, randomicidade direcionada e cobertura funcional.

Uma característica importante da linguagem SystemVerilog é que seus ambientes permitem chamadas de funções em C/C++ em através de DPI (*Direct Programming Interface*). Dessa maneira modelos de referências escritos nessas linguagens são interoperáveis com ambientes escritos em SystemVerilog.

Em resumo, pode-se dizer grosso modo que a linguagem SystemVerilog é adequada para

criar *testbenches* e descrever circuitos a nível RTL, enquanto SystemC é adequada para criar modelos de referência, já que estes podem ser uma representação do circuito em alto nível.

Os fatos anteriormente apresentados, evidenciam que SystemVerilog é uma linguagem mais adequada para realização da verificação funcional. Estes e outros motivos, abordados nos próximos capítulos, encorajaram a reformulação da metodologia VeriSC e adaptação de sua implementação de SystemC para SystemVerilog, utilizando conceitos e biblioteca de OVM (*Open Verification Methodology*), tornando esta metodologia reformulada e sua implementação mais poderosa.

# Capítulo 3

# 3 Análise comparativa entre BVM e VeriSC

## *3.1 VeriSC*

Em 2007, Silva (2007) propôs uma metodologia de verificação funcional para circuítos digitais síncronos, denominada VeriSC. Voltada para verificação de circuitos que possuem um único sinal de *clock*, esta metodologia tem como objetivo a implementação do *testbench* antes do DUV. Um importante aspecto de VeriSC é que faz reuso[5] dos próprios elementos do *testbench* para criar *testbenches hierárquicos* e emular o DUV, pretendendo desta maneira a criação de elementos do *testbench* livres de erros e a criação do ambiente de teste antes da implementação do DUV sem que para isto seja necessário escrever código adicional.

Em VeriSC, o processo de construção do *testbench* acontece de forma incremental. Vários passos são descritos para o processo de implementação deste ambiente de verificação. A substituição e reuso dos próprios elementos do *testbench* são fundamentais neste processo. Ao longo dos passos, vários *testebenches* são descritos visando ao final testar o DUV em RTL.

Como criar um ambiente de verificação para um determinado circuito ou parte dele sem ainda ter seu RTL em mãos? Foi criada uma tripla com *TDriver*, Modelo de Referência e *TMonitor* para emular um DUV em RTL. A união destes elementos faz com que esta tripla tenha entradas e saídas em níveis de sinais, justamente como um DUV real.

VeriSC foi utilizada com sucesso na verificação funcional dos IP *cores* produzidos pelo Brazil-IP até o ano de 2008. Porém, esta metodologia apresenta algumas desvantagens:

- VeriSC não possui um artifício responsável por controlar e monitorar o protocolo de comunicação do DUV com o *testbench*. Além disto, ela carece de um mecanismo para detecção de erros na fase de integração dos DUV.
- A implementação de VeriSC não possui um protocolo de comunicação padrão para a comunicação dos elementos *TDriver* e *TMonitor* com o DUV. Este protocolo é implementado pelo engenheiro de verificação que utilize esta metodologia.

Visando a criação dos aspectos não contemplados em VeriSC, um estudo de outras metodologias foi realizado. Este estudo teve como objetivo encontrar uma maneira que viabilizasse a reformulação da metodologia VeriSC e sua implementação de SystemC para SystemVerilog. As metodologias estudadas são apresentadas na próxima seção.

5 Ao longo deste trabalho, entenda-se “reuso” como sendo o reaproveitamento interno de elementos do *testbench* utilizados na verificação de um projeto específico em RTL.

## *3.2 Outras metodologias*

VeriSC utilizava a biblioteca SCV (*SystemC Verification Library*), porém essa biblioteca não apresenta compatibilidade com a linguagem SystemVerilog, a qual será adotada pela metodologia BVM. Isso implica no uso de alguma outra biblioteca, que seja compatível com SystemVerilog. Diversas metodologias voltadas para a verificação funcional estão disponíveis para serem usadas. As mais conhecidas são VMM, AVM, IPCM, UNISIM, OVM e RVM.

A metodologia VMM (*Verification Methodology Manual)* (BERGERON et al., 2005) é documentada por um livro de autoria conjunta da ARM e Synopsys (ARM AND SYNOPSYS, 2010). De acordo com Rocha (2008), a natureza orientada a objetos da metodologia VMM é apontada como o maior obstáculo para os engenheiros que desejam adotá-la, visto que classes, encapsulamento, herança, extensões entre outros aspectos da programação orientada a objetos, tornam o ambiente de verificação diferente do utilizado tradicionalmente para a construção dos *testbenches*. Esta metodologia disponibiliza quatro bibliotecas para auxílio à construção desses ambientes de verificação:

- A biblioteca VMM *Standard* que possui um conjunto de classes e *testbenches* em SystemVerilog;

- A biblioteca VMM Checker, que possui um conjunto de *assertions*, que são descritores do projeto e do comportamento do ambiente, e *checkers* em SystemVerilog;

- A biblioteca XVC Standard, que possui um conjunto de classes básicas em SystemVerilog;

- O *Framework* de Teste de Software, que possui uma biblioteca C para verificação de software.

Tal metodologia, foi baseada na metodologia RVM (*Reference Verification Methodology*), também criada pela Synopsys e que pode ser vista em detalhe a partir de Bergeron et al. (2005).

A metodologia AVM (*Advanced Verification Methodology*) foi a primeira metodologia de verificação funcional de código-aberto para a indústria no nível RTL (MENTOR GRAPHICS, 2010c). Desenvolvida pela Mentor Graphics (MENTOR GRAPHICS, 2010a), AVM fornece bibliotecas de classes base e módulos em forma de código-fonte aberto e utiliza interfaces TLM como o mecanismo de comunicação entre os componentes de verificação. Esta metodologia é documentada no livro AVM *Cookbook* (GLASSER et al., 2007). As bibliotecas da metodologia AVM consistem em uma coleção de classes básicas que facilitam a construção de *testbenches*, incluindo classes para a construção de componentes, portas e a conexão de componentes com interfaces no nível de transação. O código fonte das bibliotecas da metodologia AVM, com

exemplos de implementação em SystemC e SystemVerilog, está disponível para o usuário.

A metodologia RVM (*Reference Verification Methodology*) é uma metodologia de verificação funcional, baseada em OpenVera (OPENVERA, 2010) e criada pela Synopsys (ARM AND SYNOPSYS, 2010) em 2003. Ela define um conjunto de métricas e métodos para realização da verificação funcional de circuitos digitais complexos. VMM é uma implementação dessa metodologia baseada em SystemVerilog.

A metodologia IPCM/URM (*Incisive Plan-to-Closure Universal Reuse Methodology*) promete reduzir os riscos na verificação dos chips e SoC, por meio do fornecimento de um sistema com melhores práticas, princípios e procedimentos que visam aumentar a produtividade e previsibilidade do projeto, afim de assegurar a qualidade do sistema (ROCHA, 2008). Desenvolvida pela Cadence, IPCM/URM suporta a criação de um ambiente de verificação, o reuso, sistemas especialistas escritos em SystemC, modelos no nível de transação, co-verificação de hardware e de software, além de verificação baseada em transação e emulação de circuitos.

UNISIM (*UNIted SIMulation environment*) (UNISIM, 2010) é um ambiente de simulação modular, implementado em SystemC, com foco no reuso de lógica da controle, a qual corresponde a uma grande quantidade de código de um simulador. Este ambiente proporciona um mapeamento intuitivo entre diagramas de blocos do hardware e o módulo do simulador, em que os blocos do hardware correspondem a módulos da simulação. Além disso, ele suporta modelagem em um nível abstrato utilizando os modelos TLM e CLM (*Cycle-level Modelling*). Sabendo-se que os simuladores TLM são menos precisos, mas muito mais rápidos do que os simuladores CLM, a metodologia UNISIM utiliza simuladores híbridos CLM/TLM, possuindo um sistema de simuladores funcionais que podem ser acoplados tanto em simuladores CLM quanto em simuladores TLM (AUGUST et al., 2007, p. 1).

A metodologia OVM (*Open Verification Methodology*) (OVM, 2010) desenvolvida em conjunto pela Cadence (CADENCE DESIGN SYSTEMS, 2010) e Mentor (MENTOR GRAPHICS, 2010a), utiliza a linguagem SystemVerilog e faz reuso de metodologias que prometem possuir as melhores práticas no desenvolvimento dos componentes UVC (*Universal Verification Component*) focado na criação de SoC. Essa metodologia é documentada a partir dos documentos *Open Verification Methodology class reference* (CADENCE; MENTOR, 2009a) e *Open Verification Methodology user guide* (CADENCE; MENTOR, 2009b). OVM é uma metodologia de verificação funcional aberta, e interoperável com múltiplas linguagens e simuladores. O ambiente de verificação da metodologia OVM é composto por componentes UVC reusáveis, denominados de OVC (OVM *verification components*). Cada UVC segue uma arquitetura, que consiste de um conjunto de elementos para controlar, estimular e recolher informações de cobertura para um específico protocolo ou projeto (CADENCE; MENTOR, 2009b, p. 71). Ela oferece uma biblioteca

de verificação funcional em SystemVerilog, e também disponibiliza seu código fonte. A metodologia OVM foi criada a partir de metodologias já difundidas: AVM da Mentor e URM da Cadence.

Dentre as metodologias apresentadas, duas delas poderiam ser utilizadas no desenvolvimento de BVM: a metodologia VMM, visto que é uma metodologia já consolidada e oferece suporte à SystemVerilog, e a metodologia OVM, já que também trabalha com SystemVerilog e é resultante da união das melhores características das metodologias AVM e URM. Entretanto, o trabalho faz uso da biblioteca OVM e de boa parte dos seus conceitos, visto que, em relação à VMM, esta oferece uma documentação concisa e de fácil entendimento, além de prover uma maneira eficiente de conectar os elementos que compõe o *testbench*. No Quadro 1, são apresentadas características de VMM e OVM.

**Quadro 1 – Comparação entre VMM e OVM.**

| | **VMM** | **OVM** |
|---|---|---|
| Código a aberto | Atualmente sim (Desde 2008) | Sim |
| Oferece biblioteca para suporte à verificação funcional | Sim | Sim |
| Documentação concisa e de fácil entendimento | Não | Sim |
| Implementada em SystemVerilog | Sim | Sim |
| Baseada em metodologias consolidadas | Sim (RVM) | Sim (AVM e URM) |
| Cria ambientes de verificação reusáveis | Sim | Sim |
| Facilidade de conexão dos elementos que compõe o *testbench* | Não | Sim |
| Modo como é apresentado o ambiente de verificação | Camadas | OVC (OVM *verification component*)[6] |
| Obriga o uso de um Modelo de Referência | Não | Não |
| Fundamentada na criação do *testbench* antes da implementação do DUV | Não | Não |

BVM foi criada com objetivo de suprir as carências existentes em VeriSC, porém mantendo suas vantagens, a partir de sua reformulação e do uso da metodologia OVM. O livro OVM

6 OVC (*OVM verification component*) ambiente de verificação configurável, encapsulado e pronto ara uso, para um protocolo de interface, sub-módulo, ou um sistema completo (CADENCE; MENTOR, 2009b, p. 10).

*cookbook* (GLASSER, 2009) apresenta de maneira simples e objetiva como implementar *testebenches*, partindo de exemplos básicos, até chegar em exemplos sofisticados. Na próxima seção, são apresentados os passos necessários para a construção do ambiente de verificação de BVM, os quais são derivados da metodologia VeriSC.

## *3.3 BVM*

Visando a melhoria da metodologia VeriSC, consolidou-se, no âmbito deste trabalho, sua reformulação e implementação em uma linguagem mais voltada para a verificação funcional. Como já exposto, SystemVerilog foi a linguagem escolhida para tal tarefa. BVM, da mesma forma que VeriSC, tem a construção do *testbench* feita de forma incremental. Uma das diferenças entre os passos de tais metodologias está na inserção de um elemento responsável por controlar e monitorar os sinais do protocolo de comunicação entre o DUV e o *testbench*: o *Actor*. Outra diferença está na simplificação da comunicação dos elementos do *testbench* por meio da criação de uma FIFO de duas saídas. O uso deste tipo de FIFO e a criação do *Actor* viabilizou um mecanismo de detecção de erros na fase de integração dos DUV. Em projetos que exigem uma integração de vários módulos, esse mecanismo torna-se de grande importância, visto que se pode encontrar erros de comunicação entre eles de maneira mais eficiente. Dado este fato, um novo passo foi criado na metodologia BVM visando reduzir o tempo gasto pelos engenheiros de verificação na detecção destes erros. A seguir, serão descritos todos os passos necessários para a construção do *testbench* final de BVM. Vale relembrar que, assim como VeriSC, a construção do *testbench* em BVM é realizada antes da criação do DUV.

### *3.3.1 Testbench conception*

O primeiro passo é composto por três subpassos, tendo como finalidade a geração do *testbench* para o DUV completo. Os subpassos que o compõe são: *Single refmod, Double refmod e DUV emulation*.

#### *3.3.1.1 Single refmod*

Tem como objetivo testar a capacidade do Modelo de Referência interagir com o *testbench*. Um *Source* gera estímulos para o Modelo de Referência. O Modelo de Referência por sua vez, a partir de funções quaisquer aplicadas aos estímulos recebidos, gera transações que serão recebidas pelo *Sink.* Todo este processo de teste é realizado no nível de transação. Na Figura 7 é mostrada uma representação desse subpasso.

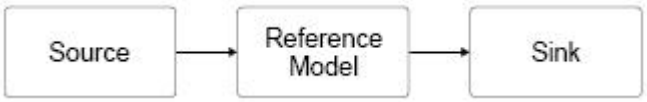


**Figura 7 – Representação do subpasso *Single refmod.***

### *3.3.1.2 Double refmod*

Neste subpasso, duas instâncias do Modelo de Referência serão estimuladas ao mesmo tempo ecom as mesmas entradas; Suas saídas serão verificadas, a fim de testar se são iguais. O *Sink* não será mais utilizado. Um novo elemento será inserido neste contexto: o *Checker*. Ele é utilizado para verificar as saídas dos dois Modelos de referência (Figura 8). O principal objetivo deste subpasso consiste em testar se o *Source* e o *Checker* estão realizando seu papel adequadamente. O *Source* deste subpasso é reusado do subpasso anterior. Em VeriSC, o subpasso *Single refmod* utiliza um *Pré-source* em vez de um *Source*. O *Pré-source* possuí apenas uma FIFO de saída única, enquanto o *Source* possuí duas. Isto significa que, em VeriSC, não se pode fazer reúso do *Pré-source* no subpasso *Double refmod*. Em BVM, isto é possível graças à criação da FIFO de duas saídas. Idealizada juntamente com professor Elmar Uwe Kurt Melcher*, a FIFO de duas saídas simplifica a comunicação entre os elementos do *testbench,* além de ser utilizada no mecanismo de detecção de erros na etapa de integração dos DUV. No subpasso *Double refmod* de VeriSC, o *Source* possui duas FIFO de saída única que repassam os valores gerados para os dois Modelos de referência. De maneira simplificada, em BVM, o *Source* deste subpasso gera estímulos que são distribuídos por uma única FIFO de duas saídas.

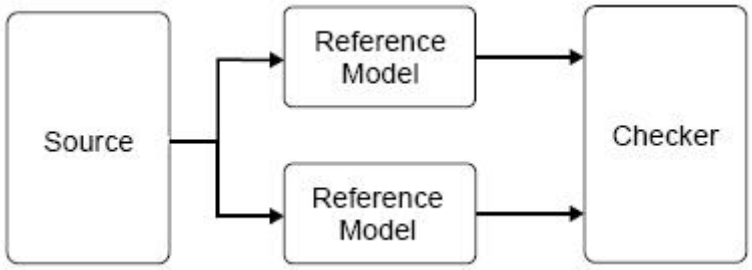


**Figura 8 – Representação do subpasso *Double refmod.***

### *3.3.1.3 DUV emulation*

Tem como objetivo introduzir e testar o *Driver* e *Monitor* no *testbench*. Ao introduzir estes dois elementos, todo o *testbench* deverá ser simulado para verificar se ambos estão realizando suas funções corretamente: O *Driver* deve transformar transações em sinais e o *Monitor* deve transformar sinais em transações. Porém, ao introduzí-los, surge a necessidade de que se tenha

*Professor associado da Universidade Federal de Campina Grande.

um DUV para receber os dados vindos do *Driver* e repassá-los para o *Monitor.* O problema é que o DUV ainda não foi implementado. Para resolver tal questão, um artificio é utilizado para emulá-lo (Figura 9): Um *Monitor* e *Driver* adicionais são acoplados ao segundo Modelo de Referência, fazendo com que este sistema tenha entradas e saídas no nível de sinais. BVM, em relação à metodologia VeriSC, possui um novo elemento neste subpasso, o *Actor* (Figura 10), responsável por controlar e monitorar os sinais do protocolo de comunicação entre o DUV e o *testbench.* Em VeriSC não existe um mecanismo para monitorar o protocolo de comunicação entre o DUV e o *testbench*. Além disto, o controle deste protocolo é implementado nos monitores (os *Tmonitor).* Assim, em VeriSC, o *TMonitor* não é um elemento passivo. Monitores passivos são desejáveis, pois podem ser inseridos em pontos de comunicação do DUV com qualquer outra entidade sem afetar esta comunicação. Em contraposição, os monitores de BVM são elementos passivos, visto que são os *Actor* que mantêm controle sobre o protocolo de comunicação entre o DUV e o *testbench*. Em BVM, os monitores apenas implementam a cobertura e repassam, no nível de transações, as saídas produzidas pelo DUV para o *Checker.*

O *Actor* é inserido entre o DUV e o *testbench*, mais especificamente entre o DUV e o *Monitor.* No caso específico deste subpasso, já que não se tem um DUV implementado em RTL, o *Actor* fica inserido entre o DUV emulado e o *Monitor* (*Actor* B)*,* como também entre o *Driver* e o DUV emulado (*Actor* A). Ambos os *Actor* possuem a mesma estrutura interna, porém com entradas e saídas relativas aos elementos nos quais estão conectados. A inserção do *Actor* A (Figura 10) visa evitar a escrita de código adicional para o controle de sinais entre o *Driver* e a tripla que emula o DUV.

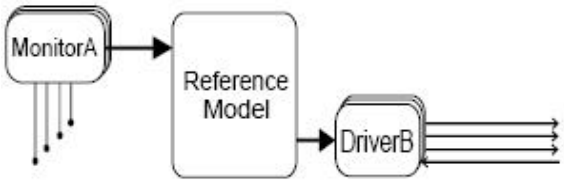


**Figura 9 – Representação do artificio usado para emular o DUV.**

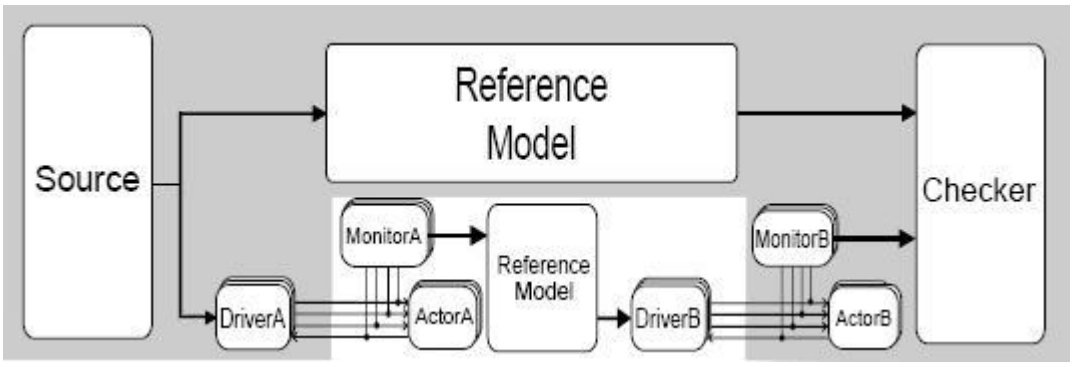


**Figura 10 – Representação do subpasso *DUV emulation*.**

Vale salientar, que assim como na metodologia VeriSC, em BVM faz-se reuso de vários dos elementos do *testbench*. Neste subpasso, por exemplo, o *Source* e o *Checker* são os mesmos utilizados no *Double refmod*.

## *3.3.2 Hierarchical refmod decomposition*

Assim como o passo *Testbench conception*, este passo é composto por três subpassos: *Refmod decomposition*, *Single hierarchical refmods* e *Hierarchical refmods verification*. O *Hierarchical refmod decomposition* tem como objetivo a realização da decomposição hierárquica do Modelo de Referência em blocos que correspondam aos blocos do DUV.

### *3.3.2.1 Refmod decomposition*

A fim de se facilitar o processo de verificação, o Modelo de Referência é dividido hierarquicamente em blocos, os quais devem ser equivalentes à decomposição hierárquica desejada para o DUV. Tal divisão tem como resultado a criação de blocos menores, os quais serão utilizados nos subpassos a seguir. Suponha que o Modelo de Referência apresentado possa ser dividido como apresentado na Figura 11.

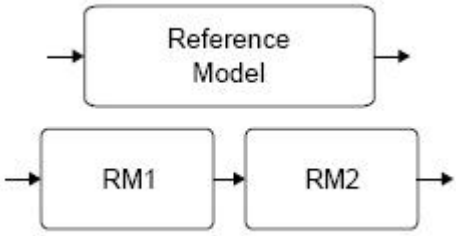


**Figura 11 – Representação da decomposição do Modelo de Referência.**

### *3.3.2.2 Single hierarchical refmods*

O *Single hierarchical refmods* realiza a mesma função do subpasso *Single refmod*. A diferença é que em vez se utilizar o Modelo de Referência inicial, utiliza-se os blocos resultantes de sua divisão hierárquica. Em outras palavras, os blocos resultantes do subpasso *Refmod decomposition*, serão estimulados individualmente para testar sua capacidade de interagir com o *testbench*. Os elementos de *testbench* aqui usados serão os mesmos do subpasso *Single refmod*: O *Source* e o *Sink.*


**Figura 12 – Representação do subpasso *Single hierarchical refmods* para o Modelo de Referência 1.**


**Figura 13 – Representação do subpasso *Single hierarchical refmods* para o Modelo de Referência 2.**

Fazer reuso de elementos que compõem o *testbench* reduz muito o tempo que o engenheiro de verificação gasta para construí-lo. Por exemplo, o *Source* apresentado na Figura 12 pode ser reusado do subpasso *Single refmod* (Figura 7), visto que ambos os modelos de referência apresentados nestas duas figuras possuem a mesma interface de entrada. Com este mesmo raciocínio, fica fácil perceber que o *Sink* apresentado na Figura 13 pode também ser reusado do passo *Single refmod*. Desta forma, não só neste subpasso, como nos demais, fazer reuso de elementos que compõem o *testbench* é de grande valia.

#### *3.3.2.3 Hierarchical refmods verification*

Neste cenário, os blocos resultantes do subpasso *Refmod decomposition* serão unidos, a fim de testar se tal união resulta na mesma funcionalidade do Modelo de Referência. Um *Source* irá estimular o Modelo de Referência e a união dos blocos. Um *Checker* irá coletar os resultados em ambas as saídas e compará-las. Caso tais saídas sejam iguais, cada bloco resultante da divisão hierárquica do Modelo de Referência passará a ser considerado um Modelo de Referência. Aqui o *Source* e o *Checker* podem ser reutilizados do passo *Double refmod*.

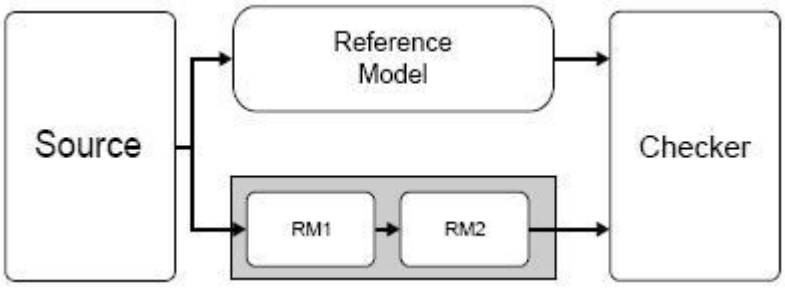


**Figura 14 – Representação do subpasso *Hierarchical refmods verification*.**

### *3.3.3 Hierarchical testbench*

Permite criar um *testbench* para cada cada um dos blocos da divisão hierárquica do DUV, vista a disponibilidade dos modelos de referência equivalentes gerados no passo anterior: *Hierarchical refmod decomposition*.

### 3.3.3.1 *Double hierarchical refmods*

Semelhante ao subpasso *Double refmod*, tem como objetivo inserir e testar o *Source* e o *Checker* para cada um dos blocos. Na Figura 15 é apresentada a mesma estrutura de *testbench* do subpasso *Double refmod* para o modelo de referência 1. De maneira análoga, na Figura 16 mostra-se tal *testbench* aplicado ao modelo de referência 2.

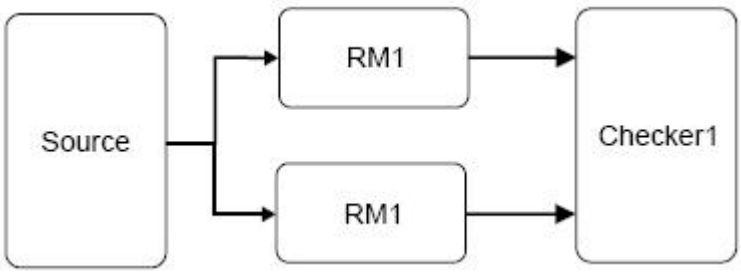


**Figura 15 – Representação do subpasso *Double hierarchical refmods* para o Modelo de Referência 1.**

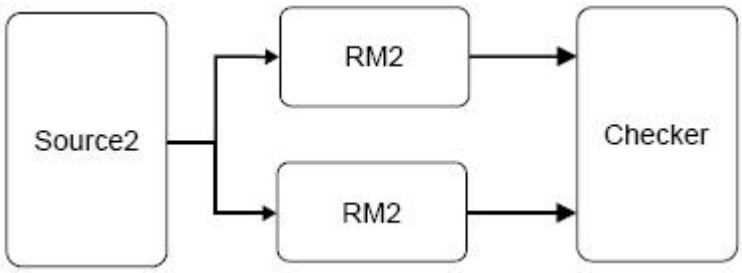


**Figura 16 – Representação do subpasso *Double hierarchical refmods* para o Modelo de Referência 2.**

### 3.3.3.2 *Hierarchical DUV emulation*

Este subpasso tem a mesma finalidade do passo *DUV emulation*, porém o *testbench* será construído individualmente para cada um dos blocos. Na Figura 17, é apresentado este *testbench* aplicado ao modelo de referência 1. Para esta primeira representação, os elementos *Driver* A, *Monitor* A e *Actor* A são reusados do subpasso *DUV emulation*, visto que o modelo de referência 1 possui as mesmas entradas do modelo de referência inicial. Da mesma maneira, a Figura 18 utiliza esta estrutura de *testbench* aplicada ao modelo de referência 2. Na segunda representação, os elementos reusados do subpasso *DUV emulation* são o *Driver* B, *Monitor* B e *Actor* B.

