Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciências da Computação

OVM_tpi: Uma Metodologia de Verificação Funcional Para Circuitos Digitais

Rômulo Calado Pantaleão Camara

Recife

31 de maio de 2011

Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciências da Computação

Rômulo Calado Pantaleão Camara

OVM_tpi: Uma Metodologia de Verificação Funcional Para Circuitos Digitais

Trabalho apresentado ao programa de pós-graduação em Ciências da Computação da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de mestre em Ciências da Computação sob orientação da professora Edna Natividade da Silva Barros.

Av. Prof. Moraes Rego, 1235 - Cidade Universitária, Recife - PE - CEP: 50670-901

Recife, 31 de maio de 2011

Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciências da Computação

Dissertação de Mestrado

Título:

OVM_tpi: Uma Metodologia de Verificação Funcional Para Circuitos Digitais

Autor:
Rômulo Calado Pantaleão Camara

Edna Natividade da Silva Barros
Orientadora
Universidade Federal de Pernambuco

Cristiano Coelho de Araújo
Banca Interna
Universidade Federal de Pernambuco

Elmar Uwe Kurt Melcher
Banca Externa
Universidade Federal de Campina Grande

Recife
31 de maio de 2011

Dedico esse trabalho a meu irmão Ronaldo Calado Pantaleão Camara (In Memorian), a minha família e aos professores que me ajudaram e incentivaram até o final desse longo caminho.

O destino conduz os que consentem,
E arrasta os que resistem.
(Lucius Annaeus Seneca)

Um raciocínio lógico leva você de A a B.
A imaginação leva você a qualquer lugar
que você quiser.
(Albert Einstein)

Agradecimentos

Por início, gostaria de agradecer à minha família, tios, tias, primos, meu irmão Rafael e principalmente meu irmão Ronaldo, que comprou meu primeiro computador na graduação e me cobrava como retribuição resultados positivos na Universidade, aos meus pais Aparecida e Raimundo os maiores incentivadores do caminho que segui. Nunca esquecerei os almoços de todos os dias que minha mãe fazia, nem as buscadas no ponto de ônibus tarde da noite que meu pai foi sem reclamar. Muito obrigado por tudo que fizeram para mim!

Agradeço também à família Ribeiro Morais e, em especial, minha namorada, Laise Ribeiro Morais, por todo amor, carinho, compreensão e paciência com alguns momentos durante esse período de mestrado, inclusive o ano de treinamento em Porto Alegre que permaneci longe dela. Você foi e sempre será muito especial para mim!

Venho também agradecer aos professores e profissionais que colaboraram com minha caminhada. Em especial gostaria de mostrar minha gratidão ao Presidente da empresa SiMantis, INC. Sasan Iman pela colaboração com seus conhecimentos e seu livro, a minha orientadora Dra Edna Natividade da Silva Barros e aos professores Dr. Elmar Uwe Kurt Melcher e Dr. Cristiano Coelho Araújo que sempre me incentivaram a fazer o treinamento na primeira turma do CI-Brasil com grandes professores qualificados. Aos três professores meu muito obrigado e que seja apenas um marco para toda uma vida de contribuições e pesquisa.

Gostaria de agradecer aos meus colegas do LINCS (Laboratório para a Integração de Circuitos e Sistemas) em especial Leonardo Nunes e Marcos Souza, pela contribuição com o trabalho, e do LAD-UFCG (Laboratório de Arquiteturas Dedicadas), por todo apoio técnico e pessoal durante esse período.

Agradeço também aos grandes amigos Leandro Max, Flávio Santos, Rogério Silva, Rafael Cantalice e todos da pensão do Eraldo e do Marcelo pelas cervejas e churrascos, e pela grande amizade quando estava longe da família.

Por fim, agradeço a todos aqueles que fizeram parte do grupo de digital do treinamento CI-Brasil fases 1 e 2, em especial Wagston Tassoni Staehler e sua esposa Fernanda, ao eterno ***team 2*** (Arquilho Silva, Cláudio Dreher e Érico Sawabe) e aos instrutores da fase 2 Patrick McKeever e Murugappan Ramaswami, o grande Muru.

A todos vocês - e àqueles que não me lembrei - meu cordial obrigado!

Resumo

O advento das novas tecnologias *Very Large Scale Integration* (VLSI) e o crescimento da demanda por produtos eletrônicos no mundo estão trazendo um aumento explosivo na complexidade dos circuitos eletrônicos. A *contrario sensu*, o tempo de mercado (*time-to-market*) de um produto eletrônico, e o tempo de projeto necessário para produção e venda de um sistema estão ficando cada vez menores. Para que o circuito integrado chegue ao mercado com o funcionamento esperado é necessário realizar testes. Parte desses testes é chamada de verificação funcional e é a parte do projeto que requer mais tempo de desenvolvimento.

Buscam-se sempre novos métodos que permitam que a verificação funcional seja realizada de forma ágil, fácil e que proveja uma maior reusabilidade e diminuição da complexidade na construção do ambiente de simulação, sem interferir negativamente na qualidade do processo de verificação e do produto. Dessa forma, o uso de uma metodologia de verificação funcional eficiente e de ferramentas que auxiliem o engenheiro de verificação funcional é de grande valia.

A metodologia OVM_tpi permite o desenvolvimento de todo o fluxo de construção de um ambiente de verificação, independente da escolha feita pela equipe desenvolvedora, de forma que o ambiente de simulação seja gerado antes da implementação do circuito a ser verificado (*Design Under Verification* - DUV). Além disso, ataca os principais desafios do processo de verificação funcional, tempo e custo de desenvolvimento, contribuindo para uma diminuição da complexidade, reusabilidade, comunicação entre o ambiente com uma interface bem definida e diminuição no tempo de desenvolvimento de um *testbench* através do uso de *templates* que criam de forma semiautomática partes do ambiente de verificação.

OVM_tpi teve como principal base a metodologia *Open Verification Methodology* (OVM), utilizando sua biblioteca para a construção do *testbench* e o paradigma de linguagem orientação objeto suportado por *SystemVerilog*, linguagem criada especialmente para verificação funcional e design. Sua validação foi através de estudos de casos que demonstraram a eficácia do seu uso, tanto para circuitos unidirecionais, quanto para bidirecionais.

Palavra-Chaves : SoC, OVM, verificação Funcional, VLSI, *testbench.*

Abstract

The advent of new technologies Very-large-scale integration (VLSI) and the growing demand for electronic products in the world are bringing a huge complexity increase of electronic circuits. On the other hand, time-to-market of an electronic product, the design time needed for the production, and sell of a system are getting smaller. Tests are needed to ensure that the integrated circuit works properly. One part of this area is called functional verification and it's the part of the project that require more time of development.

It's always important to search new methods that make functional verification easier, faster, more reusable and simpler in the development of the simulation environment, without reducing the quality of the verification process and of the final product. That's why, using an efficient functional verification methodology and a proper tool it's of huge importance.

The OVM_tpi methodology allows the monitoring of the entire development testbench flow, regardless of the flow chosen by the development team, so that the simulation environment have been developed and validated before the DUT (Design Under Test) have been describe. Moreover, OVM_tpi attacks the main challenges of the functional verification process contributing to reduction of complexity, reusability, communication between the environment with a well defined interface and reduction in a testbench development time through the use of templates that create semi-automatically the parts of environment verification.

OVM_tpi had as the main foundation the Open Verification Methodology (OVM), using its library for the build of the testbench, and the paradigm of object oriented language supported by SystemVerilog, a language created especially for functional verification and design.

Keywords: SoC, OVM, Functional Verification, VLSI, Testbench, Methodology.

Sumário

Lista de Abreviaturas e Siglas

ABV	*Assertion-Based Verification*
ASIC	*Application Specific Integrated Circuit*
BFM	*Bus-functional model*
BVE-COVER	*Brazil-IP Verification Extension*
DPCM	*Differential Pulse Code Modulation*
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*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
HDL	*Hardware Description Language*
IC	*Integrated Circuit*
IP	*Intellectual property*
IP-core	*Intellectual Property of Hardware Project*
LAD	Laboratório de Arquiteturas Dedicadas da UFCG
LEC	*Logical Equivalence Check*
OSCI	*Open SystemC Initiative*
RTL	*Register Transfer Level*
ROI	*Return of Investments*
SCV	*SystemC Verification Library*
SoC	*System on Chip*
TLM	*Transaction Level Model*
TLN	*Transaction Level Netlist*
UML	*Unified Modelling Language*
VHDL	*VHSIC Hardware Description Language*

VLSI	Very Large Scale Integration
VSIA	VSI Alliance

Lista de Figuras

Lista de Tabelas

Capítulo 1 Introdução

De acordo com a *Semiconductor Industry Association* (SIA) a crise mundial de 2008-2009 influenciou bastante o setor de semicondutores, sendo fevereiro de 2009 o pior mês em vendas do período de recessão, quando as vendas atingiram a marca de US$ 14,1 bilhões, o equivalente a 56,2% a menos que fevereiro de 2010 A SIA mostra também que apenas em fevereiro de 2010, as vendas mundiais atingiram US$ 22,0 bilhões, um decréscimo de 1,3% em relação a janeiro deste mesmo ano, que atingiu US$ 22,3 bilhões.

Apesar da diminuição nos meses de dezembro 2009, janeiro e fevereiro de 2010 o mercado mundial de chip encontra-se em expansão. Se analisarmos todo o ano de 2009 e de 2010, a venda de semicondutores alcançou um novo recorde de vendas de semicondutores, com um percentual de 32% maior, frente às vendas do ano de 2009.

O Brasil gastou, em 2008, uma quantia de US$ 4 bilhões para a importação de componentes semicondutores, isso sem levar em conta equipamentos prontos. De acordo com dados da Organização para Cooperação e Desenvolvimento Econômico (OCDE), nos países desenvolvidos, o setor eletrônico corresponde a 12% do Produto Interno Bruto (PIB). No Brasil, a indústria eletrônica é responsável por apenas 1,7% do PIB, tornando este setor deficitário na balança comercial. Para diminuir essa distorção, o governo brasileiro está investindo em construção de indústria de semicondutores e qualificação de recursos humanos especialista na área.

Assim como o mercado de semicondutores, cresce de maneira considerável o nível de complexidade dos produtos. Cada vez mais são necessários circuitos de maior complexidade em um espaço cada vez menor. Isso pode ocorrer devido à tecnologia de fabricação também se desenvolver e diminuir o tamanho do principal componente para a criação de semicondutores, o transistor, que é um dispositivo que controla a passagem da corrente elétrica através de materiais semicondutores inteiramente sólidos.

Sistemas complexos podem ser formados por diversos componentes de hardware que desempenham funcionalidades específicas. Tais componentes são blocos de hardware dedicados, denominados IP cores (*Intellectual Property core*). Quando há a junção desses IP's com uma ou mais unidades de controle (ex.: processador, microcontrolador), surge um SOC

(*System-on-Chip*), componente ainda mais complicado, com alta complexidade de verificação atrelada.

Todavia, para que um circuito eletrônico chegue ao mercado sem nenhum problema é necessário realizar vários tipos de testes[1] e verificação enquanto o semicondutor estiver sendo desenvolvido; testes esses que exigem técnicas já consolidadas para que o processo ocorra durante todo o projeto do Hardware encontrando erros ou divergências do circuito com sua especificação e focando na convergência de tempo de construção dos testes com o tempo de desenvolvimento do circuito integrado.

Dentre as técnicas de verificação utilizadas, específicas para eliminar as falhas e tentar ao máximo chegar num circuito isento de erros, tem especial relevância o processo ou metodologia de verificação funcional, que detém uma grande fatia de tempo do projeto, objetivando a detecção de erros funcionais no projeto lógico do hardware. Tal processo pode estar presente desde o início do fluxo até a sua concretização e, portanto, é o processo mais demorado e mais caro do projeto.

O uso da verificação funcional num projeto é muito importante, pois a necessidade de encontrar erros antes do circuito ser fabricado torna o processo de construção de Hardware mais confiável. Quando problemas são encontrados no circuito manufaturado, o custo de reparo é muito maior, pois o circuito terá que ser fabricado novamente com novas máscaras para serem usadas em seu processo de construção material.

Portanto, as metodologias de verificação são utilizadas para tornar o processo de correção dos erros mais baratos e prover maior qualidade ao produto final, pois este já chega ao mercado testado exaustivamente e com uma probabilidade muito reduzida de existir erro, em comparação com os que não utilizam a técnica de verificação funcional.

Caso um problema (bug) passe por essa etapa, fica mais difícil de ser encontrada a solução posteriormente, já que o foco das etapas seguintes não é encontrar erros, mas de realizar experimentações em protótipos para avaliar as especificações de tempo e as ligações entre os circuitos construídos, antes que o sistema venha realmente a ser construído. Quanto mais tardiamente se der a detecção do problema, maior será o custo para fazer o reparo do erro. A figura 1.1 mostra a forma exponencial do crescimento do custo versus o tempo de projeto no qual foi detectado o erro.

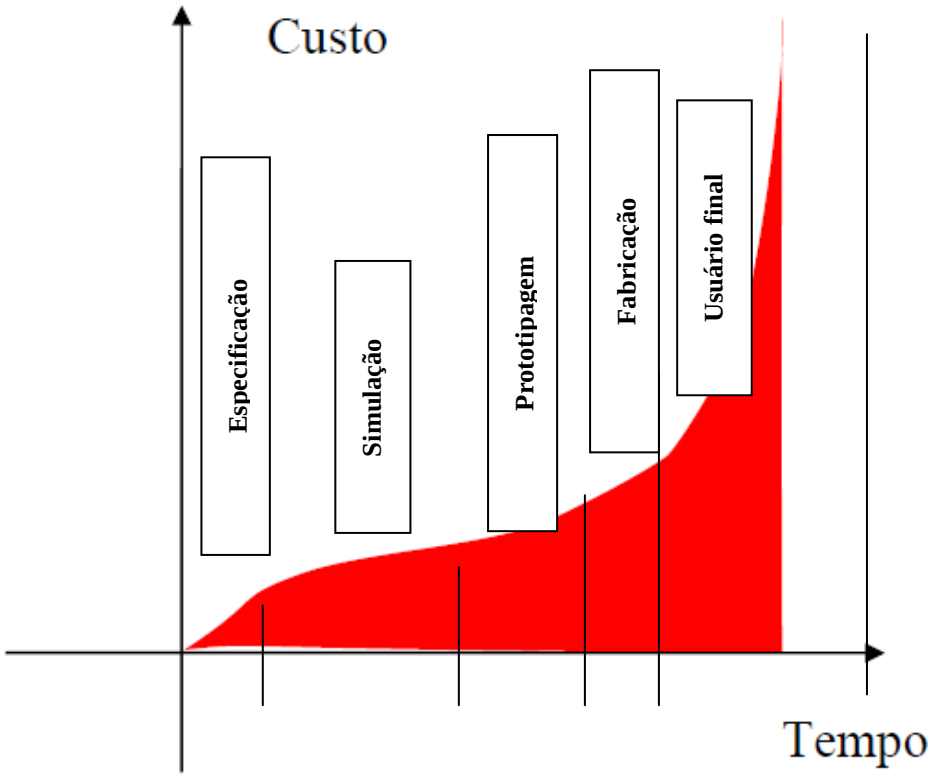


Figura 1.1: Custo de reparo de um erro com relação à fase do projeto no qual foi encontrado

Existem cinco etapas principais para inserção e descobertas de erros em um projeto de circuitos integrados. A fase de especificação é a mais suscetível à inserção de erros, pois nela todo o projeto será descrito a partir de uma comunicação entre o gerente de projeto e o cliente. Uma pequena divergência de entendimento pode criar um grande problema na construção do circuito. Porém, caso esse *bug* seja encontrado ainda nessa fase, o conserto não será tão complexo e consequentemente torna-se um reparo barato.

A etapa de simulação é a responsável por encontrar o maior número de erros de um sistema, uma vez que é o momento de realização da verificação funcional, técnica especializada em encontrar problemas no hardware. Caso um erro seja encontrado nessa etapa, terá que ser reparado pelos responsáveis pela descrição do circuito e posteriormente simulado e verificado novamente, até obter um resultado correto, sem nenhum problema encontrado.

Encontrar erros no momento da prototipagem é muito complexo devido à dificuldade de identificação de um problema quando se trabalha em um baixo nível de abstração; perde-se muito tempo para encontrar o local exato em que ocorreu a falha, e como o projeto já está na fase final de implementação, qualquer problema encontrado nessa etapa exigirá que todo o

fluxo seja revisado para concertar a descrição e posteriormente fazer todas as verificações necessárias. Além disso, caso sejam encontrados problemas nessa etapa, todas as atividades anteriores deverão ser refeitas com as mudanças ocorridas, tornando o custo muito maior.

Caso o erro continue sem ser detectado no final da etapa de prototipação, o circuito passará para o processo de fabricação e caso seja encontrado erro, seu reparo tem um custo extremamente elevado, pois todas as outras etapas anteriores deverão ser refeitas tornando o custo do projeto muito maior.

Após o chip chegar ao usuário final, qualquer defeito encontrado poderá implicar num grande problema para a empresa que o desenvolveu, pois após ser descoberto o erro, a empresa terá que fazer um *recall* além do conserto, construir novos testes para verificação, prototipar, enviar para a fabricação gerar novas máscaras, construir os chips e trocar todos os chips defeituosos vendidos no mercado. Destarte, tal procedimento é extremamente inviável, havendo casos na história dos semicondutores que refletem bem esse problema: a unidade de ponto flutuante do Intel Pentium 4; a destruição do Satélite *Mars Climate Orbiter* da NASA; e, em janeiro de 2011, a Intel declarou problema no seu processador *SandyBridge,* que custará cerca de US$ 700 milhões de dólares para reparação e substituição no mercado [52].

Segundo Dueñas[17], 65% dos circuitos falham em sua primeira prototipação em silício, e 70% desses casos ocorrem devido a uma verificação funcional mal elaborada. Bergeron [8] considera que o esforço do grupo de verificação consiste de 60% a 80% do esforço total do grupo de implementação. Keating [32] afirma que essa estimativa fica em torno de 80% do tempo de implementação de um projeto. Assim, diante da evidente importância do empenho do grupo na fase de verificação funcional, estudos são realizados constantemente, a fim de obter meios para diminuição do tempo e dos custos de verificação. Esta redução do custo e do tempo na etapa de verificação funcional é um desafio pesquisado de forma abrangente na área, com a finalidade de conseguir bons resultados com o desenvolvimento de técnicas para os problemas descritos abaixo:

1. Desenvolvimento do ambiente de verificação funcional antes do circuito já estar pronto para ser verificado;
2. Geração do ambiente de verificação de forma automática ou semiautomáticas focado nos seguintes itens:
    a. Definição de arquitetura do *testbench.*
    b. Definição da interface entre componentes do ambiente de verificação.
    c. Definição de bibliotecas de componentes do ambiente de verificação.

   d. Provimento de um fluxo de desenvolvimento que suporte a autovalidação dos componentes criados;
3. Utilização de linguagens Orientação Objeto de alto nível de abstração para a criação de bibliotecas de funções que possam ser utilizadas em geração de diversos tipos de *testbenches* (ambiente de simulação).

Observa-se a importância, por ocasião do projeto de circuito integrado, de haver trabalho paralelo entre os grupos de verificação funcional e *design*, pois, dessa forma, o engenheiro de verificação não necessitará esperar o circuito ser descrito para iniciar a construção do ambiente de verificação, podendo avançar no projeto. Outro fator importante na construção do ambiente de verificação anterior à inserção do circuito é o fato de já ter validado todo o ambiente de verificação funcional antes de verificar o circuito.

O uso da geração do ambiente de verificação de forma automática ou semiautomática é válido, também, por diminuir o tempo de construção do ambiente de verificação funcional e, consequentemente o custo e o tempo de projeto. Porém, a geração de código só poderá auxiliar no desenvolvimento do ambiente de verificação caso este seja padronizado; para tal, faz-se necessário que seus componentes tenham uma arquitetura bem definida, além de uma interface igualmente bem definida entre os componentes, e utilização de alguma ferramenta para prover essa geração semiautomática. Um fluxo de desenvolvimento que provê a autovalidação do ambiente de verificação funcional é interessante, para que todos os componentes possam ser validados por partes. Já o uso do paradigma de linguagem orientada a objeto é importante para a construção do ambiente de verificação, permitindo a reutilização das estruturas desenvolvidas em outras etapas do projeto (ou até em outros projetos).

Outro fator relevante num projeto de SOC é o tempo: quanto maior o tempo de projeto, menor o *time-to-marketing* (período que o produto fica no mercado para compra e venda) do produto. Atualmente, produtos com alta tecnologia estão ficando menos tempo no mercado, visto que aparece outro produto ou versão mais atraente para o consumidor final. Dessa forma, mecanismos que tornem o fluxo do projeto mais ágil devem ser incentivados, abrangendo as mesmas etapas do fluxo de projeto já padronizado de construção de um SOC (*System on chip*).

1.1 Fluxo padrão de desenvolvimento de um projeto de hardware

Para desenvolver um projeto de hardware é necessário seguir um fluxo de

desenvolvimento que contém algumas etapas de gerenciamento das atividades. A figura 1.2 mostra o fluxo padrão de desenvolvimento de um hardware.

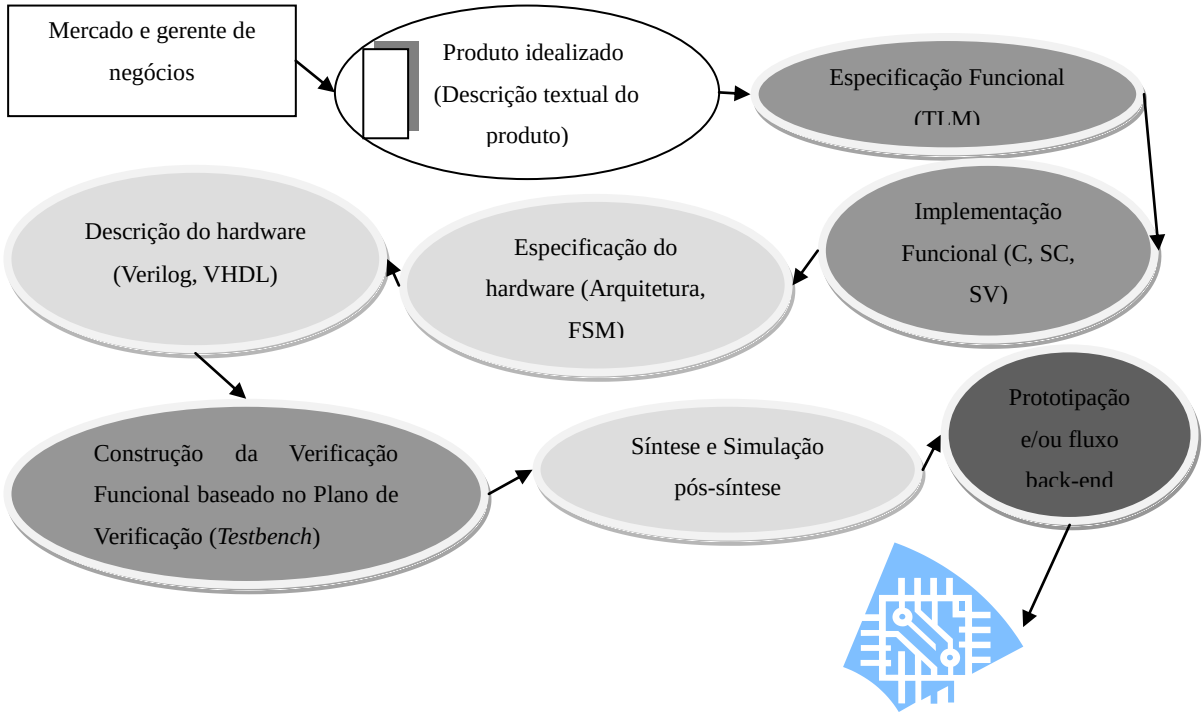


Figura 1.2: Fluxo padrão de desenvolvimento de um projeto de hardware

De acordo com a AMA(*American Marketing Association*) conceitua-se *marketing* como uma função organizacional e um conjunto de processos que envolvem a criação, a comunicação e a entrega de valor para os clientes, bem como a administração do relacionamento com estes, de modo que beneficie a organização e, consequentemente, seu público interessado. Algumas estratégias utilizadas nesse setor são: preço, pacote de produtos ou serviços, nicho do mercado, promoção, distribuição e gestão de marcas.

Após o grupo de negócios e *marketing* iniciarem o processo de criação do SOC, começa o processo de documentação dos requisitos e especificação do produto. A especificação deve ter um alto nível de abstração, ainda não havendo 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. Conterá detalhes de alto nível, tais como funcionalidades a serem executadas, informações da freqüência e todos os requisitos requeridos pelo cliente. Caso o produto seja impossível de ser construído, é devolvido um documento detalhando todas as impossibilidades encontradas no processo de idealização, a fim de que a área de negócios possa entrar em contato com o cliente e tomar as

providências cabíveis. Caso contrário, é escrita uma especificação com todo o detalhamento do produto requisitado para que possa ser utilizado tanto pelo responsável pela especificação do hardware quanto para o responsável pela especificação da verificação funcional. É importante que qualquer mudança que ocorra dentro dessa especificação seja repassada para os responsáveis tanto pela especificação do hardware quanto para os responsáveis pela especificação da verificação funcional.

Uma vez construído o documento do produto, este é repassado para o setor de produtos e engenharia de sistema, que fica responsável em receber o documento do produto e gerar a especificação funcional. Este setor interage diretamente com o setor de marketing e com os engenheiros de implementação, tirando dúvidas e verificando problemas que podem tornar o projeto inviável.

A equipe de produto e sistema é, pois, responsável por gerar a especificação funcional e o produto requisitado em alto nível, que incluirá as especificações do sistema e da verificação do hardware.

A implementação do sistema utiliza *Transaction level model* (TLM) para descrição no nível arquitetural de abstração, bem como para modelar os blocos identificados num estágio inicial de análise e exploração da arquitetura. Em geral, a modelagem em nível de transação capacita os engenheiros de sistema a especificarem comportamentos de blocos no alto nível de abstração, que tem por foco o comportamento dos blocos e a interação entre eles, sem haver preocupação com a sincronização existente no baixo nível.

Na próxima etapa do fluxo é desenvolvida a especificação do hardware. Neste momento, é crucial a descrição de alguns pontos detalhados para que os engenheiros de implementação não tenham nenhuma dúvida quanto ao produto que deve ser implementado. A seguir temos uma descrição dos pontos que devem fazer parte de uma especificação de hardware: visão geral do bloco, diagrama de blocos, interface de sinais, formas de ondas temporizadas, diagrama da máquina de estados do bloco, descrição da maquina de estado, registradores de controle do bloco (caso existam) e descrição dos possíveis caminhos críticos.

A especificação do hardware também deve incluir: informações de síntese, tecnologia do processo que será utilizado, máxima frequência, energia e geometria (máxima área) utilizada para a síntese. Por tudo que já foi dito, fica claro que a especificação do hardware necessita ser realizada por alguém que conheça os mínimos detalhes da aplicação visada, e que qualquer mudança ocorrida dentro dessa especificação deve ser repassada para a equipe de verificação e documentada pelo gerente de projeto.

A implementação do hardware é um processo que utiliza uma linguagem de descrição de hardware em um nível mais baixo quando comparado ao nível de sistema. A descrição é feita em nível de sinais, utilizando um sinal de relógio para controlar e sincronizar os blocos implementados.

Posteriormente é iniciada a etapa de verificação funcional, cujo objetivo é certificar que a implementação inicial do RTL tem as mesmas características e funcionalidades do produto idealizado pelo cliente. Com vistas a tal objetivo, é criado um ambiente, denominado de ambiente de verificação, de modo a viabilizar uma convergência entre o RTL, o produto idealizado e a especificação funcional. O produto idealizado é construído numa linguagem mais alto nível, na etapa de implementação do sistema, e irá servir para toda a etapa da verificação funcional como o modelo de referência.

Na atividade de síntese é utilizada uma ferramenta que irá transformar o código RTL desenvolvido em postas lógicas. A partir de então, é feita a verificação desse arquivo na etapa de simulação pós-síntese e o início do processo físico de desenvolvimento do hardware.

1.2 Mudança na atividade de verificação funcional

Para um projeto de hardware minimizar seu tempo de verificação funcional, é necessário seguir algumas métricas como, por exemplo, agilidade, cobertura e o uso de *Assertions*. Em verificação funcional, agilidade é uma métrica que indica o quão rápido e eficiente é o desenvolvimento do ambiente de verificação funcional. A métrica de cobertura é utilizada para verificar se todos os prováveis valores de um conjunto foram estimulados, só sendo finalizada a simulação quando o objetivo de cobertura for abrangido. O uso de *Assertions* tem o objetivo de inserir condições que o engenheiro de verificação já admite ser verdadeira em um determinado ponto do testbench; com o auxílio desse predicado, as falhas de protocolo e controle podem ser diagnosticadas mais facilmente.

Com a finalidade de conferir mais agilidade ao desenvolvimento dos ambientes de verificação funcional, foram estudadas quatro metodologias existentes de verificação funcional, de forma a analisar os pontos positivos e negativos, relacionando-os com os desafios existentes na área de verificação funcional. São elas: VeriSC, OVM, IVM e BVM.

A metodologia VeriSC[44] foi criada para o projeto de circuitos digitais síncronos que possuem um único sinal de *clock* e é caracterizada por um fluxo de atividades em que todo o modelo de referência do circuito é construído antes do início da construção do ambiente de verificação funcional (*top-down*). Como pontos positivos desta abordagem, podemos citar: a

simplicidade da arquitetura do ambiente de verificação funcional; a validação do ambiente de verificação funcional antes da inserção do circuito; a geração de forma semiautomática do ambiente de simulação, reduzindo o número de linhas de código que precisa ser implementado; provimento do reuso de códigos gerados em outras atividades de construção do ambiente de verificação.

A metodologia OVM se caracteriza por um fluxo de desenvolvimento de projeto, no qual o circuito é implementado por partes (*bottom-up*), tendo sido concebida para a criação de ambientes de verificação de qualquer sistema ou circuito digital. Os pontos relevantes nessa metodologia são: utilização de linguagens Orientação Objeto de alto nível de abstração, para a criação de bibliotecas de funções que possam ser utilizadas no desenvolvimento de diversos ambientes de verificação funcional; arquitetura do *testbench* bem definida; interface entre componentes do ambiente de verificação bem definida; biblioteca de funções bem definida e bastante utilizada para construção de *testbenches;* e provimento de reuso do ambiente de verificação para diferentes projetos.

A metodologia IVM (*Interoperable Verification Methodology*)[41] destaca-se por sua arquitetura de *testbench* concebida com vistas às virtudes e defeitos das metodologias VeriSC e principalmente a metodologia OVM, buscando a obtenção de uma metodologia melhorada. É caracterizada por um fluxo de desenvolvimento de projeto onde o circuito é implementado por partes, até a concepção do circuito digital completo (*bottom-up*). Como pontos positivos, esta metodologia buscou a geração do ambiente de verificação antes do DUV, e de forma autoverificável; definiu bem uma arquitetura do ambiente de verificação, bem como uma biblioteca de interfaces para os componentes do *testbench;* prover o suporte a comunicação bidirecional; e forneceu o conceito de reusabilidade de código em diferentes fases da construção do *testbench;*

BVM[37] é uma metodologia criada para a utilização no consórcio *Brazil-IP,* que utiliza a linguagem *SystemVerilog* e a biblioteca de OVM como base para o desenvolvimento do ambiente de verificação. Foi baseada na metodologia VeriSC, porém com a inserção do elemento *Actor,* que ajuda na verificação e monitoramento do protocolo de comunicação dos componentes do projeto, facilitando a localização de erros entre essas interfaces e diminuindo o tempo necessário para encontrá-los, caso ocorram. Seus pontos fortes são exatamente os mesmos da metodologia VeriSC.

1.3 Objetivos e contribuições

O trabalho proposto tem como principal objetivo desenvolver uma metodologia de verificação funcional que apresente uma combinação dos trabalhos relacionados, proporcionando novas funcionalidades e agilidade, através de um conjunto de objetos desenvolvidos na linguagem de programação e descrição de hardware *SystemVerilog.* Além disso, a metodologia propõe uma técnica de cobertura de controle (sinais do protocolo de comunicação) baseada em *assertions* e, uma técnica de cobertura de dados baseada em *coverage,* biblioteca pertencente à linguagem *SystemVerilog*.

Tal metodologia de verificação funcional possui as seguintes características:

- Suporte ao desenvolvimento do ambiente de simulação antes do circuito já estar pronto para ser verificado;
- Geração do ambiente de simulação de forma automática ou semiautomática;
- Fundamento em um fluxo de desenvolvimento que suporta a autovalidação dos componentes criados;
- Possibilidade de reuso do ambiente de verificação enquanto está sendo construído e validado.
- Suporte o desenvolvimento de ambientes de verificação funcional com comunicação bidirecional.
- Utilização do paradigma de linguagens orientadas a objeto, para a criação de bibliotecas de funções que possam ser utilizadas na comunicação dos componentes do *testbench.*
- Suporta o desenvolvimento de uma arquitetura do *testbench* com sua interface de comunicação entre os componentes padronizada.
- Desenvolvimento de uma biblioteca de componentes padronizada para facilitar a comunicação e a geração semiautomática do ambiente de verificação, utilizando a biblioteca de funções de OVM;

De forma a realizar o intento, foi definida uma metodologia de verificação funcional denominada OVM_tpi, visando a junção dos pontos positivos de cada uma das metodologias estudadas, acrescidas de novos objetivos como o uso de *Assertions* junto com o desenvolvimento de uma biblioteca de automatização, e com a finalidade de cobrir todos os desafios expostos, facilitando, assim, o desenvolvimento da verificação em um projeto de

hardware.

O suporte ao desenvolvimento do ambiente de simulação antes da inserção do DUV é buscado através da emulação do modelo de referência junto com os componentes responsáveis pela tradução de uma transação em sinais e de sinais em transação. Para a geração semiautomática do *testbench* foi utilizada uma ferramenta especifica para este fim, e já visualizada em outras metodologias estudadas.

A padronização dos componentes e o uso da linguagem *SystemVerilog* provê um reuso dos componentes criados de forma simples, pois estes são validados em cada etapa da construção do *testbench* final. Foi utilizada a biblioteca de OVM para prover comunicação bidirecional e unidirecional no *testbench.* Por ser uma linguagem orientada a objeto, toda a comunicação entre os objetos devem ser feitas através de funções (*gets* e *sets*) que gerenciam os atributos da transação e o empacotamento daqueles objetos já criados. Com o conceito de *Interface* será buscada a criação de um encapsulamento entre os componentes que trabalham no nível de abstração de sinais provendo um maior reuso dos componentes desenvolvidos.

O desenvolvimento de uma biblioteca de componentes padronizada tem a finalidade de facilitar a comunicação e a geração semiautomática do ambiente de verificação, e será elaborada através da implementação de *templates* para uma ferramenta com o fim específico.

1.4 Estrutura do trabalho

Para guiar o leitor deste trabalho será detalhado o conteúdo que está organizado da seguinte forma:

No capítulo 1 há uma breve introdução sobre o trabalho proposto, mostrando a importância da verificação funcional, bem como os desafios encontrados na área e quais são os objetivos e contribuições deste trabalho.

O Capítulo 2 apresenta os conceitos fundamentais referentes à verificação funcional, definindo as nomenclaturas e os principais conceitos adotados.

No Capítulo 3 são apresentadas algumas metodologias que serão utilizadas como embasamento para a construção do trabalho teórico, apogeu do estado da arte de metodologias de verificação funcional.

Já no Capítulo 4 é feita uma descrição minuciosa da metodologia de verificação funcional OVM_tpi, principal foco deste trabalho, detalhando todo seu fluxo, sempre fazendo a correlação com os objetivos e contribuições aqui propostos.

O processo de automatização da construção do ambiente de verificação, para a

Metodologia OVM_tpi, é analisado no Capítulo 5, seguido pelo Capítulo 6, que traz alguns estudos de casos, com a finalidade de mostrar os benefícios do uso dessa metodologia aliado à sua semiautomatização.

No Capítulo 7 há o relato, de forma conclusiva, dos resultados obtidos frente aos objetivos propostos, ressaltando as contribuições aqui realizadas, que poderão ser ponto de partida para futuros trabalhos.

O Apêndice A contém todos os códigos dos *templates* criados, e todos os *scripts* para as duas opções de uso da metodologia OVM_tpi, *top-down* e *bottom-up.*

Capítulo 2 Conceitos Fundamentais

No presente capítulo serão descritos os principais conceitos de verificação, basilares deste trabalho, com o intuito de facilitar o entendimento do conteúdo subsequente. Antes de definir verificação funcional, serão analisados os tipos de verificação que podem ser desenvolvidas em um projeto de circuito integrado.

Antes mesmo de definir o que é verificação, é necessário diferenciar verificação e validação. Verificação responde se o sistema está sendo construído corretamente, enquanto a validação se preocupa em checar se o sistema especificado tem o propósito esperado.

O trabalho de validação é de responsabilidade da equipe que especifica o sistema. Falhas tardiamente detectadas, ou não detectadas, nem pela validação, nem pela verificação, impactam de maneira devastadora sobre o cronograma, custo e qualidade do projeto, podendo inclusive culminar em seu cancelamento.

2.1 Tipos de verificação

Verificação significa fazer a checagem em um objeto com a finalidade de averiguar se o mesmo está se comportando como o esperado. Na microeletrônica existem, atualmente, três tipos de verificação utilizados no processo de desenvolvimento de um projeto:

- Verificação Formal;
- Verificação Funcional ou verificação baseada em simulação;
- Verificação híbrida ou semiformal.

As seções seguintes se destinam a uma revisão sobre essas três técnicas, expondo suas principais características.

2.1.1 Verificação Formal

No método de verificação formal uma propriedade é estaticamente checada. Significa que, uma vez que o processo de verificação está completo, pode-se ter certeza que o modelo de implementação satisfaz as propriedades e combinações para todos os valores de entradas. Dentre os tipos de verificação funcional existentes, os mais conhecidos são: checagem de equivalência, checagem de modelo e prova de teorema. Cada um desses métodos expressa a

especificação exatamente igual à implementação, através de um modelo matemático.

De acordo com Iman[28], verificação formal é o tipo de verificação que utiliza os conceitos de lógica e fórmulas matemáticas para provar ou refutar uma determinada propriedade do *hardware* desenvolvido.

Existem diversos tipos de verificação formal, com o intuito de verificar formalmente se o código HDL do *design* satisfaz a determinadas condições ou proposições lógicas. A verificação formal tem duas características que a diferencia de outros tipos de verificação:

- Fazer afirmações válidas a partir de uma especificação para todos os casos;
- Não necessitar de vetores de testes para ser aplicada.
- É utilizado no processo de checagem de equivalência.

A verificação formal pode provar a inexistência de erros através de equações matemáticas e verificação de modelos. No entanto, tal processo pode ser complicado, caso o circuito tenha um alto nível de complexidade provendo limitações ao uso desse tipo de verificação.

Checagem de Equivalência Formal é um processo de prova no qual são comparados dois circuitos e verificado se ambos exibem exatamente o mesmo comportamento. Isso implica que as máquinas de estados finitos de ambos os circuitos produzirão a mesma saída para todas as possibilidades de entrada.

Na verificação por equivalência, as fórmulas da especificação e da implementação são reduzidas a formas canônicas, aplicando-se transformações matemáticas. As entradas de uma ferramenta de checagem de equivalência são duas representações formais do *hardware* representado, o Arquivo de descrição RTL e o arquivo de saída do processo de síntese *netlist*. A partir dessas entradas, a ferramenta irá criar um modelo da função booleana para cada entrada, de modo que para cada função booleana existe uma única forma de representação denominada de modelo canônico.

Quando é feita uma verificação formal através de checagem de modelos, a implementação é expressa através de uma máquina de estado com as transações, e a especificação é descrita por um conjunto de propriedades. Cada propriedade descrita é verificada, percorrendo todos os estados da máquina de estado desenvolvida. Esse tipo de checagem pode ser utilizado para verificar a lógica temporal do circuito. Lógica temporal, por sua vez, representa um sistema de regras e símbolos utilizados para descrever as relações entre duas variáveis lógicas em termo de tempo.

O objetivo da prova de teorema é tentar deduzir a equivalência das fórmulas da

especificação e da implementação, que são escritas em uma dada lógica matemática. Usando as leis da lógica e a matemática discreta, a implementação pode ser deduzida da especificação, ou vice-versa.

2.1.2 Verificação Funcional Baseada em Simulação

Verificação funcional é também conhecida como verificação dinâmica, sendo desenvolvida, como o próprio nome diz, através de simulação.

A etapa de verificação funcional baseada em simulação avalia os valores em cada estado do *hardware,* de acordo com um conjunto de entradas denominados “vetores de testes”. Esse tipo de verificação testa o projeto num ambiente semelhante àquele que será utilizado no final do desenvolvimento do projeto, avaliando todas as suas funcionalidades, levando em consideração os atrasos de sinais.

Deve-se ter em mente que a verificação funcional não prova a ausência de erros, mas a presença dos mesmos. No entanto, não há limitações quanto ao tamanho de modelos a serem verificados, desde que não haja empecilho com relação ao tempo gasto na simulação. Além disso, existem métodos para certificar quanto das funcionalidades de um projeto foram testadas, o que pode dar certa garantia de quanto a verificação foi abrangente. Estes métodos são chamados de cobertura.

De acordo com Prado[40], verificação funcional é o processo que busca demonstrar a equivalência entre uma implementação e sua respectiva especificação. Bergeron[8], conceitua verificação funcional como sendo um método utilizado para comparar o DUV com sua especificação. Já para Mintz e Ekendahl[36], verificação funcional é a construção e execução de um software que dará a certeza se o *device under test* (DUT) opera de acordo com a intenção, antes de este DUT virar um chip e ser colocado no mercado.

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

De acordo com Silva[44], um aspecto importante é decidir a granularidade do circuito a ser verificado. Quando se faz a verificação de um projeto, ele é normalmente dividido em blocos para facilitar a sua verificação; isso porque o processo de verificação funcional torna-se mais complexo quando empregado em um bloco muito grande, por ser mais difícil encontrar um erro ao fazer a simulação do projeto inteiro de uma única vez. Quando se

verifica um projeto muito grande com todas suas funcionalidades em um único módulo, ao aparecer um erro, é necessário fazer uma investigação de todo o modelo para descobrir onde está a causa. Com um modelo menor, sem dúvida a investigação se torna mais fácil. Por outro lado, com essa divisão, mais ambientes de simulação devem ser criados para a verificação de cada bloco em que o DUV foi dividido.

Todo esse processo deve estar contido na especificação geral, para que tanto a equipe de verificação quanto a equipe de implementação possam descrever seus planos de desenvolvimento. A partir do conteúdo descrito na especificação, a equipe de arquitetura começa a desenvolver o sistema no nível *electronic system level* (ESL) e, posteriormente, utiliza esse sistema como um modelo de referência no ambiente de simulação criado. Para cada módulo desenvolvido pelos engenheiros de *design*, um modelo de referência deve já ter sido construído pela equipe encarregada de produzir o sistema. O plano de verificação também tem que descrever o fluxo de verificação do circuito (DUV), baseado na especificação e na arquitetura definida dos módulos do circuito.

Em uma organização que trabalha desenvolvendo chips, é fundamental que a equipe de desenvolvimento do código RTL e a equipe de sistema e verificação trabalhem separadamente, para evitar o vício de um código seguir o mesmo caminho do outro, tornando o modelo de referência tão errado quanto o módulo descrito.

A criação do modelo de referência no nível de sistema é importante, pois é muito mais simples criar as funcionalidades de um projeto utilizando uma linguagem de alto nível de abstração, como por exemplo, *SystemVerilog* ou *SystemC*. Todavia, mesmo utilizando esse artifício, Piziali[39] mostra que tendo um projeto com uma intenção, uma especificação e uma implementação, o objetivo principal é que os três conjuntos se sobreponham gerando o produto ideal. Porém, isso se torna muito complexo e a maioria dos projetos não cumprem tal requisito. O objetivo da verificação funcional é tentar maximizar essa junção, como visto na figura 2.1:

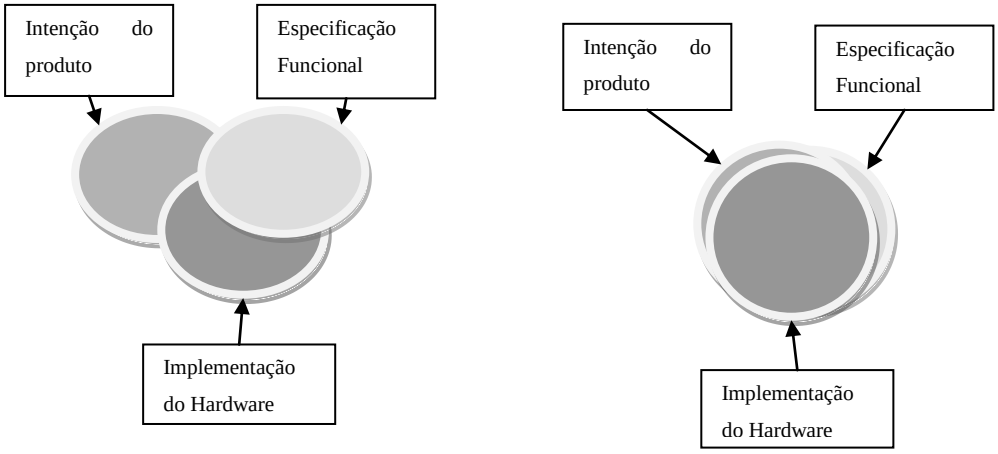


Figura 2.1: Objetivo da verificação funcional

Cabe registrar que, para fins de simplificação, por vezes o presente trabalho fala apenas em "verificação" ao invés de "verificação funcional", casos em que o aspecto funcional estará subentendido.

2.1.2.1 Testbench

Conceitua-se *testbench* como o ambiente de verificação completo, no qual se aplicam estímulos e são verificadas as respostas de um ou mais casos de testes. Um caso de teste pode ser verificado através de um *testbench* dirigido ou *testbench* randômico com o auxílio de cobertura.

O *testbench* utiliza simulação para verificar o DUV. Para que essa simulação seja possível, é necessário que haja um ambiente de verificação que possa receber o DUV, inserir estímulos e comparar suas respostas com as respostas de um modelo de referência. Para Bergeron[9], um bom *testbench* deve possuir as seguintes características básicas: ser dirigido por coberturas, possuir randomicidade direcionada, ser autoverificável e baseado em transações. A figura 2.2 mostra uma estrutura genérica de *testbench*:

Figura 2.2: Estrutura de um *testbench*

A seguir, procede-se a uma explanação sobre os tipos de verificação funcional

existentes, e outros conceitos importantes na verificação funcional.

2.1.3 Tipos de Verificação Funcional

Várias abordagens podem (e devem) ser utilizadas para que a verificação seja executada de maneira satisfatória. Dentre os tipos de verificação existentes na literatura, temos:

2.1.3.1 Caixa Preta

Este tipo de verificação está focado em um bloco ou na funcionalidade de um bloco apenas através dos sinais de entrada e saída. O sistema é estimulado e apenas observa-se as saídas geradas, desconsiderando completamente informações sobre estrutura e implementação. A abstração de detalhes internos traz vários benefícios, como por exemplo, a preocupação apenas com o tipo de transação que será utilizado para estimular o módulo, mas também traz algumas desvantagens:

- Dificuldade de verificar características relacionadas à tomada de decisão do bloco;
- Depuração de erros;
- Sempre requer um modelo de referência para fazer a verificação funcional.

A figura 2.3 mostra o esquema de uma verificação caixa-preta.

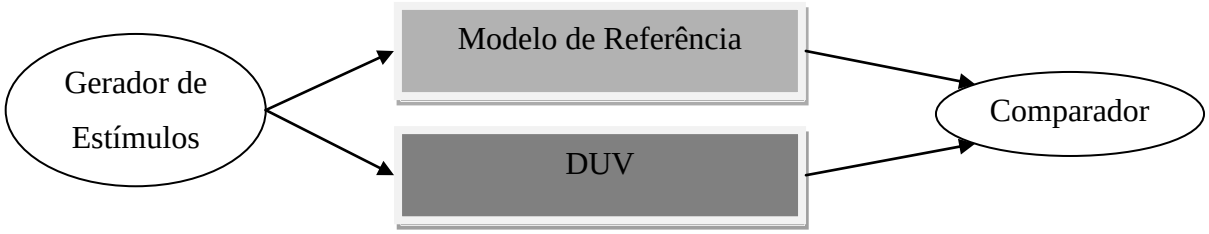


Figura 2.3: verificação Funcional caixa preta

2.1.3.2 Caixa Branca

A verificação caixa branca é um teste orientado à lógica, que avalia o comportamento interno do *hardware*, ou seja, tem acesso direto a detalhes internos da implementação do DUV. Essa técnica trabalha diretamente com a descrição RTL para avaliar aspectos como:

- Ocorrência de uma condição específica;
- Ocorrência de uma sequência de eventos.

Não necessita de modelo de referência para ser verificado o DUV: utilizando

monitoramento dos sinais e *Assertions*, a verificação é feita. Cabe salientar que o uso de *Assertions*, nesse momento, não é obrigatório, apenas recomendável. A complexidade desse tipo de verificação é determinada pelo tamanho do bloco que será verificado, e tende a um crescimento exponencial a cada funcionalidade do bloco. A figura 2.4 mostra a estrutura da verificação caixa branca:

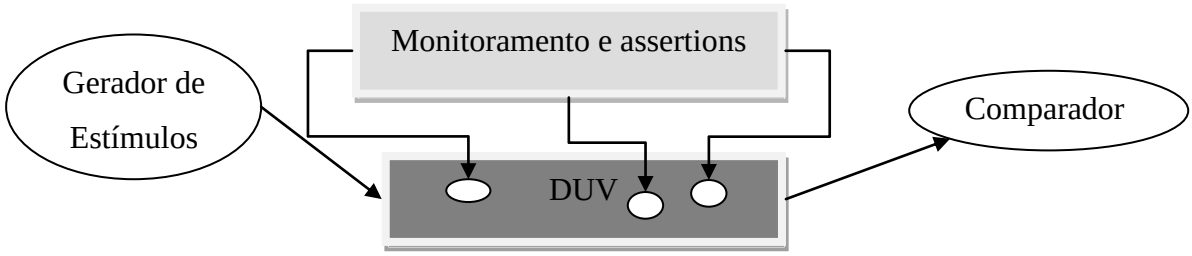


Figura 2.4: verificação Funcional caixa branca

2.1.3.3 Caixa Cinza

A verificação caixa cinza é a combinação da verificação caixa preta e caixa branca. *Monitor*es e assertions são inseridos para monitorar o comportamento dentro do DUV e o modelo de referência é utilizado. Desse modo, tenta checar com precisão todos os requisitos do modelo de referência e também reduz o esforço de depuração quando *bugs* são encontrados.

