UNIVERSIDADE FEDERAL DO PAMPA

Vinícius Bittencourt da Silva

FERRAMENTA WEB PARA GERAÇÃO SEMIAUTOMÁTICA DE AMBIENTES DE VERIFICAÇÃO UVM COM SYSTEMVERILOG

Alegrete
2018

Vinícius Bittencourt da Silva

FERRAMENTA WEB PARA GERAÇÃO SEMIAUTOMÁTICA DE AMBIENTES DE VERIFICAÇÃO UVM COM SYSTEMVERILOG

Dissertação apresentada ao Programa de Pós-graduação Stricto Sensu em Engenharia Elétrica da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica.

Orientador: Dr. Alessandro Gonçalves Girardi

Alegrete
2018

Ficha catalográfica elaborada automaticamente com os dados fornecidos
pelo(a) autor(a) através do Módulo de Biblioteca do
Sistema GURI (Gestão Unificada de Recursos Institucionais) .

S586f Silva, Vinícius Bittencourt da
Ferramenta Web Semiautomática para Geração de Ambientes de Verificação UVM com SystemVerilog / Vinícius Bittencourt da Silva.
82 p.

Dissertação(Mestrado)-- Universidade Federal do Pampa, MESTRADO EM ENGENHARIA ELÉTRICA, 2018.
"Orientação: Alessandro Gonçalves Girardi".

1. UVM. 2. Ambientes de Verificação. 3. Ferramenta Web. I. Título.

Vinícius Bittencourt da Silva

FERRAMENTA WEB SEMIAUTOMÁTICA PARA GERAÇÃO DE AMBIENTES DE VERIFICAÇÃO UVM COM SYSTEMVERILOG

Dissertação apresentada ao Programa de Pós Graduação em Engenharia Elétrica da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica.

Dissertação defendida e aprovado em 07 de Fevereiro de 2018

Banca examinadora:

Alessandro Gonçalves Girardi
Orientador

Sidinei Ghissoni
UNIPAMPA

Fábio Diniz Rossi
IFFar

Em memória à minha querida vó Eva.

AGRADECIMENTOS

Agradeço a todos que contribuíram direta e indiretamente para a realização desse trabalho. Aos meus familiares e amigos pelo apoio de cada dia. Aos colegas de laboratório e pesquisa, pelas horas de trabalho e esforço. E ao meu orientador pela paciência e ensinamentos.

"Aonde fica a saída?", Perguntou Alice ao gato que ria.
"Depende", respondeu o gato.
"De quê?", replicou Alice;
"Depende de para onde você quer ir..."
(Alice no País das Maravilhas, Lewis Carroll)

RESUMO

Atualmente, o tempo de inserção de um produto de hardware no mercado é cada vez menor apesar do crescimento de sua complexidade. Portanto, é importante que o processo de construção seja cada vez mais rápido. Entre as medidas para ganhar desempenho a otimização do tempo despendido em verificação é fundamental, pois cerca de 70% do tempo de projeto é aplicado nessa atividade. Esse processo inicia-se juntamente com o desenvolvimento, pois, caso seja detectado um erro somente no estágio final de desenvolvimento é possível que haja atrasos para cumprir os prazos de entrega. Nesse sentido, este trabalho apresenta a USAG, uma ferramenta semi-automática desenvolvida para construir ambientes de verificação usando a metodologia UVM (a qual é a metodologia padrão atualmente) aplicada ao projeto de circuitos integrados escritos em *SystemVerilog*. Esta ferramenta vem para ajudar no processo de verificação de *hardware* acelerando a criação do ambiente de verificação, uma vez que ele gera as estruturas e interconexões da metodologia e produz os arquivos para simulação. Qualquer ferramenta que suporte *SystemVerilog* juntamente com a Metodologia UVM pode executar o ambiente de verificação gerado pela USAG. Além disso, a ferramenta é baseada na *Web* para ser acessível a partir de qualquer local sem a necessidade de um sistema operacional específico ou configuração para usá-la. Finalmente, são apresentados os resultados de ambientes de verificação UVM obtidos a partir da entrada de códigos fonte em *SystemVerilog* na USAG. A partir dos resultados obtidos e da análise da utilização por parte de testadores conclui-se que a USAG é eficaz no que tange os objetivos propostos.

Palavras-chave: UVM. Ambientes de Verificação. Ferramenta Web.

ABSTRACT

Currently, the insertion time of a hardware products in the market is decreasing despite the growth of its complexity. Therefore, it is important that the construction process is getting faster and faster. Among the ways to gain performance, the optimization of the time spent in verification is fundamental, because about 70% of the project time is applied in this activity. This process starts with the development, because if an error is detected only in the final stage of development, there may be delays to comply with delivery times. In this way, this work presents USAG, a semi-automatic tool developed to construct verification environments using the UVM methodology (which is the current standard methodology) applied to the design of integrated circuits written in SystemVerilog. This tool comes to assist in the hardware verification process by accelerating the creation of the verification environment as it generates the structures and interconnections of the methodology and produces the files for simulation. Any tool that supports SystemVerilog together with the UVM Methodology can execute the verification environment generated by the USAG. In addition, the tool is web-based to be accessible from any location without the need for a specific operating system or configuration to use it. Finally, the results of UVM verification environments obtained from the entry of source codes in SystemVerilog in USAG are presented.

Key-words: UVM. Verification Environments. Web Tool.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O início da era da tecnologia da informação foi marcado pelo advento da microeletrônica. Essa por sua vez, possibilitou a criação de circuitos eletrônicos com maior integração gerando um processamento cada vez maior da informação (SEVERO, 2012). Um desses tipos de circuitos eletrônicos são os Circuitos Integrados (CI), nos quais a complexidade de desenvolvimento tem crescido ao passar dos anos (PESSOA, 2007). O principal fator que corrobora para o aumento da complexidade vem do fato que os circuitos integrados são construídos sobre elementos semicondutores, usualmente silício, em escalas micrométricas ou nanométricas (SEVERO, 2012).

Atualmente, o tempo de inserção de um produto de *hardware* no mercado é cada vez menor (FARIA, 2017), apesar do crescimento de sua complexidade. Portanto, é importante que o processo de construção seja cada vez mais rápido. Entre as medidas para aumentar o desempenho cita-se a otimização do tempo despendido em verificação, sendo que a busca por defeitos em um circuito integrado é dada no início do processo de desenvolvimento e deve se manter durante todo o processo de elaboração. Isso é fundamental, pois, uma vez que seja detectado um erro somente no estágio final de desenvolvimento é possível que haja atrasos para cumprir os prazos de entrega (CAMARA, 2011)(WEISSNEGGER et al., 2016).

Pode-se dizer que a verificação é uma tarefa ainda maior que o próprio projeto quando se trata do desenvolvimento de um circuito integrado. Diversos trabalhos apontam que mais de 70% do tempo de projeto é gasto em verificação (FOSTER, 2015), e assim, mais esforços devem ser aplicados a fim de especificar uma forma de garantir que o circuito realmente está correto (BERGERON et al., 2006). O ato de realizar a verificação de um objeto de *hardware* é geralmente chamado de verificação funcional. A verificação funcional pode ser caracterizada como a execução de um método cujo objetivo é garantir que o *Device Under Test* (DUT), o qual é o projeto de *hardware* sob verificação, esteja funcionando de forma adequada (MINTZ; EKENDAHL, 2007). Isso implica que o maior impedimento para introduzir um novo *hardware* no mercado não é o fato de desenvolver o circuito em si, mas sim o fato de realizar a verificação funcional do mesmo, sendo esse o maior gargalo num ciclo tradicional de desenvolvimento de circuitos (MINTZ; EKENDAHL, 2007) (COHEN; VENKATARAMANAN; KUMARI, 2005).

A fim de realizar a verificação funcional é necessário a utilização de alguma linguagem de verificação de *hardware* (do inglês *Hardware Verification Languages* – HVL). Nesse sentido, as linguagens VHDL e *Verilog* apresentam-se como inadequadas, uma vez que não possuem um bom suporte para tipos de dados de alto nível, cobertura funcional ou programação orientada a objetos (POO). Entre as soluções comerciais pode-se encontrar a HVL *"OpenVera"* da empresa *Synopsys*, e a HVL *"e"* da *Cadence*. Além dessas linguagens, há soluções de código aberto (*Open Source*), como *SystemC Verification Library* (SCV), *Jeda*, entre outras (BERGERON et al., 2006). Para a realização de verificações em *hardware*,

destaca-se a HVL *SystemVerilog*, a qual inclui todas as características necessárias para executar a verificação de forma adequada. Além disso, é a primeira linguagem padrão da indústria nesse âmbito e a que possui maior taxa de adoção (FOSTER, 2015).

Uma vez definida a HVL, uma das formas de realizar a verificação de um circuito integrado é por meio da utilização de *testbenchs*. Esse termo geralmente se refere à simulação de código para criar uma sequência de entradas pré-determinadas para um circuito e observar a resposta obtida como retorno (SILVA, 2007).

Apesar da completude da HVL *SystemVerilog* houve a criação de bibliotecas, *toolkits* e metodologias adicionais a ela. Dentre essas metodologias de verificação destaca-se a *Universal Verification Metodology* (UVM). A UVM não é um complemento às necessidades do *SystemVerilog*, e sim uma biblioteca. Portanto, utilizando-se apenas *SystemVerilog* é possível construir toda a estrutura UVM (BROMLEY, 2013).

Por meio da UVM é possível obter diferentes tipos de componentes de verificação através da herança de elementos da biblioteca. Esses componentes são utilizados para verificar o DUT. O *testbench* é divididos em camadas de forma a facilitar o controle da complexabilidade (SALAH, 2014).

No âmbito apresentado, o presente trabalho tem como foco explorar as características da HVL *SystemVerilog* quando utilizada juntamente com a metodologia UVM. Para tal, construiu-se a ferramenta USAG, (um acrônimo para *UVM Semi-Automatic Generator*), a qual busca auxiliar a construção de ambientes de verificação da metodologia UVM a fim de alcançar uma maior facilidade e, principalmente, agilidade para o usuário responsável por verificar objetos de *hardware*. Além das características mencionadas, a USAG explora o conceito de orientação a serviços, tornando assim o seu acesso simples, uma vez que não há necessidade de instalação de um sistema operacional específico para sua execução, somente o acesso à internet.

1.1 MOTIVAÇÃO

A complexidade de concepção de circuitos integrados ao longo dos anos tem crescido porém o tempo para alcançar o mercado é cada vez menor. Portanto, é necessário obter agilidade nesse processo. A Figura 1, ilustra esse cenário relacionando os projetos pelo número de portas lógicas (incluindo caminho de dados, excluindo memórias) em cada um.

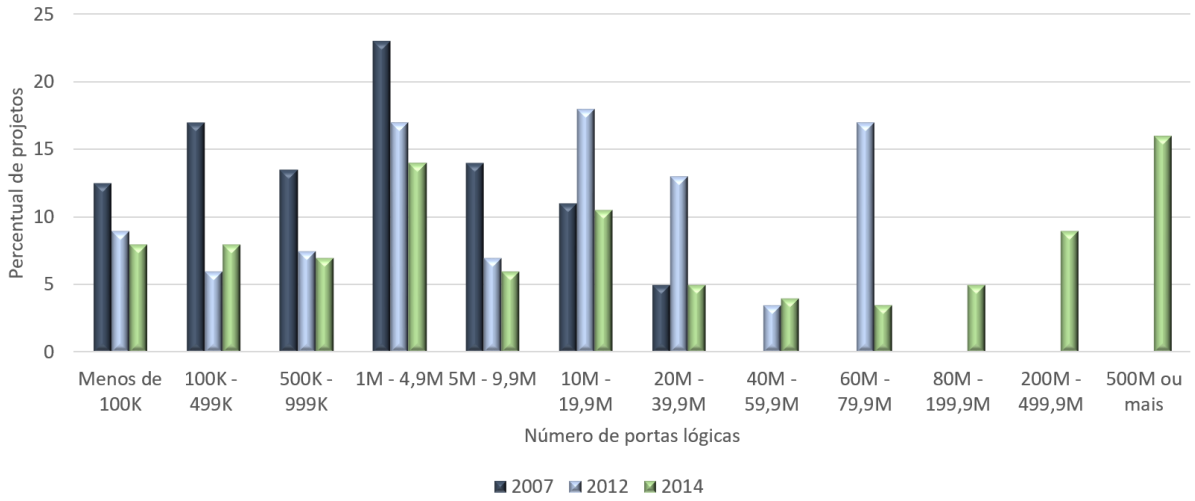

Figura 1 – Tamanho de projetos. Fonte: (FOSTER, 2015).

Durante o desenvolvimento de um circuito integrado, a maior parte do tempo é despendida na geração e execução de verificações nos objetos de *hardware* a fim de garantir que estejam corretos e atendam às necessidades pelas quais foram criados. Como pode-se observar na Figura 2, adaptada de (FOSTER, 2015), em 2014 o tempo médio utilizado para verificação era de aproximadamente 57%. Entretanto, nota-se o crescimento de projetos que empregam mais de 80% do tempo para a verificação em relação a anos anteriores. É possível observar na Figura 1 que a tendência tem sido o aumento no número de no tamanho dos projetos. Verifica-se também que somente a partir de 2014 passou-se a possuir projetos com mais de 80 milhões de portas lógicas. No mesmo ano, 31% dos projetos tinham esse valor, sendo que 17% destes possuíam mais de 500 milhões de portas.

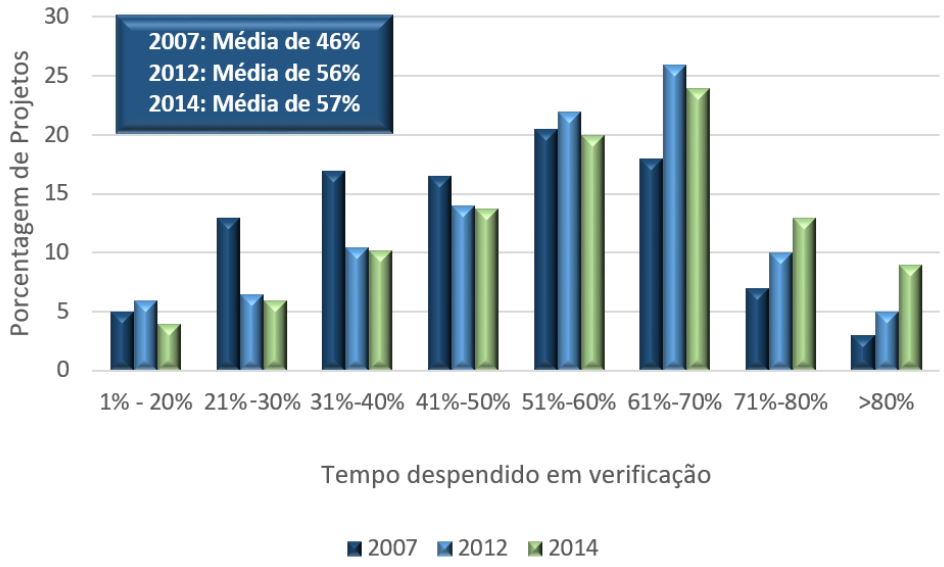

Figura 2 – Porcentagem do tempo despendido em verificação. Fonte: (FOSTER, 2015).

O aumento do tamanho dos projetos, juntamente com o aumento no tempo des-

pendido em verificação acaba também por acrescer o número de engenheiros de verificação por projeto, como é apresentado na Figura 2. É possível observar que o cenário em relação que número de engenheiros mudou drasticamente desde 2007, sendo que, em média, já há mais engenheiros de verificação que engenheiros de projeto (FOSTER, 2015).

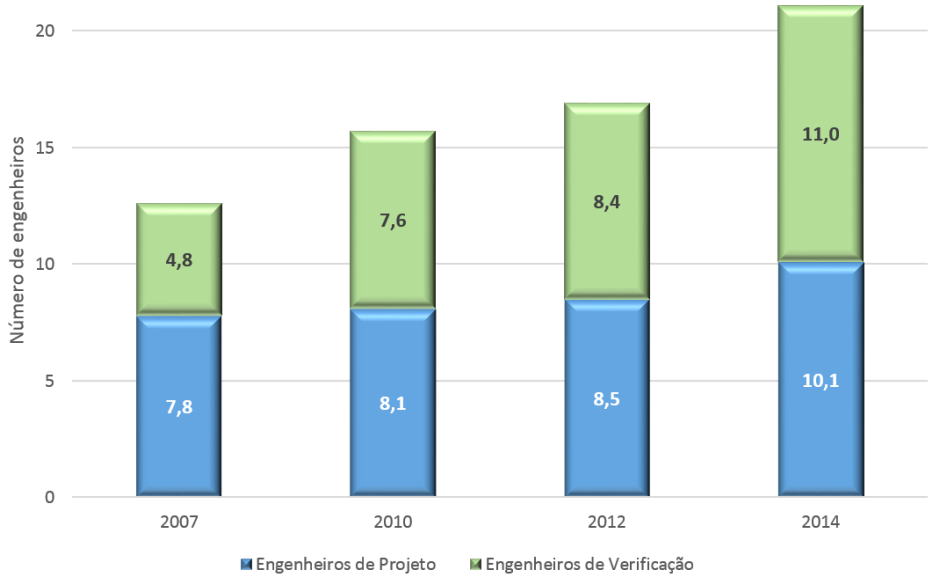

Figura 3 – Número de engenheiros por projeto. Fonte: (FOSTER, 2015).

Já entre as linguagens de verificação (HVLs) utilizadas para a construção dos ambientes a serem aferidos destaca-se o *SystemVerilog*, pois, como pode-se observar na Figura 4, é a linguagem com maior adoção para a geração de ambientes de verificação. Além disso, é possível notar que a soma da relação de porcentagem de HVLs por ano supera 100%. Isso se deve ao fato que os projetos podem utilizar mais de uma HVL para realizar a verificação.

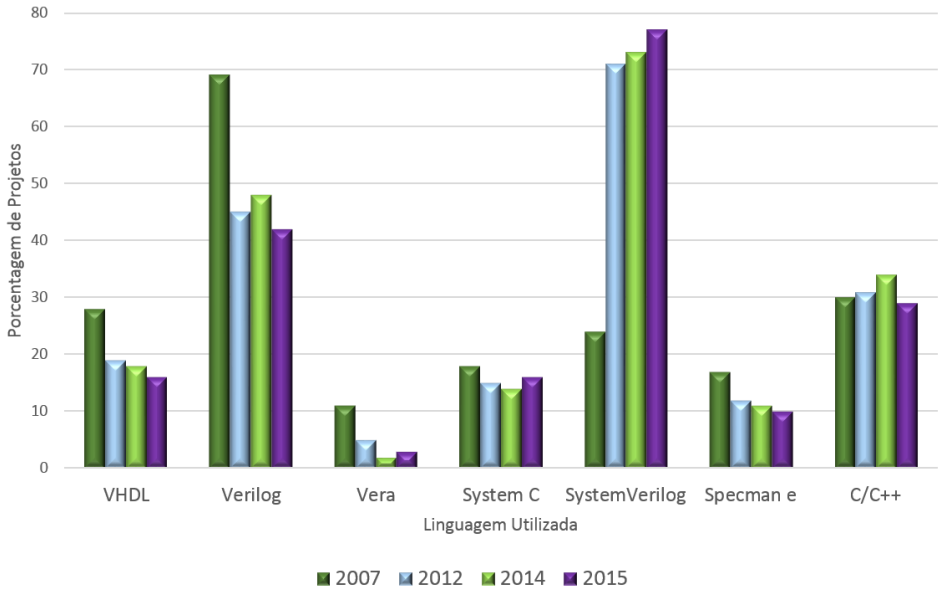

Figura 4 – Linguagens utilizadas para ambientes de verificação *(Testbenchs)*. Fonte: (FOSTER, 2015).

Apesar de ser a linguagem mais utilizada, segundo (BROMLEY, 2013), o *SystemVerilog* por si só não é capaz de impulsionar o uso de boas práticas em técnicas de verificação. Assim, pode-se perceber o surgimento de diversas ferramentas, bibliotecas e metodologias para auxiliar tais práticas. Destaca-se nesse sentido a UVM, a qual é uma metodologia focada em aplicar práticas corretas e também ao reuso de elementos de verificação. A UVM possui a maior adesão na indústria, como pode ser evidenciado na Figura 5.

