UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

**Lucas Eliseu Gonçalves de Arruda**

# IMPLEMENTAÇÃO DE GERADOR DE TESTBENCH UVM

CAMPINA GRANDE

**2019**

LUCAS ELISEU GONÇALVES DE ARRUDA

# IMPLEMENTAÇÃO DE GERADOR DE TESTBENCH UVM

**Trabalho de Conclusão de Curso submetido à Universidade Federal de Campina Grande, como requisito necessário para obtenção do grau de Bacharel em Engenharia Elétrica**

Campina Grande, novembro de 2019

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

LUCAS ELISEU GONÇALVES DE ARRUDA

Esta Monografia foi julgada adequada para a obtenção do título de Bacharel em Engenharia de Elétrica, sendo aprovada em sua forma final pelo orientador:

Prof. Dr. Marcos Ricardo Alcântara Morais
Universidade Federal de Campina Grande

Campina Grande, 18 de novembro de 2019

# Agradecimentos

Inicialmente agradeço aos meus pais, que sempre me apoiaram em todas as minhas escolhas e me deram as melhores oportunidades de aprendizado possível. Vejam só estou me tornando um engenheiro eletricista e não um juiz!

Agradeço a Sara por estar sempre presente, mesmo em momentos de dificuldades, e me fazer levantar a cabeça quando eu não acreditava em mim. Obrigado, bem!

Em seguida, sou grato pela maravilhosa iniciativa dos professores Gutemberg e Marcos Morais pela criação do Laboratório de Excelência em Microeletrônica no Nordeste (XMEN). Esta experiência que eu tive a oportunidade de vivenciar além de desafiadora, foi inspiradora e enraizou laços de amizade com os seus membros que irei carregar para o resto da vida.

Por falar neles, muito obrigado pelos momentos de riso, raiva e gritaria que passamos juntos!

Finalmente, agradeço a Idea! por ter me oferecido todos os meios para realização deste trabalho.

*"It always seems impossible until it's done."*
*(Nelson Mandela)*

# Resumo

Pertencente a um ambiente com tempo de mercado restrito, a área da microeletrônica é constantemente desafiada para entrega de dispositivos nesses prazos limitados. Esse fato é ainda mais verdadeiro para o setor de verificação, que comumente ocupa 70% do tempo de projeto.

Nesse contexto, o gerador de *testbench* UVM é desenvolvido como ferramenta de automatização para auxílio dos verificadores tanto no início de verificação de IPs quanto para realização de alterações de *design*.

**Palavras-chave**: Python, UVM, Verificação Funcional, Microeletrônica

# Abstract

Inserted in a restrict time-to-market environment, the microelectronics area is constantly challenged to deliver devices in this limited dedlines. This fact is even stronger for the verification sector, which usually accounts for 70% of the project duration.

In this context, the UVM testbench generator is developed as an automation tool to help engineers on both IP verification kickoff and to account for design modifications

**Keywords**: Python, UVM, Functional Verification, Microelectronics

# Lista de ilustrações

# Sumário

# 1 Introdução

A microeletrônica é um dos ramos da eletrônica que mais contribui com os avanços tecnológicos presenciados atualmente. É neste setor onde se desenvolvem tanto circuitos digitais como portas lógicas e registradores, quanto circuitos analógicos como amplificadores de sinal e conversores analógico-digitais/digital-analógicos. Esses circuitos são frequentemente integrados em larga escala (*very large scale integration - VLSI*) em *chips* para a formação de circuitos mais complexos como unidades de processamento centrais (*central unit processors* - CPUs), processadores digitais de sinais (*digital signal processors - DSPs*) e sistemas-em-um-chip (*systems-on-a-chip - SoC*).

Esses circuitos mais complexos vêm sendo aprimorados e vêm apresentando uma complexidade cada vez maior, representando um grande desafio para os engenheiros da área. Esses profissionais precisam desenvolver *chips* visando resolver problemas que apresentem um mercado-alvo interessante e ainda administrar corretamente o tempo de desenvolvimento dos projetos de modo a concluí-los de forma efetiva, livre de falhas e no momento adequado para o mercado.

No fluxo de produção de *chips*, é comum dividir o projeto em quatro etapas: modelagem, *design*, verificação e *backend*. Na modelagem, realiza-se uma implementação do projeto primando pela funcionalidade do conjunto, normalmente utilizando linguagens de programação com maior grau de abstração como *Python* e *MATLAB*. A arquitetura digital do sistema normalmente é definida no *design*, junto à codificação em linguagens de descrição de hardware (HDL) dos IP-Cores (*intellectual property cores*). É no *backend* onde os circuitos descritos em HDL são convertidos em transistores e são fisicamente implementados. Em paralelo a essas etapas, o funcionamento dos blocos e do sistema é assegurado pela verificação normalmente por meio de simulações funcionais, que serão explicadas ao longo deste trabalho.

O desenvolvimento de circuitos digitais só se torna factível devido à existência de linguagens de descrição de hardware (HDL) como *SystemVerilog* e VHDL, que permitem que os desenvolvedores descrevam a estrutura e o comportamento dos circuitos digitais, e das diversas ferramentas de projetos de circuito integrados (*electronic design automation - EDA*), que podem apresentar diversas funcionalidades como as listadas a seguir:

- Simulação funcional;
- Síntese lógica;
- Análise de tempo;

- Análise de potência dissipada;
- Roteamento

No âmbito da verificação de circuitos digitais, diversas metodologias podem ser empregadas para garantir a integridade tanto dos IPs individualmente quanto do sistema como um todo. São elas [Rashinkar, Paterson e Singh 2002]:

- Análises estáticas;
- Verificação formal;
- Verificação física;
- Verificação por meio de simulações;

Análises estáticas compreendem checagem de código via *lint*, que realizam conferências da sintaxe das descrições de *hardware* de interesse, e análises estáticas de tempo, nas quais verificam-se se as restrições de tempo das células, como tempo de *setup* e *hold*, são respeitadas. Na verificação formal, modelos matemáticos dos circuitos são utilizados com intuito de provar sua correção. Normalmente realizada durante o *backend*, a verificação física consiste em analisar os possíveis problemas decorrentes da materialização das células e fios que as conectam, que podem ser *crosstalk*, eletromigração, efeitos de antena, dentre outros.

A verificação por meio de simulações, que é o foco deste trabalho, consiste em inserir o dispositivo a ser verificado (*device under test* - DUT) em um ambiente virtual denominado *testbench* (TB), responsável por enviar sinais ao DUT para que se possa analisar suas respostas. Essa análise comumente lança mão do uso de modelos do DUT descritos em outras linguagens como C, *MATLAB* e *Python*, que passam a receber os mesmos estímulos e servem como referência para o comportamento esperado do DUT. Essa metodologia de verificação também é denominada verificação funcional, pois visa assegurar que o IP apresenta funcionamento correto dadas as entradas que lhe são enviadas.

Como mencionado anteriormente, os profissionais da microeletrônica estão sob constante requisição das demandas de mercado, e por isso precisam realizar suas atividades da forma mais eficaz possível de modo a conseguir entregar produtos em perfeito estado de funcionamento a tempo de mercado. Esse requerimento é ainda mais acentuado para a equipe de verificação, pois normalmente atribui-se aproximadamente 70% do tempo de projeto a este setor [Rashinkar, Paterson e Singh 2002], [Bergeron 2002].

Além disso, é comum que IPs ou subsistemas passem por modificações ao longo do projeto, desde simples renomeações de sinais de interfaces à reestruturação dos algoritmos implementados por eles. No contexto de verificação funcional, essas alterações precisam ser

levadas em conta para atualização do *testbench* e muitas vezes são trabalhos repetitivos que poderiam facilmente ser automatizados.

Como forma de padronizar os ambientes de teste, a Accellera© desenvolveu em 2009 uma metodologia de verificação funcional denominada UVM (*universal verification methodology*). Essa biblioteca de *SystemVerilog* de código aberto é equipada com diversas classes de enorme flexibilidade que permitem o desenvolvimento de VIPs (IPs de verificação), que são blocos autocontidos que podem ser inseridos no *testbench* para simular o comportamento e enviar sinais para o DUT.

Apesar da UVM diminuir a complexidade de geração de ambientes de teste e melhorar a comunicação entre os envolvidos no projeto devido à padronização dos blocos de verificação, os *testbenchs* em UVM ainda ficam susceptíveis aos problemas citados anteriormente, isto é, mudanças no design precisam ser refletidos em mudanças no ambiente.

Nesse contexto, o objetivo deste trabalho é de desenvolver um programa em *Python* capaz de gerar *testbenchs* automaticamente de modo a otimizar o trabalho dos verificadores para evitar o retrabalho tanto no início da verificação de IPs quanto ao longo de modificações de interface que os blocos a serem verificados possam passar.

Este trabalho foi desenvolvido em parceria com a empresa Idea! Electronic Systems, que possui seus próprios métodos para extração de dados de planilhas. O gerador de *testbench* UVM foi criado tendo como base estes métodos, que serão mencionados ao longo do texto como *parser*.