A figura 2.5 mostra como seria uma verificação funcional caixa cinza:

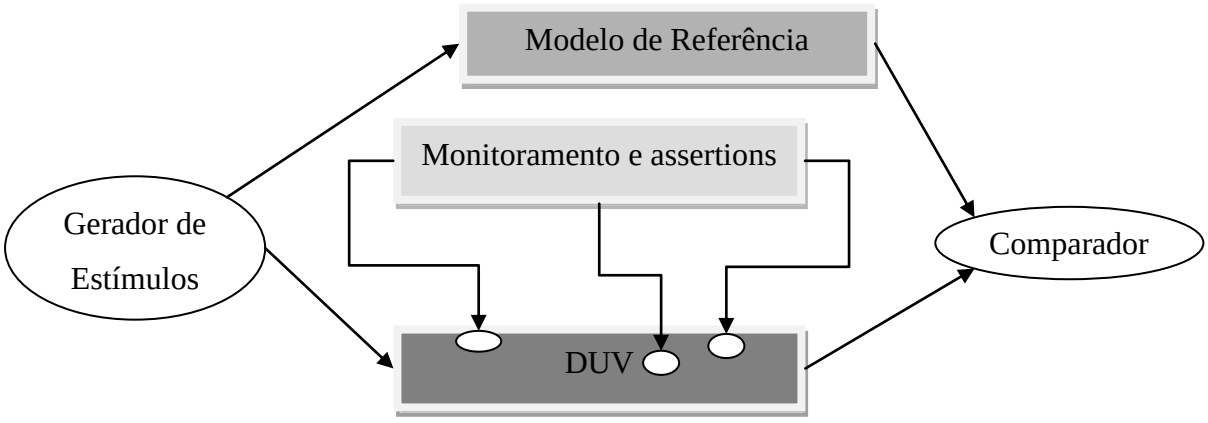


Figura 2.5: verificação Funcional caixa cinza

A tabela 2.1 mostra o esforço necessário para cada desafio encontrado na verificação

funcional, com relação ao tipo de verificação que é utilizado no desenvolvimento do projeto.

Tabela 2.1: Comparação dos tipos de verificação

Desafios da verificação funcional	**Esforço do tipo de verificação**		
	Caixa preta	**Caixa branca**	**Caixa Cinza**
Esforço de criar um modelo de Referência	Alto	Nenhum	Alto
Adicionar Assertions e monitores	Nenhum	Alto	Baixo
Verificar problemas na saída	Baixo	Alto	Médio
Esforço de encontrar problemas dentro do bloco	Alto	Baixo	Baixo
Esforço para construção da verificação	Baixo	Alto	Muito alto

2.1.4 Verificação Híbrida ou Semiformal

A verificação hibrida ou semiformal combina a verificação funcional com a verificação formal. Geralmente é utilizada para sistemas complexos que envolvem tanto hardware quanto sistema embarcado. Na verificação híbrida são utilizadas técnicas de métodos formais junto com a simulação na verificação de sistemas de pequeno, médio e grande porte, com o objetivo de diminuir o tempo de simulação necessário para cobrir todos os possíveis casos de testes. Esse tipo de verificação é muito utilizado para verificar falsos buracos de coberturas existentes na verificação.

2.2 Tipos de Metodologias de Verificação

Ao realizar uma verificação, primando sempre pela sua qualidade, seguir regras é a melhor forma para conseguir o objetivo proposto; tais regras também são

chamadas de tipos de metodologias de verificação. É importante não confundir tipos de verificação funcional e tipos de metodologias de verificação funcional. Nas subseções seguintes serão conceituados alguns tipos de metodologias de verificação existentes.

2.2.1 Verificação Baseada em Assertion

Na metodologia *Assertion-Based Verification* (ABV) os engenheiros de design usam afirmações para capturar a intenção do projeto específico e, através de simulação, verificação formal ou emulação destas afirmações, verificar se o projeto implementa corretamente a intenção.

Com linguagem e ferramenta de apoio para as afirmações amplamente disponíveis, designers e engenheiros de verificação adotaram metodologias ABV para melhorar a qualidade do design e da produtividade de verificação. Alguns benefícios do uso são:

- Garantir o comportamento formalizado correto;
- Aumentar a detecção de erros de projeto na sua origem e diminuir o tempo de observação e de depuração;
- Garantir um retorno de investimento (ROI) elevado, além de poder ser usado tanto na simulação, quanto na verificação formal e até mesmo na emulação.

O fluxo do uso de *Assertion-Based Verification* é:

- Identificar as propriedades a serem afirmadas/testadas;
- Decidir quais propriedades devem ser afirmadas;
- Decidir a ferramenta que será utilizada;
- Construir o ambiente pressuposto inserindo os *assertions* no código;
- Verificar as respostas dadas pela ferramenta.

2.2.2 Verificação Dirigida à Cobertura de Entrada/Saída

Os geradores de testes randômicos estão se tornando cada vez mais avançados e, realmente, têm ajudado a melhorar a qualidade da verificação dos dispositivos, pois geram valores aleatórios, que se simulados durante um tempo suficiente, podem cobrir todos os valores necessários para estimular o dispositivo. Entretanto, deve existir também alguma forma de gerenciar esses estímulos, a fim de saber quais valores foram já gerados pelos geradores randômicos e usados pelo DUV. Tal método de gerenciamento é chamado de

cobertura de entrada e saída.

Podemos conceituar cobertura funcional como uma medição de quais funcionalidades do projeto foram exercitadas durante os testes[45]. As funcionalidades que não são devidamente testadas durante a simulação são denominadas “buracos” de cobertura. Silva[44] aponta três causas da existência deles:

- 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, não detectando erros que causariam grandes prejuízos para o projeto. A análise da cobertura é a principal técnica para demonstrar que a simulação obteve sucesso, garantindo que o RTL foi verificado conforme desejado.

Verificação dirigida a cobertura é aquela baseada na simulação, com o foco na produtividade e no ganho de eficiência. Representa um dos tipos de metodologia mais utilizados no mundo, devido ao vasto leque de conceitos de verificação que visa:

- Verificação dirigida a transação;
- Geração de estímulos randômicos;
- Checagem de resultado automático;
- Cobertura;
- Verificação baseada em testes diretos;

2.2.2.1 Ciclo de Vida de um Projeto de Verificação Dirigido a Cobertura

Um projeto de verificação dirigido a cobertura percorre as seguintes fases:

1. Desenvolvimento do plano de verificação;
2. Implementação do ambiente de verificação;
3. Criação (abrange compilar, elaborar e rodar) do ambiente de verificação com os estímulos aleatórios;
4. Criação de testes direcionados para os *corner-cases*, testes para casos reais, e aqueles que não foram cobertos;
5. Avaliação da cobertura e checagem de todos os casos descriminados no plano de verificação.

Num primeiro estágio, o plano de verificação é desenvolvido de acordo com a especificação do projeto, mediante auxílio dos engenheiros de sistemas, da equipe de implementação e dos engenheiros de verificação funcional. O plano de verificação continua em desenvolvimento durante todo o ciclo de vida da verificação, nele sendo inseridos resultados durante o processo, podendo também vir a ser modificado caso haja necessidade, mediante aval do gerente de projeto.

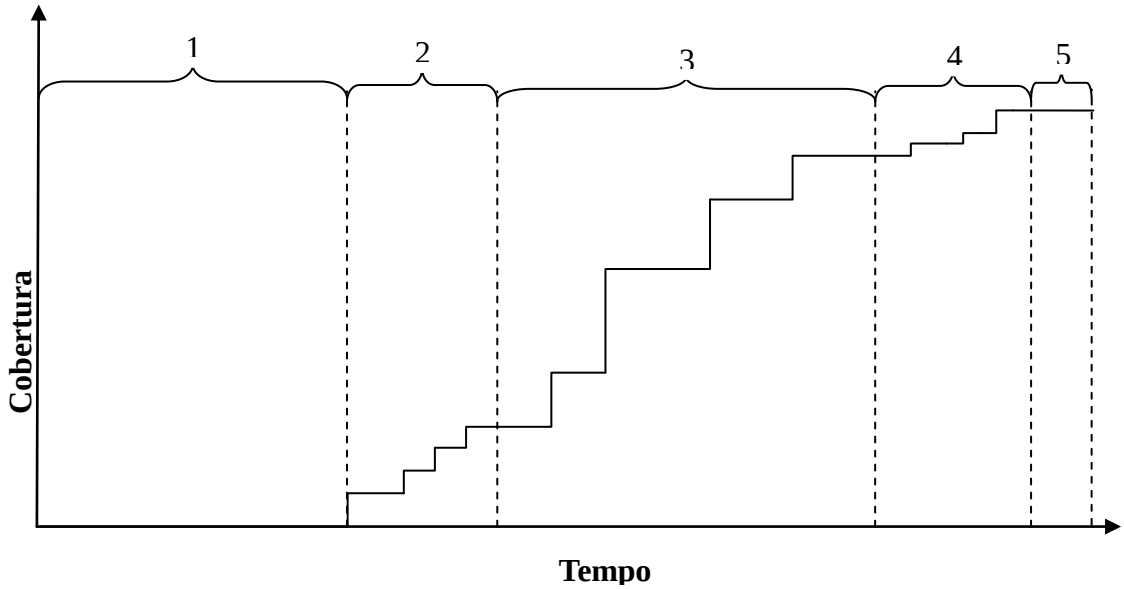
O segundo passo é a implementação do ambiente conforme os requisitos identificados no plano de verificação. Escolhe-se uma linguagem propícia para desenvolver um ambiente o mais padronizável e reusável possível. A cobertura já deve fazer parte do ambiente implementado, para que o passo posterior proceda à captação das informações necessárias.

No terceiro momento, busca-se corrigir todos os erros de compilação e elaboração do ambiente construído, de forma cautelosa, para não haver mudança na lógica implementada anteriormente. Nesse passo, trabalha-se para deixar o ambiente confiável e depurado, a fim de que seja utilizado sem nenhum problema. Os passos dois e três podem ocorrer mais de uma vez, até que o ambiente esteja perfeito para a simulação, executando o ambiente com os estímulos aleatórios.

Seguindo o fluxo, no quarto estágio serão criados os testes direcionados para os *corner-cases* definidos no plano de verificação. Isso é importante, pois o ambiente de simulação com estímulos randômicos tem a possibilidade de não cobrir todos os casos de testes como, por exemplo, os casos menos prováveis de acontecer.

Por último, é feita a verificação das saídas geradas pela execução da simulação. Caso haja alguma diferença na comparação das respostas recebidas, será encontrada nessa etapa uma mensagem comunicando que há erros no *testbench.* Analisa, também, o índice de cobertura que foi obtida durante a simulação; o ideal é que a cobertura seja maior ou igual ao especificado no plano de verificação. Se isso ocorrer, o processo de cobertura é aceito e, então, finalizado o projeto de verificação funcional com êxito.

Pode-se observar na figura 2.6 um gráfico mostrando os estágios do plano de verificação dirigido à cobertura, relacionando esta com o tempo de projeto. Os números correspondem ao fluxo de desenvolvimento do ciclo de vida de um projeto dirigido à cobertura.



1- Plano de verificação e implementação do ambiente 3- *Testbench* Randômico
2- Criar o *testbench* com estímulos diretos 4- *Corners cases* e não cobertos
5- Estímulos para casos reais

Figura 2.6: Estágios da metodologia dirigida à cobertura com relação ao tempo

2.2.3 Verificação Dirigida às Métricas

De acordo com Nick Heaton, em seu artigo denominado "*Maximizing Verification Effectiveness Using Metric-Driven Verification*"[51], ao longo dos últimos anos a verificação dirigida à cobertura tornou-se amplamente utilizada para verificação que requer a obtenção de dados de cobertura num ambiente de simulação. Porém, tal abordagem apresenta muitas limitações que afetam sua usabilidade e escalabilidade. Um exemplo de limitação é não incluir a verificação ou aspectos baseados no tempo, o que seria essencial na definição dos critérios de um circuito.

Verificação dirigida à especificação, como também é conhecida a verificação dirigida a métricas, amplia o escopo da verificação dirigida à cobertura, capturando pontos do ambiente, de tempos em tempos, através de ferramentas como *assertions* ou outros tipos de softwares, verificando as propriedades do ambiente de verificação. Outro benefício é possibilitar a criação de um plano de verificação executável, isto é, que possa ser utilizado diretamente no cenário de verificação, tendo seu progresso medido e suas falhas verificadas.

Assim, o objetivo dessa metodologia é melhorar a qualidade e a produtividade da verificação através dos seguintes princípios:

- Remover a intervenção manual, que é inquestionavelmente lenta e propícia a erros;
- Melhorar o gerenciamento do ambiente de regressão (é uma técnica que consiste na aplicação de testes já feitos em fases anteriores, com a finalidade de constatar que não surgiram novos defeitos em componentes já testados, por ocasião da integração destes com componentes novos) através da ordenação automática das execuções, contribuindo para cobertura e priorização da ordem de regressão.

2.3 Características Desejáveis de um Ambiente Construído Através de uma Metodologia de Verificação Funcional Dirigido à Cobertura

Para desenvolver um ambiente de verificação funcional é necessário estar ciente de algumas características existentes. As características são:

- Plano de verificação
- Geração de estímulos
- Cobertura
- Reuso do ambiente de verificação
- Reuso dos resultados das simulações
- Suporte a desenvolvimento *top-down* e/ou *bottom-up*
- Facilidade de comunicação entre os componentes do ambiente de verificação

2.3.1 Plano de Verificação

O plano de verificação tem sido utilizado para se referir à lista de cenários que devem ser verificados, em sequência, para completar o plano de verificação. Porém, com a complexidade dos módulos atuais e o incremento do número de abstrações, atores e tecnologias envolvidas, o termo vem sendo utilizado também para todo o projeto de verificação existente no sistema, englobando verificação funcional, formal, entre outras necessárias.

Historicamente, a verificação era uma atividade de produção, trabalhada com uma abordagem *bottom-up* para verificar módulos de hardware com bastante confiança, a fim de obter um funcionamento correto do sistema, visto como um todo. Todavia, a popularidade do uso da metodologia de verificação dirigida à cobertura e o desenvolvimento de linguagens especializadas em verificação funcional têm motivado a mudança quanto ao planejamento inicial dos requisitos para todos os tipos de verificação utilizados no projeto, assim como as

tarefas que cada equipe ficará responsável. O plano de verificação deve conter, ainda, todas as ferramentas e os diferentes tipos de abstração que serão utilizados.

Fica nítida a importância que o plano de verificação seja descrito da forma mais detalhada possível, para que todos os membros da equipe de verificação possam utilizá-lo compreensivamente. Os benefícios gerados por um bom plano de verificação são:

- Alinhamento de interpretação dos vários membros da equipe;
- Desenvolvimento comum das funcionalidades pela equipe;
- Rápida definição de *milestones* no projeto;
- Definição do plano da equipe de engenharia, atribuindo os recursos e tarefas aos membros;
- Entendimento comum das tarefas, objetivos e complexidade para todos os membros;
- Refinamento da especificação existente, identificando os buracos de cobertura, ambiguidades, ou interpretações erradas;
- Definição das modalidades de verificação que serão desenvolvidas para DUV (formal, funcional e híbrida por exemplo).

Entretanto, para que o plano de verificação se torne efetivo, deve-se tomar alguns cuidados nas considerações de todos os fatores que tenham impactos no plano e no processo de verificação. Alguns desses fatores são:

- O trabalho de vários especialistas tem que ser monitorado e mesclado;
- Milhares de testes descritos devem ser gerenciados e processados;
- Mudanças no plano devem ser feitas apenas pelo gerente da equipe;
- Gerenciamento das múltiplas tecnologias existentes no fluxo;
- Em grandes empresas, múltiplas equipes devem ser coordenadas;
- Os status devem ser coletados e reportados de acordo com o fluxo de desenvolvimento seguido;
- A ineficiência e a imprevisibilidade inerente ao projeto devem ser gerenciadas sempre.

O caminho seguido pelo plano de verificação é cíclico, que se inicia com a construção do plano e termina com o estudo dos resultados obtidos. Caso esses resultados sejam satisfatórios para o gerente de verificação, então, a verificação é completada; caso contrário, um novo ciclo será executado. Na figura 2.7, é apresentada a ilustração de como ocorre este

processo:

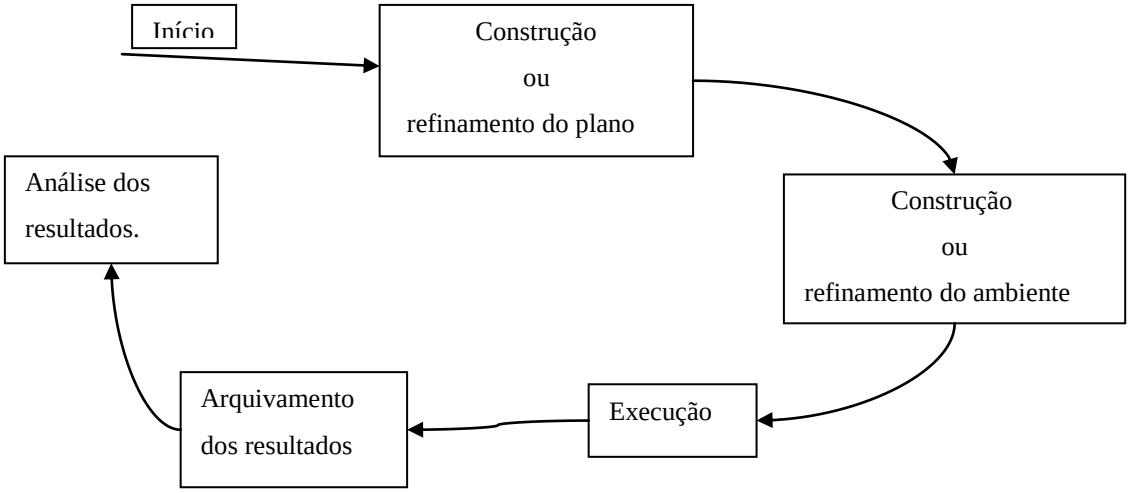


Figura 2.7: Ciclo de execução da verificação

2.3.2 Geração de Estímulos

Um cenário de verificação é composto por uma sequência de transações produzidas, sendo que cada transação contém um conjunto de dados e parâmetros que serão estimulados para verificar o DUV. Por exemplo, uma memória onde o dado de entrada e o endereço são dados que fazem parte da transação.

A geração de estímulos randômicos é a técnica que utiliza geração de estímulos de forma aleatória para o ambiente de simulação. Já a geração randômica é uma técnica importante para a verificação porque uma única simulação pode verificar múltiplos cenários e valores dos dados combinados, e assim reduzir o tempo necessário para a equipe de verificação gerar cada cenário possível.

A geração de estímulos pode ser feita de forma manual ou através de alguma ferramenta. A geração manual pode ser pouco eficiente para alcançar a cobertura, pois nem sempre cobre todas as possibilidades necessárias, enquanto através de ferramentas especializadas há uma chance maior de detectar erros, que podem até mesmo terem escapado da especificação.

Um ambiente randômico não é totalmente aleatório, pois existem formas de controlá-lo. Na linguagem *SystemVerilog*, esse controle ocorre pelo uso de pesos para os valores que poderão ser gerados. Com isso, alguns dados que terão maior peso serão gerados em maior quantidade.

A construção de um ambiente de forma aleatória não é uma tarefa trivial, posto que exige mudanças significativas em relação às técnicas tradicionais de verificação dirigidas a teste. Geração aleatória de cenários requer que cada um dessas entradas e saídas sejam automaticamente verificadas, para ter a certeza de que os estímulos estão sendo gerados e recebidos de forma correta. Isso significa que um ambiente como este necessita de um mecanismo que gere resultados esperados para cada estímulo recebido.

Ferrandi[18], em um artigo sobre o tema, informa que a geração de estímulos randômicos é uma forma adequada para verificar uma máquina de estado finito, demonstrando casos em que o DUV é descrito por uma máquina de estados finitos e para essa máquina é criado um ATPG (*Automatic Test Pattern Generation*), baseado na observação e controle. O autor descreve o algoritmo automático em cinco etapas:

1. Aquisição de dados: Analisa a lista de portas, sentenças, instruções condicionais e transições;
2. Análise das transações: busca o estado inicial e final de cada transação;
3. Numeração de sequência: busca sequenciar os casos de testes para cada vetor que deverá ser verificado;
4. Análise de sequência e produção de restrições: cria um conjunto de restrições correspondente a todas as instruções condicionais e posteriormente aplica-as tanto para sinais quanto para variáveis impossibilitado-as de valores viciados. após a numeração da sequência, para cada uma é criado um conjunto de restrições, que corresponde a todas as instruções condicionais que deverão resultar “verdade” durante a execução da sequência. Na fase final desse passo, é criado um conjunto de diferentes arquivos, e cada um é relacionado com sua sequência;
5. Verificaçcão de restrições e extração de testes: é gerada uma solução (caso exista), satisfazendo todo o conjunto de restrições associado à sequência.

O trabalho de Ziv[19] Probabilistic Regression Suites for Functional Verification, utiliza conjunto de regressões probabilísticas para a geração de estímulos randômicos, conjuntos estes produzidos com o uso de um modelo probabilístico, constatando se cada caso de teste foi coberto. O objetivo principal é diminuir o número de simulações e maximizar a cobertura, utilizando probabilidade na geração de estímulos.

2.3.3 Cobertura

É o processo utilizado, na verificação funcional, para gerenciar a qualidade dos

estímulos randômicos.

A geração de estímulos aleatórios tem o potencial de proporcionar uma melhoria significativa na produtividade e eficiência de verificação. Contudo, sem uma clara estratégia de medida do progresso de verificação, essa eficiência pode ser afetada. Medidas precisas do progresso de verificação requerem que algumas análises sejam feitas para cada simulação randômica utilizada:

- Quais os cenários incluídos no plano de verificação foram gerados na simulação;
- Quais os cenários não incluídos na verificação foram gerados pela simulação;
- Qual a contribuição de cada execução para o progresso da verificação;
- Existência de modificações de restrições necessárias para gerar cenários que faltam.

Informações de cobertura coletadas durante cada simulação randômica são utilizadas para definição da continuidade da simulação, bem como para o incremento do índice de cenários possíveis que já foram cobertos. A cobertura verifica, também, modificações no ambiente de simulação ou novos casos de testes que devem ser gerados nas futuras simulações, e casos que não deviam ser gerados pelo *testbench*.

2.3.4 Suporte à Desenvolvimento *top-down* e/ou *bottom-up*

Os conceitos abordados nesse subcapítulo são informações utilizadas também no processo de desenvolvimento de *software*. O fluxo de desenvolvimento de um ambiente de verificação funcional tem semelhança com o processo de desenvolvimento de software, devido ao ambiente de verificação ser implementado em uma linguagem OO (Orientação Objeto).

2.3.4.1 Top-Down

A visão de projeto *top-down*, tem sua primeira etapa no levantamento de requisitos de todo o sistema, seguido pela especificação, onde é criada uma descrição detalhada do sistema geral. Na especificação é descrito o comportamento do sistema com toda sua funcionalidade, e não como ele é construído. O detalhamento interno do sistema começa a aparecer quando é desenvolvida a arquitetura, que resulta na estrutura do sistema, com todos os seus componentes.

A partir da especificação construída de todos os módulos é implementado o sistema em alto nível, para verificar se todas as funcionalidades da especificação estão sendo supridas. Esse processo de desenvolvimento é adequado para ter a certeza de que o *hardware* final irá

funcionar de forma especificada no documento de especificação. Todos os módulos produzidos nessa fase são implementados utilizando uma linguagem alto nível e podem ser utilizados no ambiente de simulação como o modelo de referência.

Com base nestes componentes, podemos finalmente construir todo o RTL do produto, para posteriormente fazer a integração, a verificação e os testes dos componentes de *hardware*. Em projetos com um nível de complexidade alto, quando necessário, procede-se a uma partição em subprojetos, com o intuito de não afetar os testes do sistema e posteriormente utilizar aceleradores, emuladores e placas de prototipação adequados ao tamanho dos mesmos. As vantagens do uso desse processo de desenvolvimento são:

- Separar o desenvolvimento dos componentes do nível de abstração mais baixa daqueles do nível de abstração mais alta, levando a uma concepção modular
- O design modular confere mais facilidade na distinção das funcionalidades;
- Demonstração clara de todo o "esqueleto" do código, mostrando como os módulos de baixo nível devem ser integrados;
- Menos erros de operações, porque cada módulo tem que ser processado separadamente após todo o sistema ser construído;
- Consome menos tempo de implementação, uma vez que cada programador só é envolvido em uma parte do grande projeto;
- Forma de tratamento dos módulos otimizado, já que cada programador tem de aplicar os seus conhecimentos e experiência para suas partes (módulos);
- Fácil de gerenciar, pois ocorrendo um erro na saída, fácil será identificá-lo, assim como descobrir qual módulo do sistema inteiro está gerando-o.

2.3.4.2 Bottom-up

Bottom-up é o fluxo de desenvolvimento de projeto, consiste na verificação e implementação do sistema de forma separada. A documentação dos requisitos pode ser feita ou na totalidade do sistema, ou para cada parte do sistema que será implementado de forma incremental. A especificação segue de acordo com a documentação dos requisitos, e de maneira isolada são construídas as partes do projeto em desenvolvimento. Depois desta etapa, a integração entre os módulos é testada, de maneira incremental até que todo o sistema esteja integrado.

Esse processo de desenvolvimento é o mais utilizado pelos programadores que trabalham com linguagens OO. Todavia, esta abordagem *bottom-up* tem um ponto fraco:

requer uso de intuição para decidir a funcionalidade que deve ser fornecida por cada módulo. Isso ocorre porque não existe um modelo de referência já desenvolvido para auxiliar no processo de definição das funcionalidades intrínsecas ao projeto. Se um sistema está sendo construído a partir de um sistema preexistente, esta abordagem é mais adequada.

2.3.5 Facilidade de comunicação entre os componentes do ambiente de verificação

A disponibilidade de diversas ferramentas, linguagens e metodologias, dificultou a comunicação entre estas variáveis num mesmo projeto, visto que to das buscam os mesmos objetivos. As principais causas da falta de comunicação são:

- **Diferentes subconjuntos de linguagens** – diferentes partes das linguagens são suportadas ou usadas por ferramentas ou metodologias;
- **Interfaces incompatíveis** – falta de padronização das interfaces e protocolos associados dos componentes de verificação funcional;
- **Dependência tecnológica** – componentes ou bibliotecas muitas vezes dependem de ferramentas específicas para serem compilados ou executados.

A capacidade de comunicação entre os componentes de um ambiente de verificação é um requisito muito importante, pois permite a compatibilidade entre diferentes ferramentas, linguagens e metodologias. Se uma equipe consegue utilizar componentes de verificação de terceiros ou de projetos anteriores, isto representa, sem dúvida, um ganho de produtividade por reusabilidade de componentes, além de aumentar a confiança do ambiente de verificação funcional, que terão componentes já utilizados e validados em outros projetos.

Além das características descritas acima, um ambiente de verificação funcional dirigido a cobertura deve ser desenvolvido seguindo algumas métricas. São elas:

- Suporte a diferentes granularidades
- Completude
- Minimização do esforço manual
- Efetividade

2.3.6 Suporte a Diferentes Granularidades

De acordo com Iman[28], granularidade de verificação é a medida quantitativa de

detalhes que devem ser especificados na descrição e implementação de um cenário de verificação. A granularidade afeta diretamente a produtividade da verificação, já que exige um esforço considerável da equipe de engenheiros para lidar com os objetos.

Existem diferenças entre granularidade de verificação e granularidade de implementação: esta se refere ao nível de abstração utilizado em diferentes fases do fluxo de projeto, por exemplo, o nível de transação utilizado em diferentes fases do projeto e o cenário de verificação descrito para esse modelo,que também são descritos no nível de transação. Já a granularidade de verificação pode ser vista por:

- Descrição do cenário utilizado com uma linguagem de alto nível (Modelo de Referência);
- Implementação do ambiente de verificação utilizando o cenário descrito;

Através da descrição do cenário em alto nível, o engenheiro estará apto a eliminar o baixo nível de detalhamento com que o produto seria descrito. A implementação em alto nível agiliza a codificação do ambiente que irá testar o módulo implementado, e dá suporte à criação de estruturas que poderão ser utilizados por vários projetos diferentes.

2.3.7 Completude

Completude é a medida de quão completa é a verificação de um circuito. Geralmente, acrescenta-se ao plano de verificação um campo denominado de plano de verificação da completude, onde são inseridas as funcionalidades de acordo com um determinado peso. Em tese, todas as funcionalidades devem ser verificadas para uma total completude. Entretanto, de acordo com Iman[28], raramente os planos de verificação são completos e não é possível enumerar todos os *corners cases* de um projeto complexo.

A escolha da metodologia para verificação afeta diretamente a completude desta. Um exemplo disso é uma metodologia que utiliza apenas casos de testes diretos, onde a completude de verificação visa apenas verificar por completo os casos de testes especificados no plano de verificação. Numa metodologia onde estímulos randômicos são gerados, há probabilidade de geração de cenários além daqueles descritos no plano de verificação. Isto posto, nota-se que uma verificação randômica tem uma probabilidade maior de completude que uma apenas dirigida a casos de testes diretos.

2.3.8 Esforço Manual

Refere-se à quantidade de engenheiros exigida para a finalização da verificação

funcional, no tempo estimado. As fontes do esforço manual são:

- Escrita do plano de verificação;
- Construção do ambiente de verificação;
- Execução do cenário de verificação;
- Depuração das falhas do cenário construído.

Algumas atividades da equipe de verificação devem ser feitas por engenheiros, tal como a extração do plano de verificação a partir da especificação do projeto. Esta fase requer um grande esforço manual e um tempo de projeto considerável, visto que não pode ser automatizada.

2.3.9 Efetividade

A execução do plano de verificação geralmente consiste em múltiplas simulações rodando, cada uma com um cenário diferente de verificação. Em geral, nem todo o ciclo de simulação gera um novo cenário, e nem toda simulação é executada em um único cenário. Desse modo, a efetividade na verificação é a medida de quanto tempo de simulação é necessário para cobrir o cenário proposto.

2.3.10 Reuso de Ambiente de Verificação

É a medida utilizada para verificar a porcentagem do reuso de ambientes de verificação, desenvolvidos anteriormente, num mesmo projeto. Algumas formas de medição são:

- Percentual de reuso no ambiente gerado como um todo;
- Percentual de reuso por módulos implementados em nível de sistema.

Mudanças em um projeto podem acarretar em adição de novas funcionalidades ou alteração na arquitetura do ambiente de verificação; o reuso de ambiente auxilia justamente em casos de inserções de mudanças nos módulos criados (ou até novos módulos inseridos).

A principal forma de desenvolver um componente reusável é reunir todos os dados e funcionalidades em um módulo com interface bem definida; esta determinará como modificar, operar e interrogar (extrair dados) o componente.

Um paradigma muito utilizado para a criação dos ambientes de verificação é o OO (Orientação Objeto), por suportar a criação de objetos que se comuniquem através das interfaces, tornando o ambiente mais propício ao reuso. Num ambiente padronizado com o

paradigma OO, o acesso aos dados só é possível através da interface por diferentes formas de chamadas: chamada de função, hierarquia, classes parametrizadas e alteração em tempo de execução do comportamento ou estrutura.

2.3.11 Reusabilidade dos Resultados das Simulações

Trata-se de um dos métodos de medição da quantidade de dados produzidos durante uma simulação, que poderão ser reutilizados para responder questões ocorridas durante um tempo posterior de execução da simulação. Em tese, toda simulação pode requisitar dados da simulação anterior, e a partir dessa gerar novos dados.

É de fácil observação em um ambiente de verificação em que é inserido um modelo de cobertura. Para implementar a cobertura, é necessária alguma forma de armazenamento dos casos já simulados, para que haja a possibilidade de fazer uma checagem com todos estes casos já simulados e armazenados, e assim verificar se toda a cobertura foi realizada. Caso contrário, uma nova transação é gerada e o fluxo continua até que a cobertura seja realizada de acordo com a especificação contida no plano de verificação.

2.4 Suporte de SystemC e SystemVerilog

A área de verificação funcional tem suporte de algumas linguagens especificas para construção de ambientes de verificação. Dentre elas, podemos citar SystemC e SystemVerilog, linguagens que seguem o paradigma de orientação objeto e que são utilizadas pelas metodologias estudadas nesse trabalho. Abaixo será explicado um pouco de cada uma, deixando o leitor ciente de suas principais características.

2.4.1 SystemC

É uma biblioteca de classes C++ de código aberto, com o intuito de descrever componentes de hardware e sistemas, permitindo que estes possam ser compilados e executados usando compiladores genéricos, como o GNU GCC. Foi criada pela Synopsys Inc., em 1999, e é um padrão IEEE 1666-2005. Estruturada na programação dirigida a eventos *SystemC*, pode se comunicar em tempo de simulação com tipos de dados C++. Atualmente muito utilizada em *system-level modeling* (SLM), exploração de arquitetura, modelagem de performance, desenvolvimento de software, verificação funcional, e síntese de alto nível.

Associa-se mais comumente a *SystemC* como uma linguagem *Electronic system level* (ESL) e com *Transaction-level modeling* (TLM).

SystemC é poderosa e genérica para modelar os mais variados sistemas, mas não possui em seu conjunto um suporte nativo para verificação. Foi criada, dentro do próprio consórcio *SystemC*, a biblioteca *SystemC Verification* (SCV), que buscou reduzir as dificuldades enfrentadas na utilização de *SystemC* para verificar sistemas. Apesar de cobrir de maneira bastante satisfatória a parte de geração de estímulos, a biblioteca SCV não contempla outros aspectos também fundamentais para a verificação funcional, como a cobertura funcional e o conceito de *assertions*.

2.4.2 SystemVerilog

A Introdução de *SystemVerilog* como uma nova linguagem de descrição de hardware e verificação funcional é motivada pela necessidade de ferramentas poderosas que facilitem a implementação de metodologias mais recentes. O objetivo principal da criação é unir todo o fluxo *front-end* de um projeto de hardware em uma única linguagem. O padrão IEEE 1800 ™*SystemVerilog* é a primeira linguagem de descrição de hardware e de verificação (HDVL) unificada da indústria. *SystemVerilog* é uma extensão importante do padrão IEEE 1364 ™linguagem Verilog, desenvolvido originalmente pela empresa Accellera para melhorar drasticamente a produtividade no projeto, com um número elevado de portas lógicas. Orienta-se principalmente para a implementação de chips e fluxo de verificação. Além disso, possui alguns recursos de modelagem a nível de sistema. As principais extensões criadas foram:

- Suporte a modelagem e verificação em nível de transação. *SystemVerilog* Direct Programming Interface (DPI) habilita a linguagem a fazer ligações a funções C/C++/*SystemC*. Dessa forma, torna-se a primeira linguagem baseada em Verilog a oferecer cossimulação eficiente com blocos *SystemC*, muito importante para fazer ligação da modelagem de sistema com o ambiente de verificação e a implementação RTL do chip.
- Um conjunto de extensões para atender às exigências de design avançado que utilizam o modelo de programação orientado a objetos. Interfaces de modelagem que aceleram consideravelmente o desenvolvimento de projetos, eliminando as restrições sobre as conexões de porta do módulo, dentre outros.
- Um novo mecanismo de apoio à verificação, chamado de *assertions*, permitindo uma metodologia "*design for verification*" num projeto de verificação.

SystemVerilog descendeu da linguagem Verilog, assim como mostrado na figura 2.8:

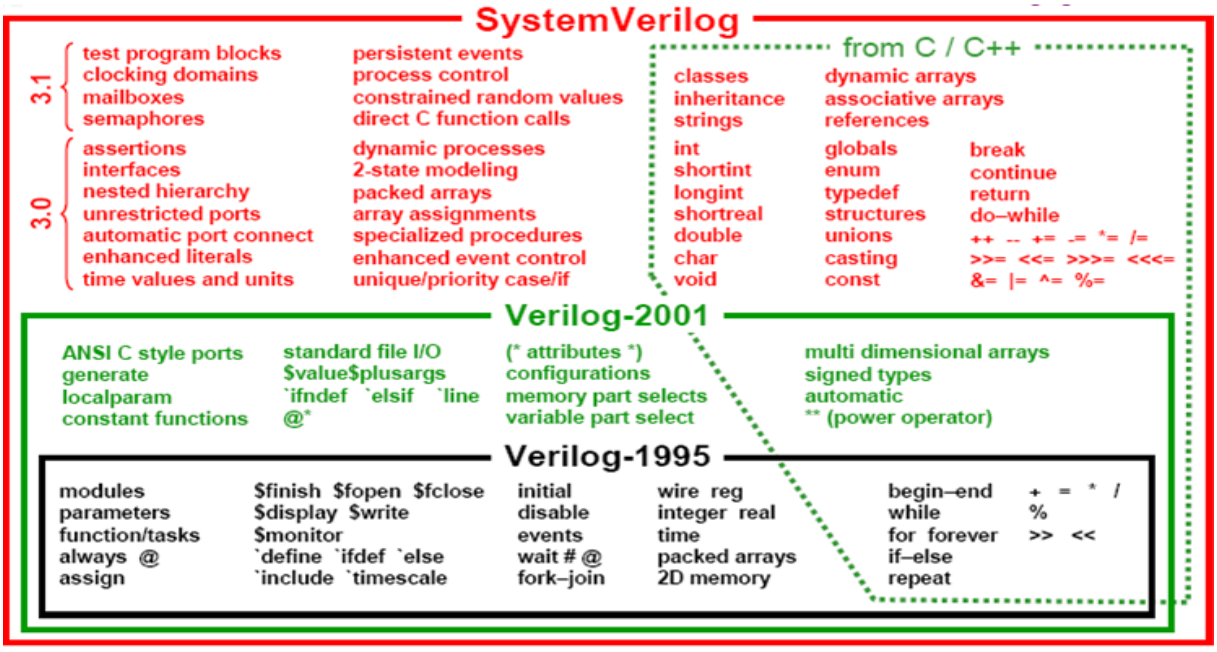


Figura 2.8:Estrutura da linguagem *SystemVerilog*[54]

SystemVerilog possui várias diferenças quando comparada a *SystemC*, dentre elas:

- *SystemC* é uma linguagem mais adequada para simulações de hardware em alto nível, tais como funções. De acordo com o comentário do engenheiro de design John Rinderknecht, da Motorola Labs[53], fazer design em *SystemC* é uma péssima escolha: na sua opinião, foi mais difícil ler o código comparado a Verilog, "devagar e complicado de escrever", difícil de depurar, ineficiente para a reutilização, representando "uma íngreme curva de aprendizado para todos".
- *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, como portas, sinais e módulos, através da biblioteca Standard *Template* Library. Dessa maneira, para criar modelos (i.e representações de um circuito digital ou parte dele) em *SystemC*, é necessário que o engenheiro aprenda C++ e tal biblioteca. De maneira oposta, *SystemVerilog* já fornece de maneira nativa recursos para modelagem de hardware.
- *SystemC* não possui recursos em sua linguagem para trabalhar com cobertura. Do lado oposto, *SystemVerilog* possui características nativas para medição da cobertura funcional.

Capítulo 3 Estado da Arte

Este capítulo detalha o estado da arte no âmbito da verificação funcional. Foram pesquisadas algumas metodologias e ferramentas que auxiliam no desenvolvimento de *testbenches*, que serviram de referencial teórico para o trabalho desenvolvido. Desse modo, o objetivo dessa seção é discutir alguns trabalhos relacionados à matéria, com a finalidade de dar uma visão geral dos vários aspectos abordados atualmente por outros pesquisadores da área.

3.1 Ferramenta de Geração Semiautomática de *Testbench* eTBc

A ferramenta eTBc[38] foi desenvolvida pelo aluno Isaac Maia Pessoa, no seu trabalho de mestrado. Com esse software, pode-se desenvolver *testbenches* de forma semiautomática, através do uso da ferramenta. Após isso, será necessária a complementação do código gerado; complementação esta que consiste na implementação do plano de cobertura funcional, incluindo a geração de estímulos de acordo com as especificações do projeto.

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

3.1.1 Funcionamento

Para a geração de código, o eTBc recebe como entrada dois arquivos: Um arquivo *Transaction Level Netlist* (TLN) descrito pelo engenheiro de verificação funcional (usuário da ferramenta), e um molde (*template*) de um dos elementos de *testbench* a ser gerado; esse molde pode ser construído pelo responsável pela metodologia de verificação funcional ou pelo desenvolvedor da ferramenta. A figura 3.1 mostra uma representação arquitetural do eTBc.

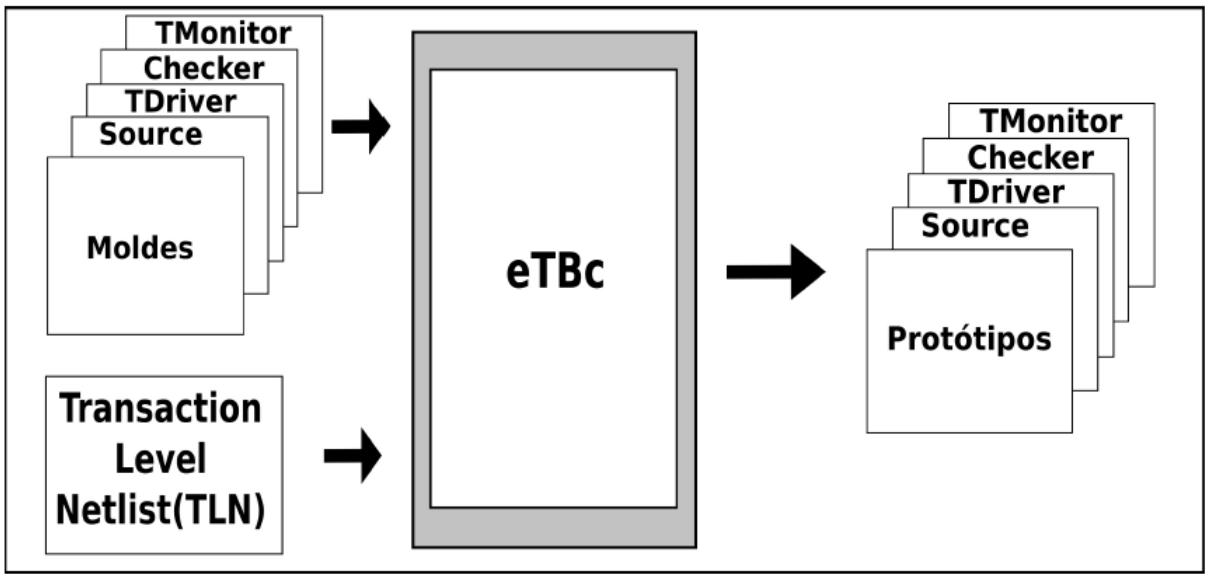


Figura 3.1: Representação arquitetural da ferramenta eTBc[38]

Os moldes desenvolvidos para a ferramenta utilizam a linguagem eTL (*eTBc Template Language*), que caracteriza-se por ser flexível, permitindo a criação de estruturas sintáticas variadas. Isso possibilita que a equipe de verificação funcional construa moldes para as suas necessidades de acordo com a linguagem escolhida para o plano de verificação do projeto. A linguagem eTL possui algumas palavras reservadas que, à medida que vão sendo detectadas pela ferramenta eTBc, é feita uma análise para identificar a devida geração de código a ser usado em substituição às mesmas. Todas as palavras reservadas nessa linguagem são definidas sintaticamente na seguinte forma: $$(palavra-reservada). A figura 3.2 mostra uma parte do código desenvolvido para servir de *template* para a geração dos componentes do *testbench.*

```
$$(foreach)  $$(module.in)
   ovm_put_port #(pkt_$$(i.name)) $$(i.name)_to_rem;
   ovm_put_port #(pkt_$$(i.name)) $$(i.name)_to_dr;
$$(endfor)
   function new(string name, ovm_component parent);
     super.new(name,parent);
   endfunction
```

Figura 3.2: Exemplo de um *looping* na linguagem eTL

Algumas características da linguagem eTL devem ser ressaltadas: capacidade de *looping* em até dois níveis, suporte a condicional, capacidade de atribuição de variáveis de

qualquer tipo de linguagem para a semântica eTL, e possibilidade de acesso ao arquivo eDL para ler o objeto requisitado.

A linguagem eDL é usada para que o usuário da ferramenta possa elaborar os TLNs referentes ao seu projeto de verificação funcional. A figura 3.3 mostra um exemplo de arquivo descrito com a linguagem eDL.

```
struct packet{
trans{
int i;
}
signals{
signed[3] data;
}
}
module dif{
input packet packet_in;
output packet in_sat;
}
module sat{
input packet in_sat;
output packet packet_out;
}
```

Figura 3.3: Linguagem eDL para escrita da TLN

Para a escrita da TLN foi criada uma linguagem eDL. Conforme exposto acima, existem palavras reservadas para a descrição da TLN. Alguns exemplos são:

- module: indica que será iniciado a descrição do módulo RTL.
- trans: faz a indicação do que irá conter a transação criada.
- signals: entre as chaves dessa palavra reservada são descritos os sinais existentes no módulo.
- input: especifica as transações de entrada do DUV que será verificado.
- output: especifica as transações de saída do DUV.

As principais características dessa linguagem são: poucas palavras reservadas coerentes com o seu significado e facilidade de aprendizagem do modo de descrição.

3.2 Metodologias de Verificação Funcional

Existem no mercado diversos tipos de metodologias de verificação funcional e ferramentas que auxiliam um engenheiro de verificação a fazer o seu trabalho. Serão detalhadas as principais técnicas estudadas para o desenvolvimento deste trabalho.

3.2.1 VeriSC

A metodologia de verificação VeriSC pode ser aplicada para a verificação funcional de circuitos digitais síncronos que utilizam um único sinal de relógio. O fato da utilização de apenas um *clock* não inviabiliza a aplicação em circuitos maiores que tenham mais de um, se este puder ser dividido em vários circuitos cada um com único *clock*.

A metodologia VeriSC é composta de um fluxo de verificação que não se inicia pela implementação do DUV. Nesse fluxo, a implementação do *testbench* e do modelo de referência antecedem a implementação do DUV. Para permitir a implementação antecipada do *testbench*, a metodologia incorpora um mecanismo para simular a presença do DUV com os próprios elementos do *testbench*, sem a necessidade da geração de código adicional, que não seja reutilizado posteriormente. Com esse fluxo, todas as partes do *testbench* podem estar prontas e validadas antes do início desenvolvimento do DUV.

O *testbench* desenvolvido usando a metodologia VeriSC possui as seguintes características básicas:

- É dirigido por cobertura, devendo a verificação funcional parar apenas quando os critérios de cobertura funcional forem alcançados. Além disso, os estímulos são direcionados pelo progresso da cobertura funcional: se a cobertura não estiver evoluindo, os estímulos devem ser mudados e direcionados para que a cobertura funcional seja atingida.
- Possui randomicidade direcionada, de forma que os estímulos randômicos são direcionados para que a cobertura seja atingida, como explicado anteriormente.
- Compara automaticamente os resultados vindos do Modelo de Referência e do DUV. Tal verificação ocorre no módulo *Checker*.
- O *testbench* é baseado em transações. Os estímulos gerados pelo módulo Source do *testbench* são todos no nível de transações.
- Utiliza a ferramenta *eTBc* para fazer a geração semiautomática dos módulos.

3.2.1.1 Arquitetura de VeriSC

A arquitetura de VeriSC é mostrada na figura 3.4, e posteriormente explanada:

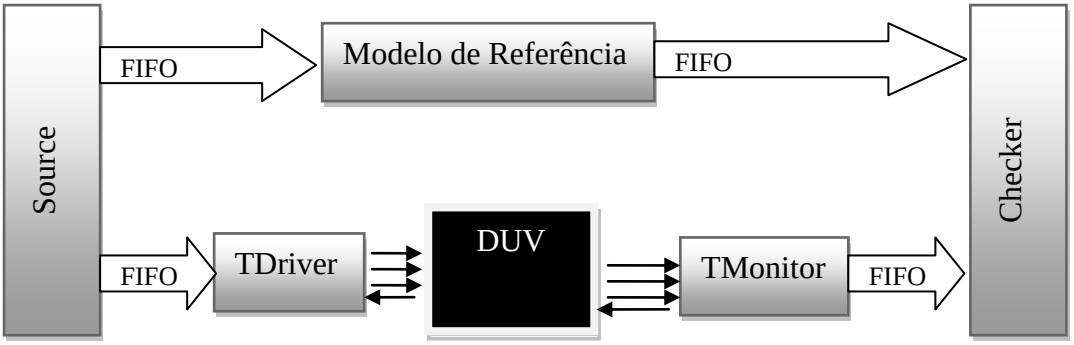


Figura 3.4:Arquitetura do testbench suportado pela metodologia VeriSC

É importante detalhar os componentes da arquitetura do *testbench* suportado pela metodologia VeriSC:

- ***Source*** – Módulo responsável pela criação e geração das transações que estimularão o projeto em verificação (DUV). As transações geradas são escritas, de maneira simultânea, em duas filas ordenadas (estruturas FIFO), de modo que os demais componentes que fizerem a leitura não percam o sincronismo. Os casos de testes são implementados neste componente; assim, para cada instância diferente do sistema que está sendo verificado, é necessário um *Source* específico. Os estímulos recomendados para serem gerados pelo *Source* são: estímulos direcionados, estímulos para situações críticas, estímulos randômicos e casos de testes reais.
- **T*Driver*** – Módulo responsável pela tradução das transações em estímulos baseados em sinais que são recebidos pelo DUV. O T*Driver* é o componente responsável pela leitura das transações geradas pelo *Source*, convertendo-as em estímulos baseados em sinais, sempre em observância ao protocolo de comunicação (*handshake*) associado. Somente é utilizado para inserir estímulos, não sendo capaz de capturar nenhuma informação fornecida pelo sistema.
- ***Reference Model*** – Módulo que implementa, em alto nível de abstração, o funcionamento do DUV que está sendo verificado, sendo normalmente atemporal e funcionando em nível de transação. Este componente recebe as transações geradas pelo *Source* através de uma interface TLM FIFO, processando-as para, em seguida, gerar novas transações, contendo as respostas que serão comparadas com

o resultado dado pelo sistema que está sendo verificado. Nesse módulo, também são inseridos construtores disponíveis numa biblioteca desenvolvida pela autora da metodologia, para o cálculo da cobertura.

- **T*Monitor*** – Módulo responsável pela captura dos sinais gerados pelo DUV e geração de uma transação contendo os valores dos mesmos. Funcionando como função inversa do T*Driver*, o T*Monitor* só captura os sinais gerados pelo sistema, sendo incapaz de gerar estímulos. Também é inserido mecanismos para o cálculo da cobertura funcional a nível de sinal utilizando uma biblioteca específica.
- **Checker** – Módulo responsável pela comparação das transações geradas pelo *Refmod* e pelo *Monitor*. Uma vez que o sincronismo é garantido pelo uso das FIFOs, o Checker carrega as transações disponíveis e faz a comparação, procurando por diferenças entre elas. Quando diferenças são detectadas, mensagens de erro são geradas, informando o valor esperado e o valor recebido, além de outras informações relevantes.

3.2.1.2 Fluxo de Desenvolvimento do Testbench

A metodologia de verificação funcional VeriSC define quais componentes serão utilizados no processo de verificação funcional, assim como também inclui a organização arquitetural do *testbench*, semiautomatização da geração dos componentes e um conjunto de passos para o desenvolvimento *top-down* do *testbench*. O fluxo de desenvolvimento do *testbench* pode ser visto na figura 3.5.

