Fabrízio Piccoli Maziero

GERAÇÃO AUTOMÁTICA DE TESTES BASEADA EM ALGORITMOS GENÉTICOS PARA VERIFICAÇÃO FUNCIONAL

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica para a obtenção do Grau de Mestre em Engenharia Elétrica.
Universidade Federal de Santa Catarina. Orientador: Prof. Dr. Djones Vinicius Lettnin

Florianópolis

2016

Ficha de identificação da obra elaborada pelo autor através do
Programa de Geração Automática da Biblioteca Universitária da
UFSC.

Maziero, Fabrízio Piccoli
Geração automática de testes baseada em algoritmos
genéticos para verificação funcional / Fabrízio Piccoli
Maziero ; orientador, Djones Vinicius Lettnin -
Florianópolis, SC, 2016.
112 p.

Dissertação (mestrado) - Universidade Federal de Santa
Catarina, Centro Tecnológico. Programa de Pós-Graduação em
Engenharia Elétrica.

Inclui referências

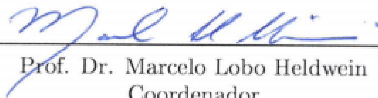
1. Engenharia Elétrica. 2. Sistemas Embarcados. 3.
Verificação Funcional. 4. Algoritmos Genéticos. I. Lettnin,
Djones Vinicius. II. Universidade Federal de Santa
Catarina. Programa de Pós-Graduação em Engenharia Elétrica.
III. Título.

Fabrízio Piccoli Maziero

GERAÇÃO AUTOMÁTICA DE TESTES BASEADA EM ALGORITMOS GENÉTICOS PARA VERIFICAÇÃO FUNCIONAL

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre em Engenharia Elétrica", e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica.

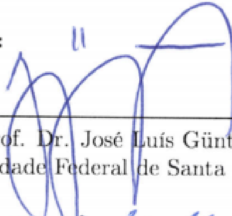
Florianópolis, 6 de Setembro 2016.



Prof. Dr. Marcelo Lobo Heldwein
Coordenador
Universidade Federal de Santa Catarina

Prof. Dr. Djones Vinicius Lettnin
Orientador
Universidade Federal de Santa Catarina

Banca Examinadora:



Prof. Dr. José Luís Güntzel
Universidade Federal de Santa Catarina

Prof. Dra. Karina Rocha Gomes da Silva
Universidade Federal de Goiás

Prof. Dr. Raimes Moraes
Universidade Federal de Santa Catarina

Dedico esse trabalho à todos que me acompanharam nesta caminhada pessoal e profissional.

AGRADECIMENTOS

Em primeiro lugar, gostaria de agradecer aos meus pais, Vilma e Nilson, por todo o apoio que me deram não apenas neste período, mas em toda a vida. Tudo que sou hoje devo à eles. Também ao meu irmão, Maurízio, pelo companheirismo e apoio.

Agradeço à todos os meus amigos, tanto de Passo Fundo, quanto Florianópolis, e de qualquer outro lugar, que estiveram presentes em minha vida neste momento. As conversas, saídas, e convívio certamente ajudaram muito para a conclusão desta etapa.

Agradeço ao meu orientador, Djones Lettnin, pela orientação e auxílio ao longo deste trabalho.

Agradeço aos colegas de laboratório, em especial ao Rogério Paludo e Vinícius Alves, por toda a contribuição e ajuda que me deram ao longo deste tempo, para o trabalho ou não.

Agradeço ao Programa de Pós Graduação em Engenharia Elétrica da UFSC e ao CNPq pela oportunidade de realizar este trabalho.

A ciência é a busca desinteressada pela verdade objetiva a respeito do mundo natural.

(Richard Dawkins)

RESUMO

O constante aumento da complexidade de sistemas embarcados requer um processo de verificação capaz de acompanhar esse crescimento e ser capaz de assegurar o correto funcionamento do sistema projetado, especialmente se tratando de aplicações críticas que lidem com vidas humanas ou com grandes investimentos. Esta responsabilidade por parte das companhias que desenvolvem tais sistemas faz com que a verificação se torne a parte mais importante no projeto de um sistema, consumindo a maior parte dos seus recursos, tanto em questão de tempo quanto financeiramente. A verificação realizada através de simulações requer a participação de um engenheiro de verificação analisando os resultados e com base nestes, modificando parâmetros para gerar novos testes. Neste trabalho é apresentada uma abordagem para uso de Algoritmos Genéticos no processo de verificação, de forma a automatizar a geração de novos vetores de teste. Esta abordagem analisa os resultados com base nas métricas de verificação definidas durante a fase de planejamento do projeto, e com estas informações gera novos testes que contribuam para a validação do sistema, adaptando-se ao funcionamento do sistema e aos resultados de cada nova iteração do processo de verificação.

Palavras-chave: Sistemas Embarcados; Verificação Funcional; Algoritmos Genéticos; Geração Automática de Testes Baseado em Cobertura.

ABSTRACT

The growing increase in embedded systems complexity requires a verification process to be able to follow this trend while capable of assuring the correctness of the designed system, especially on critical applications that deal with human lives, or big financial investments. This responsibility incurred by these system's developers makes verification the most important step in designing an embedded system, considering both development time and money. Simulation-based verification requires an engineer's work by analyzing results and creating new test vectors relevant to the process. In this work an approach for automating test vector generation through Genetic Algorithms is presented. This approach analyzes test results based on predefined verification metrics and, with this information creates new tests that aim on advancing the verification process to reach a better system validation, adapting itself to the design and its results at each step of the process.

Keywords: Embedded Systems; Functional Verification; Genetic Algorithms; Coverage-driven Test Generation.

LISTA DE FIGURAS

Figura 1 Processo de verificação manual (a) e com um Algoritmo de Aprendizado (b). . . . 25
Figura 2 Relação entre número de erros encontrados e custo de correção, em função do tempo de desenvolvimento. Adaptado de Semico Research and Consulting Group (2014). . . . 29
Figura 3 Fluxo simplificado do desenvolvimento de sistemas eletrônicos, adaptado de Iman e Joshi (2004). . . . 30
Figura 4 Arquitetura básica de um *testbench*. . . . 31
Figura 5 Arquitetura de um *testbench* UVM, adaptado de Alves (2015). . . . 34
Figura 6 Relações entre *Data Items*, Sequências e Testes. . . . 35
Figura 7 Estrutura do *Master Agent* de um UVC, com seus módulos internos: *Driver*, BFM e *Monitor*. É representada a passagem das Sequências do *Driver* para o BFM. . . . 36
Figura 8 Fluxo de projeto da metodologia TLM, adaptado de Black et al. (2011). . . . 39
Figura 9 Exemplo de dois indivíduos com genes binários para um AG. . . . 40
Figura 10 Recombinação de dois indivíduos. . . . 42
Figura 11 Fluxos de execução de um AG tradicional (a), e do AG proposto neste trabalho (b). . . . 53
Figura 12 Representações de um indivíduo do AG. . . . 55
Figura 13 Representações de um indivíduo com tamanho de cromossomo variável. . . . 55
Figura 14 Exemplo de indivíduo com os parâmteros de cada sequência e Genes Nulos. . . . 56
Figura 15 Processo de *Single-point Crossover*. . . . 59
Figura 16 Indivíduos a passarem por processo de recombinação. . . 60
Figura 17 Processo de recombinação sem mudança de parâmetros. 60
Figura 18 Agrupamento de todas as sequências B de indivíduos reprodutores. . . . 61
Figura 19 Definição dos Genes de Parâmetros do indivíduo descendente. . . . 61
Figura 20 Agrupamento de todas as sequências B dos indivíduos

reprodutores, com *fitness*. 62
Figura 21 Método da roleta para Genes de Parâmetros. 63
Figura 22 Diagrama UML da classe *Population*, responsável por todo o processo do AG. 65
Figura 23 Fluxo de Verificação com AG. 66
Figura 24 Exemplo de divisão de um sistema em interfaces. 67
Figura 25 Arquitetura do *testbench* UVM tradicional. 73
Figura 26 Arquitetura do *testbench* UVM-Lite proposto. 74
Figura 27 Diagrama UML das classes *Coverpoint* e *Covergroup*. 83
Figura 28 Formato utilizado pela população criada pelo AG. 85
Figura 29 Fluxo de implementação do *testbench* proposto. 86
Figura 30 Diagrama de sequência do funcionamento do *testbench UVM-Lite*. 86
Figura 31 FSM do módulo *ACK/NACK Layer* da UTMC. Adaptado de (BEZERRA, E.; SILVA, E.; ROCHAT, D., Outubro 2009). 90
Figura 32 Resultados para cobertura de FSM do módulo ACK/-NACK Layer utilizando seleção por método de torneio. 93
Figura 33 Resultados para cobertura de FSM do módulo ACK/-NACK Layer utilizando seleção por método de roleta. 93
Figura 34 Distribuição de resultados para cobertura de FSM do módulo ACK/NACK Layer com testes aleatórios com restrições. 94
Figura 35 Comparação de melhor cobertura de FSM encontrada entre os métodos de seleção e testes randômicos. 94
Figura 36 Resultados para cobertura de FSM e *datapath* do módulo *ACK/NACK Layer* através de Seleção por Torneio. 97
Figura 37 Distribuição de resultados para cobertura de FSM e *datapath* do módulo *ACK/NACK Layer* com testes aleatórios. 98
Figura 38 Comparação de resultados entre os métodos de recombinação de Genes de Parâmetros. 98
Figura 39 Resultados para cobertura de FSM e *datapath* do módulo *Idle Layer* através de Seleção por Torneio 101
Figura 40 Distribuição de resultados para cobertura de FSM e *datapath* do módulo *Idle Layer* com testes aleatórios. 102

LISTA DE TABELAS

Tabela 1 Comparação entre trabalhos correlatos. 49
Tabela 2 Tempo de execução para Cobertura de FSM do módulo ACK/NACK Layer. 95
Tabela 3 Tempo de execução para simulação do módulo *Idle Layer*. 103
Tabela 4 Comparação deste trabalho com trabalhos correlatos. . . 106

LISTA DE SIGLAS

AG Algoritmo Genético

ASIC *Application Specific Integrated Circuit*

BFM *Bus Functional Model*

CDV *Coverage-Driven Verification*

DUT *Design Under Test*

EDA *Electronic Design Automation*

FIFO *First In, First Out*

FPGA *Field-Programmable Gate Array*

FSM *Finite State Machine*

HDL *Hardware Description Language*

HW *Hardware*

IEEE *Institute of Electrical and Electronics Engineers*

OVM *Open Verification Methodology*

RTL *Register-Transfer Level*

SAM *System Architectural Model*

TLM *Transaction Level Modeling*

SCV *SystemC Verification Library*

SW *Software*

UML *Unified Modeling Language*

UVC UVM *Verification Component*

UVM *Universal Verification Methodology*

SUMÁRIO

1 INTRODUÇÃO 23
1.1 IDENTIFICAÇÃO DO PROBLEMA 24
1.2 OBJETIVOS 26
1.2.1 Objetivo Geral 26
1.2.2 Objetivos Específicos 26
1.3 ORGANIZAÇÃO DO TEXTO 27
2 CONCEITOS E DEFINIÇÕES 29
2.1 VERIFICAÇÃO FUNCIONAL 29
2.1.1 *Universal Verification Methodology* (UVM) 33
2.2 *SYSTEMC* 36
2.3 ALGORITMOS GENÉTICOS 38
2.4 RESUMO 42
3 TRABALHOS RELACIONADOS 45
3.1 GERAÇÃO AUTOMÁTICA DE TESTES BASEADA EM COBERTURA 45
3.2 METODOLOGIAS DE *TESTBENCH* PARA *SYSTEMC* 47
3.3 RESUMO 48
4 METODOLOGIA PROPOSTA 51
4.1 CRIAÇÃO DE POPULAÇÃO 53
4.2 SELEÇÃO E REPRODUÇÃO DE GENES BÁSICOS ... 58
4.3 SELEÇÃO E REPRODUÇÃO DE GENES DE PARÂMETROS 59
4.3.1 Cruzamento sem mudança de parâmetros 59
4.3.2 Cruzamento aleatório de parâmetros 60
4.3.3 Cruzamento de parâmetros baseado no *fitness* do indivíduo 62
4.3.4 Seleção por Roleta para Genes de Parâmetros 62
4.3.5 Seleção por Torneio para Genes de Parâmetros ... 63
4.4 MUTAÇÃO 63
4.5 IMPLEMENTAÇÃO 64
4.6 METODOLOGIA PARA UTILIZAR O ALGORITMO GENÉTICO 66
4.6.1 Separação do sistema em interfaces 67
4.6.2 Determinar os Genes Básicos 67
4.6.3 Determinar a quantidade m de Genes de Parâmetros para cada Sequência 68

4.6.4 Identificar quantos genes serão necessários em cada parâmetro para randomização de dados . . . 69
4.6.5 Definir as métricas de Cobertura da Verificação e *fitness* . . . 69
4.7 RESUMO . . . 70
5 DESENVOLVIMENTO DE UMA BIBLIOTECA SIMPLIFICADA PARA O PROJETO DE *TEST-BENCHES* BASEADA EM *SYSTEMC* E UVM . . . 71
5.1 *TESTBENCH* . . . 72
5.1.1 *Driver* . . . 74
5.1.2 *Bus Functional Model* (BFM) . . . 75
5.1.3 Monitor de Dados . . . 76
5.1.4 Monitor de Estados . . . 77
5.1.5 *Coverage Collector* . . . 78
5.1.6 Outros Módulos e Arquivos . . . 79
5.1.7 *Testbench* . . . 80
5.1.8 Função *main* . . . 80
5.2 BIBLIOTECA DE COBERTURA . . . 81
5.3 INTEGRAÇÃO COM MODELO E AG . . . 84
5.4 RESUMO . . . 87
6 RESULTADOS . . . 89
6.1 CASO DE ESTUDO: MÓDULO DE COMUNICAÇÃO DE SATÉLITE . . . 89
6.1.1 Primeiro módulo: *ACK/NACK Layer* . . . 89
6.1.2 Cobertura de FSM do primeiro módulo . . . 92
6.1.3 Tempo de Simulação do primeiro módulo . . . 95
6.1.4 Cobertura de FSM e *Datapath* do primeiro módulo 96
6.1.5 Segundo módulo: *Idle Layer* . . . 99
6.1.6 Cobertura de FSM e *Datapath* do segundo módulo 100
6.1.7 Tempo de Simulação do segundo módulo . . . 103
6.2 DISCUSSÃO DOS RESULTADOS . . . 103
6.2.1 Métodos de seleção . . . 104
6.2.2 Métodos de cruzamento de Genes de Parâmetros . . . 104
7 CONCLUSÕES E TRABALHOS FUTUROS . . . 105
7.1 CONTRIBUIÇÕES . . . 105
7.2 TRABALHOS FUTUROS . . . 107
REFERÊNCIAS . . . 109

1 INTRODUÇÃO

Atualmente, o uso de sistemas embarcados para solução de problemas tornou-se parte do cotidiano. A tendência de miniaturização de computadores possibilitou a integração entre diversos sistemas em unidades mais complexas e versáteis, difundindo seu uso nas mais variadas funções e tornando a sociedade dependente de seu funcionamento (MARWEDEL, Peter, 2011). Em muitos casos, sistemas embarcados são utilizados para tarefas que envolvem riscos diretos para a vida humana, como por exemplo, no controle de um avião, ou tarefas que, caso ocorram erros de implementação, significariam o desperdício de um enorme investimento financeiro, como no projeto de satélites. Assim, é indispensável assegurar que esses sistemas funcionem corretamente, de forma a evitar acidentes e reduzir gastos com correções de *bugs*, *recalls* e outras consequências decorrentes desses problemas.

A partir desta necessidade, a verificação funcional tornou-se um processo indispensável para assegurar o correto funcionamento de sistemas embarcados, sendo capaz de verificar *software*, *hardware* e abordagens conjuntas. A verificação funcional parte do princípio que a intenção de um projeto deve ser preservada em sua implementação (PIZIALI, Andrew, 2008), sendo essa intenção definida através de uma especificação, documento que descreve toda a funcionalidade do sistema que se deseja projetar. A partir desta especificação, o engenheiro é capaz de planejar a verificação de forma a assegurar que os requisitos definidos para o funcionamento do sistema estão corretos e a implementação é válida.

A Verificação pode ser realizada principalmente através de duas técnicas: estática (por exemplo, *Model Checking*) e dinâmica (simulações). A primeira técnica baseia-se em provar, matematicamente, que a lógica implementada em um determinado sistema é correta quando comparada com asserções que definem situações a serem validadas, por meio de uma busca exaustiva pelo espaço de estados. A simulação utiliza um modelo computacional do sistema que é excitado por vetores de entrada, percorrendo diferentes caminhos no espaço de estados e resultando no estímulo de diferentes funcionalidades. Ambas técnicas têm seus prós e contras, sendo o *Model Checking* problemático quanto ao uso de memória pela exploração exaustiva, e a simulação pela incapacidade de garantir que todas as possibilidades foram exploradas. Ainda no caso de simulações, uma métrica chamada Cobertura define o quão explorado um sistema foi, baseando-se na especificação do projeto. As-

sim, através desta métrica é decidido se em um determinado momento, o processo de verificação foi concluído, ou se novos testes devem ser realizados para cobrir uma área maior do espaço de estados.