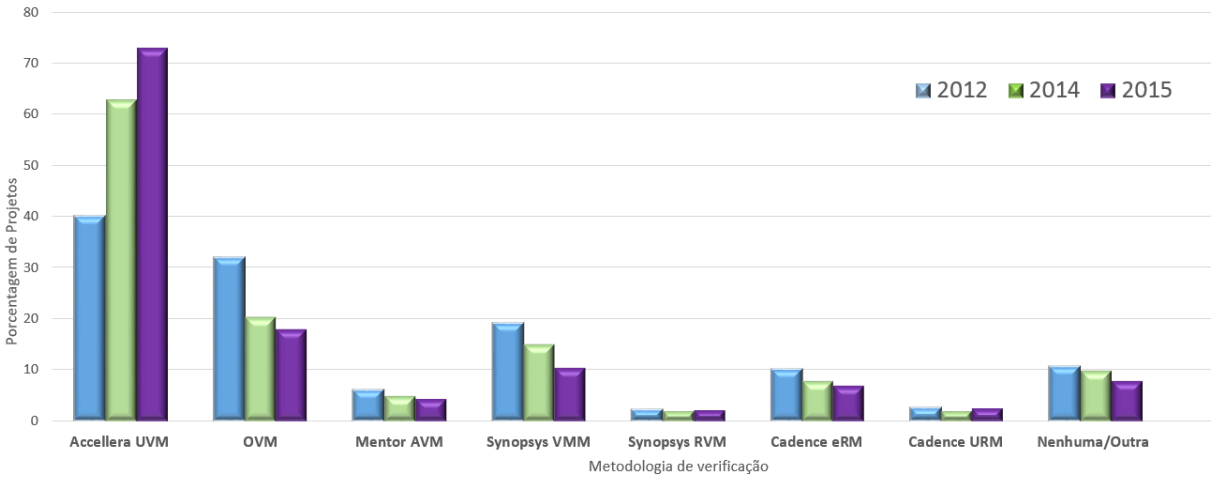

Figura 5 – Adoção de metodologias de verificação. Fonte: (FOSTER, 2015).

Dessa maneira com o aumento no tamanho dos projetos, e a não diminuição no tempo tempo despendido em verificação acarreta na necessidade do aumento do número de engenheiros por projeto. Tendo isso em vista, a motivação do presente trabalho é contribuir

para a obtenção de agilidade na construção de ambientes de verificação que foquem boas práticas e que utilizem os padrões de linguagem e metodologias amplamente aplicados pela indústria atualmente.

1.2 OBJETIVOS

Para a realização deste trabalho, partiu-se do cenário de que pessoas que realizam a verificação de circuitos integrados em desenvolvimento precisam obter uma maior agilidade na criação dos ambientes de verificação para atender às necessidades de tempo para inserção de novos objetos de *hardware* no mercado.

Além disso, a premissa de que o uso de sistemas orientados a serviço são cada vez mais frequente em todos os segmentos de tecnologia é tida como verdadeira. Adotando a presente hipótese, o principal objetivo desse trabalho é a construção de uma ferramenta chamada USAG, para a geração semiautomática de ambientes de verificação de circuitos integrados descritos em linguagem *SystemVerilog*, utilizando para tal a metodologia de verificação UVM. Por meio da USAG pretende-se tornar mais rápido a geração dos ambientes e por consequência a verificação dos mesmos.

Outro requisito do trabalho é que o sistema de geração de ambientes de verificação seja de fácil utilização e independente de sistemas operacionais. Também é desejado que não haja a necessidade de instalação/configuração do mesmo por parte do usuário. Essa característica visa possibilitar um acesso rápido às informações e dados gerados a partir da ferramenta. Assim sendo, espera-se obter uma ferramenta *online*, de fácil utilização, cujo o foco é a geração de ambientes de verificação de forma ágil a partir da linguagem *SystemVerilog* e com o uso da metodologia UVM.

No cenário exposto, define-se como objetivo geral, a obtenção de uma ferramenta capaz de gerar ambientes de verificação que explorem a metodologia UVM, de forma semiautomática por meio da interação com o usuário, utilizando como base códigos fontes descritos pelo usuário na HVL *SystemVerilog*. E, como objetivos específicos, explorar e apresentar a metodologia UVM e sua estrutura lógica de funcionamento, demonstrar ferramentas semelhantes no contexto do problema, expor a arquitetura da ferramenta, correlacionar a USAG com outras ferramentas, e, finalmente, exibir os funcionamento da USAG e resultados obtidos.

2 FUNDAMENTAÇÃO TEÓRICA

O presente capítulo tem como objetivo apresentar os conceitos teóricos correlacionados com a ferramenta USAG. Para tal, são apresentadas brevemente as definições sobre teste e verificação, as metodologias de verificação e seu histórico, assim como um comparativo entre suas características.

2.1 TESTE E VERIFICAÇÃO

No âmbito de *hardware*, quando há a utilização da terminologia de teste e verificação, muitas vezes esses dois termos se confundem. Porém, há uma sucinta diferença entre os dois. O termo teste geralmente refere-se ao ato de analisar o *hardware* já em fase de produção. Por outro lado, a verificação trata do *hardware* em sua fase de descrição. A Figura 6, extraída e adaptada de Silva (2007), elucida esse ciclo.

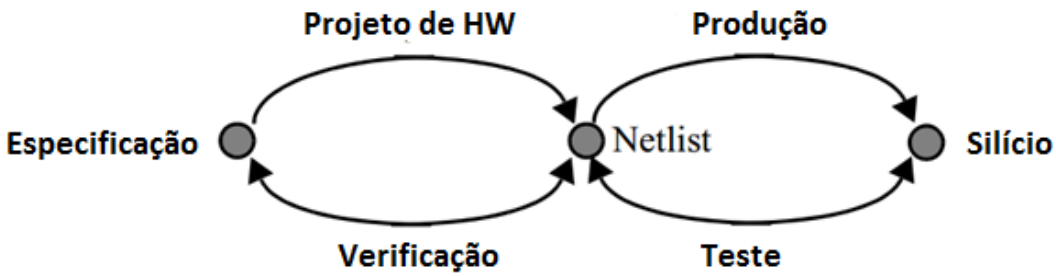

Figura 6 – Ciclos de teste e verificação. Adaptado de: Silva (2007).

Após a especificação do projeto, inicia-se a fase de "Projeto de HW", na qual é realizada a descrição do *hardware* em alguma HDL, como por exemplo, *SystemVerilog*. O código descrito é então verificado até obter-se o *Netlist* (o que poderia ser definido como o código já sintetizado e pronto para a obtenção de protótipo). Dessa forma, a verificação objetiva executar um trecho do projeto que conta-se como verdadeiro e assim confirmar a suposição a partir do resultado obtido. A cada alteração no projeto realiza-se uma nova verificação para se garantir que o que foi especificado inicialmente ainda mantém-se inalterado. A USAG correlaciona-se diretamente com esse conceito, pois objetiva auxiliar na criação dos ambientes de verificação.

Após realizada a síntese de um projeto de *hardware*, assim como geradas todas as suas interconexões, o mesmo passa por testes até se obter o objeto final em silício. O teste por si próprio objetiva encontrar novas informações que eventualmente tenham surgido após a fase de verificação. Ou seja, quando busca-se localizar um problema no objeto em silício, que não foi antecipado, então, está realizando-se um teste.

2.2 TESTES DE SOFTWARE

Os testes, quando discorre-se sobre *software*, têm como objetivo demonstrar que um sistema executa adequadamente o que ele é proposto a realizar. Esse comportamento dos testes de *software* é semelhante à verificação de *hardware*, dessa forma, muitos aspectos estão relacionados diretamente com a USAG. Como Sommerville (2007) define, o processo de testar é composto por dois objetivos distintos:

- Demonstrar ao desenvolvedor e ao cliente que o sistema executa de forma correta;
- Descobrir situações em que o sistema se comporta de forma inadequada, indesejável ou de maneira diferente de suas especificações.

Assim temos, que:

- Teste: é o ato de exercitar o sistema com casos de testes (*Test Case*);
- *Test Case*: também conhecido como caso de teste, é um grupo de condições de execução que estão atreladas ao comportamento esperado do sistema. Também pode possuir um conjunto de entradas e saídas.

É importante ressaltar a terminologia padrão adotada para testes de *software*. Segundo Jorgensen (2016), os padrões definidos pelo *Institute of Electronics and Electrical Engineers* (IEEE) podem ser descritos como segue:

- Erro: Trata-se de uma ação humana que produz um resultado incorreto. Tende a se propagar durante o desenvolvimento do projeto como, por exemplo, a interpretação errônea de uma necessidade de um cliente;
- Defeito: É um estado inconsistente ou inesperado durante a execução de um sistema. É algo que faz parte do produto e está implementado no código fonte equivocadamente. É o resultado de um erro.
- Falha: é a execução de um defeito no código.

Um teste não pode garantir que um sistema não possua defeitos ou que irá sempre se comportar da maneira correta. É possível determinar a partir de testes apenas a presença de defeitos e não a ausência dos mesmos (SOMMERVILLE, 2007). A partir desses conceitos é possível determinar que toda falha depende de um defeito, entretanto nem todo defeito gera uma falha. Além disso todo defeito é criado a partir de um erro, porém nem todos os erros acabam por refletir em defeitos. Conclui-se, portanto, que toda falha apresentada é oriunda de um erro durante a criação do código.

A Figura 7, adaptada de Sommerville (2007), apresenta o fluxo de execução de um teste onde há uma entrada de dados no sistema com valores dentre os quais alguns

acarretam em comportamento anômalo. Uma vez processado pelo sistema, o mesmo apresenta como retorno os defeitos gerados por erros ao trabalhar com aqueles valores.

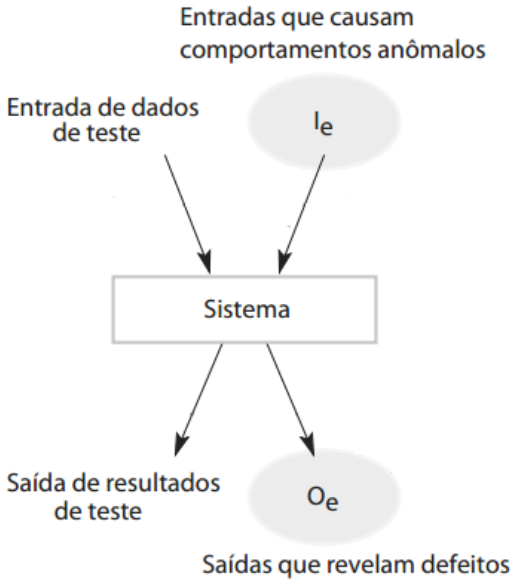

Figura 7 – Entrada de dados que revelam defeitos. Fonte: (SOMMERVILLE, 2007)

.

Já na Figura 8, adaptada de Jorgensen (2016), é apresentado um modelo de ciclo de vida para um teste. É visto que durante as fases de desenvolvimento é possível que surjam defeitos oriundos de erros humanos cometidos nesse processo. Também se observa que após o teste surge um elemento chamado de incidente. O incidente é um sintoma que está associado a uma falha no qual o usuário é alertado que a falha ocorreu efetivamente (JORGENSEN, 2016). Uma vez detectado o defeito, o mesmo é isolado e o defeito então é corrigido.

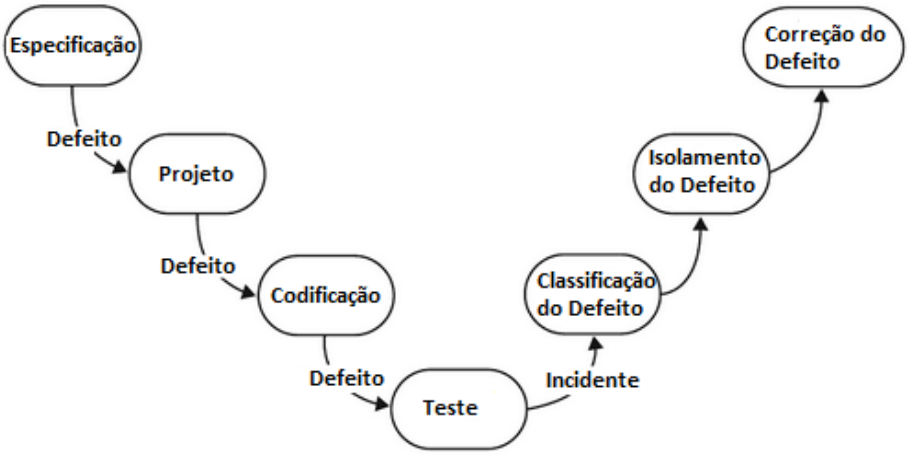

Figura 8 – Ciclo de vida de um teste. Adaptado de Jorgensen (2016).

A maneira como o fluxo e a execução do teste ocorrem implica na forma que o mesmo será classificado. Essa classificação se dá de acordo com o conhecimento na estrutura do código em que o teste está sendo aplicado. Segundo Pressman e Maxim (2016) essas técnicas são conhecidas como técnica estrutural e técnica funcional.

- Técnica estrutural – Essa técnica é conhecida como teste de caixa branca. Ela trabalha diretamente sobre o código fonte do componente elaborando-se casos de teste que cubram todas as possibilidades do mesmo;
- Técnica funcional - É conhecida como teste de caixa preta. Trata-se de uma abordagem de testes onde os testadores não possuem acesso ao código-fonte. Os testes são derivados da especificação do sistema.

A conceituação de testes de *software* apresentada nessa Seção objetiva introduzir o conceito de teste de caixa preta utilizado pela USAG.

2.3 TESTES DE CAIXA PRETA

Os testes de caixa preta se caracterizam pelo desconhecimento da implementação real do objeto no qual está sendo realizado o teste permitindo que a verificação funcional ocorra mesmo assim (SILVA, 2007).

Nos testes funcionais (ou de caixa preta) não se tem acesso ao código fonte, apenas aos seus sinais de entrada e saída. A Figura 9, apresenta graficamente esse cenário.

Figura 9 – Representação de um teste de caixa preta. Fonte: Autor.

Essa abordagem possui a vantagem de não depender de qualquer detalhe de implementação. Ou seja, mesmo que sejam realizadas modificações no código que está sob teste durante a fase de verificação, os casos de teste não precisam ser alterados contanto que não sejam alteradas as entradas e saídas do código. Por outro lado, o ponto negativo dessa abordagem é que se perde o controle da parte interna da implementação, uma vez que não se tem acesso ao conteúdo do código fonte (SILVA, 2007).

A USAG relaciona-se diretamente com essa técnica pois define os sinais que serão utilizados na construção de ambientes de verificação utilizando as entradas e saídas do código fonte em *SystemVerilog* inserido pelo usuário independentemente do restante da implementação.

2.4 FORMAS DE VERIFICAÇÃO

A verificação é o processo que demonstra se determinada descrição de *hardware* trabalha de forma adequada (PESSOA, 2007). Segundo Bergeron et al. (2006), pode-se dividir a verificação nas seguintes categorias: estática ou formal, dinâmica ou funcional, e híbrida.

A verificação formal, quando no âmbito de *hardware*, relaciona-se com a prova ou refutação da corretude de determinado sistema, sendo essa realizada a partir de determinada especificação formal ou propriedades as quais podem ser modeladas de uma forma matemática e abstrata ou por meio de métodos formais (PESSOA, 2007)(SANTOS, 2004).

Outro tipo de verificação é a dita dinâmica ou funcional. Essa verificação é semelhante à verificação formal no passo que se trata dos objetivos. Porém, para a verificação funcional não é necessária a criação de um método formal e a mesma utiliza-se de simulações para demonstrar se o produto de teste está correto. Para ambos os casos pode-se verificar a existência de erros, porém o contrário não é verdadeiro. Já verificação híbrida trata da junção da verificação formal com a verificação funcional. Dessa forma, a verificação híbrida por vezes utiliza-se uma forma de verificação, por vezes outra (PESSOA, 2007).

Assim, para auxiliar nos processos descritos, foram criadas metodologias de verificação. A USAG, utiliza uma dessas metodologias para gerar seus ambientes de verificação.

2.5 METODOLOGIAS DE VERIFICAÇÃO

Em 2000, a *Verisity Design* (posteriormente adquirida pela *Cadence*), apresentou um conjunto de boas práticas para verificação, a qual foi chamada de *Verification Advisor* (*vAdvisor*). Esse conjunto de boas práticas tinha como principal foco atender à comunidade de usuários da linguagem *e*, obtendo uma boa aceitação. Também apresentava alguns

conceitos básicos de verificação, incluindo criação de estímulos e modelos de cobertura. A *Verisity*, após 2 anos, expôs a primeira biblioteca de verificação, a qual foi chamada de *e Reuse Methodology* (eRM). Essa biblioteca trazia um grande número de características cuja estrutura futuramente seria herdada por metodologias como a OVM e a UVM. Além disso, a eRM também possuiu uma grande aceitação dos usuários (HEIGHT, 2010).

Já em 2003, a empresa *Synopsys* apresentou uma biblioteca para a linguagem de verificação *Vera*, chamada de *Reuse Verification Methodology* (RVM). Diferentemente da eRM, a RVM não apresentava diversos elementos estruturais (como capacidade de envio de mensagens), sendo considerada muitas vezes para os usuários como um subconjunto da eRM. Entretanto, a metodologia RVM trouxe como contribuição as chamadas de função "*callback*" (HEIGHT, 2010).

Em 2006 a empresa *Mentor* foi responsável por inserir no mercado a *Advanced Verification Methodology* (AVM), com capacidades de verificação de alto nível estabelecidas no núcleo da eRM na forma de classes de testes. Esta foi uma importante metodologia principalmente por ter sido a primeira solução de verificação de código aberto que utilizava *Transaction-Level Modeling* (TLM).

A empresa *Cadence* adquiriu a *Verisity* em 2005 e iniciou o desenvolvimento de uma versão em *SystemVerilog* da eRM. Enquanto muitos conceitos relacionados à verificação foram herdados em *SystemVerilog*, alguns precisavam de modificações ou melhorias em relação ao que existia em outras metodologias. Para tal, em 2007 foi introduzida a *Universal Reuse Methodology* (URM), a qual também era *open-source* e que utilizava TLM (HEIGHT, 2010).

A primeira metodologia de verificação amplamente utilizada, introduzida em 2005, foi a *Verification Methodology Manual* (VMM) (DOULOS, 2010). A VMM era a metodologia da *Synopsys* baseada na RVM para dar suporte à linguagem em crescimento para verificação *SystemVerilog*. É constituída de um conjunto de práticas cujo o foco é a criação de ambientes de verificação que possam ser reutilizados e que sejam construídos a partir dessa linguagem. A prática VMM pode explorar recursos como a orientação a objetos, randomização e cobertura funcional. Isso permite a criação de bons ambientes de verificação independentemente do nível de conhecimento do usuário (ALDEC, 2016)(DOULOS, 2010). As práticas incorporadas ao uso do VMM foram fatores importantes à criação do UVM.

Outro predecessor do UVM foi a *Open Verification Methodology* (OVM). Essa metodologia foi proposta em 2008, a partir de uma parceria das empresas a *Cadence* e *Mentor Graphics*. Trata-se de uma biblioteca de objetos e processos para a geração de estímulos, coleta de dados e controle de processos de verificação. Essa metodologia apresenta-se disponível para as linguagens *SystemVerilog* e *SystemC*. A OVM permite a fácil criação de testes direcionados ou aleatórios, utilizando comunicação em nível de transação e cobertura funcional . Como a primeira biblioteca baseada em *SystemVerilog* disponível em múltiplos simuladores, a OVM contribuiu significantemente para o desenvolvimento do

seu sucessor, a *Universal Verification Methodology* (UVM).

Finalmente, a *Universal Verification Methodology* (UVM) é uma biblioteca de código aberto de *SystemVerilog* que permite a criação de componentes de verificação flexíveis, reusáveis a partir da qual é possível a construção de ambientes de verificação robustos utilizando estímulos aleatórios e metodologias de cobertura funcional. A UVM é a combinação dos esforços de desenvolvedores e fornecedores de ferramentas, baseada no sucesso das metodologias OVM e VMM. Sua principal caracteristica é melhorar o reuso de *testbenches*, tornar o código de verificação mais portátil e criar um novo mercado universal para *Vefication IP* (*Intellectual Property*) de alta qualidade.