# 2 Embasamento Teórico

Para melhor contextualização sobre o trabalho da verificação na área da microeletrônica e para evidenciar os termos utilizados ao longo do texto, serão explicados conceitos fundamentais do setor. Em seguida, serão abordados conceitos importantes para compreensão da UVM, sendo discutida logo após a arquitetura do *testbench* em UVM a ser utilizada pelo gerador.

## 2.1 Conceitos Gerais

### 2.1.1 Linguagem de Descrição de *Hardware* (HDL)

As HDLs são linguagens de programação utilizadas principalmente para descrever o comportamento e a estrutura de circuitos eletrônicos, comumente circuitos digitais. Esta descrição pode apresentar dois níveis de abstração: o nível lógico e o nível de transferência de registros (*register transfer level* - RTL).

No nível lógico, o circuito é descrito detalhadamente levando em consideração cada porta lógica e é impraticável de ser realizado quando a complexidade dos componentes aumenta. Já em RTL, os dados são processados no caminho de dados (*datapath*), sendo submetidos a blocos que abstraem o detalhamento das portas lógicas como registradores, somadores e multiplexadores. Isso facilita o desenvolvimento de lógicas complexas constituídas de milhões de transistores e diminui consideravelmente a probabilidade de erros, pois não se faz necessário analisar todas as portas lógicas individualmente.

Essas linguagens também apresentam suporte para controle de simulações pois explicitamente inclui noção de tempo, isto é, tornam possível a criação de ambientes virtuais (*testbenchs*) nos quais estímulos podem ocorrer ao longo do tempo simulado.

No âmbito dos circuitos digitais, as HDLs mais utilizadas são o *systemverilog* e o VHDL.

### 2.1.2 Verificação Funcional

O processo de desenvolvimento de IPs digitais é composto por diversas etapas. Inicialmente, existe uma especificação de seu funcionamento, podendo conter restrições de consumo de energia e temperaturas de funcionamento. Em seguida, estas orientações podem ser implementadas em linguagens de alto nível como Python ou MATLAB, que apresentam precisão infinita e podem abstrair as restrições impostas na especificação.

A transição de modelo em alto nível para implementação em hardware implica a aplicação de diversos compromissos como a diminuição de bits de precisão, a inclusão de níveis de *pipeline* para atingir critérios de temporização e a simplificação de circuitos para redução do tamanho do circuito para atingir critérios de área por exemplo.

Nesse contexto, a verificação funcional pode ser definida como a demonstração de que a intenção de um *design* é mantida em sua implementação [Piziali 2004].

Ela é estruturada por um plano de verificação responsável por determinar três aspectos principais:

- **Análise de cobertura**: Determina a extensão dos testes e os casos especiais em que o *design* precisa ser submetido para validação;
- **Geração de estímulos**: Define que tipos de estímulos precisam ser enviados ao dispositivo de modo a garantir condições de funcionamento e fechamento de cobertura;
- **Comparação dos resultados**: Estabelece os critérios para avaliação da funcionalidade do DUT.

### 2.1.3 *Transaction Level Modeling* (TLM)

Um dos pontos para aumentar a produtividade em verificação é de trabalhar com níveis de abstração que façam sentido [Accellera 2015], isto é, não faz sentido ter que implementar todos os sinais de algum protocolo para realizar ações simples. Uma leitura em SPI, por exemplo, poderia ser representada apenas pelo dado, endereço e modo de operação (leitura).

A UVM dispõe de componentes capazes de expressar esse nível de abstração, que são as transações (*sequence_ items*) e as sequências (*sequences*). Esses elementos são detalhados ao longo do texto.

## 2.2 Arquitetura do *Testbench* em UVM

Como mencionado anteriormente, a UVM estabelece uma arquitetura típica para os *testbenchs* que está apresentado na Figura 1.

Esse formato normalmente foi tomado como base para geração de um ambiente que se adequasse às necessidades da Idea!, resultando a arquitetura apresentada na Figura 2. Os sinais de entrada tanto o DUT quanto o modelo são exercitados pela mesma interface, que é controlada pelo *driver*. No entanto, os sinais de resposta desses dois elementos são observados em interfaces diferentes. Isso garante que ambos estão recebendo os mesmos estímulos e que os *monitors* conseguem captar as respostas de cada um independentemente.

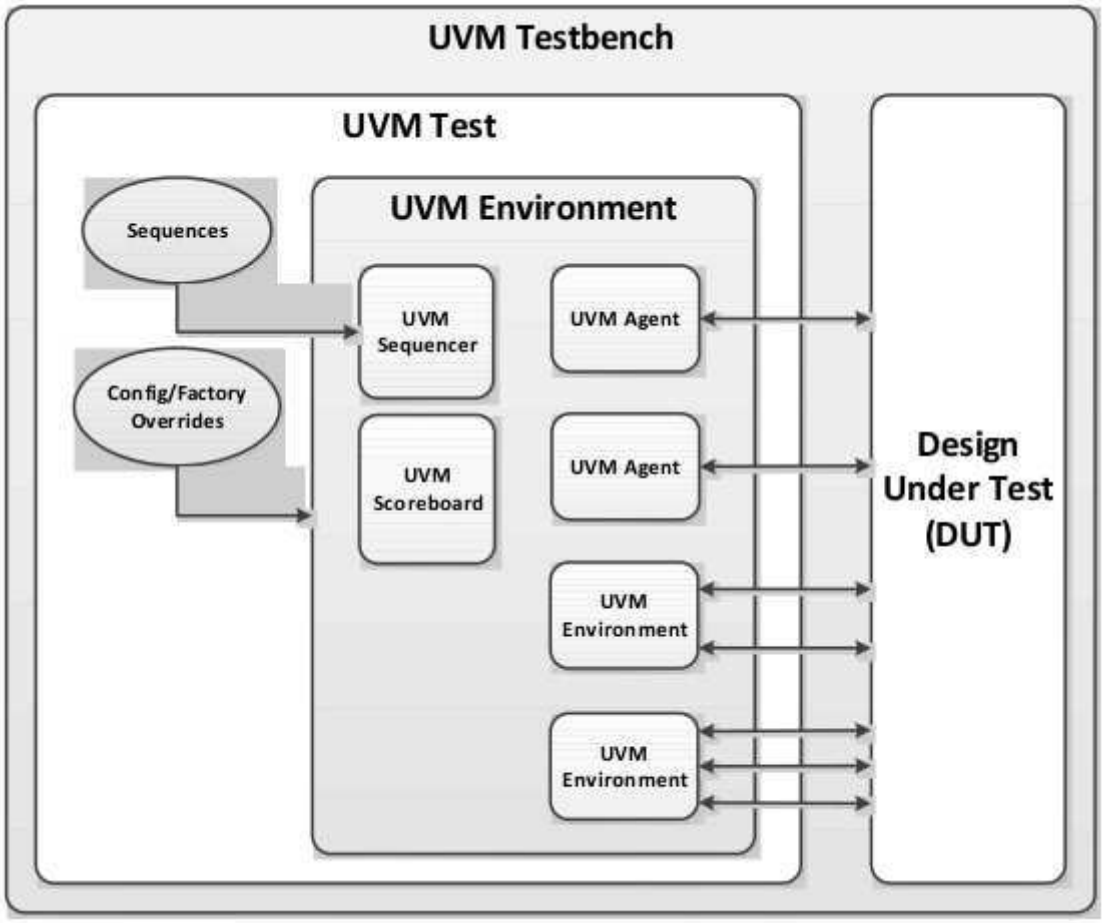



Figura 1 – Arquitetura típica de um *testbench* UVM. Fonte: Accellera 2015

Além disso, são criados *agents*, *scoreboards* e *interfaces* para cada domínio de *clock*.

Os blocos que compõem o *testbench* são descritos a seguir

## 2.3 Conceitos de UVM

A HDL SystemVerilog foi desenvolvida baseada principalmente em Verilog de modo a permitir tanto a descrição de hardware quanto a criação de *testbenchs* utilizando a mesma linguagem. Dentre as melhorias que essa linguagem oferece, vale citar:

- Programação Orientada a Objeto (POO);
- Múltiplos tipos de dados (real, int, bit, logic e tipos definidos por usuário);
- Construção de interfaces;
- Vetores dinâmicos;

Apesar das vantagens dessa linguagem, não foram definidas orientações de sua utilização no âmbito da verificação, deixando a cargo dos verificadores determinarem a