Durante o processo de verificação por simulação, diferentes vetores de entrada são enviados ao modelo, para excitar o maior número de caminhos possíveis dentro do espaço de estados. O engenheiro de verificação deve analisar os resultados (cobertura, erros, etc.) e, baseando-se nestas informações, gerar novos vetores de entrada para cobrir o maior espaço possível do modelo. Este é um processo lento e tendencioso, devido ao fato de que, para poder enviar dados significativos ao modelo, o engenheiro deve ter um certo grau de conhecimento acerca de seu funcionamento, podendo assim sofrer diferenças de interpretação em relação à intenção original do projeto. Diversos métodos tentam amenizar este problema, sendo a geração randômica de vetores de entrada o principal desses, além de ferramentas que analisam automaticamente os resultados de simulações e, baseados nesses, geram novos testes sem interferência humana. Este trabalho propõe uma abordagem utilizando aprendizado de máquina para realizar esta análise com o objetivo de modificar parâmetros da geração randômica de dados.

1.1 IDENTIFICAÇÃO DO PROBLEMA

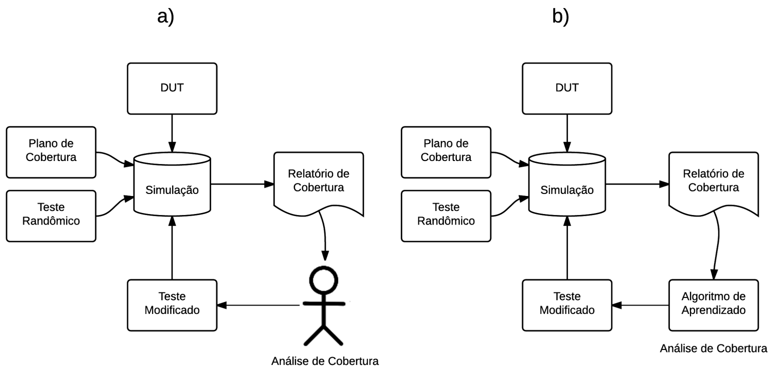
A finalização do processo de verificação de um sistema é um dos pontos mais críticos de todo esse processo. Deve-se ter informações suficientes dos testes, do sistema e dos objetivos da verificação para poder declarar o projeto como livre de erros de desenvolvimento. Caso essas necessidades não sejam respeitadas, ou as informações sejam insuficientes, o projeto pode ser lançado ao mercado, ou colocado em uso com mais problemas do que seria aceitável, causando prejuízos e possíveis danos.

Devido à complexidade dos sistemas embarcados desenvolvidos atualmente, é praticamente impossível garantir que todo o espaço de estados do sistema foi explorado exaustivamente, independente da metodologia de verificação, e que o projeto esteja completamente livre de erros. Focando-se na verificação por simulação, essa exploração consumiria uma quantidade de tempo inviável para o desenvolvimento de um produto e seu lançamento no mercado, e depende de métricas determinadas por pessoas, sendo estas também passíveis de erros e falhas. Para amenizar o impacto de erros humanos no processo de verificação, é interessante substituir certos aspectos desse por processos automati-

zados.

Partindo de uma métrica definida por um valor numérico, é possível utilizar algoritmos da classe de Aprendizado de Máquina para se realizar análises e criar novos testes de forma automática, reduzindo tempo e esforço investidos nestes aspectos. A Figura 1 mostra um comparativo entre a geração manual de testes, com um Algoritmo de Aprendizado, que retira o fator humano desta etapa do processo.

Figura 1 – Processo de verificação manual (a) e com um Algoritmo de Aprendizado (b).



Embora esses algoritmos sejam bastante conhecidos e utilizados, com sua eficácia comprovada nas mais diversas áreas, utilizá-los em um problema fora de suas aplicações habituais pode trazer complicações e variações em sua abordagem. Automatizar a geração de vetores de testes para simulações é uma das áreas que já se beneficia do uso de algoritmos de Aprendizado de Máquina. Porém, poucas vezes a metodologia de como realizar a implementação destas técnicas dentro do ambiente de testes é detalhada, haja vista a existência de diversas metodologias e técnicas próprias para verificação que dificultam uma implementação genérica.

Essas metodologias de verificação são amplamente utilizadas na indústria, com eficácia e benefícios o comprovados. Porém, em muitos casos essas metodologias têm uma curva de implementação bastante íngreme, sendo seus benefícios evidenciados após vários ciclos de verificação, por meio de economia de tempo e desempenho promovidos, principalmente, por reúso de código modular. Além disso, a padronização dessas metodologias para as linguagens de verificação disponíveis é relativamente recente, sendo que nem todas as linguagens já possuem

disponibilizados padrões guias de como realizar a implementação. Uma implementação rápida, mesmo que faltando funcionalidades, contribui com o início da verificação, especialmente quando se deseja descobrir erros em um sistema que se encontra em fase inicial de desenvolvimento, ainda como um modelo de alto nível.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O principal objetivo deste trabalho é propor uma metodologia para uso de Algoritmos Genéticos (AG) na geração automática de testes para simulações de *hardware* baseando-se na cobertura coletada nesses mesmos testes. Esta metodologia deve detalhar os passos para integrar o algoritmo a um ambiente de testes, e possibilitar implementação em diversos métodos para verificação de *hardware* e/ou *software* por simulação, independente da linguagem utilizada.

Para realização de testes que comprovem a eficácia desta metodologia, foi também desenvolvido um ambiente de testes (*testbench*) para simulação de sistemas de alto nível, independente de implementação em *hardware* ou *software*. Este será baseado em uma das principais metodologias de verificação atuais porém, simplificando várias funcionalidades de forma a facilitar a implementação, sem perda das características essenciais de um ambiente de testes. O *testbench* criado como resultado desta pesquisa é capaz de, após ser inicializado com um determinado número de testes randômicos (população inicial do Algoritmo Genético), evoluir a composição de teste para atingir uma cobertura total do sistema maior que seria atingida com realização de testes completamente randômicos, sendo este um parâmetro para comparação direta.

1.2.2 Objetivos Específicos

- Criação de um *testbench* que possa, além de ser integrado com o AG de forma a não necessitar de auxílio humano para troca de informações, ser facilmente adaptado para vários modelos a serem testados;

- Utilizar um AG para otimizar a geração de casos de testes para

as simulações, baseando-se nos resultados de cobertura, com a capacidade de ampliar a cobertura total da verificação após cada geração;

- Integrar o *testbench* com o AG e desenvolver uma metodologia para definir os parâmetros do AG de forma geral para qualquer processo de verificação.

1.3 ORGANIZAÇÃO DO TEXTO

Esta dissertação está organizada da seguinte forma:

- **Capítulo 2:** Realiza uma revisão dos principais conceitos utilizados neste trabalho;
- **Capítulo 3:** Apresenta trabalhos relacionados aos principais tópicos deste trabalho;
- **Capítulo 4:** Descreve o uso do AG e seus parâmetros para implementação no processo de verificação;
- **Capítulo 5:** Descreve a metodologia utilizada para criação do *testbench* a ser integrado e sua implementação;
- **Capítulo 6:** Apresenta estudos de caso e discute os resultados obtidos;
- **Capítulo 7:** Apresenta as conclusões desta dissertação.

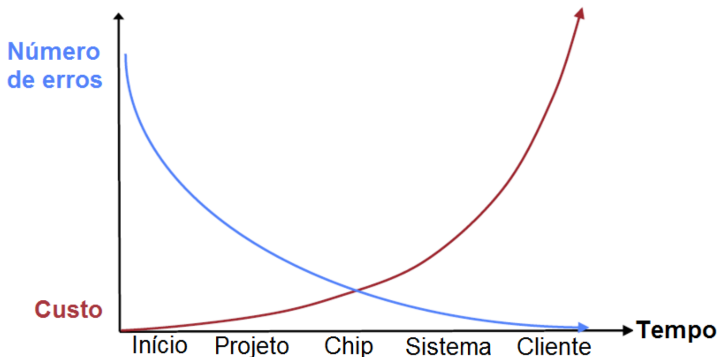
2 CONCEITOS E DEFINIÇÕES

2.1 VERIFICAÇÃO FUNCIONAL

É importante ressaltar o significado adotado para erro, falha e defeito antes de iniciar a discussão sobre verificação. Aparentemente, não há significado unânime para esses três conceitos. Assim, serão adotadas as definições utilizadas por Delamaro, Maldonado e Jino (2007). Defeito é a definição incorreta de dados; erro é a manifestação de um defeito, ou seja, um estado inconsistente alcançado durante execução; falha diz respeito ao resultado esperado para determinada tarefa. Um erro pode se tornar uma falha se o resultado produzido for diferente do esperado.

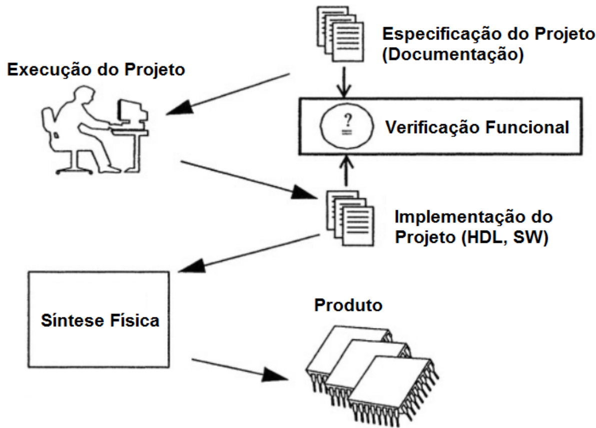
Verificação Funcional é o processo realizado no desenvolvimento de sistemas eletrônicos para assegurar seu correto funcionamento. A importância deste processo deve-se à na grande complexidade dos atuais sistemas eletrônicos e nos riscos, tanto financeiros quanto de vida em se utilizar tais sistemas se esses possuírem falhas. Além disso, quanto mais cedo no desenvolvimento estes problemas forem encontrados, menor será o custo para corrigi-los, tanto financeiro quanto em questão de tempo. A Figura 2 mostra esta relação entre o estágio que um erro ou falha é encontrado, e seu custo no projeto.

Figura 2 – Relação entre número de erros encontrados e custo de correção, em função do tempo de desenvolvimento. Adaptado de Semico Research and Consulting Group (2014).



Para entender a necessidade deste processo, primeiro deve-se compreender como é realizado o projeto de sistemas eletrônicos. A Figura 3 mostra, de forma simplificada, o fluxo de projeto típico de desenvolvimento desses sistemas.

Figura 3 – Fluxo simplificado do desenvolvimento de sistemas eletrônicos, adaptado de Iman e Joshi (2004).



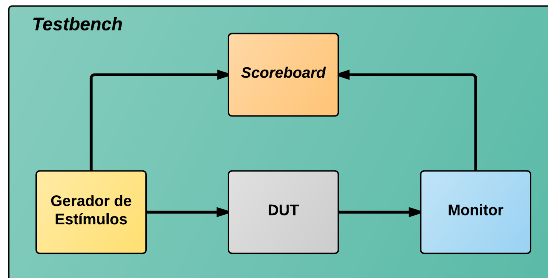
Inicia-se com o planejamento do sistema, formalizado com a criação da documentação do projeto. Esta, por meio de especificações que deverão ser interpretadas por um engenheiro, é transformada em uma implementação do sistema, normalmente realizada em uma linguagem de descrição, como uma HDL (*Hardware Description Language*) ou um *software*. A partir desta implementação é possível realizar a síntese física do *hardware*, integração com *software* (caso possua), para finalmente ser produzido fisicamente e lançado no mercado. O ponto crítico deste fluxo encontra-se entre a documentação e implementação, onde ocorrem grande parte dos erros. Estes se devem a diversos fatores, entre os quais se destacam a má interpretação do engenheiro na execução, erros na documentação ou erros na implementação (IMAN; JOSHI, 2004).

Entre as diversas metodologias de verificação existentes, pode-se classificá-las, principalmente, entre métodos estáticos (como *Model Checking*), que utilizam provas matemáticas para garantir que a implementação corresponde à intenção de projeto, e métodos dinâmicos (simulações), que por meio de um modelo do sistema e vetores de teste,

exploram possibilidades e caminhos de execução dentro deste modelo. Este trabalho foca em métodos dinâmicos, também chamados simulativos, não abordando o uso de métodos estáticos.

Para realizar simulações, é necessário um modelo do sistema que está sendo desenvolvido,denominado de DUT (*Design Under Test*), normalmente desenvolvido em uma linguagem de descrição de *hardware* (HDL), ou linguagem de descrição de arquitetura. Além disso, é necessário um ambiente de testes (*testbench*) que se comunica com o DUT. *Testbenches* normalmente são escritos na mesma linguagem do DUT, caso esta tenha suporte para verificação (*SystemVerilog*, por exemplo (IEEE..., 2009)), ou em linguagens específicas de verificação (*e Verification Language* (IEEE..., 2011), *OpenVera* (SYNOPSYS, INC.,)). Entre as principais funções do *testbench* destacam-se a geração de vetores de entrada (normalmente randomizados, com ou sem restrições), monitoramento de dados de saída e sinais internos e implementação de *checkers*, utilizados para determinar a presença de erros através da comparação dos sinais monitorados com asserções definidas pela especificação do projeto. A Figura 4 mostra a arquitetura básica de um *testbench*, com um módulo Gerador de Estímulos para gerar os vetores de entrada, Monitor para coletar dados de saída do DUT, e um *Scoreboard* onde são implementados *checkers* baseados nos sinais de entrada e saída.

Figura 4 – Arquitetura básica de um *testbench*.



Para a realização do processo de verificação, é ainda necessário definir uma métrica que estabeleça o quanto do espaço de estados do sistema foi explorado por vetores de teste. Esse valor é chamado de cobertura, e se baseia em métricas, parâmetros do ambiente de teste úteis para acompanhar o progresso da verificação em uma dada dimensão (PIZIALI, Andrew, 2008). Após a realização de um determinado número de

testes, cobertura e erros encontrados pelos *checkers* são analisados e, com base nestes, restrições e parâmetros de geração são adaptados para a criação de novos testes. Algumas categorias de métricas utilizadas são:

- Cobertura de Código: linhas de código do DUT percorridas durante simulação;
- Cobertura de Funções: funções do DUT executadas;
- Cobertura de FSM: estados de uma FSM percorridos;
- Cobertura de Dados: valores de dados de entrada ou saída atingidos.

Durante um teste, a cobertura é coletada no momento em que o evento relacionado à métrica ocorre. Porém, o valor total da cobertura é calculado apenas ao final de cada simulação. Toda vez que um estado ou valor de sinal é atingido, uma ocorrência é acrescentada na *bin* respectiva a este estado ou sinal. *Bin* é uma representação figurativa de um recipiente que armazena as ocorrências de cobertura. Os valores que podem ser armazenados dentro de uma *bin* são personalizáveis, podendo ser utilizada uma *bin* para cada valor possível de uma variável, ou intervalos contendo vários valores. Além disso, pode-se atribuir diferentes pesos para cada *bin*, tornando alguns valores ou intervalos mais importantes na métrica de cobertura.

Existem três tipos principais de cobertura, sendo estas a Cobertura de Item Único (*Single-item Coverage*), Cobertura Cruzada (*Cross-coverage*) e Cobertura de Transição (*Transistion Coverage*). A primeira dessas é a cobertura padrão, coletada na ocorrência de um determinado valor de um item. Cobertura Cruzada representa a cobertura simultânea de cada valor de dois itens diferentes (por exemplo, dois sinais de um bit necessitaram de quatro *bins* para cobertura de todas as possibilidades de cruzamento), e Cobertura de Transição representa um único item cujo valor deve mudar para outro valor específico após a ocorrência de um determinado evento temporal (normalmente medido em ciclos de *clock*, no caso de *hardware*).

Para obter eficiência na verificação, metodologias foram criadas com o objetivo de orientar desde o processo de especificação e documentação, até a criação do *testbench*. Duas metodologias bastante difundidas na indústria atualmente são a *Universal Verification Methodology* (UVM), que será abordada na Seção 2.1.1, e a *Coverage-Driven Verification* (CDV). *Coverage-Driven Verification* é uma metodologia que

enfatiza um planejamento específico da cobertura precedendo todo o processo de verificação, ao invés da realização de simulações direcionadas cujos cenários de testes são definidos previamente. Nestes, não há uma ampla exploração do sistema, já que testes que atingirem os cenários interessantes e completarem o objetivo (execução de tais cenários) poderão não explorar casos críticos não considerados inicialmente. Por outro lado, na CDV todas as possibilidades de se atingir os objetivos escolhidos (métricas de coberturas) poderão ser exploradas enquanto novos testes são realizados (PIZIALI, Andrew, 2008).