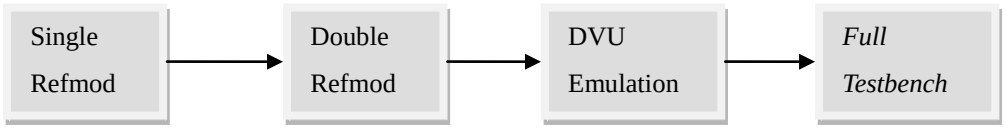


Figura 3.5: Fluxo de desenvolvimento de testbench com VeriSC

O fluxo para o desenvolvimento do ambiente de verificação funcional proposto em VeriSC está descrito a seguir.

3.2.1.2.1 Single Refmod

O objetivo dessa etapa é validar a capacidade de interação dos componentes. Os componentes que fazem parte dessa etapa são: *pré-source*, modelo de referência e *Sink*. A

forma de conexão dos componentes é mostrada na figura 3.5.

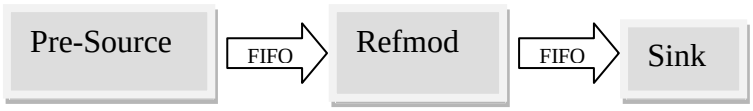


Figura 3.6: Single Refmod

Na figura 3.2 pode-se perceber que o modelo de referência já é estimulado, confirmando seu funcionamento através dos resultados lidos no módulo Sink.

O componente pré-source é um subconjunto do módulo Source, com quase as mesmas funcionalidades. A diferença é que ele gera estímulos somente para o Modelo de Referência e não pra o DUV. Já o bloco Sink possui um subconjunto das funcionalidades do Checker. Ele recebe os dados de saída apenas do Modelo de Referência, tendo assim somente uma FIFO de entrada.

3.2.1.2.2 Double Refmod

O objetivo dessa atividade é validar o processo de geração de estímulos e do analisador de respostas. O gerador de estímulos e o comparador são desenvolvidos nessa etapa, e ligados às duas instâncias do modelo de referência. A figura 3.3 mostra o *testbench* desenvolvido na atividade de *Double* Refmod.

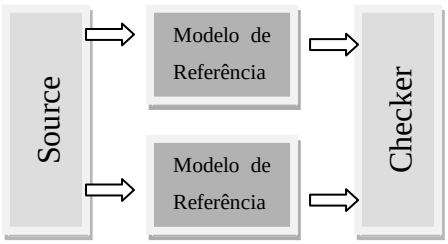


Figura 3.7: *Testbench* desenvolvido no *Double* Refmod VeriSC

3.2.1.2.3 Duv_Emulation

Para a construção final do ambiente de verificação é necessário que o *testbench* possa se comunicar com o DUV através de sinais digitais. Assim sendo, o DUV*_Emulation* tem o objetivo de criar esse mecanismo de comunicação e sempre monitorar os sinais que estão sendo gerados para o DUV, bem como as respostas enviadas por ele. A figura 3.4 mostra o *Testbench* criado, antes que o DUV seja inserido no ambiente de simulação.

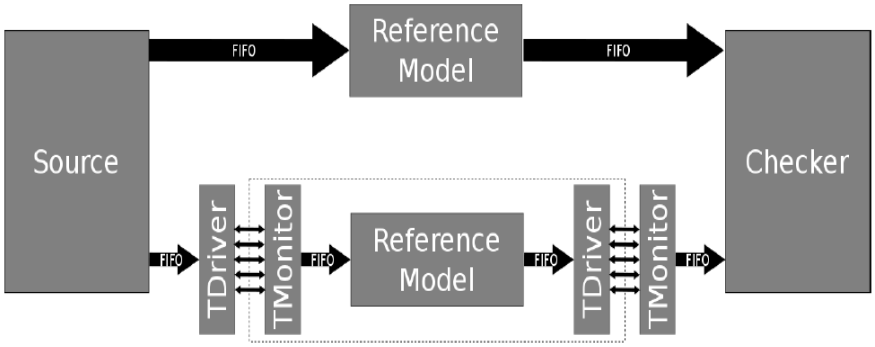


Figura 3.8: Testbench construído na atividade de DUV_Emulation

Nessa etapa o *testbench* é criado por completo, precisando do DUV apenas para iniciar o processo de verificação de suas funcionalidades. A etapa de *Full Testbench* apenas retirará a emulação do DUV, e fará a inserção do DUV desenvolvido.

3.2.1.3 Análise da Metodologia VeriSC

Os pontos fortes dessa metodologia são: suporte ao desenvolvimento do *testbench* antes da inserção do DUV; reuso dos componentes desenvolvidos durante o desenvolvimento do fluxo; suporte a geração semiautomática do *testbench;* e o suporte ao desenvolvimento *top down*. O desenvolvimento de uma biblioteca de cobertura é de suma importância para que o engenheiro possa observar a credibilidade da verificação funcional produzida.

Alguns incrementos poderiam ser desenvolvidos na metodologia para torná-la mais eficiente, tais como descentralização da geração de estímulos automáticos, suporte a comunicação bidirecional e o suporte ao uso de *Assertions.*

3.2.2 OVM (Open Verification Methodology)

Resultante de um esforço conjunto das principais empresas de *Electronic Design Automation* (EDA) do mundo, Cadence® e Mentor®, a OVM tem como objetivo a padronização e interoperabilidade de componentes de verificação. É um superconjunto de duas metodologias de verificação funcional já consolidadas: a *Universal Reuse Methodology* (URM) da Cadence® e a *Advanced Verification Methodology* (AVM) da Mentor®.

OVM é uma metodologia de verificação funcional aberta, interoperável com múltiplas linguagens e simuladores. É detalhada em dois documentos: *Open Verification Methodology class reference*[12] e *Open Verification Methodology user guide*[11] , sendo seu ambiente de verificação composto por componentes UVC (*Universal Verification Component*) reusáveis. O conjunto de UVCs já desenvolvidos com a metodologia OVM é denominado de OVC (OVM *verification components*).

Os UVCs são componentes que devem ser reusáveis, pré-verificados, configuráveis e *plug-and-play*, de modo a serem utilizados em diferentes tipos de projetos. Cada UVC segue uma arquitetura, que consiste em um conjunto de elementos para controlar, estimular e recolher informações de cobertura para uma interface específica. Após o UVC ser validado, este começa a fazer parte da OVC e pode ser utilizado em qualquer projeto que utilize a metodologia OVM para verificação funcional.

A metodologia OVM inclui uma biblioteca de verificação funcional em *SystemVerilog*, e também disponibiliza seu código fonte. A arquitetura do *testbench* da metodologia OVM pode ser vista na figura 3.5.

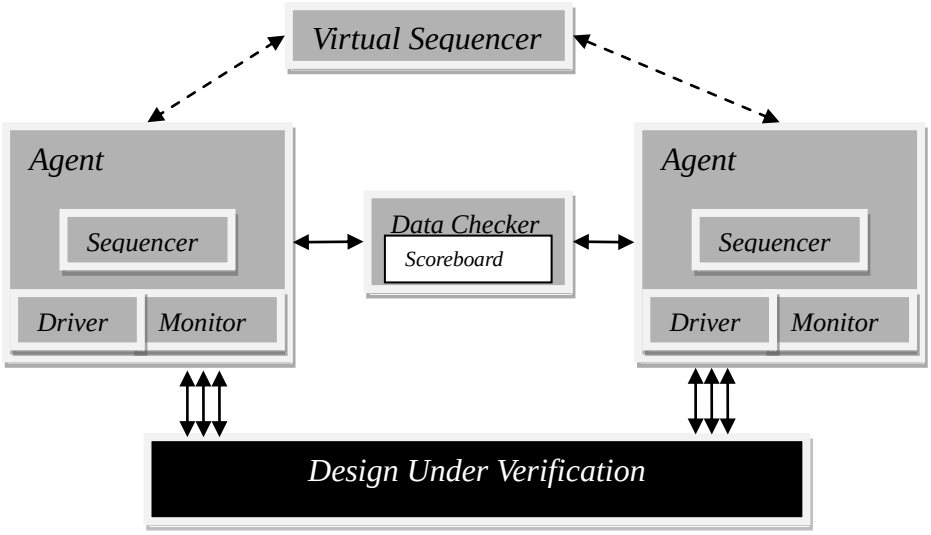


Figura 3.9: Arquitetura do testbench da metodologia OVM

- ***Virtual Sequencer*:** é responsável pela coordenação das sequências que serão geradas nos *Sequencer* instanciados no ambiente, definindo a ordem de geração das transações e criando o cenário dos casos de testes. Por conter informações específicas da instância do sistema que está sendo verificado, este componente não pode ser reusado em outros contextos.
- ***Sequencer*:** cria as transações que irão estimular uma determinada interface, através da geração de estímulo e obtenção da resposta de saída do DUV. Pode ser reusado em projetos distintos, ao contrário do *Virtual Sequencer*.
- ***Driver*:** também conhecido como Bus Functional Model (BFM), o *Driver* é uma entidade ativa que realiza a injeção e captura de estímulos, convertendo as requisições

em nível de transação (TLM) do Sequencer em estímulos baseados em sinais, e vice-versa.

- ***Monitor*:** é uma unidade passiva do ambiente de verificação, responsável pela coleta de informações de cobertura e de checagem de dados através da extração das informações que são transmitidas e recebidas do DUV. Com estas informações, as transações que serão utilizadas pelo *Data Checker* podem ser geradas, e a análise de cobertura funcional pode ser feita, além de possibilitar a checagem de eventuais violações de protocolo de comunicação.
- **Data Checker:** é o equivalente ao Refmod e Checker do VeriSC, sendo responsável tanto pela modelagem do comportamento do sistema, como pela checagem dos resultados gerados. Também é chamado de ***ScoreBoard***, e recebe as informações dos seus monitores associados, sendo capaz de inferir, a partir das entradas, as possíveis saídas geradas. Após gerar a possível saída, faz a comparação com a transação vinda do *Agent*.

A metodologia OVM não tem um fluxo de construção de *testbench* especificado. Partindo da premissa de que todas as funcionalidades de um DUV devem ser verificadas, seria necessário uma UVC para cada interface de cada módulo contido no DUV. Logo, o fluxo de desenvolvimento deve ser *bottom-up*, testando funcionalidade por funcionalidade.

3.2.2.1 Análise da Metodologia OVM

Obviamente, OVM representa um excelente esforço da indústria para a padronização da verificação funcional. No entanto, essa metodologia foi criada para a geração de um OVC após um DUV ser criado, ou seja, não é apta a criar, executar e validar o ambiente de verificação sem que um modelo comportamental (UVC) exista. Assim, teria que ser desenvolvido um novo UVC já com o DUV implementado. Caso já exista um UVC desenvolvido para o DUV, pode ser reusado de forma simples.

A metodologia OVM não suporta a geração automática (ou semiautomática) do *testbench*, porém suporta a verificação em qualquer tipo de circuito. Com relação a suporte de cobertura, OVM realiza o suporte através da biblioteca desenvolvida em SystemVerilog.

3.2.3 BVM (Brazil-IP Verification Methodology)

Diante da possibilidade de melhoria da metodologia VeriSC, surgiu a ideia de

reformulá-la, implementando-a em uma linguagem com mais suporte para verificação funcional. BVM, da mesma forma que VeriSC, tem a construção do *testbench* feita de forma incremental. Porém, há uma sutil diferença entre os passos de tais metodologias: a inserção de um elemento responsável por controlar e monitorar os sinais do protocolo de comunicação entre o DUV e o *testbench*, que foi chamado de *Actor*. Em projetos que exigem uma integração de vários módulos, esse elemento é de grande valia, devido à eficiência em encontrar erros de comunicação entre eles. Vale ressaltar que, assim como VeriSC, a construção do *testbench* em BVM é realizada antes da criação do DUV. Na figura 3.6 podemos observar a arquitetura de *testbench* da metodologia BVM:

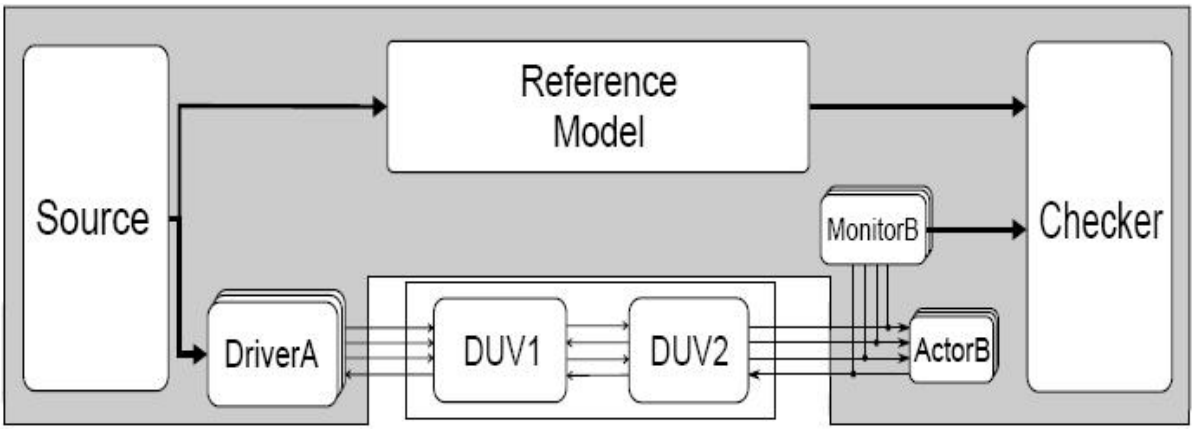


Figura 3.10: Arquitetura do *testbench* da metodologia BVM[37]

A Metodologia utiliza *SystemVerilog* como linguagem padrão e a biblioteca de OVM para a criação do ambiente de verificação; esse é um fator muito importante, pois utiliza uma biblioteca já difundida e bem documentada como base de criação do *testbench*.

O fluxo de construção do *testbench* é exatamente igual ao da metodologia VeriSC.

3.2.3.1 Análise da Metodologia BVM

O problema da metodologia BVM é que, assim como VeriSC, não dá suporte à comunicação bidirecional. Podemos observar este fato na figura 3.6, sendo a interface de ligação do *Source* com o *Reference Model* e o *Driver* uma FIFO unidirecional de duas saídas.

O suporte ao desenvolvimento do *testbench* antes da inserção do DUV é um dos pontos positivos dessa metodologia; outros merecem ser mencionados, como o reuso de componentes durante o processo de desenvolvimento e validação do ambiente, o suporte a geração automática (ou semiautomática) do *testbench* e o suporte a cobertura funcional.

3.2.4 IVM (Interoperable Verification Methodology)

A metodologia IVM foi fruto do estudo de duas metodologias (VeriSC e OVM), em observância às virtudes e defeitos de cada uma delas, com intuito de gerar uma metodologia melhorada.

A arquitetura da metodologia IVM pode ser observada na figura 3.10:

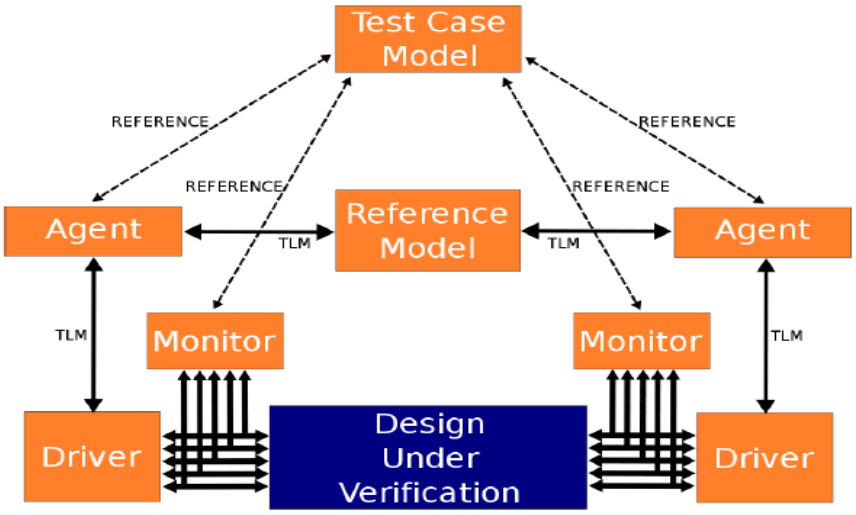


Figura 3.11: Arquitetura do *testbench* da metodologia IVM[40]

São componentes do ambiente de verificação da metodologia IVM:

- **Test Case Model:** é responsável pela coordenação dos agentes que estimularão o DUV, realizando função similar ao *Virtual Sequencer* do OVM, modelando a especificação dos casos de teste e permitindo que todas as funcionalidades fornecidas pelos agentes sejam acionadas de maneira coordenada. Este módulo possui referências aos *Agents* e *Monitor*s do ambiente, através das quais realiza o controle, proporcionando uma visão centralizada de todas as funcionalidades. Os casos de testes são implementados por extensões de classes, permitindo assim que novos testes sejam criados com a certeza de que as propriedades anteriores do ambiente serão mantidas. Por conter informações específicas da instância do sistema que está sendo verificado, este componente não pode ser reusado em outros contextos, ou seja, na verificação de outros módulos ou projetos.
- **Agent:** reúne todas as funcionalidades de um determinado componente externo ao sistema, proporcionando ao *Test Case Model* um conjunto de funcionalidades que poderão compor seus casos de teste através da geração e captura de transações que

irão estimular uma determinada interface do sistema. Além dos papéis de captura e geração de transações, este componente também é responsável pela comparação das respostas recebidas com as respostas geradas pelo *Reference Model*, sendo alheio ao contexto em que está inserido, podendo ser reusado em diversos projetos, assim como o *Sequencer* do OVM.

- ***Reference Model*:** é responsável por modelar idealmente o comportamento do sistema que está sendo verificado, normalmente implementado em um nível mais alto de abstração e sem detalhes temporais. Diferentemente do *Data Checker* do OVM, que modela o comportamento e faz checagens de respostas, o *Reference Model* apenas modela o comportamento do sistema em questão, permitindo que este componente possa ser estruturado hierarquicamente. É implementado em nível de transação (TLM), o que facilita o enfoque na funcionalidade que se deseja modelar, além de permitir uma simulação bem mais rápida.
- ***Driver*:** também conhecido como *Bus Functional Model* (BFM), o *Driver* é uma entidade ativa que realiza a comunicação entre interfaces RTL e TLM, convertendo as requisições em nível de transação (TLM) do componente *Agent* ou *Reference Model* em estímulos baseados em sinais, e vice-versa.
- ***Monitor*:** é um componente do IVM responsável pela coleta de informações de cobertura funcional e de checagem de protocolo de comunicação, funcionando através da extração das informações (como endereços acessados ou operações realizadas) que são transmitidas pelos sinais monitorados. Este módulo checa a cobertura utilizando as informações obtidas e verifica a conformidade do protocolo através de regras descritas pelo projetista.

A Metodologia IVM foi concebida para a linguagem *SystemC*, e utiliza a técnica de cobertura criada pelo autor, baseada em espaço de valores e probabilidade. Foi desenvolvido um mecanismo de decisão de parada para as simulações, que toma por base o comportamento da cobertura; esta sempre aumentará ou ficará constante, e após um determinado tempo (razoavelmente grande) de simulação, tenderá a se estabilizar em um determinado valor. Existe um mecanismo que finaliza a simulação quando se atinge um percentual fixo por um determinado tempo. De acordo com os estudos de casos exemplificados na proposta (PRADO, 2009) a cobertura funcional utilizada nunca testou todas as possibilidades devido ao conjunto de possibilidade ser grande.

Além do descrito acima, existe também em IVM a ausência do uso de automação para geração dos módulos do ambiente de verificação, como ocorre na metodologia VeriSC. Apenas provê as interfaces para cada componente existente na arquitetura do bloco. IVM e VeriSC não fazem uso de *assertions* no ambiente de verificação.

3.2.4.1 Fluxo de desenvolvimento

O fluxo de atividades do IVM é *bottom-up,* onde são desenvolvidas três atividades de validação do *testbench* para cada módulo implementado, e apenas no final é construído o ambiente de verificação funcional do DUV total.

3.2.4.1.1 Sanity Checking

Atividade semelhante ao Double Refmod da metodologia VeriSC. Tem o objetivo de validar os seguintes componentes: *Test Case Model, Referencial Model* e *Agent.* A arquitetura de *testbench* para esta atividade é mostrada na figura 3.8

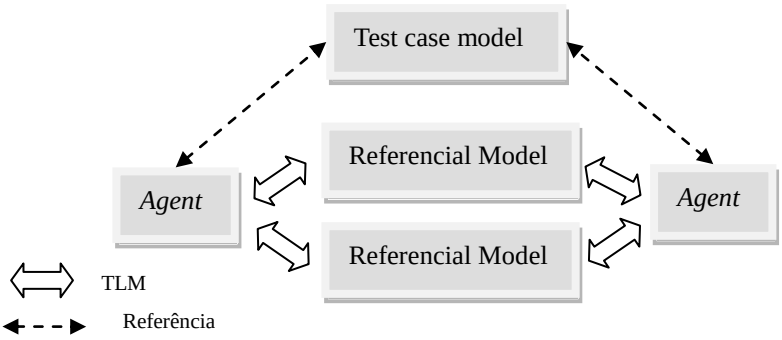


Figura 3.12: Testbench da atividade Sanity Checking

3.2.4.1.2 Interface Refinement

Essa atividade tem como objetivo refinar a interface de comunicação baseada em transação (TLM) para interfaces baseadas em sinais. Nesta etapa são inseridos os *drivers* e *monitors* para cada interface. A figura 3.9 mostra a arquitetura do ambiente de verificação criado.

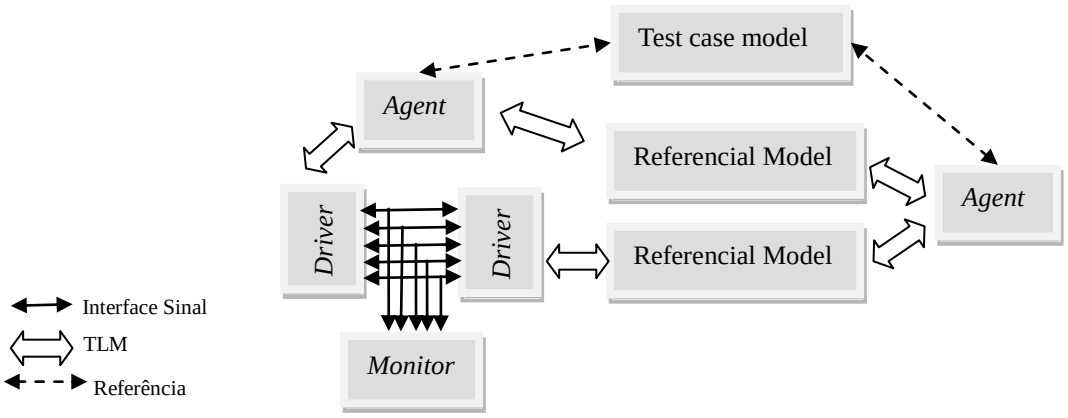


Figura 3.13: *Testbench* da atividade Interface Refinement

3.2.4.1.3 Enviroment Validation

Depois de validar cada interface de comunicação de sinal, essa atividade junta todas as interfaces num único *testbench* e emulando a presença do DUV no ambiente de verificação. A arquitetura do ambiente de verificação pode ser visto na figura 3.10.

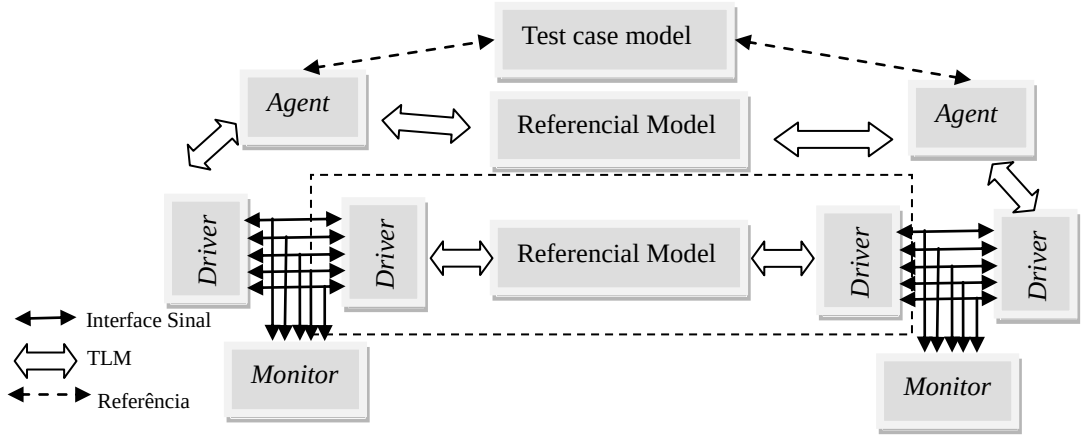


Figura 3.14: *Testbench* da atividade *Environment Validation*

A figura 3.10 cria o ambiente de verificação completo para depois inserir o DUV. Os *drivers* e o modelo de referência inseridos no retângulo pontilhado emulam a presença do DUV no ambiente de verificação.

3.2.4.2 Análise da Metodologia IVM

Como já mencionado, a metodologia IVM provê o suporte ao desenvolvimento do ambiente de verificação funcional antes da inserção do DUV. Além disso, o reuso de componentes durante o processo de desenvolvimento do *testbench*, o suporte a comunicação bidirecional e o suporte a cobertura são características importantes da metodologia. Todavia, a

metodologia não tem o suporte de geração automática ou semiautomática de *testbench*.

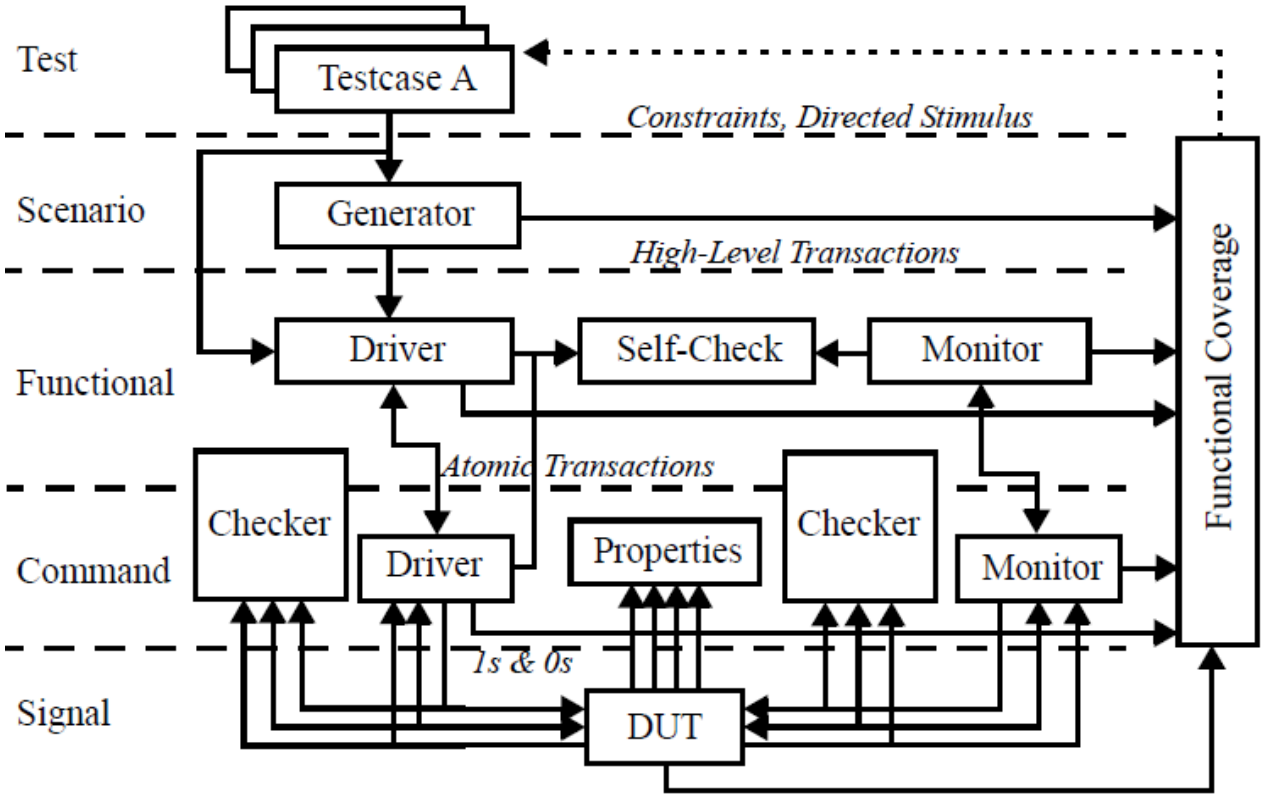
3.2.5 VMM

A metodologia VMM (Verification Methodology Manual)[9] foi documentada por um livro de autoria conjunta da ARM e Synopsys[54]. O maior marco dessa metodologia foi ser desenvolvida seguindo o paradigma de linguagem orientada a objetos. Porém, para os engenheiros de verificação, a metodologia VMM foi apontada como um obstáculo para os que gostariam de adotá-la, visto que classes, encapsulamento, herança, extensões entre outros aspectos da programação orientada a objetos, tornam o paradigma de programação totalmente diferente do utilizado tradicionalmente para a construção dos testbenches. A metodologia tem o suporte de quatro bibliotecas para 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 bÃ¡sicas 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 VMM permite que equipes de desenvolvimento de chip utilizarem SystemVerilog para criar ambientes de verificação detalhada usando nível de transação, a cobertura dirigida, técnicas de restrições aleatória, *Assertion* e especifica os blocos desenvolvidos da biblioteca de componentes para verificação interoperáveis. A arquitetura de tesbench proposta pela metodologia pode ser vista na figura abaixo:



3.15: Arquitetura de *testbench* da metodologia VMM[9]

Abaixo é descrito os componentes do testbench:

Checker - componente de verificação que verifica a validade de um protocolo. Checkers de baixo nível geralmente são implementados usando *Assertions*. Checkers pode ser combinados com os monitores.

Functional Coverage – componente responsável por medir o quão estimulado foi o ambiente com os casos de testes e condições reais como, por exemplo, *corner cases.*

Generator - componente responsável pela geração automática dos estímulos em nível de transação.

Monitor – componente reativo ou passivo que envia autonomamente relatórios de dados observados ou transações. Pode inclui uma funcionalidade de verificação verificador ou equivalente para o protocolo observado, mas não os dados ou transações transportados pelo protocolo.

Proprierties – componente desenvolvido para descrever o comportamento funcional utilizando Expressões Booleanas. Através dessas podem ser descritos eventos específicos, condições que devem ser sempre verdadeiras ou pontos de coberturas funcionais.

Testcase - a exigência do processo de verificação funcional. Normalmente, corresponde a uma característica interessante ou um *corner case* do projeto que deve ser verificado.

Driver – responsável por traduzir dados em nível de transação em sinais e enviar para o DUT. Existem dois tipos de *Drivers* existentes na metodologia, o proativo que controla a inicialização e o tipo de transação que será enviado e o reativo que não controla a inicialização e o tipo de transação, mas pode estar no controle de alguns aspectos do momento de sua execução, tais como a introdução de estados de espera. O proativo funciona da seguinte forma: Sempre que uma nova transação é fornecida pelas camadas superiores do ambiente de verificação para um driver pró-ativo, a operação é executada imediatamente na interface física. Por exemplo, um modelo de barramento funcional mestre para uma interface AMBA AHB é um driver proativo.

Já o tipo de *Driver* reativo a transação é iniciada pelo DUT, e o *Driver* reativo fornece os dados necessários para concluir com êxito a operação. Por exemplo, um modelo de barramento funcional de uma interface de memória programada é um *Driver* reativo: o DUT inicia o ciclo de leitura para buscar a nova instrução e o modelo de barramento funcional fornece o novo dado na forma da instrução codificada.

Self-Check – componente que especifica a forma correta de funcionamento do DUT. É semelhante ao *scoreboard* do OVM.

3.2.6 UVM (Universal Verification Methodology)

A metodologia UVM[56] foi desenvolvida pela Accellera Verification IP (VIP) e um grupo técnico denominado Technical Subcommittee (TSC). A metodologia está disponível para o público através de um manual de referência[57] (Reference Guide) acompanhado por uma biblioteca de classe *open-source* desenvolvidas em SystemVerilog e um guia de usuário[58] (User Guide). A norma UVM estabelece uma metodologia para melhorar a eficiência e o design da verificação, a portabilidade de dados e ferramenta de verificação e interoperabilidade VIP. O manual da metodologia UVM e o documento *Class Reference*, que detalha toda a biblioteca de classes utilizadas na metodologia, estão disponíveis sem nenhum custo no site da Accellera.

UVM coloca em prática um padrão único e aberto para o avanço da produtividade de verificação em equipes de design e em multi-empresas e verificação da concepção de esforços colaborativos. Além da referência de classe Manual, um Guia do Usuário e implementação open-source de referência estão disponíveis para acelerar a adoção na indústria.

UVM 1.0 disponibiliza recursos básicos e liberação para modificar a biblioteca de

funções da metodologia de verificação com código aberto OVM. Isto permitiu que a TSC Accellera VIP pudesse focar em adicionar características encontradas em outras metodologias comuns para satisfazer os requisitos funcionais acordados para encurtar o ciclo de desenvolvimento de padrões de verificação.

UVM 1.0 qualifica plenamente as características de base, corrige a maioria dos bugs conhecidos e implementa pedidos de melhorias. Principais novos recursos incluem um mecanismo de escalonamento, um registo de pacote (derivado de Verificação Metodologia Manual (VMM) tecnologia) e suporte para o Open SystemC Initiative (OSCI) Transaction Level Modeling-2.0 (TLM-2.0) padrão para modelo de conectividade operação de componentes e de comunicação . O Gerenciador de recursos é uma atualização para o mecanismo de configuração que a torna mais geral e inclui uma interface de linha de comando padronizado. As novas características incluem chamada, mensagem de captura e novas funcionalidades com a finalidade de gerenciar o ambiente de verificação com mais rigor.

A metodologia UVM conta com os seguintes benefícios:

- Biblioteca disponibilizada com código aberto, unificada com suporte a ambientes interoperáveis;
- Elimina a necessidade de interoperabilidade entre as várias bibliotecas de verificação;
- Baseado em uma biblioteca de classe utilizada em milhares de projetos (OVM);
- Fornece automação e capacidade de geração de testbench;
- Suporta reuso de módulos e de projeto em outros projetos;
- Incorpora o conhecimento da verificação coletiva dos membros da Accellera;
- Funciona em qualquer simulador de apoio ao padrão IEEE 1800;
- Permite multi-linguagem e plug-and-play VIP;
- Inclui um guia do usuário e a documentação de referência
- Integra-se com o fluxo de verificação dirigido a métricas.

Com o objetivo de utilizar como base a metodologia OVM e prover a junção de OVM com VMM, surge a possibilidade de URM se tornar a metodologia mais utilizada pelos engenheiro de verificação das principais empresas do mundo.

3.3 Análise Comparativa

Tabela 3.1: Análise comparativa das metodologias

	VeriSC	**BVM**	**OVM**	**IVM**	**VMM**
Fluxo de desenvolvimento	*Top Down*	*Top Down*	*Bottom up*	*Bottom Up*	*Bottom-up*
Ambiente de verificação executando Antes do DUV	Suporta	Suporta	Suporte indireto, necessitando criar um modelo funcional adicional.	Suporta	Suporte indireto, necessitando criar um modelo funcional adicional.
Geração de Estímulos	Centralizada	Centralizada	Descentralizada	Descentralizada	Descentralizada
Cobertura funcional	Biblioteca própria	Biblioteca de *SystemVerilog*	Biblioteca de *SystemVerilog*	Biblioteca própria	Biblioteca de *SystemVerilog*
Linguagem	*SystemC*	*SystemVerilog*	*SystemVerilog*	*SystemC*	*SystemVerilog*
Assertions	Não suporta	Suporta	Suporta	Não suporta	Suporta e Utiliza
Mecanismo de geração de *test-bench*	Sim	Sim	Não	Não	Não
Comunicação bidirecional	Não suporta	Não suporta	Suporta	Suporta	Suporta

A tabela 3.1 traz uma breve análise comparativa entre as metodologias de verificação funcional estudadas neste trabalho. A primeira linha da tabela descreve o processo de desenvolvimento da verificação e deixa explícito que as metodologias VeriSC, BVM utilizam

o fluxo de desenvolvimento *top-down*. Isso se deve ao fato de VeriSC e BVM serem obrigados a ter todo o modelo de referência do sistema construído inicialmente para seguir a sequência de passos para a construção do *testbench,* explicada nas duas metodologias. A metodologia IVM defende o fluxo de desenvolvimento do *testbench bottom-up,* já que desde o princípio do projeto do sistema é dividido em módulos, e cada módulo construído pode ser verificado à parte. As metodologias OVM e VMM utilizam o fluxo de desenvolvimento *bottom-up* devido aos seus princípios de criação de VCs (Verification Components).

Com relação ao ambiente de verificação ser construído antes do DUV, as metodologias VeriSC, BVM e IVM exibem passos de construção do *testbench* que suportam o desenvolvimento do ambiente sem a necessidade do DUV. A metodologia OVM, no seu conceito, explicita o desenvolvimento do ambiente de verificação já com o DUV inserido no ambiente, necessitando, assim, do DUV desde o início da construção do *testbench.* Por utilizar o conceito de *scoreboard,* a metodologia OVM só poderia suportar diretamente a construção do *testbench* antes do DUV se já existisse uma UVC na sua biblioteca OVC para cada interface do DUV, ou desenvolvendo um modelo de referência adicional.

A forma de comunicação, um dos fatores pesquisados nesse trabalho, observou que, em VeriSC, devido à utilização da biblioteca SCV (*SystemC* Verification), e em BVM, por ter sido baseado em VeriSC utilizando uma FIFO unidirecional com duas saídas, não existe suporte direto à comunicação bidirecional entre componentes. A metodologia BVM se diferencia da técnica VeriSC por utilizar a linguagem *SystemVerilog*, que dá suporte à interoperabilidade entre linguagens. Já IVM tem um suporte à comunicação dos seus componentes através de sua organização estrutural e utilização da biblioteca TLM de *SystemC*. A técnica OVM consegue prover suporte tanto à comunicação entre seus componentes quanto entre linguagens, pois esta é baseada na linguagem *SystemVerilog,* em que há um mecanismo de comunicação entre linguagens (*Direct Programming Interface*), entre componentes do UVC com o suporte de uma biblioteca construída especialmente para este fim.

Na comparação feita com relação à organização estrutural foi visto que as técnicas VeriSC e BVM são fortemente acopladas ao gerador de estímulos (*Source*) e ao verificador (*Checker*), pois todos os estímulos gerados são produzidos por único módulo, e as checagens também são feitas por apenas um componente. Já as metodologias OVM e IVM tratam os estímulos gerados em locais específicos, denominados *Virtual Senquence* (OVM) e *Testcase Model* (IVM), respectivamente. Dessa forma a organização de OVM e IVM torna-se mais

orientada a objeto e pode ser organizada em camadas para a geração de estímulos.

Tendo a cobertura funcional em foco, observa-se que para a metodologia VeriSC foi desenvolvida uma biblioteca de cobertura baseada em regra denominada *Brazil-IP Verification Extension* (BVE). As técnicas BVM e OVM utilizam uma biblioteca nativa de *SystemVerilog*. Prado[40] desenvolveu uma forma diferente de analisar a cobertura para a técnica IVM, baseada em espaços de valores e probabilidade, onde através de uma função calcula-se o percentual de cobertura.

Assertions são suportadas apenas pelas metodologias OVM e BVM, porém nenhuma dessas metodologias faz menção ao uso. Elas têm suporte devido ao *SystemVerilog* ter uma biblioteca para o suporte ao uso de *Assertions*.

As metodologias VeriSC, BVM e IVM têm seus ambientes de verificação baseados em *testbench*. Já a metodologia OVM tem sua estrutura baseada na criação de UVCs (*Universal Verification Components*). O mecanismo de geração semiautomática de *testbench* encontrado em VeriSC e BVM é a ferramenta eTBc; com esta ferramenta é possível gerar partes do ambiente de forma automática, e com isto obter um ganho de tempo, como informa Maia[35].

Conforme explanado no estado da arte, há trabalhos que criam todo o ambiente de verificação apenas posteriormente ao DUV, porém a maioria dos trabalhos estudados já estão seguindo a ideia inicial de VeriSC, que é construir o ambiente de verificação anteriormente à presença do DUV. Quando este for implementado, já deve existir o *testbench* completo para sua inserção, iniciando a fase de verificação funcional. Destarte, pode ser observado que não existe uma metodologia de verificação funcional que provê todas as características estudadas, de modo que o foco desse trabalho é desenvolver uma metodologia que provenha todas as características positivas de uma metodologia de verificação, garantindo ganho de tempo e diminuição de custo de desenvolvimento.

Capítulo 4 Metodologia OVM_tpi

Este capítulo descreve a metodologia proposta para o desenvolvimento de um ambiente de verificação funcional. O acrônimo OVM_tpi abrevia o nome da metodologia *Open Verification Methodology using Template Programming Interface.* As principais vantagens da técnica proposta é a possibilidade do desenvolvimento completo ser anterior ao desenvolvimento do DUV pela equipe de implementação, assim como a disponibilidade de ter *templates* padronizados, que permitem agilizar o processo de construção do ambiente de verificação.

Esta metodologia foi criada tendo como base um estudo anterior, que serviu como embasamento teórico, avaliando o estado da arte das técnicas existentes nessa linha de pesquisa. A técnica proposta tenta aglutinar os pontos positivos das metodologias apresentadas anteriormente, de forma a suportar a verificação funcional de diferentes tipos de projetos de hardware.

A disponibilidade de uma metodologia para o desenvolvimento de um ambiente de verificação funcional é importante devido à complexidade do desenvolvimento com um grande número de objetos especificados em diferentes níveis de abstração.

As seções a seguir detalham a arquitetura de *testbench* proposta, a biblioteca de *templates* desenvolvidos para auxiliar na criação dos objetos do *testbench*, o fluxo de construção do ambiente de verificação para um projeto *bottom-up* e, por último, a construção do mesmo projeto utilizando um fluxo de desenvolvimento *top-down*, mostrando, através de um exemplo concreto de hardware, como foi desenvolvido todos os passos para a construção do *testbench*. Como o ambiente pode ser criado anteriormente à descrição do hardware, as duas equipes do projeto (implementação e verificação) podem trabalhar de forma paralela e sem influência de um grupo no outro, diminuindo o risco de contaminação durante o desenvolvimento do mesmo. Para permitir que o *testbench* seja implementado antes, a metodologia inclui um mecanismo capaz de simular a presença do DUV usando os próprios elementos do *testbench*, sem a necessidade da geração de código adicional que não seja reutilizado depois.

A metodologia de verificação OVM_tpi pode ser aplicada para todos os tipo de circuitos digitais que a biblioteca OVM dê suporte. Um dado DUV pode conter diferentes

domínios de relógio, desde que haja uma divisão do relógio, de forma que cada circuito tenha apenas um *clock*.

As principais características da metodologia proposta são:

- Suporte ao fluxo de desenvolvimento do *testbench top-down e bottom-up*;
- Suporte a construção do ambiente de simulação através do reuso de todo o código construído;
- Permite a implementação do *testbench* em paralelo à implementação do DUV;
- Inclui um mecanismo para simular a presença do DUV unicamente com os elementos do *testbench*, sem usar nenhum código extra;
- Todas as partes do *testbench* podem estar prontas e simuladas antes do início do desenvolvimento do DUV;
- Suporte a comunicação bidirecional;
- Provê uma interface bem definida entre os módulos;
- Inclui uma interface para que qualquer DUV possa ser verificado, mesmo este não seguindo nenhum protocolo de comunicação;
- Suporte ao desenvolvimento de novos *templates* que utilizam protocolos diferentes de comunicação.

4.1 Fluxo de Desenvolvimento de um Projeto de Hardware

A figura 4.1 mostra o fluxo de projeto obtido através da iniciativa de paralelização das atividades de um projeto de hardware. O fluxo de desenvolvimento foi construído tendo como base o estado da arte e as principais definições existentes na área de verificação funcional. Foi escolhida uma abordagem de construção do ambiente de verificação antes do DUV, pois esta seria a mais adequada para a redução do tempo de projeto do sistema.

Esse fluxo de desenvolvimento de projeto foi construído partindo do pressuposto de que uma organização trabalha no desenvolvimento de hardware dividindo seus engenheiros de *front-end* (etapa de descrição do RTL, verificação, síntese e prototipação) em dois grupos distintos: implementação e verificação. Os objetos que estão desenhados e pintados com um tom mais claro de cinza representam o fluxo que a equipe de implementação irá direcionar seus esforços, sem nenhuma influência da equipe de verificação, que ficará responsável em seguir o fluxo pintado com um cinza mais escuro que o fluxo de implementação, porém mais claro que a etapa de Prototipação ou fluxo b*ack-end* (etapa de uso de ferramentas específicas

para desenvolver o *layout* do chip, para possibilitar a produção em silício).

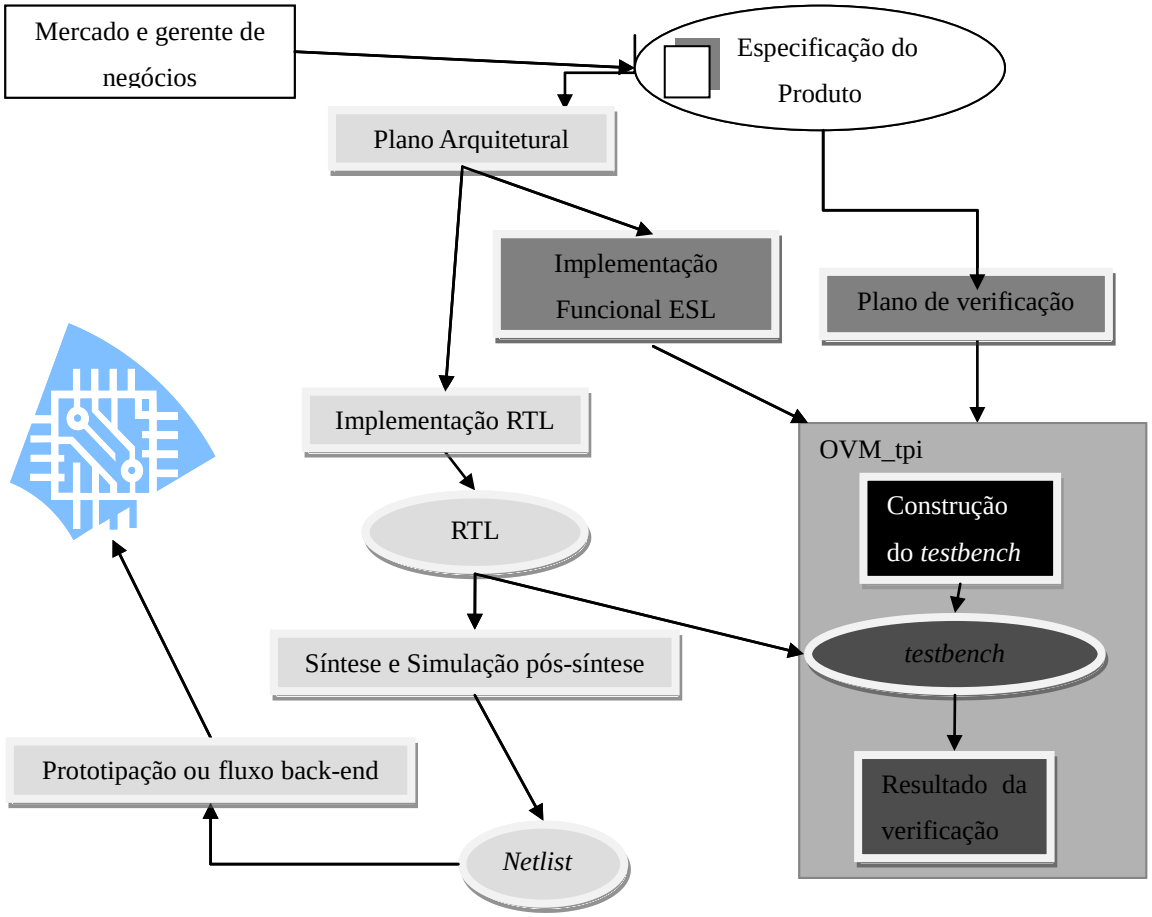


Figura 4.1: Fluxo de desenvolvimento de projeto de um hardware em paralelo

O fluxo de desenvolvimento de um projeto de circuito digital inicia com a análise de mercado, que visa encontrar possíveis nichos, dos quais possam surgir novos produtos. Após a idealização de um produto, o gerente de negócios entra em ação com a finalidade de assinar um contrato e iniciar o processo de interação **cliente/gerente de projeto** para o desenvolvimento da especificação do produto.

Tendo encerrado o processo de especificação do produto, começa um fluxo paralelo das duas equipes participantes do projeto: engenharia de hardware e engenharia de verificação. O fluxo desenvolvido pela equipe de engenharia de hardware tem início com a descrição do plano arquitetural, a partir da especificação. O plano arquitetural é o documento

crucial à descrição de alguns pontos detalhados, evitando dúvidas dos engenheiros de implementação quanto ao produto que deve ser implementado. Alguns pontos que devem fazer parte do plano arquitetural são: visão geral do bloco, diagrama de blocos, interface de sinais, formas de ondas temporizadas, diagrama da máquina de estados do bloco, descrição da maquina de estado, registradores de controle do bloco (caso existam), possíveis caminhos críticos, informações de síntese, tecnologia do processo que será utilizado, máxima freqüência, energia e geometria (máxima área) utilizada para a síntese.

Depois de concluído o plano arquitetural, a equipe de engenharia de hardware descreve o circuito no nível de RTL (*Register Transfer Level*). Feita a descrição, é desenvolvida a etapa de síntese, atividade responsável por transformar a descrição RTL em um arquivo *netlist*, onde todo o circuito será descrito em portas lógicas. Com o *netlist* gerado, inicia-se o processo de construção física do circuito com a prototipação e/ou fluxo *back-end*, até chegar ao chip final.

A equipe de verificação inicia o trabalho desenvolvendo o plano de verificação. Posteriormente, desenvolve a implementação do circuito no mais alto nível de abstração para que este sirva de modelo de referência do circuito digital que está sendo implementado. Com o modelo de referência pronto, começa o processo de desenvolvimento do *testbench*; é nessa etapa que a metodologia OVM_tpi será utilizada para o desenvolvimento do ambiente de verificação funcional, seguindo uma arquitetura de *testbench* e um fluxo de atividade pré-definida.

4.2 Arquitetura de *Testbench* Proposta

A arquitetura de *testbench* suportada pela metodologia OVM_tpi foi concebida tendo em mente as virtudes das metodologias citadas anteriormente. O ambiente de simulação em que o DUV é inserido para realizar a verificação foi dividido em objetos. Nesse ambiente o DUV é estimulado e suas saídas são comparadas com as saídas de um modelo ideal que aqui será denominado “modelo de referência”.

Dessa forma, foi concebida uma arquitetura para o *testbench* com as interfaces bem definidas entre os diversos componentes, assim como também uma interface com o DUV, que será a responsável por todas as ligações de sinais que ocorre no ambiente de verificação. A figura 4.2 mostra a arquitetura de *testbench* proposta pela metodologia OVM_tpi.