A Tabela 1, apresenta uma síntese das metodologias descritas:

Tabela 1 – Resumo das Metodologias de Verificação

Metodologia	Ano	Empresa	Principal característica
vAdvisor	2000	Verisity Design (Cadence)	Boas práticas de verificação; Atende a linguaguem e; Conceitos básicos de verificação;
eRM	2002	Verisity Design (Cadence)	Primeira biblioteca de verificação; Possui elementos estruturais e de arquitetura;
RVM	2003	Synopsys	Foca a linguagem Vera; Possui chamadas de função; Não possui elementos estruturais;
VMM	2005	Synopsys	Baseada na RVM; Amplamente utilizada;
AVM	2006	Mentor	Uso de classes de testes; Primeira OpenSource;
URM	2007	Cadence	Versão SystemVerilog da eRM; OpenSource;
OVM	2008	Cadence e Mentor	Primeira baseada em SystemVerilog; Opensource; Contribuiu com base para a UVM.
UVM	2011	Synopsys, Mentor e Cadence	Metodologia padrão; Reuso de componentes.

Tendo em vista as características da metodologia de verificação UVM, assim como o fato de ser um padrão do mercado, optou-se pela utilização da mesma para o projeto. A Seção 2.6, descrita a seguir, tem como foco elucidar as principais características quanto à estrutura, funções e organização geral dessa metodologia.

2.6 UNIVERSAL VERIFICATION METHODOLOGY

A *Universal Verification Methodology* (UVM), ou Metodologia de Verificação Universal, é uma metodologia que busca as melhores práticas para uma verificação exaustiva e eficiente. Um dos princípios da UVM é a reusabilidade de componentes de verificação, chamados de UVM *Verification Components* (UVCs), os quais são uma abstração dos

estímulos e monitoramento necessários para verificar um componente de projeto, interface ou protocolo (CADENCE, 2017);

A aplicabilidade da UVM é cabível tanto para pequenos e grandes projetos (HEIGHT, 2010). Dessa forma, por existir uma necessidade de construção de ambientes robustos de verificação e que permitam reuso, a UVM apresenta-se como uma solução promissora para atender tais necessidades (SALAH, 2014).

A metodologia UVM é um complemento para a linguagem *SystemVerilog*, adicionando componentes como Macros que implementam métodos de dados padrões como *copy* e *compare* (SHIMIZU, 2013). Um dos motivos para isso é que *SystemVerilog* sozinha é insuficiente para uma ampla adoção das melhores técnicas de verificação, as quais motivaram o seu desenvolvimento (BROMLEY, 2013)(SALAH, 2014).

Como citado anteriormente, um dos principais objetivos da metodologia UVM é o emprego da reusabilidade. Essa característica da UVM visa a redução do tempo de mercado para circuitos integrados. Para alcançar esse objetivo, a metodologia foca em obter velocidade em verificação auxiliando os desenvolvedores a encontrarem erros mais cedo durante o processo de desenvolvimento. Além disso, é possível reutilizar o teste e os casos de teste. Finalmente, é importante ressaltar a independência em relação a fornecedores, pois todos os maiores provedores de ferramentas e linguagens para a criação de circuitos integrados suportam a metodologia UVM, o que não ocorria com as demais metodologias de verificação, tornando-a assim um padrão da indústria (SALAH, 2014) (FOSTER, 2015) (RAGHUVANSHI; SINGH, 2014).

A Tabela 2, adaptada de Salah (2014), descreve resumidamente a relação das empresas fornecedoras com a metodologia UVM. Nessa tabela, evidencia-se a relação entre as empresas fornecedores de metodologias focando o desenvolvimento da UVM, juntamente com os simuladores suportados pela metodologia, assim como as versões distribuídas.

Tabela 2 – Relação entre fornecedores e UVM. Adaptado de Salah (2014).

gray!25 Empresa	UVM $= \begin{cases} OVM, AVM\,(Mentor) \\ URM\,(Cadence) \\ VMM\,(Synopsis) \end{cases}$
gray!25 Simulador	UVM Suporta todos simuladores { Questa, IUS e VCS }
gray!25 Distribuições	UVM $= \begin{cases} UVM1.0 \\ UVM1.1\,(a, b, c, d) \\ UVM1.2 \end{cases}$

Para atingir as características de reuso propostas, a metodologia UVM é altamente modularizada. A Figura 10, apresenta um ambiente típico da metodologia UVM, onde estão representados os módulos principais da metodologia. Ressalta-se, que por se tratar de um ambiente padrão de verificação, é possível modificá-lo e adaptá-lo conforme o objetivo da implementação. Além disso essa estrutura base é representada na literatura em Araujo

(2015), Madan, Kumar e Deb (2015) e Gayathri et al. (2016) sendo o modelo escolhido para a pesquisa desenvolvida.

No ambiente descrito na Figura 10 pode-se perceber estruturas interligadas que representam o escopo do ambiente de verificação. Dentre tais elementos destaca-se o módulo denominado *Device Under Test* (DUT). O DUT, muitas vezes chamado de *Design Under Verification* (DUV), representa a descrição de *hardware* do usuário que está sob verificação.

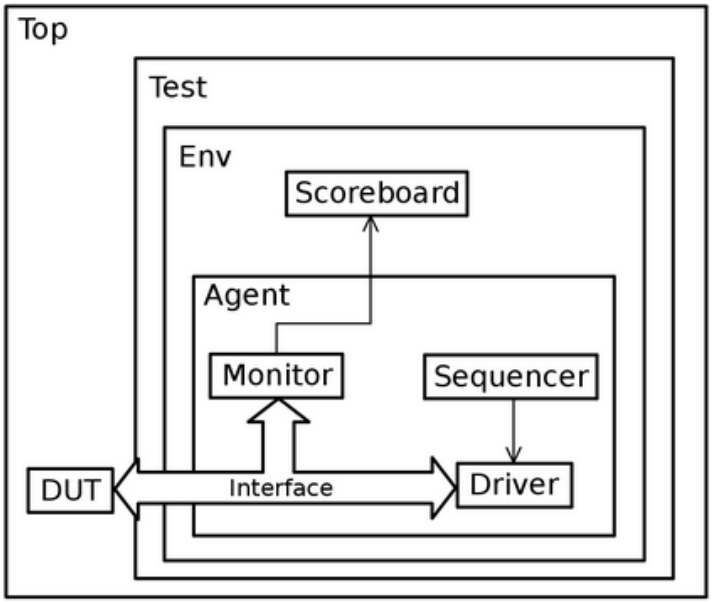

Figura 10 – Ambiente típico da metodologia UVM. Fonte: (ARAUJO, 2015)

A unidade sob verificação é interconectada por meio de uma interface ao módulo *Monitor* e ao módulo *Driver*, e por meio desses conecta-se indiretamente a todas a outras estruturas que formam esse ambiente.

O Módulo *Sequencer* é responsável por gerar estímulos para a realização da verificação dentro do UVM. A partir desse módulo cria-se os estímulos (entradas) que são enviadas via transações para o Módulo *Driver*. Esse, por sua vez, é responsável por enviar as sequências de estímulos geradas pelo Módulo *Sequencer*, por meio da interface, para o DUT, onde são processadas.

O Módulo *Monitor* é responsável por obter as entradas e saídas do DUT por meio da interface que liga o DUT ao *Driver*. Já o módulo *Scoreboard* testa os valores saídos do DUT com uma predição realizada através dos valores de entrada para atribuir valores verdadeiros ou falsos para os resultados obtidos pelo módulo testado.

A estrutura apresentada é um ambiente típico da metodologia. No entanto também pode ser construído de outras maneiras, como o não uso de monitores. Da mesma forma, cada uma das estruturas descritas possui uma organização interna com seus métodos e funções, sendo essa apenas uma visão geral sobre o escopo da metodologia.

3 TRABALHOS RELACIONADOS

Esse Capítulo tem como foco apresentar o estado da arte em relação à verificação de *hardware* utilizando *SystemVerilog* e a metodologia UVM. Além disso, apresenta as propostas de trabalhos cujo foco é auxiliar na construção desses ambientes. Para a obtenção de dados foram pesquisados artigos em diversas bases como: *IEEE Xplore Digital Library*, *ACM Digital Library*, *Springer Digital Library*, *Google Scholar*, com palavras chave como: *"UVM Generator", "SystemVerilog and UVM", "UVM Tool"* entre outras. Além disso, buscou-se na *web* por ferramentas com o foco em auxiliar a criação de ambientes de verificação UVM.

O trabalho de Bromley (2013) analisa o motivo da necessidade do uso da metodologia de verificação UVM, uma vez que se tem a linguagem *SystemVerilog* com esse foco. O trabalho exibe com clareza as capacidades da linguagem HDL como o suporte à programação orientada a objetos e características específicas para o apoio à verificação digital de *hardware*. Porém, ressalta que apesar de suas características, o *SystemVerilog* sozinho não foi capaz de impulsionar o uso de boas práticas em técnicas de verificação, sendo um dos motivos o seu tamanho. Somente o manual de referência possui 1300 páginas e a linguagem mais de 200 palavras reservadas, o que no principio fazia com que nenhuma ferramenta proprietária conseguisse prover uma implementação completa da linguagem, ocasionando problemas de reuso de código pois a variação entre ferramentas poderia fazer com que o código não funcionasse. Então, devido a essa e outras dificuldades para se obter boas práticas e reuso com *SystemVerilog*, começaram a surgir bibliotecas, conjunto de ferramentas e metodologia para atingir esse objetivo, entre as quais a UVM.

Bromley (2013) também apresenta as características da metodologia UVM e culmina no fato que a linguagem *SystemVerilog* por si só é completamente capaz de construir um ambiente de testes equivalente ao da metodologia. Entretanto, como citado, a HVL *SystemVerilog* é muito grande e o uso da UVM é importante para facilitar a aplicação de boas práticas. Como o autor cita, é análogo a utilizar funções na linguagem de programação C, como *'printf()'*. Muitos usuários acreditam que a função faz parte da linguagem base, entretanto é um componente de biblioteca. A construção de ambientes de verificação por meio da metodologia UVM segue a mesma linha, uma vez que é possível sim gerar os ambientes por meio de *SystemVerilog*, porém a integração da HVL com a metodologia gera um bom conjunto de práticas a serem utilizadas para a criação de ambientes de verificação.

No trabalho de Drechsler et al. (2014) verifica-se uma discussão entre vários especialistas da indústria (usuários e fornecedores), assim como acadêmicos sobre o futuro das metodologias de verificação em *Systems-on-chip* (SoCs), focando na metodologia UVM. Dentre os dados apresentados pelo trabalho destaca-se a Figura 11.

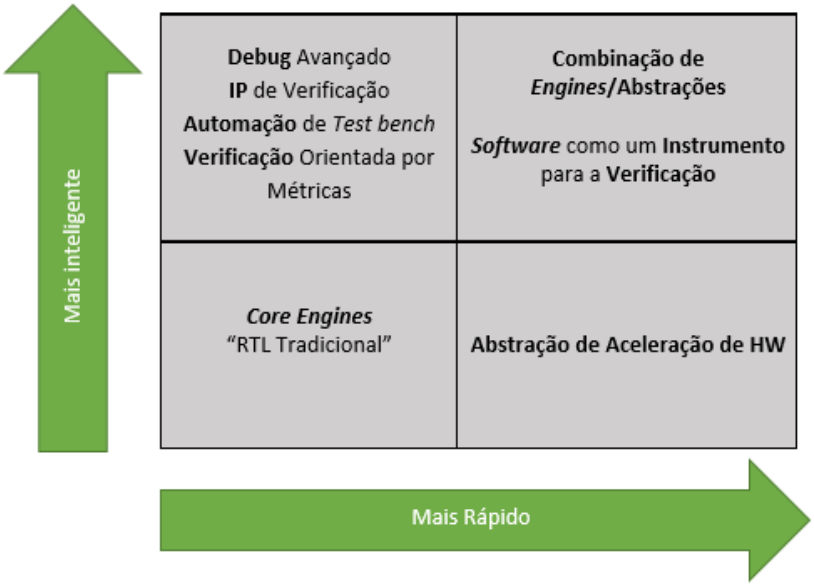

Figura 11 – Relação entre velocidade e estratégias de verificação. Fonte: (DRECHSLER et al., 2014)

É possível observar que entre as soluções mais inteligentes está a automatização da geração de *testbenches* e que entre as formas mais rápidas e, ao mesmo tempo, inteligentes, encontra-se o uso de *software* como instrumento de verificação. Esse tópico converge com a proposta definida na USAG, pois ela é um instrumento de *software* com foco em verificação, atuando como um sistema de apoio à geração de ambientes para essa finalidade.

No trabalho de PESSOA (2007) é proposto a geração de um ambiente de testes de forma semiautomática com foco no apoio ao processo de verificação funcional da metodologia *VeriSC*. A ferramenta proposta chama-se *Easy Testbench Creator* (eTBc), cuja essência é produzir ambientes de verificação de forma semiautomática.

A Ferramenta eTBc utiliza para a construção dos ambientes de verificação uma linguagem de definição de *templates* chamada de *eTBc-Template-Language* (eTL). Juntamente com a eTL, é gerado pelo autor um TLN (*Transaction Level Netlist*), que representa as descrições de dados em nível de transação ou seja, trata-se de uma representação dos módulos, transações e ligações entre módulos. A TLN é criada por meio de uma linguagem própria definida pelo autor como *eTBc Design Language* (eDL).

O TLN e o *template* gerado funcionam como entradas para a eTBc. A ferramenta então passa a funcionar como um gerador de código, possuindo dois compiladores internos, um para interpretação da TLN e outro para o *template* de entrada. Além disso, possui um gerador de código que funciona a partir do processamento das entradas pelos compiladores. O *tesbench* gerado pela eTBc necessita ser complementado por meio da implementação do plano de cobertura funcional, incluindo a geração de estímulos de acordo com as especificações do projeto. Além disso, é necessário a implementação de protocolos de comunicação (PESSOA, 2007). Esse trabalho, assemelha-se bastante com a estrutura da

USAG, pois ambos baseiam-se na entrada do usuário para a construção de ambientes de verificação e possuem caráter semiautomático para a geração desses ambientes, pois necessitam de complementação do ambiente para a execução da verificação. Porém, a ferramenta eTBc utiliza uma metodologia chamada *VeriSC*, que não é amplamente utilizada e uma linguagem própria para a construção do modelo de verificação, enquanto a USAG utiliza os padrões de linguagem e metodologias atuais no mercado.

Outra proposta semelhante à USAG é a *Unified Verification Environment* (UVE). A ferramenta UVE é uma ferramenta e uma biblioteca com o foco em projetar e gerar a estrutura de um *testbench* baseado em *SystemVerilog* e na metodologia UVM (BIANCHI; GUBLER, 2017). Apesar de gerar o *testbench* em *SystemVeriog*, a UVE utiliza como linguagem do DUT o VHDL. Já quanto ao suporte a sistemas operacionais, a UVE pode ser instalada em sistemas operacionais *Windows* ou *Unix*, porém para a execução dos ambientes gerados é necessário que se possua o simulador *Questa* instalado na máquina.

A Figura 12, extraída de Bianchi e Gubler (2017), apresenta a tela de principal e destaca os componentes básicos da ferramenta.

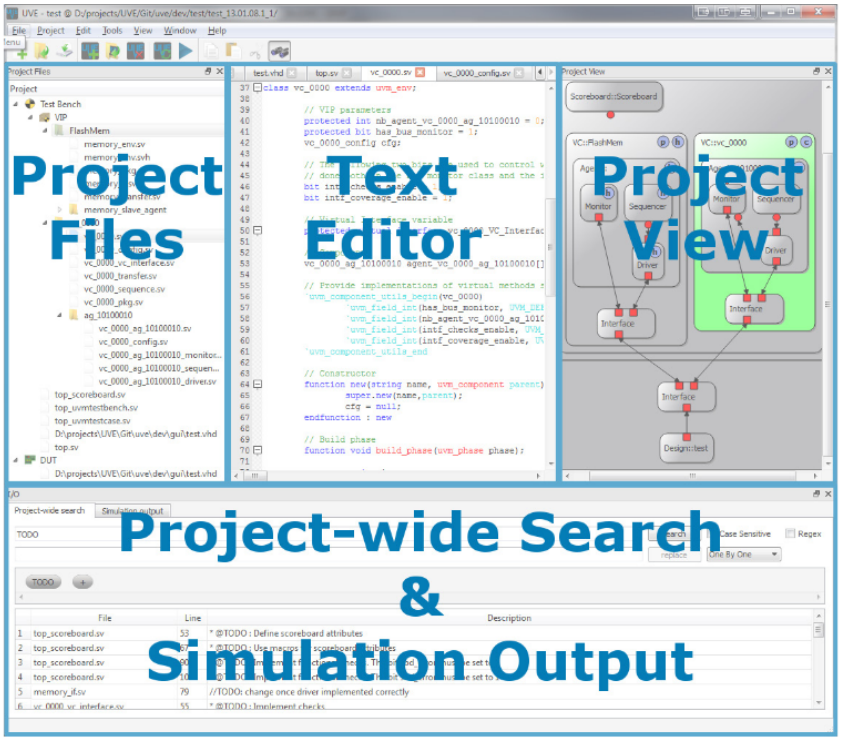

Figura 12 – Tela principal da UVE. Fonte: (BIANCHI; GUBLER, 2017)

Verifica-se que há uma seção para os arquivos do projeto, uma área para a edição de texto, um campo para pesquisa dentro do projeto e finalmente uma área que exibe graficamente a interligação entre os módulos. Da mesma forma que a eTBc de (SILVA, 2007), a UVE utiliza *templates* como entrada a fim de gerar as interconexões dos elementos da UVM. A UVE já traz consigo um conjunto de *templates*, facilitando a construção do ambiente de verificação. Os sinais de entrada do código fonte em VHDL são detectados

automaticamente pela ferramenta. Além disso, ela possui código aberto.

A UVE assemelha-se muito à USAG em alguns aspectos, como o caráter semi-automático na construção dos ambientes, a detecção automática dos sinais de entrada do código fonte, a edição de código diretamente através da interface da ferramenta, e a interconexão automática entre componentes da UVM (apesar da UVE necessitar o uso de *templates* para isso).

Trabalhando de forma análoga, o EasierUVM (DOULOS, 2010) propõe facilitar a construção dos ambientes de verificação UVM. Para tal, a ferramenta apresenta-se em duas versões, uma integrada a uma plataforma *web* da empresa *Doulos* chamada *edaplayground* e outra versão que pode-se efetuar o *download*. O gerador de código EasierUVM *offline* é um *script* na linguagem Perl e está distribuída sob a licença Apache 2.0. Entretanto, a versão *online* só pode ser utilizada dentro da plataforma, não sendo distribuída. Tanto a versão em *script* (que não possui interface gráfica) quando a versão *online* utilizam o mesmo sistema de *templates*, assim como a eTBc e a UVE. Para o EasierUVM são necessários pelo menos três arquivos de *templates* para a geração do ambiente de verificação: um *template* comum que é o arquivo principal para a geração de código, pelo menos um arquivo de interface (e um para cada interface adicional) e uma lista de pinos para definir quais sinais do DUT estão conectados em quais variáveis de interface. A Figura 13 exibe a interface gráfica da versão *web*:

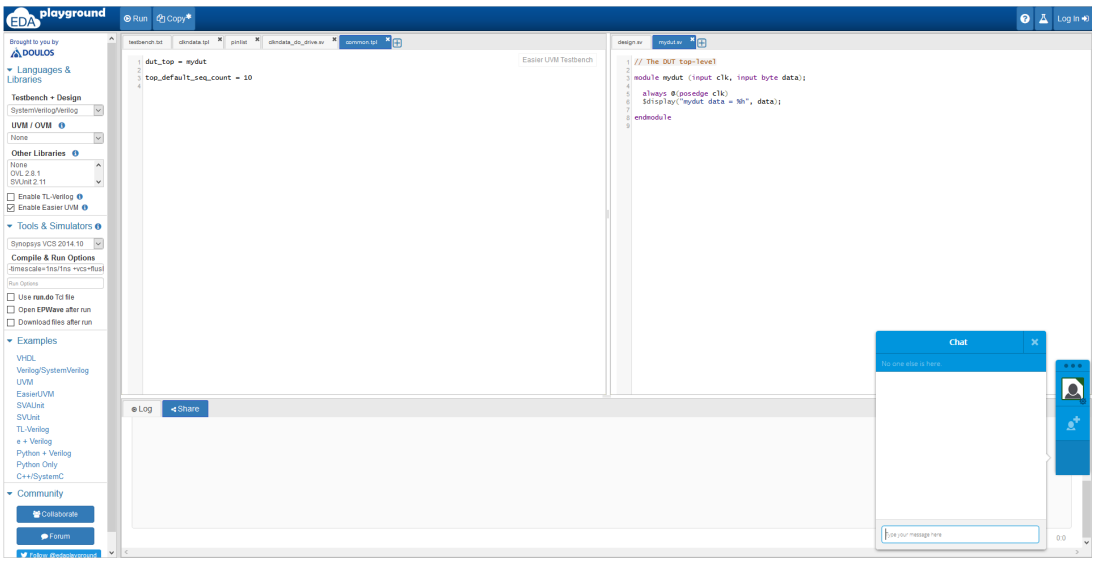

Figura 13 – Tela principal do Easier UVM *web*. Fonte: (DOULOS, 2010)

Entre as características interessantes da plataforma proposta por Doulos (2010) destaca-se a possibilidade de edição colaborativa na versão *web*, ou seja, dois ou mais usuários podem trabalhar em conjunto na edição de código na interface gráfica. Não obstante, realiza o armazenamento *online* dos projetos permitindo acesso posterior e também possui simuladores integrados, como o VCS da empresa *Synopsys*. Uma desvantagem é a interface gráfica *web* não possuir responsividade, ou seja, não se adapta a tamanhos de tela

menores, como *netbooks, tablets* e *smartphones*. O EasierUVM relaciona-se diretamente com a USAG por ambas possuírem interface gráfica *web* e ambas manterem o arquivos na *web* para acesso posterior pelo usuário.