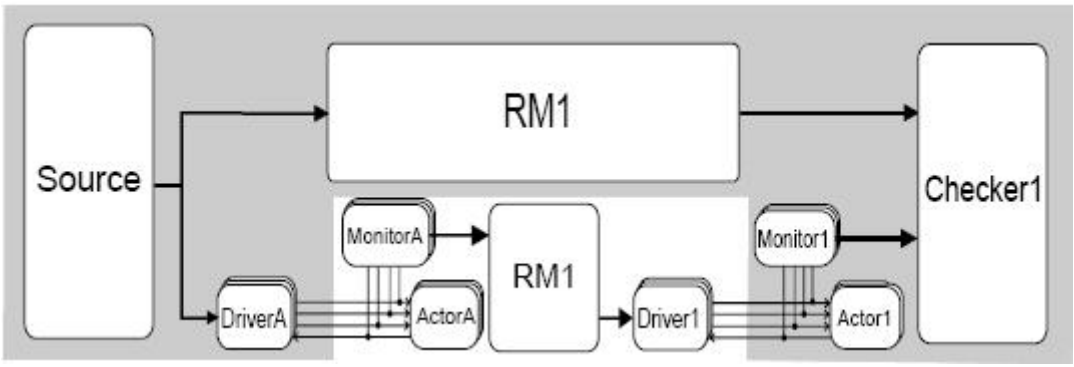


**Figura 17 – Representação do subpasso *Hierarchical DUV emulation* para o Modelo de Referência 1.**

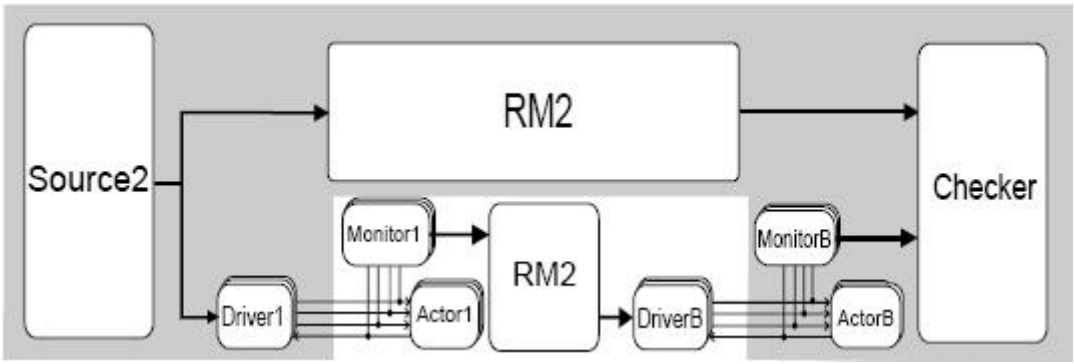


**Figura 18 – Representação do subpasso *Hierarchical DUV emulation* para o Modelo de Referência 2.**

### *3.3.3.3 Hierarchical DUV*

O *Hierarchical DUV* consiste em substituir a tripla *Driver*, Modelo de Referência e *Monitor*, pelo DUV propriamente dito de cada um dos blocos. Isso significa que a partir desse subpasso o RTL já deve ter sido implementado para realização da verificação funcional. Na Figura 19 é apresentada a a mesma estrutura de *testbench* do subpasso *Hierarchical DUV emulation* para o modelo de referência 1. No *Hierarchical DUV* o artifício utilizado para emular o DUV é substituído por um DUV descrito em RTL. De maneira análoga, na Figura 20 é mostrado tal *testbench* aplicado para o modelo de referência 2.

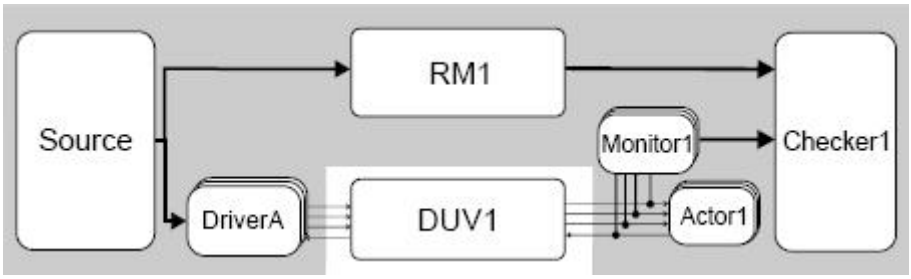


**Figura 19 – Representação do subpasso *Hierarchical DUV* para o Modelo de Referência 1.**

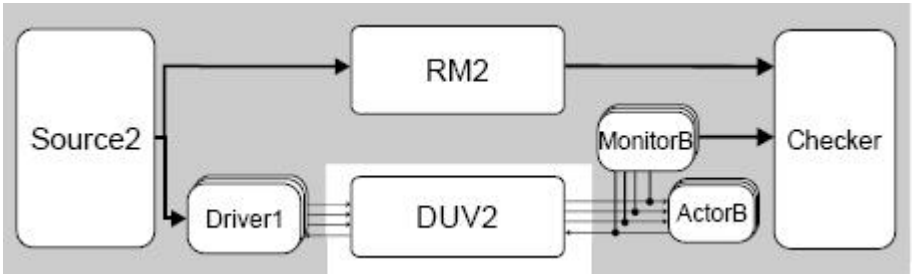


**Figura 20 – Representação do subpasso *Hierarchical DUV* para o Modelo de Referência 2.**

Caso o RTL testado neste subpasso não apresente inconsistências em relação aos modelos de referência originados da divisão hierárquica, passa-se para a próxima etapa: o passo *Full testbench*.

### *3.3.4 Full testbench*

O penúltimo passo consiste na interligação de todos os DUV resultantes do subpasso *Hierarchical* DUV. Esta união é feita para verificar se as funcionalidades providas pela união de todos os DUV é equivalente à funcionalidades do Modelo de Referência inicial. O *testbench* apresentado no subpasso *DUV emulation* será reutilizado, substituindo apenas a tripla *Driver*, Modelo de Referência e *Monitor*, pelo DUV completo, o qual é formado pela união dos pequenos DUV resultantes do subpasso *Hierarchical DUV*. Porém, nada garante que a união de todos os DUV não introduza erros. De acordo com Silva (2007), “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”. Em projetos complexos, a localização de erros de comunicação entre interfaces pode exigir grande esforço da equipe. A criação do *Actor* e o uso da FIFO de duas saídas viabilizou a criação de um mecanismo de localização de erros de comunicação entre essas interfaces, diminuindo o tempo necessário para encontrá-los, caso ocorram.

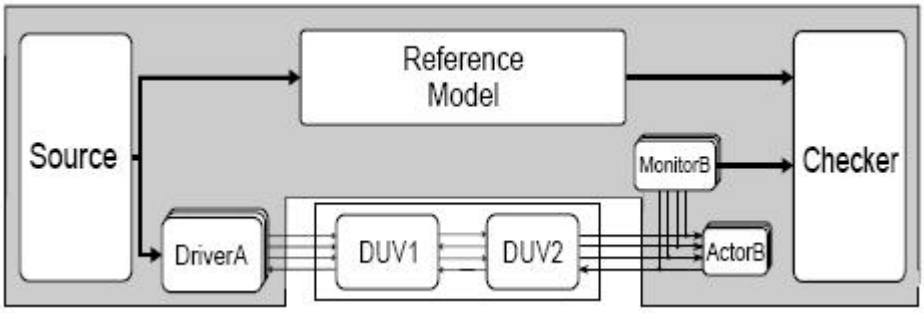


**Figura 21 – Representação do passo *Full testbench*.**

Perceba-se que não existe nenhum elemento *Actor* intermediando o protocolo de

comunicação entre os DUV (Figura 21). Isto ocorre porque o DUV 2 implementa o mesmo protocolo de comunicação descrito no *Actor* 1. Nos subpassos anteriores este elemento simulava o protocolo de comunicação do DUV 1 com o DUV 2. Sendo mais claro, o DUV 1 se comunica com o DUV 2 de maneira idêntica àquela em que se comunicava com o *Actor* 1 (Figura 22).

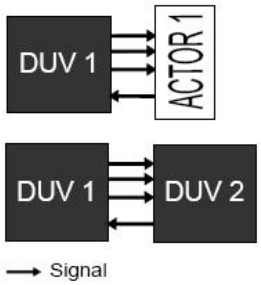


**Figura 22 – Protocolo de comunicação do DUV 1 com *Actor* 1 e do DUV 1 com o DUV 2.**

Esta equivalência no protocolo de comunicação torna possível que monitores sejam inseridos entre os DUV, viabilizando assim cobertura de valores em pontos específicos de sua união (Figura 23).

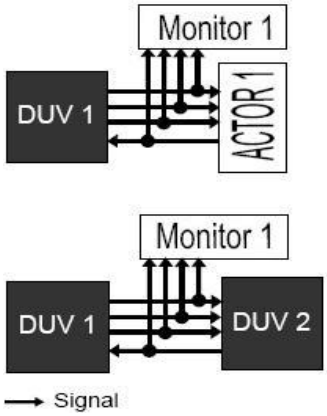


**Figura 23 – Possível inserção de monitores entre os DUV.**

Em BVM, monitores passivos são consequência da criação do elemento *Actor*. Monitores passivos podem ser inseridos entre a ligação dos DUV, visto que não interferem no protocolo de comunicação destes. Desta forma, monitores podem coletar dados de cobertura e converter os dados no nível de sinais para nível de transação. O uso de monitores passivos juntamente com o uso da FIFO de duas saídas viabilizaram um mecanismo para localização de possíveis erros na integração dos DUV. resultando em um passo inexistente na metodologia VeriSC.

## *3.4 Full testbench analysis*

Na etapa de integração dos DUV, erros de comunicação entre suas interfaces não são fáceis de detectar. Inexistente na metodologia VeriSC, o passo *Full testbench analysis* é uma solução para

detecção destes erros. Suponha um DUV que pode ser hierarquicamente dividido em quatro DUV (Figura 24). Suponha-se também que todos os passos para construção do *testbench* da metodologia BVM foram executados com sucesso, exceto no *Full testbench*.

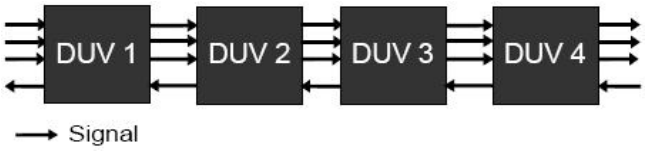


**Figura 24 – Representação da decomposição hierárquica do DUV.**

Ao analisar o cenário previamente delineado, pode-se deduzir que a integração dos DUV no último passo possui algum erro. Provavelmente existe um erro no protocolo de comunicação entre o primeiro e o segundo DUV. Em projetos complexos, a detecção deste tipo de erro utilizando a metodologia VeriSC poderia levar muito tempo, visto não ser viabilizado artifício para detectá-los. A inovação de BVM está na possibilidade de inserir monitores entre qualquer par de DUV. Este mecanismo só foi possível com a criação do elemento *Actor*, tornando estes monitores elementos passivos. Devido ao reuso de código, monitores dos passos anteriores podem ser inseridos entre qualquer par de DUV de maneira rápida. Os monitores agora podem coletar dados de cobertura do ponto desejado, detectando se neste ponto existe falha de comunicação entre os DUV. O mecanismo de detecção de erros é composto por uma das saídas da FIFO de duas saídas, um Monitor e um C*hecker*. Para fins de simplificação, na Figura 25, o mecanismo de detecção (área tracejada da Figura 25) é mostrado apenas entre o DUV 1 e o DUV 2, porém este mecanismo deve ser aplicado entre todos os pares de DUV.

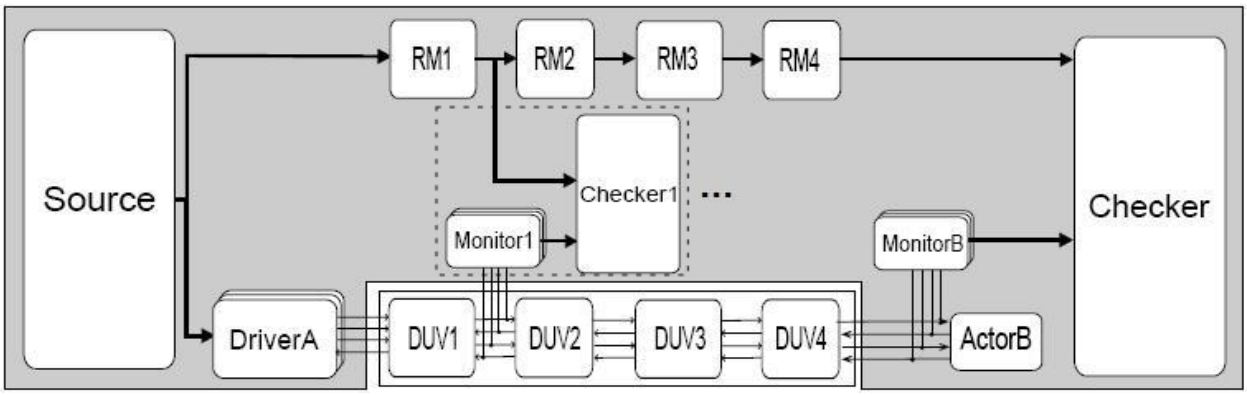


**Figura 25 – Inserção de um *Monitor* para a detecção do erro entre o DUV 1 e DUV 2.**

Para detecção de tais erros, em vez de usar o Modelo de Referência inicial, é utilizada a união dos modelos de referência gerados pelo subpasso *Refmods decomposition.* O *Monitor 1,* utilizado para coletar as saídas do DUV 1, é reusado do subpasso *Hierarchical DUV,* sendo inserido entre o DUV 1 e o DUV 2. Assim como o *Monitor 1,* o *Checker 1* também é reaproveitado

deste mesmo subpasso. O uso da FIFO de duas saídas permite reproduzir os dados que trafegam entre qualquer par de modelos de referência. Estes dados podem ser repassados para qualquer outro elemento do *testbench*. Na Figura 25, além de ser apresentada como saída do *Source*, a FIFO de duas saídas é mostrada entre os modelos de referências 1 e 2 (RM 1 e RM 2). Desta maneira, após conectar todos os elementos necessários na construção do mecanismo de detecção de erros, é possível analisar a cobertura, assim como os valores gerados entre esses dois DUV.

## *3.5 A ferramenta eTBc*

Na fase de implementação do *testbench*, linguagens de verificação de hardware são frequentemente usadas para seu desenvolvimento. A ferramenta eTBc (PESSOA, 2007) tem como objetivo automatizar o processo de construção de ambientes de verificação, reduzindo consideravelmente o tempo que o engenheiro levaria para construí-lo. Sua flexibilidade permite que este ambiente possa ser implementado em qualquer linguagem, de acordo com qualquer metodologia de verificação.

A ferramenta trabalha como uma geradora de código (Figura 26) e possui dois arquivos como entrada: um chamado de TLN (*Transaction Level Netlist*) e o outros chamados de *Template Patterns*.

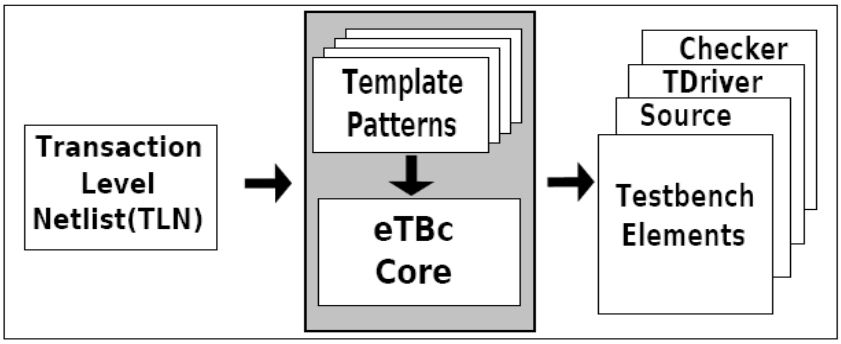


**Figura 26 – Representação da ferramenta eTBc (PESSOA, 2007).**

A TLN é um arquivo descrito pelo engenheiro de verificação, que descreve como o IP será modularizado, como também as entradas e saídas desses módulos em termos de sinais e transações. A linguagem utilizada para essa descrição é chamada de eDL (eTBc *Design Language*). Usuários da ferramenta não precisam se preocupar com os *Template Patterns*, pois estes são de responsabilidade de quem desenvolve a metodologia de verificação. Tais arquivos guiam a geração de código fonte, de acordo com a TLN definida pelo usuário. A linguagem utilizada para criação dos *Templates* é a eTL (eTBc *Template Language*). Os arquivos de saída são chamados de *Testbench Elements*, os quais são código compilável na linguagem usada para

implementação da metodologia de verificação, tal como SystemC, ou SystemVerilog.

A TLN é específica para o circuito que está sendo projetado, enquanto os *template patterns* são específicos da metodologia e linguagem utilizada. A saída da ferramenta eTBc nada mais é do que código compilável que pode representar qualquer um dos elementos do *testbench*: *Source*, *Driver*, *Monitor*, *Checker*, *Actor* e Modelo de Referência. Os elementos de *tesbench* gerados são garantidos em termos de compilação e simulação sem erros em tempo de execução desde que não haja erro no núcleo da ferramenta nem nos *template patterns*. No entanto, estes *testbench elements* necessitam de código específico, tal como a implementação das funcionalidades do Modelo de Referência ou implementação de *handshake* de protocolo de sinais, por exemplo. Desta maneira, este tipo de código terá que ser inserido de maneira manual pelo engenheiro de verificação.

Os *template patterns* apresentados a seguir (Códigos 1 e 2) são arquivos ASCII que contêm o código SystemC ou SystemVerilog que são comuns a qualquer instância de *template* de um *testbench* em eTL. A idéia dos *template patterns* é orientar a ferramenta eTBc para gerar elementos do *testbench*. Nos códigos, cada palavra-chave eTBc delimitada por "$$ (" e ")" é uma entrada eTBc *Template Language* (eTL). Quando a ferramenta analisa esses arquivos ela irá substituir estas entradas por um código baseado em declarações do arquivo TLN.

**Trecho de código do *template pattern* para gerador de estímulos em VeriSC**

```
1    ...
2    $$(foreach) $$(module.in)
3    class $$(i.name)_constraint_class: public scv_constraint_base {
4      $$(foreach) $$(var)
5       scv_bag<pair<$$(i.type),$$(i.type)> > $$(i.name)_distrib;
6      $$(endfor)
7     public:
8      scv_smart_ptr<$$(i.type)> $$(i.type)_sptr;
9      SCV_CONSTRAINT_CTOR($$(i.name)_constraint_class) {
10     $$(foreach) $$(var)
11     //$$(i.name)_distr.push(pair<$$(i.type),$$(i.type)>(?,?), ?);
12     // $$(j.type)_sptr->$$(i.name).set_mode($$(i.name)_distr);
13     $$(endfor)
14     // SCV_CONSTRAINT($$(i.type)_sptr->?() > ?);
15    }
16   }
17   ...
```

**Código 1 - Trecho de código do *template pattern* para gerador de estímulos em VeriSC.**

**Trecho de código do *template pattern* para gerador de estímulos em BVM**

```
1    ...
2    class $$(i.name) extends ovm_object;
3      $$(foreach) $$(var)
4      rand $$(i.type) $$(i.name);
5      constraint $$(i.name)_range {
```

**Trecho de código do *template pattern* para gerador de estímulos em BVM (Continuação)**

```
6       //$$(i.name) dist { [?:?] };
7      }
8    $$(endfor)
9    ...
```

**Código 2 - Trecho de código do *template pattern* para gerador de estímulos em BVM.**

Para que a ferramenta eTBc dê suporte a alguma linguagem ou metodologia, esta poderá necessitar de alguma adaptação, a partir de correções feitas em seu *core* como também na criação de novos *Template Patterns*. Antes da criação de BVM, esta ferramenta dava suporte apenas à metodologia VeriSC (SILVA, 2007).

De acordo com (PESSOA et al., 2007), o uso de eTBc na construção automática do ambiente de verificação utilizando a metodologia VeriSC, apresenta um ganho de 83% em relação a um processo manual. É por este motivo que adaptar esta ferramenta para suportar BVM é um dos objetivos do trabalho.

Utilizando BVM e a ferramenta eTBc, esta fase do processo de desenvolvimento do *hardware* poderá ser feita de maneira mais rápida, diminuindo os recursos necessários para sua realização. A seção de implementação de BVM (Seção 3.8) dará uma visão geral de como a ferramenta eTBc é utilizada na geração de *testebenches* para esta metodologia. Para obter mais detalhes sobre eTBc pode-se consultar o trabalho de Pessoa (PESSOA, 2007).

## *3.6 A biblioteca OVM*

A metodologia OVM oferece uma biblioteca para verificação dirigida por cobertura (CDV – *Coverage Driven Verification*). CDV combina geração de testes automáticos, *testbenches* auto-verificáveis (*self-checking)* e métricas de cobertura para reduzir o tempo gasto na verificação de um projeto (CADENCE; MENTOR, 2009b, p. 9).

Na metodologia BVM, esta biblioteca foi utilizada para conectar os elementos do *testbench* por TLM FIFO. O *testbench* em OVM é composto de ambientes de verificação reusáveis denominados de OVC (*OVM Verification Component*). Um OVC é um ambiente de verificação encapsulado, configurável e pronto para uso, tanto para um protocolo de interface, quanto para um submódulo de um projeto ou para um sistema inteiro. Cada componente segue uma arquitetura consistente e consiste em um conjunto de elementos para estimular, checar, e coletar informações de cobertura de um protocolo específico ou projeto (CADENCE; MENTOR, 2009b, p. 9). Os componentes que constituem esses OVC assemelham-se aos componentes utilizados na metodologia VeriSC: *Source*, *Driver*, *Monitor*, *Checker*, etc. Os componentes que constituem os

OVC são seis: *Data Item, Driver, Sequencer, Monitor, Agent e Enviroment*. A seguir, é apresentado um breve resumo sobre cada um deles, de acordo com o livro OVM User Guide (CADENCE; MENTOR, 2009b).

• ***Data Item:*** Representa a entrada do DUV. São os estímulos gerados para o DUV. Comparando com a metodologia BVM, pode-se associar esse elemento com as transações geradas pelo *testbench element Source*.

• ***Driver:*** É responsável por receber *Data items* e enviá-los ao DUV em forma de sinais. Controla sinais de escrita e leitura, barramento de endereços e barramento de dados para um número de ciclos de *clocks* para realizar uma transferência de dados. Esse elemento reflete a funcionalidade do *Driver* da metodologia BVM.

• ***Sequencer:*** Tem a mesma função do *testbench element Source* em BVM. É um gerador de estímulos que controla os *Data itens* fornecidos para o *Driver.* Este componente permite a adição de *constraints*[7] para a classe *Data Item* a fim de controlar a distribuição dos valores randômicos. Mais detalhes desse componente podem ser vistos em OVM User Guide (CADENCE; MENTOR, 2009b).

• ***Monitor:*** Coleta informações de cobertura e realiza a checagem de valores. A seguir algumas de suas características.

  • Coleta transações (*Data Itens*). Extrai informações dos sinais do barramento e transforma esta informação em uma transação, tornando-a disponível para outros componentes e engenheiro de verificação.

  • Extrai eventos. Detecta a disponibilidade de informação, estruturas de dados, e emite um evento para notificar outros componentes da disponibilidade da transação.

  • Coleta dados de cobertura e checagem de dados e protocolo para verificar se a saída do DUV atende à especificação de protocolo.

Um monitor de barramento (*bus monitor*) lida com todos os sinais e as transações em um barramento, enquanto um monitor de agente (*agent monitor*) trata apenas de sinais e operações relevantes para um agente específico.

Na metodologia BVM, as funcionalidades providas por esse componente OVC equivalem as do *testbench element Monitor*. Em BVM dados de cobertura são coletados no *Monitor*. O *Actor* realiza o monitoramento e controle dos sinais de protocolo de

7 *Constraints* são restrições aplicadas aos geradores de estímulos com o objetivo de produzir valores pseudo-randômicos.

comunicação entre o DUV e o *Monitor*. O *Actor* criado em BVM foi baseado no conceito de *responder* da metodologia OVM. Para obter mais detalhes sobre o elemento *responder* pode-se consultar o trabalho de Glasser (GLASSER, 2009, p. 21).

• ***Agent:*** É um *container* que encapsula *driver, monitor e sequencer*. Esse componente pode emular e verificar o DUV. Um OVC pode possuir um ou mais agentes. Por exemplo, eles podem iniciar transações no DUV (*master* ou *transmit agent*), enquanto outros agentes reagem a solicitações de transação. Podem ser do tipo ativo ou passivo. Os ativos emulam os DUV e direcionam transações de acordo com as diretivas de testes. Já os agentes passivos monitoram somente a atividade do DUV.

• ***Environment:*** É um componente OVC *top-level*. Contém um ou mais agentes, assim como outros componentes como *bus monitors*. O *environment* contém propriedades de configuração que permitem a personalização da topologia e comportamento, tornando-o reusável. Um exemplo de reuso de um ambiente de verificação é que agentes ativos podem ser mudados para agentes passivos. Na Figura 27, é apresentada a estrutura de um ambiente de verificação reusável. Na metodologia BVM, este componente corresponde ao ambiente de *testbench*, já que contém todos os elementos de *testbench* necessários para realização da verificação funcional.

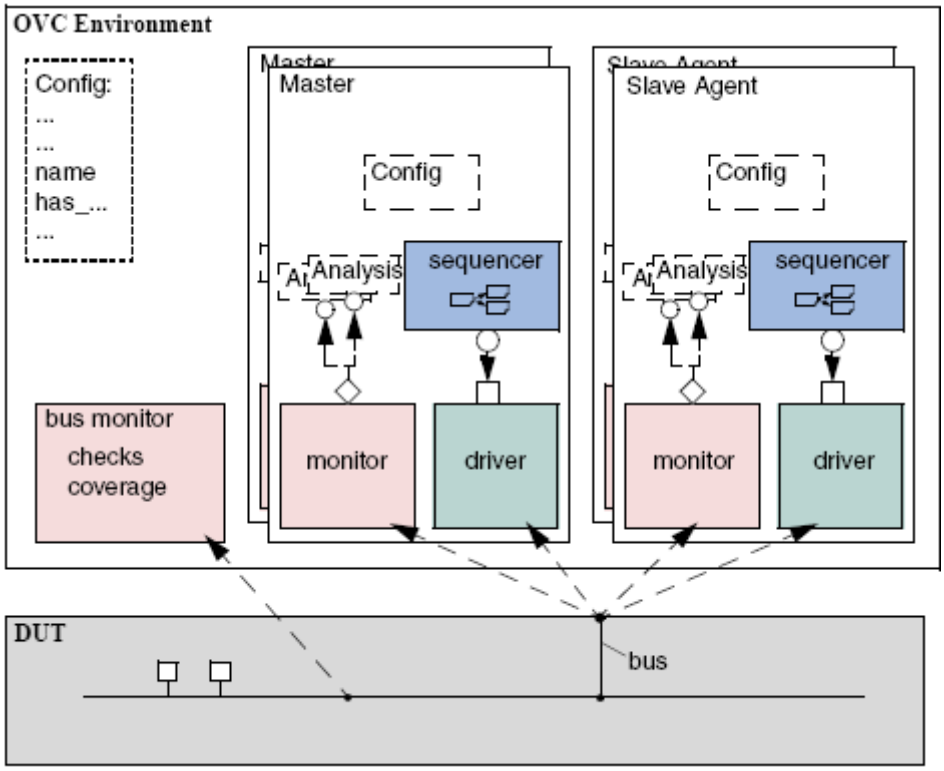


**Figura 27 – Ambiente de verificação em OVM (CADENCE; MENTOR, 2009b, p.15).**

Os criadores da metodologia OVM afirmam que a biblioteca disponibilizada "*SystemVerilog OVM class Library*" fornece todos os blocos de construção necessários para o desenvolvimento de componentes de verificação e ambiente de testes reusáveis, de maneira rápida e bem elaborada. Na Figura 28, é apresentada a hierarquia das classes que compõem a biblioteca OVM.