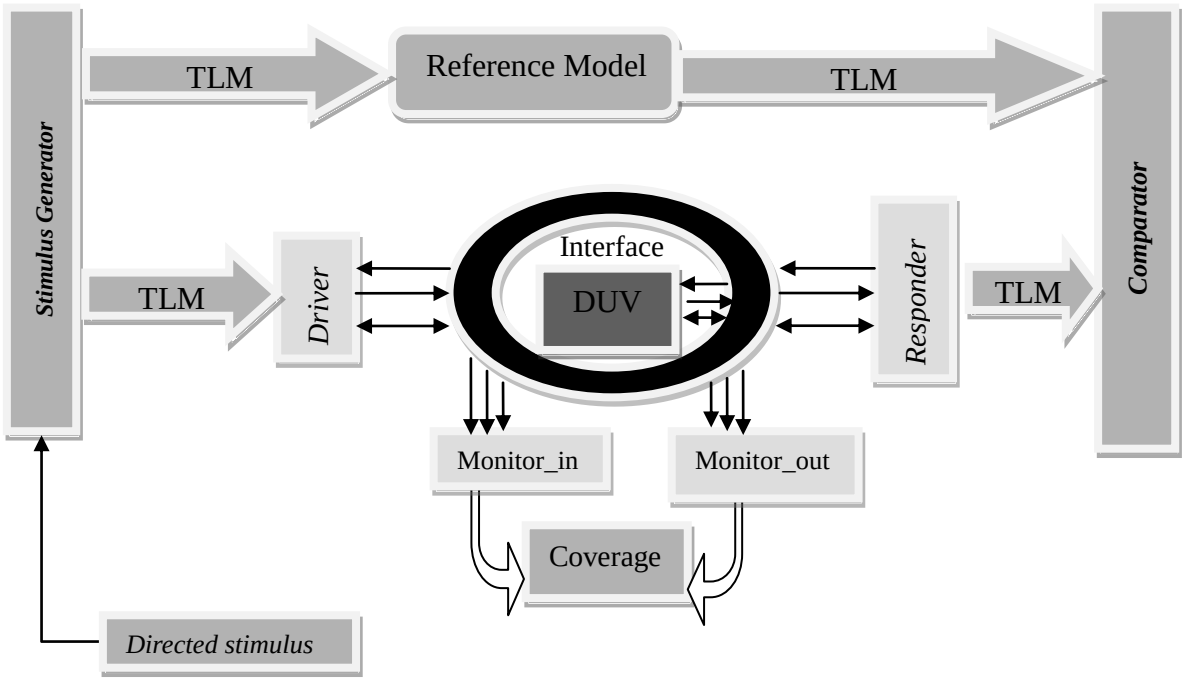


Figura 4.2: Arquitetura do *testbench* da metodologia OVM tpi

A seguir, temos uma descrição sucinta dos componentes:

***Directed Stimulus* –** o *Directed Stimulus* é um arquivo utilizado para aplicação de estímulos diretos no processo de verificação. Podemos gerar, a partir dele, estímulos direcionados para o *Stimulus Generator,* e deste enviar para o *testbench.*

***Coverage* –** agrupa os dados provenientes dos monitores para fazer a cobertura, verificando se tudo está ocorrendo conforme o esperado. O *Coverage* também pode enviar ao controlador as informações de cobertura, que servirão para o ambiente de verificação refinar mais os conjuntos de dados da cobertura através de um mecanismo de controle automático. Dessa forma, o código gerado pelo eTBc não será completo, e o engenheiro de verificação terá que agregar mais informações. O *Coverage* busca a cobertura ideal de cada DUV e tenta chegar ao conjunto máximo das suas possíveis entradas e saídas.

O C*overage* foca nos seguinte tipos de informações:

- Tráfego básico de dados para cada interface;
- Efeito combinado de tráfego em toda a interface;
- Estímulos gerados pelo *Stimulus Generator*;
- Estímulos recebidos pelo *Comparator*.

***Reference Model* –** esse objeto tem a responsabilidade de modelar idealmente o comportamento do circuito em desenvolvimento, usando uma linguagem de mais alto nível de abstração, como *SystemVerilog* ou *SystemC*. É uma implementação executável e, por definição, deve refletir a especificação de forma fiel. Normalmente, não é necessário que possua detalhes temporais do comportamento. O modelo de referência difere do *scoreboard* de OVM por ser apenas a implementação do modelo em alto nível, sem fazer nenhum tipo de checagem dos dados. Assim, pode-se notar, detalhadamente, o desacoplamento dos objetos e seus objetivos específicos. As funcionalidades do modelo de referência são estruturadas de forma hierárquica e bem especificada, suportando uma possível separação ou inserção de funcionalidades neste modelo durante a construção do ambiente completo de verificação, e, assim, dando suporte a construção de um *testbench* tanto *top-down* quanto *bottom up*.

Um dos fatores importantes na arquitetura proposta para a técnica OVM_tpi é a necessidade do engenheiro de sistema ter o foco apenas na funcionalidade do módulo, sem se preocupar com conexões existentes do *testbench.* Com isso, pode-se observar que a metodologia OVM_tpi também suporta a separação das atividades das equipes de sistema, *design* e verificação.

O modelo de referência é desenvolvido pela equipe de desenvolvimento de sistemas e repassado para a equipe de verificação, que instanciará apenas a função a ser verificada.

***Stimulus Generator* –** o objetivo do módulo *stimulus generator* é criar as transações que serão exercitadas no ambiente de simulação, tanto pelo DUV quando pelo Modelo de Referência. A geração de estímulos pode ser randômica, direta, ou direta e randômica. OVM_tpi inclui um *template* que suporta qualquer uma das três formas de estímulos do ambiente. Para utilizar estímulos diretos, é necessário que o engenheiro de verificação utilize o arquivo em branco criado pelo eTBc, cujo nome é “direct_**nome do módulo verificado**.stim”, e descreva exatamente o formato das transações que serão geradas pelo *Stimulus Generator,* de modo que este possa utilizar o arquivo para estimular o *testbench*. A biblioteca OVM_tpi suporta modularização de estímulos para criar transações de diferente complexidade, que podem ser utilizadas pelo *Stimulus Generator* para geração de transações.

***Comparator* –** a responsabilidade desse objeto é fazer a comparação das respostas recebidas do modelo de referência e do DUV, reportando mensagens de erro caso haja divergência entre as respostas. Para cada estímulo gerado pelo *Stimulus Generator* é feita a comparação das saídas do modelo de referência e do DUV (*design under verification*). Além da comparação

dos valores, é utilizado *assertion* para realizar a comparação dos tipos de dados do DUV e do modelo de referência. O módulo *Comparator* deve prover as seguintes funcionalidades:

- Suportar checagens ordenadas dos valores e tipos esperados;
- Fazer contagem do número de erros encontrados;
- Encerrar a simulação quando esta acusar a quantidade de erros especificada pelo engenheiro de verificação;
- Suporte à saída diretamente no terminal caso aconteça algum erro, ou quando o engenheiro de verificação quiser observar qualquer saída vistoriada.

Interface – a *Interface* é um módulo criado para deixar o ambiente de verificação mais desacoplado. Serve como um barramento, através do qual ocorrem as ligações em nível de sinais. Todos os módulos que utilizam sinais se ligam à Interface. OVM_tpi inclui um módulo em que é criada uma Interface para cada DUV existente no projeto, e um barramento para cada módulo do ambiente de simulação.

Driver **–** o *Driver* tem como objetivo transformar as transações recebidas do *Stimulus Generator* de transações em sinais, e gerenciar de forma organizada o envio desses sinais para a Interface. Esse gerenciamento é feito através de protocolos de *handshakes,* que controlam o envio de um dado e o posterior aguardo de uma resposta, para poder enviar um novo dado. Cada circuito pode conter um ou mais *Drivers*. Para cada interface do DUV deve ser implementado um *Driver*, pois cada *Driver* será responsável pela comunicação de uma determinada interface de entrada do DUV com o *Stimulus Generator*. A metodologia OVM_tpi inclui uma máquina de estados finitos (FSM, *Finite State Machine*) que implementa a comunicação com a interface do DUV. Apenas o *Driver* tem a responsabilidade de gerenciar o protocolo de comunicação, ficando os outros módulos do sistema sensíveis a essas mudanças. A implementação desse "tradutor" tem um alto nível de complexidade e necessita de muito cuidado na construção, para que o sincronismo com os outros módulos possa dar-se corretamente.

Responder **-** o *responder* é um componente do OVM_tpi com a responsabilidade inversa do *Driver*. Este módulo recebe informações em nível de sinal e as transforma em nível de transação. Posteriormente, a transação é enviada para o *Comparator*. O *testbench* inclui um *Responder* para cada interface de saída do DUV, e tem uma ligação, através da Interface, com o *Driver* por meio do protocolo de *handshake*, podendo sincronizar cada nova resposta que será recebida do DUV para transformá-la em transação. A metodologia *OVM*_tpi inclui uma

FSM que é sensível a mudança de sinal gerenciada pelo *Driver,* e executa sua função apenas no estado correto da FSM.

Monitor – sua responsabilidade consiste em monitorar o barramento de comunicação do *testbench* com o DUV, tanto de entrada quanto de saída. Além disso, é responsável pela monitoração e cobertura de controle com o uso de *Assertions*, transformando os sinais capturados em transações para que estas sejam enviadas ao *Coverage*. Os *Monitors* funcionam de forma passiva e não causam nenhum tipo de influência no funcionamento do ambiente de simulação, apenas gerando informação importante sobre o funcionamento do *testbench.* O *Monitor* transforma os sinais coletados em transação, seguindo o mesmo protocolo do *Driver* e do *Responder*.

4.2.1 Suporte a Circuitos com Comunicação Bidirecional

Para o caso de circuitos que tenham portas bidirecionais, existe a necessidade que o ambiente suporte a comunicação bidirecional do DUV com o *testbench*. Na metodologia OVM_tpi a comunicação bidirecional é suportada mediante a junção do *Stimulus Generator* com o *Comparator,* e do *Driver* com o *Responder,* gerando os módulos Stim_Comp e *Driver_Responder.* A figura 4.3 mostra a arquitetura de *testbench* que suporta DUV e que possui comunicação bidirecional:

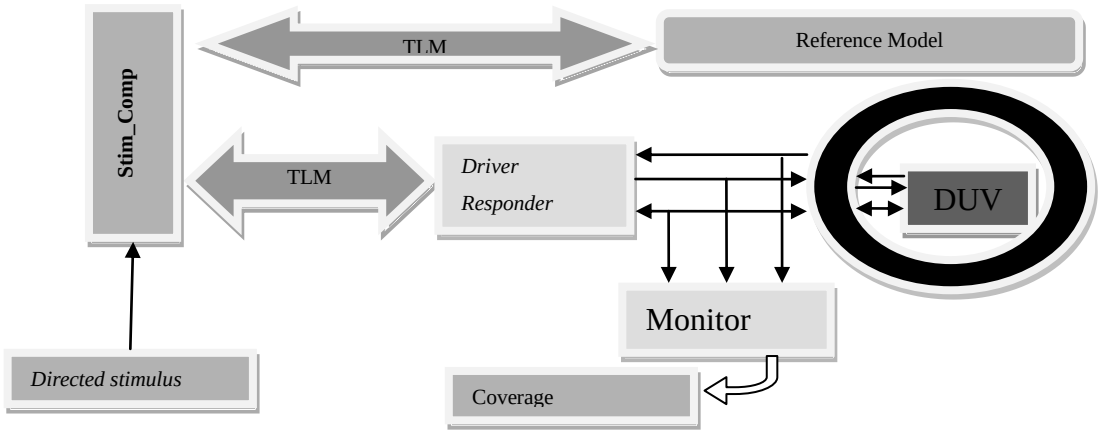


Figura 4.3: Arquitetura do *testbench* com comunicação bidirecional

***Stim Comp* –** É responsável pela geração e checagem do recebimento das respostas geradas pelo DUV. Gera estímulos randômicos ou direcionados, de acordo com a descrição das transações. Nas transações, é obrigatória a especificação da ação que será executada pelo *Stim Comp,* pois este precisa identificar qual ação deverá ser executada. Embora seja possível tanto o envio quanto a requisição de dados, deve-se observar que uma dessas ações exclui a outra:

assim, caso a opção seja de enviar um dado para o DUV, o envio é feito, sendo obstada a requisição de algum dado no mesmo ciclo. Porém, se o objetivo for ler um dado que está guardado no DUV, então é feita uma requisição, e somente após ser recebida a resposta desta é que uma nova requisição poderá ser enviada. A resposta recebida do DUV será comparada com a resposta enviada pelo modelo de referência e caso haja divergência entre as respostas, é enviada uma mensagem informando o erro e a simulação é obstada. O engenheiro de verificação tem a possibilidade de mudar a quantidade de erros que deve existir para finalizar a simulação, através de uma função existente no módulo. Isso pode auxiliar o engenheiro de verificação a observar o comportamento do testbench durante um período onde ocorreram erros.

Driver_Responder- Tem o mesmo objetivo do *driver* e do *responder* de um ambiente unidirecional. Consegue tanto transformar uma transação em sinais, como receber uma resposta em sinais e transformá-la em nível de transação. Utiliza uma FSM para gerenciar o protocolo *handshake,* onde no estado especificado os sinais são convertidos em transações, ou vice-versa. O *Driver_Responder* se comunica diretamente com o componente *Interface,* sendo o único responsável pela mudança dos sinais de protocolo.

A seguir serão descritos os fluxos propostos pela metodologia para a construção de *testbench.*

4.3 Fluxo para o Desenvolvimento do T*estbench* da metodologia OVM_tpi

A metodologia OVM_tpi propõe dois fluxos de desenvolvimento do *testbench,* dependendo apenas do fluxo de desenvolvimento do *hardware* que foi escolhido pela gerência de projeto. Os fluxos propostos suportam as metodologias *top-down* e/ou *bottom-up*. Para ambas as metodologias de projeto, o fluxo proposto na metodologia OVM_tpi tem como objetivo principal o desenvolvimento de um *testbench* que realize a verificação funcional.

O desenvolvimento do *testbench* poderá ser realizado da forma semiautomática, a partir do plano de verificação, de uma descrição TLN do sistema em desenvolvimento e de uma biblioteca de *template*, desenvolvida no contexto deste trabalho.

Para auxiliar o entendimento dos dois fluxos de desenvolvimento do *testbench*, foram escolhidos dois exemplos de circuitos digitais, um DPCM (*Differential Pulse-Code Modulation*), para seguir todas as atividades de construção do *testbench* da metodologia

OVM_tpi, utilizando uma comunicação unidirecional, e uma memória para mostrar o modo de comunicação bidirecional.

O DPCM é um codificador de sinal que utiliza a base de PCM (*Pulse-Code Modulation*). O objetivo principal desse circuito é fazer a modulação do valor recebido como entrada, procedendo da seguinte forma: recebe os valores de duas amostras consecutivas, calcula o diferencial entre elas, e satura o resultado do diferencial num intervalo especificado. A saída do circuito digital é o valor modulado.

A arquitetura do módulo DPCM é mostrada da figura 4.4:

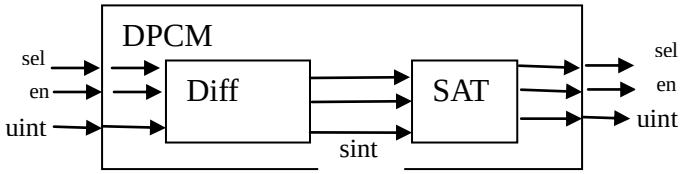


Figura 4.4: Arquitetura DPCM

A arquitetura do DPCM é composta de dois submódulos, sendo o módulo **Diff** responsável por executar o diferencial entre os valores recebidos, enquanto o submódulo **SAT** tem a funcionalidade de receber o diferencial e fazer a saturação apenas para valor inteiro positivo.

4.3.1.1 Suportando o Desenvolvimento do *Testbench* em Projetos T*op-Down*

O fluxo de desenvolvimento *top-down* tem algumas características já citadas na seção 2.5.1, importantes para o desenvolvimento do projeto e utilizados pela metodologia VeriSC e BVM. Quando se utiliza a metodologia *top-down*, o sistema é todo desenvolvido a nível ESL (*Electronic System Level*), e posteriormente dividido em objetos. Tais objetos, já testados e integrados, serão decompostos em objetos com o mínimo de funcionalidade possível em cada estrutura, e cada um servirá como o modelo de referência para os módulos RTLs que serão desenvolvidos pela equipe de implementação.

A metodologia OVM_tpi evolui uma sequência de passos baseada no estado da arte para a criação do *testbench* com uma abordagem de verificação do tipo Caixa Preta. Nesta abordagem, só interessam as entradas, saídas e função do DUV. É preciso entender a função especificada para gerar as entradas corretas e checar as saídas que foram obtidas. Os fluxos serão descritos nas seções seguintes. A figura 4.6 mostra o fluxo de desenvolvimento *top-down* do ambiente de verificação.

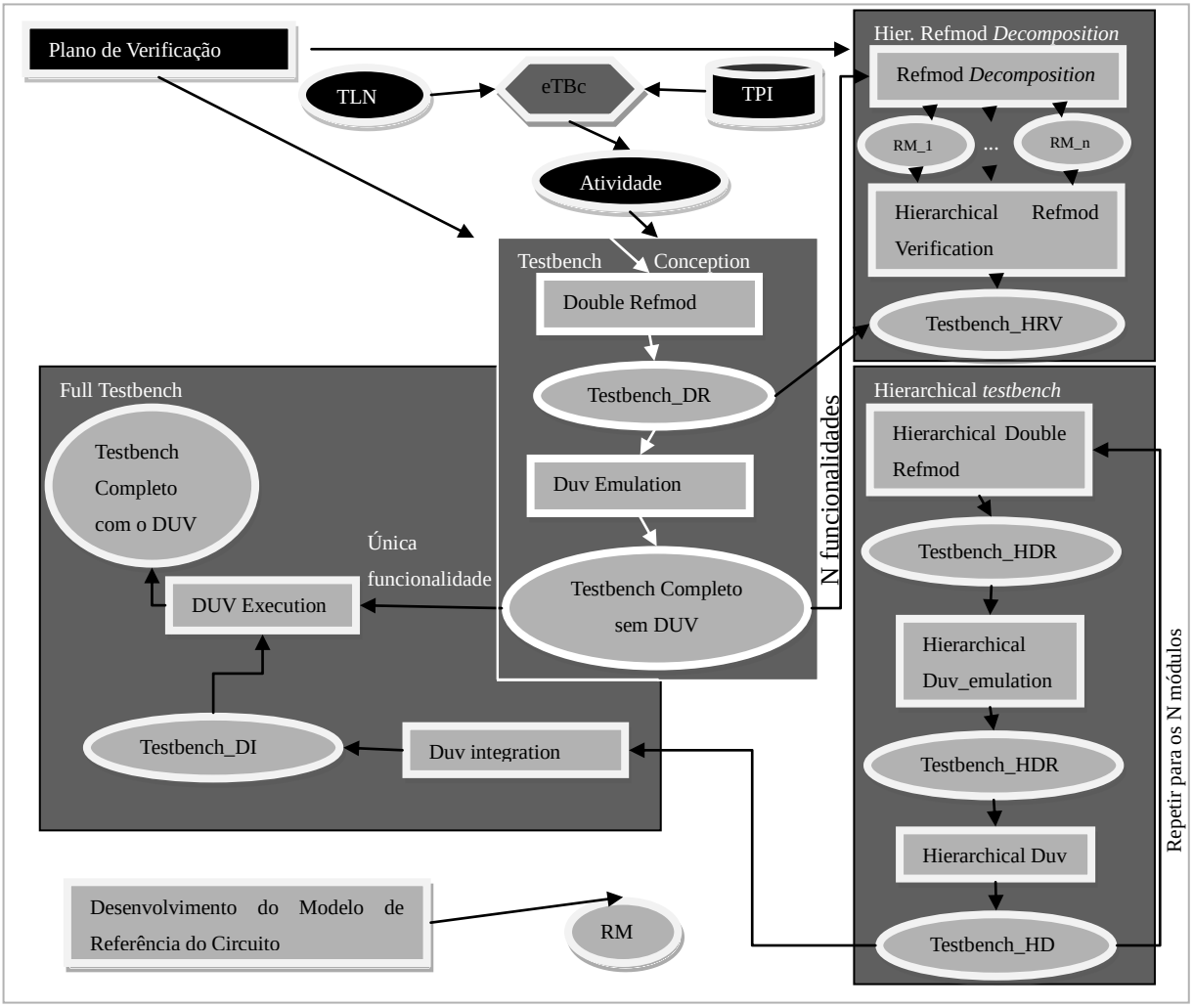


Figura 4.5: Fluxo de desenvolvimento do *testbench*

A figura 4.6 mostra que o início do fluxo de desenvolvimento se dá com a descrição do plano de verificação, da TLN, e com a biblioteca de moldes TPI. A ferramenta eTBC irá participar de todo o fluxo, gerando de forma semiautomática os componentes de cada etapa que não forem reusáveis, e o plano de verificação dará suporte ao desenvolvimento do modelo de referência do circuito. Depois de desenvolvido o modelo de referência, terão início as etapas de implementação e validação dos componentes do ambiente de verificação.

O fluxo básico de desenvolvimento de testbench para a metodologia OVM_tpi conta com três atividades que usam *self-checking*. A primeira atividade do processo é chamada de *Double*_Refmod, e busca fazer a validação do modelo de referência, do gerador de estímulos e do comparador. Todos os componentes são implementados em nível de transação. Após a validação desses componentes é iniciada outra atividade, qual seja, o processo de validação dos componentes que trabalham a nível de sinal. É importante ressaltar que nesse processo de validação não será utilizado o DUV, pois este está sendo desenvolvido em paralelo ao

ambiente de verificação. Para suprir a necessidade do DUV é desenvolvida uma emulação desse módulo através dos componentes que farão parte do *testbench*. A etapa responsável pela validação do ambiente de verificação com o DUV emulado denomina-se de *Duv_Emulation*, que é de suma importância, pois após ela todo o ambiente estará validade e pronto para que o DUV seja inserido e verificado. A terceira atividade chama-se *Duv_Execution* e tem o objetivo de verificar o DUV desenvolvido pela equipe de implementação. Após a finalização dessa etapa, a verificação do módulo estará completa e o processo de verificação validado. Na figura 4.6 podemos observar o fluxo genérico de desenvolvimento de um *testbench* utilizando a metodologia OVM_tpi.

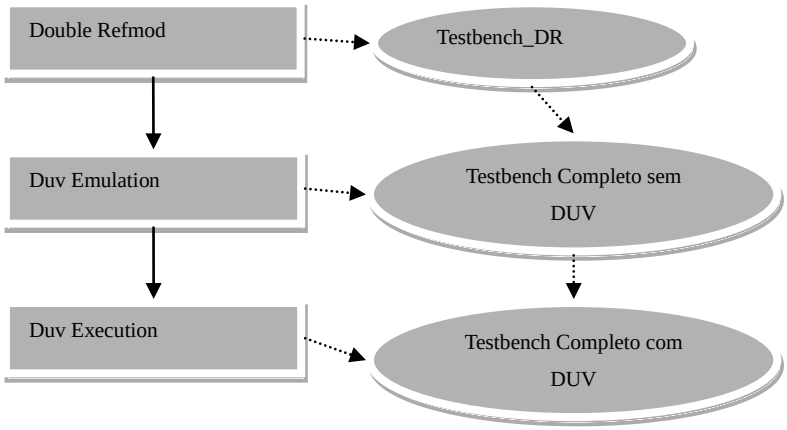


Figura 4.6: Fluxo de desenvolvimento de *testbench* utilizando a metodologia OVM_tpi

No processo de desenvolvimento do *testbench* todos os componentes validados em uma atividade serão reusados nas atividades seguintes.

4.3.1.2 *Testbench* Conception

No fluxo de desenvolvimento *top-down*, a atividade de *Testbench Conception* dá início ao desenvolvimento do fluxo de construção do *testbench.* Esse passo objetiva, principalmente, a criação dos geradores de estímulos e dos comparadores para o circuito completo, já descrito no nível ESL. Esse processo é importante, uma vez que possibilita verificar se o sistema, como um todo, está funcionando de acordo com a especificação, assim como permite validá-lo para que possa ser utilizado como modelo de referência do projeto do RTL.

A atividade de *Testbench Conception,* por sua vez, é subdividida em duas outras atividades: a atividade de *Double*_Refmod e a atividade de Duv_*Emulation*.

4.3.1.2.1 Double *Refmod*

Nesta etapa serão desenvolvidos e validados o gerador de estímulos e o comparador, garantindo que o ambiente possua capacidades mínimas de funcionamento, com comunicação e detecção de erros. O *Directed stimulus* será desenvolvido apenas se o plano de verificação contiver testes diretos e *corner cases*, que serão descritos manualmente no arquivo. Nos exemplos utilizados nesse trabalho não foram utilizados tais casos de teste e, portanto, não foi implementado o *Directed stimulus.*

Nos objetos desenvolvidos para a biblioteca de *templates* da metodologia OVM_tpi, até então, foram inseridos mecanismos de parada no componente responsável pela comparação e também no gerador e receptor (Stim_Comp), no caso da verificação funcional de ambiente bidirecional. A estrutura obtida após a atividade do *Double*_Refmod é mostrada nas figuras 4.7 e 4.8.

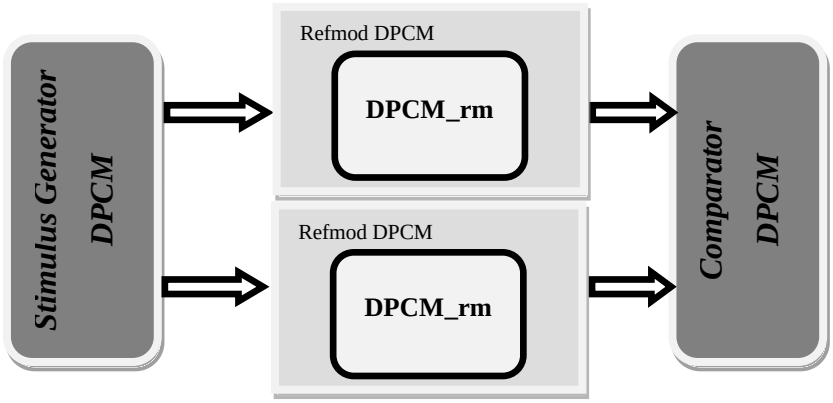


Figura 4.7: Testbench resultante da atividade 1.1 unidirecional (*Double* Refmod)

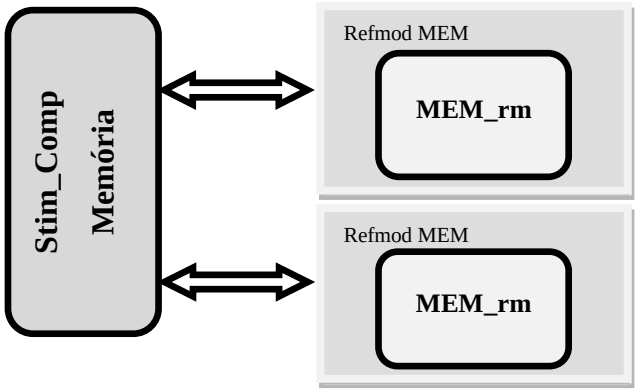


Figura 4.8: Estrutura da atividade 1.1 bidirecional (*Double* Refmod)

Além disso, é gerada automaticamente uma interface do modelo de referência, onde será necessário apenas escrever a chamada das funções que farão parte do modelo de

referência. A figura 4.9 mostra o código gerado pelo eTBc, com um comentário indicando o local do código para instanciar a chamada de função do modelo de referência.

```
tr_out_pkt_out_crc16= new();

//-------------------------------------------------------------------------
// Here goes the code that executes the reference model's functionality.
//-------------------------------------------------------------------------
tr_out_pkt_out_crc16.set_t_crc(crc16_i.Funcao do refmod);
```

Figura 4.9: Parte do modelo de referência em que será inserida a função a ser verificada

4.3.1.2.2 *Duv_Emulation*

O passo do Duv_*Emulation* tem o objetivo de implementar e validar a conversão das transações em sinais no *driver*, para cada transação gerada pelo Gerador de Estímulos, e transformar sinais em transações nos *drivers*. Neste passo são desenvolvidos os *drivers*, *responders* e os monitores.

Os *drivers* são responsáveis por transformar as informações em sinais, utilizando um protocolo de comunicação do *testbench* com o DUV; já os *responders* fazem o inverso, transformando sinais em transações, seguindo o mesmo protocolo de comunicação utilizado no *driver.* Os monitores são responsáveis por gerenciar os sinais de entrada e saída do DUV, verificar o protocolo de controle da comunicação dos módulos com a interface através de *assertions,* e enviar os dados obtidos para o módulo *Coverage*.

O processo de validação dos componentes *driver, responder e monitor* é relevante porque o DUV só reconhece estímulos descritos como sinais e o source gera estímulos no formato de transações. Com a validação, o ambiente de verificação poderá se comunicar com o DUV sem possibilidades de erros provenientes do *testbench.*

Nesse passo também são criados os objetos *Coverage e o Interface.* O componente *Coverage* será responsável pela cobertura de dados do circuito digital. Todos dos sinais de dados, tanto de entrada quanto de saída do DUV serão cobertos nesse componente. O componente *Interface* tem o objetivo de encapsular os sinais, para facilitar seu gerenciamento, além de sincronizar os componentes *Driver* e *Responder*. Funciona como uma espécie de barramento onde todos os componentes do *testbench* que trabalham no nível de sinais se ligam.

As figuras 4.10 e 4.11 mostram os *testbenchs* obtidos após a etapa de Duv_*Emulation*:

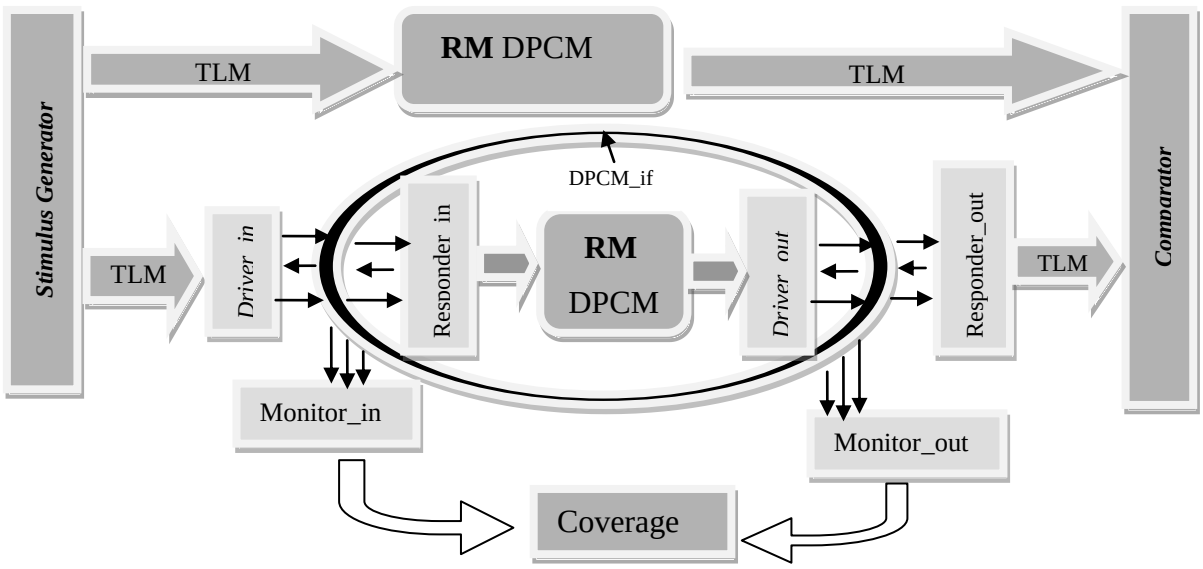


Figura 4.10: Estrutura da atividade 1.2 Unidirecional (Duv_Emulation)

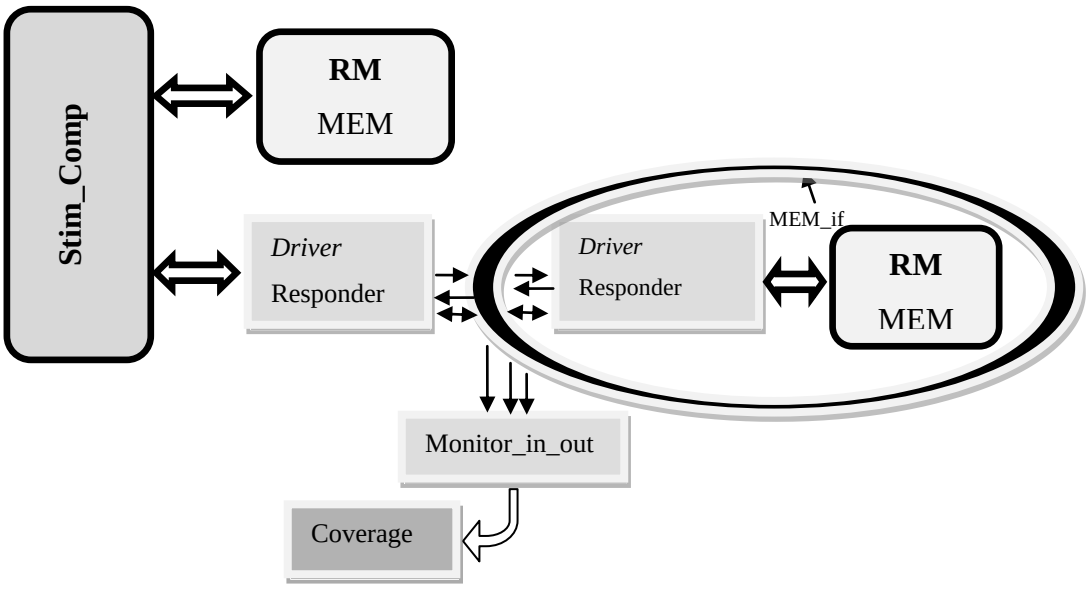


Figura 4.11: Arquitetura da atividade 1.2 Bidirecional (Duv_Emulation)

As figuras 4.10 e 4.11 mostram as estruturas que deverão ser construídas na atividade Duv_*Emulation*, para um DUV unidirecional e para um bidirecional, respectivamente. Na figura 4.10 foi desenvolvido todo o *testbench* do DPCM, com seus objetos validados. No *Double* Refmod foi validado o gerador de estímulos e o comparador, e neste momento serão validados os módulos que se comunicam com a interface, assim como o protocolo de comunicação utilizado pelo *testbench*. Note que na figura 4.10 é feita a emulação do DUV

através do uso de um Responder, que é criado para a comunicação entre a interface emulada e o Driver da interface de entrada. Já na interface de saída, é utilizado um *driver_emu* para traduzir a transação de resposta do Modelo de Referência em nível de sinal, emulando a outra interface do DUV. Os *Monitors* irão ler todos os sinais da interface que são responsáveis por monitorar e fazer a avaliação do trabalho dos protocolos, enviando os dados adquiridos para o *Coverage* checar a cobertura.

Na figura 4.11 é utilizado um "***Driver_Responder***" que tem o objetivo de transformar os sinais recebidos novamente em transação, e posteriormente enviá-la para o modelo de referência. O monitor irá checar os sinais do protocolo de comunicação e enviar os dados para o *Coverage* quando alguma resposta for encaminhada de volta para o **Stim_Comp**, módulo responsável tanto pela geração de estímulo quanto pela comparação dos resultados que recebe como resposta.

Os círculos mais escuros das figuras 4.10 e 4.11 representam o componente denominado *Interface*, que tem o objetivo de gerenciar a sincronização do ambiente de verificação. Os componentes interiores à *Interface* implementam uma emulação do DUV. Essa emulação é uma das características da metodologia VeriSC, que suporta o desenvolvimento do ambiente de simulação antes do desenvolvimento do DUV e foi introduzida nesta metodologia, permitindo validar o *testbench* completo antes de sua integração.

4.3.1.3 *Hierarchical* Refmod *Decomposition*

O objetivo da atividade de *Hierarchical* Refmod *Decomposition* é dividir a descrição em um nível mais alto de abstração, servindo de modelo de referência, por meio de submódulos. Essa divisão é feita a partir do modelo de referência já testado até sua emulação, dividindo-o em blocos de funcionalidade mínima. Assim, todas as funcionalidades do sistema serão organizadas em blocos. Após a divisão, é feita a verificação funcional para cada submódulo do modelo de referência de acordo com os passos abaixo. A decomposição é feita conforme a funcionalidade contida na especificação.

4.3.1.3.1 *Refmod Decomposition*

O passo Refmod *Decomposition* é responsável pela divisão do modelo de referência em módulos que implementam um subconjunto das funcionalidades do circuito. A figura 4.12 mostra a estrutura

do Refmod, obtida após esta atividade, que é feita de forma manual pelos projetistas:

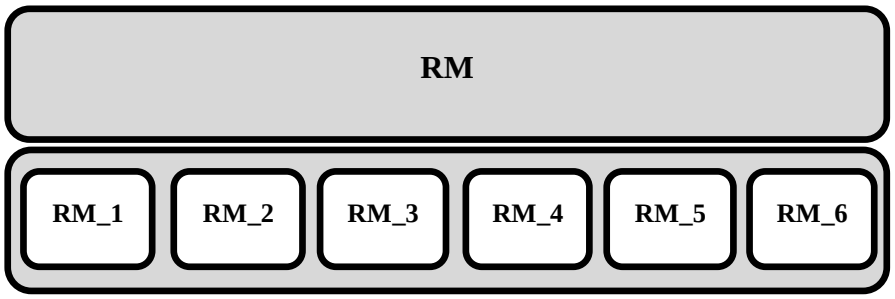


Figura 4.12: Estrutura da atividade 2.1 (Refmod *Decomposition*)

A figura 4.13 mostra a decomposição hierárquica do DPCM:

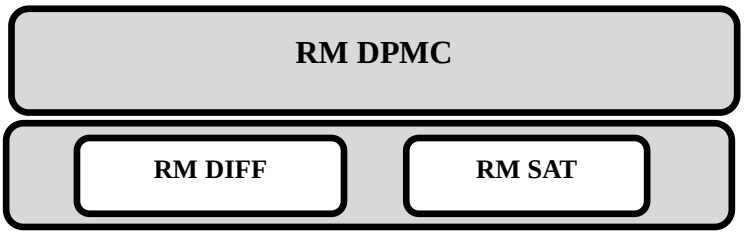


Figura 4.13 - Decomposição Hierarquica do DPCM

4.3.1.3.2 *Hierarchical Refmod* Verification

Após a divisão do modelo de referência em blocos, a atividade *Hierarchical* Refmod *Verification* é iniciada. Reusa todo o ambiente criado no *Double* Refmod, apenas trocando uma das instâncias do modelo de referência pelo modelo de referência hierárquico obtido com a divisão em submódulos.

Posteriormente, verifica-se se as respostas são as mesmas, de forma a validar o processo de decomposição do modelo de referência. O *testbench* obtido após esta atividade pode ser visto na figura 4.14. No caso do exemplo DPCM, o *testbench* após esta atividade pode ser visto na figura 4.15.

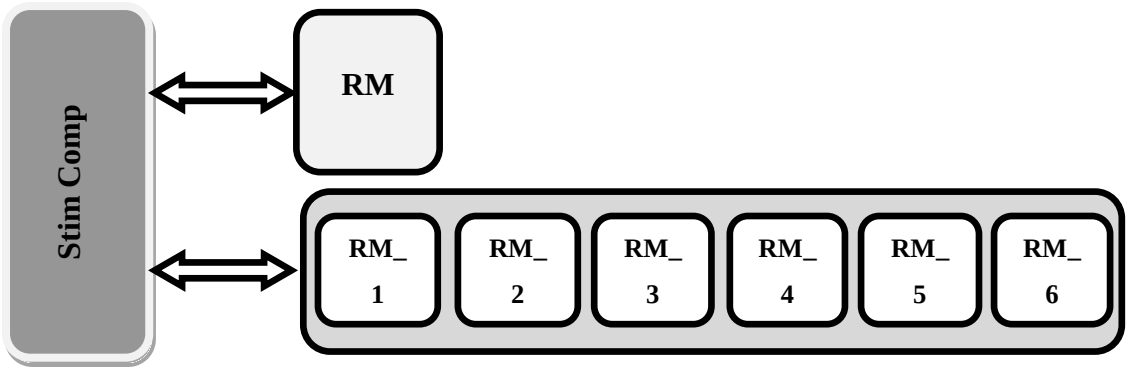


Figura 4.14: Estrutura da atividade 2.2 (*Hierarchical* Refmod Verification)

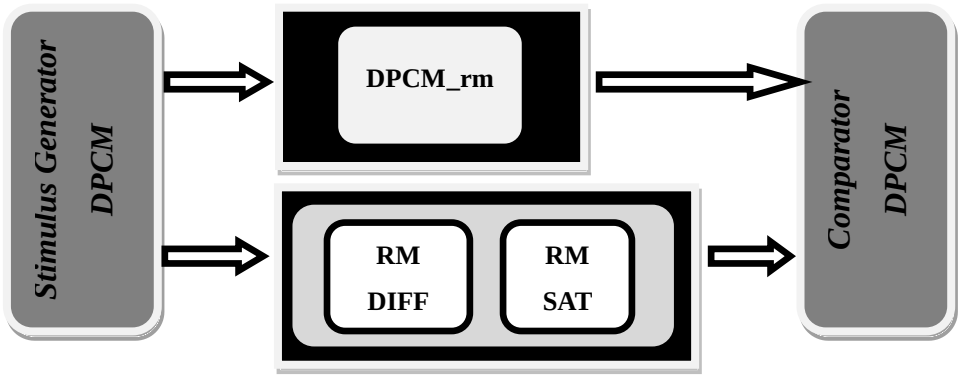


Figura 4.15: Estrutura da atividade 2.2 (*Hierarchical* Refmod *Verification*)

4.3.1.4 *Hierarchical Testbench*

A etapa *Hierarchical Testbench* é responsável pela validação de cada módulo que compõe o modelo de referência, e é repetida tantas vezes quantos forem os submódulos do modelo de referência. Isso ocorre porque o *Hierarchical Testbench* será feito para cada submódulo, a fim de verificar cada funcionalidade descrita no DUV. Esse passo é dividido em três subpassos, a seguir descritos.

4.3.1.4.1 *Hierarchical* Double *Refmod*

Esta etapa tem como objetivos a geração de estímulos para cada módulo do modelo de referência e a validação das respostas obtidas. O processo ora relatado na figura 4.16 é semelhante ao *Double*-Refmod, todavia, é feito para cada submódulo do Refmod.

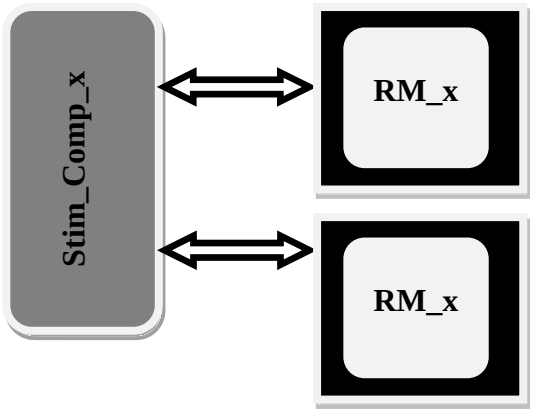


Figura 4.16: Estrutura do passo 3.1 (*Hierarchical Double* Refmod)

O modelo de referência denominado RM_x simboliza um sub-bloco do modelo de referência que será verificado. Consequentemente, pode ser necessário implementar um componente **Stim_Comp_x** responsável pela geração de estímulos e comparação dos resultados no ambiente de verificação, porque as entradas ou saídas do modelo de referência podem ser diferentes.

Como exemplo dessa necessidade, temos o módulo **Stimulus Generator DPCM**, implementado na atividade Double Refmod, que gera estímulos do tipo inteiro positivo e recebe no **Comparator** estímulos inteiros positivos. Já no ambiente de verificação, para o primeiro módulo do DPCM, o DIFF, o **Comparator** pode receber como resposta tanto inteiros positivos quanto inteiros negativos. Porém, isso não requer nenhum esforço manual do engenheiro de verificação, visto que o processo de automação da metodologia OVM_tpi constrói esse módulo de forma automática. A figura 4.17 mostra o ambiente de simulação criado na atividade *Hierarchical Double* Refmod para o módulo DIFF. Um ambiente semelhante a este deve ser criado para todos os submódulos do projeto, que no exemplo proposto serão o DIFF e o SAT.

Figura 4.17: Estrutura do *Hierarchical Double* Refmod para o módulo DIFF

O processo de desenvolvimento dessa atividade é semelhante à atividade *Double* Refmod, em que existem quatro entradas para gerar o *testbench:* a descrição TLN, modelo de referência, *templates* e *scripts*.

4.3.1.4.2 *Hierarchical Duv_Emulation*

Nesta etapa é feita a validação dos *Drivers, Monitors e Responders* para cada um dos submódulos do modelo de referência. No *Hierarchical* Duv*_Emulation* existem módulos dos passos anteriores que podem ser reusados, a depender de qual submódulo do modelo de referência está sendo considerado. A figura 4.18 mostra a arquitetura do *testbench* desenvolvido para a verificação funcional de um módulo com comunicação bidirecional. Para cada funcionalidade do modelo de referência é criada uma interface de comunicação que posteriormente será reusado no *testbench* final do projeto.

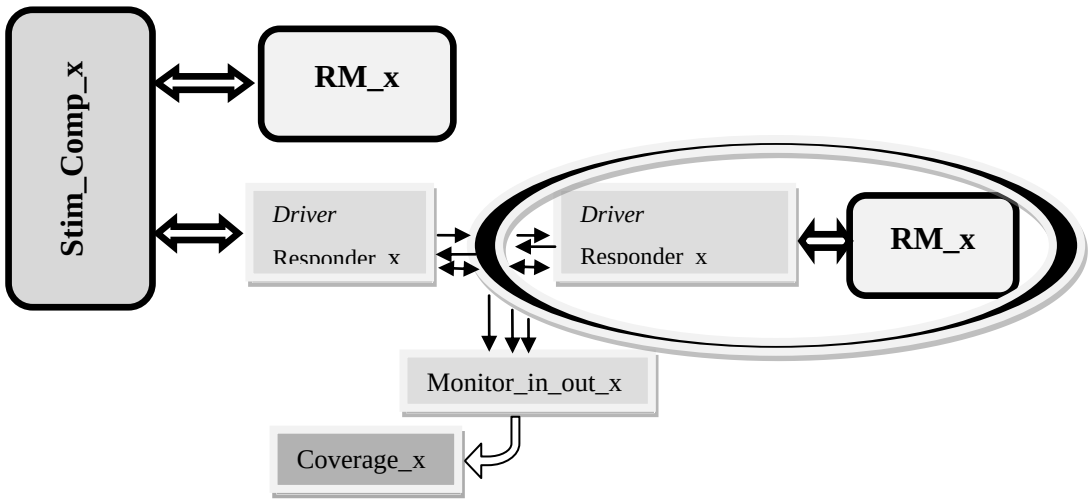


Figura 4.18: Estrutura desenvolvida para a atividade 3.2 (*Hierarchical* Duv *Emulation*)

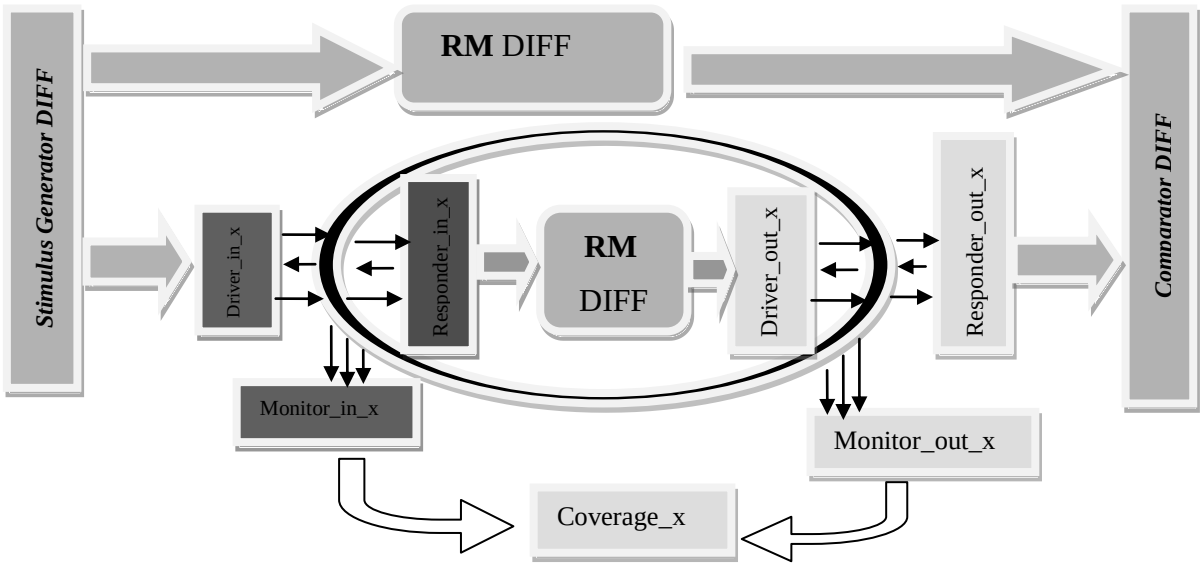


Figura 4.19: Estrutura da atividade 3.2 para o módulo DIFF do DPCM

No exemplo do DPCM, mostrado na figura 4.19, teremos para o submódulo DIFF o reuso dos seguintes módulos dos *testbenchs* criados nas atividades anteriores: todos os módulos do *Hierarchical Double* Refmod, o *driver_in*, *Responder_in*, *Monitor_in*,e parte do *coverage,* que é responsável pela cobertura de entradas de dados. Os módulos mais escuros da figura 4.19 são os que foram reusados (total ou parcialmente).

4.3.1.4.3 Hierarchical DUV

A atividade de *Hierarchical* DUV dá início à verificação do módulo do DUV descrito no nível RTL. Esse módulo RTL corresponde ao modelo de referência RM_x, onde todo o ambiente de verificação foi construído no *Hierarchical* Duv_*Emulation.*O *testbench* obtido após esta fase está mostrado na figura 4.20.