A Tabela 3 apresenta resumidamente o comparativo entre as características da USAG em relação às ferramentas encontradas na fase de pesquisa.

Tabela 3 – Comparação entre ferramentas

Característica	**USAG**	**UVE**	**EasierUVM**
Independente de Sistema Operacional	Sim	Não (Somente *Windows* ou *Linux*)	Sim (porém não adaptável a resoluções de tela pequenas).
Executa sem instalação na máquina	Sim	Não	Sim
Simulador embutido	Não	Sim (Somente para *Questa*)	Sim
Geração de Ambientes de forma semiautomática	Sim	Não, somente com *templates*	Não, somente com *templates*
Independe do uso de *templates*	Sim	Não	Não
Independe de conexão com a *internet*	Não (somente se executado em servidor local)	Sim	Sim, mas apenas em forma de *script*
HVL do DUT	*SystemVerilog*	VHDL	*SystemVerilog*
Backup de projetos na nuvem	Sim	Não	Sim
Construção automática das interconexões da UVM	Sim	Não, somente baseado em *templates*	Não, somente baseado em *templates*
Edição simultânea entre colaboradores	Não	Não	Sim

3.1 CONSIDERAÇÕES

Pode-se verificar, pela análise do estado da arte, que o uso de *SystemVerilog* juntamente com a metodologia UVM já é um padrão do mercado. Resalta-se que, exceto pela eTBc, as demais ferramentas analisadas são distribuídas pela indústria e não possuem trabalhos publicados nas bases de dados exploradas, onde as informações relativas às suas características foram apenas encontradas por meio da pesquisa na *web*. O Capítulo 4 apresenta os materiais e métodos utilizados para a construção da ferramenta USAG que explora as características de *SystemVerilog* juntamente com UVM apresentados.

4 A FERRAMENTA USAG

O trabalho objetiva criar uma ferramenta para geração de ambientes de verificação de *hardware* de forma semiautomática para a linguagem de síntese *SystemVerilog* e a metodologia de boas práticas de verificação UVM. É proposto por meio da ferramenta a construção de ambientes de verificação de forma ágil, sem a necessidade de um sistema operacional específico para sua execução e nem a instalação de quaisquer aplicativos na máquina do usuário. O presente Capítulo tem como objetivo apresentar a estrutura da ferramenta, de que forma realizou-se a sua construção e, finalmente, seu funcionamento.

4.1 ESTRUTURA E AMBIENTE DA FERRAMENTA

A ferramenta visa ser independente de sistemas operacionais e não necessitar a instalação/configuração de elementos pelo usuário para o seu funcionamento. A fim de atingir tal característica seu núcleo baseia-se em tecnologias *web* sendo a USAG distribuída como um *Software as a Service - SaaS*.

A USAG armazena no servidor dados do usuário, como informações de cadastro, projetos e conteúdos dos projetos (os dois últimos de forma opcional). Essa característica torna possível que os dados estejam acessíveis de qualquer dispositivo que tenha acesso à internet. Para o armazenamento e recuperação dos dados, optou-se por utilizar o sistema gerenciador de banco de dados (SGBD) MySQL.

Para a construção da lógica do sistema, ou seja, o lado servidor da aplicação (*back-end*) utilizou-se a linguagem de programação PHP. Do lado cliente da aplicação (*front-end*), para facilitar o uso da interface gráfica do sistema por parte do usuário utilizou-se a linguagem *Javascript*. Juntamente com o *Javascript* aplicou-se a biblioteca *JQuery* para obtenção dos efeitos e redução de código. Também utilizou-se as tecnologias HTML5 e CSS3, para estruturar e estilizar a USAG, respectivamente. Além disso, explorou-se o *framework bootstrap* para a construção do *layout* e para adicionar responsividade (ou seja a capacidade de adaptação a diferentes tamanhos de tela) ao sistema.

Primeiramente, modelou-se a base de dados da USAG. Construiu-se as tabelas para o armazenamento das informações a partir da tecnologia MySQL. Essa base foi chamada de *usag_db* e possui 3 estruturas principais:

- ***usag_user*** – Armazena os dados de usuário.
- ***user_has_projects*** – Relaciona o usuário com os projetos construídos por ele dentro da ferramenta.
- ***project_has_files*** – Une cada um dos arquivos gerados com o seu respectivo projeto. Salienta-se que na base de dados são armazenados apenas os nomes e diretórios dos arquivos, sendo seu conteúdo armazenado diretamente no servidor.

Isto torna a busca na base mais leve, uma vez que não é necessário o carregamento de vários arquivos para a execução da pesquisa.

A Figura 14 mostra um modelo Entidade-Relacionamento do banco de dados da USAG.

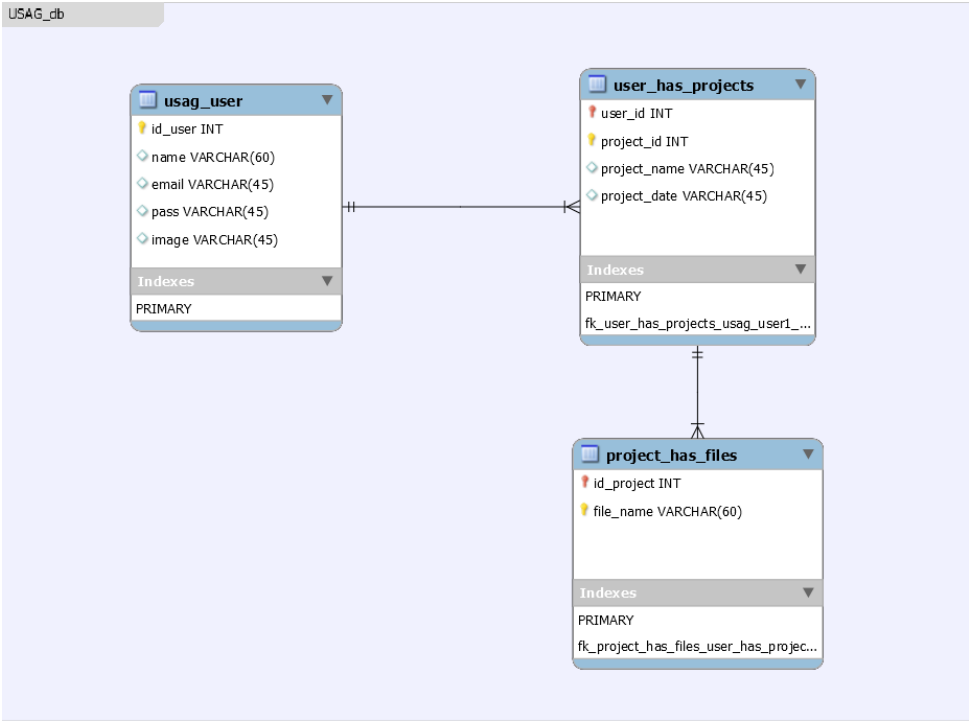

Figura 14 – Modelo Entidade-Relacionamento. Fonte: Autor.

A partir do modelo de banco de dados pode-se desenvolver os componentes da USAG. A USAG utiliza a técnica de caixa preta, sendo necessário apenas que se possua a estrutura padrão de código *SystemVerilog*, como apresentado na Figura 15, contendo os sinais de entrada e saída. Recomenda-se utilizar o DUT completo para facilitar na futura execução do código por parte do usuário, entretanto para o funcionamento adequado da ferramenta não é necessário.

```
1  module moduloExemplo (input entrada_1,
2                        input wire entrada_2,
3                        output reg saida_1);
4
```

Figura 15 – Exemplo de Código de Entrada. Fonte: Autor.

Com o intuito de obter uma maior organização de código e manutenibilidade do mesmo, para a geração de ambientes por meio da USAG criou-se a seguinte estrutura de módulos:

- ***Catcher*** – Responsável por capturar a entrada de dados do usuário e retornar a região que contém os sinais de entrada e saída, utilizados para a construção do ambiente de verificação.

- ***Interpreter*** – Com a dados obtidos pelo *Catcher* por meio desse módulo define-se os sinais as entradas e saídas do DUT.

- ***Generator*** – Por meio das entradas e saídas geradas pelo módulo *Interpreter*, constrói-se o ambiente de verificação.

A execução simplificada desses módulos base é apresentado através do diagrama de processos descrito na Figura 16.

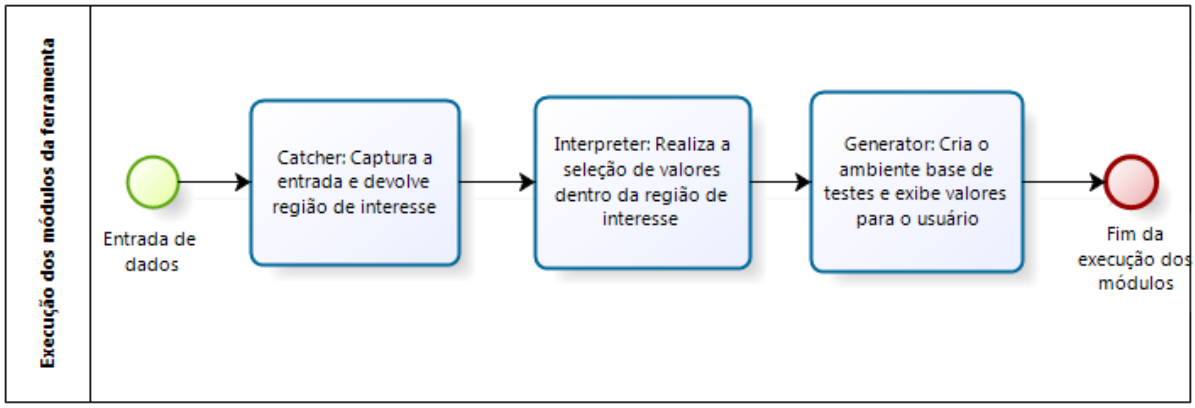

Figura 16 – Fluxo de execução da USAG. Fonte: Autor.

4.1.1 MÉTODO *CATCHER*

O método *Catcher* busca definir a região dentro do código fonte de entrada em *SystemVerilog* que contém os sinais do DUT.

A partir da entrada do usuário, percorre-se o conteúdo do código buscando a palavra reservada da linguagem *module* que indica o início de um novo módulo e consequentemente o início do DUT. Após detectada região inicial de interesse detecta-se o fechamento dela, determinada pelo símbolo de ";". Essa região é então utilizada como entrada para o método *Interpreter*.

```
364     //File name search begin
365     var nameFileBegin = str.search("module");
366     // Interface Methods
367     if(nameFileBegin != 0){ ...
370     } else {
371     //File name search end
372     var nameFileEnd = str.search("\\(");
373
374     nameFile = nameFileEnd - nameFileBegin;
375     name = str.substr(nameFileBegin,nameFile);
376
```

Figura 17 – Código para detecção do nome. Fonte: Autor.

O *Catcher* também é responsável por obter o nome do módulo e isso é feito de forma semelhante a detecção da região com os sinais do DUT. Define-se o intervalo entre a palavra reservada *module* e o primeiro parêntese encontrado no código, como pode ser visualizado na 17. Dessa região, remove-se os espaços em branco e define-se o conteúdo obtido como o nome do módulo, que posteriormente será aplicado para a construção dos códigos do ambiente de verificação.

4.1.2 MÉTODO *INTERPRETER*

O método *Interpreter* tem como objetivo a tradução dos dados capturados pelo *Catcher* de forma que os mesmos tenham sentido para a construção do ambiente de verificação. No intervalo de dados obtido, aplica-se um conjunto de expressões regulares para a remoção de palavras reservadas e espaços em branco, alguns exemplos podem ser visualizados na Figura 18.

```
399     //Remove os input
400   var re = new RegExp('(?:^|\\s)(input)(?=\\s|$)', 'g');
401     var newstr = input.replace(re, '');
402
403     //Remove os output
404     var re = new RegExp('(?:^|\\s)(output)(?=\\s|$)', 'g');
405     var newstr = newstr.replace(re, '');
406
407     //Remove os wire
408     var re = new RegExp('(?:^|\\s)(wire)(?=\\s|$)', 'g');
409     var newstr = newstr.replace(re, '');
410
```

Figura 18 – Exemplos de expressões regulares utilizadas. Fonte: Autor.

Ao final da remoção das palavras reservadas e dos espaços vazio agrupa-se essas palavras em um vetor o qual é então encaminhado para o método *Generator*.

4.1.3 MÉTODO *GENERATOR*

O método *Generator* usa o vetor de entradas recebidas do *Interpreter* e tem como principal função a construção do escopo do ambiente de verificação UVM.

Primeiramente, com o uso dos valores contidos no vetor, elabora-se o conteúdo do módulo chamado *Interface*. A *Interface* na UVM é responsável pela conexão entre o DUT e as outras estruturas da metodologia. Utiliza-se para esse processo o nome do módulo que foi detectado pelo *Catcher* juntamente com o vetor criado pelo *Interpreter*. A partir desses valores obtém-se um arquivo chamado *"nomeDoModulo_if.sv"*, é realizado o registro desse nome na base de dados e o conteúdo armazenado no servidor. Dentro do arquivo gerado, é armazenado o código fonte que possui os sinais vindos do *Interpreter* juntamente com as estruturas do componente da UVM e quaisquer alterações que o usuário tenha realizado.

Após a criação do componente *Interface* o próximo componente obtido por meio da USAG é o *Sequencer*. Por meio do *Sequencer* o usuário irá enviar estímulos para os demais módulos. Para a construção do *Sequencer* utiliza-se o como entrada o vetor de sinais detectado pelo *Interpreter* para apresentar um menu para o usuário decidir quais sinais ele deseja utilizar como entrada para a estrutura UVM. Além disso, esse menu apresenta a primeira interação direta do usuário com o sistema. Não é possível saber de forma automática quais dos sinais detectados no DUT o usuário almeja testar. Também, não é possível saber qual o tipo de dado o usuário deseja utilizar para as entradas selecionadas. Na Figura 19 pode-se observar o código de criação desse menu, e na Figura 20 um exemplo de funcionamento do mesmo para o código fonte da Figura 15.

```
485 /*************************************************************************/
486
487 //Seleciona conteúdo do menu do sequencer
488
489 function trans_ini_content(sequencer_parts){
490 $("#target").empty();
491
492 console.log(sequencer_parts);
493
494 for (i = 0; i < sequencer_parts.length; i++) {
495   var re = new RegExp('\\[.*?\\]', 'g');
496     sequencer_parts[i] = sequencer_parts[i].replace(re, '');
497     sequencer_parts[i] = sequencer_parts[i].replace(/[()]/g, '');
498
499     var checkbox = $('<input type="checkbox" checked="checked" name="seq_item" value='+sequencer_parts[i]+' id='+i+'/>'+'</input>
500                       <input id="type_'+i+'" placeholder="Tipo de dado"></input>'+sequencer_parts[i]+'<br>');
501
502     checkbox.appendTo('#target');
503     $(".bs-example-modal-sm").modal('show');
504   }
505 }
506
507 /*************************************************************************/
```

Figura 19 – Código de geração do menu. Fonte: Autor.

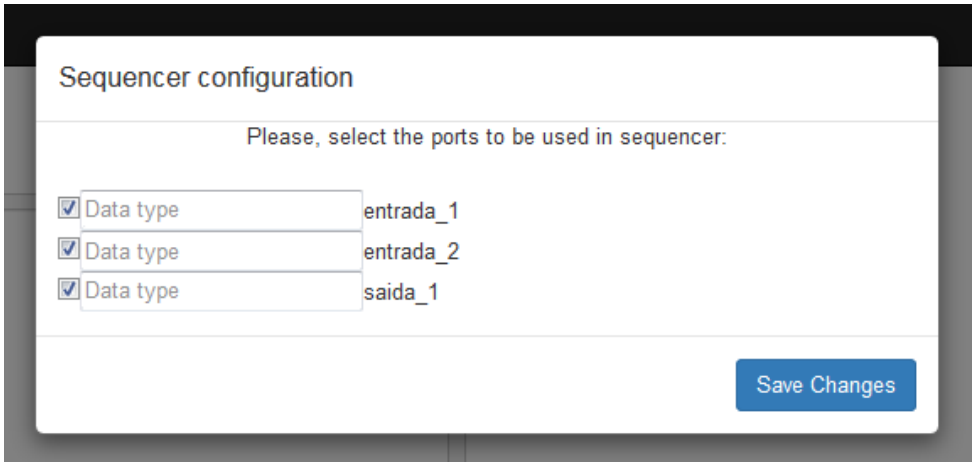

Figura 20 – Exemplo de funcionamento do menu. Fonte: Autor.

Com a definição do *Sequencer*, o método *Generator* utiliza novamente o nome do DUT detectado pelo *Catcher* para a construção do método *Driver* da UVM. O método *Driver*, irá receber os estímulos do *Sequencer* e é responsável por definir a forma como ele serão enviados para os demais módulos. Não é possível determinar a forma que o usuário deseja enviar esses estímulos, portanto, é gerada a estrutura do módulo (método construtor e demais componentes), a conexão entre os módulos e é reservado o espaço no código para que seja preenchido como o usuário deseja enviar essa informação.

Os estímulos enviados pelo *Driver* e as saídas obtidas do DUT após receber esses estímulos são observadas pelo módulo UVM chamado *Monitor*. Seguindo a estrutura padrão da metodologia proposta por (ARAUJO, 2015) e outros autores, utiliza-se na USAG dois monitores. Assim, com os dados obtidos a partir do *Catcher* é construído um *Monitor* para a saída do DUT e um para a saída do *Driver*. Para ambos, são geradas automaticamente as interconexões com os demais elementos da metodologia, iniciados os métodos construtores e funções da metodologia. Para o *Monitor* que observa as saídas do *Driver* também gera-se os elementos para cobertura funcional a partir do sinais selecionados pelo usuário no início do processo. Espera-se também, que seja criado pelo usuário no *Monitor* que recebe os sinais do *Driver*, uma predição para o resultado a ser recebido do DUT. A predição juntamente com o sinais de obtidos pelo DUT, serão utilizados como entrada para o *Scoreboard*.

O arquivo do *Scoreboard* é criado pelo *Generator*, contendo a interconexão entre os componentes juntamente com os elementos para a inicialização do módulo. É delimitado o espaço para o usuário aplicar a lógica que irá ser utilizada para a comparação entre os valores de resultados advindos do DUT, com os dados recebidos pela lógica implementada pelo usuário no módulo *Monitor*.

Como observa-se na Figura 10, os módulos obtidos por meio do *Generator* devem estar encapsulados em estruturas da metodologia UVM. Esse encapsulamento é uma característica utilizada para facilitar o reuso de componentes de verificação. O *Generator*

também cria essas estruturas, sendo gerado, a partir dos elementos de entrada do usuário, a interligação entre módulos e os métodos construtores do *Agent*, *Enviromnent*, *Test* e *Top Block*. Destaca-se o *Top Block*, no qual é necessário que o usuário gere estímulos, como sinais de *clocks*.

4.1.4 FERRAMENTAS DE INTERFACE

Com a construção dos módulos pelo método *Generator* é apresentado, por meio da interface gráfica da USAG, um conjunto de abas contendo os arquivos criados pela ferramenta. Em cada aba é exibido o conteúdo do código de verificação gerado, permitindo também a edição diretamente na tela. Além disso, é possível realizar a exclusão dos arquivos ao clicar no "x" contido nas abas. Esse menu de gerenciamento dos arquivos pode ser visualizado na Figura 21 com destaque para a opção de exclusão.