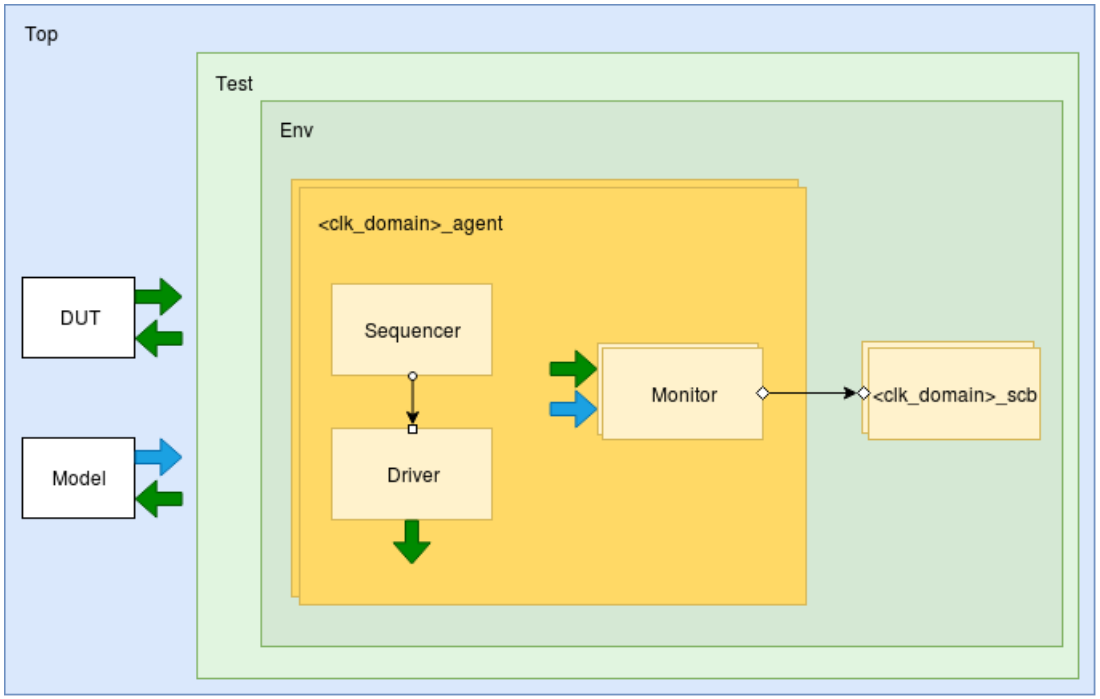



Figura 2 – Arquitetura do *testbench* deste trabalho

forma de utilizá-la. Isso implicou o desenvolvimento dos mais diversos tipos de *testbenchs*, sem necessariamente ocorrer propagação de boas práticas de uso da linguagem.

Nesse contexto, diferentes fornecedores de EDA desenvolveram de forma independente bibliotecas e orientações de SystemVerilog para verificação, como a VMM ( *Verification Methodology Manual*). Essa iniciativa já agregou uma certa padronização para os *testbenchs*, mas ainda era dependente das EDA utilizadas.

Esse problema foi resolvido quando em 2009 a Accellera© criou um subcomitê técnico para criar uma metodologia aberta que pudesse ser utilizada na maioria das ferramentas disponíveis. Essa iniciativa, junto com contribuições dos fornecedores para garantir interoperabilidade culminou na criação da *Universal Verification Methodology* (UVM).

A UVM proporciona tanto as orientações de como criar *testbenchs* padronizados, adotando conceitos de POO, reuso e flexibilidade, como também uma ampla biblioteca, com definições dos diversos elementos presentes no ambiente como *drivers*, monitores e agentes, sendo a maioria deles derivados de uma classe comum.

Para melhor compreensão do ambiente a ser gerado neste trabalho, alguns conceitos gerais de UVM precisam ser melhor descritos, como seus componentes constituintes e as fases.

### 2.3.1 Fases

As fases de UVM são o mecanismo que controla os componentes dos *testbenchs*, desde suas criações e conexões até o final da simulação, quando é possível gerar relatórios e análise de dados.

É apresentada na Figura 3 o diagrama de classes das fases predefinidas de UVM. Em amarelo estão apresentadas as fases comuns (*UVM Common Phases*), que são ocorrem de forma síncrona para todos os componentes UVM. Já em rosa, estão mostradas as fases de tempo de execução (*UVM Run-Time Phases*), que podem ocorrer assincronamente entre os componentes.

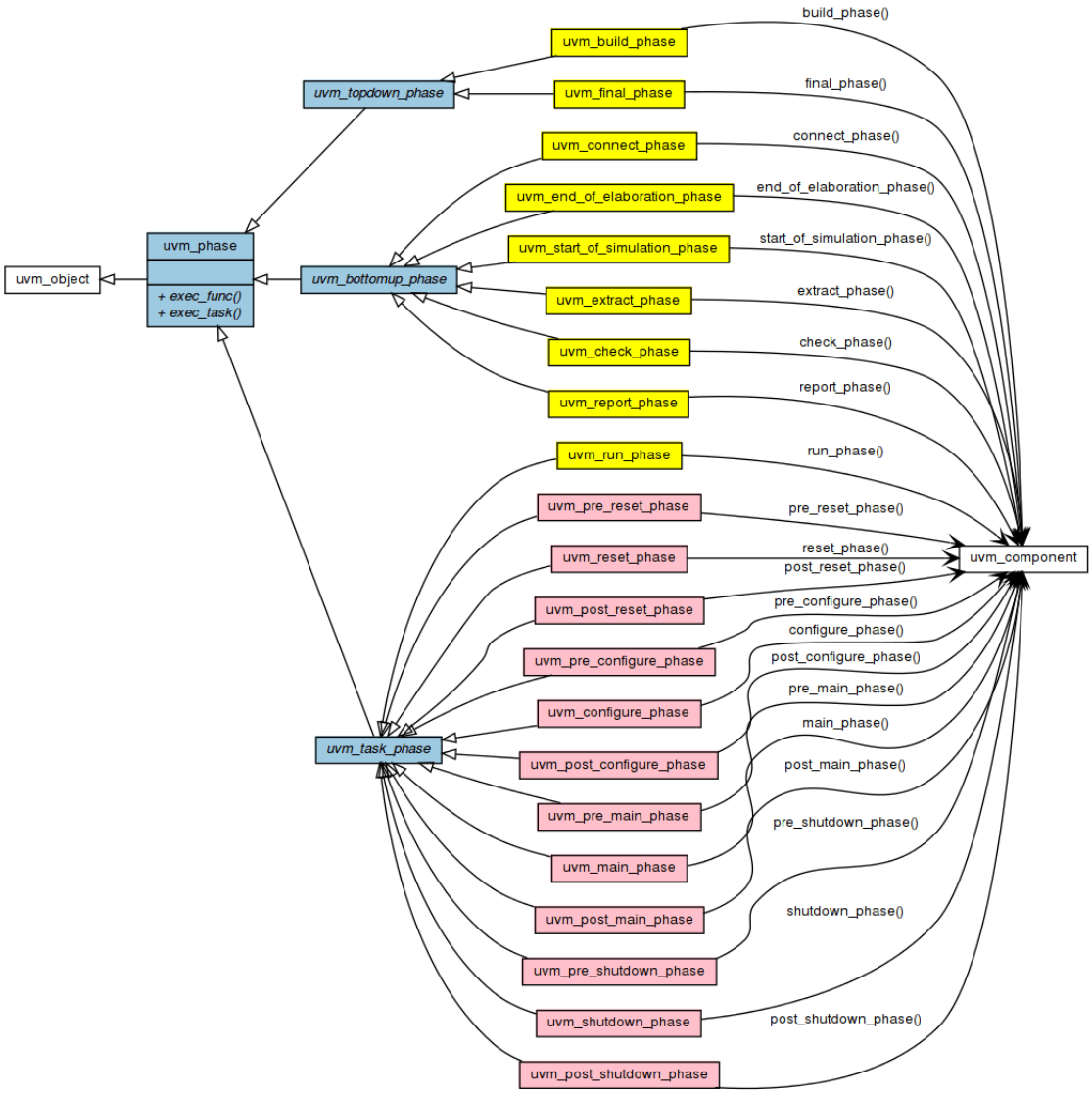



Figura 3 – Diagrama de classes para as fases de UVM. Fonte: http://cluelogic.com/2014/08/uvm-tutorial-for-candy-lovers-phasing/, acesso em nov. 2019

As *Common Phases* são apresentadas a seguir. Elas estão dispostas na ordem de chamada definida pelo UVM Class Reference. Dentre elas, a *build_phase* e a *final_phase*

são as únicas chamadas do topo para base, isto é, a fase do topo é a primeira a ser chamada, seguida a cada hierarquicamente pelas instâncias, de modo recursivo. As demais ocorrem da forma inversa.

1. ***Build phase***: Cria e configura a estrutura do *testbench*.
2. ***Connect phase***: Realiza as conexões entre os componentes.
3. ***End of elaboration phase***: Permite uma sintonia fina do ambiente após a elaboração.
4. ***Start of simulation phase***: Prepara o ambiente para simulação.
5. ***Run phase***: Estimula o DUT.
6. ***Extract phase***: Extrai dados de diferentes pontos do *testbench*
7. ***Check phase***: Checa condições não esperadas do ambiente
8. ***Report phase***: Gera relatórios sobre o teste
9. ***Final phase***: Resolve pendências finais do *testbench*

As *Run-Time Phases* também apresentam uma ordem de chamada definida, que é a mesma que elas estão dispostas a seguir.