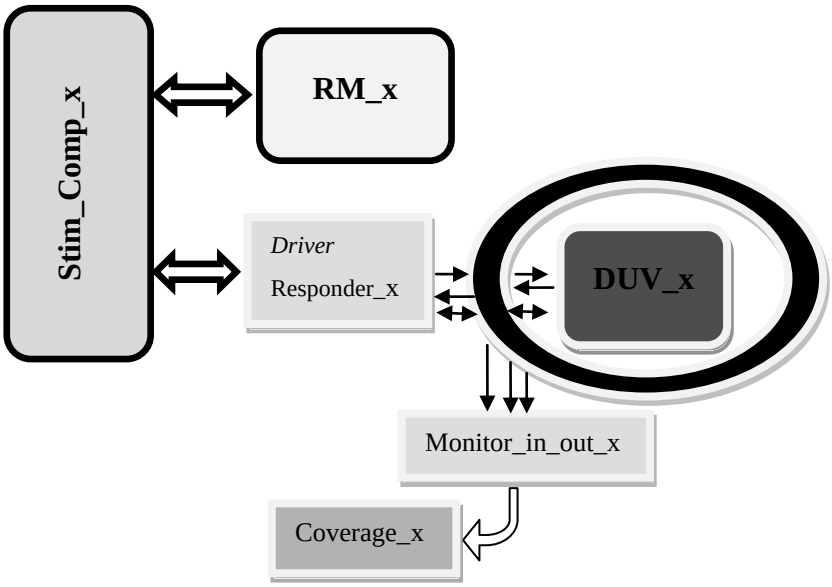


Figura 4.20: Testbench criado para a atividade 3.3 (*Hierarchical* DUV)

Antes da fase de *Hierarchical* DUV todo o *testbench* já foi construído e validado. Desta forma, o leitor pode constatar que a metodologia OVM_tpi suporta a geração de todo o *testbench* antes do DUV, assim possibilitando que as equipes de verificação e implementação de *hardware* trabalhem de forma paralela, obtendo significante redução do tempo de projeto.

No exemplo do DPCM podemos observar que, após emular o DUV, será colocado no *testbench* o código descrito no nível RTL do DIFF. A estrutura do exemplo DPCM com o submódulo DIFF é mostrado na figura 4.21.

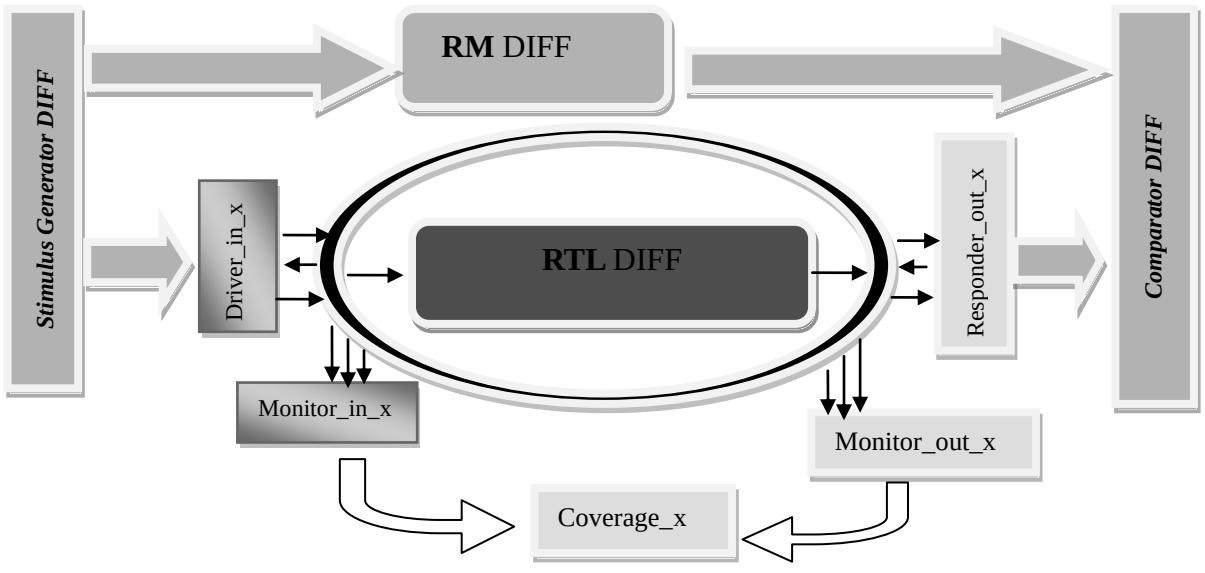


Figura 4.21: Estrutura do *Hierarchical* DUV para o módulo DIFF do DPCM

Após a finalização do *Hierarchical Testbench* para todos os módulos, inicia-se o processo de integração dos módulos.

4.3.1.5 Full Tesbench

A etapa do *Full Testbench* é a última etapa da construção de um *testbench* para o projeto de hardware. Como nesta fase todos os módulos já foram implementados e verificados separadamente, a atividade de *Full Testbench* é dividida em dois passos: a etapa de *Integration* DUV e a etapa de DUV *Execution.*

4.3.1.5.1 Integration *DUV*

Esta fase consiste na integração de todo o sistema desenvolvido, sempre verificando a funcionalidade dos módulos que estão sendo integrados. Na medida em que um módulo é integrado ao outro, o *testbench* será criado através do reuso dos ambientes criados no *Hierarchical Testbench.* Esse passo foi criado na busca de um gerenciamento sob o processo de integração do *hardware* de maneira mais detalhada e monitorada, podendo verificar se posteriormente à integração houve algum problema na integração de cada subcircuito. Essa atividade será repetida **n-1** vezes, onde **n** é o número de sub-módulos existentes no circuito e **n-1 tem que ser maior que 1**. A estrutura resultante do passo *Integration* DUV é mostrada na figura 4.22.

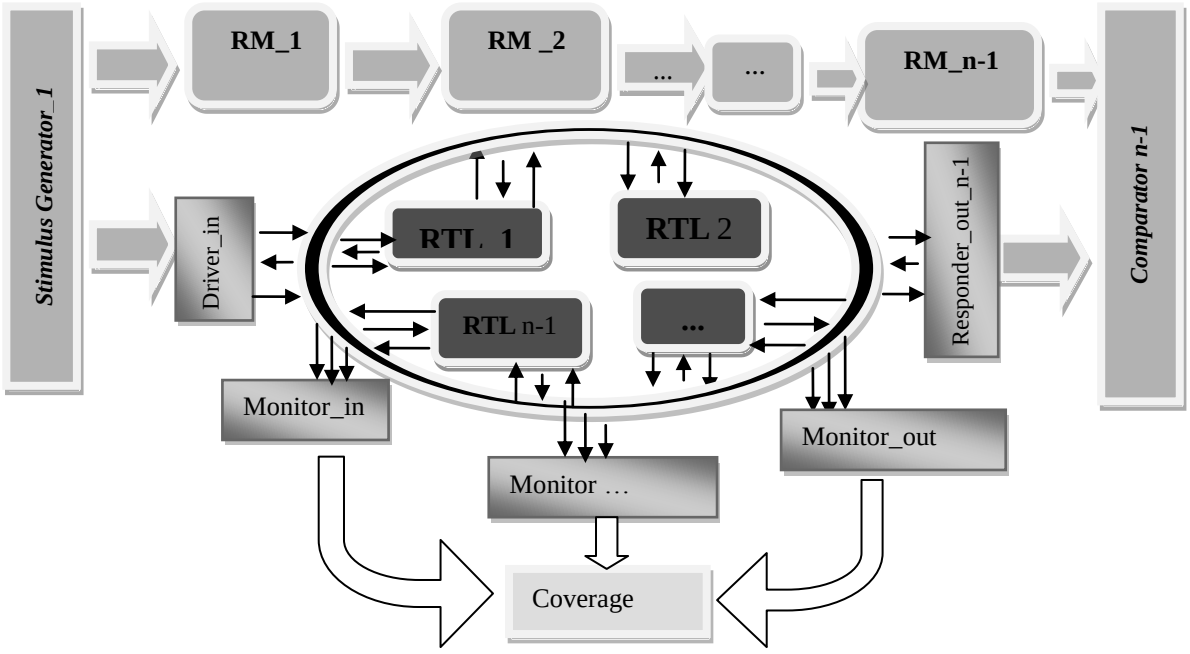


Figura 4.22: Estrutura do passo 4.1 (Integration DUV)

Caso o circuito tenha apenas dois módulos, esta atividade não é executada. Todos os submódulos serão ligados à interface de sinais do circuito. Na figura 4.22 pode-se perceber também que todos os monitores de entrada e saída dos submódulos serão ligados à interface de sinais, ou seja, os monitores criados nas atividades anteriores irão ser reusados nessa etapa.

4.3.1.5.2 *DUV* Execution

Finalizando o processo de construção do *testbench* com o fluxo de desenvolvimento *top-down*, na etapa de DUV *Execution* o Refmod hierárquico será substituído pelo modelo de referência desenvolvido e testado inicialmente, e o DUV completo será validado. Serão reusados todos os componentes criados até esse momento. Apenas necessitará gerar a parte do ambiente de conexão dos componentes.

Todos os módulos do circuito serão conectados através da interface, tornando mais simples a captura dos sinais de conexão pelos monitores. A figura 4.23 mostra a arquitetura do *testbench* final após este último passo.

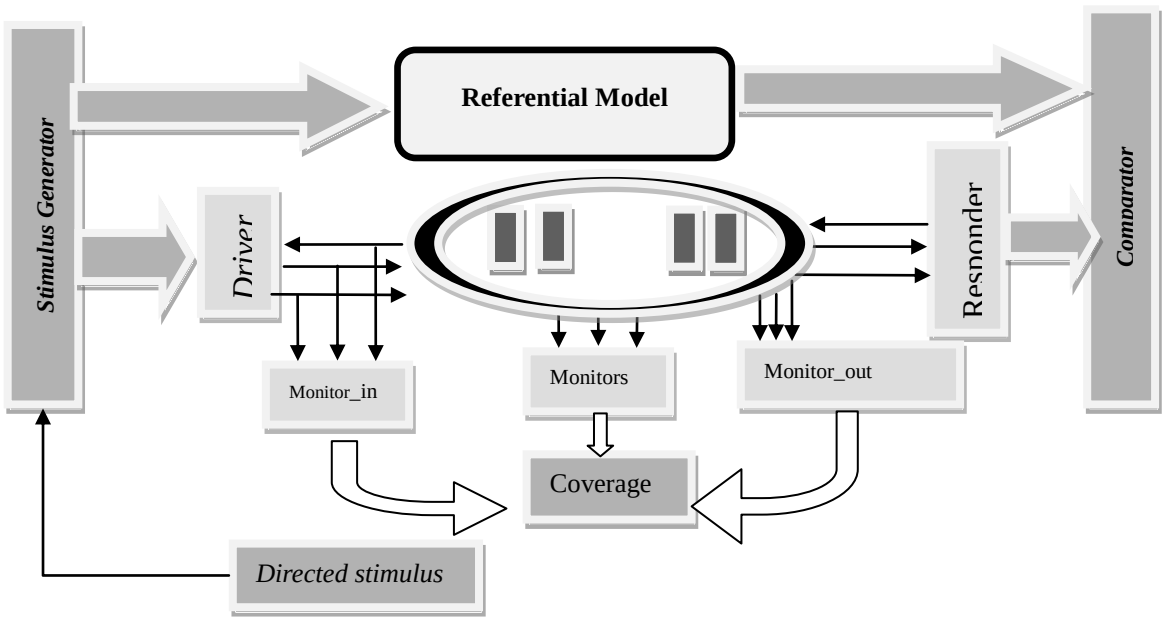


Figura 4.23: Arquitetura do passo 4.2 (Full*Testbench*)

No exemplo do DPCM não foi necessário fazer a atividade Duv *Integration*, pois só existiam dois submódulos. Desta forma, tem início a próxima (e última) atividade, que é a Duv *Execution*. A estrutura final do *testbench* criado no exemplo DPCM é mostrada na figura 4.24.

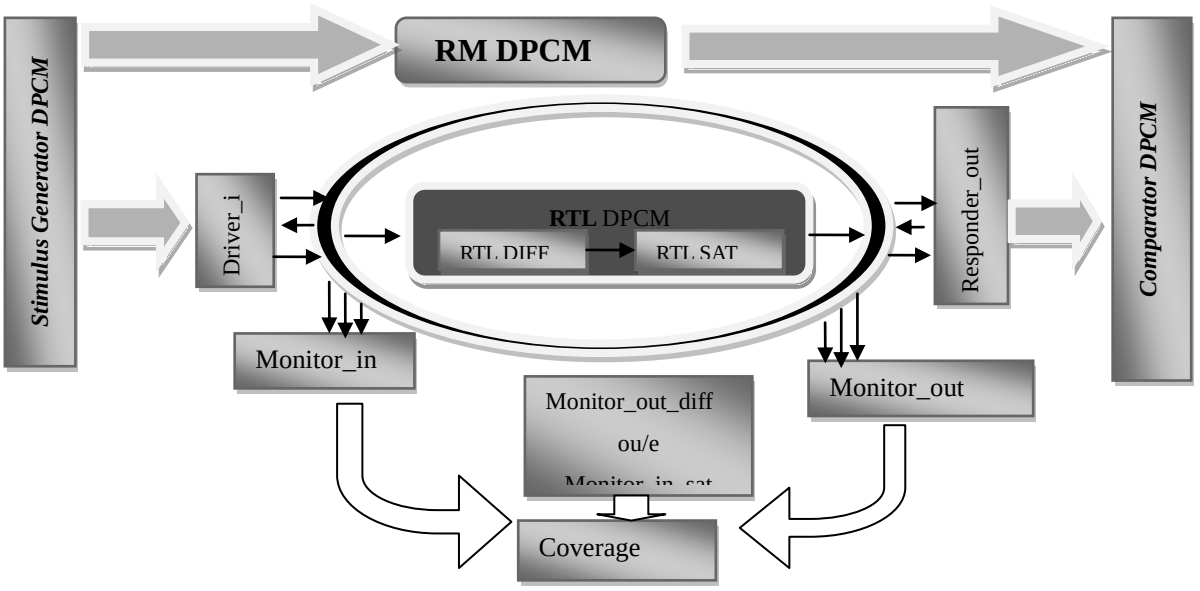


Figura 4.24: Estrutura do Duv *Execution* para o DPCM

4.3.2 Suportando o Desenvolvimento do T*estbench* em Projetos *Bottom-up*

A construção do *testbench*, seguindo um fluxo *bottom-up*, também é suportada pela metodologia OVM_tpi, que estabelece uma sequência de passos para desenvolver o *testbench* completo de forma incremental e iterativa. O *bottom-up* é um fluxo de desenvolvimento mais utilizado por linguagens orientadas a objeto. Todavia, é necessário que o gerente de projeto tenha boa experiência com esta metodologia, para que sejam estabelecidas métricas de projeto (tempo, risco, custo). O grande diferencial entre as duas abordagens suportadas por OVM_tpi é a forma de integração dos módulos do DUV, pois quando trabalhamos com o fluxo *top-down* já temos todo o sistema descrito em ESL antes de iniciar o processo de construção do *hardware*, em detrimento do fluxo *bottom-up*, em que a descrição do sistema será desenvolvida em paralelo com a construção dos módulos do circuito.

Por conseguinte, percebe-se que o fluxo de desenvolvimento do *testbench bottom-up* se torna paralelo desde o início, já que o circuito é desenvolvido em paralelo. Desta forma, diferentes equipes de construção de *testbench* podem trabalhar em diferentes submódulos, propiciando uma diminuição de tempo no desenvolvimento do projeto, porém com o uso de mais mão-de-obra.

Ademais, usando esse processo o gerente de projeto arca com um risco mais elevado, baseado na métrica da corretude final tanto do DUV quanto do ambiente de verificação, tendo toda a responsabilidade da arquitetura e da divisão do sistema, assim como das prioridades

dadas à implementação de cada submódulo do DUV e de seu respectivo *testbench*.

O desenvolvimento de um *testbench* com a metodologia OVM_tpi seguindo um fluxo *bottom-up* inclui os seguintes passos, já foram explicados na seção 4.3.1:

1. Hierarchical Testbench

Assim como num fluxo de desenvolvimento *top-down*, no fluxo *bottom-up,* a etapa *Hierarchical Testbench* é responsável pela validação de cada módulo que compõe o modelo de referência, e é repetida tantas vezes quantos forem os submódulos do modelo de referência. Isso ocorre porque o *Hierarchical Testbench* será feito para cada submódulo, a fim de verificar cada funcionalidade descrita. Esse passo é dividido em três subpassos, a seguir descritos.

1.1. Hierarchical Double Refmod

Nesta etapa serão desenvolvidos e validados o gerador de estímulos e o comparador, garantindo que o ambiente possua capacidades mínimas de funcionamento, com comunicação e detecção de erros. Essa atividade é desenvolvida para cada módulos desenvolvido, separandamente. A figura 4.25 mostra o *testbench* desenvolvido para esta atividade.

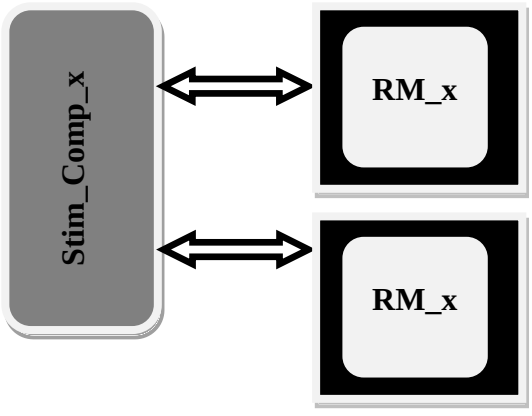


Figura 4.25: *Testbench* desenvolvido para a atividade Hierarchical Double Refmod

1.2. *Hierarchical* Duv *Emulation*

Nesta etapa é feita a validação dos *Drivers, Monitors e Responders* para cada um dos submódulos do modelo de referência. No *Hierarchical* Duv*_Emulation* existem módulos dos

passos anteriores que podem ser reusados, a depender de qual submódulo do modelo de referência está sendo considerado. A figura 4.26 mostra o *testbench* desenvolvido para esta etapa.

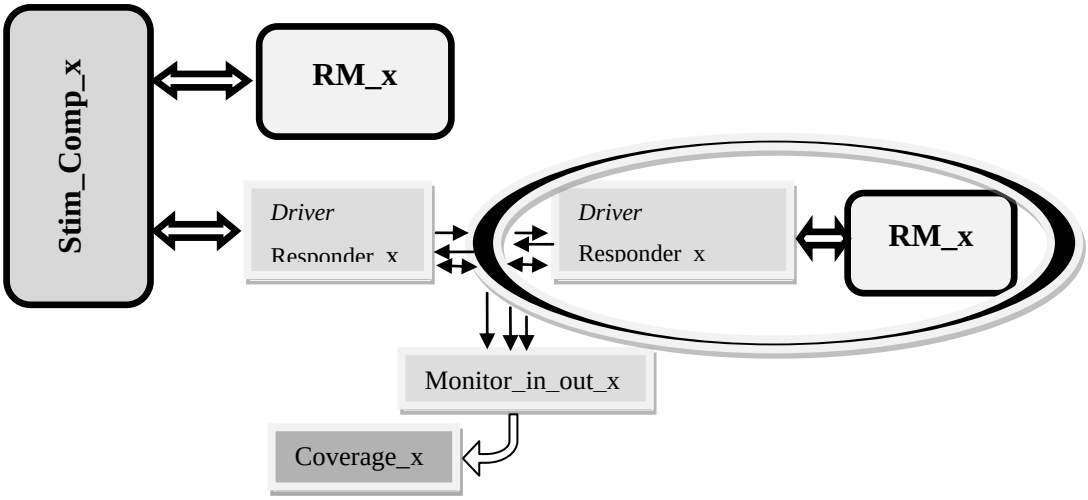


Figura 4.26: Testbench desenvolvido para a atividade de Hierarchical *DUV_Emulation*

1.3. Hierarchical Duv

A atividade de *Hierarchical* DUV dá início à verificação do módulo do DUV descrito no nível RTL. Esse módulo RTL corresponde ao modelo de referência RM_x, sendo que seu ambiente de verificação foi construído no *Hierarchical* Duv*_Emulation.*O *testbench* obtido após esta fase está mostrado na figura 4.27.

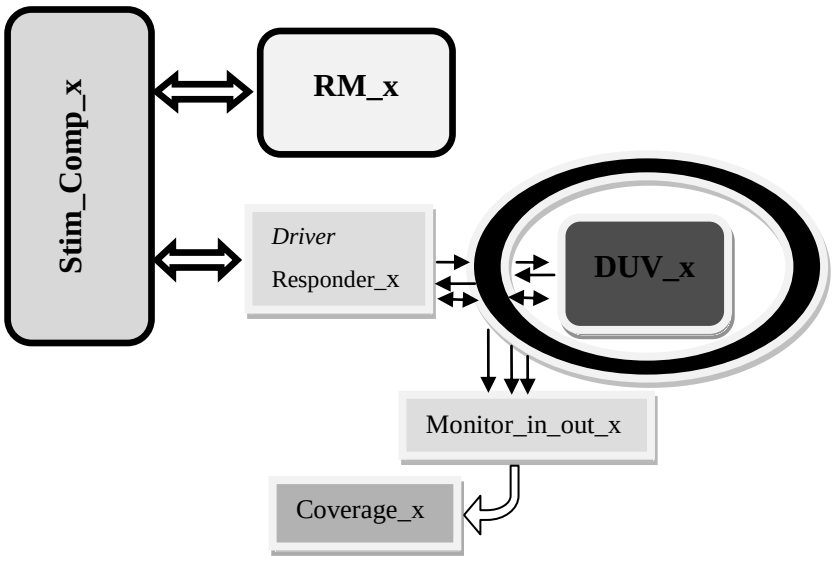


Figura 4.27: *Testbench* deselvolvido para a atividade de *Hierarchical_DUV*

2. Full Testbench

A etapa do *Full Testbench* é a última etapa da construção de um *testbench* para o projeto de hardware. Como nesta fase todos os módulos já foram implementados e verificados separadamente, a atividade de *Full Testbench* é dividida em dois passos: a etapa de *Integration* DUV e a etapa de DUV *Execution.*

2.1. Integration Duv

Esta fase consiste na integração de todo o sistema desenvolvido, sempre verificando a funcionalidade dos módulos que estão sendo integrados. Na medida em que um módulo é integrado ao outro, o *testbench* será criado através do reuso dos ambientes trazidos pelo *Hierarchical Testbench.* Esse passo busca um gerenciamento sob o processo de integração do *hardware* de maneira mais detalhada e monitorada, podendo verificar se posteriormente à integração houve algum problema nos subcircuitos. Essa atividade será repetida **n-1** vezes, onde **n** é o número de sub-módulos existentes no circuito e **n-1 tem que ser maior que 1**.

2.2. DUV *Execution*

Finalizando o processo de construção do *testbench* com o fluxo de desenvolvimento *bottom-up,* na etapa de DUV *Execution* o Refmod será um objeto principal, instanciando todos os modelos de referências dos DUV verificados anteriormente, e o DUV completo será validado. Serão reusados todos os componentes criados até esse momento, apenas necessitando gerar a parte do ambiente de conexão dos componentes.

Todos os módulos do circuito serão conectados através da interface, tornando mais simples a captura dos sinais de conexão pelos monitores. A figura 4.28 mostra a arquitetura do *testbench* final após esta última atividade.

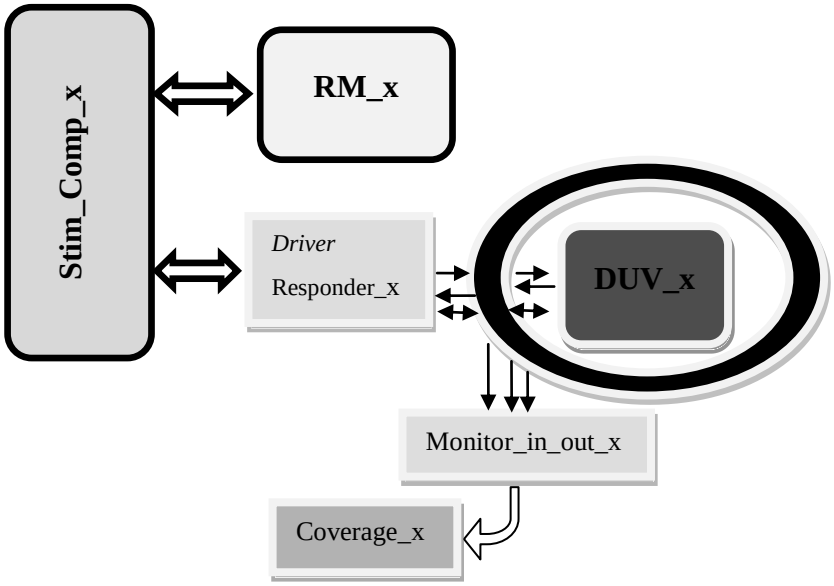


Figura 4.28: *Testbench* desenvolvido para o desenvolvimento da atividade de *DUV*_Execution

Todos os passos mencionados são realizados de forma semelhante aos passos descritos para o fluxo *top-down*, e por isso não serão detalhados novamente nessa seção. O estudo de caso da Memória *Dual Port* foi construído seguindo um fluxo *bottom-up* de desenvolvimento, e será apresentado no capítulo 6.

4.4 Suportando a Análise de Cobertura

O princípio básico da verificação funcional é simular a descrição RTL de um sistema digital com a inserção de estímulos na entrada, com posterior comparação dos resultados obtidos, na sua saída, com a saída do Modelo de Referência. Entretanto, um dos grandes desafios da verificação funcional é analisar se todas as funcionalidades especificadas foram testadas. Na busca por esse gerenciamento, foram desenvolvidas algumas técnicas de captação dos dados trafegados pelo *testbench,* sem influenciar no processo de simulação. Denomina-se essa técnica de cobertura funcional.

Cobertura é responsável por medir o progresso da verificação funcional através de várias métricas preestabelecidas, que auxiliarão o engenheiro a se localizar em relação ao término da verificação[39]. Ela mede o progresso da simulação e reporta quais funcionalidades não foram exercitadas, ou foram exercitadas mais de uma vez. Além disso, ela pode também ajudar a inspecionar a qualidade da verificação e direcionar os estímulos de forma a alcançar as funcionalidades não cobertas, os chamados buracos de cobertura.

OVM_tpi se baseia na biblioteca Coverage e Assertions de *SystemVerilog* para gerar o suporte necessário à inserção de cobertura no *testbench*, e assim conseguir gerenciar o processo de verificação de forma mais coerente e afirmativa. O uso dessas bibliotecas é um ponto forte da metodologia, em relação à IVM e VeriSC, por conter formas de coberturas mais complexas e necessárias, e não encontrada em ambas. Pode-se citar, como exemplo, o cruzamento (C*ross Coverage)* para a cobertura de dados e o uso de *Assertions* para a cobertura de controle. Portanto, OVM_tpi faz o uso da biblioteca *Coverage* para realizar a cobertura de dados e *Assertions* para realizar a cobertura de controle da comunicação.

Observa-se que a metodologia OVM_tpi foi construída para ser um tipo de metodologia baseada em cobertura, mas que inicia o uso de *Assertions* no ambiente de simulação, seguindo a orientação metodológica de ABV para o tipo de verificação caixa-preta. Caso a equipe de design construa os módulos RTL já com o uso de ABV dentro do código, a verificação funcional será conceitualmente a junção das duas: dirigida a cobertura e baseada em *Assertions*.

Nos componentes denominados *Monitors* é especificado o valor de cada sinal do protocolo de controle do *testbench*. Caso ocorra algum problema de protocolo, o monitor envia imediatamente uma informação do erro ocorrido.

Para fazer a cobertura de dados com a metodologia OVM_tpi, foi desenvolvido um componente específico para este fim; assim sendo, o componente *Coverage* concentra toda a cobertura de dado do *testbench*, tornando mais simples o processo de inserção de cobertura no ambiente de verificação. No interior do componente *Coverage* é utilizada cobertura baseada em regras suportadas pela biblioteca *Coverage de SystemVerilog:* cobertura de pontos, cruzamento de cobertura e cobertura de transição. A “casca” do componente *Coverage* é gerada com toda a estrutura necessária para a descrição das regras descritas no plano de projeto.

No Capítulo 5 será detalhado o processo de geração de código automática suportada pela metodologia OVM_tpi.

Capítulo 5 Mecanismo Para Geração Semiautomática do *Testbench* Seguindo a Metodologia OVM_tpi

De acordo com o trabalho de Dueñas[17], 65% dos IP cores falham em sua primeira prototipação em silício, e 70% destes casos são devidos a uma verificação funcional mal feita. Já Bergeron afirma que tal índice pode chegar a 74%[8]. Por esse motivo, a maior parte do tempo de um projeto de hardware, cerca de 70%, é gasto na verificação funcional[43]. Quanto maior o espaço de tempo gasto em um projeto, mais elevado será seu custo.

O objetivo do presente capítulo é detalhar como se dá a construção semiautomática do *testbench*, utilizando a metodologia OVM_tpi. Com essa semiautomatização espera-se um ganho de tempo considerável em todo o processo de verificação, e consequentemente no projeto do circuito como um todo. Para suportá-la, foi necessário o uso e modificação de uma ferramenta específica para este fim, denominada eTBc (*Easy Testbench Creator*). Tal ferramenta fora criada para suportar a semiautomatização da metodologia VeriSC e já foi utilizada pela metodologia BVM, permitindo sua semiautomatização. Todo esse histórico foi decisivo para sua utilização na metodologia OVM_tpi, para também suportar a semiautomatização na criação do *testbench*.

Com a escolha da ferramenta eTBc, foi desenvolvido um pacote de *templates* para a mesma, que será utilizado pelos engenheiros de verificação funcional que decidirem utilizar a semiautomatização suportada pela metodologia OVM_tpi.

5.1 Pacote de *Templates* Desenvolvido Para a Metodologia OVM_tpi

A biblioteca de *templates* OVM_tpi foi criada para agilizar o processo de verificação funcional, através da utilização desses objetos na criação do *testbench*. Atualmente a biblioteca OVM_tpi inclui componentes que se comunicam pelo protocolo HFPB (Harry Foster Peripheral bus) de comunicação[22]. Esse protocolo foi escolhido pela sua simplicidade, além da facilidade de definição da sua máquina de estados. Por sua vez, o HFPB foi baseado no protocolo Amba APB 2.0 da ARM[4]. No entanto, a ideia principal dessa biblioteca de *templates* é poder contar com outras opções de protocolos e, de acordo

com a especificação do módulo, de forma que possa ser utilizado o protocolo de comunicação ideal tanto para o t*estbench* quanto para a utilização do módulo. Desenvolver os templates dos componentes com protocolo é um dos passos importante para a geração automática dos componentes, pois caso o módulo não esteja com o mesmo protocolo do testbench, o engenheiro de verificação terá que optar por fazer a mudança para o protocolo do DUV ou sincronizar o protocolo do testbench com o protocolo do DUV.

O protocolo de *handshake* que foi implementado até o momento utiliza dois sinais para a sincronização da comunicação: um sinal de *enable* e outro denominado *select*, sendo através do gerenciamento desses sinais que o *Driver* irá enviar de forma organizada os dados recebidos do *Stimulus_Generator* para o DUV, e o *Responder* irá receber a resposta e enviá-la para o Comparador no momento correto. O protocolo se comporta de acordo com a máquina de estados mostrada na figura 5.1:

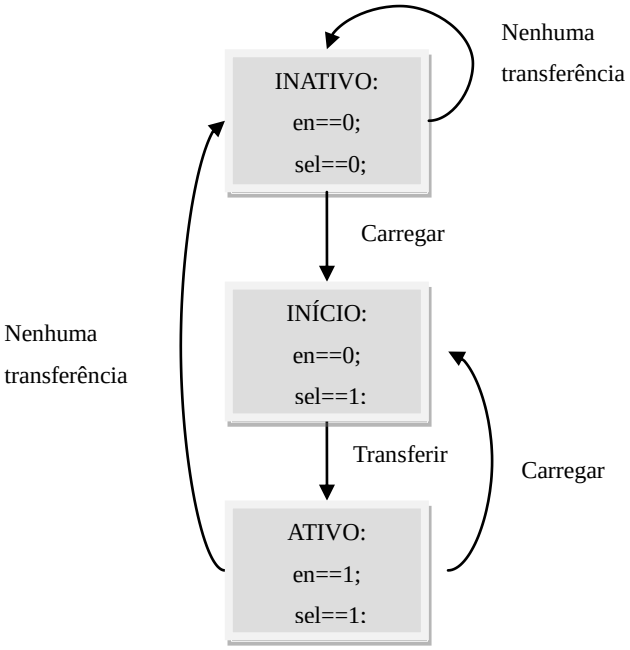


Figura 5.1: Máquina de Estados HFPB

Apenas quando os dois sinais, *enable* e *select* estão no nível alto é que pode ser enviado ou requisitado um dado para o DUV.

Foram desenvolvidos trinta e três templates com aproximadamente duas mil e quinhentas linhas de códigos, para serem utilizados no processo de semiautomatização. A seguir serão classificados todos os *templates* desenvolvidos para a metodologia OVM_tpi.

5.1.1 Classificação da Biblioteca de *Templates*

Para desenvolver o processo de semiautomação da criação de *testbench* para a metodologia OVM_tpi, foi implementado uma biblioteca de *templates*. Essa biblioteca é utilizada para gerar os componentes necessários no desenvolvimento do ambiente de verificação funcional. Para facilitar o entendimento do processo de semiautomação, todos os *templates* foram classificados em três conjuntos: básicos, componentes do testbench e empacotamento.

O conjuntos dos *templates* básicos, por sua vez, é subdividido em dois subconjuntos: as classes e as interfaces. As classes são objetos desenvolvidos para criação dos estímulos randômicos, conexão de todos os componentes do *testbench* e execução do ambiente de verificação. As interfaces têm a finalidade de tornar o ambiente de simulação mais desacoplado. O conjunto então é composto pelos seguintes itens:

Classes:

tmp_trans – arquivo que tem o objetivo de descrever todas as estruturas existentes no *testbench* unidirecional. É criado pelo eTBc, sem a necessidade de intervenção do engenheiro. Porém, o engenheiro de verificação pode fazer qualquer mudança necessária manualmente, sem haver nenhum problema ou dificuldade;

tmp_trans_bir - arquivo que tem o objetivo de descrição de todas as estruturas existentes no *testbench* bidirecional. Criado pelo eTBc através desse *template,* sem a necessidade de intervenção do engenheiro. Todavia, o engenheiro de verificação pode fazer qualquer mudança necessária manualmente, sem haver nenhum problema ou dificuldade;

tmp_top_d*ouble*_refmod – é utilizado para gerar o *toplevel* para o passo *Double* Refmod e *Hierarchical Double* Refmod*;*

tmp_clk_rst_gen – esse é o *template* que servirá de entrada para o eTBc instanciar o gerador de *clock* do ambiente de simulação. Foi desenvolvido para dar suporte a mudança da sensibilidade do sinal de *Reset,* e gera *clock* até o ambiente ser finalizado, assim como também totalmente parametrizado. Só necessitará de mudanças caso os parâmetros locais sejam diferentes;

tmp_env_*Double*_refmod – objeto que cria toda a ligação do *testbench* com as TLMs existentes nos passos *Double* Refmod e *Hierarchical Double* Refmod. Não é necessária nenhuma mudança manual no objeto gerado;

tmp_duv_*emulator* – criado para ser a “casca” do emulador do DUV. Não é necessária nenhuma modificação no objeto que será criado. Só é utilizado no passo em que há emulação

do DUV;

tmp_env_duv_*emulation* - objeto que cria toda a ligação do *testbench* com a interface emulada do DUV e com as TLMs existentes. Não é necessária nenhuma mudança no objeto gerado;

tmp_top_duv_*emulation* - é utilizado para gerar o *toplevel* para o passo DUV *Emulation* e *Hierarchical* DUV *Emulation*;

tmp_env_duv - objeto que cria toda a ligação do *testbench* com a interface do DUV e com as TLMs existentes. Não é necessária nenhuma mudança no objeto gerado;

tmp_top_duv - é utilizado para gerar o *toplevel* para o passo DUV *Execution, Hierarchical* DUV e *Integration* DUV.

Interfaces:

tmp_clk_rst_if - o **tmp_clk_rst_if** é a interface da classe **tmp_clk_rst_gen**, e sempre necessitará ser construída quando existir um **tmp_clk_rst_gen**. Através dessa interface podemos especificar diferentes *clocks* que podem ser utilizados no ambiente de simulação;

tmp_signal_interface* –** um dos principais *templates* que tem todos os sinais pertencentes ao DUV, além dos sinais que implementam o protocolo de comunicação do *testbench*. Este módulo gerado inclui todas as interfaces que utilizarão sinais, pois todos os sinais do *testbench* passam por essa interface. Apenas naqueles passos em que o DUV é usado deve haver uma mudança manual no nome dos sinais ***en1 e ***sel1*** que são recebidos pelo responder e monitor para **en** e **sel**;

tmp_interface_bir – tem o mesmo papel do objeto **tmp_signal_interface**, porém utilizado em verificação de módulos com comunicação bidirecionais. Não é necessária nenhuma mudança manual no módulo gerado, visto que é utilizado apenas o **tmp_dr_resp** para requisição e envio de respostas;

tmp_refmod – *template* que cria a "casca" do modelo de referência. Foi desenvolvida focando nos conceitos de orientação a objeto, em que apenas é necessário instanciar a chamada de função para a função desejada do modelo de referência passando todos os parâmetros necessários.

tmp_refmod_bir – *template* que cria a "casca" do modelo de referência para comunicação bidirecional. Assim como o *template* **tmp_refmod**, foi desenvolvida focando nos conceitos de orientação a objeto, em que apenas é necessário instanciar a chamada de função para a função desejada do modelo de referência passando todos os parâmetros necessários.

Componentes:

***tmp_stimulus_generator* –** *template* que cria o objeto *Stimulus Generator*. A criação desse arquivo foi idealizada para trabalhar com estímulos randômicos gerados a partir das estruturas criadas no objeto **tmp_trans.svh**. Contudo, antes de gerar um estímulo aleatório o objeto verifica se existe um estímulo direto que ele deve passar para o ambiente de verificação. Caso haja estímulos direcionados criados, estes irão primeiramente para o ambiente de verificação, e após isso é que iniciará o envio de estímulos aleatórios;

***tmp_comparator* –** esse *template* foi descrito com o objetivo de ser modificado apenas quando o engenheiro de verificação queira observar as saídas divergentes ou queira modificar o número de erros que devem ser observados para parar a simulação (por exemplo, quando encontrar três erros pare a simulação). Por *default* não é necessária nenhuma mudança nesse objeto criado;

***tmp_driver* –** o *template driver* é responsável pela descrição do *driver* do ambiente de simulação. No arquivo gerado neste passo geralmente deve haver interferência manual do engenheiro de verificação para organizar o processo de comunicação com o DUV. Porém, como estamos na atividade de emulação do DUV, o *driver* irá se comunicar com um *responder* que emulará a interface de sinal do DUV, e transformará os sinais recebidos em transação para o modelo de referência instanciado na emulação;

tmp_responder_in_*emulation* – é responsável pela criação do *responder*, que será utilizado no emulador do DUV nos passos de emulação. São necessários alguns ajustes manuais no objeto que será criado para o funcionamento adequado;

tmp_dr_out_e*mulation* – é utilizado apenas nos passos de emulação do DUV, tendo a mesma funcionalidade do *driver*, porém utilizando outros dois sinais (**en1** e **sel1**) para se comunicar com o *responder* e o *monitor_out.* Geralmente há modificações a serem feitas para prover a comunicação correta entre os módulos;

***tmp_responder* –** com esse *template*, o eTBc cria a classe *responder,* que irá receber sinais de resposta do DUV e transformá-los em transações. São necessários ajustes manuais nos locais especificados no arquivo;

***tmp_monitor*_in –** utilizado para gerar o monitor das entradas do DUV. Assim como no *driver* e no *responder,* é necessário mudança para capturar os sinais corretos, no tempo certo;

***tmp_monitor*_out -** utilizado para gerar o monitor das saídas do DUV. Assim como no *driver* e no *responder*, são necessárias mudanças para capturar os sinais corretos, no tempo certo.

Nos passos que utilizam o DUV é necessário uma mudança manual no nome dos sinais de controle da comunicação;

tmp_coverage **–** a responsabilidade desse módulo é receber uma transação com o dado monitorado pelo *monitor* e verificar se ele participa do conjunto de dados que devem passar pela entrada e pela saída do DUV. O *template* desse módulo terá que ser completado manualmente, visto que a decisão de escolha dos conjuntos para a cobertura ainda é feita manualmente. Para este módulo, é gerado o componente contendo as interfaces de conexão com o ambiente de verificação e os grupos de cobertura implementadas, faltando apenas inserir as regras de cobertura, de acordo com a especificação dentro dos grupos criados. Um exemplo é mostrado na figura 5.2:

```
covergroup crm;
  coverpoint tr_in_sample_in.s_value {
      bins cov_in[] = { [-4:3] };
      illegal_bins il_in = { [$:-5], [4:$] };
      option.at_least = 1000;
   }
  coverpoint tr_out_sample_out.s_value {
      bins cov_out[] = { [-4:3] };
      illegal_bins il_out = { [$:-5], [4:$] };
      option.at_least = 1000;
   }
endgroup
```

Figura 5.2: Exemplo de grupo criado no *Coverage*

Toda a descrição da cobertura é feita manualmente pelo engenheiro de verificação;

tmp_dr_resp – responsável pela comunicação com o DUV quando o sistema é bidirecional, esse *template* gera tanto a forma de envio dos dados por sinais para o DUV quanto o envio das respostas para o stim_comp através de transações;

tmp_stim_comp – esse *template* é utilizado para a criação do objeto que irá estimular e comparar as respostas recebidas no ambiente de simulação criado. Só é utilizado esse arquivo quando se trabalha com módulo de comunicação bidirecional.

Empacotamento:

Empacotamento são templates desenvolvidos para auxiliar no processo de desenvolvimento do *testbench*. Esses templates têm a finalidade de instanciar componentes, módulos e classes, fazendo a ligação entre eles.

tmp_tb_run – será responsável por criar o arquivo que será executado para rodar a simulação. É necessário fazer uma mudança manual após a criação: mudar o local onde está

descrito o diretório da biblioteca OVM;

tmp_tr_run_duv_*emulation* – tem a mesma funcionalidade do **tr_run** do *Double*_Refmod, havendo mudanças sintáticas inseridas para que funcione no Duv_*Emulation*;

tmp_tb_run_duv – tem a funcionalidade de gerar o arquivo responsável por executar a simulação;

***tmp_double*_refmod_pkg –** esse *template* tem o objetivo de fazer o empacotamento de todas as classes criadas para o passo *Double* Refmod. Serão instanciados todos os objetos criados, necessários para o funcionamento do ambiente de simulação. Não é necessária nenhuma mudança nesse arquivo;

tmp_duv_*Emulation*_pkg – esse objeto é utilizado apenas nos passos Duv_*Emulation* ou *Hierarchical*_Duv_*Emulation*, tenho por escopo construir um pacote com todos as classes necessárias para a execução do passo;

tmp_duv_execution_pkg – cria o pacote com todas as classes desenvolvidas nesse passo. Não é necessária nenhuma mudança no objeto gerado.

Para deixar mais claro o processo de semiautomatização da construção do ambiente de verificação utilizando a metodologia OVM_tpi, será explicado o processo de semiautomatização com o uso dos *scripts* desenvolvidos. Toda a automatização é desenvolvida por atividade, ou seja, na medida que a atividade for desenvolvida, todos os componentes necessários para o desenvolvimento daquela atividade serão criados.

5.1.2 Semiautomatização da Atividade *Double* Refmod

Como foi visto no Capítulo 4, o objetivo deste passo é validar o gerador de estímulos e do comparador; para isso, são necessários os seguintes objetos: o modelo de referência, a TLN descrita do módulo, a biblioteca de *templates* e a biblioteca de *scripts* que será responsável pela criação dos diretórios e a execução dos comandos para o eTBc.

A figura 5.3 mostra o fluxo de desenvolvimento da atividade de *Double* Refmod.

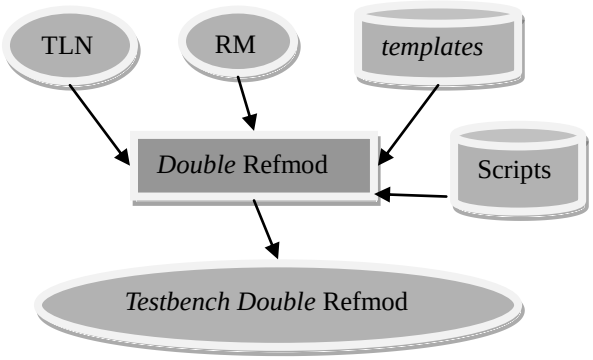


Figura 5.3: Fluxo de desenvolvimento da atividade *Double* Refmod

Podemos observar na figura 5.3 que o fluxo de automatização irá gerar a arquitetura de *testbench* do *Double* Refmod.

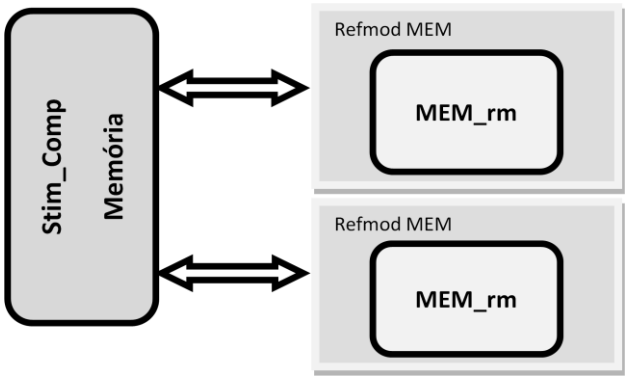


Figura 5.4: Arquitetura de testbench gerada

Após a geração automática é necessário alguns ajustes manuais para o *testbench* funcionar de forma adequada. O processo inicial para a geração desse ambiente de verificação é a escolha do script correto.

O *script* utilizado para construir o diretório e os módulos para o *Double* Refmod pode ser visto na figura 5.5.

```
#!/bin/bash

unset ETBC_TLN_DIR
unset ETBC_TMPL_DIR
export ETBC_TLN_DIR=/usr/local/share/etbc/tln
export ETBC_TMPL_DIR=/usr/local/share/etbc/templates/Ovm_tpi
rm -rf ../testbench_conception_dpcm

cd ..
mkdir testbench_conception_dpcm
cd testbench_conception_dpcm


echo ""
echo "|-------- Starting Step 1 - Testbench Conception"
echo ""

mkdir double_refmod_dpcm
cd double_refmod_dpcm

echo ""
echo "|-------- Starting Step 1.1 - Double Refmod"
echo ""
ln -s ../../double_refmod_generic/tb_clean.run tb_clean.run
etbc dpcm.tln tb_run dpcm
etbc dpcm.tln refmod dpcm
etbc dpcm.tln stimulus_generator dpcm
etbc dpcm.tln comparator dpcm
etbc dpcm.tln trans dpcm
etbc dpcm.tln top_double_refmod dpcm
etbc dpcm.tln clk_rst_if dpcm
etbc dpcm.tln clk_rst_gen dpcm
etbc dpcm.tln env_double_refmod dpcm
etbc dpcm.tln double_refmod_pkg dpcm
touch dpcm.sti
```

Figura 5.5: *Script* para a criação dos arquivos da etapa *Double* Refmod

Esse script é descrito na linguagem **bash**. Pode-se observar que a primeira coisa que o *script* faz é definir os locais onde estão a biblioteca de *template* da metodologia OVM_tpi e a TLN. Posteriormente, cria a pasta onde será desenvolvida a atividade de *testbench_conception* e dentro do diretório é criado outro diretório, denominado d*ouble*_refmod_dpcm. Por conseguinte, é feito o link para um arquivo genérico que é utilizado em todas as etapas. O arquivo **tb_clean.run** tem como objetivo apagar todos os arquivos desnecessários para uma próxima simulação. É necessário utilizá-lo sempre antes de uma nova simulação. Seguindo o *script,* é iniciado o uso do eTBc para desenvolver todos os módulos necessários para a construção do *testbench* da etapa.

Os *templates* necessários para o desenvolvimento desta etapa num ambiente de comunicação unidirecional são: **tmp_tb_run, tmp_refmod, tmp_*stimulus*_generator, tmp_comparator, tmp_trans, tmp_top_d*ouble*_refmod, tmp_clk_rst_gen, tmp_env_*double*_refmod, tmp_*double*_refmod_pkg, tmp_clk_rst_if.**

Caso o módulo a ser verificado tenha uma comunicação bidirecional, os *templates* utilizados serão: **tmp_tb_run, tmp_stim_comp, tmp_trans_bir, tmp_refmod_bir, tmp_clk_rst_gen, tmp_env_*double*_refmod_bir, tmp_*double*_refmod_pkg, tmp_top_d*ouble*_refmod.**

Após o uso do eTBc ser finalizado, é executado um comando **touch** que criará um arquivo sem nenhum caractere, para que possam ser descritos os estímulos diretos para o ambiente de verificação. A escrita dos estímulos deve ser feita de acordo com a estrutura da transação construída no módulo **tmp_trans.svh**.

O modelo de referência também é um arquivo de entrada, e este é incluído no diretório através de um link que contém o modelo de referência. Após todos os objetos necessários estarem no diretório, pode ser executado o comando tb.run e verificar se tudo está correndo como deveria.

5.1.3 Semiautomatização Para a Etapa de Duv_*Emulation*

A etapa de semiautomatização das atividades do Duv_*Emulation* é semelhante à do *Double*_refmod. Porém, são utilizados como entrada para este processo os módulos do *testbench Double*_Refmod. O *script* utilizado nesta etapa para o exemplo do DPCM pode ser visto na figura 5.6:

```
#!/bin/bash

unset ETBC_TLN_DIR
unset ETBC_TMPL_DIR
export ETBC_TLN_DIR=/usr/local/share/etbc/tln
export ETBC_TMPL_DIR=/usr/local/share/etbc/templates/Ovm_tpi
rm -rf ../testbench_RSSI/hierarchical_duv_emulation_RSSI

cd ../testbench_RSSI
mkdir hierarchical_duv_emulation_RSSI
cd hierarchical_duv_emulation_RSSI
ln -s ../hierarchical_double_refmod_RSSI/stimulus_generator.svh stimulus_generator.svh
ln -s ../hierarchical_double_refmod_RSSI/comparator.svh comparator.svh
ln -s ../hierarchical_double_refmod_RSSI/trans.svh trans.svh
ln -s ../hierarchical_double_refmod_RSSI/clk_rst_gen.sv clk_rst_gen.sv
ln -s ../hierarchical_double_refmod_RSSI/clk_rst_if.sv clk_rst_if.sv
ln -s ../hierarchical_double_refmod_RSSI/RSSI.sti RSSI.sti
ln -s ../hierarchical_double_refmod_RSSI/RSSI_refmod.svh RSSI_refmod.svh
ln -s ../hierarchical_double_refmod_RSSI/RSSI_if.sv RSSI_if.sv
etbc landell.tln driver RSSI
etbc landell.tln responder_in_emulation RSSI
etbc landell.tln dr_out_emulation RSSI
etbc landell.tln responder RSSI
etbc landell.tln monitor_in RSSI
etbc landell.tln monitor_out RSSI
etbc landell.tln duv_emulator RSSI
etbc landell.tln env_duv_emulation RSSI
etbc landell.tln top_duv_emulation RSSI
etbc landell.tln duv_emulation_pkg RSSI
etbc landell.tln tb_run_emulation RSSI
etbc landell.tln coverage RSSI
```

Figura 5.6: *Script* da atividade Duv_*Emulation*

Assim, nota-se que as dez primeiras linhas do arquivo são comandos já explicados na etapa de *Double*_Refmod. Desta forma, será explicado todo o *script* a partir da linha onze, onde tem início o processo de fazer *link* dos arquivos que foram desenvolvidos e validados na atividade de *Double*_Refmod devido ao reuso de componentes. Portanto, os comando entre as linhas onze e dezesseis são apenas cópia dos arquivos que serão reusados no Duv_*Emulation* para a criação do *testbench* Duv_*Emulation*.