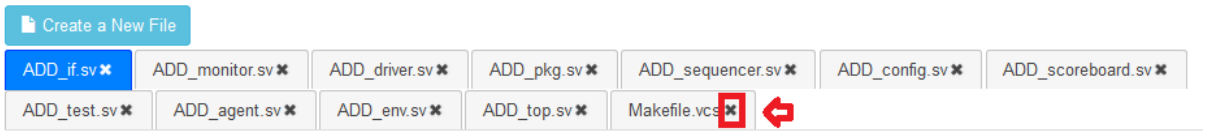

Figura 21 – Sistemas de abas. Fonte: Autor.

Após a exclusão de um arquivo por meio do sistema de gerenciamento de arquivos, o mesmo é removido do servidor e desvinculado da base de dados, não podendo mais ser recuperado. O sistema de exclusão foi implementado para permitir a personalização de arquivos para a criação do ambiente de verificação por parte do usuário. Assim, permite-se não só a exclusão de arquivos como também a criação manual dos mesmos por meio do botão *"Create New File"* o qual também pode-se visualizar na Figura 21.

Os projetos ficam armazenados no banco de dados, vinculados por usuário, permitindo o acesso posterior aos arquivos gerados pela USAG. Para obter acesso aos projetos, deve-se clicar sobre o nome de usuário e acessar a área *"My projects"*. Se não houver projetos registrados para o usuário, ele poderá realizar a criação de um novo projeto. Entretanto, havendo projetos, o mesmos são listados como pode-se evidenciar na Figura 22.

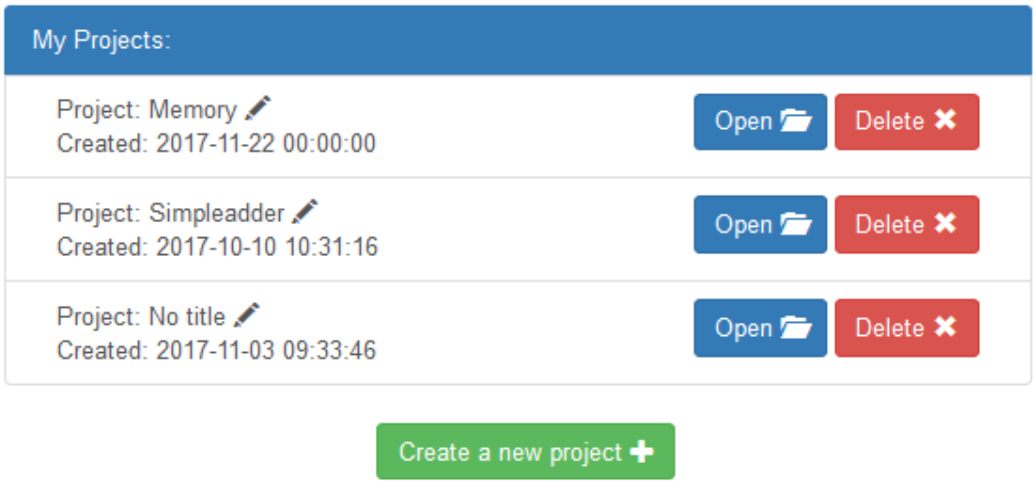

Figura 22 – Exibição de projetos. Fonte: Autor.

Por meio da área de gerenciamento de projetos é possível fazer o recarregamento de projetos por meio do menu *"Open"*. Dessa forma, os arquivos de projetos criados anteriormente são exibidos novamente nas mesmas abas como na primeira execução. Além disso, permite-se que por meio do botão *"Delete"* o usuário exclua seus projetos. Finalmente, nesse menu também permite-se que os projetos sejam renomeados, como na Figura 23.

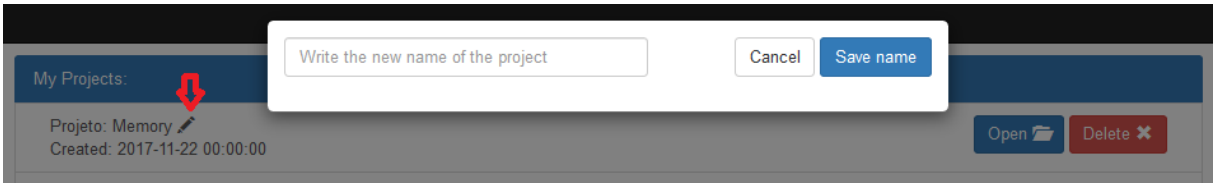

Figura 23 – Renomear projetos. Fonte: Autor.

4.2 CONSIDERAÇÕES

No presente Capítulo exibiu-se a lógica interna de construção da USAG e a forma como o usuário relaciona-se com a interface gráfica de utilização da ferramenta. Nesse sentido, pode-se verificar um contexto genérico do funcionamento da USAG para a geração dos ambientes de verificação UVM.

5 RESULTADOS

O presente Capítulo tem como objetivo apresentar os resultados do trabalho desenvolvido. Tendo em vista os objetivos definidos e a arquitetura elaborada construiu-se a USAG.

5.1 GERAÇÃO DE AMBIENTES

Após realizar o *login* na USAG e iniciar um projeto, o usuário está apto a gerar os ambientes de verificação UVM. A Figura 24, demonstra o exemplo do local destinado a entrada de dados preenchido por um código exemplo de um somador simples, simulando assim, a inserção de dados por um usuário.

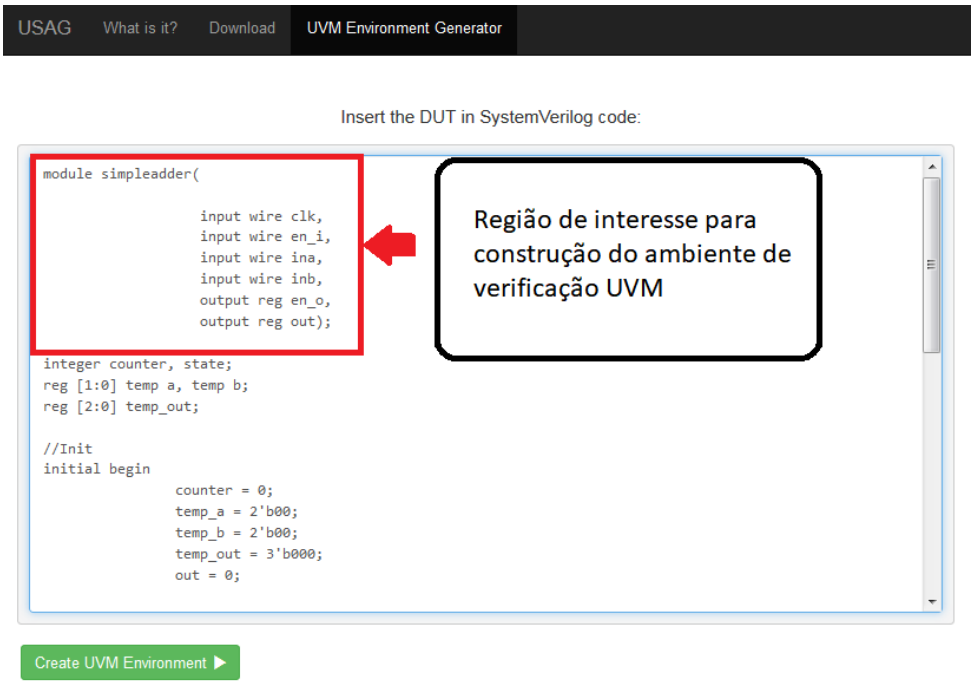

Figura 24 – Inserção de dados pelo usuário. Fonte: Autor.

Nota-se na Figura 24 que a região de interesse a ser obtida pelo *Catcher* está em destaque. Essa operação de seleção de região será disparada por meio do pressionamento do botão "*Create UVM Enviromnent*", disponível no canto inferior esquerdo do campo de entrada de dados. No código do usuário, para a geração do ambiente de verificação UVM, apenas a região destacada torna-se relevante. Isso se deve ao fato que a mesma contém as entradas e saídas do DUT, e a construção do ambiente de verificação utiliza a técnica de caixa-preta. Recomenda-se utilizar o código fonte completo como facilitador para simulações posteriores por parte do usuário. Entretanto, a USAG necessita apenas das entradas e saídas para a geração dos elementos de verificação da UVM.

Para a detecção da área de interesse, como já apresentado, o *Catcher* utiliza expressões regulares a fim de determinar onde inicia e termina o espaço que define as

variáveis de entrada e saída de dados para o DUT. Assim sendo, a partir do código fonte de um somador simples do exemplo, realizou-se o processo de verificação do componente, a fim de observar o correto funcionamento da USAG.

Após a inserção do código do somador simples na área de entrada, e pressionado o botão para gerar código espera-se que o módulo *Catcher* detecte corretamente a área de interesse e envie o resultado para o módulo *Interpreter*. O módulo *Interpreter*, por sua vez, deve receber o espaço capturado pelo *Catcher* e realizar o tratamento dos dados obtidos. As variáveis obtidas pelo *Interpreter* serão posteriormente aplicadas para a construção do ambiente de verificação. Para o código fonte sendo testado, seguindo o fluxo proposto pela USAG, espera-se obter como resultado a exibição dos sinais detectados para o usuário, assim como a opção de inserção de tipos de dados para o *Sequencer*, sendo essa a primeira interação do usuário com o código-fonte do ambiente que está sendo gerado. Pode-se observar, na Figura 25, que para o exemplo do somador esses valores são apresentados de forma correta.

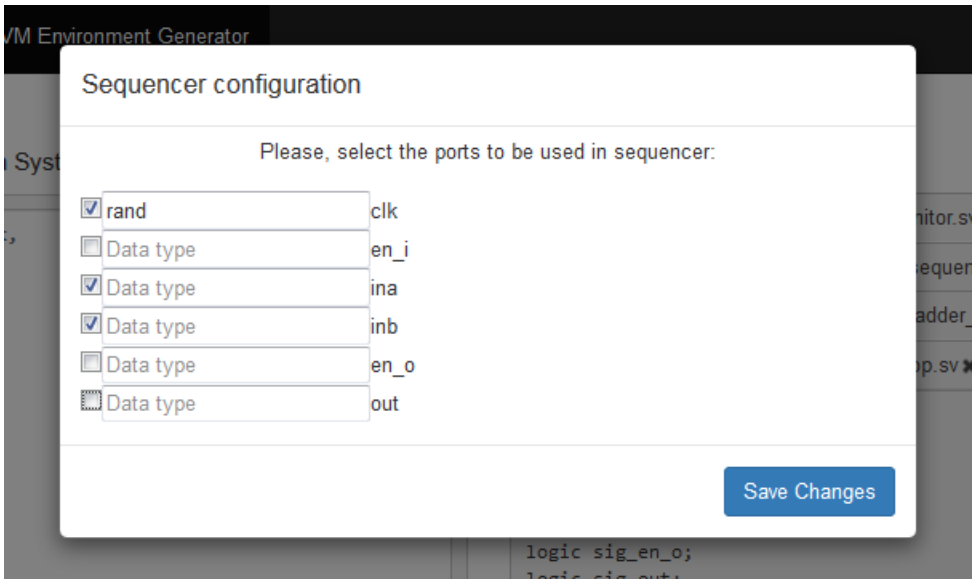

Figura 25 – Inserção de dados do usuário. Fonte: Autor.

No próximo passo da construção do ambiente, a partir das variáveis interpretadas é requisitado que o usuário insira valores a serem aplicados na construção do *Sequencer* do ambiente, como quais as variáveis devem ser utilizadas e o tipo de dados que serão contidos no mesmo. Na Figura 25, é possível notar essa seleção por meio de campos de *checkbox*. Os sinais selecionados serão utilizados para gerar a conexão com o DUT, e os demais serão desconsiderados. Esse passo é importante pois a partir do *Sequencer* os dados serão enviados para *Driver*, e essa parte será assistida pelo usuário, uma vez que o mesmo definirá a forma de envio. Assim sendo, tal seleção visa a simplificação da codificação por parte do usuário. Finalmente, após pressionado o botão "*Save*", o módulo *Generator* passa a processar todos os dados descritos.

O método *Generator* cria toda a estrutura típica da metodologia UVM, seguindo o modelo padrão descrito por diversos autores. Assim, essa estrutura é responsável por gerar todos os códigos e conexões entre os módulos da metodologia, e apresentar para o usuário. Para o código do somador simples, o resultado de uma das estruturas geradas é ilustrado na Figura 26.

Figura 26 – Código gerado para *interface*. Fonte: Autor.

Para a realização da verificação dos códigos obtidos por meio da USAG é necessário que haja a interação do usuário com alguns dos arquivos gerados. Os códigos que necessitam de interação com o usuário são:

- ***Sequencer***: No qual é necessário que seja preenchido a forma de envio dos dados para o *Driver*.
- ***Driver***: Onde é necessário a configuração de que forma os dados são encaminhados para o DUT (e.g. forma serial).
- ***Monitor***: Onde é necessário definir quais as variáveis irão ser observadas e a forma de predição utilizada pelo "*Scoreboard*".
- ***Scoreboard***: No qual é necessário se definir o teste de comparação entre os sinais recebidos pelo *Monitor*.
- ***Top Block***: O usuário deve apenas inicializar sinais como o *clock*, por exemplo.
- ***Makefile* (opcional)**: A ferramenta também gera um arquivo do tipo *Makefile* para facilitar a execução dos códigos gerados no simulador da *Synopsys*, o VCS. Nesse caso o usuário deverá alterar o caminho dos arquivos para seu diretório pessoal.

O preenchimento dos valores nesses arquivos é realizado com edição do código fonte gerado diretamente na interface gráfica da USAG.

Uma vez realizado o preenchimento dos campos necessários pelo usuário, o mesmo pode testar o código gerado. Para o DUT de exemplo utilizou-se o *script Makefile* gerado pela ferramenta juntamente com os códigos fontes. O ambiente de verificação testado foi então simulado na ferramenta *Synopsys* VCS.

A ferramenta *Synopsys* VCS foi executada em máquina virtual através do programa *Oracle VM Virtual Box*. A Figura 27 representa a execução do ambiente de verificação gerado.

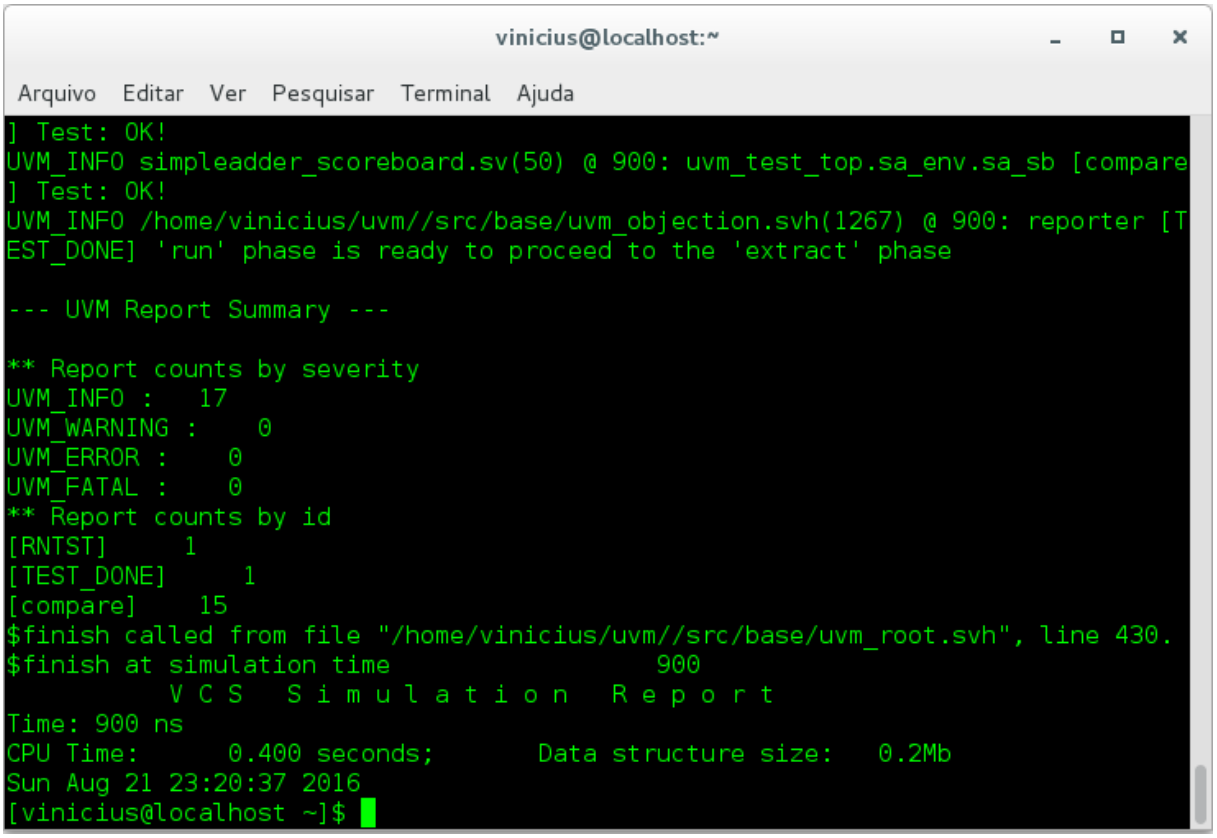

Figura 27 – Execução do código no simulador *Synopsys* VCS. Fonte: Autor.

Como pode-se observar na Figura 27, o código resultante do uso da ferramenta, com o preenchimento adequado dos dados pelo usuário, obteve um resultado satisfatório na execução da verificação.

Além desse teste de funcionamento, simulou-se, por meio da USAG, a construção de um ambiente de verificação para um modelo de memória, reproduzindo por meio da ferramenta o exemplo de (VERIFICATIONGUIDE, 2016). Para tal, utilizou-se como entrada para a USAG o código fonte do exemplo. Para garantir que o código fonte funcionasse de forma adequada na USAG removeu-se os comentários e parâmetros. O código fonte utilizado como entrada é apresentado na Figura 28. Pode-se observar também nessa figura algumas áreas em destaque que são as informações removidas.

```
//------------------------------------------------------------------------
//                Memory Model RTL - www.verificationguide.com
//------------------------------------------------------------------------
/*
                 -----------------
                 |                |
      addr ----->|                |
                 |                |------> rdata
                 | Memory Model   |
     wdata ----->|                |
                 |                |
                 -----------------
                      ^     ^
                      |     |
                    wr_en  rd_en

------------------------------------------------------------------------ */
module memory
  #( parameter ADDR_WIDTH = 2,
     parameter DATA_WIDTH = 8 ) (
    input clk,
    input reset,

    //control signals
    input [ADDR_WIDTH-1:0]  addr,
    input                   wr_en,
    input                   rd_en,

    //data signals
    input  [DATA_WIDTH-1:0] wdata,
    output [DATA_WIDTH-1:0] rdata
  );

  reg [DATA_WIDTH-1:0] rdata;

  //Memory
  reg [DATA_WIDTH-1:0] mem [2**ADDR_WIDTH];

  //Reset
  always @(posedge reset)
    for(int i=0;i<2**ADDR_WIDTH;i++) mem[i]=8'hFF;

  // Write data to Memory
  always @(posedge clk)
    if (wr_en)    mem[addr] <= wdata;

  // Read data from memory
  always @(posedge clk)
    if (rd_en) rdata <= mem[addr];

endmodule
```

Figura 28 – Código fonte da memória. Fonte: (VERIFICATIONGUIDE, 2016)

.

Ao inserir o código fonte na USAG e realizar a execução, o processo de detecção e apresentação dos sinais processados para o usuário é executado. A tela resultante dessa operação pode ser visualizada na Figura 29.