1. ***Pre reset phase***: Permite ajustes prévios ao reset.
2. ***Reset phase***: Permite execução de comandos durante o reset.
3. ***Post reset phase***: Permite ajustes após execução do reset
4. ***Pre configure phase***: Prepara o DUT para receber configurações.
5. ***Configure phase***: Configura o DUT.
6. ***Post configure phase***: Permite ajustes finos após configuração do DUT.
7. ***Pre main phase***: Prepara o ambiente antes de enviar os estímulos principais.
8. ***Main phase***: Envia os estímulos principais ao DUT.
9. ***Post main phase***: Permite ajustes finos após termino dos estímulos.
10. ***Pre shutdown phase***: Prepara o ambiente para desligamento do DUT.
11. ***Shutdown phase***: Realiza lógicas no desligamento do DUT.
12. ***Post shutdown phase***: Realiza últimas ações que requeiram que o *testbench* esteja em tempo de execução.

### 2.3.2 *Database* de Configuração

A *database* de configuração (config_db) é uma mecanismo de compartilhamento de dados da UVM. É implementada como uma *look-up table* acessada por meio de chaves e permite que componentes em diferentes hierarquias possam compartilhar variáveis entre si.

Um exemplo clássico do uso da config_db é o compartilhamento das interfaces, que são instanciadas no topo do *testbench* e precisam ser utilizadas em elementos na base da hierarquia da UVM, como os *drivers* e *monitors*.

### 2.3.3 Tipos de Classes

A UVM apresenta uma vasta quantidade de classes, a maior parte sendo derivadas de uma classe comum (uvm_object), e em geral elas podem ser classificadas em duas categorias: dados e estrutura. As classes categorizadas como dados apresentam natureza transiente, isto é, elas podem ser criadas a qualquer momento da simulação, mesmo em tempo de execução, e são responsáveis por transmitir os estímulos ao longo do *testbench*. Classes desse tipo são derivadas da classe uvm_transaction. Já no caso das estruturais, são classes que são criadas na build_phase e existem durante toda simulação. São derivadas da classe uvm_component e podem ser acessadas hierarquicamente da mesma maneira que módulos e programas de SystemVerilog.

É apresentado na Figura 4 um diagrama de classes de UVM simplificado.

### 2.3.4 *Sequence Item*

O *sequence item* é a transação que pode ser enviada para o DUT através do *driver* ou que é obtida a partir do *monitor*. É constituída principalmente por campos contendo os dados a ser enviados ou recebidos do DUT, mas também pode conter a definição de funções de utilidade para o *testbench*, como métodos de cópia, comparação e impressão, além de poder aplicar restrições aos valores que os dados podem assumir.

### 2.3.5 *Sequence*

A *sequence* é o elemento responsável por gerar e ordenar a sequência de transações que alimenta o *testbench*. Essa classe é categorizada como um dado e apresenta um mecanismo de *handshake* que opera em conjunto com o *sequencer* e o *driver*.

Esse mecanismo de comunicação é apresentado na Figura 5. A sequência inicia o *handshake* com a função start_item(), que fica bloqueada até o *driver* executar a get_next_item(), que por sua vez fica bloqueada aguardando um finish_item() advindo da *sequence*. Uma vez que essa comunicação inicial é feita, a *sequence* opera sobre a transação conforme desejado pelo verificador (podendo aleatorizar os dados, realizar

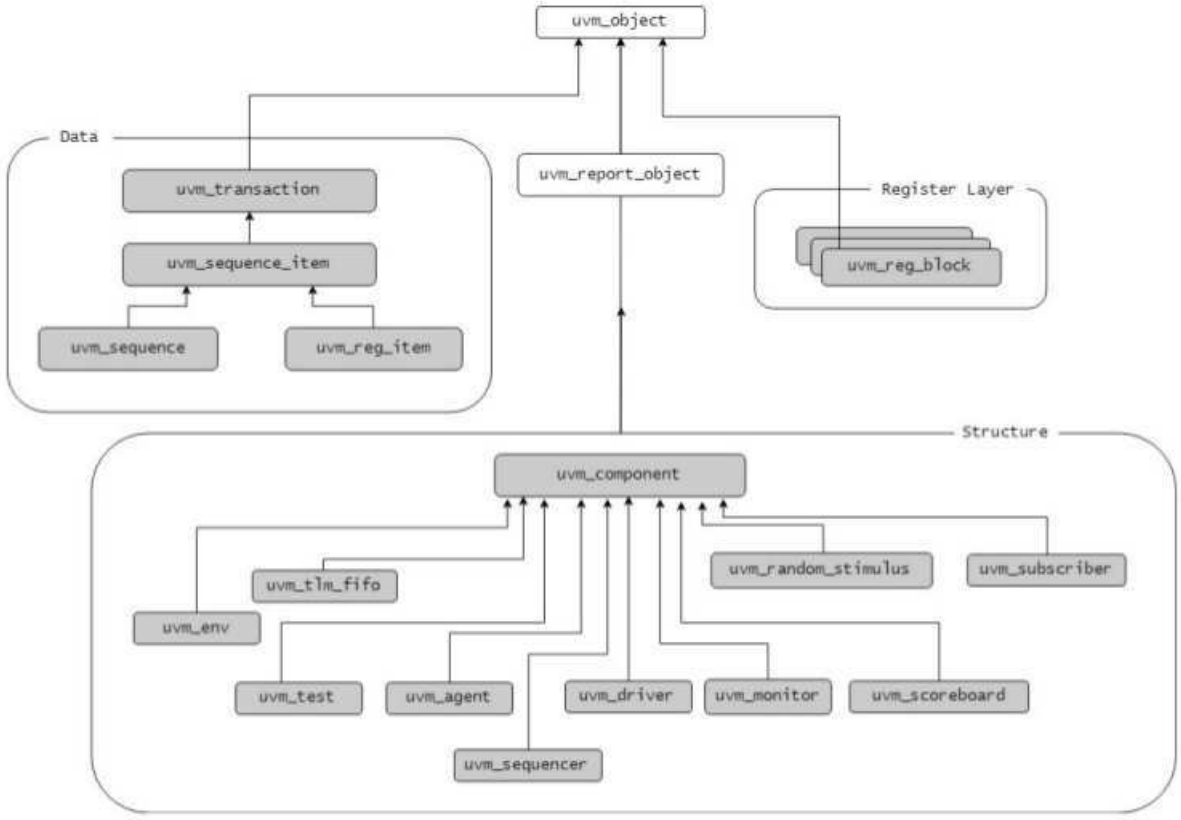



Figura 4 – Diagrama de classes de UVM simplificado. Fonte: Carvalho 2018

configurações, etc) e em seguida chama o finish_item(), que desbloqueia o *driver* para poder aplicar os estímulos diretamente ao DUT. Enquanto esse elemento não chama a função item_done(), a sequência permanece bloqueada. A UVM permite que a sequência obtenha uma resposta do *driver*, como mostrado na Figura 6.

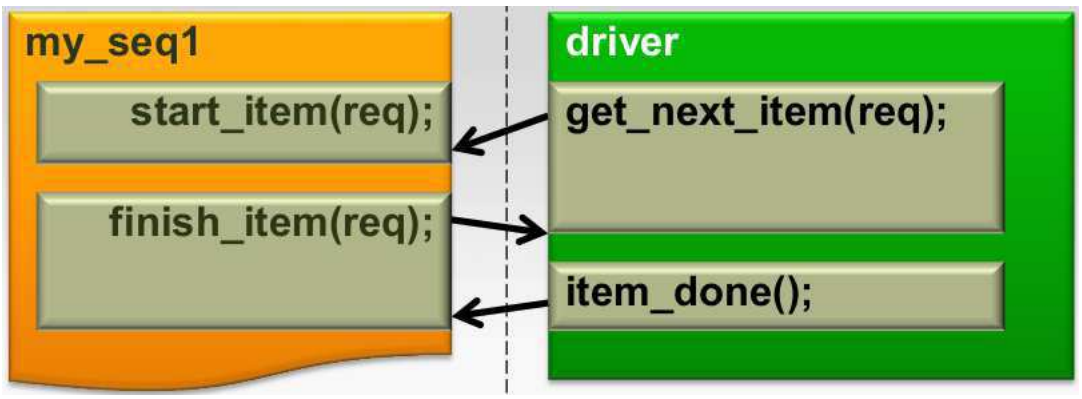



Figura 5 – *Handshake* entre *sequence* e *driver* sem resposta do *driver*. Fonte: https://verificationacademy.com/sessions/proper-care-and-feeding-sequences acesso em nov. 2019

Esse processo de comunicação é arbitrado pelo *sequencer*, que distribui as transações para o driver de modo que todas elas sejam atendidas e que o driver receba apenas uma

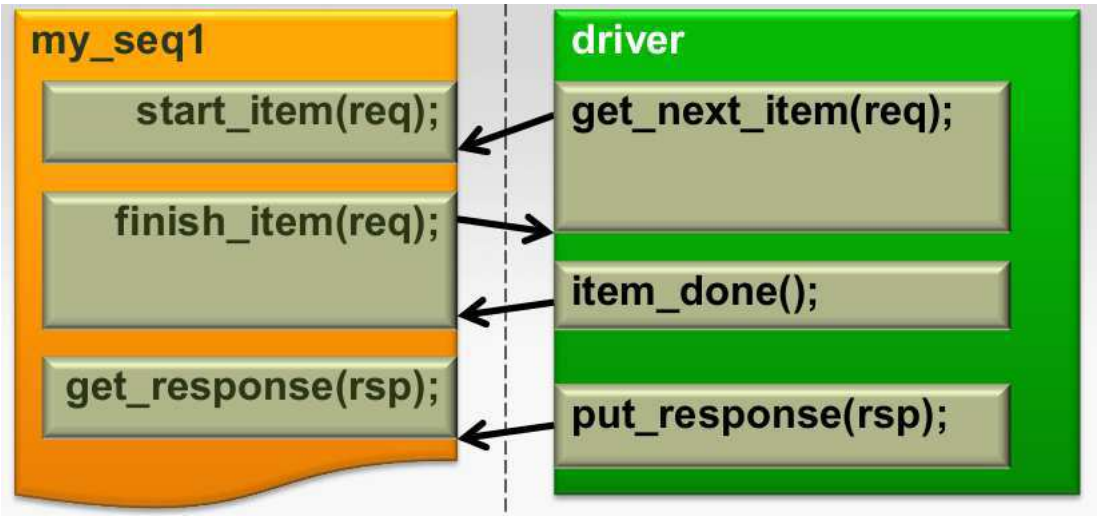



Figura 6 – *Handshake* entre *sequence* e *driver* com resposta do *driver*. Fonte: https://verificationacademy.com/sessions/proper-care-and-feeding-sequences acesso em nov. 2019

transação por vez.