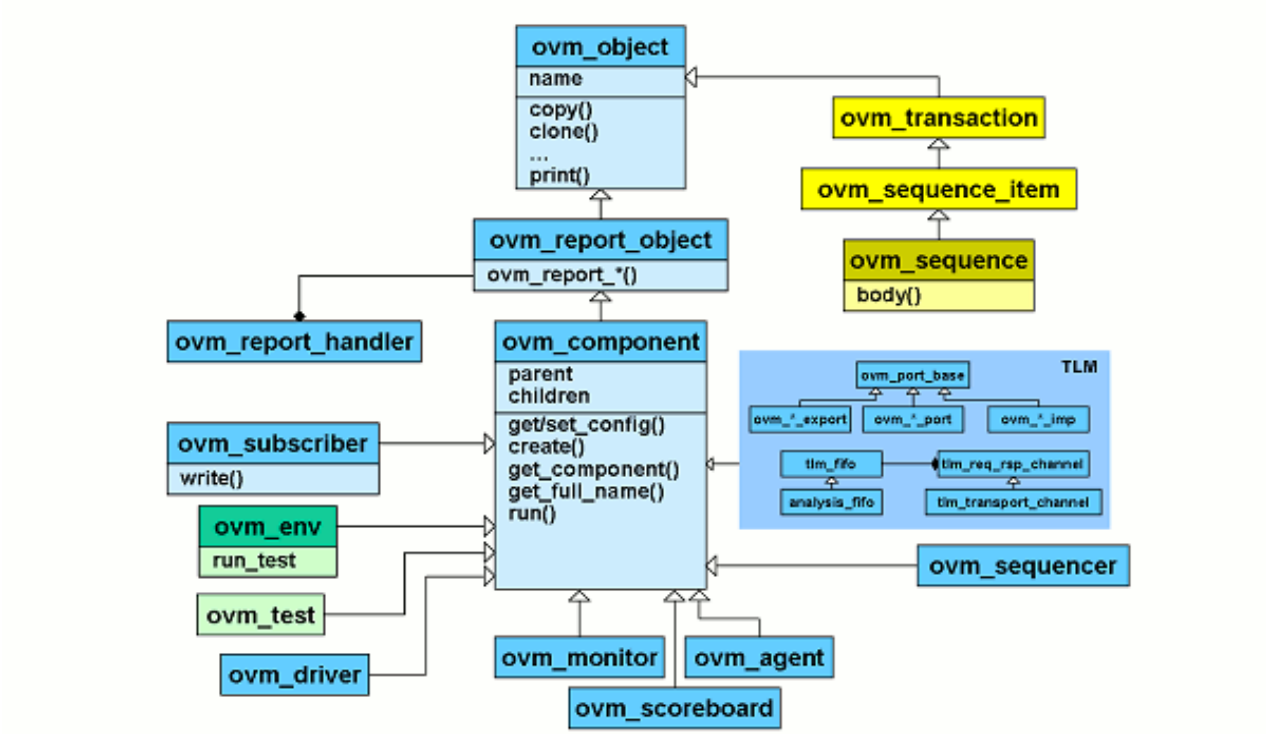


**Figura 28 – Hierarquia das classes que compõe a biblioteca OVM (CADENCE; MENTOR, 2009b, p.16).**

A seguir, é apresentada uma parte do código fonte do elemento de *testbench Source* (Código 3). Percebe-se, no código, que a classe *Source* estende a classe *ovm_component* (linha 2), apresentada na Figura 28. A chamada ao método run() faz com que os valores de saída deste gerador sejam colocados nas portas de saída "ovm_put_port #(sample) in_dpcm_to_refmod_port" (linha 3) e "ovm_put_port #(sample) in_dpcm_to_driver_port" (linha 4).

**Código para *testbench element Source***

```
1         class source extends ovm_component;
2           ovm_put_port #(sample) in_dpcm_to_refmod_port;
3           ovm_put_port #(sample) in_dpcm_to_driver_port;
4
5           function new(string name, ovm_component parent);
6            super.new(name,parent);
7            in_dpcm_to_refmod_port = new("in_dpcm_to_refmod_port",
8         this);
9            in_dpcm_to_driver_port = new("in_dpcm_to_driver_port",
10        this);
11          endfunction
12
```

**Código para *testbench element Source* (Continuação)**

```
13          task run();
14            sample tr_in_dpcm;
15            int file = $fopen("tr.sti", "r");
16            forever begin
17              tr_in_dpcm = new();
18              if(!tr_in_dpcm.read(file))
19                assert(tr_in_dpcm.randomize());
20              in_dpcm_to_driver_port.put(tr_in_dpcm);
21              in_dpcm_to_refmod_port.put(tr_in_dpcm);
22              #1;
23            end
24          endtask
25
26        endclass
```

**Código 3 - Código para *testbench element Source.***

Este é apenas um pequeno exemplo de uso da biblioteca OVM. Várias outras classes necessárias para a construção e integração dos *testbench elements* dos passos da metodologia BVM utilizam vários outros recursos. Na seção de implementação, serão apresentados mais detalhes do uso da biblioteca OVM aplicada à BVM.

É fácil perceber a semelhança entre os elementos que compõem o *testbench* da metodologia VeriSC e da metodologia OVM. Tal semelhança foi um fator que também contribuiu para uso da biblioteca OVM na construção dos elementos dos *testbenches* que compõem os passos de BVM. Surge então, o questionamento: Por que criar BVM, já que os elementos que compõem o *testbench de* OVM assemelham-se aos de VeriSC e ambas metodologias tem como objetivo a verificação funcional?

VeriSC possui bastante semelhança como a metodologia OVM, em relação aos elementos que a compõem, porém, além de ser implementada em SystemC, a estrutura e o processo de construção do *testbench* propostos por VeriSC são totalmente diferentes da metodologia OVM. Da mesma forma que VeriSC, a construção do *testbench* em BVM, além de ser um processo gradativo, é baseada na especificação do *hardware* a ser criado como também em um Modelo de Referência do mesmo. De maneira oposta, o processo de construção desse ambiente em OVM não é gradativo e não menciona o uso de um Modelo de Referência para tal. Em BVM, o Modelo de Referência e o DUV são estimulados com valores, e suas saídas são comparadas. OVM não realiza tal comparação, visto que seu *testbench* não obriga o uso de um Modelo de Referência. BVM foi criada a partir de uma reformulação de VeriSC, visando melhorias e manutenção de suas vantagens, como também uma maior aproximação da metodologia OVM.

BVM, assim como VeriSC, tem como objetivo a construção do *testbench* antes da implementação do DUV. OVM não possui esta preocupação. VeriSC foi utilizada com sucesso na

verificação funcional dos IP *cores* produzidos pelo Brazil-IP até o ano de 2008. Desta forma, BVM, por ser uma reformulação de VeriSC e implementada em uma linguagem mais voltada para verificação funcional, também foi utilizada com sucesso na verificação dos atuais IP *cores* produzidos pelo Brazil-IP.

## *3.7 Diferenças entre BVM e VeriSC*

Existem várias diferenças entre BVM e VeriSC. As diferenças vão desde a metodologia até aspectos relacionados à suas implementações. Uma diferença marcante em relação a estas metodologias é a inserção do elemento *Actor* no *testbench* de BVM. Conforme já explicado anteriormente, este elemento é de grande utilidade na verificação e monitoramento do protocolo de comunicação do DUV com o *testbench*. Além disto, monitores passivos de BVM são consequência da criação do elemento *Actor*.

A concepção da FIFO de duas saídas é uma característica intrínseca de BVM. Esse artifício foi idealizado juntamente com professor Elmar Uwe Kurt Melcher e implementado por um integrante do Brazil-IP*. O uso deste tipo de FIFO viabilizou, juntamente com o uso de monitores passivos, o mecanismo de detecção de erros na fase de integração dos DUV (ver Seção 3.4).

Outro aspecto importante a se mencionar é a forma com que a implementação destas metodologias trabalham a questão da cobertura. VeriSC usa a biblioteca de cobertura BVE (*Brazil-IP Verification Extensions*) (SILVA, 2007). Esta biblioteca foi desenvolvida pelos membros do Brazil-IP, a fim de se implementar cobertura funcional, já que SystemC não possui recursos intrínsecos em sua linguagem para trabalhar com essa questão. De maneira oposta, BVM utiliza características nativas de SystemVerilog para a execução da cobertura funcional. No Código 4, é apresentado um critério de cobertura para a metodologia VeriSC, o qual define que a simulação pare somente após ter sido alcançado 15 vezes o número zero, 15 vezes o numero 1, e 15 vezes um valor entre 3 e 7, usando a biblioteca BVE. No Código 5, é mostrado o mesmo critério usando SystemVerilog.

**Exemplo de Cobertura funcional em VeriSC**

```
1        bve_cover_bucket output_checker_cv;
2        output_checker_cv.begin();
3         BVE_COVER_COND(output_checker_cv, x==0, 15);
4         BVE_COVER_COND(output_checker_cv, x==1, 15);
5         BVE_COVER_COND(output_checker_cv, x==2, 15);
6         BVE_COVER_COND(output_checker_cv, x>=3 && x<=7, 15);
7        output_checker_cv.end();
```

**Código 4 – Exemplo de Cobertura funcional em VeriSC.**

*Jorgeluis Andrade Guerra

**Exemplo de Cobertura funcional em BVM**

```
1         covergroup cg;
2          coverpoint tr_output_checker.value {
3           bins tr_x[] = { 0, 1, 2, [3:7] };
4           option.at_least = 15;
5          }
6         endgroup
7         cg = new;
```

**Código 5 – Exemplo de Cobertura funcional em BVM.**

A forma como VeriSC trata valores ilegais também é diferente de BVM. Em VeriSC é necessário declarar uma variável do tipo “bve_cover_illegal” para que caso valores não desejados ocorram, estes sejam devidamente identificados. De maneira oposta, em BVM a cobertura de valores ilegais é feita na própria declaração do *covergroup*[8] responsável por tratar valores legais. No código 6 é apresentada a cobertura para valores ilegais em VeriSC. Os valores 8, 11 e 12 são considerados ilegais. Em BVM o mesmo critério de cobertura pode ser simplesmente adicionado ao exemplo apresentado no Código 5 (Código 7).

**Exemplo de Cobertura funcional para valores ilegais em VeriSC**

```
1      bve_cover_illegal output_illegal_checker_cv;
2      output_illegal_checker_cv.begin();
3       BVE_COVER_ILLEGAL(output_illegal_checker_cv, x==8);
4       BVE_COVER_ILLEGAL(output_illegal_checker_cv, x==11);
5       BVE_COVER_ILLEGAL(output_illegal_checker_cv, x==12);
6      output_illegal_checker_cv.end();
```

**Código 6 – Exemplo de Cobertura funcional para valores ilegais em VeriSC.**

**Exemplo de Cobertura funcional para valores ilegais em BVM**

```
1         covergroup cg;
2          coverpoint tr_output_checker.value {
3           bins tr_x[] = { 0, 1, 2, [3:7] };
4           option.at_least = 15;
5           illegal_bins bad_x = { 8, 11, 12 };
6          }
7         endgroup
8         cg = new;
```

**Código 7 – Exemplo de Cobertura funcional para valores ilegais em BVM.**

8 Em SystemVerilog um *covergroup* encapsula a especificação de um modelo de cobertura (IEEE STANDARD, 2005 b).

Outra característica importante oferecida pela implementação de BVM, devido ao uso da linguagem SystemVerilog, é a possibilidade de cruzamento de cobertura (*Cross Coverage*). Por exemplo, suponha-se que se tenha um *covergroup* com dois *coverpoints*[9] "a" e "b", os quais possuem cada um 4 *bins*[10] quaisquer. Suponha-se que se deseja obter todas as combinações destes *coverpoints*. No Código 8, é apresentado um exemplo de cobertura deste tipo.

**Exemplo de *Cross Coverage* em BVM**

```
1         bit [7:0] v_a, v_b;
2
3         covergroup cg @(posedge clk);
4         a: coverpoint v_a
5         {
6          bins a1 = { [0:63] };
7          bins a2 = { [64:127] };
8          bins a3 = { [128:191] };
9          bins a4 = { [192:255] };
10        }
11        b: coverpoint v_b
12        {
13         bins b1 = {0};
14         bins b2 = { [1:84] };
15         bins b3 = { [85:169] };
16         bins b4 = { [170:255] };
17        }
18        c : cross a, b;
19
20        endgroup
```

**Código 8 – Exemplo de *Cross Coverage* em BVM.**

No Código 8, o *cross coverage* é definido em "c" (linha 18). A combinação destes dois *coverpoints* resultará em "c" 16 *cross products*: <a1,b1>, <a1,b2>, <a1,b1>, <a1,b3>, ….,<a4,b1>, <a4,b2>, <a4,b3>, <a4,b4>. Expressões de seleção (c*ross bins)* também podem ser adicionadas no *cross coverage*, tornando este recurso ainda mais poderoso. Para obter mais detalhes, ler o padrão IEEE de SystemVerilog (IEEE STANDARDS, 2005 b).

Diferentemente de VeriSC, na metodologia BVM, o desenvolvedor não estava preocupado em criar uma classe para realizar a cobertura funcional. SystemVerilog garante uma forma eficiente e concisa de realizá-la.

A biblioteca OVM foi usada para criar *testbenches* em BVM, oferecendo uma maneira simples e eficiente de conectar e usar os elementos do *testbench* por TLM FIFO, substituindo os sc_fifo utilizados na implementação de VeriSC.

9 Em SystemVerilog *coverpoints* são pontos de cobertura definidos dentro de um *covergroup,* os quais podem ser uma variável ou uma expressão.
10 Em SystemVerilog *bins* são valores definidos dentro dos *coverpoints.*

Em VeriSC, a biblioteca SCV foi usada para gerar estímulos. *Constraints* são declaradas para gerar valores pseudo-randômicos. Um exemplo é mostrado no Código 9, no qual são gerados valores pseudo-randômicos entre zero e 10, e 15 e 40, na proporção de 15 e 20, respectivamente. SystemVerilog permite aos usuários especificar *constraints* de uma maneira compacta. Elas são então processadas por um *solver* que gera valores randômicos que atendam as *constraints*. O mesmo exemplo é apresentado no Código 10.

***Constraints* para gerar estímulos em VeriSC**

```
1        class x_constraint_class: public scv_constraint_base {
2         scv_bag<pair<int,int> > x_distrib;
3        public:
4         scv_smart_ptr<int> data;
5         x_distrib.push( pair<int,int>(0,10), 15);
6         x_distrib.push( pair<int,int>(15,40), 20);
7         data->set_mode(x_distrib);
8        };
```

**Código 9 – *Constraints* para gerar estímulos em VeriSC.**

***Constraints* para gerar estímulos em BVM**

```
1        class x;
2         rand int x_distrib;
3         constraint x_distrib_range {
4         x_distrib dist { [0:10] := 15, [15:40] := 20 };
5        }
6        endclass
```

**Código 10 – *Constraints* para gerar estímulos em BVM.**

Todos os elementos do *testbench* de BVM que são ligados por modelagem TLM estendem uma classe OVM. Por exemplo, o elemento *Source* de BVM estende a classe *ovm_component*. Todos estes elementos podem possuir portas de entradas e/ou saídas. Tais portas são conectadas de maneira simples por um módulo responsável por ligar todos os elementos do *testbench*. A seguir, é apresentado um trecho de código deste módulo para a metodologia VeriSC (Código 11) e BVM (Código 12).

***Trecho de código de módulo responsável por conectar elementos do testbench em VeriSC***

```
1    ...
2     bve_fifo<packet*>packetA_presource_refmod("packetA_presource_re
3     fmod",logfile);
4     bve_fifo<long_packet*>packetB_refmod_sink("packetB_refmod_sink”
5     ,logfile);
6
```

***Trecho de código de módulo responsável por conectar elementos do testbench* em VeriSC (Continuação)**

```
7     ...
8
9     pre_source pre_source_i("pre_source_i");
10    Sink sink_i("sink_i");
11    Refmod refmod_i("refmod_i");
12
13   ...
14    pre_source_i.pkt_a_to_refmod(packetA_presource_refmod);
15    refmod_i.pkt_a_stim(packetA_presource_refmod);
16
17    refmod_i.pkt_b_stim(packetB_refmod_sink);
18    sink_i.pkt_b_from_refmod(packetB_refmod_sink);
19   ...
```

**Código 11 – Trecho de código de módulo responsável por conectar elementos do *testbench* em VeriSC.**

As linhas 2 e 3 do código acima (Código 11) declaram um FIFO TLM que irá conectar o *pré-source* a um Modelo de Referência. As linhas 4 e 5 declaram a FIFO de conexão entre o Modelo de Referência e o *sink*. As linhas 9 a 11 instanciam todos os elementos necessários para a construção do *testbench*. O restante do código conecta todos estes elementos mediante estas FIFO. O Código 12 pode ser entendido da mesma maneira que já foi anteriormente explicado.

***Trecho de código de módulo responsável por conectar elementos do testbench em BVM***

```
1    ...
2     tlm_fifo #(packet) packetA_source_refmod =
3     new("packetA_presource_refmod", null, 3);
4     tlm_fifo #(long_packet) packetB_refmod_sink =
5       new("packetB_refmod_sink", null, 3);
6
7    ...
8     source source_i = new("source_i", null);
9     refmod_X refmod_X_i = new("refmod_X_i", null);
10    sink sink_i = new("sink_i", null);
11
12   ...
13      source_i.pkt_a_to_refmod.connect(packetA_source_refmo
14      d.put_export);
15      refmod_X_i.pkt_a_stim.connect(packetA_source_refmod.get_e
16      xport);
17
18      refmod_X.pkt_b_stim.connect(packetB_refmod_sink.put_ex
19      port);
20      sink_i.pkt_b_from_refmod.connect(packetB_refmod_sink.g
21      et_export);
22   ...
```

**Código 12 – Trecho de código de módulo responsável por conectar elementos do *testbench* em BVM.**

Outra diferença entre BVM e VeriSC é como o Modelo de Referência é usado. Em VeriSC e BVM, o Modelo de Referência é frequentemente escrito em C/C++ e isto é uma vantagem em

VeriSC porque ela usa SystemC que trabalha com ambientes C/C++. Toda chamada de função pode ser feita diretamente pelo SystemC do Modelo de Referência. Em BVM, esta tarefa pode ser executada usando DPI (*Direct Programming Interface*) que permite chamadas de funções C/C++ do ambiente SystemVerilog.

# Capítulo 4

# 4 Resultados

## *4.1 Implementação de BVM*

Para a criação de cada um dos *testbenches* dos subpassos da metodologia BVM, é necessário um conjunto de *template patterns* e uma TLN como entrada para a ferramenta eTBc. Tais *templates* foram criados no decorrer do trabalho apresentado. Nas próximas subseções, segue uma descrição de quais *template patterns* são necessários para construção de cada um desses *testbenches,* assim como suas funções e alguns detalhes de implementação.

### *4.1.1 Testbench conception*

#### *4.1.1.1 Single refmod*

Neste subpasso, foram utilizados sete *template patterns*: O *source, refmod_caller, sink, tb_single_refmod, trans, Makefile_single_refmod, tb_tcl.* Conforme já mencionado anteriormente, cada um deles é implementado na linguagem eTL (eTBc *Template Language*), funcionando como molde para geração de código compilável, seja SystemC, para o caso de VeriSC ou Systemverilog, para o caso de BVM.

Os *templates source* e *sink* são utilizados na geração do código dos *tesbench elements Source* e *Sink* respectivamente. O *refmod_caller* gera uma classe responsável por instanciar o Modelo de Referência, alimentá-lo com os estímulos gerados pelo *Source* e repassar os valores de saída para o *Sink*. O código, “moldado” pelo *template tb_single_refmod*, é responsável por instanciar e ligar cada um desses elementos. O *trans* é utilizado na criação de uma classe que contém todas as transações utilizadas no *testbench*, assim como funções aplicadas a elas. No arquivo *trans* serão definidos os estímulos utilizados para realização da simulação naquele ambiente. O *template Makefile_single_refmod* é utilizado na geração do arquivo de *Makefile* utilizado para compilar e simular o código gerado pela ferramenta eTBc. O tb_tcl é um molde para geração de um arquivo utilizado durante a simulação. Este arquivo tem como objetivo interromper a simulação quando a cobertura funcional é atingida.

Vale lembrar que, mesmo a ferramenta eTBc gerando código compilável dos *testbench elements*, os estímulos e a cobertura funcional desejada terão que ser inseridos manualmente. Para melhor entendimento de como funciona a geração de código compilável utilizando a ferramenta eTBc, segue uma breve explicação do *template pattern* do arquivo *trans* (Código 13).

**Código para t*emplate pattern do arquivo trans***

```
1         $$(type.map) $$(bool->bit)
2         $$(type.map) $$(char->byte)
3         $$(type.map) $$(short->shortint)
4         $$(type.map) $$(short int->shortint)
5         $$(type.map) $$(int->int)
6         $$(type.map) $$(long->int)
7         1$$(type.map) $$(float->shortreal)
8
9         ...
10
11        $$(type.map) $$(unsigned long long->longint)
12        $$(type.map) $$(unsigned long long int->longint)
13        $$(file) $$("trans.svh")//Code generated by eTBc software
14
15        import ovm_pkg::*;
16        $$(foreach) $$(struct)
17        class $$(i.name) extends ovm_transaction;
18          $$(foreach) $$(var)
19          rand $$(i.type) $$(i.name);
20          constraint $$(i.name)_range {
21           $$(i.name) dist { [0:1] };
22          }
23          $$(endfor)
24          function integer equal($$(i.name) tr);
25           equal = $$(foreach)$$(var)(this.$$(i.name) == tr.$$(i.name))
26        $$(if)$$(isnotlast) && $$(endif)$$(endfor);
27          endfunction
28
29          function string psprint();
30           psprint = $psprintf("($$(foreach)$$(var)$$(i.name)= %d)",$$
31        (i.name)$$(if)$$(isnotlast), $$(endif)$$(endfor));
32          endfunction
33
34          function bit read(int file);
35           if (file && !$feof(file))
36            read = 0 != $fscanf(file, "$$(foreach)$$(var)%d $$
37        (endfor)", $$(foreach)$$(var)this.$$(i.name)$$(if)$$(isnotlast), $
38        $(endif)$$(endfor));
39           else read = 0;
40          endfunction
41
42          function void do_record (ovm_recorder recorder);
43           $$(foreach) $$(var)recorder.record_field ("$$(i.name)", $$
44        (i.name), $bits($$(i.name)), OVM_HEX);$$(endfor)
45          endfunction
46
47        endclass
48        $$(endfor)
49
50        class delay; // used for delay in signal handshake
51          rand int d;
52          constraint delay_range {
53           d dist {0 :/ 8, [1:2] :/2, [3:10] :/1 };
54          }
55        endclass
56
57        $$(endfile)
```

**Código 13 – *Código para template pattern do arquivo trans.***

As linhas de 1 a 13 do *template pattern* referem-se ao mapeamento de tipos da linguagem eDL (a linguagem de descrição da TLN) para os tipos da linguagem SystemVerilog. Por exemplo, na linha 6, o código “$$(type.map) $$(long->int)” significa que a ferramenta deverá mapear o tipo “long” da linguagem eDL para o tipo “int”, no momento em que gera o código compilável em Systemverilog.

Na linha 16, o código “$$(foreach) $$(struct)” inicia um laço que varre cada estrutura contida na TLN. O laço só termina na linha 48, com o “$$(endfor)”. Esta repetição terá efeito no intervalo da linha 17 até a linha 47, tendo como resultado a criação de várias classes, que representam cada *struct* da TLN. Existem várias funções a partir da linha 24. Essas funções correspondem às funções que podem ser aplicadas a cada uma das estruturas descritas na TLN.

As linhas de 18 a 23 referem-se à criação da faixa de valores dos estímulos utilizados no *testbench* para uma dada estrutura. Essa faixa não está preenchida no *template*, visto que, a faixa de valores dos estímulos variam de acordo com a implementação de cada *refmod*.

A TLN a seguir (Código 14) representa a estrutura de um DPCM (*Differential Pulse-Code Modulation*) (SILVA et al., 2007). Ela é utilizada como entrada da ferramenta, juntamente com o *template* do arquivo *trans*. O DPCM tem como função básica converter um sinal analógico em um sinal digital. O sinal analógico é amostrado e a diferença entre o valor da atual de cada amostra e seu valor previsto (derivado de amostra(s) anterior(es)) é quantizado[11] e convertido, por codificação, para um sinal digital (WAGGENER, 1999). A simplicidade de implementação e modelagem de um DPCM o torna um bom exemplo para estudo. Ele pode ser dividido hierarquicamente em dois módulos: um que calcula a diferença entre amostras e outro que realiza uma quantização do resultado dessa operação. No estudo de caso, a quantização será uma simples saturação desse resultado (Figura 29).

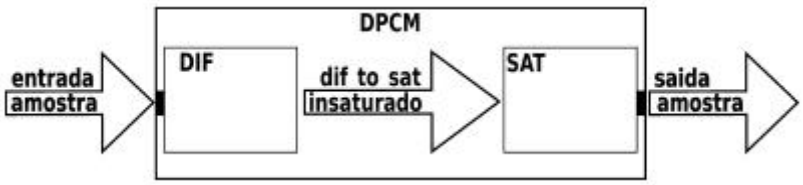


**Figura 29 – Representação de um DPCM (PESSOA, 2007, p. 27).**

Nas linhas 20 e 26, está caracterizada a divisão do DPCM em dois módulos, o dif e o sat. As únicas estruturas utilizadas nas transações do DPCM são a *sample* e a *unsatured*. No nível de transação, a *sample* pode ser considerado um número inteiro (linha 3). No nível de sinais, esta estrutura pode ser considerada um dado de 3 bits com sinal (linha 6). As linhas de 20 a 23

11 O processo de quantização consiste na atribuição de valores discretos para um sinal cuja amplitude varia entre infinitos valores.

compreendem a descrição do módulo de diferenciação. Este módulo tem como entrada uma estrutura *sample* e uma saída *unsatured* (linhas 21 e 22). Nas linhas de 32 a 38, são apresentados o DPCM como um todo, suas entradas e saídas, e como também a ligação entre os dois módulos através de um canal. Nas linhas 36 e 37, é descrito como estes dois módulos (dif e sat) estão hierarquicamente conectados. Perceba-se que a saída do dif, o "dif_out", está conectada com a entrada do sat, o "sat_in", através de um canal chamado "dif_to_sat".

**Código de uma dada TLN**

```
1         struct sample{
2          trans {
3           int s_value;
4          }
5          signals {
6           signed [3] s_value;
7          }
8         }
9
10        struct unsaturated {
11         trans {
12          int i_value;
13         }
14         signals {
15          signed [4] i_value;
16         }
17        }
18
19        //diff module
20        module dif{
21         input sample   dif_in;
22         output unsaturated dif_out;
23        }
24
25        //sat module
26        module sat{
27         input unsaturated sat_in;
28         output sample   sat_out;
29        }
30
31        //top-level module
32        module dpcm{
33         input sample in_dpcm;
34         output sample out_dpcm;
35         channel unsaturated dif_to_sat;
36         dif dif_i( .dif_in( in_dpcm )  , .dif_out( dif_to_sat ) );
37         sat sat_i( .sat_in( dif_to_sat) , .sat_out( out_dpcm  ) );
38        }
```

**Código 14 – Código de uma TLN exemplo.**

O código gerado pela ferramenta eTBc, dado o *template pattern trans* e a TLN exemplo, é apresentado a seguir (Código 15).