2.1.1 *Universal Verification Methodology* (UVM)

A *Universal Verification Methodology* (UVM) (ACCELLERA SYSTEMS INITIATIVE, 2015) é uma metodologia de verificação que estabelece orientações a respeito de como criar *testbenches* e testes assegurando modularidade, reusabilidade e eficiência. A implementação original da UVM foi realizada na linguagem de descrição de *hardware* e verificação *SystemVerilog* (IEEE..., 2009), porém, é também suportada pela *e Verification Language* (IEEE..., 2011), linguagem da qual se originaram vários conceitos integrados na metodologia, principalmente em relação à reutilização de código e testes. É um padrão *open-source*, que objetiva unificar as técnicas de verificação utilizadas comumente na indústria, facilitando a integração entre diversos simuladores e ferramentas comerciais (ROSENBERG; MEADE, 2013).

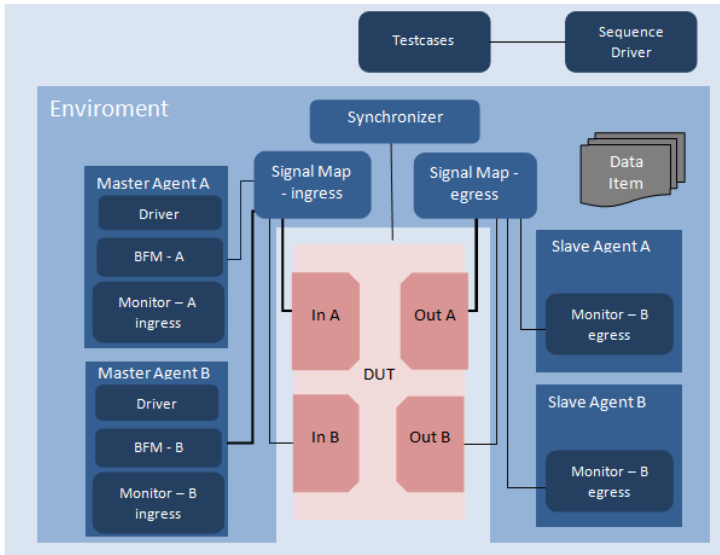
As principais características da UVM incluem:

- Criação de *testbenches* através de programação orientada a objetos (também a aspectos, dependendo da linguagem utilizada);
- Modularidade do *testbench*, permitindo que o ambiente de verificação seja dividido em módulos fáceis de gerenciar, facilitando a reutilização de código;
- Geração de estímulos configuráveis, incluindo randomização e restrições;
- Modelos de cobertura e *checks* que permitem analisar o resultado dos testes e encontrar erros;
- Comunicação entre módulos e DUT em nível de transações.

A Figura 5 mostra a arquitetura geral de um *testbench* criado seguindo a metodologia UVM. *A* e *B* representam diferentes interfaces

do DUT, sendo que, para cada interface, um mesmo grupo de módulos é necessário para sua verificação. Este conjunto de módulos é denominado UVC (UVM *Verification Component*). UVCs são reutilizáveis entre DUTs que possuam interfaces em comum, através do conceito *plug-and-play*, em que o UVC deve apenas ser inserido no *testbench* e seus sinais conectados para ser utilizado.

Figura 5 – Arquitetura de um *testbench* UVM, adaptado de Alves (2015).

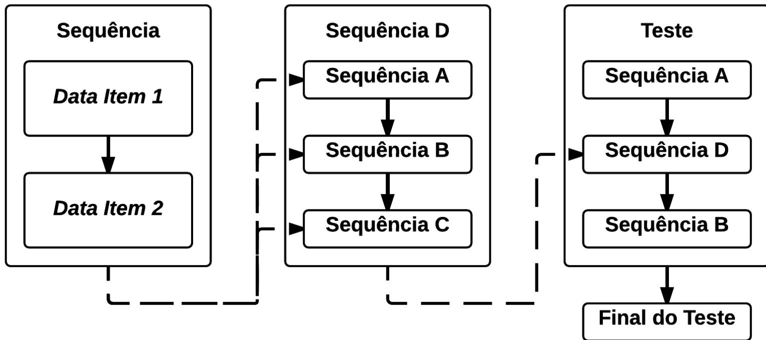


Sequências são o bloco básico para construção de testes dentro de um ambiente UVM. Essas são compostas por *Data Items*, estruturas que representam transações de sinais a serem transmitidas entre diferentes interfaces, como por exemplo, um pacote de dados. Esses *Data Items* são agrupados de forma sequencial com diferentes parâmetros e valores, compondo assim uma Sequência. Sequências ainda podem ser organizadas hierarquicamente, sendo controladas pelo *Sequence Driver* mostrado na Figura 5 para criar diferentes cenários de teste.

A Figura 6 mostra as relações entre *Data Items*, Sequências e Testes: *Data Items* são ordenados em conjuntos para formar Sequências. Essas Sequências também podem ser compostas por conjuntos de Sequências, ordenadas de forma hierárquica. Por fim, testes são conjuntos de Sequências executadas em uma determinada ordem (ou também

paralelamente, no caso de interfaces paralelas) até que a simulação seja finalizada.

Figura 6 – Relações entre *Data Items*, Sequências e Testes.



Cada interface do ambiente de verificação possui um *Agent*, o módulo que encapsula os componentes reutilizáveis do UVC. Este *Agent* pode ser *master*, caso receba Sequências para passar ao DUT, ou *slave*, caso esteja ligado à uma interface que apenas receba dados de saída. Dentro deste *Agent* são instanciados os módulos que realizam as transações de sinais entre o *testbench* e o DUT: *Driver*, *Bus Functional Model* (BFM) e *Monitor* (Figura 7).

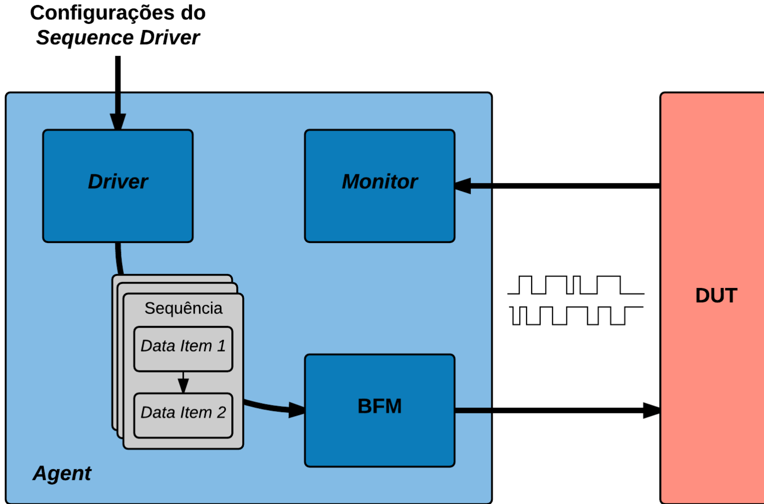
O *Driver* recebe parâmetros de Sequências do *Sequence Driver*, e as cria fisicamente. Estas são passadas ao BFM, que transforma as transações em alto nível para sinais lógicos, implementando o protocolo da interface e realizando a comunicação com o DUT. O *Monitor* observa estas transações, coletando dados relevantes ao funcionamento do modelo e cobertura. *Slave Agents* apenas monitoram a comunicação da interface, não sendo necessários *Drivers* e BFMs.

Os módulos restantes, *Synchronizer* e *Signal Map* (Figura 5) são responsáveis, respectivamente, pela geração e sincronização de sinais *clock* para o DUT, e conexões dos sinais e portas do *testbench*, tanto entre módulos internos quanto sinais externos com o DUT.

Um módulo adicional chamado *Scoreboard* pode também ser incluído, cuja função é reunir as informações dos diferentes *Monitors* do ambiente de testes, implementar *checkers* e comparar essas informações com um modelo de referência ou uma tabela de valores esperados, para comprovar o funcionamento do DUT.

Com o uso de blocos modulares, a verificação de sistemas com-

Figura 7 – Estrutura do *Master Agent* de um UVC, com seus módulos internos: *Driver*, BFM e *Monitor*. É representada a passagem das Sequências do *Driver* para o BFM.



plexos compostos por diversos modelos é realizada de maneira mais rápida e eficiente, já que um UVC criado para um modelo simples pode ser completamente reutilizado quando este modelo é integrado a um sistema maior. Configurações do *Agent* permitem que sua funcionalidade seja modificada de *master* para *slave* facilmente, deixando de gerar dados e apenas monitorando as informações que são transmitidas entre diferentes partes do DUT.

Interfaces de comunicação comumente utilizadas em projetos de *hardware*, como Ethernet, SPI ou I^2C tem seus ambientes de verificação padronizados, podendo ser desenvolvidos apenas uma vez e comercializados como soluções genéricas, utilizados em uma imensa gama de sistemas, reduzindo o tempo de verificação e, consequentemente, o tempo para um determinado sistema ser colocado em uso.

2.2 *SYSTEMC*

SystemC é uma biblioteca da linguagem *C++* com foco em simulação e verificação de sistemas em alto nível, independente da im-

plementação (*hardware* e/ou *software*). A principal vantagem de uma abordagem independente é o aumento de produtividade resultante do desenvolvimento em alto nível, o qual permite que os engenheiros possam unificar o sistema mais cedo no ciclo de projeto (BLACK et al., 2011). Com isso, os desenvolvimentos de *hardware* e *software* podem andar de forma conjunta, sem uma dependência direta entre os dois até o momento de integração. Isso também permite acelerar o processo de verificação, não apenas pelo fato de *hardware* e *software* entrarem em desenvolvimento mais rapidamente, mas também pelo ganho de desempenho na simulação de sistemas em alto nível, por serem mais simples e abstratos.

A biblioteca foi mantida pela *Open SystemC Initiative*, uma organização criada para manter a neutralidade no desenvolvimento em relação às empresas desenvolvedoras de EDA, até ser incorporada pela *Accellera Systems Initiative*, que atualmente mantém e promove diversos padrões IEEE relacionados à indústria de EDA.

Entre as principais características que *SystemC* adiciona à *C++*, pode-se destacar:

- *Kernel* de simulação baseado em eventos;
- Noção de tempo, possibilitando simulação concorrente entre diferentes módulos, e diferentes abordagens em relação a como o tempo é considerado;
- *Datatypes* específicos para Hardware;
- *Modules*, que permitem separar o sistema de forma modular e organizá-lo hierarquicamente;
- *Ports* e *Channels*, que conectam sinais e modelam transações em alto nível;
- Possibilidade de modelagem em RTL (*Register-transfer level*), TLM (*Transaction-level modeling*) e SAM (*System architectural model*).

Entre os *datatypes* incluídos na biblioteca *SystemC* pode-se destacar *sc_logic*, que possui uma lógica de quatro estados, incluindo X (indefinido) e Z (alta impedância), além de 0 e 1. Além disso, os *datatypes* de tipo inteiro podem ter seu tamanho em bits ajustado de acordo com as necessidades do sistema.

Em questão de abstração, enquanto SAM tradicionalmente serve para uma visão geral do sistema, sem preocupações com a temporização

(*Un-timed modeling*), e RTL possui uma implementação muito específica, precisa em relação à temporização (*Cycle-accurate*), a metodologia TLM serve como um meio termo para as duas abordagens. Nessa, apenas as informações relevantes ao engenheiro são modeladas, com uma temporização intermediária (*Approximate-Timed*), na qual os detalhes específicos dos protocolos de comunicação não são importantes. Porém, a ordem em que eventos transmitem estas informações, são.

TLM também é considerada uma metodologia de projeto que, embora utilize este nome, também incorpora RTL e SAM. A Figura 8 mostra este fluxo de projeto, partindo da especificação do sistema, até o modelo RTL que pode ser sintetizado para um ASIC ou FPGA. Inicia-se pela criação da especificação e requerimentos para o sistema e, a partir dessa, cria-se a arquitetura geral do sistema, o modelo SAM. No desenvolvimento do modelo SAM, mudanças podem ser realizadas na especificação, com o objetivo de otimizar pontos que não foram considerados antes da implementação.

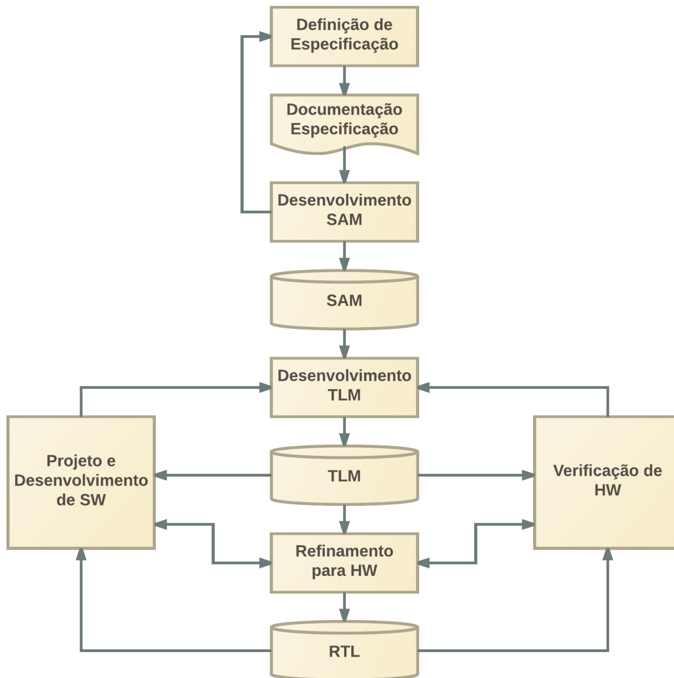
Após este estágio, inicia-se o refinamento para o modelo TLM. Aqui já é possível começar o processo de verificação do sistema, e a separação entre HW e SW, iniciando-se o desenvolvimento do segundo. O desenvolvimento em TLM é um processo contínuo, que avança junto à verificação de desenvolvimento de SW, até se tornar possível a separação entre as implementações (SW e HW). A partir deste ponto é realizado o refinamento do HW, chegando ao modelo RTL deste.

2.3 ALGORITMOS GENÉTICOS

Algoritmos Genéticos são uma classe de algoritmos de aprendizado de máquina utilizado na solução de problemas de otimização, inspirados nos conceitos de genética e seleção natural propostos originalmente por Darwin (1859). Esses algoritmos propostos por John H. Holland (1977) possibilitam explorar uma ampla gama de potenciais soluções, utilizando-se de randomização, cruzamento entre estas, e mutação de parâmetros para melhor explorar o espaço de estados de um problema (HOLLAND, 1992b). Tratando-se de um algoritmo de otimização, possui um vasto campo de aplicação em diferentes áreas, incluindo a verificação funcional de sistemas embarcados.

Estes algoritmos surgiram da necessidade de problemas computacionais que precisam de soluções capazes de se adaptar ao seu meio, observando mudanças na caracterização do problema, e então modificando sua reação (MITCHELL, 1998). Outras abordagens para solução

Figura 8 – Fluxo de projeto da metodologia TLM, adaptado de Black et al. (2011).



de problemas desta natureza incluem Redes Neurais Artificiais, e métodos de aprendizado supervisionado, como Regressões.

Por se inspirar na biologia, é possível traçar um paralelo entre a evolução de organismos vivos, e o funcionamento dos Algoritmos Genéticos. Todas as células de organismos vivos possuem cromossomos, que servem como um guia de como estas cada célula deve se desenvolver dentro deste organismo. Cromossomos são compostos por genes, que codificam características específicas destas células e, consequentemente, do organismo (MITCHELL, 1998). Para um AG, organismos ou indivíduos representam possíveis soluções para o problema apresentado. Normalmente, cada indivíduo de um AG possui apenas um cromossomo, sendo estes dois termos sinônimos neste contexto, embora para problemas complexos possam ser necessários múltiplos cromossomos por solução. Cada um destes cromossomos é codificado como um conjunto de genes, no qual cada gene representa um parâmetro ou variável da

solução.

Organismos reproduzem-se e passam suas características para seus descendentes, através de uma recombinação de seus cromossomos, fazendo com que seu descendente possua genes (características) de ambos indivíduos que o geraram. Mutações causam mudanças neste genes, normalmente provenientes de erros no processo de replicação e recombinação dos cromossomos consequentemente, aumentando a diversidade genética de uma população e, possivelmente, favorecendo a evolução (GRIFFITHS, 2012). No AG, reprodução entre indivíduos permite a criação de novas soluções que possuam características de ambos reprodutores. Mutações inserem uma possibilidade de mudanças aleatórias na solução para explorar uma área maior do espaço de estados da solução.

Por fim, organismos mais aptos ao seu ambiente possuem maior chance de passar seus genes adiante, e contribuir na evolução de uma espécie. Da mesma forma, soluções mais próximas do objetivo desejado passam suas características para gerações seguintes, encaminhando o problema a uma resposta.