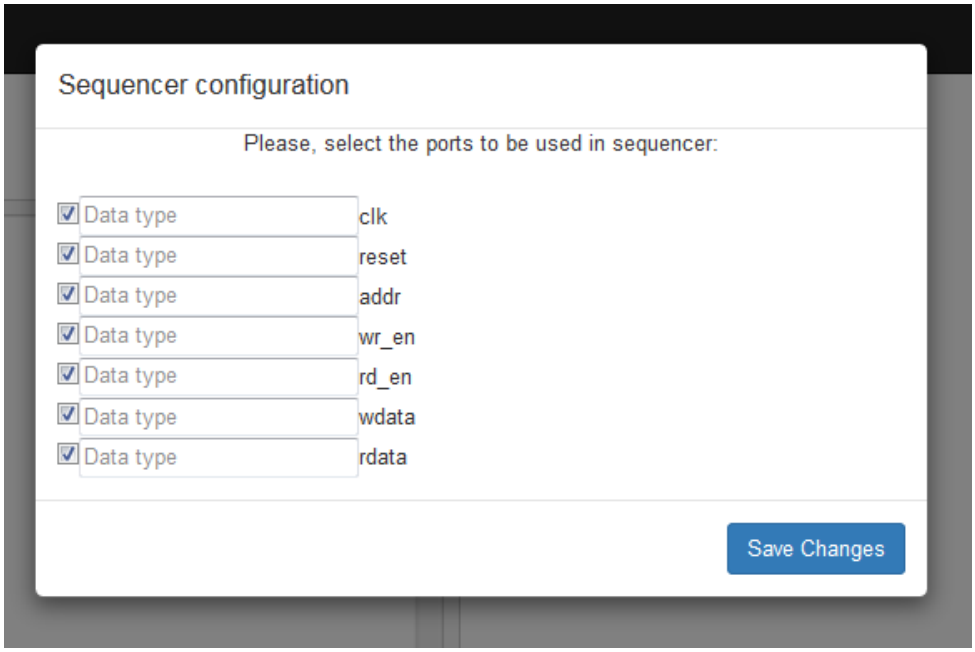

Figura 29 – Seleção de sinais. Fonte: Autor.

Analogamente ao módulo somador, após a seleção dos sinais, a USAG, através do *Generator*, gera os arquivos para que o usuário possa realizar a simulação. Nesse sentido, foram necessárias as seguintes ações manuais para obter o ambiente completo:

- ***Interface***: Foram adicionados os *Clocking Blocks* para atingir o sincronismo da memória, e adicionados os *Modports* utilizados pelo exemplo para agrupamento dos sinais.

- ***Monitor***: O exemplo utiliza apenas um monitor, não seguindo a estrutura convencional da metodologia UVM nesse ponto. Então, apagou-se diretamente na interface gráfica o trecho de código correspondente ao segundo *monitor*. Além disso, complementou-se a lógica de execução da *"Run Phase"* e do construtor.

- ***Sequencer/Sequence/Transaction***: No exemplo de memória apresentado por VerificationGuide (2016), os arquivos estão separados. No ambiente gerado pela USAG há apenas um arquivo para o *sequencer*, que engloba os demais. Para manter a estrutura original, separou-se os arquivos, utilizando para isso o menu *"New File"*.

- ***Scoreboard***: Complementou-se com o teste comparativo.

- ***Driver***: Definiu-se a lógica de envio dos dados.

- ***Agent***: O *Agent* da memória utiliza o método UVM *Collector* que é um elemento opcional para o monitoramento a nível de sinal e de transação. Esse método foi adicionado, juntamente com o complemento do método *run*.

- ***TopBlock***: Já no *Top Block* foram implementadas as gerações de *Clock*, e inseridos parâmetros da interface.

  Foi possível, apesar de haver interações com o código fonte criado pela USAG, com a base da estrutura gerada pelo DUT de entrada do usuário, a obtenção do ambiente final de verificação UVM. Esse ambiente, da mesma forma que o ambiente gerado pelo somador simples, demandou um menor esforço por parte do usuário, pois não foi necessário a construção manual de todo ambiente. Após a construção, foi realizada a simulação do ambiente gerado como observa-se na Figura 30.

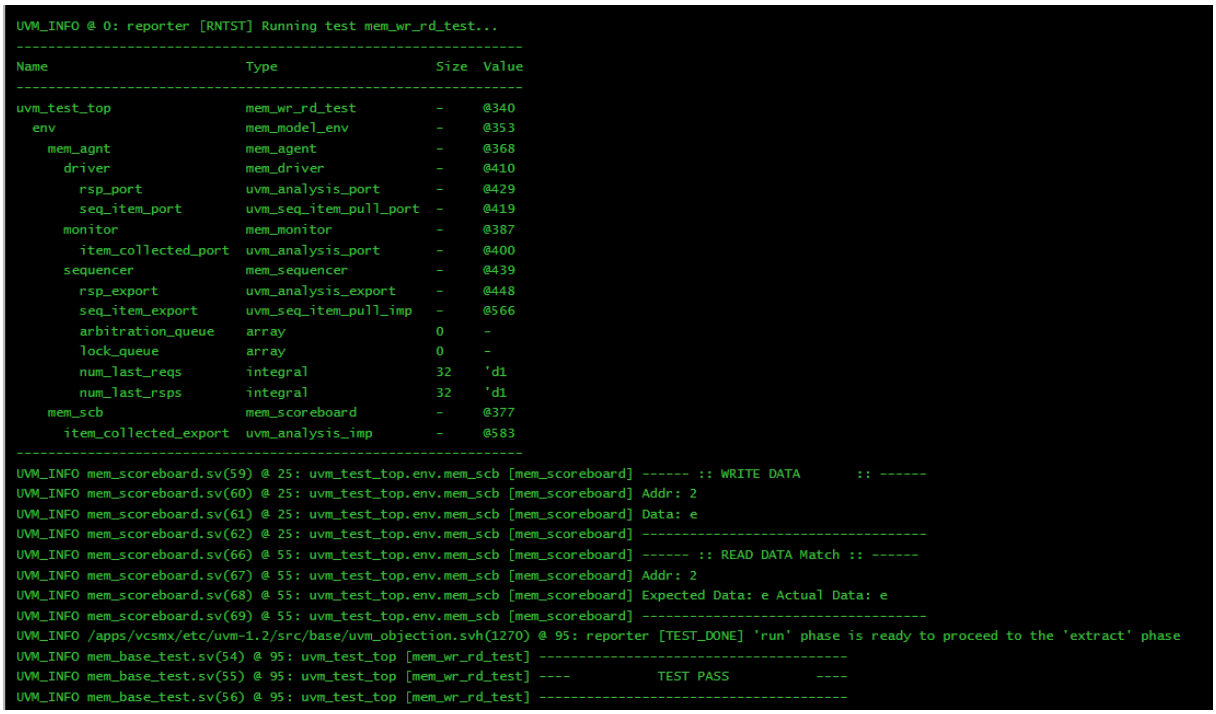

Figura 30 – Simulação da memória. Fonte: Autor

.

Além dos exemplos citados, gerou-se o ambiente de verificação para microprocessador desenvolvido pela Universidade Federal do Pampa - UNIPAMPA, chamado Pampium, que é definido por meio de uma arquitetura RISC com 80 instruções, utilizando operações apenas com registradores (ENGROFF, 2017). Para o Pampium, não realizou-se a complementação de dados pelo usuário e também não executou-se a simulação do ambiente gerado uma vez que esse não é o foco da USAG. A Figura 31 apresenta o código fonte para o *Top Block*, do microprocessador.

```
module MICRO_PROCESSADOR (input wire clk_in,rst,
                          inout wire[15:0] PORT_A,PORT_B);

logic [23:0] INST,INST_P;
logic [31:0] out_ms_out;
logic OVF_SN,OVF_SINST, OVF_SPC,OVF_ULA,ZERO, NEG,BIT;
logic  MS_PC,OP_SN,WE_N,WE_SP,WE_IR,SEL_BANK_1,SEL_BANK_2,MS_B2,MS_A2,MS_BW,WE_R,WE_STATUS,MS_OP1,WE_DM;
logic [1:0] MS_APC,MS_B1,MS_A1,MS_AW,MS_IN,MS_OP2;
logic [3:0]WE_PC,OP_ULA,out_reg_n,out_add_n;
logic [4:0]ADDR_1,ADDR_2,ADDR_W,out_ms_a1,out_ms_a2,out_ms_aw;
logic [3:0]addr_SP;
logic we_pc_at,out_ms_b1,out_ms_b2,out_ms_bw;
logic [16-1:0]out_ms_pc,addr_inst,out_add_sp_inst,out_sp,out_add_pc,out_ms_apc;
logic [15:0]OUT1,OUT2,out_ms_op1,out_ms_op2,RES,out_ms_in,out_dm,r_ana,r_ana_1;
logic int_cod_1,int_cod_2,c0, c1, c2, c3, c4, c5, c6, c7, clk_d1, clk_d,crt_clk;
logic clk;
logic [16:0] a;
div_clk c_clk1(clk_in, rst,clk);
CONTROLE Controle(INST[23:18],OVF_SN, OVF_SPC,OVF_ULA,ZERO, NEG,BIT,clk,rst,MS_PC,OP_SN,WE_N,WE_SP,WE_IR,
    SEL_BANK_1,SEL_BANK_2,MS_B2,MS_A2,MS_BW,WE_R,WE_STATUS,MS_OP1,WE_DM,MS_APC,MS_B1,MS_A1,MS_AW,MS_IN,MS_OP2,WE_PC,OP_ULA,ADDR_1,ADDR_2,ADDR_W);
MUX_WE_PC MUX_WE_PC(WE_PC,1'b0,1'b1,ZERO,(ZERO||NEG),!ZERO,((!NEG)||ZERO),!(ZERO || NEG),!(ZERO || !NEG),BIT,!BIT,we_pc_at);
REG_N1 #(16,16'b0) REG_PC(clk,rst,we_pc_at,out_ms_pc,addr_inst);
ADD_ST_INST #(16) ADD_ST_INST(addr_inst,{9'b0,1'b1},OVF_SINST,out_add_sp_inst);
STACK #(16) SP(clk,rst,WE_SP,addr_SP,out_add_sp_inst,out_sp);
MUX_2 #(4,4) MUX_INTER_N(OP_SN,out_reg_n,out_add_n,addr_SP);
REG_N #(4,4'b0)  REG_N(clk,rst,WE_N,out_add_n,out_reg_n);
ADD_ST ADD_ST(out_reg_n,4'b0001,OP_SN,OVF_SN,out_add_n);
MUX_2 #(16,16) MUX_PC(MS_PC,out_sp,out_add_pc,out_ms_pc);
ADDR_PC #(16) ADD_PC(out_ms_apc,addr_inst,OVF_SPC,out_add_pc);
M_Inst #(16) IM(clk,rst,addr_inst,INST);
//REG_N #(24,24'b0) IR(clk,rst,WE_IR,INST_P,INST);
MUX_4 #(16,16)  MUX_MS_APC(MS_APC,INST[15:0],OUT1,{15'b0,1'b1},{15'b0,1'b0},out_ms_apc);
MUX_2 #(16,16)  MUX_OP1 (MS_OP1,OUT1,{6'b0,INST[9:0]},out_ms_op1);
MUX_4 #(16,16)  MUX_OP2 (MS_OP2,{6'b0,INST[14:5]},{11'b0,INST[9:5]},{6'b0,INST[9:0]},OUT2,out_ms_op2);
ULA ULA(OP_ULA,out_ms_op1,out_ms_op2,RES,NEG,ZERO,OVF_ULA,BIT);
MUX_4 #(1,1)  MUX_B1 (MS_B1,SEL_BANK_1,INST[17],INST[15],1'b0,out_ms_b1);
MUX_4 #(5,5)  MUX_A1 (MS_A1,ADDR_1,INST[14:10],INST[4:0],5'b0,out_ms_a1);
MUX_2 #(1,1)  MUX_B2 (MS_B2,SEL_BANK_2,INST[16],out_ms_b2);
MUX_2 #(5,5)  MUX_A2 (MS_A2,ADDR_2,INST[9:5],out_ms_a2);
MUX_2 #(1,1)  MUX_BW (MS_BW,INST[17],INST[15],out_ms_bw);
MUX_4 #(5,5)  MUX_AW (MS_AW,INST[4:0],INST[14:10],ADDR_W,5'b0,out_ms_aw);
MUX_4 #(16,16)  MUX_IN (MS_IN,INST[15:0],out_dm,RES,OUT1,out_ms_in);
DM DM(clk,WE_DM,OUT1,RES,out_dm);
BANK_REG BANK_REG(clk,rst,WE_R,WE_STATUS,out_ms_bw,out_ms_b1,out_ms_b2,out_ms_a1,out_ms_a2,out_ms_aw,out_ms_in,{9'b0,OVF_SPC,OVF_SINST,OVF_SN,
    OVF_ULA,NEG,ZERO,BIT},OUT1,OUT2,r_ana,r_ana_1, PORT_A,PORT_B);
endmodule
```

Figura 31 – Código fonte *Pampium*. Fonte: (ENGROFF, 2017)

.

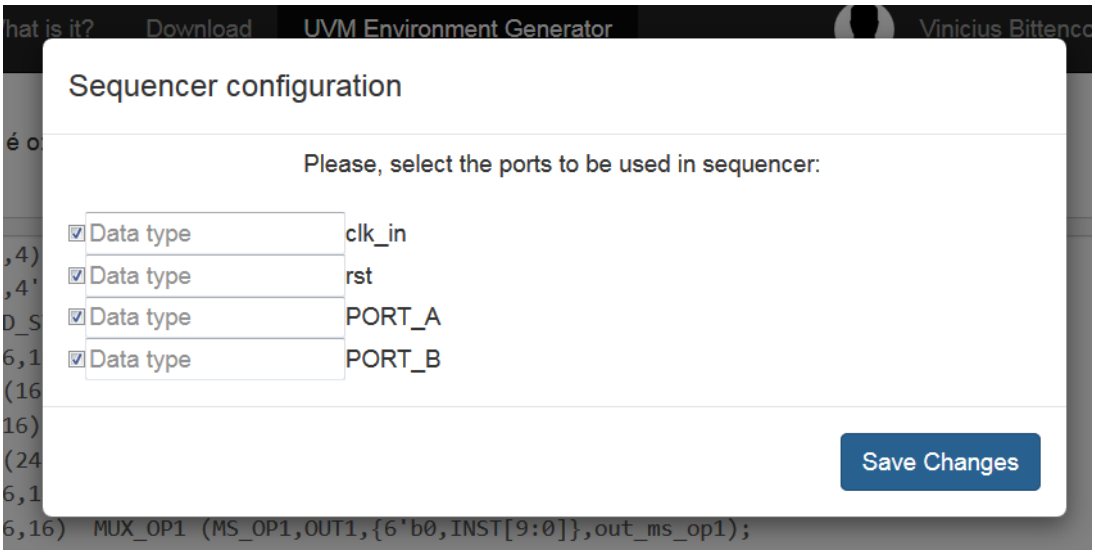

Figura 32 – Seleção de sinais para o *Sequencer*. Fonte: Autor

.

Esse código-fonte foi inserido na USAG por meio do campo de entrada na interface gráfica. Ao realizar o processamento desse código de entrada, apresentou-se o menu de seleção de sinais pelo usuário, evidenciando-se assim, que as entradas foram detectadas de forma correta como pode ser visualizado na Figura 32.

Após realizada a seleção de sinais, os arquivos gerados são exibidos para o usuário. Apresenta-se cada um dos arquivos gerados pela USAG para esse ambiente de verificação.

Pode-se verificar na Figura 33 o arquivo gerado para a interface. Como pode-se perceber, a USAG, gerou automaticamente a instanciação do código fonte da interface, inserindo os sinais detectados pelo método *Catcher*.

```
interface  MICRO_PROCESSADOR_if;
logic sig_clk_in;
logic sig_rst;
logic [15:0] sig_PORT_A;
logic [15:0] sig_PORT_B;
endinterface:  MICRO_PROCESSADOR_if
```

Figura 33 – Interface gerada para o microprocessador. Fonte: Autor

.

Na Figura 34 é possível observar o código gerado pela USAG, para a criação dos monitores do ambiente de verificação. Nesse cenário pode-se verificar que são estanciados dois monitores. O primeiro, composto de elementos como *uvm_component_utils* que registra a classe na UVM, uma *uvm_analysis_port* utilizada para realizar a transmissão dos dados, uma interface virtual e a configuração na base de configurações (*uvm_config_db*) de forma a se comunicar com o DUT, assim como os demais elementos, como construtores e elementos das fases de execução UVM. Por outro lado, o segundo monitor, além desses elementos, possui o código gerado pela USAG para realizar a cobertura funcional dos sinais detectados.

```
class MICRO_PROCESSADOR_monitor_before extends uvm_monitor;
`uvm_component_utils(MICRO_PROCESSADOR_monitor_before)
uvm_analysis_port#(MICRO_PROCESSADOR_transaction) mon_ap_before;
virtual MICRO_PROCESSADOR_if vif;
function new(string name, uvm_component parent);
super.new(name, parent);
endfunction: new
function void build_phase(uvm_phase phase);
super.build_phase(phase);
void'(uvm_resource_db#(virtual MICRO_PROCESSADOR_if)::read_by_name (.scope("ifs"), .name("MICRO_PROCESSADOR_if"), .val(vif)));
mon_ap_before = new(.name("mon_ap_before"), .parent(this));
endfunction: build_phase
task run_phase(uvm_phase phase);
//Code The run phase
 endtask: run_phase
 endclass:MICRO_PROCESSADOR_monitor_before


class  MICRO_PROCESSADOR_monitor_after extends uvm_monitor;
`uvm_component_utils(MICRO_PROCESSADOR_monitor_after)
uvm_analysis_port#(MICRO_PROCESSADOR_transaction) mon_ap_after;
virtual MICRO_PROCESSADOR_if vif;
MICRO_PROCESSADOR_transaction sa_tx;
MICRO_PROCESSADOR_transaction sa_tx_cg;
covergroup MICRO_PROCESSADOR_cg;
clk_in_cp: coverpoint sa_tx_cg.clk_in;
rst_cp: coverpoint sa_tx_cg.rst;
PORT_A_cp: coverpoint sa_tx_cg.PORT_A;
PORT_B_cp: coverpoint sa_tx_cg.PORT_B;
cross clk_in_cp,
rst_cp,
PORT_A_cp,
PORT_B;
endgroup: MICRO_PROCESSADOR_cg
function new(string name, uvm_component parent);
super.new(name, parent);
MICRO_PROCESSADOR_cg = new; endfunction: new function void build_phase(uvm_phase phase);
super.build_phase(phase);void'(uvm_resource_db#(virtual MICRO_PROCESSADOR_if)::read_by_name(.scope("ifs"), .name("MICRO_PROCESSAD
OR_if"), .val(vif)));
mon_ap_after= new(.name("mon_ap_after"), .parent(this));endfunction: build_phase
task run_phase(uvm_phase phase);
//Code here
endtask: run_phase
endclass:MICRO_PROCESSADOR_monitor_after
```

Figura 34 – Monitores gerados para o microprocessador. Fonte: Autor

.

Outro arquivo gerado pela USAG para o código do microprocessador Pampium é o arquivo para o *Sequencer*. A Figura 35, apresenta o código fonte obtido onde pode-se atentar para além das estruturas base como construtores, a declaração de elementos do tipo *'uvm_field* para os sinais detectados. Esse tipo de declaração permite que esses sinais utilizem os principais métodos para manipulação de dados da UVM. Além disso, há o espaço para preenchimento da lógica de como serão enviandas as transações para o *Driver*.

É possível verificar na Figura 36, o arquivo produzido pela USAG para o *Driver*. Nota-se que são criadas pela a USAG os construtores e elementos de comunicação com os demais elementos da UVM, restando apenas o espaço para a inserção da lógica do envio das informações.

```
class MICRO_PROCESSADOR_transaction extends uvm_sequence_item;
wire clk_in;
wire rst;
wire [15:0] PORT_A;
wire [15:0] PORT_B;

function new(string name = "");
super.new(name);
endfunction: new
`uvm_object_utils_begin(MICRO_PROCESSADOR_transaction)
`uvm_field_int(clk_in, UVM_ALL_ON)
`uvm_field_int(rst, UVM_ALL_ON)
`uvm_field_int(PORT_A, UVM_ALL_ON)
`uvm_field_int(PORT_B, UVM_ALL_ON)
`uvm_object_utils_end
 endclass:MICRO_PROCESSADOR_transaction

class MICRO_PROCESSADOR_sequence extends uvm_sequence#(MICRO_PROCESSADOR_transaction);

`uvm_object_utils(MICRO_PROCESSADOR_sequence)
function new(string name = "");
super.new(name);
endfunction: new

task body();
MICRO_PROCESSADOR_transaction MICRO_PROCESSADOR_tx;

//Complete the sequencer logic

MICRO_PROCESSADOR_tx = MICRO_PROCESSADOR_transaction::type_id::create(.name("MICRO_PROCESSADOR_tx"), .contxt(get_full_name()));
start_item(MICRO_PROCESSADOR_tx)


finish_item(MICRO_PROCESSADOR_tx);end
endtask: body
endclass:MICRO_PROCESSADOR_sequence
typedef uvm_sequencer#(MICRO_PROCESSADOR_transaction)MICRO_PROCESSADOR_sequencer;
```

Figura 35 – *Sequencer* gerado para o microprocessador. Fonte: Autor

.

```
class MICRO_PROCESSADOR_driver extends uvm_driver#(MICRO_PROCESSADOR_transaction);
`uvm_component_utils(MICRO_PROCESSADOR_driver)

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

function void build_phase(uvm_phase phase);
super.build_phase(phase);
void'(uvm_resource_db#(virtual MICRO_PROCESSADOR_if)::read_by_name(.scope("ifs"), .name("MICRO_PROCESSADOR_if"), .val(vif)));
endfunction: build_phase
task run_phase(uvm_phase phase);
drive();
endtask: run_phase
virtual task drive();
//Define the drive's logic

endtask: drive
endclass:MICRO_PROCESSADOR_driver
```

Figura 36 – *Driver* gerado para o microprocessador. Fonte: Autor

.