### 2.3.6 *Sequencer*

Como mencionado anteriormente, o *sequencer* é o elemento responsável por arbitrar as transações para o *driver*. Sua implementação em UVM é bastante robusto, de modo que normalmente o verificador não precisa se preocupar com essa arbitração pois ela já está devidamente implementada.

Por padrão, o esquema de arbitração das transações é o SEQ_ARB_FIFO, no qual as primeiras transações a chegarem ao *sequencer* serão as primeiras a ser enviadas ao *driver*, mas é possível alterar esse esquema para as opções SEQ_ARB_WEIGHTED, SEQ_ARB_RANDOM, SEQ_ARB_STRICT_FIFO, SEQ_ARB_STRICT_RANDOM E SEQ_ARB_USER. Como essas opções não são utilizadas neste trabalho, o leitor é aconselhado a consultar a especificação de UVM para obter mais detalhes a respeito delas.

Essa estrutura é classificada como ativa, pois está envolvida no envio de estímulos para o DUT.

### 2.3.7 *Driver*

O *driver* é responsável por converter as transações advindas da *sequence* sinais enviados para DUT, isto é, ele traduz informações de alto nível para estímulos que podem ser simples envios de dados até protocolos de comunicação complexos.

Assim como o *sequencer*, essa estrutura é classificada como ativa, pois está envolvida no envio de estímulos para o DUT.

### 2.3.8 *Monitor*

O *monitor* realiza a atividade inversa do *driver*, isto é, ele recebe os sinais do DUT e os converte para transações, que podem ser enviadas para o *scoreboard* para comparação com algum modelo.

Diferentemente do *driver* e do *sequencer*, essa estrutura é classificada como passiva, pois ela apenas recebe dados do DUT.

### 2.3.9 *Agent*

O *agent* é o elemento que engloba o(s) *driver(s)*, o(s) *sequencer(s)* e o(s) *monitor(s)*. Ele é responsável por criar essas estruturas em sua build_phase e realizar as devidas conexões, isto é, conectar os *sequencers* aos *drivers* e conectar as portas do(s) *monitor(s)* as suas.

A estrutura de *agent* adotada neste trabalho é constituída por um *driver*, um *sequencer* e um *monitor* para cada instância que houver no topo, isto é, se forem instanciados o DUT e um modelo de referência, dois *monitors* serão criados.

Um *agent* bem codificado apresenta a capacidade de ser instanciado de forma ativa ou passiva para ser incluso no *testbench* da maneira adequada ao verificador. Os *agents* ativos e passivos possuem em comum um *monitor*, e os ativos ainda apresentam um *driver* e um *sequencer*. Esses dois tipos são apresentados na Figura 7. Essa característica é interessante pois possibilita o reuso do componente em diversos ambientes, sem necessidade de um retrabalho.

### 2.3.10 *Scoreboard*

O *scoreboard* é o elemento que envolvido na determinação do êxito ou não dos testes. Esse componente pode variar bastante na implementação, podendo ser constituído de um modelo de referência interno e um comparador, ou apenas por uma lógica de comparação no caso de modelos de referência externados no *testbench*, que é o caso deste trabalho.

Independente da implementação, essa estrutura é capaz de contar a quantidade de erros e acertos entre DUT e modelo de referência e imprimi-los ao final da simulação, podendo associá-los a uma mensagem indicando o status final do teste.

### 2.3.11 *Environment*

O *environment* é o elemento que engloba o(s) *agent(s)* e o(s) *scoreboard(s)*. Ele é responsável por criar essas estruturas em sua build_phase e realizar as devidas conexões. A quantidade de componentes varia de acordo com o modo que o *testbench* está sendo

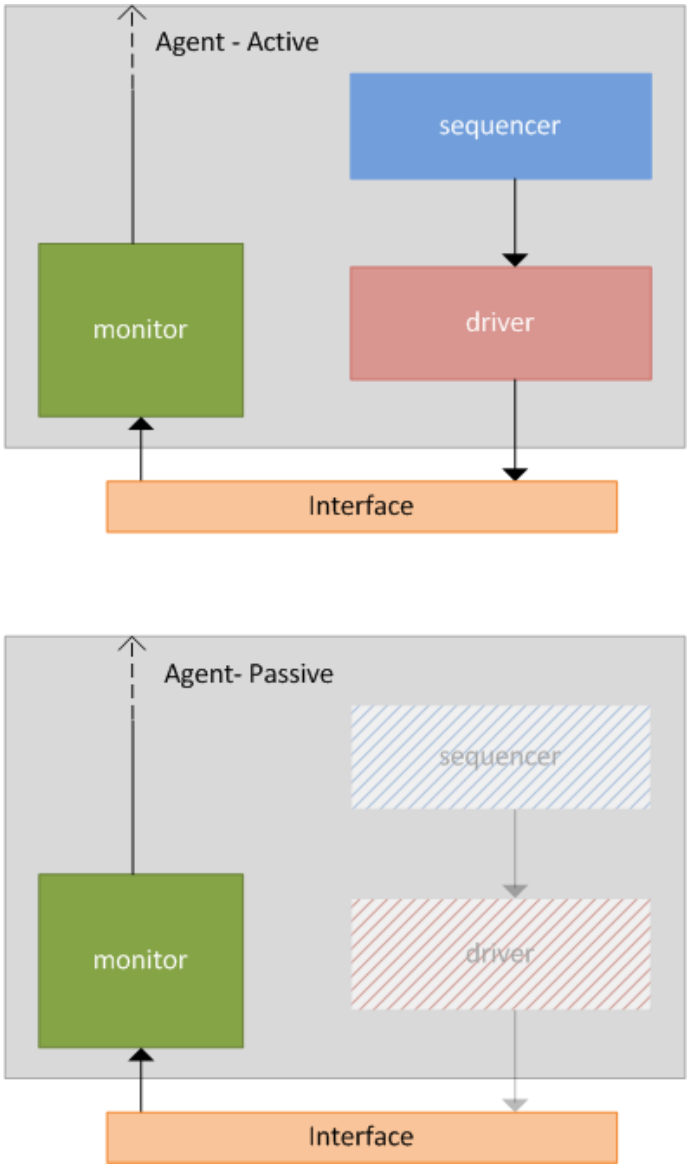



Figura 7 – Tipos de *agent*. Fonte: https://www.chipverify.com/blog/how-to-turn-an-agent-from-active-to-passive, acesso em nov. 2019

implementado. No caso deste trabalho são instanciados um *agent* e um *scoreboard* para cada domínio de clock do DUT.

No caso da implementação deste trabalho, o *environment* é o responsável por recuperar as interfaces da config_db e passá-las para seus sub-blocos.

### 2.3.12 *Test*

O *test* é o elemento de UVM no topo da hierarquia, e tem como funções instanciar e criar o *environment* e iniciar a(s) sequência(s).

# 3 Gerador de *Testbenchs* em UVM

Como discutido inicialmente, a verificação é uma das etapas mais demoradas no fluxo de microeletrônica digital, sendo de grande proveito o desenvolvimento de ferramentas de automatização de código para melhor adaptação dos *testbenchs* no caso de mudanças de *design*.

A estrutura do gerador é destrinchada a seguir. Vale salientar que alguns detalhes de implementação não serão completamente abordados por questões de segredo industrial, uma vez que este trabalho é desenvolvido em parceria com a empresa Idea!. Por simplicidade, o gerador também poderá ser mencionado como TbGen.

Os blocos a serem verificados são dispostos em planilhas nas quais cada sinal da interface está discriminado juntamente com as respectivas quantidades de bits. No caso de blocos que instanciam sub-blocos, as planilhas também indicam as conexões entre sinais da interface externa com os blocos internos, no entanto este tipo de circuito não é abordado neste trabalho.