Uma população inicial de indivíduos é gerada aleatoriamente, e avaliada em relação a uma função que determina a aptidão desta solução, chamada função *fitness*. Esta função normalmente é a função matemática para qual se deseja resolver o problema embora, caso esse não seja puramente matemático, uma função deve ser determinada para obter o *fitness*. Cada indivíduo dessa população é normalmente representado como uma palavra binária (*string*), sendo cada valor seus genes (Figura 9). Neste tipo de representação, cada valor 0 e 1, ou cada grupo de valores corresponde a um diferente parâmetro da solução, sendo que sua posição no cromossomo determina qual é este parâmetro.

Figura 9 – Exemplo de dois indivíduos com genes binários para um AG.

1	0	1	1	1	0	0	1	0	1

1	1	1	0	0	0	1	0	0	0

A partir de uma população inicial composta de diferentes indivíduos com seus respectivos valores de *fitness*, três operadores são utilizados para obtenção da nova população: seleção, reprodução e mutação.

A seleção consiste em escolher indivíduos que darão origem à próxima população. Idealmente, se espera que os indivíduos mais aptos sejam selecionados. Porém, é interessante que alguns menos aptos

também se reproduzam, de forma a manter a diversidade da população caso contrário, o algoritmo pode convergir rapidamente em soluções não-ótimas, denominadas *mínimos locais*.

Existem diversos métodos de seleção, sendo os mais comuns a Seleção por Torneio (*Tournament Selection*), e Seleção por Método de Roleta (*Roulette Wheel*).

Na Seleção por Torneio, um grupo de indivíduos (geralmente dois) é selecionado. Um número aleatório r entre 0 e 1 é gerado, e comparado a um parâmetro k definido previamente, e geralmente maior que $0,5$. Caso $r < k$, o indivíduo mais apto é selecionado e caso contrário, o menos apto. O grupo de indivíduos é retornado à população, e pode ser escolhido novamente.

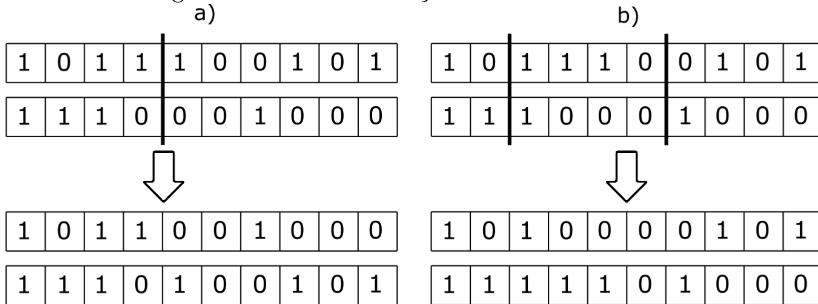
No método da Roleta, cada indivíduo é associado a uma fatia circular de uma roleta, sendo o tamanho desta fatia proporcional ao *fitness* do indivíduo. Indivíduos mais aptos possuem uma fatia maior e, consequentemente, mais chances de serem selecionados. A roleta gira um N número de vezes, sendo N dependente da quantidade de indivíduos da população, e quantos serão utilizados no processo de reprodução. Este método tem como desvantagem uma maior quantidade de processamento, já que é necessária uma classificação geral de todos os indivíduos relativo ao *fitness*, quando comparado à Seleção por Torneio, que compara apenas pequenos grupos(MITCHELL, 1998).

Após selecionados os indivíduos a se reproduzirem, é realizado o processo de recombinação, normalmente entre dois indivíduos por vez. Os métodos utilizados comumente são a Recombinação de Um Ponto (*One-point Crossover*) ou a Recombinação de Dois Pontos (*Two-point Crossover*), nos quais respectivamente um e dois pontos de recombinação *cp* (*Crossover Point*) são escolhidos no cromossomo, e a partir destes pontos as *strings* são permutadas. A Figura 10 mostra dois exemplos de recombinação, sendo em a) o *One-point Crossover* com $cp = 4$ e em b) o *Two-point Crossover*, com $cp_1 = 2$ e $cp_2 = 6$. É Também possível realizar a recombinação envolvendo mais de dois indivíduos, e com mais que dois pontos de recombinação.

Uma outra forma de recombinação menos utilizada é a Recombinação Uniforme (*Uniform Crossover*), na qual cada bit de cada cromossomo do par de *strings* que irá passar por recombinação tem uma chance de ser substituído por um bit do outro cromossomo com o qual forma par. Desta forma, tem-se efetivamente múltiplos *crossover points* no processo, potencialmente variando entre cada par de indivíduos selecionado (SYSWERDA, 1989).

A mutação é o último operador a ser aplicado, no qual cada bit do

Figura 10 – Recombinação de dois indivíduos.



cromossomo possui uma chance bastante pequena (normalmente menor que 1%) de ter seu valor invertido. Por exemplo, um dado cromossomo 1001 pode sofrer uma mutação e se tornar 1011, caso o terceiro bit seja invertido.

Cada iteração desses processo composto por seleção, recombinação e mutação é chamado de geração. A cada geração, uma população de novos indivíduos é gerada, idealmente com indivíduos com melhor valor de *fitness* que na geração anterior.

Algoritmos Genéticos tem um bom resultado como técnica de otimização devido ao Teorema de *Schemas* (*Schema Theorem*, no original em inglês), formalizada por Holland em 1975. Segundo Holland (1992a), Algoritmos Genéticos descobrem e recombinam conjuntos de dados com boa aptidão em solucionar problemas, como se fossem blocos básico para construção de soluções, chamados de *schemas*. Estes blocos consistem de *strings* que podem assumir valores 1, 0 e $*$, sendo que este último representa um valor que não importa (*don't care*). Com esses três valores podemos formar padrões, como por exemplo $1**1$, que representa todas as possibilidades de uma *string* de 4 bits cujos primeiro e último valores sejam 1. A própria presença desses *schemas* em uma população representam seu potencial em contribuir para uma boa solução, proporcionalmente à quantidade de vezes que se apresentam, já que a maior frequência em uma população representa a boa aptidão de tais *schemas*.

2.4 RESUMO

Nesta capítulo foram apresentados os principais conceitos relevantes ao trabalho desenvolvido. Verificação Funcional é o ponto de

partida deste trabalho, sendo a razão do problema a ser solucionado. A dificuldade em se verificar os complexos sistemas produzidos atualmente causa uma crescente necessidade de esforço no desenvolvimento de novas técnicas que consigam superar as limitações humanas e computacionais do processo de verificação. Metodologias como a UVM e CDV, além de linguagens específicas para desenvolvimento em alto nível, como a biblioteca *SystemC*, contribuem neste quesito, reduzindo a quantidade de tempo e investimento necessário para garantir um produto de qualidade.

Algoritmos Genéticos surgem nesse cenário como um possível método para otimização de recursos, como é o caso do problema proposto neste trabalho. Sua capacidade de se adequar a um problema, explorando soluções de maneira aleatória, e ao mesmo tempo, focando-se no objetivo perseguido, tornam esta uma atraente alternativa para substituir a análise de resultados de testes realizada tradicionalmente por engenheiros.

No próximo capítulo, serão abordados outros trabalhos com objetivos semelhantes ao deste, que se utilizam das mesmas técnicas e conceitos aqui apresentados.

3 TRABALHOS RELACIONADOS

Neste capítulo, serão abordados trabalhos correlatos nas áreas de geração automática de testes baseada em cobertura, e metodologias para criação de ambientes de teste em *SystemC*.

3.1 GERAÇÃO AUTOMÁTICA DE TESTES BASEADA EM COBERTURA

Nos trabalhos (BRITO; FRANCO; SILVA, 2013) e (FRANCO; SILVA, 2014), são descritos o processo de implementação de um AG no fluxo de verificação, e a criação de *templates* para inserir uma classe responsável por seu funcionamento em um ambiente de testes desenvolvido em *SystemC*. Essa classe desenvolvida em *C++* destina-se à implementação na metodologia de *testbenches VeriSC* (SILVA; MELCHER; ARAUJO, 2004), modificando sua estrutura de forma a acrescentar a geração de dados do AG, e uma realimentação das informações coletadas pelo *checker*. O *template* facilita a implementação de um AG no ambiente, através de métodos responsáveis pela interação do AG com o *testbench* e seus operadores.

O trabalho de Samarah et al. (2006) propõe uma metodologia para uso de Algoritmos Genéticos em verificação funcional que consiste em utilizar células compostas por um conjunto de informações no lugar de codificação binária no AG. Estas células definem dois limites numéricos (superior e inferior) para a randomização de parâmetros de entrada (como sinais do modelo) e um peso para esta célula, que representa a chance de ser gerado um valor dentro dos respectivos limites.

Para utilizar estas células no processo do AG, mudanças são realizadas nos operadores de recombinação e mutação. Na recombinação, além de *crossovers* tradicionais separando o cromossomo entre cada célula, recombinações cruzando o conteúdo das próprias células para gerar novas completamente distintas é apresentado e, na mutação, diversos operadores para mudar parâmetros por meio de deslocamento de limites e mudanças de pesos.

Uma estratégia de otimização para múltiplos objetivos é utilizada para garantir uma avaliação relevante das funções *fitness* utilizadas: primeiramente, deseja-se atingir todos os objetivos ao menos uma vez, independente da taxa de ativação, seguido de crescentes limiares que devem ser atingidos com o passar das simulações, garantindo que

todos os objetivos são atingidos de forma equilibrada. Comparando essa abordagem com testes randômicos, os autores obtiveram um valor de cobertura até três vezes maior, dependendo da métrica. Este trabalho também inclui orientações de como definir parâmetros gerais do AG como tamanho de população, número de células, entre outros, para otimizar o desempenho desta metodologia.

No trabalho de Cheng e Lim (2009), a geração automática de testes é abordada utilizando Algoritmos Genéticos junto a uma otimização multi-objetivo através do conceito de Eficiência de Pareto (PARETO, 1896). Esse princípio de economia define que em um determinado grupo de agentes, ocorre um Ótimo de Pareto quando não há formas de se melhorar o desempenho de um agente em específico sem que haja a degradação de qualquer um dos outros agentes da situação. Esse conceito tem aplicação em diversas áreas da engenharia, além de economia.

Para utilização na verificação de *hardware*, o trabalho de Cheng e Lim (2009) considera três diferentes tipos de cobertura: l (cobertura de linhas de código), t (cobertura de variações de sinais) e c (cobertura de condicionais do código), além de uma função $f_s(x)$ que avalia o tamanho do teste como diferentes objetivos de otimização e, consequentemente, diferentes agentes de Eficiência de Pareto, tentando assim otimizar a função

$$\underset{x \in X}{max} \;\; \{f_l(x), f_t(x), f_c(x)\} \;\; e \;\; \underset{x \in X}{min} \;\; \{f_s(x)\}$$

sendo $f_l(x)$, $f_t(x)$ e $f_c(x)$ as funções *fitness* correspondentes a cada cobertura.

Os resultados dos teste são avaliados para cada objetivo de forma independente. Através da classificação de Pareto, são selecionados os melhores testes do ponto de vista de otimização global e específica de cada objetivo, mantendo a diversidade da população do AG.

O trabalho de Elver e Nagarajan (2016) consiste em um *framework* para verificação de modelos de consistência de memória através da geração de testes por Algoritmos Genéticos. Esse trabalho propõe uma nova métrica a ser considerada além da função *fitness*, chamada de adequação do teste (*test suitability*, no original em inglês). Essa métrica consiste em medir o quão adequado um teste é em encontrar possíveis erros em modelos de consistência de memória utilizando conhecimento estatístico das operações que causam maior quantidade de erros. Uma recombinação seletiva é realizada para preservar sequências de cromos-

somos que contribuam com esta métrica, aumentando assim a chance de cobrir estados críticos com os testes gerados pela metodologia. Isso também é interessante para aplicações não relacionadas diretamente com memória, como encontrar possíveis *deadlocks* em um sistema a partir do momento em que hajam suspeitas que tais problemas podem ocorrer.

3.2 METODOLOGIAS DE *TESTBENCH* PARA *SYSTEMC*

Em relação à metodologias para desenvolvimento de ambientes de teste, Silva, Melcher e Araujo (2004) propõem a metodologia *VeriSC*, composta por um gerador automático de *testbenches* baseado em uma análise das entradas e saídas do DUT realizada por *parsing*, e descrições das suas transações em *SystemC*. Isto permite o início da verificação antes mesmo de um modelo completo do DUT estar disponível. O *testbench* consiste de quatro módulos: *Source*, *Driver*, *Monitor* e *Checker*. O *Source* é responsável por criar as transações que serão recebidas pelo *Driver* para serem convertidas nos sinais que estimulam o DUT. O *Monitor* recebe os sinais de saída do DUT e os converte para transações novamente, a fim de contribuir com a cobertura coletada pelo *Checker*. A metodologia, implementada em *SystemC*, utiliza a biblioteca SCV (*SystemC Verification Library*) (ACCELLERA SYSTEMS INITIATIVE, 2014) para gerenciar aspectos de geração randômica e restrições, e obteve sucesso ao encontrar novos erros em um decodificador MP3.

No trabalho de (MEFENZA; YONGA; BOBDA, 2014), é apresentado um gerador automático de *testbenches* em *SystemC* para a metodologia UVM. É necessário um DUT também em linguagem *SystemC*, além de definições de cobertura e asserções, para a geração do ambiente de testes completo. Embora o trabalho cubra todos os aspectos da UVM com o ambiente proposto, a biblioteca de cobertura e os *checkers* são implementados em *SystemVerilog* devido à ausência destes conceitos na biblioteca básica de *SystemC*. Com isso, é necessária uma integração entre linguagens para utilização deste *testbench*. O ambiente de testes é descrito detalhadamente, incluindo também o fluxo de verificação que deve ser realizado para seu uso.

O trabalho de (S.OLIVEIRA et al., 2012) introduz a metodologia SVM (*System Verification Methodology*) como uma biblioteca TLM para *SystemC*, inspirada principalmente na metodologia OVM (*Open Verification Methodology*). Esta influenciou diretamente o desenvol-

vimento da UVM, havendo muitos pontos em comum entre as duas, garantindo assim, compatibilidade em caso de integração.

A característica de reusabilidade da UVM está presente também neste trabalho, já que o *testbench* proposto utiliza apenas as interfaces do DUT durante seu processo de criação automática, possibilitando que pequenas mudanças no modelo possam ser realizadas sem interferir no *testbench*. Isso também permite que esta metodologia seja utilizada para provar equivalência funcional e um correto refinamento entre modelos de referência e sistemas em alto nível de abstração, até suas implementações RTL.

Os autores também desenvolveram uma biblioteca de cobertura (KUZNIK; MÜLLER, 2011) compatível com SVM, OVM e UVM sem dependência das funcionalidades da biblioteca padrão de verificação do *SystemC*, SCV.

A Tabela 1 mostra uma comparação entre os trabalhos mencionados neste capítulo, focando em três pontos apresentados como objetivo deste trabalho: *Testbench* de fácil utilização, Geração de testes à partir de AG com metodologia de implementação, e ambiente de testes seguindo uma metodologia de verificação.

3.3 RESUMO

Neste capítulo, foram abordados trabalhos correlatos ao trabalho descrito na presente dissertação. Embora esses trabalhos utilizem os mesmos conceitos que serão desenvolvidos ao longo desta dissertação, diferentes abordagens são possíveis, com objetivo de melhorar determinados aspectos da área de verificação.

Os trabalhos aqui abordados dividem-se principalmente em duas áreas: uso de AG para automatizar geração de testes, e desenvolvimento de metodologias para criação de *testbenches* em *SystemC*. O presente trabalho une duas áreas, desenvolvendo tanto a metodologia para uso do AG na geração de testes, quanto propondo um ambiente de testes que possa ser integrado nessa metodologia.

No próximo capítulo, será abordado o ponto principal deste trabalho: a metodologia para uso de AG em verificação, suas características e diferenças em relação a uma implementação tradicional dessa técnica, assim como mostrar como esta metodologia pode ser aplicada de forma genérica em qualquer processo de verificação por simulação.

Tabela 1 – Comparação entre trabalhos correlatos.

Trabalho	Implementação *Testbench*	AG com Metodologia de Implementação	Padrão UVM
(BRITO; FRANCO; SILVA, 2013), (SILVA; MELCHER; ARAUJO, 2004)	Gerado automaticamente	AG apenas	Padrão próprio
(SAMARAH et al., 2006)	Não informado	AG e Metodologia genérica	Não
(CHENG; LIM, 2009)	Não informado	AG e Metodologia de cobertura apenas	Não
(ELVER; NAGARAJAN, 2016)	Não informado	AG e Metodologia para modelos de memória	Não
(MEFENZA; YONGA; BOBDA, 2014)	Gerado automaticamente, necessita integração entre linguagens	Não	UVM
(S.OLIVEIRA et al., 2012)	Gerado automaticamente	Não	UVM

4 METODOLOGIA PROPOSTA

O principal objetivo deste trabalho é o desenvolvimento de uma metodologia para implementação de Algoritmos Genéticos em diversos métodos de verificação de hardware por simulação. Para isso, inicia-se com a contextualização do funcionamento de um AG dentro da área de verificação, associando os conceitos básicos do algoritmo com seus respectivos equivalentes no ambiente de testes.