Após esse comando, o eTBc é novamente executado para produzir os blocos que terão como *templates* os seguintes arquivos: **tb_run_duv_*Emulation*, interface, driver, responder_in_*Emulation*, dr_out_*Emulation*, responder, monitor_in, monitor_out, coverage, duv_emulator, env_duv_*Emulation*, top_duv_*Emulation*, duv_*Emulation*_pkg.**

Se o DUV for de comunicação bidirecional, os *templates* necessários são: **tb_run_duv_*Emulation*, interface_bir, dr_resp, monitor_bir, coverage, duv_emulator_bir, env_duv_*Emulation*_bir, top_duv_*Emulation*, duv_*Emulation*_bir_pkg.**

Assim como no *Double*_Refmod, é necessário instanciar o modelo de referência para o diretório em que está sendo desenvolvido o ambiente de verificação através de um link. A figura 5.7 mostra o fluxo de automatização da geração do *testbench* e a figura 5.8 a arquitetura do ambiente de verificação gerado.

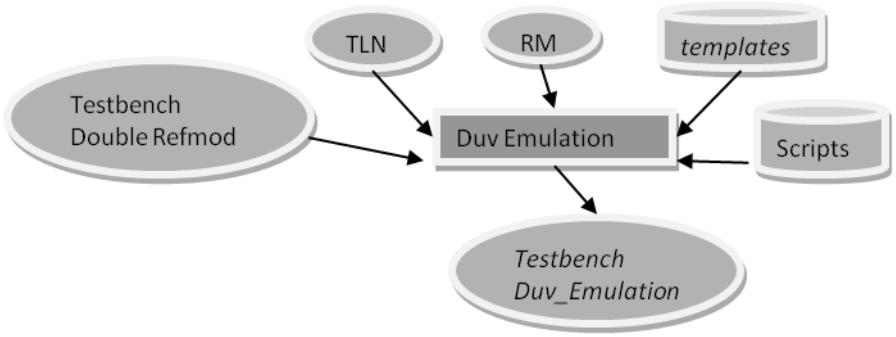


Figura 5.7: Fluxo de automatização da geração do *testbench*

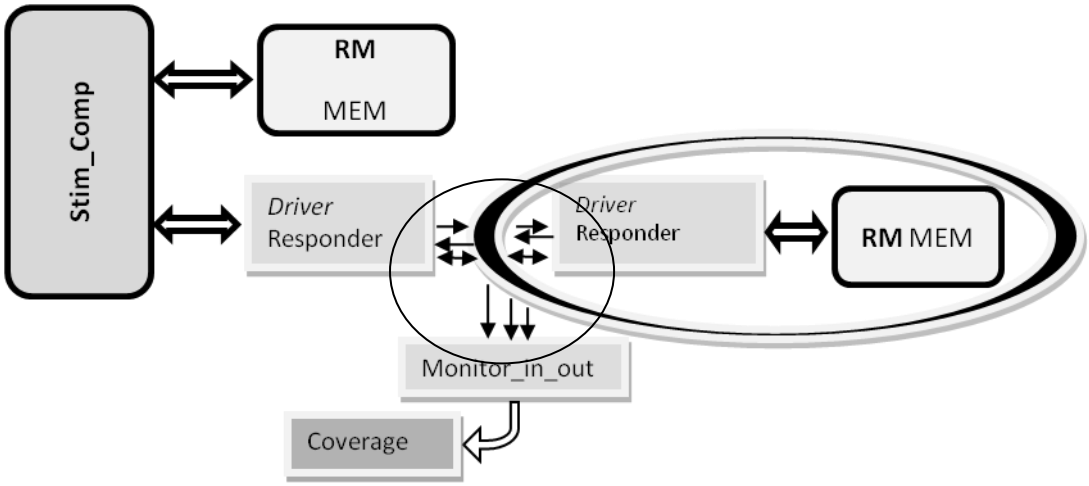


Figura 5.8: Testbench gerado com o processo de automatização

É importante deixar claro que a configuração desenvolvida para a comunicação a nível de sinais do *testbench* através de uma *interface* é uma característica importante que auxilia o reuso dos componentes.

Após implementar todos os ajustes necessários, o *testbench* estará pronto para a verificação do DUV.

5.1.4 Semiautomatização Para a Atividade de Duv_*Execution*

A atividade de Duv_*execution* é a etapa final da criação do ambiente de simulação. Como um dos objetivos é ter o *testbench* criado antes da inserção do DUV, o eTBc irá gerar apenas arquivos básicos para a conexão do ambiente de verificação com a inserção do DUV.

O processo de automatização será de acordo com a figura 5.9 onde recebe os arquivos: RTL, TLN, RM, a biblioteca de *templates* e o script de criação e fará o processo de desenvolvimento do *testbench* completo com a presença do DUV. Um exemplo de arquitetura

de *testbench* gerada nessa etapa pode ser vista na figura 5.10.

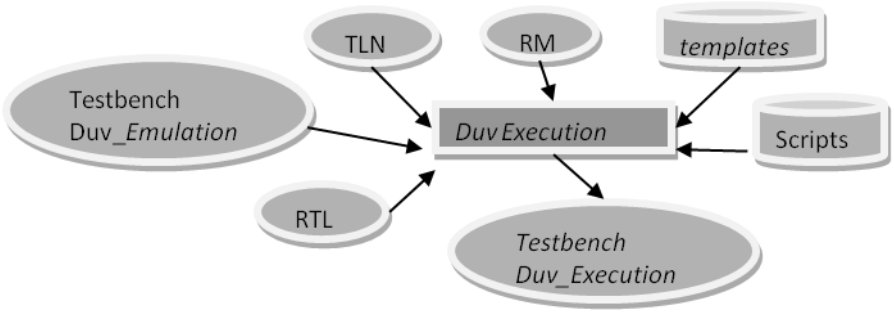


Figura 5.9: Fluxo de geração semiautomática da atividade DUV *execution*

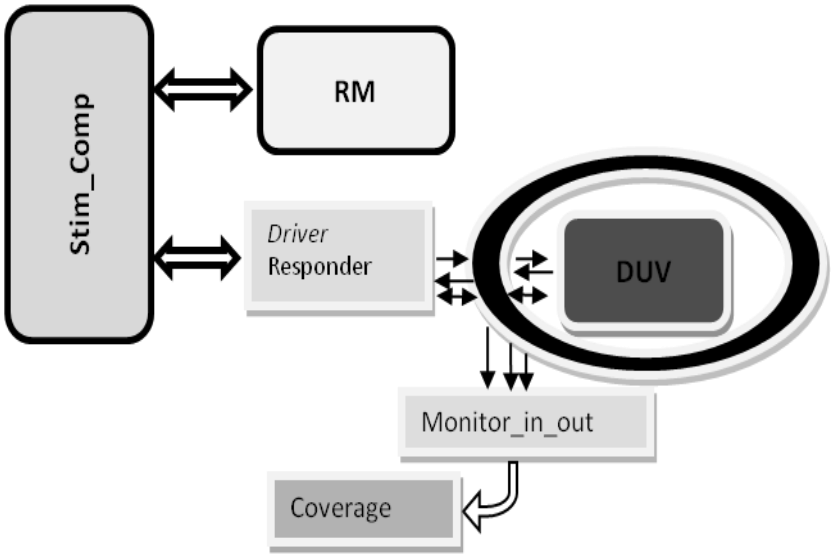


Figura 5.10: Exemplo de *testbench* gerado semiautomaticamente

Podemos ver na figura 5.11 o *script* criado para iniciar esta etapa de geração do ambiente de verificação.

No *script* é mostrado que todos os módulos necessários para a construção do *testbench* são copiados da etapa de Duv*_Emulation.* No Duv*_Execution* serão criados os seguintes módulos básicos para os módulos com comunicação unidirecional: **tb_run_duv, env_duv, top_duv, duv_execution_pkg.**

```
#!/bin/bash

unset ETBC_TLN_DIR
unset ETBC_TMPL_DIR
export ETBC_TLN_DIR=/usr/local/share/etbc/tln
export ETBC_TMPL_DIR=/usr/local/share/etbc/templates/Ovm_tpi
rm -rf ../testbench_RSSI/hierarchical_duv_RSSI

cd ../testbench_RSSI
mkdir hierarchical_duv_RSSI
cd hierarchical_duv_RSSI
ln -s ../../../../../source/SystemVerilog/ESL/RSSI/RSSI_rm.svh RSSI_rm.svh
ln -s ../../../../../source/verilog/RSSI/RSSI.v RSSI.v
ln -s ../hierarchical_double_refmod_RSSI/stimulus_generator.svh stimulus_generator.svh
ln -s ../hierarchical_double_refmod_RSSI/comparator.svh comparator.svh
ln -s ../hierarchical_double_refmod_RSSI/trans.svh trans.svh
ln -s ../hierarchical_double_refmod_RSSI/clk_rst_gen.sv clk_rst_gen.sv
ln -s ../hierarchical_double_refmod_RSSI/clk_rst_if.sv clk_rst_if.sv
ln -s ../hierarchical_double_refmod_RSSI/RSSI.sti RSSI.sti
ln -s ../hierarchical_double_refmod_RSSI/RSSI_refmod.svh RSSI_refmod.svh
ln -s ../hierarchical_double_refmod_RSSI/RSSI_if.sv RSSI_if.sv
ln -s ../hierarchical_duv_emulation_RSSI/pkt_in_RSSI_driver.svh pkt_in_RSSI_driver.svh
ln -s ../hierarchical_duv_emulation_RSSI/pkt_in_RSSI_monitor.svh pkt_in_RSSI_monitor.svh
ln -s ../hierarchical_duv_emulation_RSSI/pkt_out_RSSI_monitor.svh pkt_out_RSSI_monitor.svh
ln -s ../hierarchical_duv_emulation_RSSI/pkt_out_RSSI_responder.svh pkt_out_RSSI_responder.svh
ln -s ../hierarchical_duv_emulation_RSSI/RSSI_coverage.svh RSSI_coverage.svh
etbc landell.tln env_duv RSSI
etbc landell.tln top_duv RSSI
etbc landell.tln duv_execution_pkg RSSI
etbc landell.tln tb run duv execution RSSI
```

Figura 5.11: Script da atividade Duv_Execution

Para comunicação bidirecional devem ser utilizados os seguintes *templates*: **tb_run_duv_bir, env_duv, top_duv, duv_execution_bir_pkg.** Após a execução do *script*, é necessário fazer o link com o modelo de referência criado e com o arquivo RTL descrito.

5.1.5 Máquina de Estados Para o Gerenciamento do Protocolo de Comunicação

Para construir os *templates* utilizados na semiautomatização, foi escolhido inicialmente o protocolo de comunicação HFPB. Após pesquisar outros protocolos de comunicação, constatou-se que a maioria deles se comporta através de uma máquina de estados semelhantes ao HFPB (Amba APB, Amba4 AXI[3,5], OCP-IP, CAN, LIN). Daí surgiu a ideia de criar uma máquina de estados genérica que pudesse ser modificada de acordo com o protocolo de comunicação escolhido.

Dessa forma, foi desenvolvida, nos três blocos responsáveis pela comunicação, em nível de sinais, uma máquina com quatro estados que seria responsável pela comunicação do *testbench* com o DUV. Como visto na figura 5.1, a FSM (*Finite State Machine*) contém três estados: INACTIVE, START, ACTIVE.

Para que o protocolo de comunicação seja confiável, apenas um dos módulos será encarregado de mudar os sinais. Os demais o seguirão, tornando o processo uma comunicação “mestre-escravo”. Como o *driver* é o objeto que envia os sinais para o DUV, ele é o

responsável pela mudança de estado da máquina. O *responder* e o *monitor* trabalham de forma “escrava”, recebendo os dados enviados pelo DUV ou *driver*, e realizando seus objetivos. A figura 5.12 demonstra a máquina utilizando apenas três estados para se comunicar com o DUV.

```
class Driver;

  typedef enum {
    INACTIVE, START, ACTIVE
  } state_e;
  local state_e m_state;
function void start_of_simulation();
  m_state = INACTIVE;
endfunction
// Conceptual state-machine to emulate bus protocol activity
case(m_state)

  INACTIVE : begin
  if(!slave_port.try_get(m_req)) begin
    m_bus_if.sel <= 0;
    m_state = INACTIVE;
    continue;
  end

  ...
  m_bus_if.en <= 0;
  m_bus_if.sel <= 1
  m_state = START;

  ...
  end // INACTIVE
  START : begin

    ...
    m_bus_if.sel <= 1
    m_bus_if.en <= 1;
    m_state = ACTIVE;
  end // START

  ACTIVE : begin
  ...
  m_bus_if.en <= 0;
  m_bus_if.sel <= 0;
  m_state = INACTIVE;
```

Figura 5.12: Máquina de estados do *Driver* se comunicando através do protocolo HFPB

Observa-se que o uso de sinais *Nonblocking* é feito para que os dois sinais recebam novos valores no mesmo instante.

O *responder* e *monitor* são sensíveis a mudança desses sinais, como mostra a figura 5.13.

```
  typedef enum bit [1:0]{ INACTIVE = 2'b00,
                          START    = 2'b10,
                          ACTIVE   = 2'b11
                        } state_t;

  function new( string name, ovm_component parent);
    super.new( name, parent );
    crg = new;
  endfunction
  ---

 task run();

    state_t state;

// begin codeblock monitor_loop
    forever begin
      @( posedge $$(module.name)_if_i.clk );
      ---
      state = state_t'(( ($$(module.name)_if_i.sel != 0), $$(module.name)_if_i.en ));
      ---
```

Figura 5.13: Maquina de Estados do monitor sensível a troca de sinais do *driver*

A linha destacada mostra como é adquirido o estado da máquina através da interface, fazendo uma concatenação dos sinais **sel** e **en**. Assim, tanto o *responder* quanto o *monito*r trabalham de forma “escrava” em relação ao *driver.*

Capítulo 6 Resultados

Neste capítulo são apresentados os principais resultados obtidos com o uso da metodologia OVM_tpi. Todos os resultados experimentais realizados são detalhados, de forma a demonstrar que o trabalho proposto, de fato, atende aos requisitos propostos e colabora construtivamente com o estado da arte na área de verificação funcional.

Foram desenvolvidos dois estudos de caso, com características e complexidades diversas, de modo a explorar da maneira mais abrangente possível os diversos cenários que a metodologia proposta suporta. Os estudos de casos escolhidos foram: DPCM (*Differential Pulse-Code Modulation*) e memória com duas portas (*Dual Port Memory*).

O DPCM foi escolhido por ter sido um estudo de caso utilizado por Silva[44] e por Oliveira[37] para validação de suas metodologias. Dessa forma, será mais simples fazer um estudo comparativo coerente das metodologias envolvidas. Já a memória com duas portas foi escolhida por ter sido utilizada como estudo de caso na validação da metodologia IVM.

6.1 Estudo de Caso: DPCM

Como já visto nos exemplos da metodologia OVM_tpi quando explicadas as atividades, o DPCM é um sistema codificador e decodificador de sinais que utiliza um tipo de modulação PCM (*Pulse Code Modulation*) com a adição de uma base de previsão das amostras de sinais, podendo esse sinal ser analógico ou digital. Caso seja analógico, terá que ser transformado em discreto antes de ser enviado como entrada para o DPCM. Assim, o sistema desenvolvido consiste em calcular a diferença de duas amostras de áudio digital subsequentes, realizando a operação de saturação no resultado dessa diferença. A figura 6.1 mostra a arquitetura do DPCM.

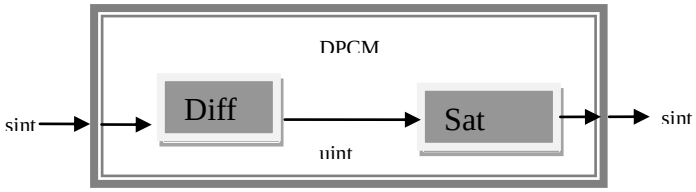


Figura 6.1: Arquitetura do DPCM

O DPCM é um módulo composto de dois submódulos: o diferencial e o saturador. O submódulo denominado de **diferencial** é responsável por calcular a diferença entre o dado recebido e o dado anterior, enquanto o submódulo denominado **saturador** irá, como o próprio nome indica, fazer a saturação do dado diferencial num espaço de valores.

6.1.1 Aplicação da Metodologia OVM_tpi

Após a definição da arquitetura do circuito, o plano de verificação deve ser definido antes de iniciar a aplicação da metodologia OVM_tpi. O plano de verificação do DPCM é a parte mais importante do projeto de verificação funcional, pois ele contém todos os passos que serão realizados no processo de verificação do módulo DPCM.

Para iniciar a execução do plano de verificação seguindo a metodologia OVM_tpi com o fluxo de verificação *top-down*, foi criado um modelo de referência do DPCM na linguagem *SystemVerilog*. Após a descrição do DPCM em *SystemVerilog* funcional, foi necessário a descrição da estrutura do DPCM na linguagem eDL (*eTBc-Design-Language*) para ser utilizado pela ferramenta eTBc na geração dos objetos juntamente com os *templates*. Foi descrito então a TLN (*Transaction Level Netlist*).

O fluxo de desenvolvimento de projeto adotado foi o *top down*. Para seguir este fluxo, antes de iniciar as atividades de construção do *testbench* foi implementado todo o modelo de referência do *Differential Pulse-Code Modulation*. Posteriormente, foi iniciado o processo de desenvolvimento do ambiente de verificação.

6.1.1.1 *Double* Refmod

O passo *Double* Refmod gera um *testbench* que inclui o modelo de referência, o gerador de estímulos e um comparador no nível de transação. Para isso, utilizou-se um *script* específico, desenvolvido no contexto desse trabalho.

Após a execução do script, foram criados todos os arquivos que serão utilizados no primeiro passo da construção do *Testbench*. O objeto denominado **dpcm.sti** será modificado manualmente, inserindo-se os casos de testes diretos que estão descritos no plano de verificação.

Para o estudo de caso, foi feita uma análise comparativa entre a metodologia OVM_tpi e BVM. Foi escolhida a metodologia BVM por ser semelhante à metodologia VeriSC e utilizar SystemVerilog, eTBc e a biblioteca OVM para construção do *testbench*. Dessa forma, as metodologias OVM_tpi e BVM têm o mesmo suporte inicial.

A tabela 6.1 mostra a análise comparativa entre as metodologias OVM_tpi e BVM.

Tabela 6.1: Análise do passo *Double* Refmod para o estudo de caso DPCM

	OVM_tpi	BVM
Número de objetos criados	13	10
Número de linhas Geradas	384	354
Número de Linhas Modificadas	1*	1*
Tempo de desenvolvimento total do passo	2 minutos	5 minutos e 23 segundos**
Percentual de automatização do testbench	99.72%	99.71%

*Inserção do modelo de referência no ambiente de verificação

**Tempo para fazer as atividades de *Single* refmod e *Double* refmod

Da análise da tabela, pode-se perceber, em relação à atividade de validação do gerador de estímulos, modelo de referência e comparador:

- Foram gerados mais módulos na metodologia OVM_tpi do que na BVM, devido ao seu paradigma de linguagem de programação;
- Quantidade de linhas geradas expressivas para ambas as metodologias de verificação funcional;
- Tempo maior no desenvolvimento em BVM, devido à existência da atividade de *Single* refmod anterior ao *Double* Refmod;
- O percentual de automatização é praticamente igual, ou seja, foi necessário adicionar ou modificar pouca coisa;
- O percentual de reuso do testbench criado no passo do *Double* Refmod é maior no BVM, pois já utiliza um módulo gerado na etapa de *Single* Refmod;

Uma vez encerrado o passo do *Double* Refmod, a comunicação do gerador de estímulos com o modelo de referência e do modelo de referência com o comparador pelos dois canais foi validada, de forma que pode ser usada nos próximos passos para a continuação

do *testbench.*

6.1.1.2 Duv *Emulation*

Uma vez concluída a etapa do *Double* Refmod o processo da criação do Duv *Emulation* é iniciado. O principal objetivo dessa etapa é validar o *testbench* final usando uma emulação do DUV.

Após a criação automática dos blocos especificados no Capítulo 5, para esta etapa é preciso fazer mudanças manuais. São necessárias mudanças nos *Driver*s, *responders* e *monitors* quando a transação contém mais de um atributo inserido. Isso ocorre porque a ferramenta utilizada para a automatização não consegue varrer sinais e transações no mesmo *looping,* e então são criadas todas as transações e, posteriormente, todos os sinais. Um exemplo de como é gerado pela ferramenta é mostrado na figura 6.2.

```
/* Here put the transactions data according signal dut */
    alu_if_i.pkt_alu_in29_data_29_bits_s <= m_req.get_data_29_t();
    alu_if_i.pkt_alu_in29_opcode_s <= m_req.get_opcode_t();

/* Here put the transactions data according signal dut */
    alu_if_i.pkt_alu_in29_data_29_bits_s <= alu_if_i.pkt_alu_in29_opcode_s <=
    m_req.get_data_29_t(); m_req.get_opcode_t();
```

Figura 6.2: Exemplo de mudanças manuais que devem ser feitas no Driver

As linhas dois e três do código da figura 6.2 representam o formato que devem obedecer após o concerto manual, enquanto as duas linhas finais da figura mostram como são geradas pela ferramenta eTBc.

Para a etapa do Duv *Emulation* foram gerados os seguintes resultados:

Tabela 6.2: Análise do passo Duv *Emulation* para o estudo de caso DPCM

	OVM_tpi	BVM
Número de objetos criados	11	9
Número de linhas Geradas	415	285

Número de Linhas Modificadas	0*	0*
Percentual de Cobertura de Dados	100%	100% **
Percentual de automatização	100%	Menos de 100%***
Percentual de reuso de componentes	100%	100%

* Não foram incluídos na contagem o módulo Coverage nem as coberturas feitas em diferentes módulos de BVM.

** A cobertura atingiu 100% utilizando o protocolo AMBA AXI para o módulo DPCM.

**O *testbench* teve que ser modificado para se adequar ao protocolo do módulo desenvolvido.

- OVM_tpi gerou mais módulos quando comparado a BVM, o que se deve tanto ao paradigma de linguagem utilizada, quanto ao padrão de descrição dos objetos. Um exemplo do padrão de descrição orientada a objetos é que todos os atributos de uma classe só podem se comunicar com outras através de funções específicas. Na metodologia OVM_tpi, quando se utiliza os *templates* criados, isso é padrão e todos os atributos são obrigados a ter as funções **gets** e **sets.** Já BVM não se preocupa com esse tipo de formalidade, tornando a leitura do código mais complexa e propícia a erros.
- A cobertura gerada por OVM_tpi obteve o valor de 100% em relação ao que foi descrito no módulo Coverage.
- O percentual de automação foi de 100% com a metodologia OVM_tpi, visto que não foi necessário nenhuma mudança nas conexões dos *Drivers, Responders e Monitors*. Como a automação da metodologia BVM foi desenvolvida partindo da premissa que todos os módulos a serem testados seguiam o protocolo AMBA AXI, foram feitas mudanças manuais para adequar o testbench ao design que será testado. Exemplo de mudança foi a criação de um módulo que contém DPCM como instância, tendo a função de passar o sinal do protocolo do *testbench* por fora do módulo DPCM.
- Foram reusados 100% dos componentes da atividade de *Double* Refmod tanto em OVM_tpi quanto em BVM.

Após essa etapa ter sido finalizada, teve início a decomposição do módulo DPCM e a verificação dos seus submódulos.

6.1.1.3 *Hierarchical Decomposition*

A execução desse passo tem uma grande importância face à crescente complexidade dos sistemas digitais. Isso ocorre porque é necessário dividir as funcionalidades do sistema em sub-blocos com o mínimo possível de funcionalidades. Assim, possibilita que a verificação funcional atinja todas as funcionalidade do sistema. O DPCM tem duas funcionalidades: obter o diferencial e fazer a saturação do pulso. Portanto, a decomposição do sistema foi feito em dois blocos denominados **diff** e **sat**. Não foi pesquisado nenhum resultado da decomposição dos módulos, visto que é um processo totalmente manual, em que o foco é apenas o modelo de referência. Desta forma, será iniciado o processo de construção do *Hierarchical Double* Refmod para todos os submódulos do DPCM.

6.1.1.3.1 *Hierarchical Double* Refmod DIFF

Iniciando o processo de verificação funcional para os módulos separadamente, foi construído o ambiente de verificação primeiramente para o bloco diff, responsável por realizar o diferencial. A figura 4.19 mostra o *testbench* desenvolvido para o DIFF na etapa *Hierarchical Double* Refmod. Os resultados obtidos nesta etapa do estudo de caso podem ser vistos na tabela 6.3:

DIFF

Tabela 6.3: Análise do passo *Hierarchical Double* Refmod para o módulo DIFF

	OVM_tpi	BVM
Número de objetos criados	13	10
Número de linhas Geradas	384	354
Número de Linhas Modificadas	1	1
Percentual de automati-	99.72%	99.71%

zação		
Percentual de reuso	0%	33%*

*Foram utilizados módulos da atividade existentes em BVM *single* Refmod.

Todos os resultados obtidos foram iguais aos da atividade de *Double* Refmod, pois segue o mesmo processo de desenvolvimento, apenas mudando o módulo fim. Como o módulo **Diff** tem apenas uma interface de entrada e outra de saída, a construção do *testbench* obteve os mesmos resultados. A única diferença detectada no processo foi na cobertura e descrição do modelo de referência.

Após a finalização do *Hierarchical Double* Refmod para o Diff, continuou em andamento a construção do *testbench* para o módulo responsável pelo cálculo do diferencial. O próximo passo foi a construção do *Hierarchical* Duv_*Emulation*.

6.1.1.3.2 *Hierarchical Duv Emulation* DIFF

Para o módulo diferencial, foi também utilizada a ferramenta eTBc para a geração dos blocos do *Testbench*, e alguns blocos criados no *Hierarchical Double* Refmod foram reusados. Os resultados desse passo estão detalhados na tabela 6.4.

DIFF

Tabela 6.4: Análise do passo *Hierarchical* Duv *Emulation* para o módulo DIFF

	OVM_tpi	BVM
Número de objetos criados	11	9
Número de linhas Geradas	415	285
Número de Linhas Modificadas	0*	0*
Percentual de Cobertura de Dados	100%	100% **
Percentual de automatização	100%	Menos de 100%
Percentual de reuso	100%	100%

* Não foram incluídos na contagem o módulo Coverage nem as coberturas feitas em diferentes

módulos de BVM.

** A cobertura atingiu 100% utilizando o protocolo AMBA AXI para o módulo DIFF.

6.1.1.3.3 *Hierarchical Duv execution* DIFF

A etapa de *Hierarchical* Duv *Execution* é o passo onde inserimos o DUV para a verificação. Após todo o ambiente construído e emulado no passo anterior, foi retirado o bloco de emulação e inserido o DUV. A tabela 6.5 expõe os resultados obtidos nesta etapa:

DIFF

Tabela 6.5: Análise do passo *Hierarchical* Duv para o módulo DIFF

	OVM_tpi	BVM
Número de objetos criados	3	3
Número de linhas Geradas	83	48
Número de Linhas Modificadas	9	5*
Percentual de Cobertura de Dados	100%	100%
Percentual de automatização	89.15%	Menos de 89.5%*
Percentual de reuso dos componentes	100%	100%

* Foram apenas contabilizadas as linhas mudadas dos objetos gerados.

Na tabela acima podemos observar as seguintes peculiaridades:

- Os objetos gerados dizem respeitos apenas às conexões dos *testbenches* desenvolvidos com o DUV do módulo responsável pelo cálculo do diferencial.
- Foi necessário desenvolver uma "casca" para o módulo DIFF, a fim de obter o ambiente especificado no plano de verificação, em que não há nenhum protocolo interior ao módulo.
- As linhas modificadas na metodologia OVM_tpi são necessárias para alterar o nome dos sinais do protocolo do *testbench* (de **en1** e **sel1** para **en** e **sel**), assim como para fazer a conexão do módulo DIFF com a interface de sinais.

- Todos os componentes foram desenvolvidos nas atividades anteriores, e houve reuso de 100% do *testbench*, tornando clara a inserção apenas do DUV no ambiente de verificação desenvolvido anterior.

6.1.1.4 *Hierarchical Double* Refmod SAT

Para o módulo de aturação, o *testbench* foi construído de forma similar ao módulo Diferencial. Os resultados conseguidos no passo *Hierarchical Double* Refmod estão relatados na tabela 6.6:

SAT

Tabela 6.6: Análise do passo *Hierarchical Double* Refmod para o módulo SAT

	OVM_tpi	BVM
Número de objetos criados	13	10
Número de linhas Geradas	384	354
Número de Linhas Modificadas	1	1
Percentual de automatização	99.72%	99.71%
Percentual de reuso	0%	33%*

*Foram utilizados módulos da atividade existente em BVM *single* Refmod.

6.1.1.4.1 *Hierarchical Duv Emulation* SAT

Para a etapa *Hierarchical* Duv *Emulation* foi usado o mesmo processo do módulo Diferencial, e os resultados obtidos foram especificados na tabela 6.7:

SAT

Tabela 6.7: Análise do passo *Hierarchical* Duv *Emulation* para o módulo SAT

	OVM_tpi	BVM
Número de objetos criados	11	9
Número de linhas Geradas	415	285
Número de Linhas Modificadas	0*	0*
Percentual de Cobertura de Dados	100%	100% **
Percentual de automatização	100%	Menos de 100%
Percentual de reuso	100%	100%

* Não foram incluídos na contagem o módulo Coverage nem as coberturas feitas em diferentes módulos de BVM.

** A cobertura atingiu 100% utilizando o protocolo AMBA AXI para o módulo SAT.

6.1.1.4.2 *Hierarchical Duv* SAT

Para verificar o RTL do Saturador foi criado todo o ambiente de simulação no passo anterior. No atual passo, foi retirada a parte do emulador utilizada para a criação do *testbench* e inserido o DUV, obtendo os resultados da tabela 6.8:

SAT

Tabela 6.8: Análise do passo *Hierarchical* Duv para o módulo SAT

	OVM_tpi	BVM
Número de objetos criados	3	3
Número de linhas Geradas	83	48
Número de Linhas Modificadas	9	5*

Percentual de Cobertura de Dados	100%	100%
Percentual de automatização	89.15%	89.5%*
Percentual de reuso dos componentes	100%	100%

* Foram apenas contabilizadas as linhas mudadas dos objetos gerados.

Todas as observações resultantes nesta tabela são semelhante às encontradas na atividade de *Hierarchical Duv* para o módulo DIFF, no item 6.1.1.3.3.

6.1.1.5 Duv Execution

Após a verificação de todos os módulos separadamente, o processo de integração e verificação de todo o sistema foi iniciado. Como o DPCM contém apenas dois módulos, o passo de *integration* DUV não é necessário (note que **n-1 é igual a 1**). Dessa forma, a atividade de DUV *execution* foi desenvolvida.

No passo de DUV *execution* foi feita a integração dos dois módulos pertencentes ao DPCM (DIFF e SAT) e criado um *toplevel* que foi ligado à interface. Posteriormente, o Duv *Emulation* criado na seção 6.1.1.2 foi substituído pelo DUV do sistema completo.

Dessa forma, todo o sistema foi ligado ao *Testbench* para iniciar o processo da verificação funcional do módulo DPCM e os resultados obtidos estão na tabela 6.9.

DPCM

Tabela 6.9: Análise do passo DUV Execution para o módulo DPCM

	OVM_tpi	BVM
Número de objetos criados	3	3
Número de linhas Geradas	83	48
Número de Linhas Modificadas	9	5*
Percentual de Cobertura de Dados	100%	100%

Percentual de automatização	89.15%	89.5%*
Percentual de reuso dos componentes	100%	100%
Tempo de simulação	252,740US	2700US
Tempo de execução	3m33.368s	0m19.405s

* Foram apenas contabilizadas as linhas mudadas dos objetos gerados.

Ao final do estudo de caso do DPCM, temos como resultados:

1. Devido ao uso de um novo paradigma de linguagem e um padrão de desenvolvimento dos módulos que fizeram parte do testbench, foram desenvolvidos mais objetos com suas respectivas funcionalidades. Assim, o ambiente de verificação pode ser desenvolvido com a validação dos componentes do *testbench* e o reuso de cada parte com um nível de 100%.
2. Foi constatado que BVM desenvolve sua automação para módulos que seguem o protocolo de comunicação AMBA AXI, o que gera complicações quando o módulo não recebe nenhum sinal de controle, tendo que ser gerado um módulo "casca" que se comporte semelhante ao funcionamento do *testbench.* Na metodologia OVM_tpi o uso da interface fez com que esse problema não tivesse nenhum impacto na construção do ambiente. Quando o DUV dispuser, em sua especificação, que utilizará o protocolo AMBA_axi, se torna mais viável, uma vez que não precisará de nenhuma mudança no protocolo de controle do código gerado automaticamente. Na metodologia OVM_tpi é sempre necessário fazer as mudanças nas nove linhas do ambiente para poder trocar os sinais de **en1** e **sel1** para **en** e **sel**.
3. A cobertura de dados que foi executada no estudo de caso pode cobrir todas as possibilidades especificadas. Nos módulos desenvolvidos com a metodologia BVM foi constatado que para seguir a metodologia VeriSC foi inserida cobertura de dados e dos sinais em dois locais (Modelo de Referência e *Actor*) do *testbench*. Com a metodologia OVM_tpi, o processo foi simplificado, fazendo apenas a cobertura dos sinais de controle nos monitores utilizando *assertions* e repassando a responsabilidade de cobertura de todos os sinais de entrada e saída de dados do DUV para o módulo.
4. O processo de automatização da criação dos objetos do *testbench* foi utilizado e mostrou-se totalmente eficaz, com resultados de aproveitamento dos códigos gerados

sempre acima de 89% para a metodologia OVM_tpi.

Portanto fica visível que o tempo gasto na verificação funcional utilizando a metodologia OVM_tpi num sistema unidirecional pode ser bastante otimizado com a automatização, simplificando a comunicação entre os módulos e padronizando mais os objetos criados através do uso de interface e empacotamento, além de prover um conjunto de componentes que poderão ser utilizados em outros projetos com mais facilidade, obtendo o reuso total dos componentes em cada etapa da construção do ambiente de simulação final.

No próximo estudo de caso será observado o comportamento da metodologia OVM_tpi na hipótese de comunicação bidirecional. O desenvolvimento do ambiente de verificação funcional se deu através do uso da metodologia OVM_tpi com um fluxo de desenvolvimento *bottom-up*.

6.2 Estudo de Caso: *Dual Port* Memory

Um dos componentes mais usados em sistemas digitais é a memória, que possibilita guardar informações no formato binário para serem usadas quando necessário, por qualquer sistema a qual ela pertença. Um dos estudos de caso desenvolvidos neste trabalho foi o projeto de uma memória com duas interfaces de comunicação, que permite acesso concorrente e paralelo ao seu conteúdo, tanto para operações de leitura como de escrita.

O estudo de caso em análise tem como principais objetivos: a validação da metodologia para projetos de controle, como processadores e memórias; a demonstração da reusabilidade dos componentes de verificação; o alto índice de automatização da metodologia proposta, reduzindo o tempo de desenvolvimento de cada etapa e ilustrando a aplicação do fluxo de atividades proposta a projetos com múltiplos níveis de hierarquia.

A arquitetura da memória aqui utilizada está ilustrada na figura 6.3, de forma a permitir ao leitor uma visão geral das partes que compõe este sistema e de como estas partes estão conectadas.

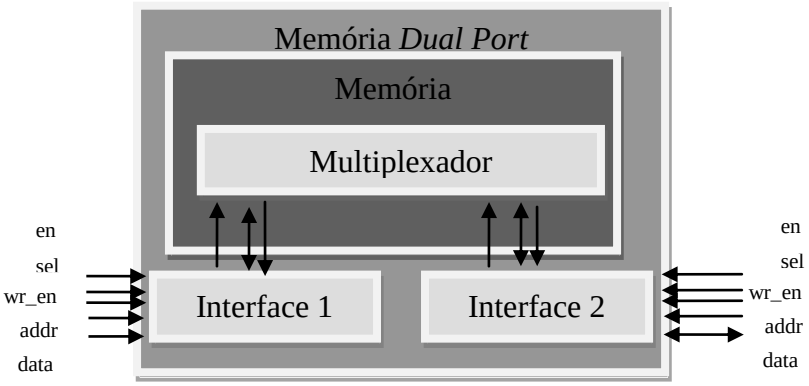


Figura 6.3: Arquitetura da memória *Dual Port*

A memória é composta por três módulos: uma unidade de controle de acesso (Mutiplexador), que previne condições de corrida; duas unidades de interfaces (Interface 1 e 2), para receber e responder as requisições; e a memória (Memória) propriamente dita, que irá armazenar e carregar os dados.

6.2.1 Aplicação da Metodologia OVM_tpi

Para construir o *testbench* aplicando a metodologia OVM_tpi, foram necessários primeiramente a construção de um plano de verificação e o desenvolvimento da arquitetura do sistema. Conforme definido no plano de verificação, o ambiente de simulação será construído através do fluxo de atividades *bottom-up*.

Como existem duas funcionalidades na memória (primeiro definir qual interface tem acesso a memória, e depois comunicar-se com a memória), o sistema foi quebrado em duas partes: Multiplexador e Memória. Para iniciar o processo de construção do *testbench* o primeiro módulo verificado foi o Multiplexador.

6.2.1.1 *Hierarchical* Doubel Refmod Multiplexador

Inicialmente foi descrita a arquitetura da memória *Dual Port* na linguagem eDL para o eTBc, gerando o arquivo TLN necessário para a automatização dos passos. A partir de então, foram utilizados os scripts implementados na linguagem de scripts ***Bash*** para gerar automaticamente todos os módulos necessários para esta etapa. O Multiplexador irá receber requisições de duas interfaces de comunicação e escolherá qual interface terá acesso à memória. O ambiente resultante pode ser visto na figura 6.4.

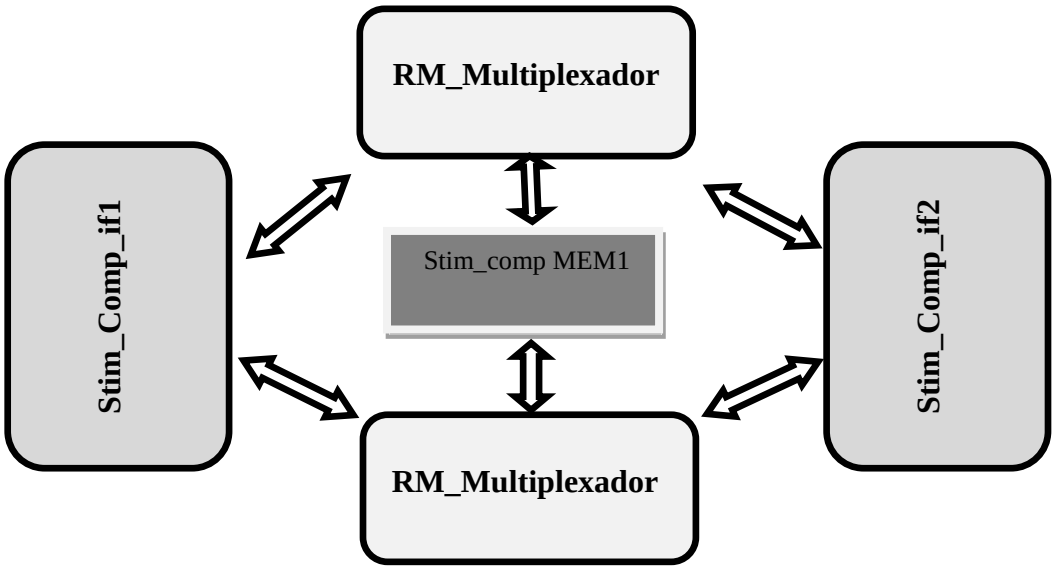


Figura 6.4: *Testbench Hierarchical Double* Refmod para o Multiplexador

A figura 6.4 mostra o modelo de referência já conectado com suas três interfaces bidirecionais em cada **Stim_comp** necessário para a verificação. O objetivo dessa atividade foi validar os geradores e receptores de estímulos, o modelo de referência e suas interfaces de comunicação. Foi então utilizado o script "**double_refmod_bir.sh**" para gerar todos os módulos pertencentes a esse passo. Foram necessários alguns ajustes manuais para que o *testbench* funcionasse de maneira correta.

Em seguida, para a finalização dessa etapa e validação dos componentes foi desenvolvido um estudo comparativo dos resultados obtidos utilizando a metodologia OVM_tpi com os resultados encontrados em Prado[40], que pode ser analisado na tabela 6.10:

Tabela 6.10:Análise do passo *Hierarchical Double* Refmod para o Multiplexador

	OVM_tpi	IVM
Número de objetos criados	8	5
Número de linhas Geradas	486	148
Número de linhas modificadas.	36	148
Percentual de automatização	92.59%	0%
Percentual de reuso	0%*	0%*

*Valor considerando objetos que foram desenvolvidos manualmente.

Fica fácil concluir que, apesar de existirem mais módulos e mais linhas de códigos no ambiente, o processo foi feito automaticamente e necessitou apenas de inserção e/ou modificação de algumas linhas geradas. Dessa forma, tivemos o uso de aproximadamente 90% da geração automática feita para o passo *Hierarchical Double* Refmod.

Uma vez finalizado o processo de validação do gerador e comparador de estímulos, bem como o modelo de referência, tem início o próximo passo.

6.2.1.2 *Hierarchical* Duv *Emulation* Multiplexador

Para que o *testbench* esteja completo e validado, apenas aguardando o DUV, foi desenvolvida essa etapa, onde se emula o DUV e se constrói todo o ambiente. O *testbench* do multiplexador resultante desta atividade pode ser visto na figura 6.5:

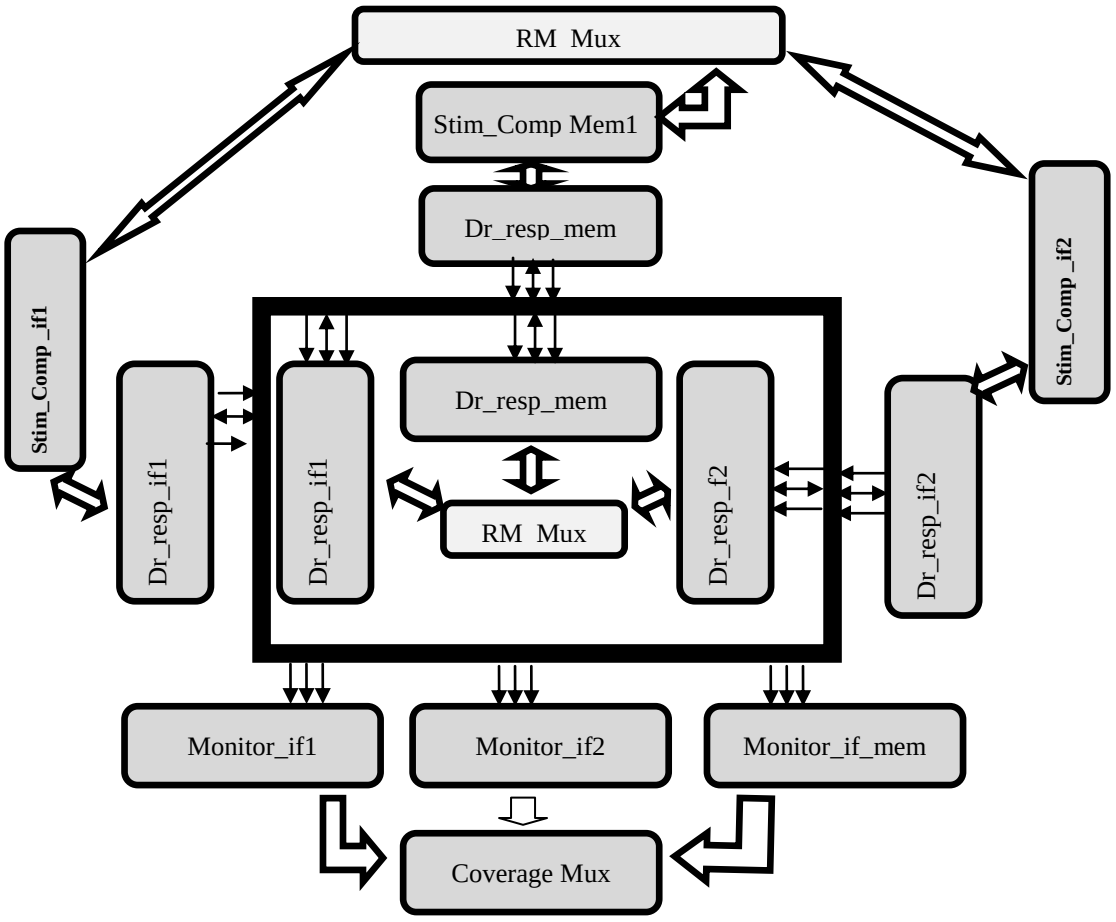


Figura 6.5: *Testbench* desenvolvido do *Hierarchical Duv Emulatiom* para o Multiplexador

De acordo com a figura 6.5 pode-se perceber que temos três estimuladores trabalhando em paralelo, estimulando os modelos de referências. Isso demonstra o suporte do ambiente no desenvolvimento de sistemas de controle, e o alto grau de reusabilidade dos componentes do *testbench*. Os resultados obtidos estão elencados na tabela 6.11, ao lado dos resultados atingidos com o uso da metodologia IVM.

Tabela 6.11: Análise do passo *Hierarchical* Duv *Emulation* para o módulo Multiplexador

	OVM_tpi	IVM
Número de objetos criados	12	4
Número de linhas Geradas	681	127
Número de linhas modificadas.	25**	127
Percentual de automatização	96.32%	0%
Percentual de reuso dos componentes	98.16%	57.96%***

*Não há geração de código

** Não foram contabilizadas as inserções de cobertura no módulo Coverage.

***Valor considerando objetos que foram desenvolvidos manualmente.

A tabela traz um comparativo dos resultados gerados pela atividade de emulação do Multiplexador. Foram gerados doze objetos, pois para cada interface existem um **dr_resp** e um **monitor,** mesmo sendo estes com as mesmas interfaces. Isso ocorre porque o eTBc cria um objeto para cada interface descrita na TLN.

O percentual de reuso foi calculado através da média da soma do reuso dos componentes criados no passo Double_Refmod com o percentual de automação do passo.

6.2.1.3 *Hierarchical* Duv Multiplexador

Finalizado e validado todo o ambiente com a emulação do DUV, é dado início ao processo de inserção do DUV já construído pela equipe de *design* no *testbench*. Para isso, retiramos todo o emulador e inserimos o objeto central da verificação: o DUV. O *testbench* resultante desse processo pode ser visto na figura 6.6.

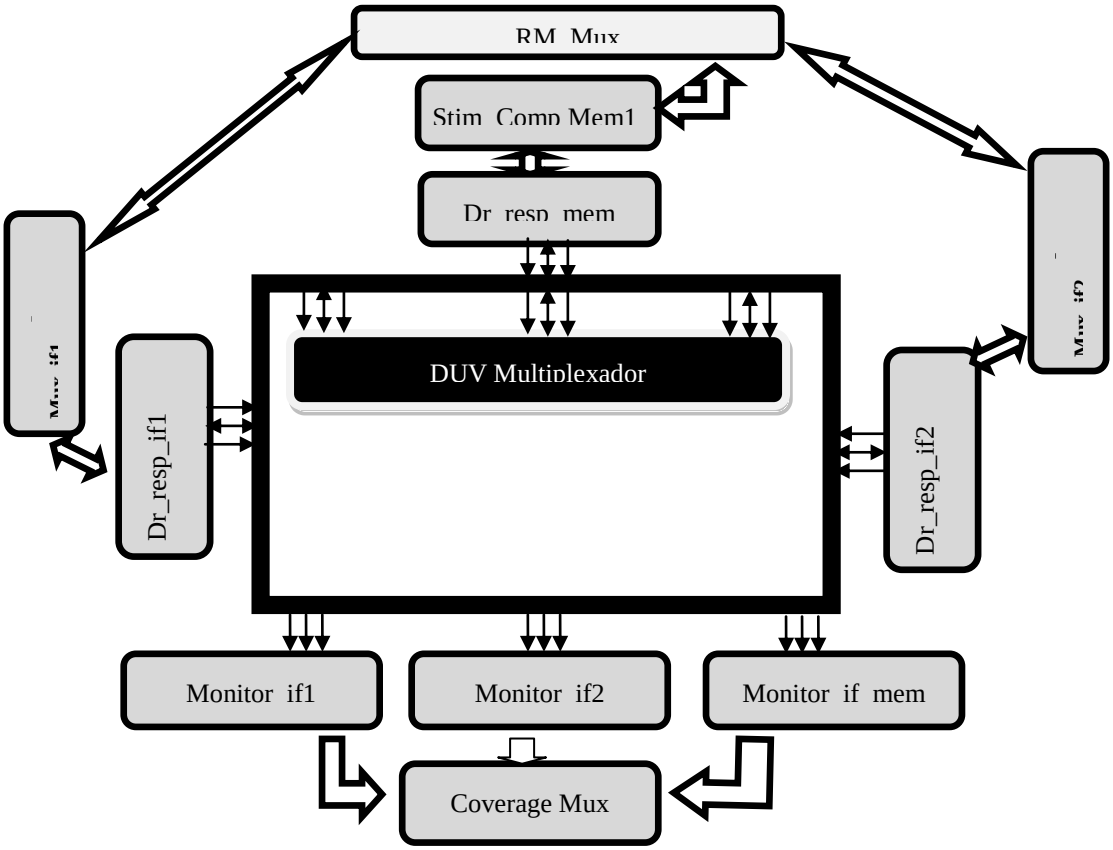


Figura 6.6: Arquitetura do *Hierarchical* Duv para o módulo Multiplexador

Os resultados dessa etapa constam da tabela 6.12:

Tabela 6.12: Análise do passo *Hierarchical* Duv para o módulo Multiplexador

	OVM_tpi	IVM
Número de objetos criados	3	*
Número de linhas Geradas	82	*
Número de linhas modificadas.	1	*
Percentual de automatização	98.78%	0%
Percentual de reuso dos componentes	100%	*
Cobertura Randômica	100%	63.93%**

*Não foram descritos os campos para essa etapa

Todo o processo foi repetido para o módulo Memória. Por este motivo, serão apenas

explicitados o *testbench* e a tabela com os resultados da atividade de *Hierarchical Testbench* para a Memória.