A partir dessas planilhas, o *parser* disponibiliza para o gerador todos os sinais de interface também com suas quantidades de bits, e a partir desse ponto o gerador passa a criar hierarquicamente os objetos de sua estrutura. É apresentado nas Figuras 8 e 9 o diagrama de classes do gerador. Vale ressaltar que foram destacadas apenas as funções mais importantes por simplicidade de representação.

Esse diagrama é construído com inspiração na arquitetura de *testbench* implementada pelo gerador (Figura 2). Um objeto *tb* é criado recebendo como parâmetro um objeto do tipo *parser* que, como mencionado acima, contém todos os dados necessários para caracterizar os sinais do DUT. Este objeto é responsável por criar outros quatro: *o param_ obj*, que gera um arquivo contendo todas as informações relacionadas a quantidade de bits; o *pkg*, que cria um pacote de SystemVerilog incluindo todos os elementos UVM; o *dut*, que é um objeto que oferece funcionalidades auxiliares para o TbGen; o *interface*, que além de ter informações sobre os sinais, é responsável pela escrita das definições de *interface*, sendo gerado um objeto desse tipo para cada domínio de *clock*; e o *test*.

Esse último objeto além de escrever o *test*, também agrupa objetos *interface* em um dicionário para criar três objetos: o *sequence*, responsável pela escrita da sequência (uma por domínio de *clock*); o *transaction*, que escreve o *sequence_ item* (também uma para cada domínio de *clock*); e o *environment*.

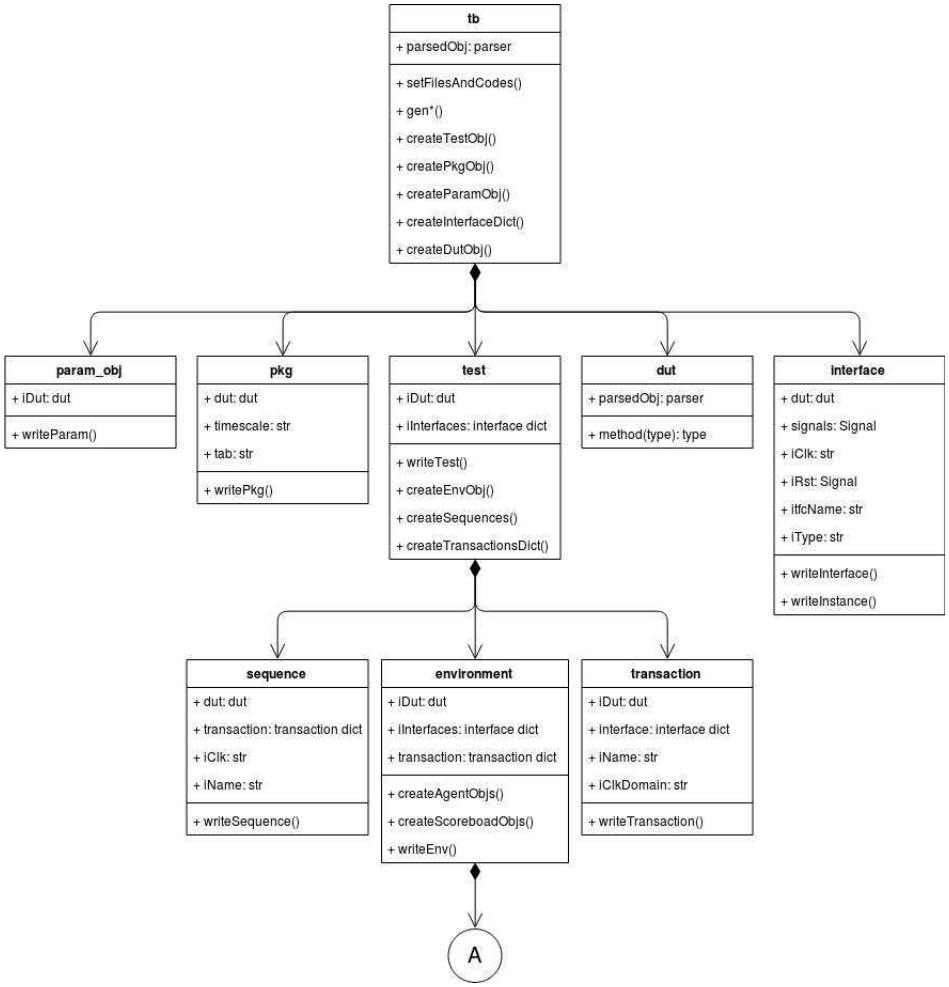



Figura 8 – Diagrama de classes do TBGen. Fonte: Autoria própria

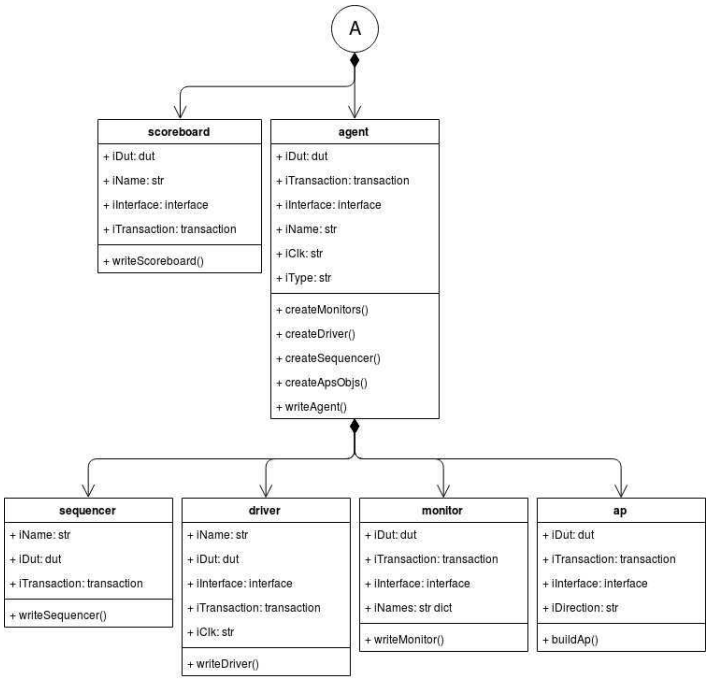



Figura 9 – Continuação do diagrama de classes do TBGen. Fonte: Autoria própria

O *environment* além de escrever seu componente homônimo, cria um dicionário com as *transactions* e utiliza o de *interfaces* que foi passado pelo *test* para criar objetos do tipo *scoreboard* e *agent*, também um para cada domínio de *clock*.

Finalmente, cada *agent* é responsável pela criação de quatro tipos de objetos: *sequencer*, *driver*, *monitor* e *ap* (*analysis port*).

Cada objeto criado contém um método de escrita write* do seu componente homônimo (writeTest, writeInterface, etc). Esses métodos são mapeados hierarquicamente no *tb* pelas funções gen* (genTb, genTest, genInterface, etc). Dessa forma, o tb pode executar a função setFilesAndCodes(), que retorna um dicionário contendo o nome de cada componente como chave e o texto equivalente aos componentes como valor. Isso permite que um programa que instancie o *tb* possa escrever os arquivos correspondentes a cada elemento.

Os códigos de componentes em UVM apresentam muitas funções que se repetem com frequência, contendo apenas alguns campos onde alterações são necessárias. Por esse motivo, foram criados *templates* de cada componente contendo indicadores especiais nas regiões a serem modificados. Esses indicadores são etiquetas definidas da seguinte forma: {:*etiqueta*:}. Os *templates* são carregados antes das chamadas das funções write*, que utilizam o método de Python str.replace('{:*etiqueta*:}', variável) para sobrescrever localmente o *template* e gerar o código desejado. Os códigos dessas estruturas estão apresentadas no Anexo A

Alguns elementos que apresentam códigos bastante mutáveis como as interfaces não utilizam *templates* para escrita, passando a ser escritos linha a linha nas funções write*.