**Código gerado pela ferramenta eTBc utilizando TLN do DPCM e *template* do arquivo *trans***

```
1         //Code generated by eTBc software
2
3         import ovm_pkg::*;
4
5         class sample extends ovm_transaction;
6
7           rand int s_value;
8           constraint s_value_range {
9            s_value dist { [0:1] };
10          }
11
12          function integer equal(sample tr);
13           equal = (this.s_value == tr.s_value);
14          endfunction
15
16          function string psprint();
17           psprint = $psprintf("(s_value= %d)",s_value);
18          endfunction
19
20          function bit read(int file);
21           if (file && !$feof(file))
22            read = 0 != $fscanf(file, "%d ", this.s_value);
23           else read = 0;
24          endfunction
25
26          function void do_record (ovm_recorder recorder);
27            recorder.record_field ("s_value", s_value, $bits(s_value),
28        OVM_HEX);
29          endfunction
30
31        endclass
32
33        class unsaturated extends ovm_transaction;
34
35          rand int i_value;
36          constraint i_value_range {
37           i_value dist { [0:1] };
38          }
39
40          function integer equal(unsaturated tr);
41           equal = (this.i_value == tr.i_value);
42          endfunction
43
44          function string psprint();
45           psprint = $psprintf("(i_value= %d)",i_value);
46          endfunction
47
48          function bit read(int file);
49           if (file && !$feof(file))
50            read = 0 != $fscanf(file, "%d ", this.i_value);
51           else read = 0;
52          endfunction
53
54          function void do_record (ovm_recorder recorder);
55            recorder.record_field ("i_value", i_value, $bits(i_value),
56        OVM_HEX);
57          endfunction
```

**Código gerado pela ferramenta eTBc utilizando TLN do DPCM e *template* do arquivo *trans* (Continuação)**

```
58
59        endclass
60
61        class delay; // used for delay in signal handshake
62          rand int d;
63          constraint delay_range {
64           d dist {0 :/ 8, [1:2] :/2, [3:10] :/1 };
65          }
66        endclass
```

**Código 15 – Código gerado pela ferramenta eTBc utilizando TLN do DPCM e template do arquivo *trans*.**

Foram geradas três classes a partir do *template trans*. A classe *sample*, refere-se à estrutura *sample*, descrita nas linhas de 1 a 8 da TLN. De maneira análoga, a classe *unsatured* foi gerada. O código das linhas 7 a 10 e 35 a 38 do arquivo gerado corresponde às linhas 18 a 23 do *template* do arquivo *trans* (Código 13). Os valores pseudo-randômicos gerados por ambas as classes são do tipo inteiro, visto que o tipo de valores definidos para transações em ambas *structs* são do tipo inteiro. Se, por exemplo, na TLN, a *struct sample* tivesse sua parte "*trans*" definida como "float s_value", em vez de "int s_value", a classe *sample* gerada criaria estímulos pseudo-randômicos do tipo "shortreal", já que no *template* do arquivo *trans*, variáveis do tipo "float" definidas na TLN são mapeadas para o tipo "shortreal" da linguagem SystemVerilog.

O campo responsável pela faixa de valores pseudo-randômicos do arquivo *trans* deverá ser preenchido pelo engenheiro de verificação (linha 9 e linha 37). Por *default,* as classes sample e *unsatured* geram valores zeros e uns. Para rápido entendimento de geração de valores pseudo-randômicos, consultar o livro *SystemVerilog for verification* (SPEAR, 2006), Seção 6.4.6 *Weighted distribuitions*.

Este foi apenas um exemplo de como foram desenvolvidos os *template patterns* para uso na ferramenta eTBc. Não cabe neste trabalho explicar a sintaxe e semântica de cada linha de todos os *template patterns* desenvolvidos. As próximas subseções darão apenas uma visão geral da função de cada um deles na criação do *testbench*. O código fonte de cada um pode ser visto em detalhes no Apêndice A.

#### *4.1.1.2 Double refmod*

Assim como no *Single refmod*, a construção do *testbench* deste subpasso utiliza sete *template patterns*: O *source, refmod_caller, checker, tb_double_refmod, trans, Makefile_double_refmod e tb_tcl.* O *checker* substitui o *sink* do subpasso anterior. A estrutura do *testbench* desse subpasso é diferente do passo *Single refmod.* Por isto, o *template* que ligava os elementos do *testbench* também será substituído por outro: o *tb_double_refmod* (Apêndice A,

código A.19). A estrutura interna do *Makefile* contém comandos para compilar e os arquivos e simular o *testbench*. Como a nomenclatura dos arquivos e a estrutura do *testbench* mudaram, o arquivo de *Makefile* também deve ser substituído. A seguir, a estrutura do *template* do *Makefile* referente a este passo (Código 16):

**Código do *template Makefile_double_refmod***

```
1         $$(file) $$("Makefile")#Code generated by eTBc software
2
3         #Variables
4         MODULE = $$(module.name)
5         OVM = -ovm
6         SCRIPT = -input tb.tcl
7         ACCESS = -access +r
8         COVER = -coverage all
9         TB = tb.sv
10        RM = refmod_$(MODULE).svh $(MODULE).svh
11        C = $(MODULE).c
12
13        #Targets
14        cov_work : .tb_cpld tb.tcl
15             rm -rf cov_work; irun $(OVM) $(SCRIPT) $(ACCESS) $(COVER) $
16        (TB)
17
18        .tb_cpld : trans.svh source.svh checker.svh definitions.svh $(RM)
19        $(TB)
20             irun $(OVM) -compile $(TB) && touch .tb_cpld
21
22        run :
23             rm -rf cov_work; $(MAKE)
24
25        #Cleans
26        clean :
27             rm -rf cov_work cov.cmd INCA_libs waves.shm *.log *.key *.txt
28        .tb_cpld .simvision *~ *#
29
30        $$(endfile)
```

**Código 16 – Código do *template Makefile_double_refmod*.**

Em um terminal Linux, com o caminho da biblioteca OVM devidamente definido e a ferramenta IUS 8.2 da Cadence devidamente instalada, o comando "*make cov_work*" (linha 14) compila e simula todo o ambiente do *testbench*. O comando "*make clean*" (linha 26) apaga os arquivos criados durante a simulação.

Este subpasso utiliza o mesmo *source* do subpasso anterior. Porém, para alimentar os dois modelos de referência, o *tb_tb_double_refmod* instancia e conecta a estrutura que representa a FIFO de duas saídas na porta de saída deste *source*. O *template* do *checker* não difere muita coisa do *sink*. A diferença é que o *sink* coleta apenas a saída de um modelo de referência, enquanto o *checker* coleta de dois modelos, comparando-as. A seguir, o código referente ao *template pattern* de ambos elementos (Código 17 e Código 18).

**Código do *template pattern source***

```
1         ...
2         $$(file) $$("source.svh")
3
4         class source extends ovm_component;
5
6          $$(foreach) $$(module.in)
7           ovm_put_port #($$(i.type)) $$(i.name)_to_refmod_port;
8          $$(endfor)
9           function new(string name, ovm_component parent);
10           super.new(name,parent);
11          $$(foreach) $$(module.in)
12           $$(i.name)_to_refmod_port = new("$$(i.name)_to_refmod_port",
13        this);
14          $$(endfor)
15          endfunction
16          task run();
17           $$(foreach) $$(module.in)
18            $$(i.type) tr_$$(i.name);
19           $$(endfor)
20            int file = $fopen("tr.sti", "r");
21            recording_detail = OVM_FULL;
22            forever begin
23             $$(foreach) $$(module.in)
24              tr_$$(i.name) = new();
25              if(!tr_$$(i.name).read(file))
26                assert(tr_$$(i.name).randomize());
27              assert(begin_tr(tr_$$(i.name), "source", "tr_source"));
28              $$(i.name)_to_refmod_port.put(tr_$$(i.name));
29             $$(endfor)
30              #20ns;
31              end_tr(tr_$$(i.name));
32            end
33          endtask
34
35        endclass
36
37        $$(endfile)
```

**Código 17 – Código do *template pattern source.***

Na linha 7 do *template pattern source* é instanciada a porta de conexão deste elemento com o modelo de referência. Tal conexão é do tipo "*ovm_put_port*", já que é uma porta de saída do *source*.

**Código do *template pattern checker***

```
1         ...
2         $$(file) $$("checker.svh")
3
4         class checker extends ovm_threaded_component;
5
6           $$(foreach) $$(module.out)
7           ovm_get_port #($$(i.type)) $$(i.name)_from_refmod_port;
8           ovm_get_port #($$(i.type)) $$(i.name)_from_duv_port;
9           $$(endfor)
10          function new(string name, ovm_component parent);
```

**Código do *template pattern checker* (Continuação)**

```
11          super.new(name, parent);
12          $$(foreach) $$(module.out)
13          $$(i.name)_from_refmod_port = new("$$
14       (i.name)_from_refmod_port", this);
15          $$(i.name)_from_duv_port = new("$$(i.name)_from_duv_port",
16       this);
17          $$(endfor)
18        endfunction
19
20        task run();
21         $$(foreach) $$(module.out)
22         $$(i.type) tr_duv_$$(i.name),tr_modref_$$(i.name);
23         $$(endfor)
24         string msg;
25
26         recording_detail = OVM_FULL;
27
28         forever begin
29          $$(foreach) $$(module.out)
30          $$(i.name)_from_duv_port.get(tr_duv_$$(i.name));
31          $$(i.name)_from_refmod_port.get(tr_modref_$$(i.name));
32          if(!tr_duv_$$(i.name).equal(tr_modref_$$(i.name))) begin
33           msg = $psprintf("received: %s expected %s", tr_duv_$$
34       (i.name).psprint(), tr_modref_$$(i.name).psprint());
35           ovm_report_error("Checker", msg);
36           assert(record_error_tr("checker"));
37           100ns;
38           global_stop_request();
39          end
40          $$(endfor)
41         end
42        endtask
43       endclass
44
45       $$(endfile)
```

**Código 18 – Código do *template pattern checker.***

Da mesma maneira que o *source*, o *template* do *checker* (Código 18), nas linhas 7 e 8 são instanciadas as portas de conexão desse elemento com os modelos de referência. Estas conexões são do tipo "*ovm_get_port*", pois são portas de entrada do *checker.* No *template* do *sink,* não existe a parte relativa à comparação dos valores de saída dos modelos de referência (linhas 32 a 39), sendo necessária apenas a declaração de uma porta.

#### *4.1.1.3 DUV emulation*

Neste subpasso, são utilizados dezessete *template patterns*: O *source, refmod_caller, checker, tdriver, tdriver_duv, tmonitor, tmonitor_duv, in_actor, out_actor, tb_duv_emulation, trans, duv_emulation, top, Makefile_duv_emulation, top_tcl, axi_cover, e gene_clock..* Os arquivos gerados por meio dos *template pattern* do *source, refmod_caller e checker* podem ser

aproveitados do subpasso anterior. Em BVM, o reúso interno de elementos de *testbench* de um determinado projeto*,* diminui o tempo gasto no desenvolvimento de novos *tesbench*. Porém, cabe ao engenheiro de verificação perceber quais elementos podem ser reaproveitados dos *tesbench* existentes.

O código gerado a partir do *tdriver_duv* e o *tmonitor_duv* serão utilizados juntamente com uma instância do Modelo de Referência, criando dessa maneira a tripla que emulará um DUV já mencionada na Seção 3.2.1.3. Essa tripla será representada pelo código gerado a partir do *template* duv_emulation. Os *templates in_actor* e *out_actor* são utilizados para gerar o código dos *Actors* entre o *tdriver* e o DUV emulado e entre o DUV emulado e o *tmonitor* respectivamente.

Os sinais do protocolo de comunicação entre estes módulos são controlados pelos *Actors.* Estes, por sua vez, repassam tais sinais para um módulo denominado *AXI_cover*. Este módulo é gerado, a partir do *template axi_cover,* sendo responsável por realizar a cobertura do protocolo de comunicação entre *tdriver* e o DUV emulado e entre o DUV emulado e o *tmonitor*.

A implementação de VeriSC não possui um protocolo de comunicação padrão para a comunicação dos elementos *TDriver* e *TMonitor* com o DUV. Este protocolo é implementado pelo engenheiro de verificação que utilize esta metodologia. De maneira oposta, BVM possui um protocolo de comunicação padrão já introduzido nos *template patterns* dos elementos que se comunicam diretamente com o DUV: os *Driver* e os *Actor.* O protocolo implementado é baseado na especificação do protocolo AMBA AXI (ARM, 2010).

O *gene_clock* nada mais é do que um molde para a geração de um módulo responsável pelo *clock* do sistema. Os demais arquivos já foram mencionados anteriormente, tendo as mesmas funções já descritas, com a diferença de que são específicos para o subpasso DUV emulation.

## *4.1.2 Hierarchical refmod decomposition*

### *4.1.2.1 Refmod decomposition*

Este subpasso consiste em dividir o Modelo de Referência hierarquicamente em blocos menores, os quais devem ser equivalentes à decomposição hierárquica desejada para o DUV. O engenheiro de verificação terá que dividir o Modelo de Referência de maneira manual. Esta tarefa não é trivial, visto que na maioria das vezes não é tão fácil separar as funcionalidades espalhadas no código fonte do Modelo de Referência.

### *4.1.2.2 Single hierarchical refmods*

Aqui são utilizados sete *template patterns*: O *source, refmod_caller, sink,*

*tb_single_refmod, trans, Makefile_single_refmod, tb_tcl.* Os *templates* utilizados neste subpasso são os mesmos utilizados no subpasso *Single refmod*. A diferença é que o *testbench* gerado será aplicado aos modelos de referência resultantes do subpasso anterior, o *Refmod decomposition*.

#### *4.1.2.3 Hierarchical refmods verification*

Neste subpasso, os modelos de referência resultantes do subpasso *Refmod decomposition* serão unidos para verificar se sua união mantém a funcionalidade provida pelo Modelo de Referência. Os *templates* utilizados para a construção do *testbench* desse subpasso são : *source, refmod_caller, checker, trans, tb_hierachical_refmod, Makefile_hierarchical_refmod, tb_tcl.* Vale lembrar que este subpasso realiza operações apenas no nível de transação, visto que não se faz uso dos elementos do *testbench Driver* nem do *Monitor*.

### *4.1.3 Hierarchical testbench*

#### *4.1.3.1 Double hierarchical refmods*

De maneira análoga ao subpasso *Doube refmod*, a construção do *testbench* deste subpasso também utiliza seus mesmos *template patterns*: O *source, refmod_caller, checker, tb_double_refmod, trans, Makefile_double_refmod e tb_tcl.* A diferença é que este subpasso é voltado para testar se o *Source* e o Checker estão realizando seu papel corretamente em relação aos modelos de referência resultantes do subpasso *Refmod decomposition.*

#### *4.1.3.2 Hierarchical DUV emulation*

Neste subpasso, os mesmos dezessete *template patterns* utilizados no subpasso *DUV emulation* serão usados: O *source, refmod_caller, checker, tdriver, tdriver_duv, tmonitor, tmonitor_duv, in_actor, out_actor, tb_duv_emulation, trans, duv_emulation, top, Makefile_duv_emulation, top_tcl, axi_cover, e gene_clock.* Porém, o *testbench* criado será aplicado aos modelos de referência resultantes do subpasso *Refmod decomposition*, em vez de ser aplicado ao Modelo de Referência completo.

#### *4.1.3.3 Hierarchical DUV*

Este supasso é semelhante ao passo anterior, porém neste passo os DUV emulados serão substituídos por DUV reais. Vale lembrar que é a partir desse subpasso que os DUV reais serão utilizados. Os *template patterns* que compõem esse *testbench* são: *source, refmod_caller, checker, tdriver, tmonitor, out_actor, tb_duv, trans, duv_hierarchical, top, Makefile_duv_emulation, top_tcl, axi_cover, e gene_clock.*

### 4.1.4 Full testbench

O *Full testbench* consiste no penúltimo passo da metodologia BVM. Todos os DUV utilizados no passo anterior agora serão unidos e testados, verificando se esta união reflete a mesma funcionalidade do Modelo de Referência. É importante lembrar que mesmo se todos os outros passos da metodologia funcionaram de maneira adequada, não significa que este passo estará livre de erros. Conforme já mencionado na Seção 3.3, erros de protocolo de comunicação entre as interfaces desses elementos podem acontecer. Por isto, é importante que o engenheiro de verificação esteja atento quanto ao uso do elemento *Actor* na detecção de tais erros. Os *templates* que compõem o *testbench* desse subpasso são: *source, refmod_caller, checker, tdriver, tmonitor, out_actor, tb_duv, trans, top, Makefile_duv, top_tcl, axi_cover, e gene_clock.*

## 4.2 Full testbench analysis

O *Full testbench analysis* é o último passo da metodologia BVM e visa a detecção de erros na etapa de integração dos DUV. Diferentemente do passo *Full testbench*, que utiliza o Modelo de Referência original na criação de seu *testbench*, este subpasso utiliza os Modelos de Referência obtidos no subpasso *Refmod decomposition*. Esta estrutura de *testbench* torna possível o uso da FIFO de duas saídas entre cada par de Modelos. No *Full testbench analysis,* os monitores implementados ao longo dos passos da metodologia BVM são inseridos entre todos os pares de DUV. Os *templates* que compõem o *testbench* deste subpasso são: *source, refmod_caller, checker, tdriver, tmonitor, out_actor, tb_duv_analysis, trans, top_analysis, Makefile_duv_analysis, top_tcl, axi_cover, e gene_clock.*

## 4.3 Estudo comparativo preliminar entre BVM e VeriSC

Em 2007, o Brazil-IP ofereceu treinamento a um grupo de 16 alunos de graduação com o objetivo de proporcionar o aprendizado da metodologia VeriSC. Os alunos foram orientados a desenvolver um pequeno projeto utilizando a referida metodologia. Para todos esses alunos, este foi o primeiro contato com uma metodologia de verificação funcional. O projeto consistiu em um DPCM (*Differential Pulse Code Modulation*). DPCM codifica sinais digitais, calculando as diferenças entre as amostras subseqüentes e saturando-as. A implementação DPCM é hierarquicamente dividida em um módulo de diferença e um de saturação. O projeto consistiu na criação *testbench*, implementação do RTL e verificação funcional. Ao final do treinamento, um questionário foi aplicado, para saber quão boa VeriSC foi para eles. Da mesma forma, no final do ano de 2008, o Brazil-IP ofereceu um outro treinamento, mas com a metodologia BVM. Um grupo diferente de 19 alunos de graduação foi orientado a fazer o mesmo projeto utilizado em 2007, porém aplicando BVM. Ao final do treinamento, o mesmo questionário foi utilizado para coletar a

opinião desses estudantes. BVM foi o primeiro contato com uma metodologia de verificação funcional para este grupo também. Assim, as respostas coletadas de todos esses usuários foram filtradas e analisadas para fazer uma comparação preliminar entre as duas metodologias. Em 2008, BVM estava em sua fase inicial de desenvolvimento e ainda não possuía em sua implementação o mecanismo para detecção de erros na etapa de integração dos DUV. Porém, BVM apresentava-se suficiente madura para realização de um estudo comparativo preliminar em relação à VeriSC. Este trabalho resultou em um artigo apresentado no Apêndice B. As questões aplicadas são apresentadas no Quadro 2.

**Quadro 2 – Questões sobre BVM e VeriSC.**

| Questões |
|---|
| 1. Como você classifica a metodologia (muito ruim, ruim, boa, muito boa, excelente)? |
| 2. A metodologia é fácil de entender(sim, não) ? |
| 3. A metodologia é fácil de aplicar(sim, não) ? |
| 4. A linguagem usada por esta metodologia foi fácil de entender?(sim, não) |
| 5. A linguagem usada por esta metodologia foi fácil de aplicar?(sim, não) |
| 6. Quão útil você classifica o eTBc para construir *testbenches* semi-automáticos para esta metodologia (muito ruim, ruim, boa, muito boa, excelente)? |
| 7. Cite a pior coisa sobre esta metodologia |

As respostas ao questionário indicaram que os estudantes poderiam facilmente se adaptar a BVM ou VeriSC. Nas Tabelas 1 a 7, são apresentadas as respostas a cada uma das perguntas do questionário.

**Tabela 1 – Respostas sobre a questão 1.**

| Questão 1: Como você classifica a metodologia (muito ruim, ruim, boa, muito boa, excelente)?? | | | | | | |
|---|---|---|---|---|---|---|
| | Muito ruim | Ruim | Boa | Muito boa | Excelente | Não respondeu |
| Estudantes de VeriSC | 0% | 0% | 87,5% (14 estudantes) | 6,25% (1 estudantes) | 0% | 6,25% (1 estudante) |
| Estudantes de BVM | 0% | 0% | 68,42% (13 estudantes) | 10,53% (2 estudantes) | 0% | 21,05% (4 estudantes) |

Como pode ser visto na Tabela 1, em se tratando das respostas à primeira questão, “a opinião dos estudantes sobre as metodologias” indicam que a maioria as classificam como boa e muito boa.

**Tabela 2 – Respostas sobre a questão 2.**

| Questão 2: A metodologia é fácil de entender(sim, não)? | | | |
|---|---|---|---|
| | Sim | Não | Não respondeu |
| Estudantes de VeriSC | 93,75% (15 estudantes) | 0% | 6,25% (1 estudante) |
| Estudantes de BVM | 100% (19 estudantes) | 0% | 0% |

**Tabela 3 – Respostas sobre a questão 3.**

| Questão 3: A metodologia é fácil de aplicar(sim, não)? | | | |
|---|---|---|---|
| | Sim | Não | Não respondeu |
| Estudantes de VeriSC | 68,75% (11 estudantes) | 31,25% (5 estudantes) | 0% |
| Estudantes de BVM | 84,21% (16 estudantes) | 15,79% (3 estudantes) | 0% |

Para quase todos os estudantes, a metodologia estudada (VeriSC ou BVM) é de fácil entendimento (Tabela 2). Porém, de acordo com a Tabela 3, algumas diferenças, relacionadas à facilidade de aplicação destas metodologias, são visíveis. Para os estudantes, BVM parece mais fácil de ser aplicada que VeriSC.

**Tabela 4 – Respostas sobre a questão 4.**

| Questão 4: A linguagem usada por esta metodologia foi fácil de entender?(sim, não) | | | |
|---|---|---|---|
| | Sim | Não | Não respondeu |
| Estudantes de VeriSC | 75% (12 estudantes) | 18,75% (3 estudantes) | 6,25% (1 estudante) |
| Estudantes de BVM | 89,47% (17 estudantes) | 10,53% (2 estudantes) | 0% |

Na Tabela 4, são apresentadas algumas diferenças entre as linguagens utilizadas para criar *testbenches* em VeriSC e BVM. VeriSC é baseada em SystemC e BVM baseada em SystemVerilog. De acordo com a Tabela 4, é correto afirmar que 14,47% (89,47% - 75%) mais estudantes acham que SystemVerilog é fácil de entender.

**Tabela 5 – Respostas sobre a questão 5.**

| Questão 5: A linguagem usada por esta metodologia foi fácil de aplicar?(sim, não) | | | |
|---|---|---|---|
| | Sim | Não | Não respondeu |
| Estudantes de VeriSC | 75% (12 estudantes) | 25% (4 estudantes) | 0% |
| Estudantes de BVM | 84,21% (16 estudantes) | 15,79% (3 estudantes) | 0% |

As respostas apresentadas na Tabela 7 são muito similares às respostas apresentadas na Tabela 5. Isto indica que a facilidade de aplicar a metodologia está associada com a linguagem usada para criar o seu *testbench*.

**Tabela 6 – Respostas sobre a questão 6.**

| Questão 6: Quão útil você classifica o eTBc para construir *testbenches* semi-automáticos para esta metodologia (muito ruim, ruim, boa, muito boa, excelente)? | | | | | | |
|---|---|---|---|---|---|---|
| | Muito ruim | Ruim | Boa | Muito boa | Excelente | Não respondeu |
| Estudantes de VeriSC | 0% | 0% | 93,75% (15 estudantes) | 6,25% (1 estudantes) | 0% | 0% |
| Estudantes de BVM | 0% | 0% | 47,37% (9 estudantes) | 42,1% (8 estudantes) | 10,53% (2 estudantes) | 0% |

De acordo com a Tabela 6, para todos os estudantes, a ferramenta eTBc foi considerada muito útil para criar elementos do *testbench* de maneira semi-automática. A ferramenta eTBc agilizou o processo de criação *testbench*.

**Tabela 7 – Respostas sobre a questão 7.**

| Questão 7: Cite a pior coisa sobre esta metodologia | | | | |
|---|---|---|---|---|
| | A linguagem usada | A metodologia tem muitos passos | Outra | Não respondeu |
| Estudantes de VeriSC | 25% (4 students) | 62,5% ( 10 estudantes) | 12,5% (4 estudantes) | 0% |
| Estudantes de BVM | 0% | 84,2% ( 16 estudantes) | 15,78% ( 3 estudantes) | 0% |

Para a última pergunta, muitos estudantes indicaram que a metodologia VeriSC e BVM têm muitos passos para realização da verificação funcional (Tabela 7). Para os estudantes de VeriSC, 25% acham que a linguagem usada para criar o *testbench* é a pior aspecto da metodologia. Os estudantes de BVM ficaram satisfeitos com a linguagem SystemVerilog. O tempo gasto para

concluir o projeto usando BVM foi de cerca de 20% menor do que utilizando a metodologia VeriSC. Na formação com VeriSC, os estudantes necessitaram de cerca de 28 horas para concluir o projeto. Os estudantes BVM necessitaram de cerca de 22 horas para terminá-lo.

Em Setembro de 2009, as equipes do Brazil-IP participaram do *Symposium on Integrated Circuits and Systems Design* (SBCCI, 2010). Neste período a metodologia BVM já apresentava-se finalizada. O SBCCI é um fórum internacional, que ocorre anualmente no Brasil, dedicado a circuitos integrados e design de sistemas, testes e CAD. Atualmente 17 universidades participam do programa Brazil-IP: UFCG, UFPE, USP, UNICAMP, UEFS, UNB, UFC, UFPA, UFPB, UFRN, UFS, UFSC, UFSM, UNESP, UNIFEI, UNIPAMPA e UNIVALI. Quatorze equipes do Brazil-IP apresentaram protótipos funcionais de IP *cores* no evento. Todos estes foram desenvolvidos utilizando processo de desenvolvimento de IP *cores* ipPROCESS e verificados utilizando a metodologia BVM. Dentre elas, apenas duas não conseguiram apresentar o protótipo do IP *core* em tempo. Não foi feita uma avaliação profunda quanto ao fato de esses dois protótipos não terem sido desenvolvidos em tempo hábil para apresentação no evento. Porém, não houve registro de ocorrências de problemas quanto ao uso da metodologia BVM. Todas as equipes dispunham de fácil acesso para elucidar dúvidas específicas em relação à metodologia.

Uma contribuição deste trabalho foi a criação do elemento *Actor*, para controlar e monitorar o protocolo de comunicação entre o DUV e o *tesbench*. Tal criação tornou os Monitores elementos passivos nos *testbench* da metodologia BVM. Em contraposição, os Monitores de VeriSC detinham o controle sobre esse protocolo de comunicação, não permitindo a inserção destes elementos entre pares de DUV na etapa de integração dos DUV.

Outra contribuição, talvez a principal delas, foi a criação de um passo, inexistente em VeriSC, para a detecção de erros na etapa de integração dos DUV. Conforme mencionado no decorrer deste trabalho, a criação da FIFO de duas saídas realizou um papel importante em BVM. Além de simplificar a comunicação entre elementos que compõe o *testbench*, o uso desta FIFO juntamente com Monitores passivos viabilizou o mecanismo para detecção de erros presente no novo passo. O *testbench* apresentado neste novo passo encontra, de forma eficiente, possíveis erros de comunicação entre as interfaces de qualquer par de DUV, tornando BVM uma metodologia mais poderosa do que VeriSC.