6.2.1.4 *Hierarchical* Doubel Refmod Mémoria

Testbench:

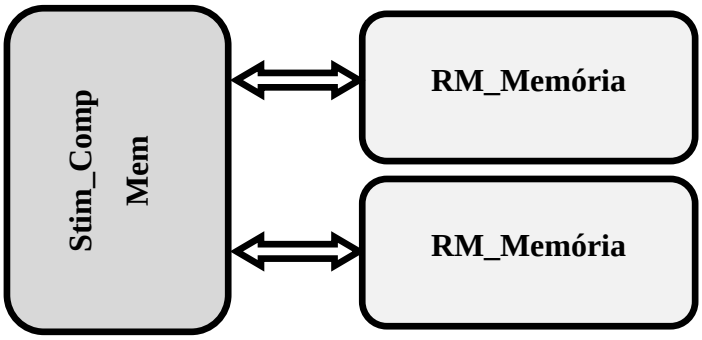


Figura 6.7:Testbench do *Hierarchical Double* Refmod para a Memória

Tabela de resultados:

Tabela 6.13:Análise do passo *Hierarchical Double* Refmod para a Memória

	OVM_tpi	IVM
Número de objetos criados	6	3
Número de linhas Geradas	244	80
Número de linhas modificadas.	13	80
Percentual de automatização	100%	0%
Percentual de reuso	100%	0%

Após a validação dos componentes de geração de estímulos, checagem de resultados, do modelo de referência da memória e suas comunicações, foi iniciada a etapa de criação e validação dos componentes que trabalham com sinais: *drivers, monitors, responders* e *interface*.

6.2.1.5 *Hierarchical* Duv *Emulation* Mémoria

Testbench:

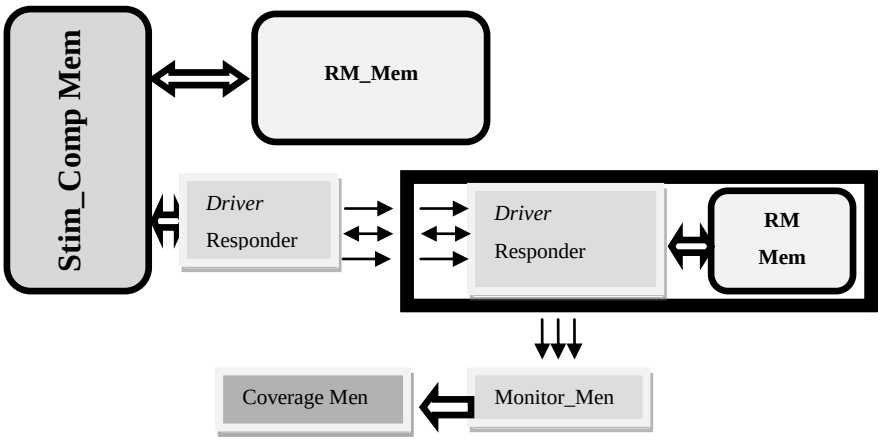


Figura 6.8: Testbench para a etapa de *Hierarchical* Duv *Emulation* para a Mémoria

Tabela de resultados:

Tabela 6.14: Análise do passo *Hierarchical* Duv *Emulation* para o módulo Memória

	OVM_tpi	IVM
Número de objetos criados	8	2
Número de linhas Geradas	259	148
Número de linhas modificadas.	1*	0
Percentual de automatização	99.61%	0%
Percentual de reuso dos componentes	99.80%	54.05%

* Não foram contabilizadas as inserções manuais de cobertura no módulo Coverage.

6.2.1.6 *Hierarchical* Duv Mémoria

Testbench:

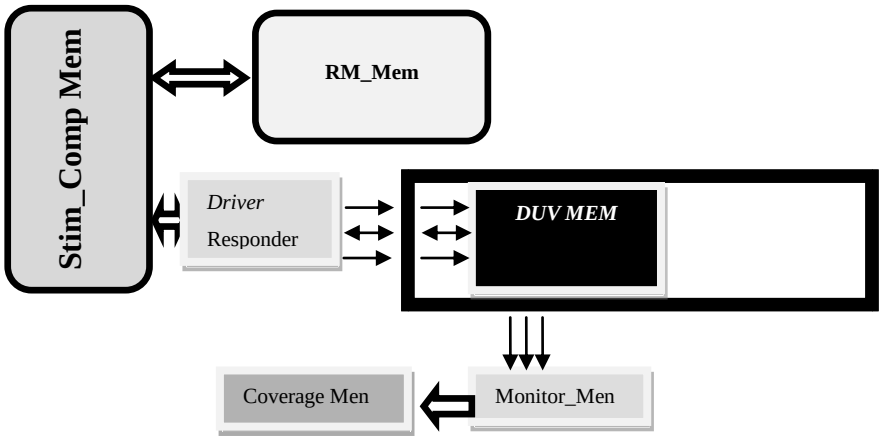


Figura 6.9: Testbench do *Hierarchical* Duv para o módulo Mémoria

Tabela de resultados:

Tabela 6.15: Análise do passo *Hierarchical* Duv para o módulo Mémoria

	OVM_tpi	IVM
Número de objetos criados	3	*
Número de linhas Geradas	75	*
Número de Linhas Modificadas	1	*
Percentual de Cobertura Randômica	100%	72.46%
Percentual de automatização	98.66%	0%
Percentual de reuso	100%	100%

6.2.1.7 *Hierarchical Double* Refmod Interface

Testbench:

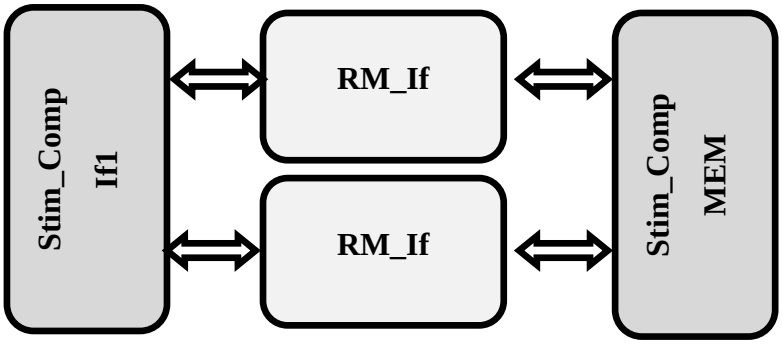


Figura 6.10:Testbench do *Hierarchical Double* Refmod para a Interface

Tabela de resultados:

Tabela 6.16:Análise do passo *Hierarchical Double* Refmod para a Interface

	OVM_tpi	IVM
Número de objetos criados	3	4
Número de linhas Geradas	59	130
Número de linhas modificadas.	1	130
Percentual de automatização	98.30%	0%
Percentual de reuso	98.30%	20%

6.2.1.8 *Hierarchical* Duv *Emulation* Interface

Testbench:

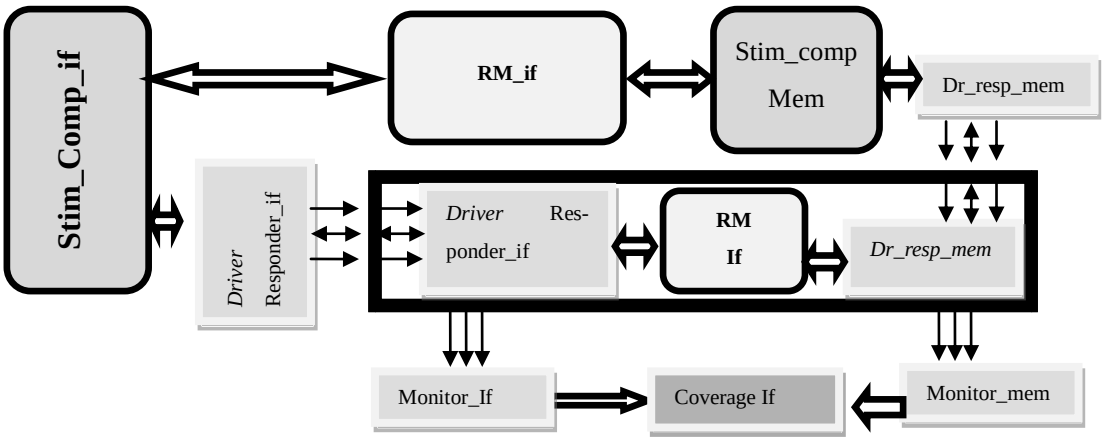


Figura 6.11: Testbench para a etapa de *Hierarchical* Duv *Emulation* para a Interface

Tabela de resultados:

Tabela 6.17: Análise do passo *Hierarchical* Duv *Emulation* para o módulo Interface

	OVM_tpi	IVM
Número de objetos criados	4	2
Número de linhas Geradas	120	148
Número de linhas modificadas.	0	148
Percentual de automatização	100%	0%
Percentual de reuso dos componentes	100%	54.05%

* Não foram contabilizadas as inserções manuais de cobertura no módulo Coverage.

Foram reusados todos os componentes existentes nos *testbenches* anteriores. Porém, foi necessário gerar o env_duv_emulation_bir, o top, o duv_emulator_bir e a interface de

sinais que são responsáveis pela conexão de todo o ambiente. Com a finalização desta etapa, teve início a atividade de *Hierarchical Duv* Interface.

6.2.1.9 *Hierarchical* Duv Interface

Testbench:

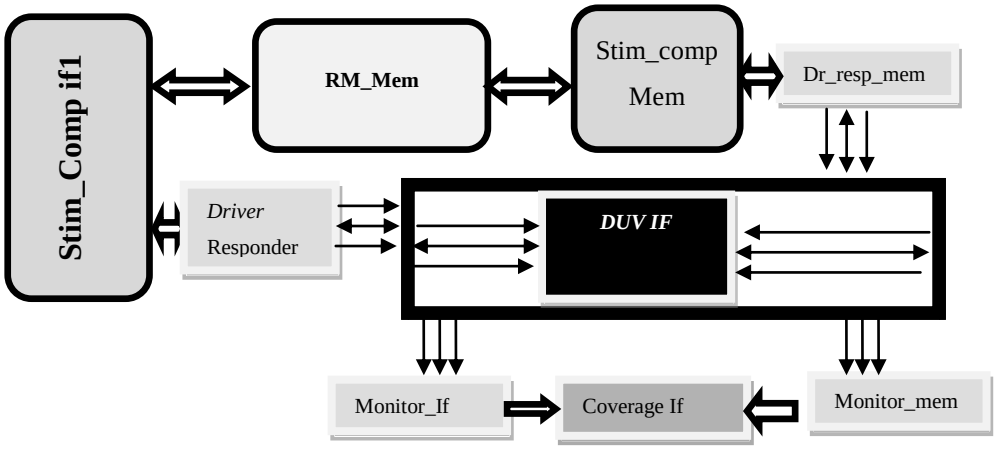


Figura 6.12: Testbench do *Hierarchical* Duv para o módulo Interface

Tabela de resultados:

Tabela 6.18: Análise do passo *Hierarchical* Duv para o módulo Interface

	OVM_tpi	IVM
Número de objetos criados	3	*
Número de linhas Geradas	75	*
Número de Linhas Modificadas	1	*
Percentual de Cobertura Randômica	100%	72.46%
Percentual de automatização	98.66%	0%

Percentual de reuso dos componentes	100%	*

* Não foi encontrado nenhum dado no documento

6.2.1.10 Duv *Integration*

Testbench Multiplexador com Memória:

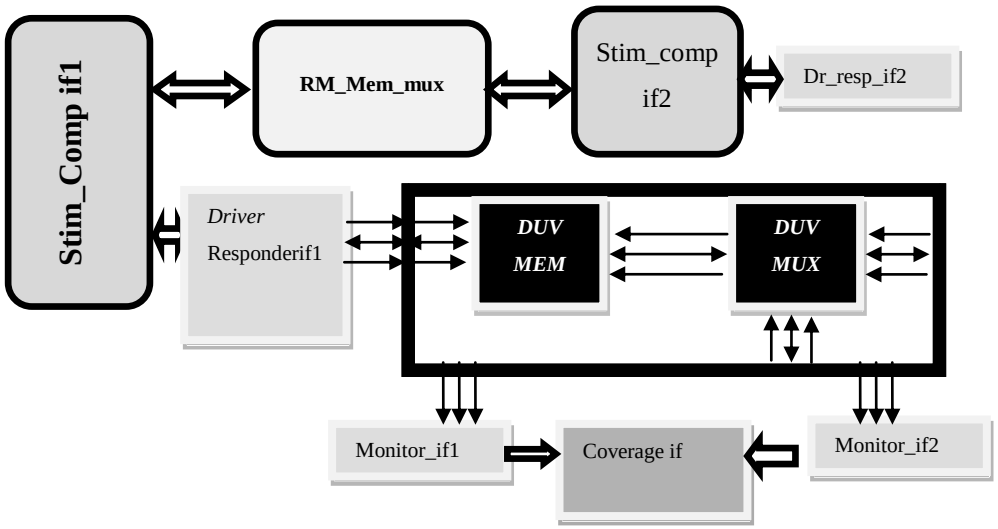


Figura 6.13: Testbench da atividade de Duv Integration da memória com o multiplexador

Tabela de resultados:

Tabela 6.19: Análise do passo *Duv Integration Memória e Multiplexador*

	OVM_tpi	IVM
Número de objetos criados	4	*
Número de linhas Geradas	87	*
Número de Linhas Modificadas	5	*
Percentual de Cobertura Randômica	100%	72.46%
Percentual de automati-	94.25%	0%

zação		
Percentual de reuso	100%	*

* Não foi encontrado nenhum dado no documento

Foi necessária a execução da atividade de Duv *Integration*, pois a quantidade de módulos do estudo de caso era superior a dois. Dessa forma, foram integrados os dois submódulos mais internos do circuito. Todo o ambiente já tinha sido desenvolvido nos passos anteriores, e foram gerados os módulos básicos necessários para a conexão do ambiente: top, env_duv_bir, interface_bir e duv_pgk. As mudanças ocorreram na ligação dos submódulos com a interface de sinais.

A cobertura desenvolvida pela metodologia OVM_tpi é semelhante à utilizada nas atividades anteriores: todos os endereços foram cobertos, tanto na escrita quando na leitura.

6.2.1.11 Duv *Execution*

Após a verificação de todos os módulos separados da Memória *Dual Port*, começou o processo de integração e verificação de todo o sistema. A memória é dividida em quatro submódulos, sendo dois deles semelhantes, de modo que a integração destes foi a última atividade do desenvolvimento do *testbench* para a memória *Dual Port*.

Na fase DUV execution foi desenvolvida a integração das Interfaces com o DUV integrado na etapa anterior. Foi criado um *toplevel* que instancia todos os submódulos do projeto. Posteriormente, todos os módulos criados nas atividades anteriores foram reusados.

Para tornar mais fácil o entendimento desse passo, o *testbench* do Duv Execution para a *Dual Port* pode ser visto na figura 6.14.

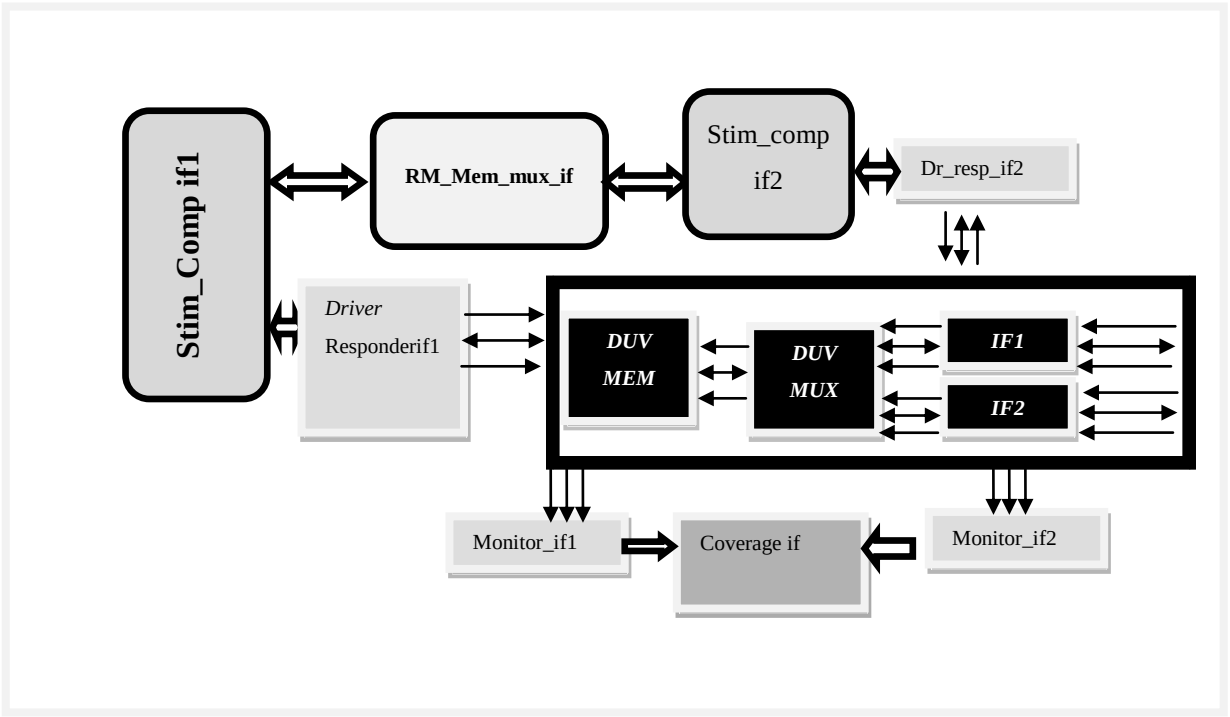


Figura 6.14: *Testbench* completo do estudo de caso Memória *Dual Port*

Na figura 6.14 pode ser visto o *testbench* completo para a memória *Dual Port*, com todo o circuito já inserido no ambiente de simulação. Foram reusados os componentes desenvolvidos nos *testbench* para os módulos hierárquico,s devido às transações terem estruturas semelhantes às transações de uma memória *Dual Port*. O retângulo preto representa a *interface* do ambiente de verificação e no interior está todo o circuito da memória *Dual Port*, que será verificado e coberto de acordo com os valores especificados no plano de verificação.

Tabela de resultados:

Tabela 6.20: Análise do passo *Hierarchical* Duv para o módulo Interface

	OVM_tpi	IVM
Número de objetos criados	4	*
Número de linhas Geradas	87	*
Número de Linhas Mo-	6	*

dificadas		
Percentual de Cobertura Randômica	100%	72.46%
Percentual de automatização	94.25%	0%
Percentual de reuso dos componentes	100%	*

* Não foi encontrado nenhum dado no documento desenvolvido para a metodologia

Foram gerados quatro módulos para que ao ambiente de comunicação entre os componentes fosse o mais próximo possível do ideal. Dessa forma, só foi necessário modificar seis linhas do código, das quais três foram fazem a ligação entre os submódulos, duas ligam os módulos Interface 1 e Interface 2 com a interface de sinais, e uma foi modificada no top para criação de um segundo *clock* para a Interface 2.

Podemos perceber que, de acordo com o estudo de caso desenvolvido acima, o ganho de tempo foi bastante considerável, através do uso da semiautomação provida pela metodologia OVM_tpi. Em todas as atividades o percentual de uso dos módulos criados superou 90%. IVM não tem esse processo de automação e todos os módulos tiveram que ser feitos manualmente. Outras características podem ser observadas:

- O percentual de reuso das atividades de *Double* Refmod levou em conta a semiautomatização utilizada para a criação do *testbench* daquela etapa.
- As modificações que ocorreram durante a criação do ambiente de verificação funcional foram mínimas, porém o engenheiro deve conhecer a área e a forma de criação do *testbench*, para não ter muito trabalho nas modificações.

6.3 Resultados Obtidos

Os estudos de casos realizados permitiram a análise dos resultados gerados tanto para metodologia OVM_tpi, quanto para as metodologias BVM (primeiro estudo de caso) e IVM (segundo estudo de caso). Para a geração dos resultados foram usadas as ferramentas *Incisive Simulation* 9.2 e SystemC 2.2. Foi utilizado o período de *clock* de 10ns para OVM_tpi e para a metodologia BVM, e de 83ns para a Metodologia IVM. Na tabela 6.21 estão inseridos os resultados obtidos nas simulações de cada ambiente de verificação funcional.

Tabela 6.21: Resultados de simulação

	OVM_TPI DPCM	BVM DPCM	OVM_tpi Memória	IVM Memória
Tempo de simulação	212,318 ns	2700 µs	670 µs	30391µs
Tempo de execução	0m23s	2m17s	0m59s	7m32s
Vetores de cobertura	12.846	37.512	11.167	122.056

O processo de execução dos ambientes de simulação, tanto para OVM_tpi, quanto para BVM foi gerenciado. De acordo com os resultados obtidos e mostrados na tabela 6.21, podemos perceber que a metodologia OVM_tpi foi mais rápida que a BVM: o tempo de simulação utilizando OVM_tpi é quase onze vezes menor do que usando a metodologia BVM. Na comparação entre o tempo total de execução do *testbench*, a metodologia OVM_tpi também consegue um ganho de aproximadamente onze vezes em relação a metodologia OVM_tpi. Um dos fatores observado no processo de cobertura de OVM_tpi, é que a metodologia utiliza as funções da biblioteca *Coverage* para desenvolver todo o processo de cobertura. Já a metodologia BVM utiliza um *script* com um algoritmo específico para esse gerenciamento.

Podemos observar na tabela 6.21 que o tempo gasto na execução de toda a simulação, utilizando casos de testes randômicos com a metodologia OVM_tpi, foi menor que o resultado obtido na documentação da metodologia IVM: aproximadamente sete vezes menos tempo. Comparando os tempos de simulação da memória *Dual Port* desenvolvida em ambas as metodologias (IVM e OVM_tpi), o ganho de tempo foi aproximadamente cinquenta vezes maior.

Quando relacionamos os números de vetores de coberturas que foram gerados pelo *testbench* podemos ver que a metodologia OVM_tpi necessitou de menos vetores para produzir o resultado esperado em ambos os projetos. Fazendo uma comparação entre as metodologias OVM_tpi e BVM, podemos perceber que foi necessário apenas um terço dos estímulos para obter a mesma cobertura alcançada pela metodologia BVM com o módulo DPCM.

Na comparação entre as metodologias OVM_tpi e IVM, tendo como base a memória

Dual_port, foi visto que foram necessários 11.167 vetores de teste para obter uma cobertura de 74% do conjunto de valores possíveis. Como visto na documentação da metodologia IVM, ela conseguiu a obtenção de 60.55% dos valores possíveis com uma quantidade de 122.056 vetores de testes. Além disso, a cobertura desenvolvida com a metodologia OVM_tpi foi finalizada apenas com todos os endereços da memória cobertos, tanto para escrita quanto para a leitura.

Tabela 6.22: Análise de Cobertura

	OVM_tpi
DPCM	100%
Dual Port	100%*

* De acordo com o critério de cobertura, porém 74% do conjunto universal dos valores.

Os resultados obtidos na análise de cobertura dos dois estudos de casos foram baseados nas regras descritas pelo engenheiro de verificação. A cobertura desenvolvida não buscou contemplar todo o conjunto de valores probabilísticos de cada circuito, porém buscou cobrir todas as funcionalidades de cada circuito, e obteve valores bastantes representativos, como se pode ver na tabela 6.22.

Capítulo 7 Conclusão

Considerando a importância da verificação funcional, o trabalho desenvolvido teve o propósito de abordar uma metodologia ágil de verificação funcional de circuitos. Pode-se perceber que área de verificação tem desafios bem definidos, com base nas três principais metas de um projeto de circuito integrado: tempo, qualidade e custo.

Na estrutura do trabalho foram introduzidos conceitos essenciais em um projeto de circuito integrado, bem como os principais desafios existentes na área. Foi visto também que a verificação requer um esforço considerável dentro de um projeto de hardware e que, caso um erro seja encontrado em etapas posteriores do projeto, o custo de correção é bastante elevado. Ainda na introdução, foram expostos os objetivos do trabalho, assim como sua contribuição científica em relação à área estudada.

Em seguida, introduzimos conceitos fundamentais, assim como as linguagens mais utilizadas para construção de um *testbench.* Após esta etapa, foram pesquisados trabalhos relacionados e relevantes para o trabalho aqui apresentado, fazendo uma análise comparativa que mostrou os principais locais em que deveria ser focado o estudo.

Após a análise do Estado da Arte, foi desenvolvida uma nova metodologia de verificação funcional, que tem como objetivo principal a geração semiautomática do ambiente de verificação Funcional proposto, com o uso de *Interface* e *assertions*. Foram explicadas e exemplificadas todas as atividades necessárias para a criação do *testbench* e a validação de seus componentes. Após a teorização da metodologia, foi detalhado todo o processo de automação que OVM_tpi suporta, diminuindo o tempo de construção do ambiente de verificação funcional através do uso da biblioteca de *templates* e *scripts* definida.

Para evidenciar o cumprimento das metas traçadas pelo trabalho, foram desenvolvidos dois estudos de casos, que permitem ao leitor constatá-lo. Fica nítido o incremento no estado da arte da verificação funcional, através do uso de um paradigma de linguagem orientação-objeto, com arquitetura bem definida, novas técnicas de cobertura de controle, geração de códigos automática e nova forma de comunicação do *testbench*.

7.1 Contribuições

O resultado do estudo das metodologias citadas no estado da arte mostrou que cada uma delas tinha seus pontos positivos, porém com suas omissões peculiares. Foi focando

neste ponto que toda a pesquisa desenvolvida criou uma metodologia, com o uso da biblioteca OVM, para construção de um *testbench* com as seguintes contribuições para a área de verificação:

- Suportar o desenvolvimento do Ambiente de verificação funcional sem a necessidade do DUV;
- Suportar a construção de um ambiente de verificação funcional seguindo ou o fluxo *top-down* ou *bottom-up;*
- Prover suporte a comunicação bidirecional e unidirecional;
- Usar *templates* para a criação semiautomática do *testbench*, obtendo um ganho de tempo considerável;
- Prover um fluxo de construção *self-checking* dos componentes criados em cada atividade do processo;
- Utilizar A*ssertions* para fazer a cobertura do protocolo de comunicação;
- Utilizar o processo de cobertura (*coverage*) como condição de parada, sendo o critério especificado no plano de verificação;
- Componentes projetados, padronizados e estruturados em camadas (Classes) e empacotados (Package), de forma que seu reuso em outros projetos possa ser assegurado;
- Arquitetura do *testbench* com a interface de comunicação padronizada.

Para esclarecer mais as contribuições aqui desenvolvidas, foi construída uma tabela comparativa com todas as metodologias estudadas, acrescentando-se nessa comparação a metodologia OVM_tpi. Na tabela 7.1 podemos perceber que os objetivos especificados pelo trabalho foram desenvolvidos focando nos principais desafios encontrados na área de verificação funcional.

Tabela 7.1: Análise comparativa das metodologias estudadas

	VeriSC	**BVM**	**OVM**	**IVM**	**OVM_tpi**
Fluxo de desenvolvimento	*Top Down*	*Top Down*	*Bottom up*	*Bottom Up*	*Top down ou bottom up*
Ambiente de verificação executando Antes do DUV	Suporta	Suporta	Suporte indireto	Suporta	Suporta
Suporte a várias linguagens	Não	Sim	Sim	Não	Sim
Geração de Estímulos	Centralizada	Centralizada	Descentralizada	Descentralizada	Unidirecional: Único gerador e checador. Bidirecional: Em camadas
Cobertura funcional	Biblioteca própria	Biblioteca de *SystemVerilog*	Biblioteca de *SystemVerilog*	Biblioteca própria	Biblioteca de *SystemVerilog*
Linguagem	*SystemC*	*SystemVerilog*	*SystemVerilog*	*SystemC*	*SystemVerilog*
Assertions	Não suporta	Suporta	Suporta	Não suporta	Tem suporte e deve ser utilizada
Mecanismo de geração de *test-bench*	Sim	Sim	Não	Não	Sim
Comunicação bidirecional	Não suporta	Não suporta	Suporta	Suporta	Suporta

7.2 Trabalhos Futuros

No decorrer desse trabalho foram observadas algumas formas de incrementar a

pesquisa desenvolvida. Tais observações foram listadas como trabalho futuro, e podem trazer excelentes contribuições tanto para esta metodologia específica, quanto para a área de verificação como um todo. Entre essas contribuições podemos citar:

- Gerador de documentação semelhante ao javadoc;
- Criação de *template*s para outros protocolos de comunicação;
- Criação de templates que suportam a interoperabilidade entre linguagens e metodologias (*SystemC*, VMM);
- Geração do arquivo eDL a partir de uma descrição UML;
- Criação de uma interface gráfica para construção do *testbench*, utilizando a metodologia OVM_tpi;
- Criação de ferramenta que possibilite rodar simulações em GRID;
- Implementação de uma forma de geração automática de espaços de cobertura em tempo de execução.

Por fim, podemos observar que o desenvolvimento de qualquer desses trabalhos supracitados permitirão incrementos significativos na produtividade, na diminuição dos custos, na qualidade da verificação ou na interoperabilidade entre linguagens e metodologias.

Capítulo 8 Referências

[1] ACCELLERA. Disponível em: <http://www.accellera.org>. Acesso em: 20 de nov. 2009.

[2] ACQUAVIVA, A.; BOMBIERI, N.; FUMMI, F.; S. Automatic customization of device drivers for IP-cores used with assorted CPU organizations. **Proceedings of the 7th**, p. 173-182. Retrieved from http://portal.acm.org/citation.cfm?id=1629460&dl=ACM, 2009.

[3]ARM. AMBA AXI Protocol Specification. Disponível em: <http://infocenter.arm.com/help/index.jsp., 2010.

[4]ARM. AMBA APB Protocol Specification. Disponível em: <http://infocenter.arm.com/help/index.jsp>, 2010.

[5]ARM. AMBA 4 AXI4-Stream Protocol. Disponível em: <http://infocenter.arm.com/help/index.jsp>, 2010.

[6]BOMBIERI, N.; DEGANELLO, N.; FUMMI, F. **Integrating RTL IPs into TLM Designs Through Automatic Transactor Generation.** 2008 Design, Automation and Test in Europe, p. 15-20. Ieee. doi: 10.1109/DATE.2008.4484653, 2008.

[7]BOMBIERI, N.; FUMMI, F.; QUAGLIA, D. **System/network design-space exploration based on TLM for networked embedded systems.** ACM Transactions on Embedded Computing Systems, v. 9, n. 4, p. 1-32. doi: 10.1145/1721695.1721703, 2010.

[8]BERGERON, J. **Writing Testbenches using System VerilogWriting**. p.411. Boston, MA: Springer US. doi: 10.1007/0-387-31275-7, 2006.

[9]BERGERON, J.; CERNY, E.; HUNTER, A.; NIGHTINGALE, A. **Verification Methodology Manual for *SystemVerilog***. p.504. Springer, 2005.

[10]BHUTADA, S. **A Scalable Approach for TLM across *SystemC* and *SystemVerilog*.** 2009.

[11]CADENCE DESIGN SYSTEMS; METOR GRAPHICS. **Open Verification Methodology User Guide**. Version 2.1.1. USA, 2010

[12]CADENCE DESIGN SYSTEMS; METOR GRAPHICS. **Open Verification Methodology Class Reference**. Version 2.1.1. USA, 2010

[13]CADENCE DESIGN SYSTEMS. **Cadence Incisive Enterprise Simulator**, 2010. Disponível em: <http://www.cadence.com/products/sd/enterprise_simulator/pages/default.aspx>. Acesso em: mai.2010e.

[14]CHOCKLER, H.; PURANDARE, M. **Coverage in Interpolation-based Model Checking**. Annual ACM IEEE Design Automation Conference, p. 182-187. Retrieved from http://doi.acm.org/10.1145/1837274.1837320, 2010.

[15]DAS, D.; CHAKRABARTI, P. P.; KUMAR, R. **Scenario-based timing verification of multiprocessor embedded applications.** ACM Transactions on Design Automation of Electronic Systems, v. 14, n. 3, p. 1-58. doi: 10.1145/1529255.1529259, 2009.

[16]DAS, D.; CHAKRABARTI, P. P.; KUMAR, R. **Thermal analysis of multiprocessor SoC applications by simulation and verification.** ACM Transactions on Design Automation of Electronic Systems, v. 15, n. 2, p. 1-52. doi: 10.1145/1698759.1698765, 2010.

[17]DUENAS, C. A. M. **Verification and test challenges in SoC designs**. Proceedings of the 17th symposium on Integrated circuits and system design - SBCCI ’04, p. 9-9. New York, New York, USA: ACM Press. doi: 10.1145/1016568.1016573, 2004.

[18]FERRANDI, F.; RENDINE, M.; SCIUTO, D. **A Scalable Approach for TLM across *SystemC* and *SystemVerilog*** Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition, p. 744-751. IEEE Comput. Soc. doi: 10.1109/DATE.2002.998382, 2002.

[19]FINE, S.; UR, S.; ZIV, A. **Probabilistic regression suites for functional verification**. Proceedings of the 41st annual conference on Design automation - DAC ’04, p. 49. New York, New York, USA: ACM Press. doi: 10.1145/996566.996581, 2004.

[20]FINE, S.; ZIV, A. **Coverage directed test generation for functional verification using bayesian networks.** Proceedings of the 40th conference on Design automation - DAC ’03, p. 286. New York, New York, USA: ACM Press. doi: 10.1145/775832.775907, 2003.

[21]GHOSH, I.; RAVI, S. **On automatic generation of RTL validation test benches using circuit testing techniques.** Proceedings of the 13th ACM Great Lakes Symposium on VLSI - GLSVLSI ’03, p. 289. New York, New York, USA: ACM Press. doi: 10.1145/764808.764884, 2003.

[22]GLASSER, M. **Open Verification Methodology CookbookMedia**. 1st ed., p.235. Springer, 2009.

[23]HENFTLING, R.; ZINN, A; BAUER, M.; ECKER, W.; ZAMBALDI, M. **A Scalable Approach for TLM across *SystemC* and *SystemVerilog*** 2003 Design, Automation and Test in Europe Conference and Exhibition, p. 1038-1043. IEEE Comput. Soc. doi: 10.1109/DATE.2003.1253741, 2003.

[24]HENFTLING, R.; ZINN, A; BAUER, M.; ZAMBALDI, M.; ECKER, W. **Re-use-centric architecture for a fully accelerated testbench environment.** Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451), p. 372-375. Ieee. doi: 10.1109/DAC.2003.1219027, 2003.

[25]HIERONS, R. M.; KRAUSE, P.; LÜTTGEN, G., et al. **Using formal specifications to support testing**. ACM Computing Surveys, v. 41, n. 2, p. 1-76. doi: 10.1145/1459352.1459354, 2009.

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

[27]IEEE STANDARDS; IEEE 1666-2005 **Standard *SystemC* Language Reference Manual**, USA 2005.

[28]IMAN, S. **Step-by-Step Funcional Verification with *SystemVerilog* and OVM.** p.500. 1st ed. Santa Clara, CA. Springer,2008.

[29]JENIHHIN, M.; RAIK, J.; CHEPUROV, A.; UBAR, R. **PSL Assertion Checking Using Temporally Extended High-Level Decision Diagrams.** Journal of Electronic Testing, v. 25, n. 6, p. 289-300. doi: 10.1007/s10836-009-5116-4, 2009.

[30]JERINIC, V.; LANGER, J.; HEINKEL, U.; MULLER, D. **New Methods and Coverage Metrics for Functional Verification.** Proceedings of the Design Automation & Test in Europe Conference, p. 1-6. Ieee. doi: 10.1109/DATE.2006.243901, 2006.

[31]JIANG, Z. **Automated analysis of load testing results.** symposium on Software testing and analysis, p. 143-146. Retrieved from http://portal.acm.org/citation.cfm?id=1831726, 2010.

[32]KEATING, M.; BRICAUD, P. **Reuse Methodology Manual For System-on-a-Chip Design**. p.291. Kluwer Academic Publishers, 2002.

[33]MAKE TOOL. **Make Makefile Automation Tool**, 2010. Disponível em:<http://www.nondot.org/sabre/Mirrored/GNUMake/make_3.html>. Acesso em: agosto. 2010.

[34]MENTOR GRAPHICS INC. **Mentor Graphics**, 2010. Disponível em: <http://www.mentor.com/>.Acesso em: julho. 2010a.

[35]MAIA, I.; SILVA, KARINA R G; MAX, L.; CAMARA, R.; MELCHER, E. U. K. **eTBc: A Semi-Automatic Testbench Generation Tool.** IP Based SoC Design Conference & Exhibition, p. 1-5, 2007.

[36]MINTZ, M.; EKENDAHL, R. **Hardware Verification with *SystemVerilog***. 1st ed., p.313. Springer, 2007.

[37]OLIVEIRA, H. F. D. A. **Reformulação, baseada em OVM, da metodologia de verificação funcional VeriSC**, 2010.

[38]PESSOA, I. M. **Geração semiautomática de Testbenches para Circuitos Integrados Digitais**, 2007.

[39]PIZIALI, A. **Functional Verification Coverage Measurement and analysis**. 1st ed., p.213. Kluwer Academic Publishers, 2004.

[40]PRADO, B. O. P. **IVM : Uma Metodologia de verificação Funcional Interoperável , Iterativa e Incremental**, 2009.

[41]PRADO, B.; BARROS, E.; SILVA, L., et al. IVM : **An Interoperable Verification Methodology for Iterative and Incremental Digital System Design**. VLSISOC, 2009.

[43]RODRIGUES, C. L.; SILVA, KARINA R. G. DA; CUNHA, H. N. **Improving functional verification of embedded systems using hierarchical composition and set theory.** Proceedings of the 2009 ACM symposium on Applied Computing - SAC '09, p. 1632. New York, New York, USA: ACM Press. doi: 10.1145/1529282.1529650, 2009.

[44]SILVA, KARINA R G. **Uma Metodologia de verificação Funcional para Circuitos DigitaisMemory**, 2007.

[45]SPEAR, C. ***SystemVerilog* for VerificationÇ A Guide to Learning the Testbench Language FeaturesControl**. 1st ed., p.301. Springer, 2006.

[46] ABADIR, M. S.; WANG, L.-C. The verification and test of complex digital ICs [Guest Editor's Introduction]. **IEEE Design & Test of Computers**, v. 21, n. 2, p. 80-82. doi: 10.1109/MDT.2004.1277899, 2004.

[47]BOMBIERI, N.; DEGANELLO, N.; FUMMI, F. Integrating RTL IPs into TLM Designs Through Automatic Transactor Generation. **2008 Design, Automation and Test in Europe**, p. 15-20. Ieee. doi: 10.1109/DATE.2008.4484653, 2008.

[48]GAJSKI, D. D.; ABDI, S.; GERSTLAUER, A.; SCHIRNER, G. **Embedded System DesignMedia**. First ed., p.352. Springer Dordrecht Heidelberg London New York. doi: 10.1007/978-1-4419-0504-8, 2009.

[49]GHOSH, I.; MUKHERJEE, R.; PRASAD, M.; FUJITA, M. Tutorial 3 High Level Design Validation : Current Practices and Future Directions. **Vlsi Design**, p. 3-5. doi: http://doi.ieeecomputersociety.org/10.1109/ICVD.2004.1260892, 2004.

[50]HABIBI, A.; TAHAR, S.; SAMARAH, A.; MOHAMED, O. A. Efficient Assertion Based Verification using TLM. **Proceedings of the Design Automation & Test in Europe Conference**, p. 1-6. IEEE. doi: 10.1109/DATE.2006.244005, 2006.

[51]Heaton, N. Maximizing Verification Effectiveness Using Metric-Driven Verification. **CADENCE DESIGN SYSTEMS, INC.**, White Paper, 2010.

[52]UOL, Routers. Disponível em: <http://tecnologia.uol.com.br/ultimas-noticias/reuters/2011/01/31/intel-encontra-falha-em-chip-corta-previsao-de-receita.jhtm>. Acesso em: 02 de fevereiro de 2011.

[53] EETIMES. Disponível em: <http://www.eetimes.com/story/OEG20010604S0113>. Acesso em: 10 de nov. 2009.

[54]VMM Central. Disponível em: <http://www.vmmcentral.org/systemverilog.html>. Acesso em: 02 de fevereiro de 2011.

[55]SystemVerilog ASIC. Disponível em: <http://system-verilog-asic-design.blogspot.com/>. Acesso em: 26 de outubro de 2010.

[56]ACCELLERA. **Universal Verification Methodology**. Disponível em:< http://www.uvmworld.org/ >. Último acesso: 27 de fevereiro de 2011.

[57] ACCELLERA. **Universal Verification Methodology Reference**. Version 1.0. USA, 2011.

[58] ACCELLERA. **Universal Verification Methodology User Guide**. Version 1.0. USA, 2011.

Apêndice A: *Templates* Desenvolvidos Para a Metodologia OVM_tpi

Clk_rst_gen

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("clk_rst_gen.sv")
module clk_rst_gen( interface i );
  //set time unit and timeprecision
  timeunit       1ns ;
```

```
timeprecision 100ps ;
//change if RESET is active in low.
parameter bit ACTIVE_RESET = 1;
parameter bit NOT_ACTIVE_RESET = 0;
bit stop;

initial begin
  stop = 0;
end

task run();
  static int reset_hold = 2;
  static int half_period = 5;
  static int count = 0;
  i.clk = 0;
  i.rst = NOT_ACTIVE_RESET;

  for( int i = 0; i < reset_hold;i++ ) begin
    tick( half_period );
  end

  i.rst = ACTIVE_RESET;

  for( int i = 0; i < 2*reset_hold;i++ ) begin
    tick( half_period );
  end

  i.rst = NOT_ACTIVE_RESET;

  for( int i = 0; (i < count || count == 0) && !stop; i++ ) begin
    tick( half_period );
  end
endtask

task tick( int half_period );
  # half_period;
  i.clk = !i.clk;
endtask
```

```
endmodule
```

Clk_rst_if

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short-> shortint unsigned )
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("clk_rst_if.sv")
interface clk_rst_if;
 //set time unit and timeprecision
  timeunit       1ns ;
  timeprecision 100ps ;

  bit clk;
  bit rst;
```

```
endinterface

$$(endfile)
```

Comparator

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("comparator.svh")
class comparator extends ovm_threaded_component;

  $$(foreach)  $$(module.out)
  ovm_get_port #(pkt_$$(i.type)) $$(i.name)_from_rem;
  ovm_get_port #(pkt_$$(i.type)) $$(i.name)_from_duv;
  $$(endfor)
```

```
int unsigned number_Of_Error;

function new(string name, ovm_component parent);
  super.new(name,parent);
  number_Of_Error = 0;

endfunction
function void build();
$$(foreach)  $$(module.out)
  $$(i.name)_from_rem = new("pkt_$$(i.name)_from_rem", this);
  $$(i.name)_from_duv = new("pkt_$$(i.name)_from_duv", this);
$$(endfor)
endfunction

function int getNumberOfError();
  return number_Of_Error;
endfunction

function void setNumberOfError();
  this.number_Of_Error++;
endfunction

task run();
  $$(foreach)  $$(module.out)
  pkt_$$(i.type) tr_duv_$$(i.name),tr_rem_$$(i.name);
  $$(endfor)
  string msg;

  forever begin
    $$(foreach)  $$(module.out)
    $$(i.name)_from_duv.get(tr_duv_$$(i.name));
    $$(i.name)_from_rem.get(tr_rem_$$(i.name));
    if(!tr_duv_$$(i.name).comp(tr_rem_$$(i.name))) begin
      msg = $psprintf("received:  %s  expected  %s",
        tr_duv_$$(i.name).psprint(), tr_rem_$$(i.name).psprint() );
      ovm_report_error("Comparator", msg);
      setNumberOfError();
      assert(record_error_tr("Comparator"));
      if (getNumberOfError() > 5) begin
```

```
          #100ns; //Stop the simulation 100ns after error
          global_stop_request();
        end
      end
    /* else begin
      msg = $psprintf("received:  %s  expected  %s",
          tr_duv_$$(i.name).psprint(), tr_rem_$$(i.name).psprint() );
        ovm_report_info("Comparator", msg);
      end*/
      $$(endfor)
    end
  endtask
endclass

$$(endfile)
```

Coverage

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
```

```
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("$$(module.name)_coverage.svh")class $$(module.name)_coverage extends ovm_component;
  typedef virtual $$(module.name)_if.coverage_$$(module.name) $$(module.name)_if_t;
  $$(module.name)_if_t $$(module.name)_if_i;$$(foreach)$$(module.in)
  ovm_blocking_get_port #(pkt_$$(i.type)) $$(i.name)_get_analysis_port;
  local pkt_$$(i.type) $$(i.name)_trans;$$(endfor)$$(foreach)$$(module.out)
  ovm_blocking_get_port #(pkt_$$(i.type)) $$(i.name)_get_analysis_port;
  local pkt_$$(i.type) $$(i.name)_trans;
$$(endfor)
  typedef enum bit [1:0]{ INACTIVE = 2'b00,
                  START    = 2'b10,
                  ACTIVE   = 2'b11,
                  ERROR    = 2'b01
                  } state_t;


  //data coverage
  //OLHAR DIREITO QUE O ETBC NAO CONSEGUE FAZER FOREACH MODULE.IN COM VAR
$$(foreach)$$(module.in)
   covergroup $$(module.name)_in;
     $$(foreach)$$(var)
     coverpoint $$(j.name)_trans.get_$$(i.name)() {
     bins tr_out0 = {0};// o bin devera aparecer 2 vezes
     option.at_least = 2;
     }
    $$(endfor) $$(endfor)

  endgroup
$$(foreach)$$(module.out)
   covergroup $$(module.name)_out;
     $$(foreach)$$(var)
     coverpoint $$(j.name)_trans.get_$$(i.name)() {
     bins tr_out0 = {0};// o bin devera aparecer 2 vezes
     option.at_least = 2;
   }
```

```
  $$(endfor) $$(endfor)

 endgroup

 function new( string name, ovm_component parent);
  super.new( name, parent );
  $$(module.name)_in = new;
  $$(module.name)_out = new;
$$(foreach)$$(module.in)   $$(i.name)_trans = new();$$(endfor)$$(foreach)
$$(module.out)   $$(i.name)_trans = new();$$(endfor)

 endfunction
 function void build();
$$(foreach)$$(module.in)   $$(i.name)_get_analysis_port = new("$$(i.name)_get_analysis_port", this);
$$(endfor)$$(foreach)$$(module.out)                    $$(i.name)_get_analysis_port          =
new("$$(i.name)_get_analysis_port", this);
 endfunction
 function void set_vif(virtual $$(module.name)_if vif);
  $$(module.name)_if_i = vif;
 endfunction
  task run();
   state_t state;
      forever begin
           @( negedge dpcm_if_i.clk );
        while(dpcm_if_i.rst) @(negedge dpcm_if_i.clk);
        state = state_t'({dpcm_if_i.sel, dpcm_if_i.en});
        case( state )
          INACTIVE : begin
          end
          START : begin
          end
          ACTIVE : begin

$$(foreach)$$(module.in)$$(foreach)$$(var)$$(j.name)_get_analysis_port.get($$(j.name)_trans);$$(endfor
)$$(endfor)

$$(foreach)$$(module.out)$$(foreach)$$(var)$$(j.name)_get_analysis_port.get($$(j.name)_trans);$$(endf
or)$$(endfor)
           $$(module.name)_in.sample();
```

```
            $$(module.name)_out.sample();
          end
        endcase
      end
  endtask
endclass
```

Double_refmod_pkg

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short-> shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("double_refmod_pkg.sv")
package double_refmod_pkg;
 import ovm_pkg::*;
```

```
  `include "trans.svh"
  `include "$$(module.name)_rm.svh"
  `include "$$(module.name)_refmod.svh"
  `include "stimulus_generator.svh"
  `include "comparator.svh"
  `include "env_double_refmod.svh"

endpackage

$$(endfile)
```

Dr_out_emulation

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
```

```
$$(foreach) $$(module.out)$$(file) $$("$$(i.name)_driver.svh")
class $$(i.name)_driver extends ovm_threaded_component;

  typedef enum {
  INACTIVE, START, ACTIVE, ERROR
  } state_e;

  typedef virtual $$(module.name)_if.master_$$(i.type)_$$(i.name) $$(module.name)_if_t;
  $$(module.name)_if_t $$(module.name)_if_i;
  ovm_get_port #(pkt_$$(i.type)) $$(i.name)_from_sg_in;
  local state_e m_state;
  local pkt_$$(i.type) m_req;

  function new(string name, ovm_component parent);
    super.new(name,parent);
  endfunction



  function void build();
    $$(i.name)_from_sg_in = new("$$(i.name)_from_sg_in", this);
  endfunction
  function void set_vif(virtual $$(module.name)_if vif);
    $$(module.name)_if_i = vif;
  endfunction

  function void start_of_simulation();
    m_state = INACTIVE;
  endfunction

  task run();



  string m_trans_str;
  $$(module.name)_if_i.en1 = 0;
  $$(module.name)_if_i.sel1 = 0;

  forever begin
    @(negedge $$(module.name)_if_i.clk);
    //Attention: RESET default is HIGH. Change if the RESET is low.
```

```
    while($$(module.name)_if_i.rst) begin
        @(negedge $$(module.name)_if_i.clk);
     $$(module.name)_if_i.en1 <= 0;
     $$(module.name)_if_i.sel1 <= 0;
        m_state = INACTIVE;
    end

    case(m_state)

    INACTIVE : begin
      // Get next transaction in the input stream
      // begin codeblock try_get
      m_req = new();
      $$(i.name)_from_sg_in.get(m_req);
         if(m_req==null)begin
                $$(module.name)_if_i.sel1 <= 0;
                m_state = INACTIVE;
                continue;
         end else begin
              $$(module.name)_if_i.en1 <= 0;
              $$(module.name)_if_i.sel1 <= 1;
              m_state = START;
          end
    end //INACTIVE
    START : begin
       // Setup bus controls to transition to an ACTIVE state
       $$(module.name)_if_i.sel1 <= 1;
       $$(module.name)_if_i.en1 <= 1;
$$(foreach)$$(signal)        $$(module.name)_if_i.$$(j.name)_$$(i.name) <= $$(endfor)$$(foreach)$$(var)
m_req.get_$$(i.name)();
$$(endfor)
        m_state = ACTIVE;
      end // START
    ACTIVE : begin
       // Setup bus controls to transition to an INACTIVE state
       $$(module.name)_if_i.en1 <= 0;
       $$(module.name)_if_i.sel1 <= 0;
           m_state = INACTIVE;
      end // ACTIVE
```

```
    endcase
  end
  endtask
 endclass
$$(endfile)
$$(endfor)
```

Dr_resp_bir

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short-> shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(foreach) $$(module.inout)$$(file) $$("$$(i.name)_dr_resp_bir.svh")
class $$(i.name)_dr_master extends ovm_threaded_component;
   typedef enum {
   INACTIVE, START, ACTIVE, ERROR
```

```
} state_e;

typedef virtual $$(module.name)_if.master_$$(i.type)_$$(i.name) $$(module.name)_if_t;
$$(module.name)_if_t $$(module.name)_if_i;
ovm_slave_port #(pkt_$$(i.type), pkt_$$(i.type)) $$(i.name)_req_rsp;

local state_e m_state;
local pkt_$$(i.type) m_req;
local pkt_$$(i.type) m_rsp;

function new(string name, ovm_component parent);
  super.new(name,parent);
endfunction

function void build();
  $$(i.name)_req_rsp = new("$$(i.name)_req_rsp Slave Port", this);
endfunction

function void set_vif(virtual $$(module.name)_if vif);
  $$(module.name)_if_i = vif;
endfunction

function void start_of_simulation();
  m_state = INACTIVE;
endfunction

task run();

string m_trans_str;
$$(module.name)_if_i.en = 0;
$$(module.name)_if_i.sel = 0;

forever begin
   @(negedge $$(module.name)_if_i.clk);
  //Attention: RESET default is HIGH. Change if the RESET is low.
  while($$(module.name)_if_i.rst) begin
        @(negedge $$(module.name)_if_i.clk);
    $$(module.name)_if_i.en <= 0;
    $$(module.name)_if_i.sel <= 0;
```