# 4 Resultados

A estrutura explanada no capítulo 3 é o núcleo do TBGen, e é utilizada por um programa envelope chamado TBGenerator, que é responsável pela criação do *parser*, do *tb* e pela escrita dos arquivos. A partir dele então é possível gerar todo o ambiente de verificação por meio do seguinte comando:

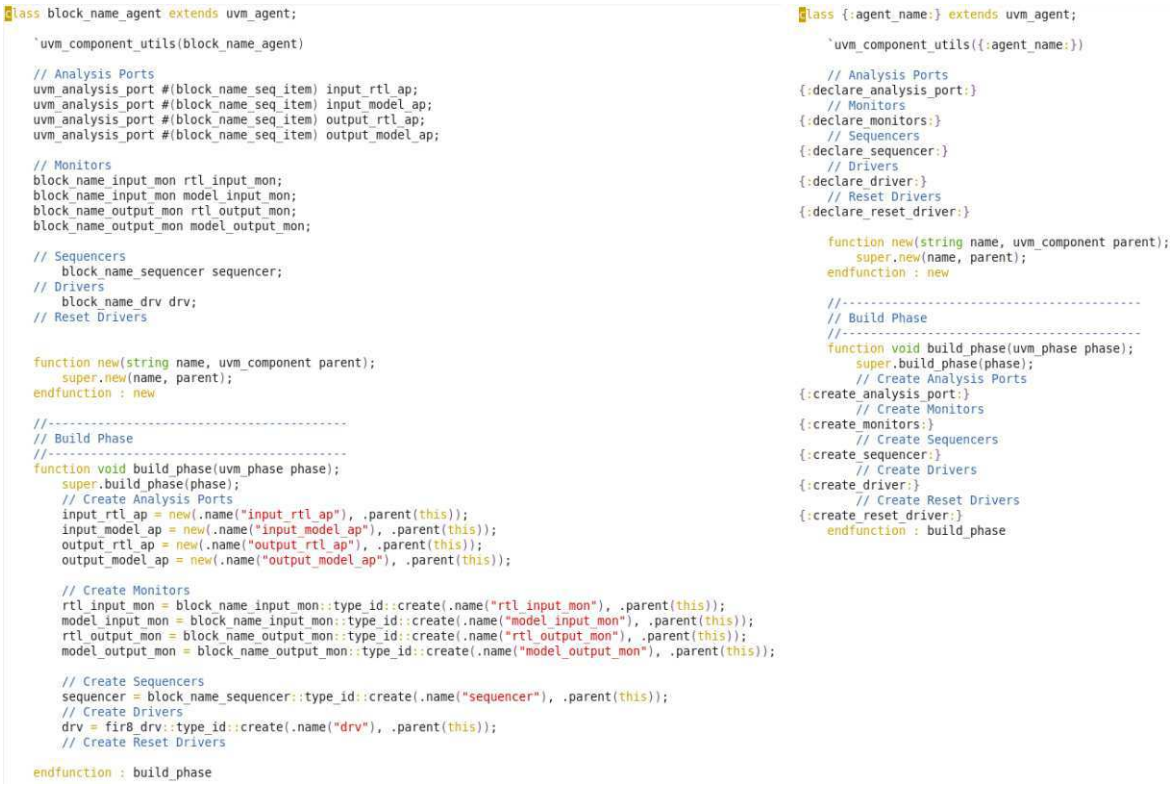
```
1 block-gen -n blk_name -source <spreadsheet>
```

É apresentado na Figura 10 uma lista dos arquivos gerados. Os código-fontes não serão disponibilizados por questões de segredo industrial, no entanto suas estruturas podem ser derivadas dos *templates* do Anexo A

É ilustrado na Figura 11 um exemplo do código gerado acompanhado pelo respectivo *template*. É possível observar a substituição das etiquetas por trechos compiláveis de código em SystemVerilog, nesse caso para geração do *agent*.

```
[lucas.arruda@artemis tb]$ ls -l
total 80
-rw-r--r--+ 1 lucas.arruda domain users 2895 Nov 17 13:07 blk_name_agent.svh
-rw-r--r--+ 1 lucas.arruda domain users  720 Nov 17 13:07 blk_name_base_sequence.svh
-rw-r--r--+ 1 lucas.arruda domain users 1632 Nov 17 13:07 blk_name_drv.svh
-rw-r--r--+ 1 lucas.arruda domain users 2473 Nov 17 13:07 blk_name_env.svh
-rw-r--r--+ 1 lucas.arruda domain users 1301 Nov 17 13:07 blk_name_if.sv
-rw-r--r--+ 1 lucas.arruda domain users 1810 Nov 17 13:07 blk_name_input_mon.svh
-rw-r--r--+ 1 lucas.arruda domain users 1575 Nov 17 13:07 blk_name_output_mon.svh
-rw-r--r--+ 1 lucas.arruda domain users 3092 Nov 17 13:07 blk_name_output_scb.svh
-rw-r--r--+ 1 lucas.arruda domain users  633 Nov 17 13:07 blk_name_package.sv
-rw-r--r--+ 1 lucas.arruda domain users  601 Nov 17 13:07 blk_name_params.svh
-rw-r--r--+ 1 lucas.arruda domain users  880 Nov 17 13:07 blk_name_seq_item.svh
-rw-r--r--+ 1 lucas.arruda domain users  259 Nov 17 13:07 blk_name_sequencer.svh
-rw-r--r--+ 1 lucas.arruda domain users 1052 Nov 17 13:07 blk_name_test.svh
-rw-r--r--+ 1 lucas.arruda domain users 2854 Nov 17 13:07 blk_name_top_tb.sv
```

Figura 10 – Lista de arquivos gerados pelo TbGen. Fonte: *print screen* de tela do computador



```
class block_name_agent extends uvm_agent;

    `uvm_component_utils(block_name_agent)

    // Analysis Ports
    uvm_analysis_port #(block_name_seq_item) input_rtl_ap;
    uvm_analysis_port #(block_name_seq_item) input_model_ap;
    uvm_analysis_port #(block_name_seq_item) output_rtl_ap;
    uvm_analysis_port #(block_name_seq_item) output_model_ap;

    // Monitors
    block_name_input_mon rtl_input_mon;
    block_name_input_mon model_input_mon;
    block_name_output_mon rtl_output_mon;
    block_name_output_mon model_output_mon;

    // Sequencers
        block_name_sequencer sequencer;
    // Drivers
        block_name_drv drv;
    // Reset Drivers


    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    //----------------------------------------
    // Build Phase
    //----------------------------------------
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        // Create Analysis Ports
        input_rtl_ap = new(.name("input_rtl_ap"), .parent(this));
        input_model_ap = new(.name("input_model_ap"), .parent(this));
        output_rtl_ap = new(.name("output_rtl_ap"), .parent(this));
        output_model_ap = new(.name("output_model_ap"), .parent(this));

        // Create Monitors
        rtl_input_mon = block_name_input_mon::type_id::create(.name("rtl_input_mon"), .parent(this));
        model_input_mon = block_name_input_mon::type_id::create(.name("model_input_mon"), .parent(this));
        rtl_output_mon = block_name_output_mon::type_id::create(.name("rtl_output_mon"), .parent(this));
        model_output_mon = block_name_output_mon::type_id::create(.name("model_output_mon"), .parent(this));

        // Create Sequencers
        sequencer = block_name_sequencer::type_id::create(.name("sequencer"), .parent(this));
        // Create Drivers
        drv = fir8_drv::type_id::create(.name("drv"), .parent(this));
        // Create Reset Drivers

    endfunction : build_phase
```

```
class {:agent_name:} extends uvm_agent;

    `uvm_component_utils({:agent_name:})

    // Analysis Ports
{:declare_analysis_port:}
    // Monitors
{:declare_monitors:}
    // Sequencers
{:declare_sequencer:}
    // Drivers
{:declare_driver:}
    // Reset Drivers
{:declare_reset_driver:}

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    //----------------------------------------
    // Build Phase
    //----------------------------------------
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        // Create Analysis Ports
{:create_analysis_port:}
        // Create Monitors
{:create_monitors:}
        // Create Sequencers
{:create_sequencer:}
        // Create Drivers
{:create_driver:}
        // Create Reset Drivers
{:create_reset_driver:}
    endfunction : build_phase
```

Figura 11 – Comparação entre código gerado (esquerda) e *template* (direita). Fonte: *print screen* de tela do computador

# 5 Considerações Finais

Desenvolveu-se um programa em Python capaz de gerar *testbenchs* em UVM, sendo uma ferramenta eficaz para aumento na produtividade de equipes de verificação de IPs digitais. O intuito deste trabalho não foi de desenvolver um gerador autossuficiente, isto é, algo que verificaria os IPs simplesmente ao apertar um botão, mas sim a geração de um arcabouço robusto o suficiente para que os verificadores realizassem apenas pequenas modificações.

Realizou-se um estudo extensivo sobre conceitos de microeletrônica, verificação e UVM, e foram adquiridos diversos aprendizados tanto nessas áreas quanto em programação em Python, que era desconhecida do autor no início das atividades.

Trabalhos futuros para esse projeto incluem o desenvolvimento de um ambiente de verificação para IPs que instanciem outros IPs e utilização da UVM *Register Layer* para validação de *designs* que apresentem utilização de bancos de registradores. Além disso, é desejável a implementação de uma lógica capaz de ler os códigos já alterados pelos verificadores e atualizar apenas os trechos que são automaticamente gerados.

# Referências

ACCELLERA, S. I. Universal verification methodology (uvm) user's guide. *Estados Unidos*, 2015. Citado 2 vezes nas páginas 22 e 23.

BERGERON, J. Writing testbenches - functional verification of hdl models. *Kluwer Academic Publishers, Nova Iorque, Estados Unidos*, 2002. Citado na página 18.

CARVALHO, H. d. L. Ambiente de verificação funcional de um ip-core em uvm. *TCC (Engenharia Elétrica), UFCG, Campina Grande Brasil*, 2018. Citado na página 28.

PIZIALI, A. Functional verification coverage measurement and analysis. *Kluwer Academic Publishers, Nova Iorque, Estados Unidos*, 2004. Citado na página 22.

RASHINKAR, P.; PATERSON, P.; SINGH, L. System-on-a-chip verification. *Kluwer Academic Publishers, Nova Iorque, Estados Unidos*, 2002. Citado na página 18.