As entradas do sistema serão randomizadas com o auxílio do AG, com o objetivo de encontrar a maior cobertura funcional possível em cada teste. Para não lidar com estas entradas em baixo nível, através de sinais RTL, será utilizado o conceito de Sequências da UVM. Embora seja próprio da metodologia, este conceito pode ser facilmente aplicado em *testbenches* que não utilizem a UVM.

Sequências são elementos da verificação que definem um conjunto de transações a ser realizada durante um teste. A definição de uma sequência deve ser feita durante o planejamento da verificação, a partir das necessidades de testes do projeto. Exemplos de sequências são desde um único dado de um bit sendo passado ao *hardware* durante um ciclo de clock (que representaria apenas uma transação), até um complexo protocolo de comunicação de uma interface (como I2C, por exemplo, que representa uma coleção de complexas transações de dados.). Sequências são próprias de cada interface de um sistema, e definem a ordem pela qual estas interfaces são excitadas. O *testbench* desenvolvido receberá estas sequências e as executará de forma sequencial, uma por vez. Para evitar confusões com o significado usual da palavra "sequência", será utilizado "Sequência", iniciando com letra maiúscula no restante deste trabalho, para se referir ao elemento de verificação.

A ordem pela qual as Sequências serão executadas define um teste, sendo este o objeto de otimização do Algoritmo Genético. Dessa forma, cada indivíduo do AG representa um teste, ou uma simulação a ser realizada.

Além das Sequências, os seus parâmetros também serão randomizados, de forma que o algoritmo genético não influencie apenas a ordem de execução das Sequências, mas também os valores numéricos passados em cada pacote ou transação das interfaces. Parâmetros a serem randomizados incluem por exemplo, campos de um pacote de dados, valor numérico de um sinal de entrada, ou duração de um pulso de entrada em ciclos de *clock*. Estes dados são randomizados a partir

de restrições definidas pelo AG, determinando intervalos de valores que estes sinais ou campos podem assumir.

O Algoritmo 1 mostra, de forma geral, como é realizada a verificação nesta metodologia. Destacam-se três funções, *pop_generation*(), *run_test*() e *genetic_alg*() que realizam a geração da população inicial, execução de cada teste e processamento do Algoritmo Genético. A primeira e terceira funções serão detalhadas neste capítulo, enquanto *run_test*() será abordada no Capítulo 6.

Algoritmo 1: Algoritmo da implementação geral do trabalho proposto

Entrada: Número de gerações *gen_num*, número de indivíduos por geração *ind_num*, semente de randomização *seed*, número de sequências por teste *chromossome_size*

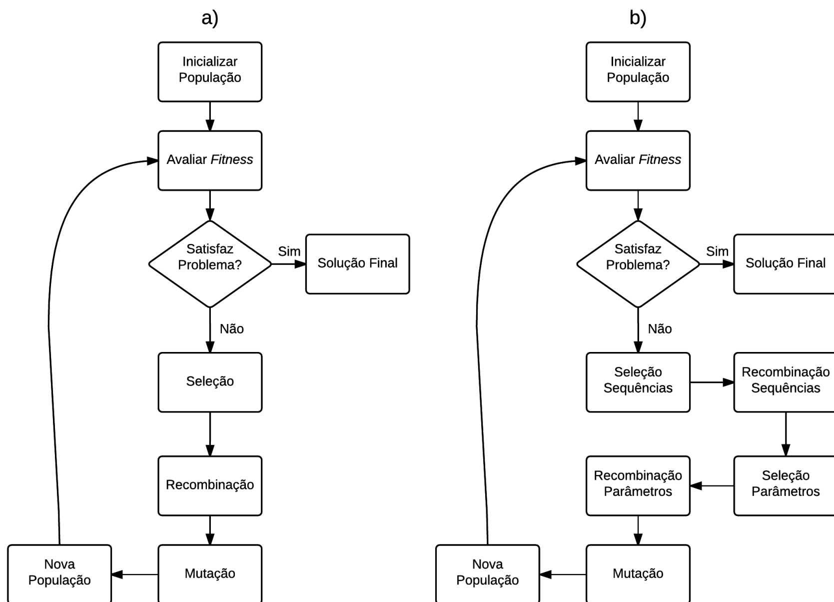
```
1 pop_generation(ind_num, chromossome_size, seed);
2 para g ← 0 até gen_num faça
3 |   para i ← 0 até ind_num faça
4 |   |   run_test(i, seed);
5 |   |   rename_cov_file(i, g);
6 |   fim
7 |   genetic_alg(g, ind_num, seed);
8 fim
```

Iniciando na Linha 1, é realizada a geração da população aleatória inicial, baseada em uma semente de randomização. Após gerada esta população, iniciam-se dois laços que percorrem o número de gerações determinado (*gen_num*) e o número de indivíduos de cada geração (*ind_num*), realizando cada teste na Linha 4. Na Linha 5, o arquivo de saída gerado pelo teste é renomeado de acordo com o indivíduo e com a geração atual e, por fim, o AG é executado ao final de cada geração, dando origem a uma nova população.

Diferentemente do fluxo de um Algoritmo Genético convencional, mostrado na Figura 11 (a), que segue uma ordem de seleção, reprodução e mutação, este trabalho propõe uma estrutura diferente, mostrada na Figura 11 (b). Essa estrutura consiste de um primeiro processo de seleção seguido por reprodução, para determinar a ordem das Sequências. Após definidas as novas sequências dos indivíduos descendentes, um novo processo de seleção e reprodução é feito para os parâmetros destas Sequências, finalizando com mutação aplicada a todos os genes.

Com essa modificação na estrutura original, é torna-se possível configurar de forma diferente os parâmetros de cada processo, permitindo uma customização maior do funcionamento do AG.

Figura 11 – Fluxos de execução de um AG tradicional (a), e do AG proposto neste trabalho (b).



4.1 CRIAÇÃO DE POPULAÇÃO

O Algoritmo 2 descreve o processo de criação randômica da população inicial. Para isso são necessários três parâmetros: o número de indivíduos, ou testes por população, o número de Sequências que cada teste irá executar, e por fim o número de Parâmetros que cada tipo Sequência contém. A seguir, será explicado o funcionamento deste processo, assim como os detalhes da inserção do AG no processo de verificação.

Algoritmo 2: Função *pop_generation*()

```
Entrada: Número de indivíduos por geração ind_num,
         número de Sequências por teste
         chromossome_size, vetor de número de
         parâmetros por Sequência parameter_pool,
         semente de randomização seed
1 para i ← 0 até ind_num faça
2 |  para j ← 0 até chromossome_size faça
3 |  |  random_basic_gene();
4 |  |  para k ← 0 até parameter_pool faça
5 |  |  |  random_parameter_gene();
6 |  |  fim
7 |  fim
8 |  save_population();
9 fim
```

Inicialmente são abertos dois laços; o primeiro (linha 1) executa o código subsequente para cada indivíduo da população inicial, e o segundo (linha 2) para cada Sequência do indivíduo correspondente. Na linha 3 é gerada a Sequência e, em seguida, para cada Sequência gerada são criados seus parâmetros (linha 4), baseando-se nas quantidades de parâmetros necessários para a Sequência respectiva (*parameter_pool*). Por fim, a população é salva em um arquivo *.txt* (linha 8).

A Figura 12 mostra a organização das Sequências e parâmetros de um único indivíduo. Em a), apenas os genes de Sequência, denominados aqui Genes Básicos (G_b) estão presentes, representando a ordem em que as Sequências devem ser executadas pelo sistema. Essa é a organização tradicional utilizada para representação de cromossomos em um AG. Já em b), o indivíduo é expandido verticalmente apresentando os genes que configuram a randomização de parâmetros, denominados aqui Genes de Parâmetros (G_p). Nesta representação são utilizadas as notações G_{bn} e G_{npm}, sendo n a quantidade de Genes Básicos de um indivíduo, e m quantos parâmetros cada Gene Básico possui.

Como sequências diferentes podem necessitar de diferentes números de parâmetros, a representação de um indivíduo é realizada efetivamente como mostrado na Figura 13 (a). Nesta forma, o principal problema ocorre devido aos diferentes tamanhos entre cada coluna, impossibilitandoo processamento dos dados em forma de vetores. Para isso, é inserido um Gene Nulo (NG) em todos os campos que ficariam vazios, como mostrado na Figura 13 (b), fazendo o tamanho total do

Figura 12 – Representações de um indivíduo do AG.

a)

G_{b0}	G_{b1}	G_{b2}	⋯	G_{bn}

b)

G_{b0}	G_{b1}	G_{b2}	⋯	G_{bn}
G_{0p0}	G_{1p0}	G_{2p0}	⋯	G_{np0}
G_{0p1}	G_{1p1}	G_{2p1}	⋯	G_{np1}
G_{0p2}	G_{1p2}	G_{2p2}	⋯	G_{np2}
⋮	⋮	⋮		⋮
G_{0pm}	G_{1pm}	G_{2pm}	⋯	G_{npm}

indivíduo depender do tamanho do maior gene. Este Gene Nulo tem apenas a função de igualar o tamanho de todas as colunas, sendo efetivamente ignorado em seu funcionamento como gene ou parâmetro.

Figura 13 – Representações de um indivíduo com tamanho de cromossomo variável.

a)

A	B	A	A	C
A_{p0}	B_{p0}	A_{p0}	A_{p0}	
A_{p1}	B_{p1}	A_{p1}	A_{p1}	
A_{p2}		A_{p2}	A_{p2}	
A_{p3}		A_{p3}	A_{p3}	

b)

A	B	A	A	C
A_{p0}	B_{p0}	A_{p0}	A_{p0}	NG
A_{p1}	B_{p1}	A_{p1}	A_{p1}	NG
A_{p2}	NG	A_{p2}	A_{p2}	NG
A_{p3}	NG	A_{p3}	A_{p3}	NG

Nesse exemplo da Figura 13, há três Genes Básicos diferentes (A, B e C), que possuem respectivamente quatro, dois e nenhum parâmetro a ser definido pelo AG.

Cada sequência deve ter seus parâmetros codificados em um número finito de Genes de Parâmetros, de preferência relativamente pequeno, de forma a evitar uma quantidade desnecessariamente grande de genes, o que aumenta consideravelmente o esforço em processamento, sem um impacto significativo no processo. Para a randomização de dados com grande alcance, deve-se separar o intervalo total em diversos

menores. Por exemplo, uma entrada A de 8 bits (256 valores) poderia ser separada em quatro intervalos $A0$, $A1$, $A2$ e $A3$, cada intervalo correspondendo a 256/4 valores. A Figura 14 mostra um exemplo deste tipo, com parâmetros de cada Sequência, e Genes Nulos. Aqui os Genes de Parâmetros são representados pela letra correspondente ao Gene Básico (A, B ou C), seguido de um número que corresponde ao intervalo de randomização.

Figura 14 – Exemplo de indivíduo com os parâmteros de cada sequência e Genes Nulos.

A	B	A	A	C
A3	B1	A0	A1	NG
A0	B0	A7	A5	NG
A2	NG	A2	A1	NG
A0	NG	A1	A1	NG

Embora não tenha sido especificada a quantidade de valores possíveis para cada parâmetro neste exemplo, não é necessário que todos estes parâmetros de uma determinada sequência utilizem todos os Genes de Parâmetros. Pode-se dizer, por exemplo, que o parâmetro A_{p0} utiliza de $A0$ a $A3$, enquanto A_{p1} utiliza de $A0$ a $A7$.

Definida a População Inicial e realizados os testes com a mesma (que será abordado no Capítulo 6), obtém-se o *fitness* de cada indivíduo, e inicia-se o processamento dos resultados pelo AG. O Algoritmo 3 descreve este fluxo.

Algoritmo 3: Função *genetic_alg*()

Resultado: Arquivo *.txt* contendo nova população *next_pop* à satestada

Entrada: Geração atual *gen*, número de indivíduos por geração *ind_num*, semente de randomização *seed*, método de seleção *selection_method*, método de recombinação de parâmetros *crossover_method*, chance de mutação de Sequência *mut_seq*, chance de mutação de parâmetro *mut_par*

```
1  se gen = 0 então
2  |  load_initial_population();
3  senão
4  |  load_last_population();
5  fim
6  selecione selection_method faça
7  |  caso seleção por torneio faça tournament();
8  |  caso seleção por roleta faça roulette_wheel();
9  fim
10 sequence_crossover();
11 selecione crossover_method faça
12 |  caso aleatório faça random_parameter_cross();
13 |  caso weighted faça weighted_parameter_cross();
14 |  senão faça nada;
15 fim
16 sequence_mutation(mut_seq);
17 parameter_mutation(mut_par);
18 new_pop();
19 save_population();
```

Entre as linhas 1 a 4, verifica-se qual a geração atual para carregar o arquivo correto de população. Caso seja a primeira geração, a População Inicial gerada pelo Algoritmo 2 é carregada. Caso contrário, a última população gerada pelo AG, é escolhida. Após isso, entre as linhas 6 e 9 escolhe-se o método de seleção para os Genes Básicos, seguido pela recombinação dos mesmos. É realizada, então a seleção e recombinação para os Genes de Parâmetro de forma conjunta, escolhendo um dos três métodos presentes entre as linhas 12 e 14 (detalhados na Seção 4.3).

A mutação é realizada em sequência, primeiramente para os Ge-

nes Básicos (linha 16), e após, para os Genes de Parâmetros (linha 17). Ao final, a nova população é montada com o resultado das operações anteriores (linha 18), e salva em um arquivo *.txt* da mesma forma que foi armazenada a População Inicial. Nas próximas seções, esses operadores do AG serão detalhados.

4.2 SELEÇÃO E REPRODUÇÃO DE GENES BÁSICOS

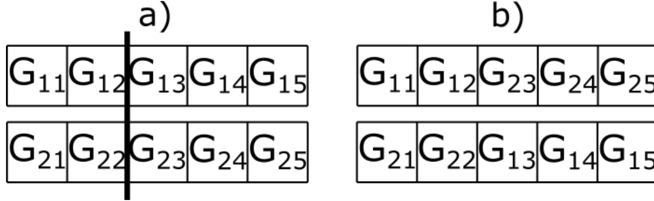
Para o processo de seleção, são utilizados os métodos clássicos de Algoritmos Genéticos, sem a necessidade de mudanças devido ao uso em verificação funcional. Uma dada população é simulada, e com base em sua cobertura, o *fitness* de cada teste é calculado. Como a cobertura de testes realizados por simulação é um valor único que varia de 0 a 100%, torna-se ideal utilizar este valor diretamente como o resultado da função *fitness* para o AG.

Foram utilizados neste trabalho, os métodos *Roulette Wheel* e *Tournament Selection*.

Após a seleção dos indivíduos aptos, inicia-se o processo de recombinação para Genes Básicos. Aqui é interessante utilizar os métodos de *One-point Crossover* ou *Two-point Crossover*, já que estes preservam os agrupamentos de genes que determinam a sequência de execução dos testes. *Uniform Crossover* não é tão interessante para os Genes Básicos devido ao fato de que, como esta aplicação do AG tem uma dependência grande em relação à ordem dos genes no cromossomo, múltiplos *crossover points* poderiam quebrar blocos com boa aptidão com maior frequência e perturbar a evolução da população. O lado positivo desse método é a diminuição da chance de uma população convergir rapidamente e gerar descendentes que sejam clones dos seus pais. Isso pode ser resolvido por meio de algoritmos que utilizem força bruta para evitar criação de descendentes clones (SPEARS; JONG, 1990).

A recombinação entre dois indivíduos é mostrada na Figura 15. Nesta, os genes são representados como G_{mn}, sendo m o número do indivíduo antes da recombinação, e n a posição do gene no indivíduo. Em a) estão representados dois indivíduos antes da recombinação, neste caso de ponto único, com a divisão realizada após o segundo gene ($cp = 2$). Em b) os descendentes gerados desta forma.

Figura 15 – Processo de *Single-point Crossover*.



4.3 SELEÇÃO E REPRODUÇÃO DE GENES DE PARÂMETROS

Após realizado o *Crossover* entre os Genes Básicos, inicia-se o processo para os Genes de Parâmetros, no qual três métodos diferentes foram implementados. Para isto, considere a Figura 16 como exemplo de dois indivíduos, também considerando $cp = 3$, e a respeito das possibilidades de randomização dos Genes A, B e C:

- Gene A possui 4 parâmetros:

  A_{p0} varia em $A0 - A3$;

  A_{p1} varia em $A0 - A7$;

  A_{p2} varia em $A0 - A3$;

  A_{p0} varia em $A0 - A1$;

- Gene B possui 2 parâmetros:

  B_{p0} varia em $B0 - B1$;

  B_{p1} varia em $B0 - B1$;

- Gene C não possui nenhum parâmetro.

4.3.1 Cruzamento sem mudança de parâmetros

No primeiro método, a recombinação dos Genes Básicos leva consigo os Genes de Parâmetros respectivos. Desta forma, as Sequências permanecem inalteradas, modificando apenas a ordem em que são executadas. Isso é mostrado na Figura 17, onde em a) estão os indivíduos antes da recombinação, com $cp = 3$. Em b), os indivíduos descendentes destes, mantendo os mesmo Genes de Parâmetros que suas respectivas Sequências possuíam nos indivíduos reprodutores.