Em relação à implementação, pode-se citar a facilidade de uso de BVM, dado fato que foi implementada em SystemVerilog e OVM. Outro aspecto é que BVM implementa de maneira nativa um protocolo de comunicação padrão para os elementos que comunicam-se diretamente com o DUV. Em VeriSC não existe um protocolo de comunicação implementado para esta comunicação, ou seja, o engenheiro de verificação deverá implementá-lo manualmente. O uso de AMBA AXI em BVM padroniza a comunicação dos *Driver* e dos *Actor* com o DUV.

Outro resultado obtido com o trabalho foi o aprimoramento do core da ferramenta eTBc, assim como a detecção de erros na ferramenta IUS da Cadence (CADENCE DESIGN SYSTEMS, 2010). Tal ferramenta foi utilizada na realização simulação durante a fase de verificação funcional. Em relação à ferramenta eTBc, falhas eram reportadas para que pudessem ser corrigidas. O uso massivo da ferramenta pelas equipes do Brazil-IP, deram uma maior robustez a mesma, submetendo-a a vários tipos de testes. Além disto, para dar suporte à metodologia BVM, a ferramenta também teve que ser adaptada. Recursos foram adicionados tanto na linguagem eDL como eTL, para viabilizar a construção dos *testbench elements* baseados em SystemVerilog e OVM. Em relação aos erros detectados na ferramenta IUS, estes foram reportados e rapidamente corrigidos pela equipe de desenvolvedores da ferramenta.

# Capítulo 5

# 5 Considerações finais e sugestões para trabalhos futuros

Os resultados obtidos mostraram que BVM apresentou um aumento de 20% na produtividade, em comparação à metodologia VeriSC. Com base nos questionários, foi possível observar que BVM é mais fácil de aplicar e compreender, em comparação à VeriSC. Destaca-se, também, que a linguagem utilizada para criar *testbenches* em VeriSC pode ser considerada um obstáculo.

São várias as vantagens de se utilizar a linguagem SystemVerilog na implementação de BVM. Por exemplo, a facilidade e os recursos que SystemVerilog oferece em relação à SystemC para verificação funcional, a torna mais atraente para engenheiros de verificação. Outra vantagem é que SystemVerilog abrange modelagem em nível de sistema, RTL e verificação, simplificando a realização de projetos *top-down.* É possível criar modelos em nível de sistema e então refinar cada bloco para um nível mais baixo. Desta maneira, os modelos originais em nível de sistema podem servir como modelos de referência. Estas são algumas características que podem ser apontadas como vantagens de SystemVerilog em relação a SystemC. A seção 2.7 pode ser consultada para detalhes.

Todos os alunos consideraram eTBc uma ferramenta útil para geração de *testbenches* de maneira semi-automática, para as duas metodologias. A ferramenta eTBc facilitou muito a criação de *testbenches*. Se os alunos de ambas as metodologias não tivessem usado a ferramenta eTBc, os resultados apresentados nas tabelas seriam, provavelmente, muito diferentes. A criação semi-automática de *testbenches* “mascara” algumas das dificuldades para criação de um *testbench* em VeriSC.

A linguagem utilizada para a metodologia VeriSC, a biblioteca para realização da cobertura funcional e a forma como os componentes de *testbench* são conectados nesta metodologia é muito mais complexa do que BVM. BVM se mostrou de mais fácil entendimento e de aplicação. Considerando estes aspectos, BVM reduz o tempo, risco e recursos da verificação funcional. Além disto, as mudanças estruturais nos *testbench* de BVM, com a criação do elemento *Actor*, FIFO de duas saídas e de um mecanismo para detecção de erros na etapa de integração do DUV, torna esta metodologia mais poderosa do que VeriSC. Tais mudanças também facilitam o trabalho do engenheiro de verificação na detecção de erros durante o processo de verificação.

Todos esses fatos podem ser validados a partir do sucesso da metodologia BVM na verificação dos protótipos dos IP *cores* produzidos pelas equipes do Brazil-IP em 2009: uma CPU com suporte a IEEE 1149.1 e *On-Chip Debug*, um controlador programável de motor de passo, um compressor sem perdas de sinais biológicos e imagens médicas, um decodificador de áudio

MPEG-2 AAC-LC, um módulo transmissor HDMI, um módulo de processamento para a filtragem digital de imagens em tempo real, uma unidade aritmética em ponto flutuante padrão IEEE-754, um leitor de memória, um cliente USB 2.0, dentre outros.

## *5.1 Sugestões para trabalhos futuros*

Um estudo comparativo entre a metodologia OVM e BVM poderia ser realizado, visto que a implementação de ambas é baseada na linguagem SystemVerilog. Mediante estudo de casos, poderia ser feita uma pesquisa abordando o quão eficiente é cada uma destas metodologias, em relação ao tempo e aplicabilidade na verificação funcional de projetos de circuitos digitais. Tais projetos poderiam abranger desde os mais simples até os mais complexos, de interfaces permitindo assim uma melhor análise do comportamento e aplicabilidade dessas metodologias em cada um destes projetos.

Também poderia ser feita uma análise mais abrangente da utilização de BVM em outros projetos, com acompanhamento efetivo e aplicação de questionários, no intuito de avaliar sua eficiência como também descobrir suas possíveis deficiências. Um trabalho como este poderia resultar em melhorias da metodologia de verificação funcional BVM.

# Referências bibliográficas


ACCELLERA. Disponível em: <http://www.accellera.org>. Acesso em: 12 de julho de 2010.

ARM. *AMBA AXI Protocol Specification.* Version 2.0. 2010.

ARM AND SYNOPSYS. Disponível em: <http://www.vmm-sv.org/>. Acesso em: 12 de julho de 2010.

AUGUST, D.; CHANG, J.; GIRBAL, S.; PEREZ, D.; MOUCHARD, G.; PENRY, D.; TEMAM, O.; VACHHARAJANI, N. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. IEEE Computer Architecture Letters, 2007.

BERGERON, J.; CERNY, E.; HUNTER, A.; NIGHTINGALE, A. *Verification Methodology Manual for SystemVerilog*. First Edition. Synopsys, Inc., EUA, 2005.

BERGERON, J. *Writing Testbenches using SystemVerilog*. Springer, USA, 2006.

BERGERON, J. *Writing Testbenches: Functional Verification of HDL Models.* Second Edition. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

BRUNELLI, M.; BATTÚ, L.; CASTELNUOVO, A.; SFORZA, F. Functional verification of an hw block using vera. Synopys Users Group, 2001.

CADENCE DESIGN SYSTEMS. Disponível em: <http://www.cadence.com>. Acesso em: 12 de julho de 2010.

CADENCE DESIGN SYSTEMS; METOR GRAPHICS. *Open Verification Methodology Class Reference.* Version 2.0.3. USA, 2009a.

CADENCE DESIGN SYSTEMS; METOR GRAPHICS. *Open Verification Methodology User Guide.* Version 2.0.3. USA, 2009b.

DUEÑAS, C. A. Verification and test challenges in soc designs. Invited Talk, September 2004.

GLASSER, M. *Open en Verification Methodology Cookbook.* Springer. USA, 2009.

GLASSER, M. ; ROSE, A.; FITZPATRICK, T.; RICH, D.; FOSTER, H. *Advanced Verification Methodology Cookbook.* Mentor Graphics. 2007.

GOERING, R. Disponível em: <http://www.eetimes.com/story/OEG20010604S0113>. Acesso em: 12 de julho de 2010.

IEEE STANDARDS; *IEEE 1666-2005 Standard SystemC Language Reference Manual*, USA 2005 a.

IEEE STANDARDS. *IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language.* Institute of Electrical and Electronics Engineers, Inc., USA, 2005 b.

MENTOR GRAPHICS. Disponível em: <http://www.mentor.com>. Acesso em: 12 de julho de 2010a.

MENTOR GRAPHICS. Mentor Graphics Attacks Verification Bottlenecks With New Questa Verification Products Supports SystemVerilog, VHDL, PSL, and SystemC. Disponível em: http://www.mentor.com/products/fv/news/questa_launch. Acesso em: 12 de julho de 2010b.

MENTOR GRAPHICS. Mentor Graphics Delivers the Next Generation of Functional Verification. Disponível em: http://www.mentor.com/products/fv/news/questa_avm. Acesso em: 12 de julho 2010c.

MELCHER, E. U. K.; ROCHA, A. K.; LIRA, P.; YUN JU, Y.; BARROS, E. Silicon validated IP cores designed by the Brazil-IP Network. In: IP-SOC 2006, 2006, Grenoble. IP Based SoC Design Conference & Exhibition, 2006. p. 437-441.

MINTZ, M.; EKENDAHL, R. *Hardware Verification with SystemVerilog*. Massachusetts, EUA: Springer, 2007. 313 f.

MOLINA, A.; CADENAS, O. Functional verification: aproaches and challenges. Em: *Latin American Applied Research*, 2007.

OPENVERA. Disponível em: <http://www.open-vera.com/>. Acesso em: 12 de julho de 2010.

OVM. Disponível em: <http://www.ovmworld.org/>. Acesso em: 12 de julho de 2010.

PESSOA, I. M. *Geração semi-automática de Testbenches para circuitos integrados*. 2007. 52 f. Dissertação - Universidade Federal de Campina Grande, Campina Grande, PB, 2007.

PESSOA, I. M. ; SILVA, K. R. G. ; MELCHER, E. U. K. ; SILVA, L. M. L. ; CAMARA, R. C. P. ; NASCIMENTO NETA, M. L. ; MELO, F. G. L. ; OLIVEIRA, H. F. A. ; RODRIGUES, M. B. E. . eTBc: A Semi-Automatic Testbench Generation Tool. Em: IP 2007 Conference, 2007, Grenoble. Lecture Notes in Computer Science, 2007.

ROCHA, A. K. O. *DigiSeal Um Estudo de Caso para Modelagem de Transações Temporais Assíncronas na Metodologia VeriSC*. Paraíba: UFCG, 2008. 99 p. Dissertação (Mestrado) – Ciência da Computação, Universidade Federal de Campina Grande, Campina Grande, Paraíba, 2008.

RODRIGUES, C. L.; SILVA, K. R. G.; CUNHA, H. N. Improving Functional Verification of Embedded Systems Using Hierarchical Composition and Set Theory. Honolulu, Hawaii, USA, 2009.

SBCCI. Disponível em: <http://www.lsi.usp.br/chipinsampa/sbcci.htm>. Acesso em: 12 de julho de 2010.

SILVA, K. R. G. *Uma metodologia de Verificação Funcional para circuitos digitais*. 2007. 119 p. Tese - Universidade Federal de Campina Grande, Campina Grande, PB, 2007.

SILVEIRA, G. S. ; SILVA K. R. G. ; MELCHER, E. U. K. . Functional Verification of an MPEG-4 Decoder DesignUsing a Random Constrained Movie Generator. In: SBCCI 2007 Conference, 2007, Rio de Janeiro.

SPEAR, C. *Systemverilog for verification*. A guide to learning the testbench language features. USA, MA: Springer, 2006. 301 p.

SILVA, K. R. G. ; MELCHER, E. U. K. ; PESSOA, I. M. ; CUNHA, H. N. A methodology aimed at better integration of functional verification and rtl design. *Design Automation for Embedded Systems*, 10(4):285–298, 2007.

SUTHERLAND, S. ; DAVIDMANN, S. ; FLAKE, P. *SystemVerilog For Design: A Guide to Using SystemVerilog for Hardware Design and Modeling.* Second Edition. Springer, USA, 2006.

SYNOPSYS. *VMM Standard Library User Guide.* Version D-2009.12. Synopsys, Inc., USA, 2009.

UNISIM. Disponível em:<http://unisim.org>. Acesso em: 12 de julho de 2010.

WAGGENER, B. *Pulse Code Modulation Systems Design (1st ed.)*. Boston, MA: Artech House, 1999.

# Glossário

**AAC -** Do inglês *Advanced Audio Coding*. Esquema de codificação para compressão com perda de dados de som digital.

**AVM -** Do inglês *Advanced Verification Methodology*. Primeira metodologia de verificação funcional de código-aberto para a indústria no nível RTL desenvolvida pela Mentor Graphics.

**BVE-COVER -** Do inglês *Brazil-IP Verification Extension.* Biblioteca desenvolvida pelos membros do Brazil-IP, a fim de se implementar cobertura funcional em SystemC.

**Brazil-IP -** Esforço colaborativo entre instituições brasileiras para criar um conjunto de centros de desenvolvimento de circuitos integrados capazes de produzir núcleos de propriedade intelectual (IP *cores*).

**BVM -** Do inglês *Brazil-IP Verification Methodology.* Reformulação da metodologia de verificação funcional VeriSC, baseada em OVM (*Open Verification Methodology)* e implementada em SystemVerilog.

**CDV -** Do inglês *Coverage Driven Verification.* A verificação dirigida por cobertura combina geração de testes automáticos, *testbenches* auto-verificáveis e métricas de cobertura para reduzir o tempo gasto na verificação de um projeto.

**CLM -** Do inglês *Cycle-level Modelling*. Modelagem, da comunicação dos módulos que compõe o sistema, baseada em ciclos de *clock*.

**CPU -** Do inglês *Central Processing Unit.* A Unidade Central de Processamento é a parte de um sistema computacional que executa as instruções de um programa de computador.

**DPI -** Do inglês *Direct Programming Interface.* É uma interface de programação direta, que permite chamadas de funções C/C++ em ambientes SystemVerilog.

**DUT -** Do inglês *Design Under Test*, refere-se ao projeto sob teste. É o projeto de implementado em alguma linguagem de descrição de *hardware*.

**DUV -** Do inglês *Design Under Verification, refere-se ao* projeto sob verificação. Sinônimo de DUT.

**EDA -** Do inglês *Electronic Design Automation*, refere-se a uma categoria de ferramentas focadas no projeto, concepção e produção de sistemas eletrônicos, abrangendo desde o projeto de circuitos integrados até o desenho de placas de circuito impresso

**eDL -** Do inglês *eTBc Design Language*. Linguagem utilizada na descrição da TLN.

**ESL -** Do inglês *Electronic System Level*. Abstração utilizada a fim de melhorar a compreensão no nível de sistema e aumentar a probabilidade de uma implementação de funcionalidade bem sucedida, com uma boa relação custo-benefício.

**eTBc -** Do inglês *Easy Testbench Creator*. Ferramenta para construção semi-automática de ambientes de verificação das metodologias VeriSC e BVM.

**eTL -** Do inglês *eTBc Template Language*. Linguagem utilizada para criação dos *Template Patterns.*

**FIFO -** Do inglês *First In First Out*. Estrutura de dados do tipo fila.

**HDL -** Do inglês *Hardware Description Language*. Linguagem de descrição de *hardware*. Uma diferença significava entre esta e outras linguagens de programação voltadas para *software* é que a sintaxe e a semântica de linguagens HDL incluem notações explícitas que expressam concorrência e tempo.

**HDMI -** Do inglês *High-Definition Multimedia Interface*. Interface compacta de áudio/vídeo para transmitir dados digitais sem compressão.

**HVL -** Do inglês *Hardware Verification Language*. Linguagem de verificação de *hardware*.

**IEEE -** Do inglês *Institute of Electrical and Electronics Engineers*. Organização profissional sem fins lucrativos, fundada nos Estados Unidos. Sua meta é promover conhecimento no campo da engenharia elétrica, eletrônica e computação. Um de seus papéis mais importantes é o estabelecimento de padrões para formatos de computadores e dispositivos.

**IEEE 1149.1 -** Padrão para o JTAG.

**IEEE-754 -** Padrão para computação em ponto-flutuante.

**JTAG -** Do inglês *Joint Test Action Group*. Padrão mundial para leitura/varredura/escrita de componentes eletrônicos como Circuitos Integrados, memórias, entre outros.

**IP *core* -** Do inglês *Intellectual Property core*. Projeto de *hardware* que desempenha funcionalidade específica e permite reuso em diferentes sistemas. IP *cores* podem ser integrados para criação de um SoC.

**IPCM/URM -** Do inglês *Incisive Plan-to-Closure Universal Reuse Methodology*. Metodologia de verificação funcional desenvolvida pela Cadence.

**ISO -** Do inglês *International Organization for Standardization*. Organização que aprova normas internacionais em todos os campos técnicos.

**IUS -** Do inglês *Incisive Unified Simulator*. Ferramenta de simulação da Cadence.

**MPEG -** Do inglês *Moving Picture Experts Group.* Padrão de compressão de dados para vídeo

digital em formato de arquivo desenvolvido por um grupo de trabalho pertencente ao ISO, o grupo de especialistas em imagens com movimento.

**MPEG-2 AAC-LC -** Do inglês MPEG 2 *Advanced Audio Coding Low Complexity.* Padrão para compressão com perdas e codificação de áudio digital.

**OSCI -** Do inglês *Open SystemC Initiative*. Grupo dedicado à definição de padrões avançados de SystemC para projetos em nível de sistema (ESL).

**OVC -** Do inglês *OVM verification components*. Componentes UVC reusáveis.

**OVM -** Do inglês *Open Verification Methodology*. Metodologia de verificação funcional aberta, desenvolvida em conjunto pela Cadence e Mentor. Interoperável com múltiplas linguagens e simuladores, criada a partir de metodologias já difundidas: AVM da Mentor e URM da Cadence.

**RTL -** Do inglês *Register Transfer Level*. Descrição da operação de um circuito digital síncrono.

**RVM -** Do inglês *Reference Verification Methodology*. Metodologia de verificação funcional criada pela Synopsys.

**SBCCI -** Do inglês *Symposium on Integrated Circuits and Systems Design*. Fórum internacional, que ocorre anualmente no Brasil, dedicado a circuitos integrados e design de sistemas, testes e CAD.

**SCV -** Do inglês *SystemC Verification Library*. Biblioteca de SystemC que disponibiliza meios para para verificação funcional.

**SoC -** Do inglês *System on a chip*. Circuito integrado composto por mais de um IP *core*.

**SystemC -** Conjunto de classes C++ e macros que fornecem meios para simulação dirigida por eventos, permitindo o engenheiro de verificação simular processos simultâneos, descritos usando a sintaxe da linguagem C++.

**SystemVerilog -** Linguagem de descrição e verificação de *hardware.*

***Template Patterns* -** Arquivos de entrada da ferramenta eTBc, que guiam a geração de código fonte, de acordo com a TLN definida pelo usuário. São implementados na linguagem eTL (*eTBc Template Language*), funcionando como molde para geração de código compilável, seja SystemC, para o caso de VeriSC ou Systemverilog, para o caso de BVM.

***Testbench* -** Ambiente de verificação funcional do DUV.

**TLM -** Do inglês *Transaction Level Modelling*. Abordagem de alto nível para modelagem de comunicação entre blocos funcionais ou elementos do *testbench* de um sistema digital.

**TLN -** Do inglês *Transaction Level Netlist*. Arquivo descrito pelo engenheiro de verificação, que descreve como o IP será modularizado, como também as entradas e saídas desses módulos em

termos de sinais e transações

**UNISIM -** *UNIted SIMulation environment*. Ambiente de simulação modular, implementado em SystemC, com foco no reuso de lógica da controle, a qual corresponde a uma grande quantidade de código de um simulador

**USB -** Do inglês *Universal Serial Bus*. Especificação para estabelecimento de comunicação entre dispositivos e seus controladores.

**UVC -** Do inglês *Universal Verification Component*. Conjunto de elementos para controlar, estimular e recolher informações de cobertura para um específico protocolo ou projeto

**Verilog -** Linguagem de descrição de *hardware.*

**VeriSC –** Metodologia de verificação funcional implementada em SystemC, que visa implementação do *testbench* antes do DUV.

**VHDL -** Do inglês *VHSIC Hardware Description Language*. Linguagem de descrição de *hardware.*

**VMM -** Do inglês *Verification Methodology Manual*. Metodologia de verificação funcional criada pela Synopsys e documentada por um livro de autoria conjunta com a ARM.

# Apêndice A

## Código fonte dos *template patterns*

A seguir, será apresentado o código fonte dos *template patters* desenvolvidos durante o trabalho realizado.

### A.1 *axi_cover*

**Código fonte do *template pattern axi_cover***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(file) $$("AXI_cover.sv")
30
31      module AXI_cover (input clk, ready, valid);
32       logic ready_, valid_;
33       int count_ready_stable = 1, count_valid_stable = 1;
34       int ready_up, ready_down, valid_up, valid_down;
35
36       covergroup c_up;
37         coverpoint ready_up {
38           bins up[] = { [1:10] };
39           option.at_least = 50;
40         }
41         coverpoint valid_up {
42           bins up[] = { [1:10] };
43           option.at_least = 50;
44         }
45       endgroup
46       c_up c_up_i = new;
```

**Código fonte do *template pattern axi_cover* (Continuação)**

```
47
48      covergroup c_down;
49        coverpoint ready_down {
50          bins dn[] = { [1:10] };
51          option.at_least = 50;
52        }
53        coverpoint valid_down {
54          bins dn[] = { [1:10] };
55          option.at_least = 50;
56        }
57      endgroup
58      c_down c_down_i = new;
59
60      always@(posedge clk) begin
61        if(ready_ == ready) count_ready_stable += 1;
62        else begin
63          if(ready_) begin
64            ready_up = count_ready_stable;
65            c_up_i.sample();
66          end
67          else begin
68            ready_down = count_ready_stable;
69            c_down_i.sample();
70          end
71          count_ready_stable = 1;
72        end
73        ready_ <= ready;
74      end
75
76      always@(posedge clk) begin
77        if(valid_ == valid) count_valid_stable += 1;
78        else begin
79          if(valid_) begin
80            valid_up = count_valid_stable;
81            c_up_i.sample();
82          end
83          else begin
84            valid_down = count_valid_stable;
85            c_down_i.sample();
86          end
87          count_valid_stable = 1;
88        end
89        valid_ <= valid;
90      end
91
92     endmodule
93
94     $$(endfile)
```

## A.2 *checker*

**Código fonte do *template pattern checker***

```
1      $$(type.map) $$(bool->bit)
2      $$(type.map) $$(char->byte)
3      $$(type.map) $$(short->shortint)
4      $$(type.map) $$(short int->shortint)
```

**Código fonte do *template pattern checker* (Continuação)**

```
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(file) $$("checker.svh")
30
31      class checker extends ovm_threaded_component;
32
33        $$(foreach) $$(module.out)
34        ovm_get_port #($$(i.type)) $$(i.name)_from_refmod_port;
35        ovm_get_port #($$(i.type)) $$(i.name)_from_duv_port;
36        $$(endfor)
37        function new(string name, ovm_component parent);
38         super.new(name, parent);
39         $$(foreach) $$(module.out)
40         $$(i.name)_from_refmod_port = new("$$
41      (i.name)_from_refmod_port", this);
42         $$(i.name)_from_duv_port = new("$$(i.name)_from_duv_port",
43      this);
44         $$(endfor)
45        endfunction
46
47        task run();
48         $$(foreach) $$(module.out)
49         $$(i.type) tr_duv_$$(i.name),tr_modref_$$(i.name);
50         $$(endfor)
51         string msg;
52
53         recording_detail = OVM_FULL;
54
55         forever begin
56          $$(foreach) $$(module.out)
57          $$(i.name)_from_duv_port.get(tr_duv_$$(i.name));
58          $$(i.name)_from_refmod_port.get(tr_modref_$$(i.name));
59          if(!tr_duv_$$(i.name).equal(tr_modref_$$(i.name))) begin
60           msg = $psprintf("received: %s expected %s", tr_duv_$$
61      (i.name).psprint(), tr_modref_$$(i.name).psprint());
62           ovm_report_error("Checker", msg);
```

**Código fonte do *template pattern checker* (Continuação)**

```
63          assert(record_error_tr("checker"));
64          100ns;
65          global_stop_request();
66         end
67         $$(endfor)
68        end
69      endtask
70    endclass
71
72    $$(endfile)
```

## A.3 *duv*

**Código fonte do *template pattern duv***

```
1     $$(type.map) $$(short->signed)
2     $$(type.map) $$(short int->signed)
3     $$(type.map) $$(int->signed)
4     $$(type.map) $$(long->signed)
5     $$(type.map) $$(long int->signed)
6     $$(type.map) $$(long long->signed)
7     $$(type.map) $$(long long int->signed)
8     $$(type.map) $$(float->signed)
9     $$(type.map) $$(double->signed)
10    $$(type.map) $$(signed->signed)
11    $$(type.map) $$(signed int->signed)
12    $$(type.map) $$(signed short->signed)
13    $$(type.map) $$(signed short int->signed)
14    $$(type.map) $$(signed long->signed)
15    $$(type.map) $$(signed long int->signed)
16    $$(type.map) $$(signed long long->signed)
17    $$(type.map) $$(signed long long int->signed)
18
19    $$(file) $$("$$(module.name).sv")
20
21    module $$(module.name) (input reset, clk, $$(foreach)$$(module.in) $$
22    (foreach) $$(signal)
23      input $$(i.type) [$$(i.size)-1:0] $$(j.name)_$$(i.name),
24      input $$(j.name)_valid,
25      output logic $$(j.name)_ready,$$(endfor)$$(endfor) $$(foreach)$$
26    (module.out) $$(foreach)$$(signal)
27      output logic $$(i.type) [$$(i.size)-1:0] $$(j.name)_$$(i.name),
28      output logic $$(j.name)_valid,
29      input $$(j.name)_ready$$(endfor)$$(if)$$(isnotlast),$$(endif)$$
30    (endfor));
31      $$(foreach) $$(module.channel) $$(foreach) $$(signal)
32      logic $$(i.type) [$$(i.size)-1:0] $$(j.name)_$$(i.name);
33      input $$(i.name)_valid;
34      output logic $$(i.name)_ready;
35      $$(endfor) $$(endfor)
36
37    endmodule
38    $$(endfile)
```

## A.4 *duv_emulation*

**Código fonte do *template pattern duv_emulation***

```
1   $$(type.map) $$(bool->bit)
2   $$(type.map) $$(char->byte)
3   $$(type.map) $$(short->shortint)
4   $$(type.map) $$(short int->shortint)
5   $$(type.map) $$(int->int)
6   $$(type.map) $$(long->int)
7   $$(type.map) $$(long int->int)
8   $$(type.map) $$(long long->longint)
9   $$(type.map) $$(long long int->longint)
10  $$(type.map) $$(float->shortreal)
11  $$(type.map) $$(double->real)
12  $$(type.map) $$(signed->int)
13  $$(type.map) $$(unsigned->int unsigned)
14  $$(type.map) $$(signed int->int)
15  $$(type.map) $$(unsigned int->int unsigned)
16  $$(type.map) $$(signed short->shortint)
17  $$(type.map) $$(signed short int->shortint)
18  $$(type.map) $$(unsigned short->shortint)
19  $$(type.map) $$(unsigned short int->shortint)
20  $$(type.map) $$(signed long->int)
21  $$(type.map) $$(signed long int->int)
22  $$(type.map) $$(unsigned long->int)
23  $$(type.map) $$(unsigned long int->int)
24  $$(type.map) $$(signed long long->longint)
25  $$(type.map) $$(signed long long int->longint)
26  $$(type.map) $$(unsigned long long->longint)
27  $$(type.map) $$(unsigned long long int->longint)
28
29  $$(file) $$("$$(module.name).sv")
30
31  module $$(module.name)(input reset, clk, $$(foreach)$$(module.in) $$
32  (foreach) $$(signal)
33   input $$(i.type) [$$(i.size)-1:0] $$(j.name)_$$(i.name),
34   input $$(j.name)_valid,
35   output logic $$(j.name)_ready,$$(endfor)$$(endfor) $$(foreach)$$
36  (module.out) $$(foreach)$$(signal)
37   output logic $$(i.type) [$$(i.size)-1:0] $$(j.name)_$$(i.name),
38   output logic $$(j.name)_valid,
39   input $$(j.name)_ready$$(endfor)$$(if)$$(isnotlast),$$(endif)$$
40  (endfor));
41
42   `include "trans.svh" $$(foreach) $$(module.in)
43   `include "$$(i.name)_actor.svh"
44   `include "$$(i.name)_monitor.svh" $$(endfor)
45   `include "refmod_$$(module.name).svh" $$(foreach) $$(module.out)
46   `include "$$(i.name)_driver.svh" $$(endfor)
47
48   // input monitor(s) $$(foreach)$$(module.in)
49   $$(i.name)_actor $$(i.name)_actor_i = new ("$$(i.name)_actor_i",
50  null);
51   $$(i.name)_monitor $$(i.name)_monitor_i = new ("$$
52  (i.name)_monitor_i", null);
53   tlm_fifo #($$(i.type)) $$(i.name)_monitor_refmod = new("$$
54  (i.name)_monitor_refmod", null, 3);
55   $$(endfor)refmod_$$(module.name) refmod_$$(module.name)_i =
56  new("duv.refmod_$$(module.name)_i", null);
```

**Código fonte do *template pattern duv_emulation* (Continuação)**