Todos esses elementos criados pela a USAG para o ambiente de verificação do microprocessador, estão interligados pela estrutura do *Agent*. O código gerado para essa estrutura pode ser visualizado na Figura 37. Como pode-se evidenciar são instanciadas as portas de análise (usadas para transmitir os sinais do *Agent*), o *Sequencer*, os *Monitors*, e o construtor do *Agent*.

```
class MICRO_PROCESSADOR_agent extends uvm_agent;
`uvm_component_utils(MICRO_PROCESSADOR_agent)
uvm_analysis_port#(MICRO_PROCESSADOR_transaction) agent_ap_before;
uvm_analysis_port#(MICRO_PROCESSADOR_transaction) agent_ap_after;

MICRO_PROCESSADOR_sequencer sa_seqr;
MICRO_PROCESSADOR_driver sa_drvr;
MICRO_PROCESSADOR_monitor_before sa_mon_before;
MICRO_PROCESSADOR_monitor_after sa_mon_after;
function new(string name, uvm_component parent);
super.new(name, parent);
endfunction: new
function void build_phase(uvm_phase phase);
super.build_phase(phase);
 agent_ap_before = new(.name("agent_ap_before"), .parent(this));
agent_ap_after  = new(.name("agent_ap_after"), .parent(this));

sa_seqr = MICRO_PROCESSADOR_sequencer::type_id::create(.name("sa_seqr"), .parent(this));
sa_drvr = MICRO_PROCESSADOR_driver::type_id::create(.name("sa_drvr"), .parent(this));
sa_mon_before = MICRO_PROCESSADOR_monitor_before::type_id::create(.name("sa_mon_before"), .parent(this));
sa_mon_after  = MICRO_PROCESSADOR_monitor_after::type_id::create(.name("sa_mon_after"), .parent(this));
endfunction: build_phase
function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
sa_drvr.seq_item_port.connect(sa_seqr.seq_item_export);
sa_mon_before.mon_ap_before.connect(agent_ap_before);
sa_mon_after.mon_ap_after.connect(agent_ap_after);
endfunction: connect_phase
endclass: MICRO_PROCESSADOR_agent
```

Figura 37 – *Agent* gerado para o microprocessador. Fonte: Autor

.

No *Scoreboard* é realizado a análise comparativa dos dados obtidos por meio do processo de verificação entre os valores esperados pelo responsável por esse processo e os valores advindos do DUT. Nesse sentido, o arquivo gerado para o *Scoreboard* possui funções como *"uvm_analysis_imp_decl"* para que o mesmo suporte entradas de várias fontes, funções de comparação, construtores e fases de execução. Na Figura 38 é possível observar o código gerado.

```
`uvm_analysis_imp_decl(_before)
`uvm_analysis_imp_decl(_after)
class MICRO_PROCESSADOR_scoreboard extends uvm_scoreboard;
`uvm_component_utils(MICRO_PROCESSADOR_scoreboard)
uvm_analysis_export #(MICRO_PROCESSADOR_transaction) sb_export_before;
uvm_analysis_export #(MICRO_PROCESSADOR_transaction) sb_export_after;
uvm_tlm_analysis_fifo #(MICRO_PROCESSADOR_transaction) before_fifo;
uvm_tlm_analysis_fifo #(MICRO_PROCESSADOR_transaction) after_fifo;
MICRO_PROCESSADOR_transaction transaction_before;
MICRO_PROCESSADOR_transaction transaction_after;
function new(string name, uvm_component parent);
super.new(name, parent);
transaction_before = new("transaction_before");
transaction_after = new("transaction_after");
endfunction: new
function void build_phase(uvm_phase phase);
super.build_phase(phase);
sb_export_before = new("sb_export_before", this);
sb_export_after = new("sb_export_after", this);
before_fifo = new("before_fifo", this);
after_fifo = new("after_fifo", this);
endfunction: build_phase
function void connect_phase(uvm_phase phase);
sb_export_before.connect(before_fifo.analysis_export);
sb_export_after.connect(after_fifo.analysis_export);
endfunction: connect_phase
task run();
 forever begin
 before_fifo.get(transaction_before);
after_fifo.get(transaction_after);
compare();
end
endtask: run
virtual function void compare();
if(transaction_before.out == transaction_after.out) begin
`uvm_info("compare", {"Test: OK!"}, UVM_LOW);
end
 else begin
 `uvm_info("compare", {"Test: Fail!"}, UVM_LOW);
end
endfunction: compare
endclass: MICRO_PROCESSADOR_scoreboard
```

Figura 38 – *Scoreboard* gerado para o microprocessador. Fonte: Autor

.

O *Scoreboard* juntamente com o *Agent* (e os elementos englobados por ele) são encapsulados por outro módulo chamado *Environment*. O código fonte obtido pela USAG para o *Env* pode ser visualizado na Figura 39. Pode-se verificar que além da instanciação do *Env* a USAG gera automaticamente a instanciação dos componentes *Agent* e *Scoreboard* e seu encapsulamento pelo elemento gerado.

```
class MICRO_PROCESSADOR_env extends uvm_env;
`uvm_component_utils(MICRO_PROCESSADOR_env)
MICRO_PROCESSADOR_agent sa_agent;
MICRO_PROCESSADOR_scoreboard sa_sb;
function new(string name, uvm_component parent);
super.new(name, parent);
endfunction: new
function void build_phase(uvm_phase phase);
super.build_phase(phase);
sa_agent = MICRO_PROCESSADOR_agent::type_id::create(.name("sa_agent"), .parent(this));
sa_sb = MICRO_PROCESSADOR_scoreboard::type_id::create(.name("sa_sb"), .parent(this));
endfunction: build_phase
function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
sa_agent.agent_ap_before.connect(sa_sb.sb_export_before);
sa_agent.agent_ap_after.connect(sa_sb.sb_export_after);
endfunction: connect_phase
endclass: MICRO_PROCESSADOR_env
```

Figura 39 – *Env* gerado para o microprocessador. Fonte: Autor

.

Todos os itens citados estão encapsulados pelo bloco *Test*. Esse bloco é gerado pela USAG, que, além de criar o arquivo, realiza a instanciação desses elementos automaticamente. Isto pode ser verificado na Figura 40.

```
class MICRO_PROCESSADOR_test extends uvm_test;
`uvm_component_utils(MICRO_PROCESSADOR_test)
MICRO_PROCESSADOR_env sa_env;
function new(string name, uvm_component parent);
super.new(name, parent);
endfunction: new
function void build_phase(uvm_phase phase);
super.build_phase(phase);
sa_env = MICRO_PROCESSADOR_env::type_id::create(.name("sa_env"), .parent(this));
endfunction: build_phase
task run_phase(uvm_phase phase);
 MICRO_PROCESSADOR_sequence sa_seq;
phase.raise_objection(.obj(this));
sa_seq = MICRO_PROCESSADOR_sequence::type_id::create(.name("sa_seq"), .contxt(get_full_name()));
assert(sa_seq.randomize());
sa_seq.start(sa_env.sa_agent.sa_seqr);
phase.drop_objection(.obj(this));
endtask: run_phase
endclass: MICRO_PROCESSADOR_test
```

Figura 40 – *Test* gerado para o microprocessador. Fonte: Autor

.

Além dos elementos citados, tem-se também o arquivo *Top*. Esse módulo além de englobar todos os outros descritos, realiza a conexão com o DUT. Da mesma forma que os demais módulos, a USAG gera esse arquivo automaticamente. O código gerado para esse elemento pode ser verificado na Figura 41.

```
`include "MICRO_PROCESSADOR_pkg.sv"
`include "MICRO_PROCESSADOR.sv"
`include "MICRO_PROCESSADOR_if.sv"
module MICRO_PROCESSADOR_tb_top;
import uvm_pkg::*;
//Interface declaration
MICRO_PROCESSADOR_if vif();
//Conect the interface on the DUT
MICRO_PROCESSADOR dut(vif.sig_clk_in,
vif.sig_rst,
vif.sig_[15:0]PORT_A,
vif.sig_PORT_B);
initial begin
uvm_resource_db#(virtual MICRO_PROCESSADOR_if)::set(.scope("ifs"), .name("MICRO_PROCESSADOR_if"), .val(vif));
run_test();
end
initial begin
end
endmodule
```

Figura 41 – *Top* gerado para o microprocessador. Fonte: Autor

.

Além das estruturas citadas, a USAG gera um arquivo de *Package*, ou seja, que engloba todas as estruturas citadas podendo ser chamado a qualquer momento pelos demais elementos, como pelo *Top*, por exemplo. O arquivo gerado para o *Package* pode ser vislumbrado na Figura 42.

```
package MICRO_PROCESSADOR_pkg;
import uvm_pkg::*;
`include "MICRO_PROCESSADOR_sequencer.sv"
`include "MICRO_PROCESSADOR_monitor.sv"
`include "MICRO_PROCESSADOR_driver.sv"
`include "MICRO_PROCESSADOR_agent.sv"
`include "MICRO_PROCESSADOR_scoreboard.sv"
`include "MICRO_PROCESSADOR_config.sv"
`include "MICRO_PROCESSADOR_env.sv"
`include "MICRO_PROCESSADOR_test.sv"
endpackage: MICRO_PROCESSADOR_pkg
```

Figura 42 – *Package* gerado para o microprocessador. Fonte: Autor

.

A USAG também gera um arquivo do tipo *Configuration*, e o instancia permitindo que o usuário utilize configurações personalizadas para os elementos UVM. Para o Microprocessador, o código gerado é apresentado na Figura 43.

```
class MICRO_PROCESSADOR_configuration extends uvm_object;
`uvm_object_utils(MICRO_PROCESSADOR_configuration)function new(string name = "");
super.new(name);
endfunction: new
endclass: MICRO_PROCESSADOR_configuration
```

Figura 43 – *Package* gerado para o microprocessador. Fonte: Autor

.

Finalmente, da mesma forma que os demais exemplos de ambientes de verificação obtidos por meio da USAG, é criado também para o microprocessador Pampium um arquivo do tipo *Makefile* opcional. Esse arquivo pode ser utilizado para executar o ambiente de verificação, o que não acontece diretamente na ferramenta, uma vez que esse não é o escopo da USAG. É possível também, por meio da análise dos ambientes gerados, obter como valores quantitativos a geração automática de 12 arquivos fontes e em média 250 linhas de código (sendo esse número variável em função do número de sinais inseridos). Esses valores não levam em consideração a interação do usuário e possíveis linhas de código digitadas por ele, ou arquivos criados manualmente.

5.2 VALIDAÇÃO COM USUÁRIOS

Uma vez construídos e simulados os ambientes de verificação UVM por meio da USAG, requisitou-se para um grupo de usuários que utilizassem a ferramenta, e que por meio dela realizassem a construção de um ambiente de verificação. Além disso, também pediu-se que utilizassem a ferramenta EasierUVM (a qual mais assemelha-se a USAG) a fim de observar por meio de qual das ferramentas a construção do ambiente era mais intuitiva e mais rápida.

O grupo de usuários testadores foi composto de: 1 mestre em engenharia elétrica, 2 mestrandos em engenharia elétrica, 1 graduando em engenharia elétrica e 1 graduando em ciência da computação. Para esses 5 usuários, foi requisitado que construíssem com o auxílio das ferramentas o escopo do ambiente de verificação do somador simples apresentado anteriormente. Não foi requisitado a simulação desse ambiente.

Primeiramente, foi questionado aos usuários sobre o seu domínio de *SystemVerilog* e sobre o domínio da metodologia UVM. Nesse sentido, todos os usuários possuíam conhecimento da HVL, entretanto, não utilizavam a metodologia. Então, apresentou-se para eles o funcionamento da UVM. Uma vez instruídos, os usuários geraram o ambiente de verificação dentro de ambas ferramentas.

Uma vez finalizado, foi questionado o quão intuitivo era a geração do ambiente de verificação por meio da USAG. Para isso, deveria-se dar uma nota numa escala de 0 a 10. Nesse sentido, a usabilidade da USAG ficou compreendida entre 7 e 10 como pode-se verificar na Figura 44.

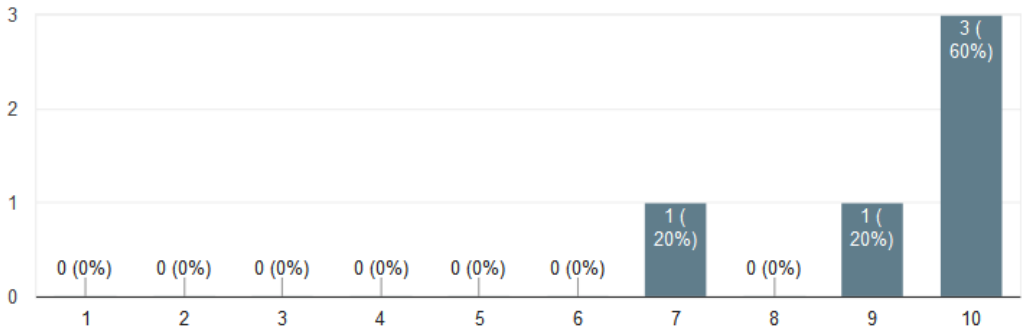

Figura 44 – O quão intuitivo é uso da USAG? Fonte: Autor.

Logo após questionou-se a usabilidade da ferramenta EasierUVM aplicando-se a mesma escala de 0 a 10, onde as notas ficaram compreendidas entre 2 e 5 como pode ser visualizado na Figura 45.

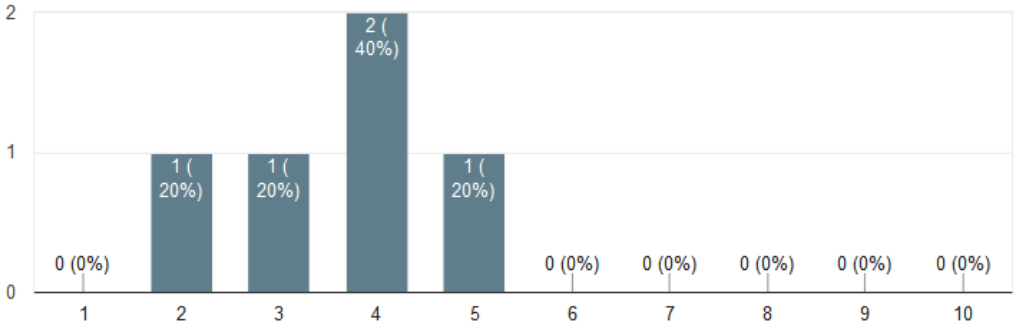

Figura 45 – O quão intuitivo é uso da EasierUVM? Fonte: Autor.

Além da usabilidade foi requisitado aos usuários descrever o quão rápido era possível construir o escopo do ambiente de verificação para um usuário leigo à ferramenta. Quando em relação à USAG, a maior parte dos usuários descreveu o tempo como entre 5 e 10 minutos, como pode ser visualizado na Figura 46.

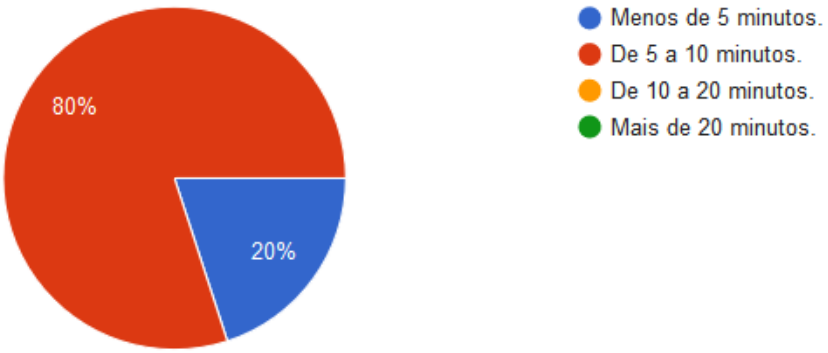

Figura 46 – Tempo necessário para construir o ambiente de verificação na USAG. Fonte: Autor.

Também houve a solicitação para que os usuários descrevessem o tempo necessário para criar o mesmo ambiente de verificação na EasierUVM. Nesse sentido, os usuários descreveram como tempo ideal de 10 a 20 minutos, entretanto, houve uma parcela que julga ser necessário mais de 20 minutos para a geração do ambiente como pode ser visto na Figura 47.

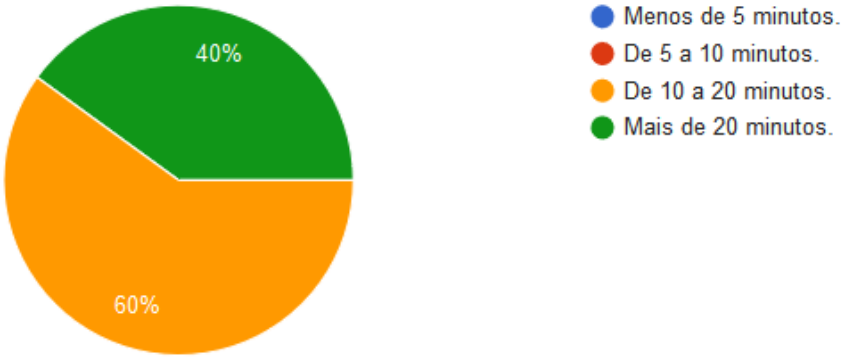

Figura 47 – Tempo necessário para construir o ambiente de verificação na EasierUVM. Fonte: Autor.

Finalmente, os usuários avaliaram qual a ferramenta que acreditavam que viabilizar a construção dos ambientes de verificação UVM de forma mais ágil. Todos os usuários julgaram que a USAG é mais ágil para a construção dos ambientes (quando comparado com a construção manual ou com auxílio da ferramenta EasierUVM), como é apresentado na Figura 48.

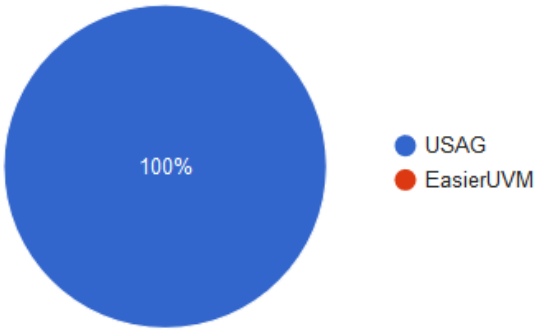

Figura 48 – Construção de ambientes de verificação UVM de forma mais ágil. Fonte: Autor.

5.3 CONSIDERAÇÕES

O presente Capítulo focou em apresentar os resultados obtidos por meio da USAG. Para verificar a viabilidade do uso da ferramenta desenvolvida, criou-se ambientes de verificação por meio de DUTs de entrada e, após, simulou-se esses ambientes. Também gerou-se um ambiente de verificação para um microprocessador Pampium, o qual não foi simulado. Para os DUT testados na USAG, pode-se notar, de forma qualitativa,

que em relação ao proposto a ferramenta é capaz de gerar os ambientes de verificação da metodologia UVM. De forma quantitativa, realizou-se testes com usuários, os quais relataram que em um intervalo de tempo entre 5 e 10 minutos é possível a geração do escopo de um ambiente de verificação e que isso tona o processo mais ágil do que as ferramentas existentes, e também, mais ágil do que realizado de forma totalmente manual.

6 CONCLUSÃO

O trabalho busca facilitar e ganhar agilidade na construção de ambientes de verificação da metodologia UVM. Para tal, foi proposto uma ferramenta semiautomática chamada USAG. A USAG baseia-se na entrada do usuário para gerar esses ambientes.