Figura 16 – Indivíduos a passarem por processo de recombinação.

A	B	A	A	C
A3	B1	A0	A1	NG
A0	B0	A7	A5	NG
A2	NG	A2	A1	NG
A0	NG	A1	A1	NG

fitness = 0,4

B	C	B	A	A
B0	NG	B1	A2	A0
B0	NG	B1	A1	A0
NG	NG	NG	A3	A2
NG	NG	NG	A0	A0

fitness = 0,8

Figura 17 – Processo de recombinação sem mudança de parâmetros.

a)

A	B	A	A	C
A3	B1	A0	A1	NG
A0	B0	A7	A5	NG
A2	NG	A2	A1	NG
A0	NG	A1	A1	NG

B	C	B	A	A
B0	NG	B1	A2	A0
B0	NG	B1	A1	A0
NG	NG	NG	A3	A2
NG	NG	NG	A0	A0

b)

A	B	A	A	A
A3	B1	A0	A2	A0
A0	B0	A7	A1	A0
A2	NG	A2	A3	A2
A0	NG	A1	A0	A0

B	C	B	A	C
B0	NG	B1	A1	NG
B0	NG	B1	A5	NG
NG	NG	NG	A1	NG
NG	NG	NG	A1	NG

4.3.2 Cruzamento aleatório de parâmetros

No segundo método, os Genes de Parâmetros dos indivíduos reprodutores são aleatoriamente atribuídos aos indivíduos descendentes. Para isto, todos os Genes Básicos e Genes de Parâmetros de cada Sequência são agrupados e, para cada parâmetro do indivíduo des-

cendente, um gene da Sequência respectiva é aleatoriamente escolhido. Dessa forma, Genes de Parâmetros mais comuns tem uma chance maior de passarem à próxima geração. Isso é mostrado na Figura 18, onde o agrupamento de todas as sequências B de dois indivíduos reprodutores é apresentado. Para cada sequência B nos indivíduos gerados por este método, B_{p0} terá 33% de chance de ser $B0$, e 66% de chance de ser $B1$ (Figura 19), devido ao fato que cada gene do agrupamento tem a mesma probabilidade de ser selecionado.

Figura 18 – Agrupamento de todas as sequências B de indivíduos reprodutores.

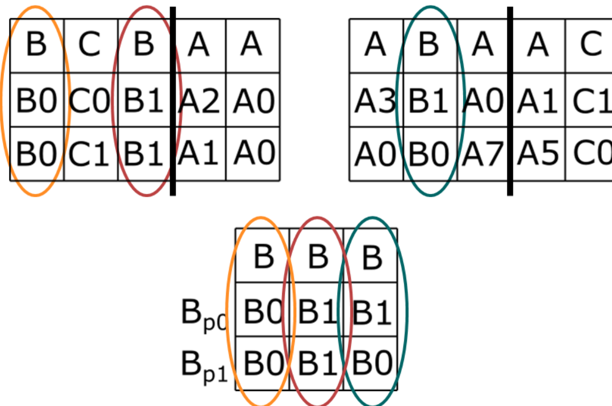
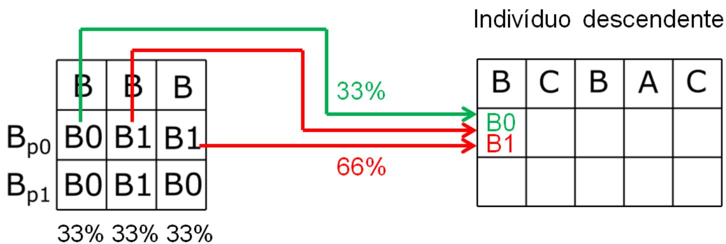


Figura 19 – Definição dos Genes de Parâmetros do indivíduo descendente.

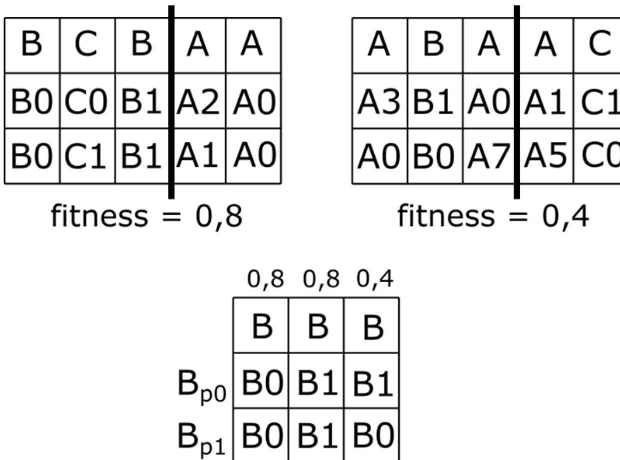


4.3.3 Cruzamento de parâmetros baseado no *fitness* do indivíduo

No terceiro método, os Genes de Parâmetros são agrupados da mesma forma mostrada no Cruzamento aleatório de parâmetros, porém a cada Sequência é associado o *fitness* do indivíduo reprodutor que a contém. Ao invés de uma seleção aleatória de cada Gene de Parâmetro dos indivíduos descendentes, o *fitness* possibilita que sejam utilizados métodos de seleção como os que foram empregados para os Genes Básicos (*roulette wheel, tournament selection*).

A Figura 20 exemplifica isso, mostrando novamente o agrupamento de todas as sequências B dos indivíduos reprodutores, com um valor de *fitness* para cada Sequência acima, representando o *fitness* do indivíduo do qual esta se originou.

Figura 20 – Agrupamento de todas as sequências B dos indivíduos reprodutores, com *fitness*.



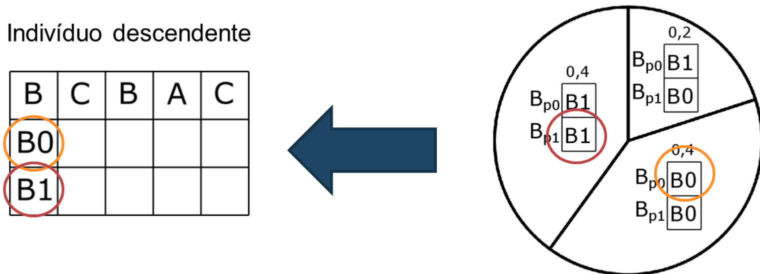
4.3.4 Seleção por Roleta para Genes de Parâmetros

Para realizar o método da roleta com Genes de Parâmetros, para cada diferente tipo de Sequência existente, uma roleta é criada, contendo em cada fatia todos os Genes de Parâmetros pertencentes à

mesma Sequência. Para cada Sequência no indivíduo descendente, a roleta é girada m vezes, sendo m o número de Genes de Parâmetro, sem contar Genes Nulos. Para cada giro, o Gene de Parâmetro correspondente G_{npm} é passado para a Sequência resultante.

A Figura 21 mostra este processo utilizando o mesmo exemplo das Sequências B utilizados na Figura 20. Aqui, para cada sequência B nos indivíduos descendentes, a roleta é girada duas vezes: uma para o parâmetro B_{p0} e outra para B_{p1}. Os valores $0,4$, $0,4$ e $0,2$ correspondem às probabilidades normalizadas dos valores de *fitness* mostrados na Figura 20.

Figura 21 – Método da roleta para Genes de Parâmetros.



4.3.5 Seleção por Torneio para Genes de Parâmetros

Na seleção por torneio, dois Genes de Parâmetro G_{npm} com mesmo m são selecionados para cada rodada do torneio. Executa-se o mesmo processo descrito anteriormente para este método de seleção, incluindo o número aleatório r e a constante k.

Em qualquer um dos métodos, pode ocorrer uma situação em que o agrupamento de todas as Sequências de um determinado tipo contenha apenas uma Sequência. Nesse caso, o processo de seleção é desnecessário, já que qualquer indivíduo descendente que possua tal Sequência receberá uma cópia exata dessa.

4.4 MUTAÇÃO

O processo de mutação inicia-se realizando uma mutação convencional nos Genes Básicos do novo indivíduo gerado. Cada gene possui

uma pequena chance de ser alterado, mudando para um gene diferente. Caso ocorra uma mutação em um Gene Básico, todos os Genes de Parâmetro associados a este são gerados aleatoriamente, já que é impossível reutilizar os mesmos parâmetros por se tratarem de Sequências de tipos diferentes.

Após a mutação dos Genes Básicos, todos os Genes de Parâmetro passam pelo mesmo processo, podendo ser alterados para qualquer outro gene pertencente à mesma Sequência. Genes que foram gerados aleatoriamente por uma mutação de Gene Básico também podem sofrer mutação novamente.

Como regra geral, a mutação de Genes Básicos deve ter uma chance menor que a de Genes de Parâmetro, já que possui um impacto maior no indivíduo, por alterar diversos genes de uma única vez.

4.5 IMPLEMENTAÇÃO

O Algoritmo Genético foi implementado como uma classe *Population* em *C++*, sendo esta linguagem escolhida para facilitar a integração com o *tesbench*, criado em *C++* com o uso da biblioteca de simulação *SystemC*. A Figura 22 mostra o diagrama de classe para *Population*.

Os membros da classe podem ser divididos em três principais grupos, sendo o primeiro destes relativo às configurações gerais do AG, que incluem tamanho da população (*m_population_size*), do indivíduo (*m_individual_size*), do conjunto de genes (*m_gene_pool*), semente de randomização (*m_seed*), entre outras.

O segundo diz respeito à população em si, incluindo vetores contendo a população (*m_individuals*), os indivíduos com melhor *fitness* (*m_top_individuals*), índices de indivíduos reprodutores (*breedersA* e *breedersB*). O tipo *gene3Dvec* foi criado para conter populações inteiras, sendo uma matriz de três dimensões de *GA::gene*, um *enum* contendo todos os possíveis genes do AG.

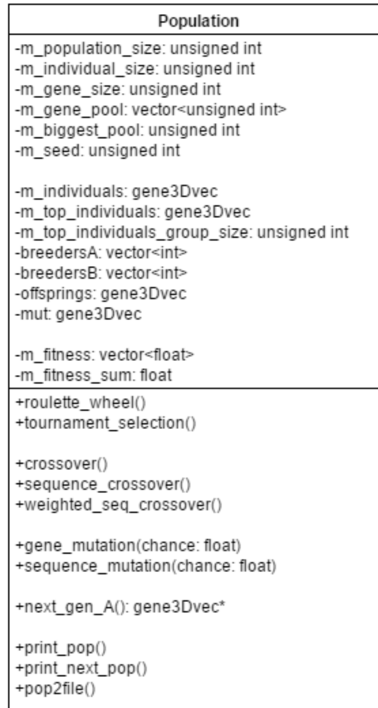
O terceiro e último grupo tem relação com o *fitness*, contendo como membros, um vetor que armazena os resultados de toda a população e variáveis auxiliares nos cálculos (*m_fitness_sum*).

As funções também podem ser divididas em vários grupos, seguindo a mesma ordem da Figura 22:

- Métodos de seleção (utilizar apenas um):

    Método da roleta;

Figura 22 – Diagrama UML da classe *Population*, responsável por todo o processo do AG.



Método de torneio;

- Métodos de recombinação:

  Crossover de Genes Básicos;

  Crossover de Genes de Parâmetros de forma aleatória;

  Crossover de Genes de Parâmetros baseado no *fitness*;

- Métodos de mutação:

  Mutação de Genes Básicos;

  Mutação de Genes de Parâmetros;

- Montagem de população;

- Métodos Auxiliares:

Exibir população no console;

Exibir próxima geração no console;

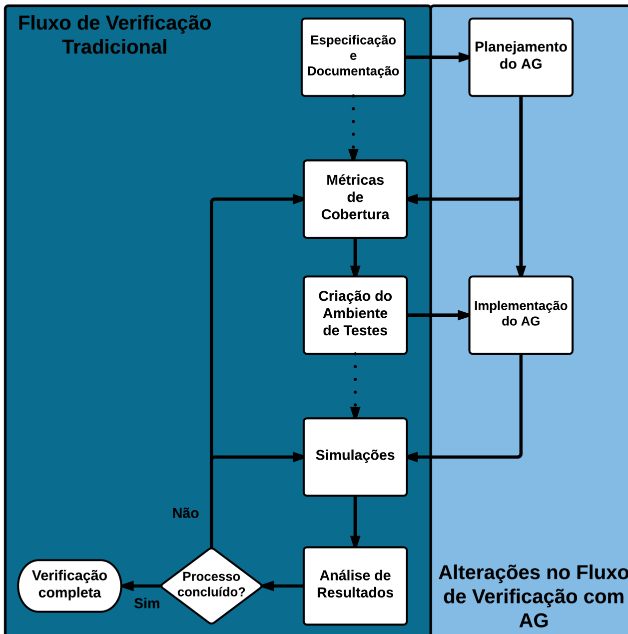
Passar população para arquivo;

Os dados de *fitness* e população atual são lidos de arquivos no formato *.txt*, assim como a nova população resultante do AG é passada ao *testbench* através de arquivos no formato *txt*.

4.6 METODOLOGIA PARA UTILIZAR O ALGORITMO GENÉTICO

Para implementar este Algoritmo Genético na verificação de um determinado sistema, algumas mudanças são realizadas no fluxo de verificação tradicional (Figura 23). Em azul escuro o fluxo tradicional de verificação, e as alterações para inclusão do Algoritmo Genético em azul claro. As setas pontilhadas representam passos do fluxo que foram alterados pela inclusão do AG.

Figura 23 – Fluxo de Verificação com AG.



A maioria destas mudanças ocorre na fase de Especificação e

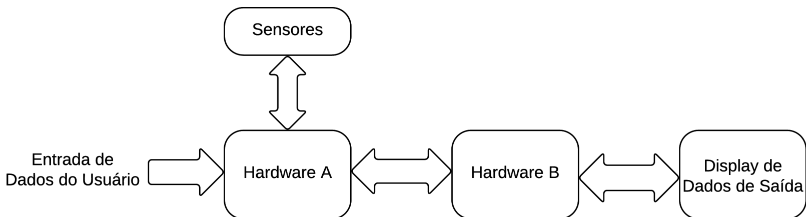
Documentação, podendo ser consideradas partes integrantes deste processo. Os passos desta etapa de Planejamento do Algoritmo Genético são descritos a seguir.

4.6.1 Separação do sistema em interfaces

Com conhecimento do DUT, o mesmo deve ser dividido em diferentes interfaces de acordo com seu funcionamento, para a definição de Sequências do AG.

De forma geral, uma interface pode ser definida como um conjunto de sinais de entrada e/ou saída que se comunicam com um mesmo módulo externo ao DUT, ou que fazem parte todos de uma mesma finalidade no modelo. A Figura 24 mostra um exemplo de sistema contendo múltiplas interfaces. Caso o Hardware A seja o DUT, este possui três interfaces: entrada de dados do usuário, sensores, e comunicação com Hardware B. No caso do Hardware B, apenas duas: comunicação com Hardware A e interface com *display* de saída.

Figura 24 – Exemplo de divisão de um sistema em interfaces.



Não é necessária a existência de mais do que uma interface. Em alguns casos, o DUT pode se comunicar apenas com um módulo externo, sobre o qual a ordem de diferentes sequências será otimizada.

Algumas interfaces podem ser apenas saídas do sistema, sem que haja entrada de dados. Neste caso, esta interface não possui importância para o AG e é desconsiderada deste passo, pois apenas dados de entrada do sistema podem ser gerados.

4.6.2 Determinar os Genes Básicos

Genes Básicos correspondem diretamente às Sequências de cada interface. Assim, para cada interface identificada no item anterior, ao

menos uma Sequência deve ser criada, para permitir a inclusão de tal interface no Algoritmo Genético. Uma Sequência é um bloco básico para comunicação entre interfaces, normalmente consistindo de parâmetros configuráveis (ou randômicos) que determinam exatamente que informações são transmitidas entre estas interfaces.

Exemplos de Sequências variam desde um simples valor numérico na entrada de um *hardware* somador, um sinal de um bit transmitindo um pulso de largura variável, até um pacote de dados de um complexo protocolo de comunicação, contendo como parâmetros determinados valores do pacote a serem fixos (cabeçalho, por exemplo). Assim, Sequências também podem variar entre casos genéricos de teste (valor qualquer passado a um sinal de entrada), ou casos bastante específicos (valor mínimo ou máximo que um sinal pode assumir, para testar casos críticos como *overflow*). Mais referências sobre construção de sequências são encontradas no Manual de Usuário da metodologia UVM (ACCELLERA SYSTEMS INITIATIVE, 2015).

4.6.3 Determinar a quantidade m de Genes de Parâmetros para cada Sequência

Para cada sequência criada no item anterior, devem ser determinados quantos Genes de Parâmetros serão necessários para todas as configurações desejadas. Este normalmente é um processo bastante direto: para cada variável a ser gerada ou randomizada em uma Sequência, um novo parâmetro é criado. Embora juntar mais de uma variável em um único parâmetro não seja algo errado, deve-se tomar cuidado para não comprometer a exploração do sistema, dessa forma impossibilitando que certas combinações de dados de entrada ocorram.