# Anexos

# ANEXO A – *Templates*

```
class {:agent_name:} extends uvm_agent;

    `uvm_component_utils({:agent_name:})

    // Analysis Ports
{:declare_analysis_port:}
    // Monitors
{:declare_monitors:}
    // Sequencers
{:declare_sequencer:}
    // Drivers
{:declare_driver:}
    // Reset Drivers
{:declare_reset_driver:}

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    //---------------------------------------
    // Build Phase
    //---------------------------------------
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        // Create Analysis Ports
{:create_analysis_port:}
        // Create Monitors
{:create_monitors:}
        // Create Sequencers
{:create_sequencer:}
        // Create Drivers
{:create_driver:}
        // Create Reset Drivers
{:create_reset_driver:}
    endfunction : build_phase

    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
{:connect_analysis_port:}
{:connect_driver:}
    endfunction : connect_phase

```

```
endclass : {:agent_name:}
```

Listing A.1 – *Template* do *agent*

```
class {:driver_name:} extends uvm_driver#({:seq_item_name:});

    `uvm_component_utils({:driver_name:})

{:declare_virtual_if:}
    {:seq_item_name:} seq_item;
    //-----------------------------------------
    // Implementation
    //-----------------------------------------
    function new(string name = "{:driver_name:}", uvm_component parent);
        super.new(name, parent);
    endfunction : new

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
    endfunction : build_phase

    task run_phase(uvm_phase phase);
        //-----------------------------------------
        // Drive Data
        //-----------------------------------------

        fork
            run_process();
        join
    endtask : run_phase

    task run_process;
        forever begin
            seq_item_port.get_next_item(seq_item);
{:assignment:}
            @(posedge {:clock:});
            seq_item_port.item_done();
        end
    endtask : run_process

    task clean_up;
{:rst_assignment:}
    endtask : clean_up
endclass : {:driver_name:}
```

Listing A.2 – *Template* do *driver*

```
class {:environment_name:} extends uvm_env;
    `uvm_component_utils({:environment_name:})
```

```
3
4  {:declare_agents:}
5  {:declare_scoreboards:}
6  {:declare_vif:}
7  {:declare_reg_model:}
8  {:declare_reg_adapter:}
9  {:declare_reg_predictor:}
10     function new (string name, uvm_component parent);
11         super.new(name, parent);
12         {:create_reg_model:}
13     endfunction : new
14
15     function void build_phase(uvm_phase phase);
16         super.build_phase(phase);
17 {:create_agents:}
18 {:create_scoreboards:}
19 {:create_vif:}
20 {:build_reg_model:}
21 {:build_reg_adapter:}
22 {:build_reg_predictor:}
23     endfunction : build_phase
24
25     function void connect_phase(uvm_phase phase);
26         super.connect_phase(phase);
27 {:set_mapper_sequencer:}
28 {:set_predictor_map:}
29 {:connect_analysis_port:}
30 {:set_mon_vif:}
31 {:set_drv_vif:}
32     endfunction : connect_phase
33
34 endclass : {:environment_name:}
```

Listing A.3 – *Template* do *env*

```
1  class {:monitor_name:} extends uvm_monitor;
2      `uvm_component_utils({:monitor_name:})
3
4      //------------------------------------------
5      // Data Members
6      //------------------------------------------
7  {:analysis_port_declare:}
8  {:transaction_declare:}
9  {:declare_vif:}
10     //------------------------------------------
11     // Coverage
12     //------------------------------------------
13 {:declare_coverage:}
```

```
14      //-----------------------------------------
15      // Implementation
16      //-----------------------------------------
17      function new(string name="{:monitor_name:}", uvm_component parent);
18          super.new(name, parent);
19          // Transactions
20  {:create_transactions:}
21          // Covergroups
22          //-- create cover groups
23  {:create_cover_groups:}
24      endfunction: new
25
26      function void build_phase(uvm_phase phase);
27          super.build_phase(phase);
28  {:build_analysis_port:}
29      endfunction : build_phase
30
31      task run_phase(uvm_phase phase);
32          @(posedge {:reset:});
33          fork
34  {:collect_ports:}
35          join
36      endtask: run_phase
37
38  {:collector_task_begin:}
39      task collect_{:port_name:}();
40          forever begin
41              {:create_item:}
42  {:get_interface:}
43              {:port_name:}.write({:port_transaction:});
44              @(posedge {:port_clock:});
45          end
46          // Coverage : Collect Function
47  {:collect_functions:}
48      endtask: collect_{:port_name:}
49
50  {:collector_task_end:}
51      extern task clean_up();
52
53  endclass : {:monitor_name:}
54
55  task {:monitor_name:}::clean_up;
56  {:reset_interfaces:}
57  endtask
```

Listing A.4 – *Template* do *monitor*

```
1  `timescale 1ns/1ps
```

```
2
3 import uvm_pkg::*;
4
5 package {:package_name:};
6
7 {:parameters:}
8     `include "uvm_macros.svh"
9
10 {:includes:}
11
12 endpackage : {:package_name:}
```

Listing A.5 – *Template* do *package*

```
1 class {:scoreboard_name:} extends uvm_scoreboard;
2     `uvm_component_utils({:scoreboard_name:})
3
4 {:declare_analysis_port:}
5 {:declare_tlm_analysis_fifo:}
6 {:declare_vif:}
7
8     //------------------------------------------
9     // Scoreboard Elements
10    //------------------------------------------
11    // Transactions
12 {:declare_transaction:}
13    // Local Variables
14    //--
15
16    function new (string name, uvm_component parent);
17        super.new(name, parent);
18    endfunction : new
19
20    function void build_phase(uvm_phase phase);
21        super.build_phase(phase);
22 {:create_analysis_port:}
23 {:create_tlm_analysis_fifo:}
24 {:create_vif:}
25    endfunction : build_phase
26
27    function void connect_phase(uvm_phase phase);
28 {:connect_analysis_ports:}
29    endfunction : connect_phase
30
31    virtual task pre_main_phase(uvm_phase phase);
32        super.pre_main_phase(phase);
33  // This Phase is executed before the first stimulus is applied. Is
    used to ensure all the required components are ready
```

```
34   // to start generating stimulus. Export file variables can be defined
      and initialized here.
35   // $display("================== Pre Main Phase ==================");
36
37     endtask : pre_main_phase
38
39     task run_phase(uvm_phase phase);
40         forever begin
41       @(posedge {:reset:});
42       fork
43           clean_up();
44     run_process();
45       join
46       @(negedge {:reset:});
47       disable fork;
48       clean_up();
49   end
50     endtask : run_phase
51
52     protected task run_process();
53         forever begin
54 {:data_get:}
55   end
56     endtask
57
58     function void check_data();
59         //-- Check Data Logic
60     endfunction : check_data
61
62     function void clean_up();
63         //-- Reset Local Variables
64     endfunction : clean_up
65
66     function void report_phase (uvm_phase phase);
67         super.report_phase(phase);
68   // The report phase is used to display the results of the simulation
      to the standard output
69   // or to write the results to file. This phase is usually used by
      Analysis Components.
70   // Executed after Run Phase is finished, export variables can be
      closed here.
71   // $display("================== Report Phase ==================");
72     endfunction : report_phase
73
74 endclass : {:scoreboard_name:}
```

Listing A.6 – *Template* do *scoreboard*

```
class {:sequence_name:} extends {:sequence_parent:}
    `uvm_object_utils({:sequence_name:})

    function new(string name = "{:sequence_name:}");
        super.new(name);
    endfunction: new

    virtual task body();
       `uvm_do_with(
        //-- Replace req.input constraints with Pop variables when using DataGen
            req,{
                {:req_instance:}
            }
        }
    endtask: body
endclass: {:sequence_name:}
```

Listing A.7 – *Template* da *sequence*

```
class {:test_name:} extends uvm_test;
    `uvm_component_utils({:test_name:})
    int ITERATIONS = 1000; // Change the test duration here
{:declare_envs:}

    function new (string name="{:test_name:}", uvm_component parent=null);
        super.new(name, parent);
    endfunction : new

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
{:create_envs:}
    endfunction : build_phase

    task run_phase(uvm_phase phase);
{:sequence_declare:}

        for(int i=0; i<ITERATIONS; i++) begin
            `uvm_info(get_type_name(), $sformatf("\n=== TEST IMPLEMENTED %d ITERATION ===", i), UVM_LOW)
            phase.raise_objection(.obj(this));

{:sequence_run:}
            phase.drop_objection(.obj(this));
        end
    endtask

```

```
endclass : {:test_name:}
```

Listing A.8 – *Template* do *test*

```
class {:trans_name:} extends {:sequence_type:};

    //-----------------------------------------
    // Data Members
    //-----------------------------------------
    // Input / Outputs
{:inouts_def:}
    // Auxiliar Vars
{:aux_def:}

    //-----------------------------------------
    // Field automation
    //-----------------------------------------
    `uvm_object_utils_begin({:trans_name:})
{:obj_utils:}
    `uvm_object_utils_end

    function new(string name = "{:trans_name:}");
        super.new(name);
    endfunction

endclass: {:trans_name:}
```

Listing A.9 – *Template* da *transaction*