A USAG foi construída utilizando tecnologias *web* para que não haja a necessidade de instalação por parte do usuário. Além disso, a torna viável para acesso por meio de navegadores de *internet* de diferentes dispositivos. Para atender a dispositivos com tamanho de telas variados, construiu-se a USAG de forma a ser responsiva, ou seja, adapta-se a diferentes resoluções de tela automaticamente. Além disso, a ferramenta permite o armazenamento dos ambientes de verificação gerados no servidor, dessa forma, se o usuário desejar, pode acessar os dados de qualquer lugar, a qualquer momento.

Para a construção dos ambientes de verificação a USAG explora o uso de expressões regulares. Essas expressões, detectam os sinais do código fonte utilizado como entrada (DUT), e permitem ao usuário, por meio da interação com a interface gráfica da ferramenta, obter ambientes completos de verificação da metodologia UVM.

Foram construídos ambientes de verificação testes por meio da USAG, e esses ambientes simulados através do *Synopsys VCS*, a USAG não realiza a verificação dos ambientes dentro da própria ferramenta. Os ambientes simulados funcionaram demonstrando assim a viabilidade da utilização da USAG. Foi proposto a construção de um dos ambientes de verificação por um grupo de usuários. Tais usuários descreveram a USAG como sendo mais simples de utilizar e mais rápida para a construção dos ambientes de verificação do que outras ferramentas como a EasierUVM. Além disso, a necessidade de codificação manual de todo ambiente é diminuída, ganhando-se em desempenho nesse aspecto, corroborando com a proposta do trabalho.

A USAG tem como algumas de suas principais características ser *OpenSource* (o código-fonte da USAG encontra-se disponível em: https://github.com/usagcore/USAG), independente de Sistemas Operacionais, não necessitar o uso de *Templates*, e obter de forma semiautomática os ambientes de verificação seguindo o ambiente padrão da metodologia UVM. Como limitações, pode-se se citar o número de ambientes de verificação testados na USAG, não haver uma ferramenta de simulação embutida (não implementado devido a questões de direitos autorais e de distribuição).

6.1 TRABALHOS FUTUROS

A fim de crescimento e ampliação da USAG, propõe-se a implementação dos seguintes itens como trabalhos futuros:

- **Importação de Módulos**: Adicionar a habilidade de se importar arquivos de um projeto para outro, por exemplo, reutilizar o *"Agent"* de um projeto, dentro de outro.

- **Criar UVCs na tela**: Desenvolver uma interface gráfica onde possa criar e interligar UVCs diretamente na tela, de forma a ampliar a personalização do ambiente de verificação, tornando-o mais amplo por não limitar ao escopo do ambiente padrão da metodologia.

REFERÊNCIAS

ALDEC. *UVM, OVM and VMM*. 2016. <https://www.aldec.com/en/solutions/functional_verification/uvm_ovm_vmm>. Acessado em 18/05/2016. Citado na página 32.

ARAUJO, P. *UVM Guide for Beginners*. 2015. <https://colorlesscube.com/uvm-guide-for-beginners/>. Acessado em 20/11/2015. Citado 3 vezes nas páginas 15, 35 e 48.

BERGERON, J. et al. *Verification methodology manual for SystemVerilog*. [S.l.]: Springer Science & Business Media, 2006. Citado 2 vezes nas páginas 21 e 31.

BIANCHI, C.; GUBLER, O. *UVE - Unified Verification Environment*. 2017. Disponível em: <https://www.hevs.ch/en/rad-instituts/institut-systemes-industriels/projects/uve-unified-verification-environment-6263>. Citado 2 vezes nas páginas 15 e 39.

BROMLEY, J. If systemverilog is so good, why do we need the uvm? sharing responsibilities between libraries and the core language. In: IEEE. *Specification & Design Languages (FDL), 2013 Forum on*. [S.l.], 2013. p. 1–7. Citado 4 vezes nas páginas 22, 25, 34 e 37.

CADENCE. *Universal Verification Mehodology*. 2017. Disponível em: <https://www.cadence.com/content/cadence-www/global/en_US/home/alliances/standards-and-languages/universal-verification-methodology.html>. Citado na página 34.

CAMARA, R. C. P. Ovm_tpi: uma metodologia de verificação funcional para circuitos digitais. Universidade Federal de Pernambuco, 2011. Citado na página 21.

COHEN, B.; VENKATARAMANAN, S.; KUMARI, A. *SystemVerilog Assertions Handbook:–for Formal and Dynamic Verification*. [S.l.]: vhdlcohen publishing, 2005. Citado na página 21.

DOULOS. *VMM Golden Reference Guide*. [S.l.]: Doulos Ltda., 2010. Citado 3 vezes nas páginas 15, 32 e 40.

DRECHSLER, R. et al. Panel: Future soc verification methodology: Uvm evolution or revolution? In: *DATE*. [S.l.: s.n.], 2014. p. 1–5. Citado 3 vezes nas páginas 15, 37 e 38.

ENGROFF, A. M. Asipampium: uma ferramenta de desenvolvimento automatico de processadores de aplicação específica. 2017. Disponível em: <https://drive.google.com/file/d/0B3TA5y53cWfpUUQ3eUV5TE13c3c/view?usp=sharing>. Citado 3 vezes nas páginas 15, 57 e 58.

FARIA, P. D. Automatização de teste em ambiente ci (continuous integration) para a validação de hardware. 2017. Disponível em: <https://repositorio-aberto.up.pt/bitstream/10216/102620/2/180901.1.pdf>. Citado na página 21.

FOSTER, H. D. Trends in functional verification: a 2014 industry study. In: IEEE. *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. [S.l.], 2015. p. 1–6. Citado 7 vezes nas páginas 15, 21, 22, 23, 24, 25 e 34.

GAYATHRI, M. et al. A sv-uvm framework for verification of sgmii ip core with reusable axi to wb bridge uvc. *Advanced Computing and Communication Systems (ICACCS)*, v. 1, p. 1–4, 2016. Citado na página 35.

HEIGHT, H. *A practical guide to adopting the universal verification methodology (UVM)*. [S.l.]: Cadence Design Systems, Inc., 2010. Citado 2 vezes nas páginas 32 e 34.

JORGENSEN, P. C. *Software testing: a craftsman's approach*. [S.l.]: CRC press, 2016. Citado 4 vezes nas páginas 15, 28, 29 e 30.

MADAN, R.; KUMAR, N.; DEB, S. Pragmatic approaches to implement self-checking mechanism in uvm based testbench. *International Conference on Advances in Computer Engineering and Applications (ICACEA)*, v. 1, p. 632–636, 2015. Citado na página 35.

MINTZ, M.; EKENDAHL, R. *Hardware Verification with System Verilog: An Object-Oriented Framework*. [S.l.]: Springer Science & Business Media, 2007. v. 230. Citado na página 21.

PESSOA, I. M. Geração semi-automatica de testbenches para circuitos integrados digitais. *Master's thesis, Universidade Federal de Campina Grande, Av. Aprigio Veloso, Campina Grande*, 2007. Citado 3 vezes nas páginas 21, 31 e 38.

PRESSMAN, R.; MAXIM, B. *Engenharia de Software-8ª Edição*. [S.l.]: McGraw Hill Brasil, 2016. Citado na página 30.

RAGHUVANSHI, S.; SINGH, V. Review on universal verification methodology (uvm) concepts for functional verification. *Int J Electr Electron Data Commun*, v. 2, n. 3, p. 101–107, 2014. Citado na página 34.

SALAH, K. A uvm-based smart functional verification platform: Concepts, pros, cons, and opportunities. In: IEEE. *2014 9th International Design and Test Symposium (IDT)*. [S.l.], 2014. p. 94–99. Citado 3 vezes nas páginas 17, 22 e 34.

SANTOS, O. *Verificação formal de sistemas distribuídos modelados na gramática de grafos baseada em objetos*. 2004. Citado na página 31.

SEVERO, L. C. Uma ferramenta para o dimensionamento automático de circuitos integrados analógicos considerando análise de produtividade. 2012. Disponível em: <http://cursos.unipampa.edu.br/cursos/ppgee/files/2010/03/severo_dissertation.pdf>. Citado na página 21.

SHIMIZU, K. *UVM Tutorial for Candy Lovers – 14. Field Macros*. 2013. Disponível em: <http://cluelogic.com/2012/12/uvm-tutorial-for-candy-lovers-field-macros/>. Citado na página 34.

SILVA, K. R. G. da. *Uma metodologia de Verificacão Funcional para circuitos digitais*. Tese (Doutorado) — Tese de Doutorado, Universidade Federal de Campina Grande, 2007. Citado 6 vezes nas páginas 15, 22, 27, 30, 31 e 39.

SOMMERVILLE, I. Engenharia de software, 8ª edição, tradução: Selma shin shimizu mel-nikoff, reginaldo arakaki, edilson de andrade barbosa. *São Paulo: Pearson Addison-Wesley*, v. 22, p. 103, 2007. Citado 3 vezes nas páginas 15, 28 e 29.

VERIFICATIONGUIDE. *UVM TestBench to verify Memory Model*. 2016. Disponível em: <http://www.verificationguide.com/p/uvm-testbench-example.html>. Citado 4 vezes nas páginas 15, 54, 55 e 56.

WEISSNEGGER, R. et al. A novel simulation-based verification pattern for parallel executions in the cloud. *EuroPLoP'16*, p. 9, 2016. Citado na página 21.

Anexos

ANEXO A – ORIENTAÇÕES DE USO DA USAG

Ao realizar o primeiro acesso ao sistema é necessário que haja o cadastramento do usuário. Isso ocorre para que se possa identificar e tornar único as informações contidas para o utilizador. A Figura 49 demonstra a tela inicial da aplicação, onde no menu *Sign Up*, em destaque, permite que o usuário seja redirecionado para o ambiente de cadastro. Uma vez cadastrado é possível, realizar o acesso posterior ao sistema pelo menu *login* como pode-se verificar também na Figura 49.

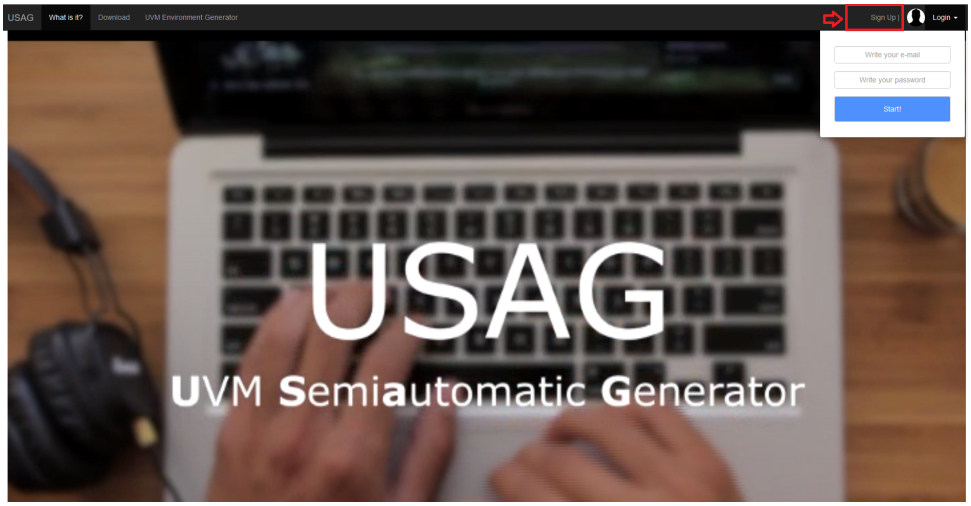

Figura 49 – Tela inicial da aplicação. Fonte: Autor.

Uma vez redirecionado para o ambiente de cadastro é necessário realizar o preenchimento de alguns dados para criar a conta de acesso, como pode ser visualizado na Figura 50. As informações obrigatórias para a realização do cadastro são: nome, e-mail e senha. A opção de adicionar uma foto é opcional. O menu de cadastro é protegido contra ataques do tipo *MySql Injection* e a senha possui proteção criptográfica *MD5*, as quais foram implementadas por meio da tecnologia PHP. Há também a verificação dos dados inseridos, sendo possível realizar somente um cadastro por *e-mail*.

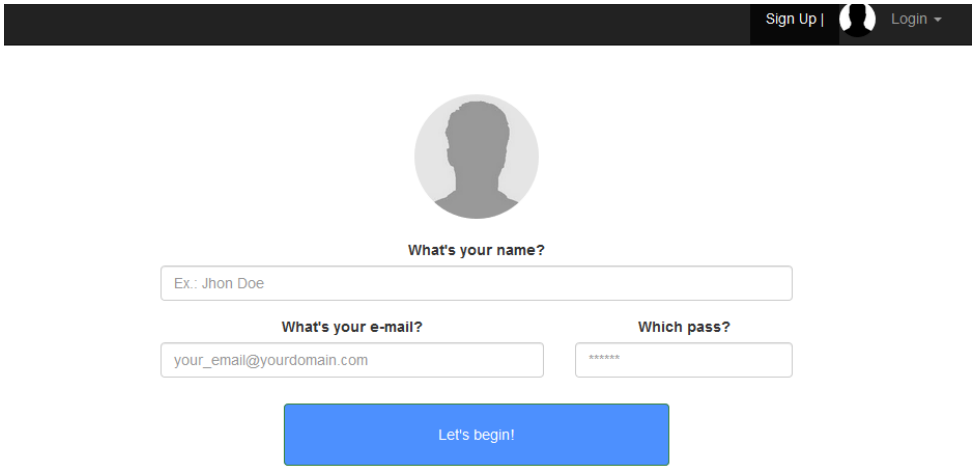

Figura 50 – Cadastro de usuário. Fonte: Autor.

Após registrado, o utilizador recebe um ID único de identificação dentro da USAG e é gerada uma pasta no servidor com esse ID. Há então o redirecionado para a tela inicial de usuários autenticados dentro da USAG. Enfatiza-se que o sistema é baseado em sessões da linguagem PHP, isto é, um usuário só pode acessar as funções se realizar o *login* no sistema. Além disso, essa funcionalidade permite que não haja nenhum acesso indevido aos arquivos armazenados na USAG.

Como pode-se observar a Figura 51 apresenta a tela inicial dentro da USAG onde verifica-se duas possibilidades de interação. Criar um novo projeto, ou abrir um projeto existente.

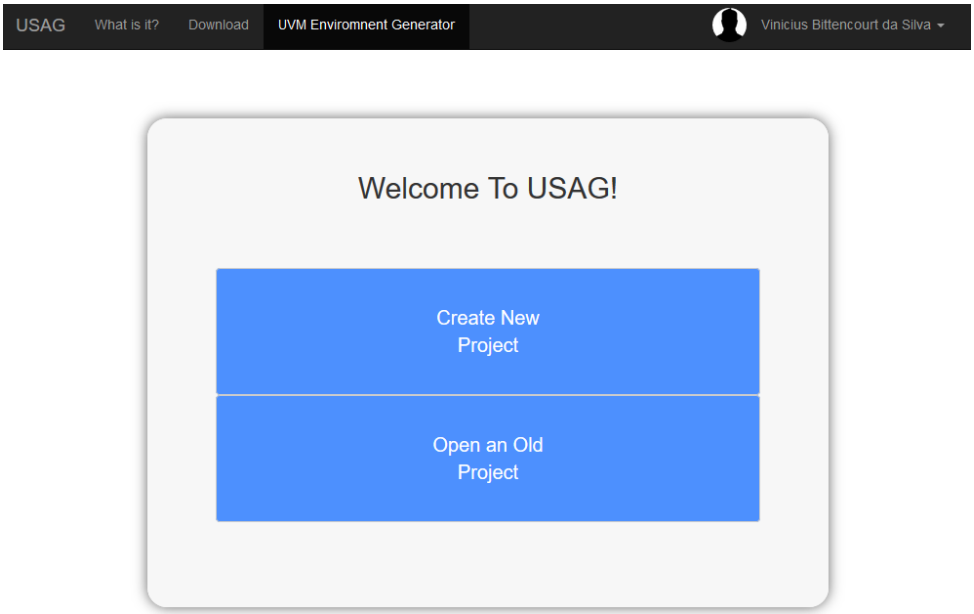

Figura 51 – Tela inicial de usuário autenticado. Fonte: Autor.

Optando pela a criação de um novo projeto inicia-se o processo de geração de um novo ambiente de verificação e realizando a abertura de projetos antigos acessa-se a tela de gerenciamento de projetos. Quando cria-se um novo ambiente de verificação o mesmo recebe um identificador único ID e ao mesmo tempo gera-se para o usuário ativo uma pasta no servidor para armazenar os arquivos produzidos, nomeada com o ID de projeto e ilustrada pela Figura 52.

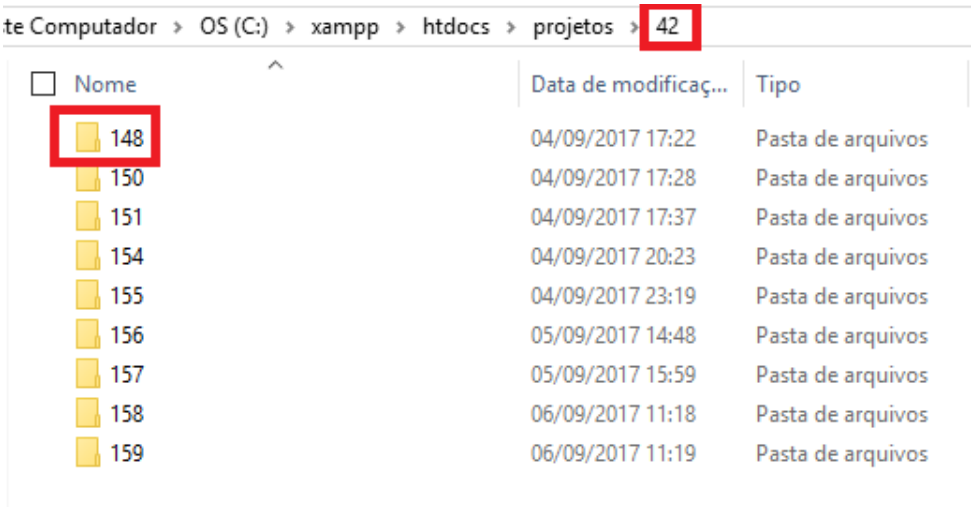

Figura 52 – Pasta de projetos no servidor. Fonte: Autor.

Uma vez gerado o projeto, apresenta-se a tela de inserção de dados e criação do ambiente de verificação pelo usuário que pode ser vista na Figura 53. Pode-se observar uma área principal onde há espaço para a inserção de texto (entrada do DUT), um botão

para a geração do ambiente de verificação e um botão para a criação de novos arquivos. A inserção dos dados pelo usuário devem seguir algumas regras simples:

- ***Não haver cabeçalhos*** – Para que ocorra a detecção correta do código fonte do usuário o código em *SystemVerilog* deve sempre iniciar com a palavra reservada *module*. Caso haja algum cabeçalho (por exemplo, algum comentário) o sistema irá detectar e informar ao usuário sobre o fato.
- ***Não haver comentário nos sinais do DUT*** – Uma vez que os sinais são detectados de forma automática, não deve-se utilizar comentários logo após a declaração de um sinal no código do DUT. É possível que haja uma má detecção dos sinais nesse cenário.

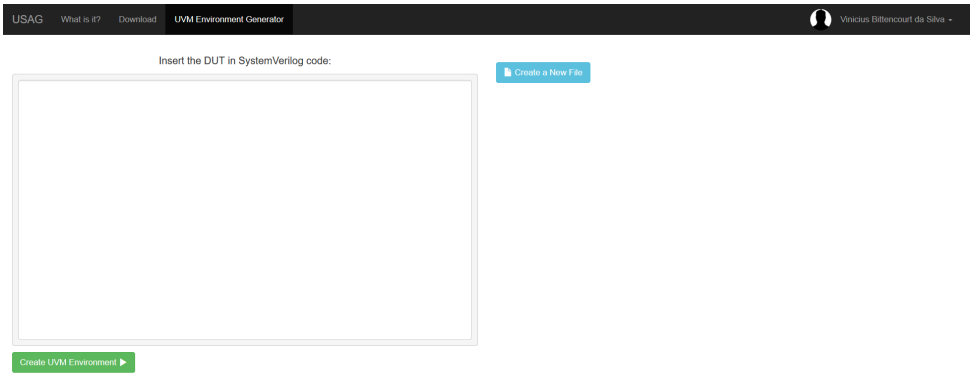

Figura 53 – Tela principal de edição de código. Fonte: Autor.

Após o preenchimento do DUT é possível gerar o ambiente de verificação por meio do botão *"Create UVM Environment"*.