Não é absolutamente necessário que uma Sequência possua parâmetros: um caso em que essa realize algo bastante específico, como por exemplo um sinal cuja única função é emitir um pulso lógico de um bit com duração fixa, não necessita de nenhuma outra configuração para ser realizada. Assim como combinar diversos parâmetros, aqui deve-se tomar cuidado para não omitir nenhuma situação possivelmente interessante para o processo de verificação.

4.6.4 Identificar quantos genes serão necessários em cada parâmetro para randomização de dados

Definidas as sequências e a quantidade de parâmetros, deve-se identificar como serão distribuídas as configurações dos parâmetros de forma discreta, lembrando que cada parâmetro deve possuir número finito de genes que podem ser associados a este. Recomenda-se manter este número de genes pequeno, de forma a evitar diminuir a significância de cada gene, que ocorre quando há demasiados Genes de Parâmetros, e uma mudança em um destes não impacta a verificação de forma significativa. Para cada Gene de Parâmetro G_{npm} existirá um número finito de genes Gx, onde x corresponde à todas as possibilidades de randomização para este parâmetro.

No caso de parâmetros com uma ampla gama de valores, como sinais com vários bits, atribuir Genes de Parâmetro para cada valor possível é inviável, preferindo-se assim atribuir Genes de Parâmetros a intervalos de valores. Uma exceção são casos em que o valor máximo ou mínimo de uma variável seja interessante do ponto de vista da verificação, sendo útil atribuir um Gene de Parâmetro a um único valor do sinal.

Em parâmetros que definam sinais de controle, ao contrário de dados, deve-se atribuir Genes de Parâmetros a todos os valores, já que estes sinais controlam o funcionamento do sistema diretamente, e omitir ou randomizar intervalos neste caso pode causar a não-exploração do espaço de estados completo.

4.6.5 Definir as métricas de Cobertura da Verificação e *fitness*

Após todas as definições relativas ao AG em si, resta definir a função *fitness* que irá guiar todo o processo realizado pelo algoritmo. Na maioria dos casos é interessante utilizar diretamente as métricas de coberturas definidas inicialmente no plano de verificação. Porém, algumas precauções extras podem ser tomadas:

1. Genes Básicos normalmente irão contribuir em métricas de caminhos de exploração, como máquinas de estados ou cobertura de funções, enquanto Genes de Parâmetros tem maior relação com cobertura de dados de entrada e saída, tornando interessante ter uma função *fitness* que cubra aspectos simultaneamente;

2. Levar em consideração que, nesta metodologia, cada teste tem

uma duração fixa, definida pelo tamanho do cromossomo de Genes Básicos. Assim, dependendo do tamanho definido, a métrica total pode nunca ser atingida pela falta de Sequências. Idealmente, deve-se definir esta duração do teste no mesmo momento em que as métricas de cobertura para testes individuais são definidas.

4.7 RESUMO

Neste capítulo, foi abordado o uso de Algoritmos Genéticos no processo de verificação por simulação. Embora esses algoritmos sejam uma abordagem muito utilizada para diversos problemas de naturezas diferentes, mudanças em sua estrutura podem se tornar necessárias dependendo dos requerimentos de cada problema, e diversas vezes tais mudanças não são óbvias. A principal mudança aqui apresentada ocorre na estrutura do cromossomo, que passa a ter uma dimensão adicional, responsável pelos parâmetros que cada instância de uma Sequência possui, independentemente. Isso requer também adaptações nos operadores tradicionais do AG, caracterizando-se aqui por um processo de seleção e recombinação adicional para estes parâmetros adicionados ao cromossomo.

Foi apresentada também a metodologia para se aplicar este AG de forma genérica em ambientes de testes, focando nas considerações que devem ser feitas à respeito do DUT para determinar os parâmetros do algoritmo.

No próximo capítulo, será apresentado o ambiente de testes e a biblioteca de cobertura desenvolvidos para uso com esta metodologia, ainda que esta seja genérica o suficiente para ser utilizada com qualquer *testbench* independente de metodologia de verificação ou linguagem.

5 DESENVOLVIMENTO DE UMA BIBLIOTECA SIMPLIFICADA PARA O PROJETO DE *TESTBENCHES* BASEADA EM *SYSTEMC* E UVM

Para utilizar a metodologia de geração automática de testes proposta e comprovar seu funcionamento, é necessário um ambiente de verificação que possa ser integrado com essa metodologia. Como o AG foi desenvolvido na linguagem de programação *C++*, *SystemC* se torna a opção ideal para a realização dos testes pois, como se trata de uma biblioteca de *C++*, o esforço em integração do AG com o *tesbench* torna-se mínimo, e independente de ferramentas e linguagens proprietárias, já que esses são *open-source*. Por outro lado, metodologias de verificação como a UVM ainda se encontram em estágio de padronização para uso com *SystemC*, por esta se tratar de uma linguagem de desenvolvimento relativamente recente. Embora estejam disponíveis guias gerais de como se utilizar estas metodologias, ainda existem inconsistências, funcionalidades não implementadas, e questões a serem discutidas pela organização responsável pela padronização (*Accellera Systems Initiative*) no momento da realização deste trabalho.

Para resolver esta questão, foi desenvolvido um ambiente de verificação simplificado, inspirado na metodologia UVM, que preserva suas principais características simplificando, removendo ou abstraindo outras funcionalidades que não sejam essenciais a este trabalho. Ao *testbench* simplificado, foi dado o nome de *UVM-Lite*, e este capítulo detalha sua construção, funcionamento, e integração com o AG descrito no Capítulo 4. Esta biblioteca para projeto de *testbenches* foi desenvolvida com o suporte de Vinícius Alves, tendo sido objeto de trabalho de seu projeto PIBIC (Programa Institucional de Bolsas de Iniciação Científica) (ALVES, 2015).

O Algoritmo 4 descreve o funcionamento geral deste *testbench* durante um teste integrado ao Algoritmo Genético, sendo parte integrante do Algoritmo 1 mostrado anteriormente no Capítulo 4. Para simular vários testes por população, este algoritmo é executado independentemente para cada indivíduo do AG, podendo também ser utilizado para testes individuais que não utilizem nenhuma metodologia de geração de testes.

Algoritmo 4: Função *run_test*()

Resultado: Arquivo *.txt* contendo a cobertura de um teste
Entrada: Teste correspondente ao indivíduo *ind* extraído do arquivo de população do AG, número de Sequências por teste *chromossome_size*, semente de randomização *seed*

```
1  load_individual(ind);
2  connect_signals();
3  sc_start();
4  para i ← 0 até chromossome_size faça
5  |   load_sequence(i);
6  |   calculate_constraints(i);
7  |   pass_to_DUT(i);
8  |   monitor_interfaces();
9  fim
10 calculate_coverage();
11 write_coverage();
```

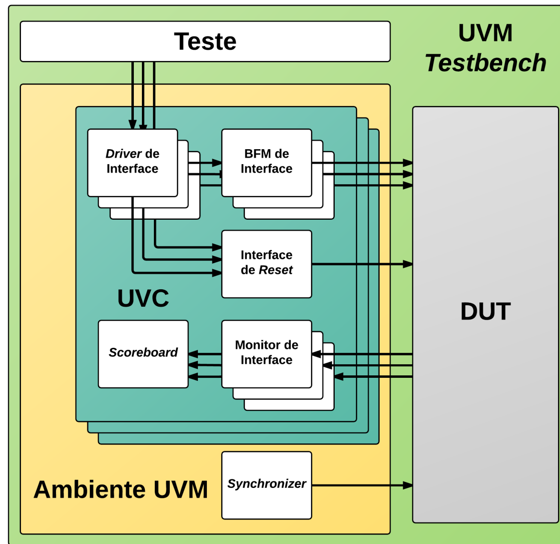
O *testbench* inicia carregando a população gerada inicialmente (ou anteriormente, pelo AG), e selecionando o indivíduo correspondente ao teste (linha 1). Após, são instanciados e conectados manualmente os sinais e módulos do ambiente em *SystemC* (linha 2), para que o teste em si possa ser iniciado (linha 3). Após o início da simulação, para cada Sequência recebida do AG, esta é carregada pelo *Driver* para a geração dos valores com base nas restrições impostas pelos Genes de Parâmetros (linhas 5 e 6), passada para o *BFM*, onde está implementado o protocolo de comunicação, passando os sinais para o DUT (linha 7). Após os eventos da Sequência serem simulados, monitores coletam os sinais de entrada e saída, além de sinais internos relevantes para a cobertura (linha 8), e enviam estas informações para o *coverage collector*, módulo do *testbench* que calcula a cobertura após o final da simulação (linha 10). Por fim, o resultado de cobertura é escrito em um arquivo *.txt* para ser carregado pelo AG (linha 11).

5.1 *TESTBENCH*

O *testbench* foi desenvolvido em *SystemC* visando simplificar a metodologia UVM para uma rápida implementação. Os principais conceitos a serem preservados são a reusabilidade entre diferentes modelos,

mecanismo de testes baseado em Sequências (para utilização do AG), e estrutura modular semelhante. As Figuras 25 e 26 mostram, respectivamente, a implementação tradicional de um *testbench* UVM, e a arquitetura geral do *testbench* UVM-Lite proposto em *SystemC* neste trabalho.

Figura 25 – Arquitetura do *testbench* UVM tradicional.

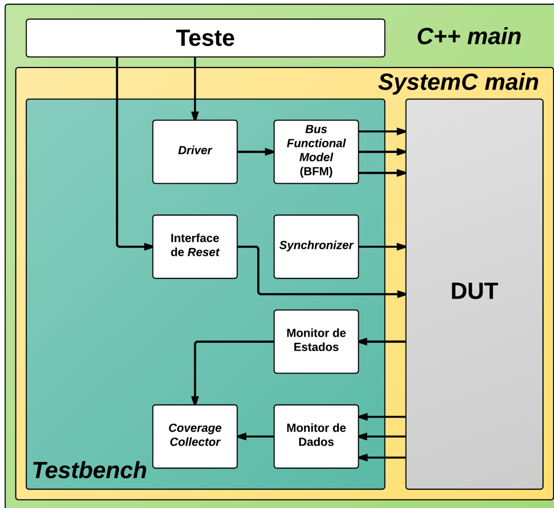


A principal diferença entre as duas arquiteturas é a unificação dos vários UVCs e BFMs/Monitores exclusivos de cada interface, para apenas uma instância. Para manter a reusabilidade, cada interface é implementada através de um método de seu respectivo módulo, tornando esse reutilizável entre diferentes modelos que compartilhem a mesma interface. Caso um DUT tenha, por exemplo, quatro interfaces, seu BFM e Monitor terão quatro métodos no *testbench* UVM-Lite.

Outra diferença é a existência de um Monitor de Estados no *testbench* proposto por este trabalho. Esse tem a função de coletar dados em relação a máquinas de estados exclusivamente, também monitorando a quantidade de Sequências já executadas, a fim de encerrar cada teste. Ao contrário da metodologia UVM tradicional, na qual diferentes mecanismos de fim de teste podem ser utilizados, neste trabalho, o número de Sequências a serem executadas é fixo devido ao funcionamento do AG, sendo esta a condição de fim de teste.

A seguir, são apresentados em maiores detalhes os principais

Figura 26 – Arquitetura do *testbench* UVM-Lite proposto.



módulos do *testbench*. Para apresentar um código genérico e sucinto, será considerado que o DUT possui apenas uma interface, denominada InA.

5.1.1 *Driver*

O *Driver* tem a finalidade de receber as Sequências geradas pelo AG, identificar os Genes de Parâmetros e associá-los às respectivas restrições para randomização. Ao contrário da UVM tradicional em que, cada interface requer um *driver* próprio, apenas um *driver* é utilizado aqui. O Código 1 mostra a estrutura do *Driver* do *testbench*.

Código 1 – driver.h

```
1   SC_MODULE(driver_uni)
2   {
3     std::vector< sequences > InA_iface;
4
5     sc_event item_done;
6     void sequence_scheduler(void);
7     template<typename T> void get_next_item(T spp);
8
9     SC_CTOR(driver_uni) {
10    sequence_scheduler();}
11  };
```

Neste módulo, é criado um vetor de Sequências para cada interface representando uma FIFO, assim como um evento para ser sinalizado pelo BFM quando este completar o protocolo da transação, de modo a permitir o envio de uma nova Sequência (linhas 3 e 5). A função *sequence_scheduler*() (linha 6) é responsável por classificar o indivíduo carregado nas FIFOs de cada interface, enquanto o *template get_next_item*() (linha 7) cria uma função personalizada para cada interface, que é chamada pelo BFM para que este possa receber a próxima Sequência a ser executada. Além disso, essa função também carrega as restrições definidas pelos Genes de Parâmetros.

5.1.2 *Bus Functional Model* (BFM)

O *Bus Functional Model* (BFM) é o módulo responsável por transformar as transações em alto nível geradas pelo *Driver* em sinais físicos (nível RTL, por exemplo) que estimulam o DUT. Da mesma forma que no *Driver*, diferentes interfaces são representadas por métodos do BFM, ao invés de diferentes instâncias do mesmo módulo, como é realizado originalmente na UVM.

Como o *testbench* executa os cromossomos gerados pelo AG de forma sequencial, é necessário um sincronismo entre os métodos de cada interface, de forma a evitar que múltiplas Sequências sejam passadas ao BFM simultaneamente. Isto é realizado através do evento *item_done* mostrado no Código 1 do *Driver*: cada método do BFM é executado como uma diferente *thread*, e estas chamam simultaneamente a função *get_next_item*(), mencionada também no Código 1. Caso a próxima Sequência seja a correta para a interface, essa é executada, e o evento emitido. Caso contrário, a *thread* entra em um laço de espera até que outra *thread* execute a Sequência e emita o evento.

O Código 2 descreve a estrutura deste módulo: sinais utilizados por todas as interfaces são declarados (linha 2), uma referência ao *Driver* é necessária para emitir o evento *item_done* (linha 4) e os métodos respectivos de cada interface são declarados (linha 6). Em seguida, todos os métodos declarados são inicializados com *threads* próprias (linha 9).

Código 2 – bfm.h

```
SC_MODULE(bfm_uni) {
  sc_inout< sc_logic > InA_signal;

  driver_uni* driver;
  void InA_tcm(void);
  SC_HAS_PROCESS(bfm_uni);

  bfm_uni(sc_module_name name) : sc_module(name) {
    SC_THREAD(InA_tcm)
  }
};
```

5.1.3 Monitor de Dados

O primeiro dos monitores é responsável pela aquisição de dados relacionados aos sinais de entrada, saída, e internos do DUT. O *Datapath Monitor* pode ser considerado um BFM reverso, que utiliza o mesmo protocolo para transformar sinais RTL em transações novamente, para serem contabilizadas na cobertura, caso sejam relevantes. Neste módulo, são também necessários métodos separados para cada interface. Porém, ao contrário do BFM, podem ser necessários dois métodos para cada interface, caso possuam sinais de entrada (método *ingress*) e sinais internos ou de saída (método *egress*) a serem coletados. Outra diferença em relação ao BFM, é que nos métodos deste módulo deve ser implementada uma condição para se iniciar a cobertura. Por exemplo, a cada ciclo de *clock*, um determinado evento ou valor de sinal, já que não há um equivalente aqui para a passagem de Sequências do *Driver*.

O Código 3 descreve a estrutura deste módulo. Os sinais relevantes às interfaces sendo monitoradas devem ser declarados (linha 2), acompanhados por referência ao módulo *Coverage Collector*, responsável por implementar o modelo de cobertura, que será abordado na seção 5.1.5. Em sequência, os métodos *ingress* e *egress* de cada interface são declarados (linhas 6 e 7) e atribuídos à suas *threads* (linhas 10 e 11).

Código 3 – datapath_mon.h

```
1  SC_MODULE(datapath_mon) {
2    sc_inout< sc_logic > InA_signal;
3
4    coverage_collector* cov_collector;
5
6    void InA_tcm_ingress(void);
7    void InA_tcm_engress(void);
8
9    SC_CTOR(datapath_mon) {
10     SC_THREAD(InA_tcm_ingress)
11     SC_THREAD(InA_tcm_engress)
12   }
13 };
```

5.1.4 Monitor de Estados

O segundo monitor é encarregado de coletar informações acerca das transições de máquinas de estados presentes no DUT, controlar o mecanismo de fim de teste, quando todas as Sequências já tiverem sido executadas, e de detectar possíveis *deadlocks* causados pelo *testbench* ou em relação ao funcionamento do DUT.