```
57
58       // output driver(s) $$(foreach)$$(module.out)
59       tlm_fifo #($$(i.type)) $$(i.name)_refmod_driver = new("$$
60      (i.name)_refmod_driver", null, 3);
61       $$(i.name)_driver $$(i.name)_driver_i = new("$$(i.name)_driver_i",
62      null);
63       $$(endfor)
64       initial begin // connect the fifos
65        $$(foreach) $$(module.in)
66        $$(i.name)_monitor_i.$$(i.name)_to_duv_port.connect($$
67      (i.name)_monitor_refmod.put_export);
68        refmod_$$(module.name)_i.$$(i.name)_port.connect($$
69      (i.name)_monitor_refmod.get_export);
70        $$(endfor) $$(foreach) $$(module.out)
71        refmod_$$(module.name)_i.$$(i.name)_port.connect($$
72      (i.name)_refmod_driver.put_export);
73        $$(i.name)_driver_i.$$(i.name)_from_duv_port.connect($$
74      (i.name)_refmod_driver.get_export);
75        $$(endfor)
76        run_test();
77       end
78
79      endmodule
80
81      $$(endfile)
```

## A.5 *duv_hierarchical*

**Código fonte do *template pattern duv_hierarchical***

```
1       $$(type.map) $$(short->signed)
2       $$(type.map) $$(short int->signed)
3       $$(type.map) $$(int->signed)
4       $$(type.map) $$(long->signed)
5       $$(type.map) $$(long int->signed)
6       $$(type.map) $$(long long->signed)
7       $$(type.map) $$(long long int->signed)
8       $$(type.map) $$(float->signed)
9       $$(type.map) $$(double->signed)
10      $$(type.map) $$(signed->signed)
11      $$(type.map) $$(signed int->signed)
12      $$(type.map) $$(signed short->signed)
13      $$(type.map) $$(signed short int->signed)
14      $$(type.map) $$(signed long->signed)
15      $$(type.map) $$(signed long int->signed)
16      $$(type.map) $$(signed long long->signed)
17      $$(type.map) $$(signed long long int->signed)
18
19      $$(file) $$("$$(module.name).sv")
20
21      module $$(module.name)(input reset, clk, $$(foreach)$$(module.in) $$
22      (foreach) $$(signal)
23       input $$(i.type) [$$(i.size)-1:0] $$(j.name)_$$(i.name),
24       input $$(j.name)_valid,
25       output logic $$(j.name)_ready,$$(endfor)$$(endfor) $$(foreach)$$
26      (module.out) $$(foreach)$$(signal)
```

**Código fonte do *template pattern duv_hierarchical* (Continuação)**

```
27       output logic $$(i.type) [$$(i.size)-1:0] $$(j.name)_$$(i.name),
28       output logic $$(j.name)_valid,
29       input $$(j.name)_ready$$(endfor)$$(if)$$(isnotlast),$$(endif)$$
30      (endfor));
31
32      endmodule
33
34      $$(endfile)
```

## A.6 *gene_clock*

**Código fonte do *template pattern gene_clock***

```
1       $$(file) $$("gene_clock.sv")
2
3       module gene_clock(output logic reset, clk);
4
5        initial clk <= 0;
6        always #5ns clk <= !clk;
7
8        initial begin
9         reset <= 1;
10        #12ns;
11        reset <= 0;
12       end
13
14      endmodule
15
16      $$(endfile)
```

## A.7 *in_actor*

**Código fonte do *template pattern in_actor***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
```

**Código fonte do *template pattern in_actor* (Continuação)**

```
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(foreach) $$(module.in)
30      $$(file) $$("$$(i.name)_actor.svh")
31
32      class $$(i.name)_actor extends ovm_threaded_component;
33
34        function new(string name, ovm_component parent);
35         super.new(name, parent);
36        endfunction
37
38        task run();
39         bit wait_for_valid;
40         delay d = new;
41
42         $$(i.name)_ready <= 0;
43         @(posedge clk);
44         while(reset) @(posedge clk);
45
46         forever begin
47           wait_for_valid = $urandom_range(0, 1);
48
49           if (wait_for_valid) begin
50             while(!$$(i.name)_valid) @(posedge clk);
51             assert(d.randomize());
52             repeat(d.d) @(posedge clk);
53             @(negedge clk);
54             $$(i.name)_ready <= 1;
55             @(posedge clk);
56             while(!$$(i.name)_valid) @(posedge clk);
57             $$(i.name)_ready <= 0;
58             @(posedge clk);
59           end
60           else begin
61             @(negedge clk);
62             $$(i.name)_ready <= 1;
63             @(posedge clk);
64             while(!$$(i.name)_valid) @(posedge clk);
65             $$(i.name)_ready <= 0;
66             assert(d.randomize());
67             repeat(d.d) @(posedge clk);
68             @(posedge clk);
69           end
70
71         end
72        endtask
73       endclass
74
75      AXI_cover $$(i.name)_cover(.clk(clk), .ready($$(i.name)_ready),
76      .valid($$(i.name)_valid));
77      $$(endfor)
78      $$(endfile)
```

## A.8 *Makefile_double_refmod*

**Código fonte do *template pattern Makefile_double_refmod***

```
1       $$(file) $$("Makefile")
2
3       #Variables
4       MODULE = $$(module.name)
5       OVM = -ovm
6       SCRIPT = -input tb.tcl
7       ACCESS = -access +r
8       COVER = -coverage all
9       TB = tb.sv
10      RM = refmod_$(MODULE).svh $(MODULE).svh
11      C = $(MODULE).c
12
13      #Targets
14      cov_work : .tb_cpld tb.tcl
15        rm -rf cov_work; irun $(OVM) $(SCRIPT) $(ACCESS) $(COVER) $(TB)
16
17      .tb_cpld : trans.svh source.svh checker.svh definitions.svh $(RM) $
18      (TB)
19        irun $(OVM) -compile $(TB) && touch .tb_cpld
20
21      run :
22        rm -rf cov_work; $(MAKE)
23
24      #Cleans
25      clean :
26        rm -rf cov_work cov.cmd INCA_libs waves.shm *.log *.key *.txt
27      .tb_cpld .simvision *~ *#
28
29      $$(endfile)
```

## A.9 *Makefile_duv*

**Código fonte do *template pattern Makefile_duv***

```
1       $$(file) $$("Makefile")
2
3       cov_work : .tb_cpld .duv_cpld top.sv gene_clock.sv AXI_cover.sv
4       top.tcl
5         rm -rf cov_work; irun -ovm -input top.tcl -access +r -coverage all
6       top.sv gene_clock.sv AXI_cover.sv
7
8       .tb_cpld : tb.sv trans.svh source.svh refmod_$$(module.name).svh $$
9       (module.name).svh checker.svh $$(foreach)$$(module.in)$$
10      (i.name)_driver.svh $$(endfor) $$(foreach)$$(module.out)$$
11      (i.name)_actor.svh $$(i.name)_monitor.svh $$(endfor)
12        irun -ovm -compile tb.sv && touch .tb_cpld
13
14      .duv_cpld : $$(module.name).sv
15        irun -ovm -compile $$(module.name).sv && touch .duv_cpld
16
17      clean :
18        rm -rf cov_work cov.cmd INCA_libs waves.shm *.log *.key *.txt
19      .tb_cpld .duv_cpld .simvision *~ #*
20      $$(endfile)
```

## A.10 *Makefile_duv_emulation*

**Código fonte do *template pattern Makefile_duv_emulation***

```
1       $$(file) $$("Makefile")
2
3       cov_work : .tb_cpld .duv_cpld top.sv gene_clock.sv AXI_cover.sv
4       top.tcl
5         rm -rf cov_work; irun -ovm -input top.tcl -access +r -coverage all
6       top.sv gene_clock.sv AXI_cover.sv
7
8
9       .tb_cpld : tb.sv trans.svh source.svh refmod_$$(module.name).svh $$
10      (module.name).svh checker.svh $$(foreach)$$(module.in)$$
11      (i.name)_driver.svh $$(endfor) $$(foreach)$$(module.out)$$
12      (i.name)_actor.svh $$(i.name)_monitor.svh $$(endfor)
13        irun -ovm -compile tb.sv && touch .tb_cpld
14
15      .duv_cpld : $$(module.name).sv trans.svh $$(foreach)$$(module.in)$$
16      (i.name)_actor.svh $$(i.name)_monitor.svh $$(endfor) refmod_$$
17      (module.name).svh $$(module.name).svh $$(foreach)$$(module.out)$$
18      (i.name)_driver.svh $$(endfor)
19        irun -ovm -compile $$(module.name).sv && touch .duv_cpld
20
21      clean :
22        rm -rf cov_work cov.cmd INCA_libs waves.shm *.log *.key *.txt
23      .tb_cpld .duv_cpld .simvision *~ #*
24
25      $$(endfile)
```

## A.11 *Makefile_full*

**Código fonte do *template pattern Makefile_full***

```
1       $$(file) $$("Makefile")
2
3       cov_work : .tb_cpld .duv_cpld top.sv gene_clock.sv AXI_cover.sv
4       top.tcl
5         rm -rf cov_work; irun -ovm -input top.tcl -access +r -coverage all
6       top.sv gene_clock.sv AXI_cover.sv
7
8       .tb_cpld : tb.sv trans.svh source.svh refmod_$$(module.name).svh $$
9       (module.name).svh checker.svh $$(foreach)$$(module.in)$$
10      (i.name)_driver.svh $$(endfor) $$(foreach)$$(module.out)$$
11      (i.name)_actor.svh $$(i.name)_monitor.svh $$(endfor)
12        irun -ovm -compile tb.sv && touch .tb_cpld
13
14      .duv_cpld : $$(foreach)$$(module.inst)$$(i.type).sv $$(foreach)$$
15      (module.in)$$(i.name)_actor.svh $$(endfor) $$(endfor) $$
16      (module.name).sv trans.svh refmod_$$(module.name).svh $$
17      (module.name).svh
18        irun -ovm -compile $$(module.name).sv && touch .duv_cpld
19
20      clean :
21        rm -rf cov_work cov.cmd INCA_libs waves.shm *.log *.key *.txt
22      .tb_cpld .duv_cpld .simvision *~ #*
23
24      $$(endfile)
```

## A.12 *Makefile_full_analysis*

**Código fonte do *template pattern Makefile_full_analysis***

```
1   $$(file) $$("Makefile")
2
3   cov_work : .tb_cpld .duv_cpld top.sv gene_clock.sv AXI_cover.sv
4   top.tcl
5     rm -rf cov_work; irun -ovm -input top.tcl -access +r
6   -coverage all
7   top.sv gene_clock.sv AXI_cover.sv
8
9
10  .tb_cpld : tb.sv trans.svh source.svh $$(foreach)$$(module.inst)$
11  $refmod_$$(i.type).svh $$(endfor)
12  $$(foreach)$$(module.in)$$
13  (i.name)_driver.svh $$(endfor) $$(foreach)$$(module.out)$$
14  (i.name)_actor.svh $$(i.name)_monitor.svh $$(endfor)
15    irun -ovm -compile tb.sv && touch .tb_cpld
16
17
18  .duv_cpld : $$(foreach)$$(module.inst)$$(i.type).sv $$(foreach)$$
19  (module.in)$$(i.name)_actor.svh $$(endfor) $$(endfor) $$
20  (module.name).sv trans.svh refmod_$$(module.name).svh $$
21  (module.name).svh $$(foreach)$$(module.channel)$$
22  (i.name)_monitor.svh$$(endfor)
23    irun -ovm -compile $$(module.name).sv && touch .duv_cpld
24
25  clean :
26    rm -rf cov_work cov.cmd INCA_libs waves.shm *.log *.key *.txt
27  .tb_cpld .duv_cpld .simvision *~ #*
28
29
30  $$(endfile)
```

## A.13 *Makefile_hierarchical_refmod*

**Código fonte do *template pattern Makefile_hierarchical_refmod***

```
1   $$(file) $$("Makefile")
2
3   cov_work : .tb_cpld tb.tcl
4     rm -rf cov_work; irun -ovm -input tb.tcl -access +r -coverage all
5   tb.sv
6
7
8   .tb_cpld : tb.sv trans.svh source.svh refmod_$$(module.name).svh $$
9   (module.name).svh $$(foreach)$$(module.inst)refmod_$$(i.type).svh $$
10  (i.type).svh $$(endfor)checker.svh
11    irun -ovm -compile tb.sv && touch .tb_cpld
12
13  clean :
14    rm -rf cov_work cov.cmd INCA_libs waves.shm *.log *.key *.txt
15  .tb_cpld .duv_cpld .simvision *~ #*
16
17  $$(endfile)
```

## A.14 *Makefile_single_refmod*

**Código fonte do *template pattern Makefile_single_refmod***

```
1       $$(file) $$("Makefile")
2
3       cov_work : tb.sv trans.svh source.svh refmod_$$(module.name).svh
4       $$
5       (module.name).svh sink.svh tb.tcl
6         rm -rf cov_work; irun -ovm -input tb.tcl -access +r -coverage all
7       tb.sv
8
9       clean :
10        rm -rf cov_work cov.cmd INCA_libs waves.shm *.log *.key *.txt
11      .tb_cpld .duv_cpld .simvision *~ #*
12
13      $$(endfile)
```

## A.15 *out_actor*

**Código fonte do *template pattern out_actor***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(foreach) $$(module.out)
30      $$(file) $$("$$(i.name)_actor.svh")
31
32      class $$(i.name)_actor extends ovm_threaded_component;
33
34        function new(string name, ovm_component parent);
35
```

**Código fonte do *template pattern out_actor* (Continuação)**

```
36      super.new(name,parent);
37        endfunction
38
39        task run();
40
41         delay d = new;
42
43         $$(i.name)_ready <= 1;
44         @(posedge clk);
45         while(reset) @(posedge clk);
46         forever begin
47
48           while(!$$(i.name)_valid) @(posedge clk);
49           $$(i.name)_ready <= 0;
50           assert(d.randomize());
51           repeat(d.d) @(posedge clk);
52           @(negedge clk);
53           $$(i.name)_ready <= 1;
54           @(posedge clk);
55
56         end
57        endtask
58       endclass
59
60      AXI_cover $$(i.name)_cover(.clk(clk), .ready($$(i.name)_ready),
        .valid($$(i.name)_valid)); // prestar atencao
61      $$(endfor)
62      $$(endfile)
```

## A.16 *refmod_caller*

**Código fonte do *template pattern refmod_caller***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
```

**Código fonte do *template pattern refmod_caller* (Continuação)**

```
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(file) $$("refmod_$$(module.name).svh")
30
31      `include "$$(module.name).svh"
32
33      class refmod_$$(module.name) extends ovm_component;
34
35        $$(foreach)$$(module.in)ovm_get_port #($$(i.type)) $$
36      (i.name)_port;
37        $$(i.type) tr_in_$$(i.name);
38        $$(endfor)
39        $$(foreach)$$(module.out)ovm_put_port #($$(i.type)) $$
40      (i.name)_port;
41        $$(i.type) tr_out_$$(i.name);
42        $$(endfor)
43        covergroup crm;
44         $$(foreach)$$(module.out)$$(foreach)$$(var)
45         coverpoint tr_out_$$(j.name).$$(i.name) {
46           bins tr[] = { 0 };
47           option.at_least = 1;
48         }
49         $$(endfor)$$(endfor)
50        endgroup
51
52        function new(string name, ovm_component parent);
53         super.new(name,parent);
54         $$(foreach)$$(module.in)$$(i.name)_port = new("$$
55      (i.name)_port",
56      this);
57         $$(endfor)
58         $$(foreach)$$(module.out)$$(i.name)_port = new("$$
59      (i.name)_port",
60      this);
61         $$(endfor)
62         crm = new;
63        endfunction
64
65        task run();
66
67         recording_detail = OVM_FULL;
68
69         forever begin
70          $$(foreach)$$(module.in)$$(i.name)_port.get(tr_in_$$
71      (i.name));
72          $$(endfor)
73          $$(foreach)$$(module.out)tr_out_$$(i.name)= new();
74          $$(endfor)
75          //-------------------------------------------------------------
76          // Here goes the code that executes the reference model's
77            functionality.
78          //-------------------------------------------------------------
79
80          crm.sample();
81          $$(foreach)$$(module.out)$$(i.name)_port.put(tr_out_$$
82      (i.name));
```

**Código fonte do *template pattern refmod_caller* (Continuação)**

```
83          $$(endfor)
84         end
85       endtask
86     endclass
87
88     $$(endfile)
```

## A.17 *sink*

**Código fonte do *template pattern sink***

```
1      $$(type.map) $$(bool->bit)
2      $$(type.map) $$(char->byte)
3      $$(type.map) $$(short->shortint)
4      $$(type.map) $$(short int->shortint)
5      $$(type.map) $$(int->int)
6      $$(type.map) $$(long->int)
7      $$(type.map) $$(long int->int)
8      $$(type.map) $$(long long->longint)
9      $$(type.map) $$(long long int->longint)
10     $$(type.map) $$(float->shortreal)
11     $$(type.map) $$(double->real)
12     $$(type.map) $$(signed->int)
13     $$(type.map) $$(unsigned->int unsigned)
14     $$(type.map) $$(signed int->int)
15     $$(type.map) $$(unsigned int->int unsigned)
16     $$(type.map) $$(signed short->shortint)
17     $$(type.map) $$(signed short int->shortint)
18     $$(type.map) $$(unsigned short->shortint)
19     $$(type.map) $$(unsigned short int->shortint)
20     $$(type.map) $$(signed long->int)
21     $$(type.map) $$(signed long int->int)
22     $$(type.map) $$(unsigned long->int)
23     $$(type.map) $$(unsigned long int->int)
24     $$(type.map) $$(signed long long->longint)
25     $$(type.map) $$(signed long long int->longint)
26     $$(type.map) $$(unsigned long long->longint)
27     $$(type.map) $$(unsigned long long int->longint)
28
29     $$(file) $$("sink.svh")
30
31     class sink extends ovm_component;
32
33       $$(foreach)$$(module.out)ovm_get_port #($$(i.type)) $$
34     (i.name)_from_refmod_port;
35       $$(endfor)
36       function new(string name, ovm_component parent);
37        super.new(name,parent);
38        $$(foreach)$$(module.out)$$(i.name)_from_refmod_port = new("$$
39     (i.name)_from_refmod_port", this);
40        $$(endfor)
41       endfunction
42
43       task run();
44         $$(foreach)$$(module.out)$$(i.type) tr_$$(i.name);$$(endfor)
45
```

**Código fonte do *template pattern sink* (Continuação)**

```
46          forever begin
47           $$(foreach)$$(module.out)$$
48      (i.name)_from_refmod_port.get(tr_$$(i.name));
49           $$(endfor)
50          end
51        endtask
52      endclass
53
54      $$(endfile)
```

## A.18 *source*

**Código fonte do *template pattern source***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(file) $$("source.svh")
30
31      class source extends ovm_component;
32
33       $$(foreach) $$(module.in)
34        ovm_put_port #($$(i.type)) $$(i.name)_to_refmod_port;
35       $$(endfor)
36        function new(string name, ovm_component parent);
37         super.new(name,parent);
38        $$(foreach) $$(module.in)
39         $$(i.name)_to_refmod_port = new("$$(i.name)_to_refmod_port",
40      this);
41        $$(endfor)
42
```

**Código fonte do *template pattern source* (Continuação)**

```
43        endfunction
44        task run();
45         $$(foreach) $$(module.in)
46          $$(i.type) tr_$$(i.name);
47         $$(endfor)
48          int file = $fopen("tr.sti", "r");
49          recording_detail = OVM_FULL;
50          forever begin
51           $$(foreach) $$(module.in)
52            tr_$$(i.name) = new();
53            if(!tr_$$(i.name).read(file))
54              assert(tr_$$(i.name).randomize());
55            assert(begin_tr(tr_$$(i.name), "source", "tr_source"));
56            $$(i.name)_to_refmod_port.put(tr_$$(i.name));
57           $$(endfor)
58            #20ns;
59            end_tr(tr_$$(i.name));
60          end
61        endtask
62      endclass
63
64      $$(endfile)
```

## A.19 *tb_double_refmod*

**Código fonte do *template pattern tb_double_refmod***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(file) $$("tb.sv")
```

**Código fonte do *template pattern tb_double_refmod* (Continuação)**

```
30
31      module tb;
32
33        `include "trans.svh"
34        `include "source.svh"
35        `include "refmod_$$(module.name).svh"
36        `include "checker.svh"
37
38        source sou = new("sou", null);
39        refmod_$$(module.name) rem1 = new("rem1", null);
40        refmod_$$(module.name) rem2 = new("rem2", null);
41        checker che = new("che", null);
42
43        $$(foreach) $$(module.in)
44        tlm_fifo #($$(i.type)) $$(i.name)_source_refmod1 = new("$$
45      (i.name)_source_refmod1", null, 3);
46        tlm__analysis_fifo #($$(i.type)) $$(i.name)_source_refmod2 = new("$$
47      (i.name)_source_refmod2", null);
48        $$(endfor)
49
50        $$(foreach) $$(module.out)
51        tlm_fifo #($$(i.type)) $$(i.name)_refmod1_checker = new("$$
52      (i.name)_refmod1_checker", null, 3);
53        tlm_fifo #($$(i.type)) $$(i.name)_refmod2_checker = new("$$
54      (i.name)_refmod2_checker", null, 3);
55        $$(endfor)
56
57        initial begin
58        $$(foreach) $$(module.in)
59
60          //Source to Refmod1
61          sou.$$(i.name)_to_refmod_port.connect($$
62      (i.name)_source_refmod1.put_export);
63          rem1.$$(i.name)_port.connect($$
64      (i.name)_source_refmod1.get_export);
65
66          //Source to Refmod2 (by emulated FIFO's 2nd head)
67          $$(i.name)_source_refmod1.put_ap.connect($$
68      (i.name)_source_refmod2.analysis_export)
69          rem2.$$(i.name)_port.connect($$
70      (i.name)_source_refmod2.get_export);
71        $$(endfor)
72
73          //Refmod1 to Checker
74      $$(foreach) $$(module.out)
75          rem1.$$(i.name)_port.connect($$
76      (i.name)_refmod1_checker.put_export);
77          che.$$(i.name)_from_refmod_port.connect($$
78      (i.name)_refmod1_checker.get_export);
79
80          //Refmod2 to Checker
81          rem2.$$(i.name)_port.connect($$
82      (i.name)_refmod2_checker.put_export);
83          che.$$(i.name)_from_duv_port.connect($$
84      (i.name)_refmod2_checker.get_export);
85        $$(endfor)
86
87          $shm_open("waves.shm");
88          run_test();
```

**Código fonte do *template pattern tb_double_refmod* (Continuação)**

```
89       end
90
91       logic cov;
92       initial cov = 1;
93       always #25us cov = !cov;
94
95      endmodule
96
97      $$(endfile)
```

## A.20 *tb_duv*

**Código fonte do *template pattern tb_duv***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29
30      $$(file) $$("tb.sv")
31
32      module tb(input reset, clk, $$(foreach)$$(module.in) $$(foreach) $$
33      (signal)
34       output logic $$(i.type) [$$(i.size):1] $$(j.name)_$$(i.name),
35       output logic $$(j.name)_valid,
36       input $$(j.name)_ready,$$(endfor)$$(endfor) $$(foreach)$$
37      (module.out) $$(foreach)$$(signal)
38       input $$(i.type) [$$(i.size):1] $$(j.name)_$$(i.name),
39       input $$(j.name)_valid,
40       output logic $$(j.name)_ready$$(endfor)$$(if) $$(isnotlast),$$
41      (endif)$$(endfor));
42
```

**Código fonte do *template pattern tb_duv* (Continuação)**

```
43      `include "trans.svh"
44      `include "source.svh"
45      `include "refmod_$$(module.name).svh"
46      `include "checker.svh"
47      $$(foreach)$$(module.in)
48      `include "$$(i.name)_driver.svh"
49      $$(endfor)
50      $$(foreach)$$(module.out)
51      `include "$$(i.name)_actor.svh"
52      `include "$$(i.name)_monitor.svh"
53      $$(endfor)
54
55      // source, refmod, checker and fifos that connect them to one
56     another
57      source sou = new("sou", null);
58      $$(foreach)$$(module.in)
59      tlm_fifo #($$(i.type)) $$(i.name)_source_refmod = new("$$
60     (i.name)_source_refmod", null, 3);
61      $$(endfor)
62      refmod_$$(module.name) refmod_$$(module.name)_i = new("tb.refmod_$$
63     (module.name)_i", null);
64      $$(foreach)$$(module.out)
65      tlm_fifo #($$(i.type)) $$(i.name)_refmod_checker = new("$$
66     (i.name)_refmod_checker", null, 3);
67      $$(endfor)
68      checker che = new("che", null);
69
70      // driver(s) and fifo(s) that connect to them
71      $$(foreach)$$(module.in)
72      tlm_analysis_fifo #($$(i.type)) $$(i.name)_source_driver = new("$$
73     (i.name)_source_driver", null);
74      $$(i.name)_driver $$(i.name)_driver_i = new("$$(i.name)_driver_i",
75     null);
76      $$(endfor)
77
78      // monitor(s) and fifo(s) that conecto from them
79      $$(foreach)$$(module.out)
80      $$(i.name)_actor $$(i.name)_actor_i = new("$$(i.name)_actor_i",
81     null);
82      $$(i.name)_monitor $$(i.name)_monitor_i = new("$$
83     (i.name)_monitor_i", null);
84      tlm_fifo #($$(i.type)) $$(i.name)_monitor_checker =new("$$
85     (i.name)_monitor_checker", null, 3);
86      $$(endfor)
87
88      initial begin // connect the fifos
89
90        // source to refmod
91        $$(foreach)$$(module.in)
92        sou.$$(i.name)_to_refmod_port.connect($$
93     (i.name)_source_refmod.put_export);
94        refmod_$$(module.name)_i.$$(i.name)_port.connect($$
95     (i.name)_source_refmod.get_export);
96        $$(endfor)
97
98         // refmod to checker
99        $$(foreach)$$(module.out)
100       refmod_$$(module.name)_i.$$(i.name)_port.connect($$
101    (i.name)_refmod_checker.put_export);
```

**Código fonte do *template pattern tb_duv* (Continuação)**