```
        m_state = INACTIVE;
    end
    case(m_state)

        INACTIVE : begin


         // Get next transaction in the input stream
                // begin codeblock try_get
                m_req = new();
                $$(i.name)_from_sg_in.try_get(m_req);
                if(m_req==null) begin
                $$(module.name)_if_i.sel <= 0;
                        m_state = INACTIVE;
                        continue;
                end
                // If not idle, setup bus controls to transition to a START state
              if (m_req.is_idle()) begin
                $$(module.name)_if_i.sel <= 0;
                m_state = INACTIVE;
                void'(slave_port.try_put(null));
                continue;
           end
                $$(module.name)_if_i.en <= 0;
             $$(module.name)_if_i.sel <= 1;
             m_state = START;
                // Setup bus controls for write

             if (m_req.is_write()) begin
                //This signal maybe changed according with the
                //Interface
                        $$(module.name)_if_i.trasmit_en <= 1;
         /* Here put the transactions data according signal dut */
                        $$(foreach)$$(signal)$$(module.name)_if_i.$$(j.name)_$$(i.name)
<=$$(endfor)$$(foreach)$$(var) m_req.get_$$(i.name)();
$$(endfor)
          end
          else
                $$(module.name)_if_i.trasmit_en <= 0;
```

```
          end //INACTIVE

    START : begin

      // Setup bus controls to transition to an ACTIVE state

      $$(module.name)_if_i.sel <= 1;
      $$(module.name)_if_i.en <= 1;
      m_state = ACTIVE;
     end // START

     ACTIVE : begin
                  assert($cast(m_rsp,m_req.clone()));

                  if (!m_req.is_write()) begin
                  $$(module.name)_if.write <= 0;
                   // Setup bus controls to transition to an INACTIVE state
                  // Here put the transactions data according signal dut

         $$(foreach)$$(var)m_rsp.set_$$(i.name)$$(endfor)$$(foreach)$$(signal)($$(module.name)_if_i.$$
(j.name)_$$(i.name));
      $$(endfor)
                  end
       // Setup bus controls to transition to an INACTIVE state

       $$(module.name)_if.en <= 0;
       $$(module.name)_if.sel <= 0;
           m_state = INACTIVE;
          // begin codeblock try_put
           $$(i.name)_from_sg_in.try_put(m_rsp);
       if(m_rsp==null)
         begin
          ovm_report_error ("Driver Responder Error:",
                      "put response failed");
         end
// end codeblock try_put caption path
      end // ACTIVE
    endcase
```

```
   end
 endtask
endclass $$(endfile)$$(endfor)
```

driver

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(foreach) $$(module.in)$$(file) $$("$$(i.name)_driver.svh")
class $$(i.name)_driver extends ovm_threaded_component;

   typedef enum {
   INACTIVE, START, ACTIVE, ERROR
   } state_e;
   typedef virtual $$(module.name)_if.master_$$(i.type)_$$(i.name) $$(module.name)_if_t;
```

```
$$(module.name)_if_t $$(module.name)_if_i;
ovm_get_port #(pkt_$$(i.type)) $$(i.name)_from_sg_in;
local state_e m_state;
local pkt_$$(i.type) m_req;

function new(string name, ovm_component parent);
  super.new(name,parent);
endfunction

function void build();
  $$(i.name)_from_sg_in = new("$$(i.name)_from_sg_in", this);
endfunction
function void set_vif(virtual $$(module.name)_if vif);
  $$(module.name)_if_i = vif;
endfunction

function void start_of_simulation();
  m_state = INACTIVE;
endfunction

task run();
  string m_trans_str;
  $$(module.name)_if_i.en = 0;
  $$(module.name)_if_i.sel = 0;

  forever begin
    @(negedge $$(module.name)_if_i.clk);
  //Attention: RESET default is HIGH. Change if the RESET is low.
  while($$(module.name)_if_i.rst) begin
        @(negedge $$(module.name)_if_i.clk);
    $$(module.name)_if_i.en <= 0;
    $$(module.name)_if_i.sel <= 0;
        m_state = INACTIVE;
  end

  case(m_state)

  INACTIVE : begin
     // Get next transaction in the input stream
```

```
        // begin codeblock try_get
        m_req = new();
        $$(i.name)_from_sg_in.get(m_req);
            if(m_req==null)begin
                    $$(module.name)_if_i.sel <= 0;
                    m_state = INACTIVE;
                    continue;
            end else begin
                  $$(module.name)_if_i.en <= 0;
                  $$(module.name)_if_i.sel <= 1;
                  m_state = START;
              end
      end //INACTIVE
      START : begin
         // Setup bus controls to transition to an ACTIVE state
        $$(module.name)_if_i.sel <= 1;
        $$(module.name)_if_i.en <= 1;
             /* Here put the transactions data according signal dut */
             $$(foreach)$$(signal)$$(module.name)_if_i.$$(j.name)_$$(i.name)
<=$$(endfor)$$(foreach)$$(var) m_req.get_$$(i.name)();
$$(endfor)
         m_state = ACTIVE;
       end // START
       ACTIVE : begin
         // Setup bus controls to transition to an INACTIVE state
         $$(module.name)_if_i.en <= 0;
         $$(module.name)_if_i.sel <= 0;
              m_state = INACTIVE;
       end // ACTIVE

      endcase

    end
    endtask
  endclass
$$(endfile)
$$(endfor)
```

Duv_emulation_pkg

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short-> shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("duv_emulation_pkg.sv")
package duv_emulation_pkg;
 import ovm_pkg::*;
 `include "trans.svh"
 `include "$$(module.name)_rm.svh"
 `include "$$(module.name)_refmod.svh"
 `include "comparator.svh"
 `include "stimulus_generator.svh"
 `include "$$(module.name)_coverage.svh"
  $$(foreach)$$(module.in)
 `include "$$(i.name)_driver.svh"
```

```
  `include "$$(i.name)_responder.svh"
  `include "$$(i.name)_monitor.svh"
  $$(endfor)
  $$(foreach)$$(module.out)
  `include "$$(i.name)_driver.svh"
  `include "$$(i.name)_responder.svh"
  `include "$$(i.name)_monitor.svh"
  $$(endfor)
  `include "$$(module.name)_emulator.svh"
  `include "env_duv_emulation.svh"
endpackage

$$(endfile)
```

Duv_emulator

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint)
$$(type.map) $$(unsigned short int->shortint)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
```

```
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("$$(module.name)_emulator.svh")//Code generated by eTBc software
class $$(module.name)_emulator extends ovm_component;

 $$(module.name)_refmod rem2;

 $$(foreach) $$(module.in)

 $$(i.name)_responder $$(i.name)_res_emu;
 tlm_fifo #(pkt_$$(i.type)) $$(i.name)_res_emu_to_rem2;

 $$(endfor)
 $$(foreach) $$(module.out)
 $$(i.name)_driver $$(i.name)_driver_emu;
 tlm_fifo #(pkt_$$(i.type)) $$(i.name)_rem2_to_dr_emu;

 $$(endfor)

 virtual $$(module.name)_if $$(module.name)_if;

 function new(string name, ovm_component parent = null);
  super.new(name, parent);
 endfunction

function void build();

 rem2       = new("$$(module.name)_refmod2", this);
 $$(foreach) $$(module.in)
 $$(i.name)_res_emu = new("$$(i.name)_res_emulave",this);
 $$(i.name)_res_emu_to_rem2= new("$$(i.name)_res_emu_to_refmod2_fifo", this);

 $$(endfor)
 $$(foreach) $$(module.out)
 $$(i.name)_driver_emu = new("$$(i.name)_driver_emuaster",this);
 $$(i.name)_rem2_to_dr_emu = new("$$(i.name)_rem2_dr_emu_emulation_fifo", this);
 $$(endfor)
```

```
endfunction

function void connect();

$$(foreach) $$(module.in)
$$(i.name)_res_emu.$$(i.name)_to_comparator.connect($$(i.name)_res_emu_to_rem2.put_export);
rem2.$$(i.name)_get_stim.connect($$(i.name)_res_emu_to_rem2.get_export);

$$(endfor)  $$(foreach) $$(module.out)
rem2.$$(i.name)_put_stim.connect($$(i.name)_rem2_to_dr_emu.put_export);
$$(i.name)_driver_emu.$$(i.name)_from_sg_in.connect($$(i.name)_rem2_to_dr_emu.get_export);
$$(endfor)
endfunction

function void set_vif(virtual $$(module.name)_if vif);
  $$(foreach)$$(module.in)$$(i.name)_res_emu.set_vif(vif);$$(endfor)
  $$(foreach)$$(module.out)$$(i.name)_driver_emu.set_vif(vif);$$(endfor)
endfunction

endclass

$$(endfile)
```

Duv_execution_pkg

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
```

```
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short-> shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("duv_execution_pkg.sv")
package duv_execution_pkg;
 import ovm_pkg::*;
 `include "trans.svh"
 `include "$$(module.name)_rm.svh"
 `include "$$(module.name)_refmod.svh"
 `include "stimulus_generator.svh"
 `include "comparator.svh"
  $$(foreach)$$(module.in)
 `include "$$(i.name)_driver.svh"
 `include "$$(i.name)_monitor.svh"
 $$(endfor)
 $$(foreach)$$(module.out)
 `include "$$(i.name)_responder.svh"
 `include "$$(i.name)_monitor.svh"
  $$(endfor)
 `include "env_duv.svh"
endpackage

$$(endfile)
```

Env_double_refmod

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

```
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("env_double_refmod.svh")
class env_double_refmod extends ovm_component;

  stimulus_generator sg_in;
  $$(module.name)_refmod rem1;
  $$(module.name)_refmod rem2;
  comparator comparator_i;

  $$(foreach) $$(module.in)
  tlm_fifo #(pkt_$$(i.type)) $$(i.name)_sg_in_rem1;
  tlm_fifo #(pkt_$$(i.type)) $$(i.name)_sg_in_rem2;
  $$(endfor)

  $$(foreach) $$(module.out)
  tlm_fifo #(pkt_$$(i.type)) $$(i.name)_rem1_comparator;
  tlm_fifo #(pkt_$$(i.type)) $$(i.name)_rem2_comparator;
```

```
$$(endfor)

virtual $$(module.name)_if $$(module.name)_if_i;

function new(string name, ovm_component parent = null);
  super.new(name, parent);
endfunction

function void build();

  sg_in        = new("Stimulus_generator_in", this);
  comparator_i        = new("comparator", this);
  rem1       = new("$$(module.name)_refmod1", this);
  rem2       = new("$$(module.name)_refmod2", this);
$$(foreach) $$(module.in)
  $$(i.name)_sg_in_rem1 = new("$$(i.name)_sg_in_rem1_fifo", this);
  $$(i.name)_sg_in_rem2 = new("$$(i.name)_sg_in_rem2_fifo", this);
$$(endfor)
$$(foreach) $$(module.out)
  $$(i.name)_rem1_comparator = new("$$(i.name)_rem1_comparator_fifo", this);
  $$(i.name)_rem2_comparator = new("$$(i.name)_rem2_comparator_fifo", this);
$$(endfor)

endfunction

function void connect();
  $$(foreach) $$(module.in)
  sg_in.$$(i.name)_to_rem.connect($$(i.name)_sg_in_rem1.put_export);
  sg_in.$$(i.name)_to_dr.connect($$(i.name)_sg_in_rem2.put_export);
  rem1.$$(i.name)_get_stim.connect($$(i.name)_sg_in_rem1.get_export);
  rem2.$$(i.name)_get_stim.connect($$(i.name)_sg_in_rem2.get_export);
$$(endfor)  $$(foreach) $$(module.out)
  rem1.$$(i.name)_put_stim.connect($$(i.name)_rem1_comparator.put_export);
  rem2.$$(i.name)_put_stim.connect($$(i.name)_rem2_comparator.put_export);
  comparator_i.$$(i.name)_from_rem.connect($$(i.name)_rem1_comparator.get_export);
  comparator_i.$$(i.name)_from_duv.connect($$(i.name)_rem2_comparator.get_export);
$$(endfor)
endfunction
```

```
  function void set_vif(virtual $$(module.name)_if vif);
    $$(module.name)_if_i = vif;
  endfunction
endclass
$$(endfile)
```

Env_duv

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("env_duv.svh")
class env_duv extends ovm_component;

  stimulus_generator sg_in;
  $$(module.name)_refmod rem1;
```

```
comparator comparator_i;

$$(foreach) $$(module.in)
$$(i.name)_driver $$(i.name)_dr;
$$(i.name)_monitor $$(i.name)_mon;
tlm_fifo #(pkt_$$(i.type)) $$(i.name)_sg_in_rem1;
tlm_fifo #(pkt_$$(i.type)) $$(i.name)_sg_in_to_dr;
$$(endfor)

$$(foreach) $$(module.out)
$$(i.name)_responder $$(i.name)_res;
$$(i.name)_monitor $$(i.name)_mon_out;
tlm_fifo #(pkt_$$(i.type)) $$(i.name)_rem1_check;
tlm_fifo #(pkt_$$(i.type)) $$(i.name)_rem2_check;
$$(endfor)

virtual $$(module.name)_if $$(module.name)_if_i;

function new(string name, ovm_component parent = null);
  super.new(name, parent);
endfunction

function void build();

  sg_in        = new("stimulus_generator", this);
  comparator_i        = new("comparator", this);
  rem1       = new("$$(module.name)_refmod1", this);



$$(foreach) $$(module.in)
  $$(i.name)_dr = new("$$(i.name)_driver",this);
  $$(i.name)_mon = new("$$(i.name)_monitor_in", this);
  $$(i.name)_sg_in_rem1 = new("$$(i.name)_sg_in_rem1_fifo", this);
  $$(i.name)_sg_in_to_dr = new("$$(i.name)_sg_in_to_driver_fifo", this);
$$(endfor)
$$(foreach) $$(module.out)
  $$(i.name)_res = new("$$(i.name)_responder",this);
  $$(i.name)_mon_out = new("$$(i.name)_monitor_out", this);
```

```
  $$(i.name)_rem1_check = new("$$(i.name)_rem1_check_fifo", this);
  $$(i.name)_rem2_check = new("$$(i.name)_rem2_check_fifo", this);
 $$(endfor)

 endfunction

 function void connect();
  $$(foreach) $$(module.in)
  sg_in.$$(i.name)_to_rem.connect($$(i.name)_sg_in_rem1.put_export);
  sg_in.$$(i.name)_to_dr.connect($$(i.name)_sg_in_to_dr.put_export);
  rem1.$$(i.name)_get_stim.connect($$(i.name)_sg_in_rem1.get_export);
  $$(i.name)_dr.$$(i.name)_from_sg_in.connect($$(i.name)_sg_in_to_dr.get_export);
  $$(i.name)_dr.set_vif($$(module.name)_if_i);
  $$(i.name)_mon.set_vif($$(module.name)_if_i);
  $$(endfor)  $$(foreach) $$(module.out)
  rem1.$$(i.name)_put_stim.connect($$(i.name)_rem1_check.put_export);
  comparator_i.$$(i.name)_from_rem.connect($$(i.name)_rem1_check.get_export);
  $$(i.name)_res.$$(i.name)_to_comparator.connect($$(i.name)_rem2_check.put_export);
  comparator_i.$$(i.name)_from_duv.connect($$(i.name)_rem2_check.get_export);
  $$(i.name)_res.set_vif($$(module.name)_if_i);
  $$(i.name)_mon_out.set_vif($$(module.name)_if_i);
 $$(endfor)

 endfunction

 function void set_vif(virtual $$(module.name)_if vif);
  $$(module.name)_if_i = vif;
 endfunction

endclass
$$(endfile)
```

Env_duv_emulation

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
```

```
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("env_duv_emulation.svh")
class env_duv_emulation extends ovm_component;

 stimulus_generator sg_in;
 $$(module.name)_refmod rem1;
 $$(module.name)_emulator duv_emulator_i;
 comparator comparator_i;
 $$(module.name)_coverage coverage_i;
 $$(foreach) $$(module.in)
 $$(i.name)_driver $$(i.name)_driver;
 $$(i.name)_monitor $$(i.name)_mon;
 tlm_fifo #(pkt_$$(i.type)) $$(i.name)_sg_in_rem1;
 tlm_fifo #(pkt_$$(i.type)) $$(i.name)_sg_in_to_dr;
 tlm_analysis_fifo #(pkt_$$(i.type)) $$(i.name)_anafifo_to_cov;
 $$(endfor)

 $$(foreach) $$(module.out)
 $$(i.name)_responder $$(i.name)_res;
```

```
$$(i.name)_monitor $$(i.name)_mon;
tlm_fifo #(pkt_$$(i.type)) $$(i.name)_rem1_comparator;
tlm_fifo #(pkt_$$(i.type)) $$(i.name)_rem2_comparator;
tlm_analysis_fifo #(pkt_$$(i.type)) $$(i.name)_anafifo_to_cov;
$$(endfor)

virtual $$(module.name)_if $$(module.name)_if_i;

function new(string name, ovm_component parent = null);
  super.new(name, parent);
endfunction

function void build();

  sg_in        = new("Stimulus_generator", this);
  comparator_i        = new("comparator", this);
  rem1       = new("$$(module.name)_refmod1", this);
  duv_emulator_i = new ("$$(module.name)_duv_emulator",this);
  coverage_i = new ("$$(module.name)_coverage",this);
$$(foreach) $$(module.in)
  $$(i.name)_driver = new("$$(i.name)_driver",this);
  $$(i.name)_mon = new("$$(i.name)_monitor_in", this);
  $$(i.name)_sg_in_rem1 = new("$$(i.name)_sg_in_rem1_fifo", this);
  $$(i.name)_sg_in_to_dr = new("$$(i.name)_sg_in_to_driver_fifo", this);
  $$(i.name)_anafifo_to_cov = new("$$(i.name)_analysis_mon_to_cov_fifo", this);
$$(endfor)
$$(foreach) $$(module.out)
  $$(i.name)_res = new("$$(i.name)_res",this);
  $$(i.name)_mon = new("$$(i.name)_monitor_out", this);
  $$(i.name)_rem1_comparator = new("$$(i.name)_rem1_comparator_fifo", this);
  $$(i.name)_rem2_comparator = new("$$(i.name)_rem2_comparator_fifo", this);
  $$(i.name)_anafifo_to_cov = new("$$(i.name)_analysis_mon_to_cov_fifo", this);
$$(endfor)

endfunction

function void connect();
  $$(foreach) $$(module.in)
  sg_in.$$(i.name)_to_rem.connect($$(i.name)_sg_in_rem1.put_export);
```

```
  sg_in.$$(i.name)_to_dr.connect($$(i.name)_sg_in_to_dr.put_export);
  rem1.$$(i.name)_get_stim.connect($$(i.name)_sg_in_rem1.get_export);
  $$(i.name)_driver.$$(i.name)_from_sg_in.connect($$(i.name)_sg_in_to_dr.get_export);
  $$(i.name)_driver.set_vif($$(module.name)_if_i);
  $$(i.name)_mon.set_vif($$(module.name)_if_i);
  $$(i.name)_mon.$$(i.name)_analysis_port.connect($$(i.name)_anafifo_to_cov.analysis_export);
  coverage_i.$$(i.name)_get_analysis_port.connect($$(i.name)_anafifo_to_cov.blocking_get_export);
  duv_emulator_i.set_vif($$(module.name)_if_i);
  $$(endfor)  $$(foreach) $$(module.out)
  rem1.$$(i.name)_put_stim.connect($$(i.name)_rem1_comparator.put_export);
  comparator_i.$$(i.name)_from_rem.connect($$(i.name)_rem1_comparator.get_export);
  $$(i.name)_res.$$(i.name)_to_comparator.connect($$(i.name)_rem2_comparator.put_export);
  comparator_i.$$(i.name)_from_duv.connect($$(i.name)_rem2_comparator.get_export);
  $$(i.name)_res.set_vif($$(module.name)_if_i);
  $$(i.name)_mon.set_vif($$(module.name)_if_i);
  $$(i.name)_mon.$$(i.name)_analysis_port.connect($$(i.name)_anafifo_to_cov.analysis_export);
  coverage_i.$$(i.name)_get_analysis_port.connect($$(i.name)_anafifo_to_cov.blocking_get_export);
  duv_emulator_i.set_vif($$(module.name)_if_i);
 $$(endfor)
  coverage_i.set_vif($$(module.name)_if_i);
 endfunction

 function void set_vif(virtual $$(module.name)_if vif);
  $$(module.name)_if_i = vif;
 endfunction
endclass
$$(endfile)
```

interface

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
```

```
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short-> shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("$$(module.name)_if.sv")
interface $$(module.name)_if (input bit clk, input bit rst);
  //set time unit and timeprecision
  timeunit       1ns ;
  timeprecision 100ps ;

  bit sel;
  bit en;
  bit sel1;
  bit en1;
  $$(foreach) $$(module.in) $$(foreach) $$(signal)
  bit [$$(i.size)-1:0]  $$(j.name)_$$(i.name);
  $$(endfor) $$(endfor)
  $$(foreach) $$(module.out) $$(foreach) $$(signal)
  bit [$$(i.size)-1:0]  $$(j.name)_$$(i.name);
  $$(endfor) $$(endfor)$$(foreach) $$(module.in)
  modport master_$$(i.type)_$$(i.name)(
    input clk,
    input rst,
    output sel,
    output en,$$(foreach)$$(signal)
    output $$(j.name)_$$(i.name)$$(if)$$(isnotlast),$$(endif)$$(endfor)
```

```
);
modport slave_$$(i.type)_$$(i.name)(
  input clk,
  input rst,
  input sel,
  input en,$$(foreach) $$(signal)
  input $$(j.name)_$$(i.name)$$(if)$$(isnotlast),$$(endif)$$(endfor)
);
modport monitor_$$(i.type)_$$(i.name)(
  input clk,
  input rst,
  input sel,
  input en,  $$(foreach) $$(signal)
  input $$(j.name)_$$(i.name)$$(if)$$(isnotlast),$$(endif)$$(endfor)
);  $$(endfor) $$(foreach) $$(module.out)
modport master_$$(i.type)_$$(i.name)(
  input clk,
  input rst,
  output sel1,
  output en1,$$(foreach) $$(signal)
  output $$(j.name)_$$(i.name)$$(if)$$(isnotlast),$$(endif)$$(endfor)
);
modport slave_$$(i.type)_$$(i.name)(
  input clk,
  input rst,
  input sel1,
  input en1,$$(foreach)$$(signal)
  input $$(j.name)_$$(i.name)$$(if)$$(isnotlast),$$(endif)$$(endfor)
);
  modport monitor_$$(i.type)_$$(i.name)(
  input clk,
  input rst,
  input sel1,
  input en1,$$(foreach)$$(signal)
  input $$(j.name)_$$(i.name)$$(if)$$(isnotlast),$$(endif)$$(endfor)
);$$(endfor)
  modport coverage_$$(module.name)(
  input clk,
  input rst,
```

```
   input sel,
   input en,
   input sel1,
   input en1,$$(endfor)$$(foreach) $$(module.out)$$(foreach)$$(signal)
   input $$(j.name)_$$(i.name), $$(endfor)$$(endfor) $$(foreach) $$(module.in) $$(foreach)$$(signal)
   input $$(j.name)_$$(i.name)$$(if)$$(isnotlast),$$(endif)$$(endfor)
 ); $$(endfor)
 endinterface
$$(endfile)
```

Monitor_in

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(foreach) $$(module.in)
```

```
$$(file) $$("$$(i.name)_monitor.svh")

class $$(i.name)_monitor extends ovm_component;

  typedef virtual $$(module.name)_if.monitor_$$(i.type)_$$(i.name) $$(module.name)_if_t;
  $$(module.name)_if_t $$(module.name)_if_i;

  ovm_analysis_port #(pkt_$$(i.type)) $$(i.name)_analysis_port;

  local pkt_$$(i.type) m_trans;

  typedef enum bit [1:0]{ INACTIVE = 2'b00,
                          START    = 2'b10,
                          ACTIVE   = 2'b11,
                          ERROR    = 2'b01
                        } state_t;


  function new( string name, ovm_component parent);
    super.new( name, parent );
  endfunction

  function void build();
    $$(i.name)_analysis_port = new("$$(i.name)_analysis_port", this);
  endfunction

  function void set_vif(virtual $$(module.name)_if vif);
    $$(module.name)_if_i = vif;
  endfunction

 task run();

    state_t state;

// begin codeblock monitor_loop
    forever begin
      @( negedge $$(module.name)_if_i.clk );

      while($$(module.name)_if_i.rst) @(negedge $$(module.name)_if_i.clk);
```

```
    state = state_t'({ ($$(module.name)_if_i.sel != 0),
                       $$(module.name)_if_i.en });
    case( state )

      INACTIVE : begin
      //Assertions com o seguinte parametro, sel = 0 e en=0
      end

      START : begin
      //Assertions com o seguinte parametro, sel = 1 e en=0
        if(!$$(module.name)_if_i.sel)
          break;
        m_trans = new();
      end
      ACTIVE : begin
        $$(foreach)$$(var)

m_trans.set_$$(i.name)$$(endfor)$$(foreach)$$(signal)($$(module.name)_if_i.$$(j.name)_$$(i.name));
        $$(endfor)
        $$(i.name)_analysis_port.write(m_trans);

        //Assertions com o seguinte parametro, sel = 1 e en=1
        if(!$$(module.name)_if_i.en)
          break;

      end
    endcase
  end
// end codeblock monitor_loop caption path
 endtask
endclass
$$(endfile)
$$(endfor)
```

Monitor_out

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

```
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short-> shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(foreach) $$(module.out)
$$(file) $$("$$(i.name)_monitor.svh")
class $$(i.name)_monitor extends ovm_component;
  typedef virtual $$(module.name)_if.monitor_$$(i.type)_$$(i.name) $$(module.name)_if_t;
  $$(module.name)_if_t $$(module.name)_if_i;
  ovm_analysis_port #(pkt_$$(i.type)) $$(i.name)_analysis_port;
  local pkt_$$(i.type) m_trans;

  typedef enum bit [1:0]{ INACTIVE = 2'b00,
                          START    = 2'b10,
                          ACTIVE   = 2'b11,
                          ERROR    = 2'b01
                        } state_t;

  function new( string name, ovm_component parent);
```

```
  super.new( name, parent );
 endfunction

 function void build();
  $$(i.name)_analysis_port = new("$$(i.name)_analysis_port", this);
 endfunction

 function void set_vif(virtual $$(module.name)_if vif);
  $$(module.name)_if_i = vif;
 endfunction

task run();

  state_t state;

// begin codeblock monitor_loop
  forever begin
    @( negedge $$(module.name)_if_i.clk );

    while($$(module.name)_if_i.rst) @(negedge $$(module.name)_if_i.clk);

    state = state_t'({ ($$(module.name)_if_i.sel1 != 0),
                       $$(module.name)_if_i.en1 });
    case( state )

      INACTIVE : begin
      //Assertions com o seguinte parametro, sel = 0 e en=0
      end

      START : begin
      //Assertions com o seguinte parametro, sel = 1 e en=0
       if(!$$(module.name)_if_i.sel1)
          break;

       m_trans = new();


      /*ovm_report_info("MONITOR_START", m_trans.do_sprint());*/
```

```
//         ovm_report_info();

       end

       ACTIVE : begin

        //Assertions com o seguinte parametro, sel = 1 e en=1
         if(!$$(module.name)_if_i.en1)
           break;
         $$(foreach)$$(var)

m_trans.set_$$(i.name)$$(endfor)$$(foreach)$$(signal)($$(module.name)_if_i.$$(j.name)_$$(i.name));
         $$(endfor)
         $$(i.name)_analysis_port.write(m_trans);
       end
     endcase
   end
// end codeblock monitor_loop caption path
  endtask
endclass
$$(endfile)
$$(endfor)
```

Refmod

```
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
```

```
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)

$$(file) $$("$$(module.name)_refmod.svh")
class $$(module.name)_refmod extends ovm_component;

  $$(foreach)$$(module.in)ovm_get_port #(pkt_$$(i.type)) $$(i.name)_get_stim;
  pkt_$$(i.type) tr_in_$$(i.name);
  $$(endfor)
  $$(foreach)$$(module.out)ovm_put_port #(pkt_$$(i.type)) $$(i.name)_put_stim;
  pkt_$$(i.type) tr_out_$$(i.name);
  $$(endfor)
  $$(module.name)_rm $$(module.name)_i;

  function new(string name, ovm_component parent);
    super.new(name,parent);
  endfunction

  function void build();
  $$(foreach)$$(module.in)$$(i.name)_get_stim = new("pkt_$$(i.name)_get_stim", this);
   $$(endfor)
   $$(foreach)$$(module.out)$$(i.name)_put_stim = new("pkt_$$(i.name)_put_stim", this);
  $$(endfor)
  $$(module.name)_i = new;
  endfunction

  task run();
    string msg;
    forever begin
      $$(foreach)$$(module.in)tr_in_$$(i.name) = new();
      $$(endfor)
      $$(foreach)$$(module.in)$$(i.name)_get_stim.get(tr_in_$$(i.name));
      $$(endfor)
      $$(foreach)$$(module.out)tr_out_$$(i.name)= new();
      $$(endfor)
```

```
      //-------------------------------------------------------------------
      // Here goes the code that get instatiate the reference model's functionality.
      //-------------------------------------------------------------------
          $$(foreach)$$(module.out)$$(foreach)$$(var)
      tr_out_$$(j.name).set_$$(i.name)($$(module.name)_i.Funcao do refmod);
          $$(endfor)$$(endfor)
      $$(foreach)$$(module.out)$$(i.name)_put_stim.put(tr_out_$$(i.name));
      $$(endfor)
    end
  endtask
endclass

$$(endfile)
```

Refmod_bir

```
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)

$$(file) $$("$$(module.name)_refmod.svh")
class $$(module.name)_refmod extends ovm_component;
```

```
  $$(foreach)$$(module.inout)ovm_slave_port #(pkt_$$(i.type), pkt_$$(i.type)) $$(i.name)_slave_stim;
  pkt_$$(i.type) tr_in_$$(i.name);
  pkt_$$(i.type) tr_out_$$(i.name);
  $$(endfor)

  $$(module.name)_rm $$(module.name)_i;

  function new(string name, ovm_component parent);
    super.new(name,parent);
  endfunction

  function void build();
  $$(foreach)$$(module.inout)$$(i.name)_slave_stim = new("pkt_$$(i.name)_slave_stim", this);
   $$(endfor)
  $$(module.name)_i = new;
  endfunction

  task run();
    string msg;

    forever begin
      $$(foreach)$$(module.inout)tr_in_$$(i.name) = new();
      tr_out_$$(i.name) = new();
      $$(endfor)
      $$(foreach)$$(module.inout)$$(i.name)_slave_stim.get(tr_in_$$(i.name));
      $$(endfor)
      //------------------------------------------------------------------
      // Here goes the code that instantiate the reference model's functionality.
      //------------------------------------------------------------------
          $$(foreach)$$(module.inout)$$(foreach)$$(var)
      tr_out_$$(j.name).set_$$(i.name)($$(module.name)_i.Funcao do refmod);
          $$(endfor)$$(endfor)
      $$(foreach)$$(module.inout)$$(i.name)_slave_stim.put(tr_out_$$(i.name));
      $$(endfor)
    end
   endtask
endclass
$$(endfile)
```

Responder

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short-> shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(foreach) $$(module.out)
$$(file) $$("$$(i.name)_responder.svh")
class $$(i.name)_responder extends ovm_threaded_component;

   typedef virtual $$(module.name)_if.slave_$$(i.type)_$$(i.name) $$(module.name)_if_t;
   $$(module.name)_if_t $$(module.name)_if_i;
   ovm_put_port #(pkt_$$(i.type)) $$(i.name)_to_comparator;
   local pkt_$$(i.type) m_req;

   typedef enum bit [1:0]{ INACTIVE = 2'b00,
                           START    = 2'b10,
```

```
                ACTIVE   = 2'b11,
                ERROR    = 2'b01
               } state_t;
function new( string name, ovm_component parent);
  super.new( name, parent );
endfunction

function void build();
  $$(i.name)_to_comparator = new("$$(i.name)_to_comparator", this);

endfunction

function void set_vif(virtual $$(module.name)_if vif);
  $$(module.name)_if_i = vif;
endfunction

task run();

  string s_trans_str;
  state_t state;
forever begin
  @(negedge $$(module.name)_if_i.clk);
  //Attention: RESET default is HIGH. Change if the RESET is low.
 while($$(module.name)_if_i.rst) @(negedge $$(module.name)_if_i.clk);
  state = state_t'( {$$(module.name)_if_i.sel1, $$(module.name)_if_i.en1});

  case( state )

  INACTIVE : begin

  end //INACTIVE
  START : begin

    m_req = new();

  end //START

  ACTIVE : begin
    // Setup bus controls to transition to an INACTIVE state
```

```
            // Here put the transactions data according signal dut

$$(foreach)$$(var)m_req.set_$$(i.name)$$(endfor)$$(foreach)$$(signal)($$(module.name)_if_i.$$(j.name)
_$$(i.name));
       $$(endfor)
       $$(i.name)_to_comparator.put(m_req);

     end // ACTIVE

     endcase

   end
   endtask

 endclass

$$(endfile)
$$(endfor)
```

Responder_in_emulation

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short-> shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
```

```
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(foreach) $$(module.in)$$(file) $$("$$(i.name)_responder.svh")
class $$(i.name)_responder extends ovm_threaded_component;

  typedef virtual $$(module.name)_if.slave_$$(i.type)_$$(i.name) $$(module.name)_if_t;
  $$(module.name)_if_t $$(module.name)_if_i;
  ovm_put_port #(pkt_$$(i.type)) $$(i.name)_to_comparator;
  local pkt_$$(i.type) m_req;

  typedef enum bit [1:0]{ INACTIVE = 2'b00,
                  START    = 2'b10,
                  ACTIVE   = 2'b11,
                  ERROR    = 2'b01
                 } state_t;
  function new( string name, ovm_component parent);
    super.new( name, parent );
  endfunction

  function void build();
    $$(i.name)_to_comparator = new("$$(i.name)_to_comparator", this);

  endfunction

  function void set_vif(virtual $$(module.name)_if vif);
    $$(module.name)_if_i = vif;
  endfunction

  task run();

    string s_trans_str;
    state_t state;
  forever begin
```

```
    @(negedge $$(module.name)_if_i.clk);
    //Attention: RESET default is HIGH. Change if the RESET is low.
    while($$(module.name)_if_i.rst) @(negedge $$(module.name)_if_i.clk);
    state = state_t'( {$$(module.name)_if_i.sel, $$(module.name)_if_i.en} );

    case( state )

    INACTIVE : begin

    end //INACTIVE
    START : begin

      m_req = new();

    end //START

    ACTIVE : begin
          // Setup bus controls to transition to an INACTIVE state
          // Here put the transactions data according signal dut

$$(foreach)$$(var)m_req.set_$$(i.name)$$(endfor)$$(foreach)$$(signal)($$(module.name)_if_i.$$(j.name)
_$$(i.name));
      $$(endfor)
      $$(i.name)_to_comparator.put(m_req);

    end // ACTIVE

    endcase

  end
  endtask

 endclass

$$(endfile)
$$(endfor)
```

Stim_comp_bir
$$(type.map) $$(bool->bit)

```
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(foreach) $$(module.inout) $$(file) $$("stim_comp_$$(module.name).svh")
class stim_compp_$$(module.name) extends ovm_component;
   ovm_transport_port #(pkt_$$(i.type), pkt_$$(i.type)) $$(i.name)_to_rem;
   ovm_transport_port #(pkt_$$(i.type), pkt_$$(i.type)) $$(i.name)_to_dr_resp;
   typedef enum {IDLE, WRITE, READ} task_trans_t;
   rand task_trans_t        task_trans_type;
 $$(foreach)$$(var)
  rand bit [$$(i.size)-1:0]  $$(i.name)_t;
  $$(endfor)
  int unsigned numberOfError;
  int unsigned numberOfTrans;
  function new(string name, ovm_component parent);
    super.new(name,parent);
```

```
  numberOfError = 0;
  numberOfTrans = 0;
endfunction
function void build();
  $$(i.name)_to_rem = new("transport_port_refmod", this);
  $$(i.name)_to_dr_resp = new("transport_port_to_dr_resp", this);
endfunction
function int getNumberOfError();
  return numberOfError;
endfunction
function void setNumberOfError();
  this.numberOfError++;
endfunction
function int getNumberOfTrans();
  return numberOfTrans;
endfunction
function void setNumberOfTrans();
  this.numberOfTrans++;
endfunction
task run();
          string msg;
          pkt_$$(i.type) tr_$$(i.name)_req, tr_$$(i.name)_rsp1, tr_$$(i.name)_rsp2;
      int file = $fopen("$$(module.name).sti", "r");
string s;

ovm_report_info("run", "Testbench starting to generate stimulus");
//Start. Change this looping for the size module.
//This automatic generate code with 1024 position. This loop put 0 for each
          for(int j = 0; j < 1023; j++) begin
       tr_req = new();
       tr_$$(i.name)_req.set_addr(j);
            tr_$$(i.name)_req.set_wdata(0);
       ovm_report_info("Transmit", "The reception is ignore");
       $$(i.name)_to_rem.transport(tr_$$(i.name)_req,tr_$$(i.name)_rsp1);
            $$(i.name)_to_dr_resp.transport(tr_$$(i.name)_req,tr_$$(i.name)_rsp2);
     end

// begin codeblock master_loop
 forever begin
```

```
        tr_req = new();
        if(tr_req.is_write) begin
                if(!tr_$$(i.name).read(file))
                assert(tr_$$(i.name).randomize());
                        $$(i.name)_to_rem.transport(tr_$$(i.name)_req,tr_$$(i.name)_rsp1);
                        $$(i.name)_to_dr_resp.transport(tr_$$(i.name)_req,tr_$$(i.name)_rsp2);
        #0;
    end


//reception loop
        if(tr_req.is_read) begin
                tr_req.set_wdata(0);

        $$(i.name)_to_rem.transport(tr_$$(i.name)_req,tr_$$(i.name)_rsp1);
        $$(i.name)_to_dr_resp.transport(tr_$$(i.name)_req,tr_$$(i.name)_rsp2);
        setNumberOfTrans();
    if(!tr_$$(i.name)_rsp1.comp(tr_$$(i.name)_rsp2)) begin
      msg = $psprintf("received:  %s  expected  %s",
        tr_duv_$$(i.name).psprint(), tr_rem_$$(i.name).psprint() );
      ovm_report_error("Comparator", msg);
      setNumberOfError();
      assert(record_error_tr("Comparator"));
      if (getNumberOfError() > 1) begin
        #100ns; //Stop the simulation 100ns after error
        global_stop_request();
      end
    end
  /* else begin
    msg = $psprintf("received:  %s  expected  %s",
        tr_duv_$$(i.name).psprint(), tr_rem_$$(i.name).psprint() );
      ovm_report_info("Comparator", msg);
    end*/
  end

endtask // end codeblock master_loop

function void report;
```

```
   string s;
   s =   $psprintf("Number of transactions: %d, Number of errors: %d", getNumberOfTrans(), getNumberOfError());
   ovm_report_info("report", s);
 endfunction

endclass $$(endfile)$$(endfor)
```

Stimulus_generator

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("stimulus_generator.svh")
class stimulus_generator extends ovm_component;
```

```
$$(foreach)  $$(module.in)
  ovm_put_port #(pkt_$$(i.type)) $$(i.name)_to_rem;
  ovm_put_port #(pkt_$$(i.type)) $$(i.name)_to_dr;
$$(endfor)
  function new(string name, ovm_component parent);
    super.new(name,parent);
  endfunction
  function void build();
  $$(foreach)  $$(module.in)
    $$(i.name)_to_rem = new("pkt_$$(i.name)_to_rem", this);
    $$(i.name)_to_dr = new("pkt_$$(i.name)_to_dr", this);
  $$(endfor)
  endfunction

  task run();
    $$(foreach)  $$(module.in)
      pkt_$$(i.type) tr_$$(i.name);
    $$(endfor)
      int file = $fopen("$$(module.name).sti", "r");
      forever begin
        $$(foreach)  $$(module.in)
          tr_$$(i.name) = new();
          if(!tr_$$(i.name).read(file))
            assert(tr_$$(i.name).randomize());
          $$(i.name)_to_dr.put(tr_$$(i.name));
          $$(i.name)_to_rem.put(tr_$$(i.name));
        $$(endfor)
          #0;
      end
  endtask

endclass

$$(endfile)
```

Tb_run

```
$$(file) $$("tb.run")
irun -clean  -ovmhome ~/ovm-2.1 -access +rw -coverage all ./$$(module.name)_if.sv ./clk_rst_gen.sv ./double_refmod_pkg.sv ./clk_rst_if.sv top.sv
```

```
$$(endfile)
```

Tb_run_duv_execution

```
$$(file) $$("tb.run")
irun -clean  -ovmhome ~/ovm-2.1 -access +rw -coverage all ./$$(module.name)_if.sv ./clk_rst_gen.sv ./duv_execution_pkg.sv ./clk_rst_if.sv top.sv $$(module.name).sv
$$(endfile)
```

Tb_run_emulation

```
$$(file) $$("tb.run")
irun -clean  -ovmhome ~/ovm-2.1 -access +rw -coverage all ./$$(module.name)_if.sv ./clk_rst_gen.sv ./duv_emulation_pkg.sv ./clk_rst_if.sv top.sv
$$(endfile)
```

Top_double_refmod

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
```

```
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("top.sv")
import ovm_pkg::*;
import double_refmod_pkg::*;
module top_double_refmod;
  //set time unit and timeprecision
  timeunit       1ns ;
  timeprecision 100ps ;

  env_double_refmod e;
  clk_rst_if cr();
  clk_rst_gen ckr (cr);
  $$(module.name)_if $$(module.name)_if_i (cr.clk, cr.rst);

  initial begin
    e = new("environment");


     // The set_vif() call is used to pass the virtual interface from the
    // module-based comdponent world into the class-based component
    // world
    e.set_vif($$(module.name)_if_i);

    // start the clock running
    fork
      ckr.run();
    join_none
    run_test();
  end

  logic cov;
  initial cov = 0;
  always #100 cov = !cov;

endmodule
$$(endfile)
```

Top_duv

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("top.sv")
import ovm_pkg::*;
import hierarchical_duv_pkg::*;
module top_duv;
  //set time unit and timeprecision
  timeunit       1ns ;
  timeprecision 100ps ;

  env_duv e;
  clk_rst_if cr();
  clk_rst_gen ckr (cr);
```

```
  $$(module.name)_if $$(module.name)_if_i (cr.clk, cr.rst);
  $$(module.name) $$(module.name)_i(.clk(cr.clk),.*);
  initial begin
    e = new("environment");
     // The set_vif() call is used to pass the virtual interface from the
    // module-based comdponent world into the class-based component
    // world
    e.set_vif($$(module.name)_if_i);

    // start the clock running
    fork
      ckr.run();
    join_none
    run_test();
  end

  logic cov;
  initial cov = 0;
  always #100 cov = !cov;

endmodule
$$(endfile)
```

Top_duv_emulation

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
```

```
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("top.sv")
import ovm_pkg::*;
import duv_emulation_pkg::*;
module top_duv_emulation;
 //set time unit and timeprecision
 timeunit       1ns ;
 timeprecision 100ps ;

 env_duv_emulation e;
 clk_rst_if cr();
 clk_rst_gen ckr (cr);
 $$(module.name)_if $$(module.name)_if_i (cr.clk, cr.rst);
 initial begin
   e = new("environment");
    // The set_vif() call is used to pass the virtual interface from the
   // module-based comdponent world into the class-based component
   // world
   e.set_vif($$(module.name)_if_i);
   // start the clock running
   fork
     ckr.run();
     $shm_open("waves.shm");
     $shm_probe(cr.clk,          cr.rst,          $$(module.name)_if_i.sel,          $$(module.name)_if_i.sel1,
$$(module.name)_if_i.en,                                                          $$(module.name)_if_i.en1,
$$(foreach)$$(module.in)$$(foreach)$$(signal)$$(module.name)_if_i.$$(j.name)_$$(i.name),$$(endfor)$$(
endfor)$$(foreach)$$(module.out)$$(foreach)$$(signal)$$(module.name)_if_i.$$(j.name)_$$(i.name)$$(if)
$$(isnotlast), $$(endif)$$(endfor)$$(endfor));
```

```
  join_none
  run_test();
 end

 logic cov;
 initial cov = 0;
 always #100 cov = !cov;

endmodule
$$(endfile)
```

trans

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
```

```
$$(file) $$("trans.svh")$$(foreach) $$(struct)
class pkt_$$(i.name) extends ovm_transaction;
  $$(foreach) $$(var)rand $$(i.type) $$(i.name);
  /*constraint $$(i.name)_range {
    $$(i.name) dist { [0:1] };
  }*/
  $$(endfor)

  function new();
    default_transaction();
  endfunction

  function void default_transaction();
  $$(foreach) $$(var)
    $$(i.name) = 0;
  $$(endfor)
  endfunction



  function ovm_object clone();
    pkt_$$(i.name) t1= new;
    t1.copy(this);
    return t1;
  endfunction

  function void copy(input pkt_$$(i.name) t);
    $$(foreach) $$(var)
    this.$$(i.name) = t.$$(i.name);
    $$(endfor)
  endfunction

  function bit comp(input pkt_$$(i.name) t);

return ($$(foreach)$$(var)(t.$$(i.name)==this.$$(i.name))$$(if)$$(isnotlast)&&
$$(endif)$$(endfor));
  endfunction

 function string psprint();
   string s;
```

```
   $sformat(s,"($$(foreach)$$(var)$$(i.name)_data=          %b$$(if)$$(isnotlast),$$(endif)$$(endfor))",
$$(foreach)$$(var)$$(i.name)$$(if)$$(isnotlast),$$(endif)$$(endfor));
    return s;
  endfunction
  function bit read(int file);
    if (file && !$feof(file))
      read = 0 != $fscanf(file, "$$(foreach)$$(var)%d $$(endfor)",
     $$(foreach)$$(var)this.$$(i.name)$$(if)$$(isnotlast),$$(endif)$$(endfor));
      else read = 0;
   endfunction
  function void do_record (ovm_recorder recorder);
     $$(foreach)      $$(var)recorder.record_field      ("$$(i.name)",      $$(i.name),      $bits($$(i.name)),
OVM_HEX);$$(endfor)
  endfunction
  $$(foreach) $$(var)
  function void set_$$(i.name)($$(i.type) $$(i.name)_t);
    this.$$(i.name) = $$(i.name)_t;
  endfunction

  function $$(i.type) get_$$(i.name)();
    return this.$$(i.name);
  endfunction
  $$(endfor)

endclass
 $$(endfor)

$$(endfile)
```

Trans_bir

```
$$(type.map) $$(bool->bit)
$$(type.map) $$(char->byte)
$$(type.map) $$(short->shortint)
$$(type.map) $$(short int->shortint)
$$(type.map) $$(int->int)
$$(type.map) $$(long->int)
$$(type.map) $$(long int->int)
$$(type.map) $$(long long->longint)
```

```
$$(type.map) $$(long long int->longint)
$$(type.map) $$(float->shortreal)
$$(type.map) $$(double->real)
$$(type.map) $$(signed->int)
$$(type.map) $$(unsigned->int unsigned)
$$(type.map) $$(signed int->int)
$$(type.map) $$(unsigned int->int unsigned)
$$(type.map) $$(signed short->shortint)
$$(type.map) $$(signed short int->shortint)
$$(type.map) $$(unsigned short->shortint unsigned)
$$(type.map) $$(unsigned shortint->shortint unsigned)
$$(type.map) $$(signed long->int)
$$(type.map) $$(signed long int->int)
$$(type.map) $$(unsigned long->int)
$$(type.map) $$(unsigned long int->int)
$$(type.map) $$(signed long long->longint)
$$(type.map) $$(signed long long int->longint)
$$(type.map) $$(unsigned long long->longint)
$$(type.map) $$(unsigned long long int->longint)
$$(file) $$("trans.svh") $$(foreach) $$(struct)
class pkt_$$(i.name) extends ovm_transaction;
 //Please put all tasks of the module
 typedef enum {IDLE, WRITE, READ} bus_trans_t;
 rand bus_trans_t bus_trans_type;
 $$(foreach) $$(var) rand $$(i.type) $$(i.name);
  /*constraint $$(i.name)_range {
    $$(i.name) dist { [0:1] };
  }*/
  $$(endfor)
  function new();
    default_transaction();
  endfunction
 //Change default transaction case necessary
  function void default_transaction();
    bus_trans_type = IDLE;
  $$(foreach) $$(var)
    $$(i.name) = 0;
  $$(endfor)
  endfunction
```

```
  function ovm_object clone();
    pkt_$$(i.name) t1= new;
    t1.copy(this);
    return t1;
  endfunction

  function void copy(input pkt_$$(i.name) t);
   $$(foreach) $$(var)
    this.$$(i.name) = t.$$(i.name);
    $$(endfor)
  endfunction
  function bit comp(input pkt_$$(i.name) t);

return ($$(foreach)$$(var)(t.$$(i.name)==this.$$(i.name))$$(if)$$(isnotlast)&&
$$(endif)$$(endfor));
  endfunction

 //function to string. Change for string all data

 function string psprint();
   string s;
   $sformat(s,"($$(foreach)$$(var)$$(i.name)_data=          %b$$(if)$$(isnotlast),$$(endif)$$(endfor))",
$$(foreach)$$(var)$$(i.name)$$(if)$$(isnotlast),$$(endif)$$(endfor));
   return s;
 endfunction
   function bit read(int file);
   if (file && !$feof(file))
     read      =      0      !=      $fscanf(file,      "$$(foreach)$$(var)%d      $$(endfor)",
$$(foreach)$$(var)this.$$(i.name)$$(if)$$(isnotlast), $$(endif)$$(endfor));
     else read = 0;
  endfunction
 function void do_record (ovm_recorder recorder);
    $$(foreach)      $$(var)recorder.record_field      ("$$(i.name)",      $$(i.name),      $bits($$(i.name)),
OVM_HEX);$$(endfor)
 endfunction
 //----------------------------------------------------------------
 // accessor functions
 //
```

```
  // A collection of accessor functions.  We recommend you use these to
  // avoid creating dependencies on the specific structure of the
  // transaction object.
  //-----------------------------------------------------------------
  // if the tasks change, remove the line above until the function set_read, else uncomment this part of the
code
/*  function bit is_idle;
    return (bus_trans_type == IDLE);
  endfunction

  function bit is_write;
    return (bus_trans_type == WRITE);
  endfunction

  function bit is_read;
    return (bus_trans_type == READ);
  endfunction

  function void set_idle();
    bus_trans_type = IDLE;
  endfunction

  function void set_write();
    bus_trans_type = WRITE;
  endfunction

  function void set_read();
    bus_trans_type = READ;
  endfunction
*/
  $$(foreach) $$(var)
  function void set_$$(i.name)($$(i.type) $$(i.name)_t);
    this.$$(i.name) = $$(i.name)_t;
  endfunction

  function $$(i.type) get_$$(i.name)();
    return this.$$(i.name);
  endfunction
  $$(endfor)
```

```
endclass
$$(endfor)
$$(endfile)
```