```
102       che.$$(i.name)_from_refmod_port.connect($$
103     (i.name)_refmod_checker.get_export);
104       $$(endfor)
105
106       // source to driver(s)
107       $$(foreach)$$(module.in)
108       $$(i.name)_source_refmod.put_ap.connect($$
109     (i.name)_source_driver.analysis_export);
110       $$(i.name)_driver_i.$$(i.name)_from_source_port.connect($$
111     (i.name)_source_driver.get_export);
112       $$(endfor)
113
114       // monitor(s) to checker
115       $$(foreach)$$(module.out)
116       $$(i.name)_monitor_i.$$(i.name)_to_checker_port.connect($$
117     (i.name)_monitor_checker.put_export);
118       che.$$(i.name)_from_duv_port.connect($$
119     (i.name)_monitor_checker.get_export);
120       $$(endfor)
121
122       run_test();
123      end
124
125     endmodule
126
127     $$(endfile)
```

## A.21 *tb_duv_analysis*

**Código fonte do *template pattern tb_duv_analysis***

```
1     $$(type.map) $$(bool->bit)
2     $$(type.map) $$(char->byte)
3     $$(type.map) $$(short->shortint)
4     $$(type.map) $$(short int->shortint)
5     $$(type.map) $$(int->int)
6     $$(type.map) $$(long->int)
7     $$(type.map) $$(long int->int)
8     $$(type.map) $$(long long->longint)
9     $$(type.map) $$(long long int->longint)
10    $$(type.map) $$(float->shortreal)
11    $$(type.map) $$(double->real)
12    $$(type.map) $$(signed->int)
13    $$(type.map) $$(unsigned->int unsigned)
14    $$(type.map) $$(signed int->int)
15    $$(type.map) $$(unsigned int->int unsigned)
16    $$(type.map) $$(signed short->shortint)
17    $$(type.map) $$(signed short int->shortint)
18    $$(type.map) $$(unsigned short->shortint)
19    $$(type.map) $$(unsigned short int->shortint)
20    $$(type.map) $$(signed long->int)
21    $$(type.map) $$(signed long int->int)
22    $$(type.map) $$(unsigned long->int)
23    $$(type.map) $$(unsigned long int->int)
24    $$(type.map) $$(signed long long->longint)
25    $$(type.map) $$(signed long long int->longint)
```

**Código fonte do *template pattern tb_duv_analysis* (Continuação)**

```
26     $$(type.map) $$(unsigned long long->longint)
27     $$(type.map) $$(unsigned long long int->longint)
28
29     $$(file) $$("tb.sv")
30
31     module tb(input reset, clk, $$(foreach)$$(module.in) $$(foreach) $$
32     (signal)
33      output logic $$(i.type) [$$(i.size):1] $$(j.name)_$$(i.name),
34      output logic $$(j.name)_valid,
35      input $$(j.name)_ready,$$(endfor)$$(endfor) $$(foreach)$$
36     (module.out) $$(foreach)$$(signal)
37      input $$(i.type) [$$(i.size):1] $$(j.name)_$$(i.name),
38      input $$(j.name)_valid,
39      output logic $$(j.name)_ready$$(endfor)$$(if) $$(isnotlast),$$
40     (endif)$$(endfor));
41
42      `include "trans.svh"
43      `include "source.svh"
44      `include "refmod_$$(module.name).svh"
45      `include "checker_$$(module.name).svh"
46      $$(foreach)$$(module.in)
47      `include "$$(i.name)_driver.svh"
48      $$(endfor)
49      $$(foreach)$$(module.out)
50      `include "$$(i.name)_actor.svh"
51      $$(endfor)
52
53      $$(foreach) $$(module.inst)
54      `include "refmod_$$(i.type).svh"
55      $$(endfor)
56
57      //You will need include all the checkers from Hierarchical duv
58      $$(foreach) $$(module.channel)
59      `include "checker_$$(i.name).svh" //Intermediate checker (rename)
60      `include "$$(i.name)_monitor.svh" //Intermediate monitor
61      $$(endfor)
62
63      $$(foreach)$$(module.out)
64      `include "$$(i.name)_monitor.svh"
65      $$(endfor)
66       // source and toplevel checker
67      source source_i = new("source_i", null);
68      checker_$$(module.name) checker_$$(module.name)_i = new("checker_$$
69     (module.name)_i", null);
70
71     //Reference models and fifos to connect them
72      $$(foreach) $$(module.inst)
73      refmod_$$(i.type) refmod_$$(i.type)_i = new ("refmod_$$(i.type)_i",
74     null);
75      $$(foreach) $$(module.in)
76      tlm_fifo #($$(i.type)) $$(i.name)_$$(j.name) = new("$$(i.name)_$$
77     (j.name)", null, 3);
78      $$(endfor)
79      $$(endfor)
80
81       //Fifos to connect toplevel checker
82      $$(foreach) $$(module.out)
83      tlm_fifo #($$(i.type)) $$(i.name)_checker = new("$$
84      $$(i.name)_fifo", null, 3);
```

**Código fonte do *template pattern tb_duv_analysis* (Continuação)**

```
85      $$(endfor)
86
87      // driver(s) and fifo(s) that connect to them
88      $$(foreach)$$(module.in)
89      tlm_analysis_fifo #($$(i.type)) $$(i.name)_source_driver = new("$$
90      (i.name)_source_driver", null);
91      $$(i.name)_driver $$(i.name)_driver_i = new("$$(i.name)_driver_i",
92      null);
93      $$(endfor)
94
95      //toplevel monitor(s) and fifo(s) that connect from them
96      $$(foreach)$$(module.out)
97      $$(i.name)_actor $$(i.name)_actor_i = new("$$(i.name)_actor_i",
98     null);
99      $$(i.name)_monitor $$(i.name)_monitor_i = new("$$
100    (i.name)_monitor_i", null);
101     tlm_fifo #($$(i.type)) $$(i.name)_monitor_checker =new("$$
102     $$(endfor)
103
104
105     // intermediate checker(s), monitor(s) and fifos that connect them
106    to another
107     $$(foreach) $$(module.channel)
108     tlm_analysis_fifo #($$(i.type)) $$(i.name)_fifo_an =
109    new("$$(i.name)_fifo_an", null);
110     tlm_fifo #($$(i.type)) $$(i.name)_fifo =
111    new("$$(i.name)_fifo", null);
112     $$(i.name)_monitor $$(i.name)_monitor_i = new("$$
113    (i.name)_monitor_i", null);
114     checker_$$(i.name) $$(i.name)_checker_i = new("$$
115    (i.name)_checker_i", null);//intermediate checker (rename)
116     $$(endfor)
117
118
119     initial begin // fifo connection
120
121      // Connect the source to the first refmod
122       $$(foreach)$$(module.in)
123       source_i.$$(i.name)_to_refmod.connect($$
124    (i.name)_fifo.put_export);
125    $$(endfor)
126
127      //Create the connection between refmods
128      $$(foreach) $$(module.inst)
129      $$(foreach)$$(module.in)
130      refmod_$$(j.name)_i.$$(i.name)_port.connect($$(i.name)_$$
131    (j.name).get_export);
132      $$(endfor)
133      $$(endfor)
134
135      //Create the analysys fifo to send data to intermediate checkers
136
137
138
139
140
141
142
143
```

**Código fonte do *template pattern tb_duv_analysis* (Continuação)**

```
144
145
146
147
148
149
150       // Connect the last refmod to toplevel checker
151
152
153
154
155
156
157
158       // Connectiong the monitor(s) to checker
159       $$(foreach)$$(module.out)
160       $$(i.name)_monitor_i.$$(i.name)_to_checker_port.connect($$
161     (i.name)_monitor_checker.put_export);
162       che.$$(i.name)_from_duv_port.connect($$
163     (i.name)_monitor_checker.get_export);
164       $$(endfor)
165
166       run_test();
167      end
168
169     endmodule
170
171     $$(endfile)
```

## A.22 *tb_hierarchical_refmod*

**Código fonte do *template pattern tb_hierarchical_refmod***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
```

**Código fonte do *template pattern tb_hierarchical_refmod* (Continuação)**

```
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(file) $$("tb.sv")
30
31      module tb;
32       `include "trans.svh"
33       `include "source.svh"
34       `include "refmod_$$(module.name).svh"
35       `include "checker.svh"
36
37       $$(foreach) $$(module.inst)
38       `include "refmod_$$(i.type).svh"
39       $$(endfor)
40
41       source sou = new("sou", null);
42       refmod_$$(module.name) rem1 = new("rem1", null);
43       $$(foreach) $$(module.inst)
44       refmod_$$(i.type) $$(i.type)_i = new("$$(i.type)_i", null);
45       $$(endfor)
46       checker che = new("che", null);
47
48       $$(foreach) $$(module.in)
49       tlm_fifo #($$(i.type)) $$(i.name)_source_refmod1 = new("$$
50      (i.name)_source_refmod1", null, 3);
51       tlm_analysis_fifo #($$(i.type)) $$(i.name)_source_refmod2 = new("$$
52      (i.name)_source_refmod2", null, 3);
53       $$(endfor)
54
55       $$(foreach) $$(module.channel)
56       tlm_fifo #($$(i.type)) $$(i.name)_channel = new("$$
57      (i.name)_channel", null, 3);
58       $$(endfor)
59
60       $$(foreach) $$(module.out)
61       tlm_fifo #($$(i.type)) $$(i.name)_refmod1_checker = new("$$
62      (i.name)_refmod1_checker", null, 3);
63       tlm_fifo #($$(i.type)) $$(i.name)_refmod2_checker = new("$$
64      (i.name)_refmod2_checker", null, 3);
65       $$(endfor)
66
67       initial begin
68        //Fazer conexoes manualmente
69
70        //conectar Source ao Modelo de Referencia inicial. Usar FIFO de
71        //duas saidas
72
73        //conectar todos os Modelos de Referencia resultantes da divisao
74        //hierarquica
75
76        //conectar o checker ao Modelo de Referencia inicial e a uniao dos
77        //outros modelos de referencia
78
79        $shm_open("waves.shm");
80        run_test();
81       end
82
```

**Código fonte do *template pattern tb_hierarchical_refmod* (Continuação)**

```
83       logic cov;
84       initial cov = 0;
85       always #100 cov = !cov;
86
87      endmodule
88
89      $$(endfile)
```

## A.23 *tb_single_refmod*

**Código fonte do *template pattern tb_single_refmod***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(file) $$("tb.sv")
30      //Code generated by eTBc software
31
32      module tb;
33       `include "trans.svh"
34       `include "source.svh"
35       `include "refmod_$$(module.name).svh"
36       `include "sink.svh"
37
38       source source_i = new("source_i", null);
39       refmod_$$(module.name) refmod_$$(module.name)_i = new("refmod_$$
40      (module.name)_i", null);
41       sink sink_i = new("sink_i", null);
42       $$(foreach)$$(module.in)tlm_fifo #($$(i.type)) $$
43      (i.name)_source_refmod = new("$$(i.name)_source_refmod", null, 3);
44       $$(endfor)
```

**Código fonte do *template pattern tb_single_refmod* (Continuação)**

```
45      $$(foreach)$$(module.out)tlm_fifo #($$(i.type)) $$
46     (i.name)_refmod_checker = new("$$(i.name)_refmod_checker", null, 3);
47      $$(endfor)
48
49      initial begin
50
51        $$(foreach)$$(module.in)
52        source_i.$$(i.name)_to_refmod_port.connect($$
53     (i.name)_source_refmod.put_export);
54        refmod_$$(module.name)_i.$$(i.name)_port.connect($$
55     (i.name)_source_refmod.get_export);
56        $$(endfor)
57
58        $$(foreach)$$(module.out)
59        refmod_$$(module.name)_i.$$(i.name)_port.connect($$
60     (i.name)_refmod_checker.put_export);
61        sink_i.$$(i.name)_from_refmod_port.connect($$
62     (i.name)_refmod_checker.get_export);
63        $$(endfor)
64        $shm_open("waves.shm");
65        run_test();
66      end
67
68      logic cov;
69      initial cov = 0;
70      always #100 cov = !cov;
71
72     endmodule
73
74     $$(endfile)
```

## A.24 *tb_tcl*

**Código fonte do *template pattern tb_tcl***

```
1      $$(file)$$("tb.tcl")set idmp 0
2      proc covering {} {
3       upvar 1 idmp idmp
4       rm -rf ./cov_work/design/$idmp
5       set idmp [expr {$idmp+1}]
6       coverage -dump $idmp
7       set fileId [open "cov.cmd" w]
8       puts $fileId "load_test $idmp\nreport_summary -cgopt tb\nquit\n\n"
9       close $fileId
10      exec iccr cov.cmd | grep DC > DC_log.txt
11      set fileId [open "DC_log.txt" r]
12      set log [read $fileId]
13      close $fileId
14      if {[regexp "DC: 100%" $log] == 1} {
15       puts "================================= Coverage completed
16     ================================="
17       quit
18      } else {
19       puts "Coverage $log"
20      }
21     }
```

**Código fonte do *template pattern tb_tcl* (Continuação)**

```
22      stop -create -condition {#tb.cov == 1'b1} -execute covering -continue
23      run
24      quit
25      $$(endfile)
```

## A.25 *tdriver*

**Código fonte do *template pattern tdriver***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(foreach) $$(module.in)
30      $$(file) $$("$$(i.name)_driver.svh")
31
32       class $$(i.name)_driver extends ovm_threaded_component;
33
34        ovm_get_port #($$(i.type)) $$(i.name)_from_source_port;
35        $$(i.type) p_$$(i.name);
36
37        function new(string name, ovm_component parent);
38         super.new(name,parent);
39         $$(i.name)_from_source_port = new("$$
40      (i.name)_from_source_port", this);
41        endfunction
42
43        task run();
44
45         delay d = new;
46         $$(i.name)_valid <= 0;
47
```

**Código fonte do *template pattern tdriver* (Continuação)**

```
48
49          @(posedge clk);
50          while(reset) @(posedge clk);
51          forever begin
52
53           $$(i.name)_from_source_port.get(p_$$(i.name));
54           p_$$(i.name).rec_begin(fiber_$$(foreach)$$(var)$$(i.name)$$
55       (endfor),"$$(i.name)");$$(foreach) $$(signal)
56           @(negedge clk);
57           $$(j.name)_$$(i.name) $$(endfor) <= p_$$(i.name).$$(foreach)$
58       $(var)$$(i.name)$$(endfor);
59           $$(i.name)_valid <= 1;
60           @(posedge clk);
61           while(!$$(i.name)_ready) @(posedge clk);
62           p_$$(i.name).rec_end();
63           $$(i.name)_valid <= 0;
64           assert(d.randomize());
65           repeat(d.d) @(posedge clk);
66          end
67        endtask
68       endclass
69
70       $$(endfor)
71       $$(endfile)
```

## A.26 *tdriver_duv*

**Código fonte do *template pattern tdriver_duv***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
```

**Código fonte do *template pattern tdriver_duv* (Continuação)**

```
28      $$(foreach) $$(module.out)
29      $$(file) $$("$$(i.name)_driver.svh")
30
31
32       class $$(i.name)_driver extends ovm_threaded_component;
33
34        ovm_get_port #($$(i.type)) $$(i.name)_from_duv_port;
35        $$(i.type) p_$$(i.name);
36
37        function new(string name, ovm_component parent);
38         super.new(name,parent);
39         $$(i.name)_from_duv_port = new("$$(i.name)_from_duv_port",
40      this);
41        endfunction
42
43        task run();
44
45         delay d = new;
46
47         $$(i.name)_valid <= 0;
48         while(reset) @(posedge clk);
49         forever begin
50          $$(i.name)_from_duv_port.get(p_$$(i.name));
51          p_$$(i.name).rec_begin(fiber_$$(foreach)$$(var)$$(i.name)$$
52      (endfor),"$$(i.name)");$$(foreach) $$(signal)
53          @(negedge clk);
54
55          $$(j.name)_$$(i.name) $$(endfor) <= p_$$(i.name).$$(foreach)$
56      $(var)$$(i.name)$$(endfor);
57
58          $$(i.name)_valid <= 1;
59          @(posedge clk);
60          while(!$$(i.name)_ready) @(posedge clk);
61          p_$$(i.name).rec_end();
62          $$(i.name)_valid <= 0;
63          assert(d.randomize());
64          repeat(d.d) @(posedge clk);
65
66         end
67        endtask
68       endclass
69
70      $$(endfor)
71      $$(endfile)
```

## A.27 *tmonitor*

**Código fonte do *template pattern tmonitor***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
```

**Código fonte do *template pattern tmonitor* (Continuação)**

```
8          $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(foreach) $$(module.out)
30      $$(file) $$("$$(i.name)_monitor.svh")
31
32       class $$(i.name)_monitor extends ovm_threaded_component;
33
34        ovm_put_port #($$(i.type)) $$(i.name)_to_checker_port;
35        $$(i.type) tr_$$(i.name);
36
37        covergroup cm;
38          $$(foreach)$$(var)
39          coverpoint tr_$$(j.name).$$(i.name) {
40            bins p[] = { 0 };
41            option.at_least = 1;
42          }
43          $$(endfor)
44        endgroup
45
46        function new(string name, ovm_component parent);
47         super.new(name,parent);
48         $$(i.name)_to_checker_port = new("$$(i.name)_to_checker_port",
49      this);
50         cm = new;
51        endfunction
52
53        task run();
54
55         @(posedge clk);
56         while(reset) @(posedge clk);
57         forever begin
58
59           while(!($$(i.name)_ready && $$(i.name)_valid)) @(posedge
60      clk);
61           tr_$$(i.name) = new();
62           tr_$$(i.name).rec_begin(fiber,"$$(i.name)");
63           tr_$$(i.name).$$(foreach)$$(var)$$(i.name)$$(endfor) = $$
64      (foreach) $$(signal) $$(j.name)_$$(i.name) $$(endfor);
65           cm.sample();
66           $$(i.name)_to_checker_port.put(tr_$$(i.name));
```

**Código fonte do *template pattern tmonitor* (Continuação)**

```
67          @(posedge clk);
68          tr_$$(i.name).rec_end();
69
70        end
71       endtask
72      endclass
73
74     $$(endfor)
75     $$(endfile)
```

## A.28 *tmonitor_duv*

**Código fonte do *template pattern tmonitor_duv***

```
1      $$(type.map) $$(bool->bit)
2      $$(type.map) $$(char->byte)
3      $$(type.map) $$(short->shortint)
4      $$(type.map) $$(short int->shortint)
5      $$(type.map) $$(int->int)
6      $$(type.map) $$(long->int)
7      $$(type.map) $$(long int->int)
8      $$(type.map) $$(long long->longint)
9      $$(type.map) $$(long long int->longint)
10     $$(type.map) $$(float->shortreal)
11     $$(type.map) $$(double->real)
12     $$(type.map) $$(signed->int)
13     $$(type.map) $$(unsigned->int unsigned)
14     $$(type.map) $$(signed int->int)
15     $$(type.map) $$(unsigned int->int unsigned)
16     $$(type.map) $$(signed short->shortint)
17     $$(type.map) $$(signed short int->shortint)
18     $$(type.map) $$(unsigned short->shortint)
19     $$(type.map) $$(unsigned short int->shortint)
20     $$(type.map) $$(signed long->int)
21     $$(type.map) $$(signed long int->int)
22     $$(type.map) $$(unsigned long->int)
23     $$(type.map) $$(unsigned long int->int)
24     $$(type.map) $$(signed long long->longint)
25     $$(type.map) $$(signed long long int->longint)
26     $$(type.map) $$(unsigned long long->longint)
27     $$(type.map) $$(unsigned long long int->longint)
28
29     $$(foreach) $$(module.in)
30     $$(file) $$("$$(i.name)_monitor.svh")
31
32      class $$(i.name)_monitor extends ovm_component;
33       ovm_put_port #($$(i.type)) $$(i.name)_to_duv_port;
34       $$(i.type) p_$$(i.name);
35
36       covergroup cmp;
37         $$(foreach)$$(var)
38         coverpoint p_$$(j.name).$$(i.name) {
39           bins p[] = { 0 };
40           option.at_least = 1;
41         }
42         $$(endfor)
```

**Código fonte do *template pattern tmonitor_duv* (Continuação)**

```
43        endgroup
44          }
45          $$(endfor)
46        endgroup
47
48        covergroup cmc;
49          $$(foreach)$$(var)
50          coverpoint p_$$(j.name).$$(i.name) {
51            bins p[] = { 0 };
52            option.at_least = 1;
53          }
54          $$(endfor)
55        endgroup
56
57        function new(string name, ovm_component parent);
58         super.new(name,parent);
59         $$(i.name)_to_duv_port = new("$$(i.name)_to_duv_port", this);
60         cmp = new;
61         cmc = new;
62        endfunction
63
64
65        task run();
66
67         @(posedge clk);
68         while(reset) @(posedge clk);
69         forever begin
70
71           while(!($$(i.name)_ready && $$(i.name)_valid)) @(posedge
72      clk);
73           p_$$(i.name) = new();
74           p_$$(i.name).rec_begin(fiber_$$(foreach)$$(var)$$(i.name)$$
75      (endfor),"$$(i.name)");
76           p_$$(i.name).$$(foreach)$$(var)$$(i.name)$$(endfor) = $$
77      (foreach) $$(signal) $$(j.name)_$$(i.name) $$(endfor);
78           cmp.sample();
79           cmc.sample();
80           $$(i.name)_to_duv_port.put(p_$$(i.name));
81           @(posedge clk);
82           p_$$(i.name).rec_end();
83         end
84
85        endtask
86
87       endclass
88
89      $$(endfor)
90      $$(endfile)
```

## A.29 *top*

**Código fonte do *template pattern top***

```
1       $$(type.map) $$(bool->bit)
2       $$(type.map) $$(char->byte)
3       $$(type.map) $$(short->shortint)
```

**Código fonte do *template pattern top* (Continuação)**

```
4       $$(type.map) $$(short int->shortint)
5       $$(type.map) $$(int->int)
6       $$(type.map) $$(long->int)
7       $$(type.map) $$(long int->int)
8       $$(type.map) $$(long long->longint)
9       $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28
29      $$(file) $$("top.sv")
30
31      module top;
32
33       logic reset,clk;
34
35       $$(foreach) $$(module.in)
36       $$(foreach) $$(signal)
37       logic $$(i.type) [$$(i.size):1] $$(j.name)_$$(i.name);
38       $$(endfor)
39       logic $$(j.name)_valid, $$(j.name)_ready;
40       $$(endfor)
41
42       $$(foreach) $$(module.out)
43       $$(foreach) $$(signal)
44       logic $$(i.type) [$$(i.size):1] $$(j.name)_$$(i.name);
45       $$(endfor)
46       logic $$(j.name)_valid, $$(j.name)_ready;
47       $$(endfor)
48
49       gene_clock gene_clock_i(.*);
50       tb tb_i(.*);
51       $$(module.name) $$(module.name)_i(.*);
52
53       initial begin
54        $shm_open("waves.shm");
55        $shm_probe(reset,clk,$$(foreach)$$(module.in)$$(foreach)$$
56      (signal)
57              $$(j.name)_$$(i.name),$$(endfor)
58              $$(j.name)_ready,
59              $$(j.name)_valid,$$(endfor)$$(foreach)$$(module.out)$$
60      (foreach)$$(signal)
61              $$(j.name)_$$(i.name),$$(endfor)
62              $$(j.name)_ready,
```

**Código fonte do *template pattern top* (Continuação)**

```
63              $$(j.name)_valid$$(if)$$(isnotlast),$$(endif)$$
64      (endfor) );
65       end
66
67       logic cov;
68       initial cov = 0;
69       always #10us cov = !cov;
70
71      endmodule
72      $$(endfile)
```

## A.30 *top_analysis*

**Código fonte do *template pattern top_analysis***

```
 1      $$(type.map) $$(bool->bit)
 2      $$(type.map) $$(char->byte)
 3      $$(type.map) $$(short->shortint)
 4      $$(type.map) $$(short int->shortint)
 5      $$(type.map) $$(int->int)
 6      $$(type.map) $$(long->int)
 7      $$(type.map) $$(long int->int)
 8      $$(type.map) $$(long long->longint)
 9      $$(type.map) $$(long long int->longint)
10      $$(type.map) $$(float->shortreal)
11      $$(type.map) $$(double->real)
12      $$(type.map) $$(signed->int)
13      $$(type.map) $$(unsigned->int unsigned)
14      $$(type.map) $$(signed int->int)
15      $$(type.map) $$(unsigned int->int unsigned)
16      $$(type.map) $$(signed short->shortint)
17      $$(type.map) $$(signed short int->shortint)
18      $$(type.map) $$(unsigned short->shortint)
19      $$(type.map) $$(unsigned short int->shortint)
20      $$(type.map) $$(signed long->int)
21      $$(type.map) $$(signed long int->int)
22      $$(type.map) $$(unsigned long->int)
23      $$(type.map) $$(unsigned long int->int)
24      $$(type.map) $$(signed long long->longint)
25      $$(type.map) $$(signed long long int->longint)
26      $$(type.map) $$(unsigned long long->longint)
27      $$(type.map) $$(unsigned long long int->longint)
28      $$(file) $$("top.sv")
29
30      module top;
31       logic reset,clk;
32
33       $$(foreach) $$(module.in)
34       $$(foreach) $$(signal)
35       logic $$(i.type) [$$(i.size):1] $$(j.name)_$$(i.name);
36       $$(endfor)
37       logic $$(j.name)_valid, $$(j.name)_ready;
38       $$(endfor)
39
40       $$(foreach) $$(module.out)
41       $$(foreach) $$(signal)
```

**Código fonte do *template pattern top_analysis* (Continuação)**

```
42      logic $$(i.type) [$$(i.size):1] $$(j.name)_$$(i.name);
43      $$(endfor)
44      logic $$(j.name)_valid, $$(j.name)_ready;
45      $$(endfor)
46
47      gene_clock gene_clock_i(.*);
48      tb tb_i(.*, $$(foreach)$$(module.channel)$$(foreach)$$(signal)
49                  .$$(j.name)_$$(i.name)($$(module.name).$$(j.name)_$$
50     (i.name) ),
51                  .$$(j.name)_valid($$(module.name).$$(j.name)_valid),
52                  .$$(j.name)_ready($$(module.name).$$(j.name)_ready)$$
53     (if)$$(isnotlast),$$(endif)$$(endfor));
54      $$(module.name) $$(module.name)_i(.*);
55
56      initial begin
57       $shm_open("waves.shm");
58       $shm_probe(reset,clk,$$(foreach)$$(module.in)$$(foreach)$$
59     (signal)
60              $$(j.name)_$$(i.name),$$(endfor)
61              $$(j.name)_ready,
62              $$(j.name)_valid,$$(endfor)$$(foreach)$$(module.out)$$
63     (foreach)$$(signal)
64              $$(j.name)_$$(i.name),$$(endfor)
65              $$(j.name)_ready,
66              $$(j.name)_valid$$(if)$$(isnotlast),$$(endif)$$
67     (endfor) );
68      end
69
70      logic cov;
71      initial cov = 0;
72      always #10us cov = !cov;
73
74     endmodule
75     $$(endfile)
```

## A.31 *top_tcl*

**Código fonte do *template pattern top_tcl***

```
1      $$(file)$$("top.tcl")set idmp 0
2      proc covering {} {
3       upvar 1 idmp idmp
4       rm -rf ./cov_work/design/$idmp
5       set idmp [expr {$idmp+1}]
6       coverage -dump $idmp
7       set fileId [open "cov.cmd" w]
8       puts $fileId "load_test $idmp\nreport_summary -cgopt top\nquit\n\n"
9       close $fileId
10      exec iccr cov.cmd | grep DC > DC_log.txt
11      set fileId [open "DC_log.txt" r]
12      set log [read $fileId]
13      close $fileId
14      if {[regexp "DC: 100%" $log] == 1} {
15       puts "================================= Coverage completed
16     ================================="
17       quit
```

**Código fonte do *template pattern top_tcl* (Continuação)**

```
18       } else {
19         puts "Coverage $log"
20       }
21     }
22     stop -create -condition {#top.cov == 1'b1} -execute covering
23     -continue
24     run
25     quit
26     $$(endfile)
```

## A.32 *trans*

**Código fonte do *template pattern trans***

```
1      $$(type.map) $$(short->shortint)
2      $$(type.map) $$(short int->shortint)
3      $$(type.map) $$(int->int)
4      $$(type.map) $$(long->int)
5      $$(type.map) $$(long int->int)
6      $$(type.map) $$(long long->longint)
7      $$(type.map) $$(long long int->longint)
8      $$(type.map) $$(float->shortreal)
9      $$(type.map) $$(double->real)
10     $$(type.map) $$(signed->int)
11     $$(type.map) $$(unsigned->int unsigned)
12     $$(type.map) $$(signed int->int)
13     $$(type.map) $$(unsigned int->int unsigned)
14     $$(type.map) $$(signed short->shortint)
15     $$(type.map) $$(signed short int->shortint)
16     $$(type.map) $$(unsigned short->shortint)
17     $$(type.map) $$(unsigned short int->shortint)
18     $$(type.map) $$(signed long->int)
19     $$(type.map) $$(signed long int->int)
20     $$(type.map) $$(unsigned long->int)
21     $$(type.map) $$(unsigned long int->int)
22     $$(type.map) $$(signed long long->longint)
23     $$(type.map) $$(signed long long int->longint)
24     $$(type.map) $$(unsigned long long->longint)
25     $$(type.map) $$(unsigned long long int->longint)
26
27     $$(file) $$("trans.svh")//Code generated by eTBc software
28
29     import ovm_pkg::*;
30     $$(foreach) $$(struct)
31     class $$(i.name) extends ovm_transaction;
32       $$(foreach) $$(var)
33       rand $$(i.type) $$(i.name);
34       constraint $$(i.name)_range {
35        $$(i.name) dist { [0:1] };
36       }
37       $$(endfor)
38       function integer equal($$(i.name) tr);
39        equal = $$(foreach)$$(var)(this.$$(i.name) == tr.$$(i.name))$$
40     (if)$
41     $(isnotlast) && $$(endif)$$(endfor);
42       endfunction
```

**Código fonte do *template pattern trans* (Continuação)**

```
43        function string psprint();
44         psprint = $psprintf("($$(foreach)$$(var)$$(i.name)= %d)",$$
45      (i.name)
46      $$(if)$$(isnotlast), $$(endif)$$(endfor));
47        endfunction
48
49        function bit read(int file);
50         if (file && !$feof(file))
51          read = 0 != $fscanf(file, "$$(foreach)$$(var)%d $$(endfor)",
52      $$
53      (foreach)$$(var)this.$$(i.name)$$(if)$$(isnotlast), $$(endif)$$
54      (endfor));
55         else read = 0;
56        endfunction
57
58        function void do_record (ovm_recorder recorder);
59         $$(foreach) $$(var)recorder.record_field ("$$(i.name)", $$
60      (i.name),
61      $bits($$(i.name)), OVM_HEX);$$(endfor)
62        endfunction
63
64      endclass
65      $$(endfor)
66
67      class delay; // used for delay in signal handshake
68        rand int d;
69        constraint delay_range {
70         d dist {0 :/ 8, [1:2] :/2, [3:10] :/1 };
71        }
72      endclass
73
74      $$(endfile)
```

# Apêndice B

# An OVM based Functional Verification Methodology with Testbench Generator

Helder Fernando, Isaac Maia, Elmar U. K. Melcher

SBCCI 2009, Chip on the Dunes

## Abstract

The process of building a complex digital circuit is composed of several steps, one of them is functional verification. This step can be considered one of the most important, because it aims to demonstrate that the functionality of the circuit to be produced conforms with its specification. This step consumes about 70% of project resources and functional verification complexity grows as the hardware becomes more complex. Thus the use of an effective functional verification methodology and tools that help the functional verification engineer are very important. This paper presents a new methodology for functional verification called BVM (Brazil-IP Verification Methodology), which is based on OVM, along with the eTBc (Easy Testbech Creator) tool that helps creating BVM testbenches. BVM is compared with VeriSC methodology showing a 20% increase in productivity.

## B.1 Introduction

Functional verification is a process used to demonstrate that the project objective is preserved in its implementation[3]. That means if the Functional verification is good then the final system conforms with the specification. VeriSC[5, 7] is a methodology used for functional verification. It was used to do the functional verification of some chips developed by the Brazil-IP network. Brazil-IP emerged in 2001 as a collaborative effort among Brazilian institutions to teach design of Intellectual Property cores (IP-cores). Until 2006, three IP cores were developed: a MPEG-4 video decoder , a MP3 audio decoder and a 8051 microcontroller, all three functioning in their first prototype. The UFCG (Universidade Federal de Campina Grande) was responsible to develop the

MPEG-4 video decoder. This IP-core is the most complex digital chip developed by a Brazilian institution until nowadays. Actually, the Brazil-IP portfolio is composed by 21 chips, wich include controllers, memories, microprocessors and others. According with [8] 65% of IP cores fail at first silicon prototyping, and 70% of these cases occur due to a bad functional verification. It is estimated that about 70% of the time a project of hardware is spent at this stage. That is why there is a need for tools and methodologies for effective verification, that help the engineer in this phase of the project, reducing time, risk and money necessary for its implementation. Many others methodologies are available: VMM (Verification Methodology Manual)[2, 17], OVM (Open Verification Methodology)[11, 4], AVM (Advanced Verification Methodology)[10] , IPCM (Incisive Plan-to-Closure Methodology) [19, 20], (RVM) OpenVera Reference Verification Methodology[17, 18] and others. The success of VeriSC in the functional verification of IPs produced by Brazil-IP can be presented as the main motivation for the new methodology BVM (Brazil-IP Verification Methodology) being a reimplementation of VeriSC using Systemverilog[1] and OVM (Open Verification Methodology)[11]. In order to speed up the testbench construction process, both VeriSC and BVM use the tool is eTBc (Easy Testbech Creator)[13, 14]. eTBc creates templates of testbenches and reduces the time compared to building the entire verification setup by hand. This paper presents the BVM methodology, and its benefits compared to VeriSC through observing and interviewing users. Section 2 and 3 explains the VeriSC and BVM methodologies. Section 4 discusses eTBc. Finally section 6 shows a comparison between both methodologies.

## B.2 VeriSC methodology

The methodology VeriSC describes concepts and models of the Brazil-IP verification methodology. In [7, 5] it is possible to obtain complete explanation about VeriSC. It uses the SystemC, SystemC Verification Library (SCV)[12] and the library BVE COVER to create random-constraint, coverage-driven, self-checking and transaction-based testbenches [6]. VeriSC enables the creation of the testbench before the DUV (Design Under Verification) without the need for any extra code that is not reused in the final testbench. Therefore, the methodology reduces the total time to verify and allows testbench to be debugged before the DUV is implemented. Important characteristics of VeriSC methodology are:

a) Testbenches must be built in SystemC[12] and SCV

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.

A testbench created using VeriSC is composed by some basic blocks: Source, TDriver, Reference model (RM), TMonitor and Checker. They are all connected by FIFOs. The DUV is connected into the testbench by signals. The testbench representation is shown in Figure 1.

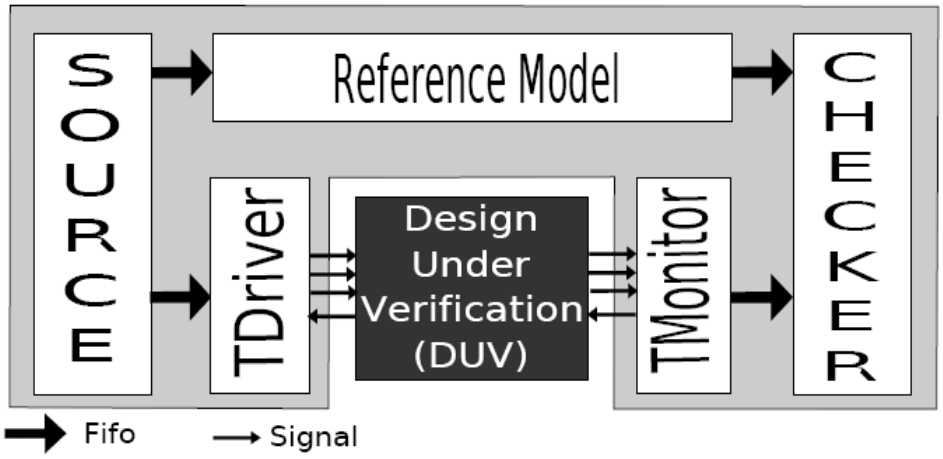


**Figure 1: Testbench representation using VeriSC methodology**

- 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(Reference Model) 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 cite-brahme2000[15][21][16] and according to VSIA [9], 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 will not be reused.

## B.3 BVM methodology

The BVM methodology is a re-implementation of VeriSC, but it is based in Systemverilog[1] and OVM instead of SystemC[12] and SCV. BVM aims to achieve the same performance of VeriSC, on verification of IP-cores. The decision of re-implementing VeriSC with SystemVerilog was made because of the growing popularity of SystemVerilog for functional verification at the expense of SystemC. Simulation and synthesis from Systemverilog is nowadays being supported by almost all tool vendors. SystemVerilog is a combined Hardware Verification Language and Hardware Description Language based on extensions to Verilog. OVM offers all the structural elements needed for functional verification, is non-vendor specific and interoperable with simulators from different vendors. BVM methodology is composed by all blocks mentioned in VeriSC methodology: source, TDriver, reference model(RM), TMonitor and checker. However, BVM methodology added a new block, the Actor, in order to conform better to OVM methodology.

- Actor: The actor is placed between two blocks wich exchange signals: DUV and TMonitor. The Actor element was added according with the Responder concept presented in OVM methodology. Its responsible to control the signals protocol. The testbench representation is shown in Figure 2

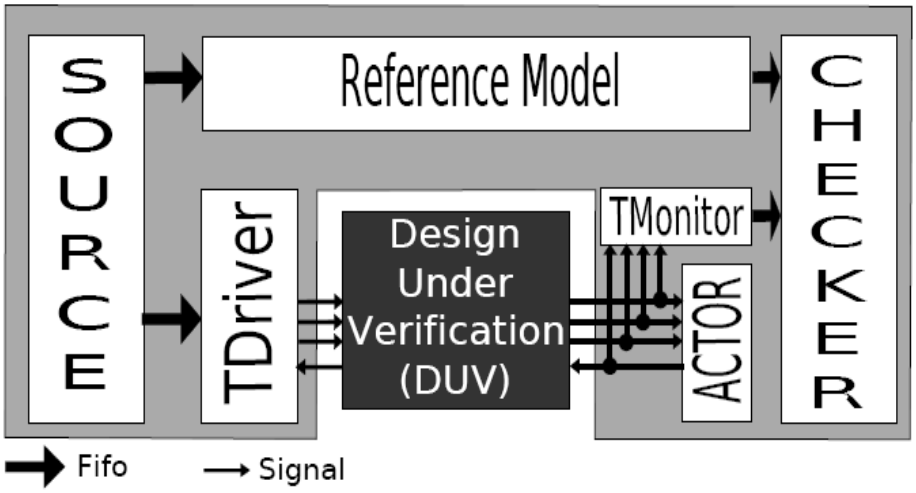


**Figure 2: Testbench representation using the BVM methodology**

Like VeriSC, BVM enables the creation of the testbench before the DUV without the need of any extra code to be written that is not reused in the final testbench.

## B.4 eTBc tool

The eTBc(Easy Testbench Creator)[13, 14] tool implements all of the BVM and VeriSC steps. With this tool one can generate testbench elements in an incremental fashion and end up with a complete testbench environment. In order to accomplish this task, the engineer needs to define a hierarchical TL(Transaction Level) structure of the design in the tool specific eDL language (eTBc Design Language). The tool speeds up the verification process generating complete sets of testbench templates for both VeriSC and BVM methodologies. The eTBc tool receives as user input a transaction level netlist (TLN) written in eDL and template patterns written in eTL (eTBc Template Language) written by developers of eTBc or by the Verification Manager. Each one is described bellow:

- Transaction Level Netlist (TLN): Is a le described by the functional verification engineer (user of eTBc). This le is a model of the IP-core using Transaction data and RTL data. The RTL data used in this level is only the name of modules I/O ports. The remaining of the description is only transaction data. The language used in this level is eTBc Design Language (eDL). eDL is a simple language used to describe modules, connections, FIFOs and some data in the RTL level.

- Template Pattern: Is a model of the testbench element that will be generated. The eTBc template pattern is a way that eTBc works to generate testbench elements. The role of templates is to guide eTBc to generate code, based in one TLN. The TLN defines the

model of the system(transactions and signals) and the templates defines the model of testbench.

The TLN is specific to the circuit being designed whereas the template patterns are specific to the methodology and language used. The output of eTBc is a template for specified elements from the testbench: Source, TDriver, Tmonitor , checker, Actor and Reference Model. Templates for all testbench elements can be created. The templates are guaranteed to compile and simulate without run-time errors provided there is no bug in the eTBc core nor in the template patterns. However, the templates lack function specific code, such as the implementation of the reference model functionality, handshake implementation, for example. This code the verification engineer has to insert into the templates by hand. The eTBc tool architecture is reported in Figure 3.

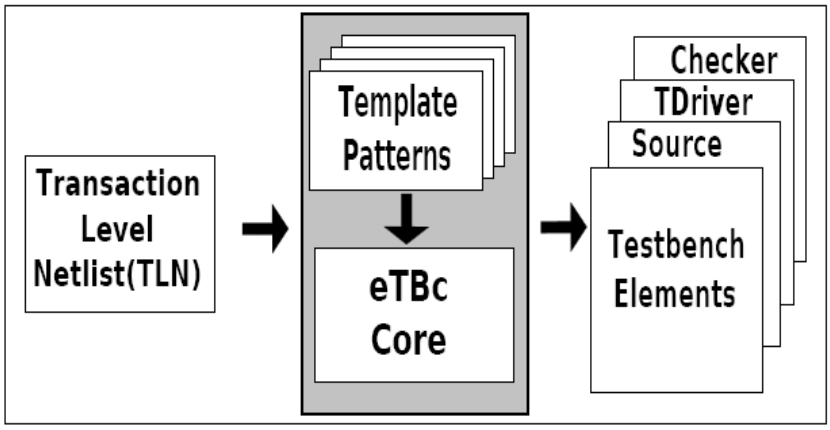


**Figure 3: eTBc tool - Architecture Diagram**

The template patterns shown in the Figure 4 and 5 are ASCII files that contain the SystemC or Systemverilog code that are common to any template instance of a testbench in eTL. The idea of template patterns is to guide eTBc to generate the output template. In Figure 4 and 5 each eTBc keyword delimited by $$( __ ) is an entry in eTBc Template Language(eTL). When eTBc parses these files it will replace this entries by some code based in some TLN definitions on the project file.

**VeriSC - Piece of template pattern for stimulus generator**

```
 ...
$$(foreach) $$(module.in)
class $$(i.name)_constraint_class: public scv_constraint_base {
  $$(foreach) $$(var)
    scv_bag<pair<$$(i.type),$$(i.type)> > $$(i.name)_distrib;
  $$(endfor)
public:
  scv_smart_ptr<$$(i.type)> $$(i.type)_sptr;
  SCV_CONSTRAINT_CTOR($$(i.name)_constraint_class) {
  $$(foreach) $$(var)
  // $$(i.name)_distrib.push(pair<$$(i.type),$$(i.type)>(?,?), ?);
  // $$(j.type)_sptr->$$(i.name).set_mode($$(i.name)_distrib);
  $$(endfor)
  //  SCV_CONSTRAINT($$(i.type)_sptr->?() > ?);
  }
};  ...
```

**Figure 4: Piece of template pattern for stimulus generator in VeriSC methodology**

**BVM - Piece of template pattern for stimulus generator**

```
 ...
 class $$(i.name) extends ovm_object;
   $$(foreach) $$(var)
   rand $$(i.type) $$(i.name);
   constraint $$(i.name)_range {
     //$$(i.name) dist { [?:?] };
   }
   $$(endfor)
 ...
```

**Figure 5: Piece of template pattern for for stimulus generator in BVM methodology**

For each design project, verification and design engineers need to create a TLN file. In this file are defined all the TL(Transaction Level) information and signal interfaces and how the individual modules of the design are linked together at transaction level to form the top level DUV[13,15].

## B.5 BVM x VeriSC

Many differences between VeriSC and BVM can be presented. A remarkable difference

between BVM and VeriSC is how coverage is done. VeriSC uses the BVE (Brazil-IP Verification Extensions)[5] cover class. This class was developed by members of Brazil-IP in order to to implement functional coverage. BVM uses native Systemverilog features to perform it. In figure 6 is shown a VeriSC cover criterion that only permit to stop the simulation after reaching 15 times the number 0, 15 times the number 1, and 15 times a value between 3 and 7 using the class BVE. In figure 7 is show the same criterion using Systemverilog.

```
output_checker_cv.begin();
  BVE_COVER_COND(output_checker_cv, x==0, 15);
  BVE_COVER_COND(output_checker_cv, x==1, 15);
  BVE_COVER_COND(output_checker_cv, x==2, 15);
  BVE_COVER_COND(output_checker_cv, x>=3 && x<=7, 15);
output_checker_cv.end();
```

**Figure 6: Functional Coverage in VeriSC**

```
covergroup cg;
  coverpoint tr_output_checker.value {
    bins tr_x[] = { 0, 1, 2, [3:7]  };
    option.at_least = 15;
  }
endgroup
```

**Figure 7: Functional Coverage in BVM**

Different of VeriSC, in BVM methodology the developer was not concerned to create a class to perform the functional coverage. Systemverilog ensures a more effcient way to perform it. The OVM library was used to create testbenches in BVM methodology. It offers a simply and efficient way to connect and use testbench elements by tlm fifo. In VeriSC the SCV library was used to generate stimuli. Constraints are declared to generate random values. An example is shown in Figure 8. It generates a random values between 0 and 10, and 15 and 40, weighted ratio of 15 and 20 respectively. Systemverilog allows users to specify constraints in a compact, declarative way. They are then processed by a solver that generates random values that meet the constraints. The same example is shown in Figure 9.

```
class x_constraint_class: public scv_constraint_base {
  scv_bag<pair<int,int> > x_distrib;
public:
  scv_smart_ptr<int> data;
  x_distrib.push( pair<int,int>(0,10),  15);
  x_distrib.push( pair<int,int>(15,40), 20);
  data->set_mode(x_distrib);
};
```

**Figure 8: Constraint to generate stimulus in VeriSC**

```
class x;
  rand int x_distrib;
  constraint x_distrib_range {
    x_distrib dist { [0:10] := 15, [15:40] := 20 };
  }
endclass
```

**Figure 9: Constraint to generate stimulus in BVM**

Other difference between BVM and VeriSC is how to Reference Model(RM) is used. In VeriSC and BVM the Reference Model(RM) is ofter written in C/C++ and this is an advantage in VeriSC because it uses SystemC that works in C/C++ environment. A function call can be done directly by SystemC from the RM. In BVM this task has to be performed by using DPI(Direct Programming Interface) that enables C/C++ function call from Systemverilog environment.

## B.6 Results

In 2007, Brazil-IP offered training for a group of 16 students to teach VeriSC methodology. The students were oriented to develop a little project using VeriSC. For all theses students, it was the first contact with a functional verification methodology. The project consisted of a DPCM (Differential Pulse Code Modulation). DPCM does digital signal coding by calculating differences between subsequent samples and saturating. The DPCM implementation is hierarchically divided into a difference and a saturation module. The project consisted in testbench creation, RTL implementation and functional verification. At end of the training, a questionnaire was applied to them, to know how good VeriSC was for them. In the same way, in 2008, Brazil- IP offered another training, but with BVM methodology. A different group of 19 students was oriented to do the same project used in 2007, but applying BVM. At the end of the training the same questionnaire was used to collect the opinion of theses students. BVM was the first contact with a functional verification methodology for this group too. Thus, the answers collected from all this users were filtered and examined to make a comparison between both methodologies. The questions applied are shown below in Table 1.

| Questions |
|---|
| 1. How you classify the methodology (very bad, bad, good, very good, excellent)? |
| 2. The methodology is easy to understand (yes, no) ? |
| 3. The methodology is easy to apply(yes, no)? |
| 4. The language used by this methodology was easy to understand?(yes, no) |
| 5. The language used by this methodology was easy to apply(yes, no)? |
| 6. How helpful you classify the eTBc to construct automatic testbenches for this methodology (very bad, bad, good, very good, excellent)? |
| 7. Quote the most difficult thing about this methodology |

**Table 1 – Questions about BVM and VeriSC**

The answers from the questionnaire applied to students showed how easily them could to adapt themselves to BVM or VeriSC. Seven tables are shown below, each referring to a question from the questionnaire.

| Question 1: How you classify the methodology (very bad, bad, good, very good, excellent)? | | | | | | |
|---|---|---|---|---|---|---|
| | Very bad | Bad | Good | Very good | Excellent | Didn´t answer |
| Students of VeriSC | 0% | 0% | 87,5% (14 students) | 6,25% (1 student) | 0% | 6,25% (1 student) |
| Students of BVM | 0% | 0% | 68,42% (13 students) | 10,53% (2 students) | 0% | 21,05% (4 students) |

**Table 2 – Answers about question 1**

As can be seen on the table 2, in the question number one, the students opinion about the methodologies are located between good and very good. Almost all think the methodology studied is good or very good.

| Question 2: The methodology is easy to understand? | | | |
|---|---|---|---|
| | Yes | No | Didn´t answer |
| Students of VeriSC | 93,75% (15 students) | 0% | 6,25% (1 student) |
| Students of BVM | 100% (19 students) | 0% | 0% |

**Table 3 – Answers about question 2**

| Question 3: The methodology is easy to apply? | | | |
|---|---|---|---|
| | Yes | No | Didn´t answer |
| Students of VeriSC | 68,75% (11 students) | 31,25% (5 students) | 0% |
| Students of BVM | 84,21% (16 students) | 15,79% (3 students) | 0% |

**Table 4 – Answers about question 3**

For almost all students the methodology studied (VeriSC or BVM) is easy to understand, Table 3. But by the table 4, some differences about the facility to apply this methodologies are

visible. For the students, BVM appears easier to apply.

| Question 4: The language used by this methodology was easy to understand? | | | |
|---|---|---|---|
| | Yes | No | Didn´t answer |
| Students of VeriSC | 75% (12 students) | 18,75% (3 students) | 6,25% (1 student) |
| Students of BVM | 89,47% (17 students) | 10,53% (2 students) | 0% |

**Table 5 – Answers about question 4**

In Table 5 is shown some differences between the languages used to create testbenches in VeriSC and BVM. VeriSC is based in SystemC and BVM based in Systemverilog. By the Table, is correct to say that 14,47% more of students think Systemverilog is easier to understand.

| Question 5: The language used by this methodology was easy to apply? | | | |
|---|---|---|---|
| | Yes | No | Didn´t answer |
| Students of VeriSC | 75% (12 students) | 25% (4 students) | 0% |
| Students of BVM | 84,21% (16 students) | 15,79% (3 students) | 0% |

**Table 6 – Answers about question 5**

The answers presented in Table 6 are very similar to the answers presented in Table 4. This indicates that the facility to apply the methodology is associated with the language used to create its testbenches.

| Question 6: How helpful you classify the eTBc to construct automatic testbenches for this methodology (very bad, bad, good, very good, excellent)? | | | | | | |
|---|---|---|---|---|---|---|
| | Very bad | Bad | Good | Very good | Excellent | Didn´t answer |
| Students of VeriSC | 0% | 0% | 93,75% (15 students) | 6,25% (1 student) | 0% | 0% |
| Students of BVM | 0% | 0% | 47,37% (9 students) | 42,1% (8 students) | 10,53% (2 students) | 0% |

**Table 7 – Answers about question 6**

For all students, the eTBc tool is felt to be very helpful to create automatically testbench elements. The eTBc tool speed up the testbench creation process.

| Question 7: Quote the worst thing about this methodology | | | | |
|---|---|---|---|---|
| | The language used | The methodology have many steps | Other | Didn´t answer |
| Students of VeriSC | 25% (4 students) | 62,5% ( 10 students) | 12,5% (4 students) | 0% |
| Students of BVM | 0% | 84,2% ( 16 students) | 15,78% ( 3 students) | 0% |

**Table 8 – Answers about question 7**

For the last question, many students think the methodology VeriSC and BVM have many steps to perform the functional verification. For VeriSC students, 25% think the language used to create the testbench is the worst thing in this methodology. BVM students are satisfied with Systemverilog language. The time spent to finish the project using BVM was about 20% less than using the Verisc methodology. At the training with VeriSC students spent about 28 hours to finish the project. BVM students spent about 22 hours to finish it.

## B.7 Conclusion

BVM showed 20% increase in productivity compared with VeriSC methodology. Based on the questionnaires, it is possible to see that BVM is easier to apply and understand compared to VeriSC. The language used to create testbenchs in VeriSC can be considered a hurdle. All the students considered eTBc tool helpful to generate automatic testbenches for both methodologies. The eTBc tool facilitate a lot the testbench creation. If the students of both methodologies would not have used eTBc the results shown in the tables would certainly be very different. The automatic creation of testbenchs mask some of the difficulty to create a testbench in VeriSC. The language used for VeriSC methodology, the library to perform functional coverage, and the way the testbenches components are connected in this methodology is much more complex than BVM. BVM methodology is easy to understand and easy to apply. For this aspects BVM reduces time, risk and resources of functional verification. BVM is now used to verify the Brazil-IP designs currently under development so in the future comparison data based on bigger designs will be available.

## References

[1] I. Acellera Organization. Systemverilog 3.1a language reference manual accelleras extensions to verilog, 2004.

[2] ARM and Synopsys. http://www.vmm-sv.org/.

[3] J. Bergeron, editor. *Writing Testbenches*. Springer, Boston, 2003.

[4] I. CADENCE DESIGN SYSTEMS, INC; MENTOR GRAPHICS. Open verication methodology user guide. Technical report, EUA, 2008.

[5] K. R. G. da Silva. *Uma metodologia de Verificacão Funcional para circuitos digitais*. PhD thesis,Universidade Federal de Campina Grande, Av. Aprigio Veloso, Campina Grande, 2007.

[6] K. R. G. da Silva, E. U. K. Melcher, G. Araujo, and V. A. Pimenta. An automatic testbench

generation tool for a systemc functional verification methodology. In *SBCCI '04*, pages 66-70, 2004.

[7] 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.

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

[9] V. A. Functional Verification Development Group. Specification for vc/soc functional verification version 1.0 (ver 21.0). 2004.

[10] M. Graphics. http://www.mentor.com.

[11] M. Graphics and Cadence. http://www.ovmworld.org/.

[12] O. S. Initiative. http://www.systemc.org.

[13] I. Maia, K. R. G. da Silva, L. Max, R. Camara, and E. U. K. Melcher. etbc: A semi-automatic testbench generation tool. 2007.

[14] I. M. Pessoa. Geração semi-automatica de testbenches para circuitos integrados digitais. Master's thesis, Universidade Federal de Campina Grande, Av. Aprigio Veloso, Campina Grande, 2007.

[15] 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.

[16] P. Rashinkar, P. Paterson, and L. Singh, editors. *System-on-a-Chip Verication Methodology and*

*Techniques*. Kluwer Academic Publishers, Massachusetts, 2001.

[17] Synopsys. http://www.synopsys.com.

[18] Synopsys. http://inter.viewcentral.com/events/uploads/Synopsys/openverarvm_fcd.html.

[19] C. D. Systems. http://www.cadence.com/products/fv/plan_to_closure.

[20] C. D. Systems. http://www.cadence.com.

[21] I. Wagner, V. Bertacco, and T. Austin. Stresstest: an automatic approach to test generation via activity monitors. In *DAC '05*, pages 783-788, New York, NY, USA, June 2005