A função *collect*() deste módulo lê o valor de um ponteiro do tipo *enumerado* referente ao estado (ponteiro que está conectado diretamente ao DUT), e contabiliza o estado encontrado em um *array* contendo o número de ocorrências de cada estado da FSM. Este processo é realizado a cada ciclo de *clock*, a coletando o estado atual, mesmo que não ocorra nenhum avanço na FSM. A função *deadlockControl*() pode ser utilizada para encerrar um teste caso a FSM esteja parada em um estado, sem avançar por um determinado número de ciclos de *clock*. Isto é útil para evitar possíveis casos de problema no *testbench*, caso o BFM esteja aguardando uma condição impossível de ocorrer para finalizar uma Sequência. Caso seja chamada, o teste é interrompido. Porém sua cobertura ainda é considerada para o AG.

O Código 4 mostra a estrutura geral deste módulo. Referências são feitas ao *Driver* (linha 2), para realizar limpeza das FIFOs que guardam Sequências ao final do teste, ao *Coverage Collector* (linha 3), passar os dados referentes à cobertura de FSM, e para chamar a função que contabiliza a cobertura total após o término do teste (já que o término é também gerenciado por este módulo). Um *array* para armazenar a contagem de estados é instanciado (linha 5) e seus valores iniciados (linhas 13 e 14). As funções *collect*() e *deadlockControl*() são declaradas (linhas 9 e 10), porém, apenas a primeira é atribuída a uma

thread (linha 15), já que essa realiza uma chamada à segunda, caso necessário. A função *cover*() (linha 17) é também chamada por *collect*(), sendo responsável por encontrar a posição do *array* que contém o estado atual para armazenar a contagem

Código 4 – state_mon.h

```
SC_MODULE(state_mon) {
  driver_uni* driver;
  coverage_collector* cov_collector;

  int st_states[state_num];

  SC_HAS_PROCESS(state_mon);

  void collect(void);
  bool deadlockControl(void);

  state_mon(sc_module_name name) : sc_module(name) {
    st_states[0] = 0;
    st_states[1] = 0;
    SC_THREAD(collect);
  }
void cover(ack_layer_state CS);
};
```

5.1.5 *Coverage Collector*

O módulo *Coverage Collector* coleta dados dos dois monitores do *testbench* e calcula a cobertura total do teste com o auxílio da biblioteca de cobertura desenvolvida para este trabalho (Seção 5.2).

Três funções principais são utilizadas neste módulo. A primeira é $metrics_definition()$, que configura as métricas de cobertura, definindo as variáveis que devem ser coletadas, seu peso, e os intervalos de cobertura. A função $transactionCover()$ define os intervalos de amostragem para cada grupo de cobertura, como por exemplo, todo ciclo de *clock* para FSM, ou apenas quando algum monitor enviar dados para cobertura de sinais. A terceira função, $getCoverage()$, é chamada pelo Monitor de Estados quando o teste está concluído, para mostrar os valores coletados e calcular a função *fitness* do AG.

O Código 5 apresenta a estrutura deste módulo, iniciando pelas três funções mencionadas acima sendo declaradas (linhas 2 a 4). Dessas, $transactionCover()$ é iniciada como uma *thread* (linha 7), $metrics_definition()$ é executada apenas na inicialização do construtor (linha 8), e $getCoverage()$ não é inicializada, já que será chamada pelo Monitor de Estados. Uma função para cálculo do *fitness* é declarada (linha 11), porém esta será chamada por $getCoverage()$. Por

fim, objetos *Covergroup* da biblioteca de cobertura são criados para armazenar os dados referentes às métricas (linhas 13 e 14).

Código 5 – coverage_collector.h

```
1  SC_MODULE(coverage_collector) {
2    void metrics_definition();
3    void transactionCover();
4    float getCoverage();
5
6    SC_CTOR(coverage_collector) {
7      SC_THREAD(transactionCover);
8      metrics_definition();
9    }
10
11   float fitness_calc();
12
13   Covergroup data;
14   Covergroup states;
15 };
```

A função $fitness_calc()$, utilizada para calcular *fitness* deve ser implementada pelo usuário, de forma a se adequar às suas métricas ou aos parâmetros do AG, no caso deste trabalho.

5.1.6 Outros Módulos e Arquivos

Outros módulos presentes no *testbench* incluem:

- Ambiente de *reset*;
- *Synchronizer*.

O ambiente de *reset* configura parâmetros relativos ao *reset* do *testbench*, como conexão do sinal responsável, duração do *reset*, entre outros. O *Synchronizer* tem um papel semelhante, mas relativo ao *clock* do sistema.

Além de módulos, outros arquivos configuram parâmetros do *testbench*:

- Definições globais: configurações globais tanto do *testbench*, quando para o AG, como tamanho de população e cromossomos, número de genes e Sequências disponíveis, taxas de mutação, lista de genes e sua equivalência em Sequências, entre outras;
- Definições de interfaces: define os sinais pertencentes a cada interface, e os organiza em *structs*;
- Extensões SCV: implementação macros da biblioteca SCV relativos à randomização e restrições;

5.1.7 *Testbench*

O módulo *Testbench* é responsável por instanciar todos os outros módulos já mostrados do ambiente e por realizar as conexões entre estes e seus sinais. O Código 6 mostra a estrutura deste módulo: primeiramente são declarados os sinais relevantes, que farão conexões internas (linha 2). Em seguida, todos os módulos internos são declarados (linhas 4 a 10) e inicializados no construtor (linhas 13 a 15). Os sinais são conectados entre seus respectivos módulos (linhas 16 e 17), e as referências entre estes módulos são realizadas (linhas 19 a 25).

Código 6 – testbench.h

```
1  SC_MODULE(testbench) {
2    sc_inout< sc_logic > InA_signal;
3
4    synchronizer sync;
5    driver_uni driver;
6    bfm_uni bfm;
7    state_mon fsm_mon;
8    datapath_mon datapath_mon;
9    reset_env rse;
10   coverage_collector cov_collector;
11
12   SC_CTOR(tal_testbench) :
13   sync("sync"), driver("driver"),bfm("bfm"),
14   state_mon("state_mon"), datapath_mon("datapath_mon"),
15   rse("rse"), cov_collector("cov_collector") {
16     bfm.InA_signal.bind(InA_signal);
17     datapath_mon.InA_signal.bind(InA_signal);
18
19     bfm.driver = &driver;
20     state_mon.driver = &driver;
21     state_mon.rse = &rse;
22     state_mon.tt = this;
23     state_mon.cov_collector = &cov_collector;
24     datapath_mon.cov_collector = &cov_collector;
25     datapath_mon.driver = &driver;
26   }
27 };
```

5.1.8 Função *main*

Na função *main* tem-se a instanciação do *testbench* e DUT, além de conexões de sinais entre estes dois, e início da simulação. O código 7 descreve esta estrutura, iniciando com a criação to teste, carregado a partir do arquivo gerado pelo AG e realizado pela função *individual_creation*() (linha 2). Essa função necessita de diversas informações acerca do AG para realizar sua função, incluindo tamanho

do indivíduo, índice deste na população do AG e geração atual, entre outras. Em seguida o *testbench* e DUT são instanciados (linhas 4 e 5) e seus sinais externos declarados e conectados, incluindo um ponteiro para a FSM do modelo (linhas 7 a 11). O teste é iniciado através da chamada *sc_start*() (linha 13), e após seu encerramento, um arquivo de cobertura é gerado com o *fitness* resultante do teste (linha 14).

Código 7 – main.cpp

```
1  int sc_main(int argc, char* argv[]) {
2    individuals = individual_creation(population_size, individual_size
         , gene_size, gene_pool, individual_num, generation_num);
3
4    example_dut dut("dut");
5    testbench tb("tb");
6
7    sc_signal< sc_logic > InA_signal;
8
9    dut.InA_signal.bind(InA_signal);
10   tb.InA_signal.bind(InA_signal);
11   state_ptr = dut.state_ptr_t;
12
13   sc_start();
14   write_cov_file();
15   return 0;
16 }
```

5.2 BIBLIOTECA DE COBERTURA

Para calcular o *fitness* de cada teste, necessário ao funcionamento do AG, foi desenvolvida uma biblioteca de cobertura que possibilite coletar sinais e estados do modelo e, a partir destes, medir quanto do sistema foi explorado pelos testes gerados. Essa biblioteca consiste de duas classes chamadas *Coverpoint* e *Covergroup*. A primeira implementa a cobertura de variáveis em *C++*, as definições de intervalos, tipos de intervalos, entre outros aspectos. A segunda é responsável pela integração de vários *Coverpoints*, agrupando-os e possibilitando a realização de coberturas múltiplas, como Cobertura Cruzada.

Para realizar a cobertura de uma determinada variável, primeiro deve ser definida a variável a ser coberta, e seus intervalos (*bins*). Apenas variáveis do tipo inteiro podem ser cobertas por essa biblioteca. Os intervalos de cobertura são definidos por um vetor que recebe um par de valores, cada par representando o limite inferior e superior desse intervalo. Assim, para uma variável qualquer X, a representação deste vetor $interv_X$ para um número n de intervalos seria

$$interv_X = \{[Linf_0,\ Lsup_0],\ [Linf_1,\ Lsup_1],\ ...,\ [Linf_n,\ Lsup_n]\}$$

para $Linf_n$ e $Lsup_n$ representando, respectivamente, o limite inferior e superior de cada intervalo. Por exemplo, caso se deseje realizar a cobertura de uma variável de dois bits, considerando dois intervalos, o vetor correspondente seria $\{[0, 1], [2, 3]\}$. Além deste, é possível definir um vetor *illegal*, para valores que nunca devem ser alcançados na cobertura. Caso um valor desse vetor seja coberto, é possível configurar o *testbench* para emitir um aviso, ou encerrar a simulação com um erro. Valores que não estejam presentes em qualquer um destes dois vetores de intervalos serão desconsiderados, caso cobertos. Esses vetores são definidos pelo usuário no módulo *Coverage Collector*, através de uma instância da classe *Coverpoint*, e utilizando funções desta biblioteca para passar os valores numéricos dos seus limites.

Os valores numéricos correspondentes a quantas vezes cada intervalo foi atingido são armazenados em outro vetor, contendo metade do tamanho do vetor de intervalos, já que cada par de limites representa uma *bin* a ser coberta.

Para realizar a cobertura cruzada de dois *Coverpoints*, é necessário instanciá-los dentro de um mesmo *Covergroup*. Dentro desse, é possível realizar o cruzamento entre determinados intervalos de cada *Coverpoint*, ou entre todos os intervalos de dois ou mais *Coverpoints*.

A Figura 27 mostra o diagrama UML para as classes *Coverpoint* e *Covergroup*. Na primeira, o ponteiro $*samplingPoint$ aponta para a variável do tipo inteiro que deverá ser coberta por este objeto, enquanto $coverValue$ recebe o valor da variável. Vetores são criados para armazenar os intervalos de valores (*bins*) que deverão ser cobertos ($intervals$), assim como intervalos ilegais. Intervalos que não se tornarem parte de um destes vetores simplesmente não serão cobertos pelo *Coverpoint*. Esta classe possui três métodos principais: $sample()$, que realiza a coleta dos valores de $coverValue$ e $*samplingPoint$, chamando a função privada $cover()$ em seu corpo, $newInterval()$, para a criação de novos intervalos de valores a serem cobertos, e $resetHits()$, para zerar os valores de cobertura, caso necessário. O restante das funções retorna o valor de algum parâmetro para uso em aquivos de cobertura, ou exibição no console, como $getNumberIntervals()$, que retorna o número de intervalos criados em um determinado objeto *Coverpoint*, $getIntervalBounds$, para os limites de um intervalo, e $getHits()$, para a quantidade de pontos cobertos em todos os intervalos deste *Cover-*

point.

Figura 27 – Diagrama UML das classes *Coverpoint* e *Covergroup*.

Para a classe *Covergroup*, um vetor *cp_vector* é criado para armazenar os *Coverpoints* que se deseja agrupar. Outros vetores armazenam os intervalos de cruzamento (*crossIntervals*) e referências às variáveis sendo cobertas (*crossSamplingPoints*). Esta classe possui duas funções *newCrossInterval* para criar intervalos cruzados: uma necessita dos índices dos *Coverpoints* no vetor *cp_vector*, e dos índices dos intervalos no vetor *intervals* da classe *Coverpoint* como parâmetros para selecionar os intervalos a serem cruzados. A outra implementação

da mesma função necessita apenas dos índices do vetor *cp_vector* e realiza um cruzamento entre todos os intervalos de ambos *Coverpoints*. Da mesma forma como na classe anterior, funções com o prefixo *get* apenas retornam valores internos para propósitos de exibição.

A cobertura coletada é contabilizada na forma de *hits* (acertos) em cada intervalo definido por um dado *Coverpoint*. A transformação destas informações em um valor numérico único, de 0% a 100% (ou para a função *fitness* desejada) é realizada pela função $fitness_calc()$ do módulo *Coverage Collector*, (Seção 5.1.5). Esta função deve acessar os valores cobertos através das funções com prefixo *get* das duas classes aqui abordadas e implementar uma função matemática para cálculo do *fitness*.

5.3 INTEGRAÇÃO COM MODELO E AG

Com o *testbench* concluído, resta integrá-lo ao processo de verificação, adaptando-o para simular o modelo desejado e, no caso deste trabalho, integrando-o ao AG. Como todo o ambiente e o AG foram implementados com a mesma linguagem, assim como os modelos que se deseja verificar, a integração não requer esforço do ponto de vista de comunicação entre suas diferentes partes.

A integração é realizada através de um *Shell Script* em Linux, implementando o Algoritmo 1 em três executáveis *C++*: geração de população inicial, teste de indivíduo e execução do Algoritmo Genético. Os testes geram arquivos contendo o seu valor *fitness* respectivo, enquanto os outros dois executáveis geram arquivos contendo a população no formato mostrado na Figura 28. No exemplo desta Figura, cada bloco de 8 linhas representa um indivíduo: na primeira linha os Genes Básicos, representados por valores de 01 a 03 (neste caso, apenas três sequências diferentes) e, cada linha de cada coluna representa um valor de Gene de Parâmetro. O valor decimal do Gene de Parâmetro representa a qual sequência ele está conectado, e valores 00 representam Genes Nulos. O número de colunas e linhas por bloco representam, respectivamente, a quantidade de Sequências por teste, e a quantidade de parâmetros que o Gene Básico com maior número de parâmetros possui.

A Figura 29 mostra o fluxo de implementação deste *testbench*, sem o AG no processo de verificação. Partindo do *testbench* genérico UVM-Lite criado, inicia-se definindo as interfaces, com base no DUT a ser utilizado. Conhecendo-se as interfaces, é possível definir e adicionar

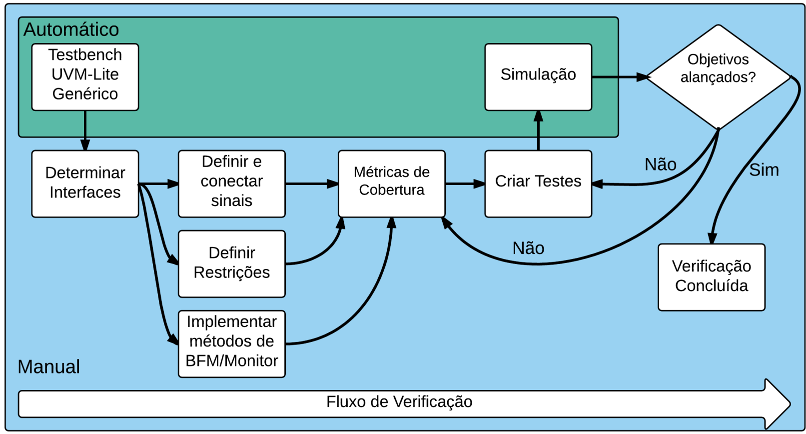
Figura 28 – Formato utilizado pela população criada pelo AG.

```
 1 01 02 03 02 03 01 03 02 02 01 02 01 02 03 02 01 01 01 02 03
 2 11 21 00 21 00 11 00 21 21 10 21 11 21 00 21 11 10 11 20 00
 3 10 00 00 00 00 10 00 00 00 10 00 10 00 00 00 11 10 11 00 00
 4 10 00 00 00 00 10 00 00 00 10 00 11 00 00 00 10 10 10 00 00
 5 15 00 00 00 00 15 00 00 00 11 00 15 00 00 00 11 12 10 00 00
 6 11 00 00 00 00 11 00 00 00 10 00 11 00 00 00 11 11 12 00 00
 7 17 00 00 00 00 12 00 00 00 17 00 10 00 00 00 15 15 14 00 00
 8 12 00 00 00 00 12 00 00 00 12 00 10 00 00 00 13 13 11 00 00
 9
10 01 02 02 03 02 01 03 02 02 02 02 03 01 01 03 03 02 03 01 03
11 10 21 21 00 20 10 00 21 20 20 21 00 11 11 00 00 20 00 10 00
12 10 00 00 00 00 11 00 00 00 00 00 00 10 11 00 00 00 00 11 00
13 11 00 00 00 00 11 00 00 00 00 00 00 11 10 00 00 00 00 11 00
14 14 00 00 00 00 11 00 00 00 00 00 00 13 10 00 00 00 00 16 00
15 11 00 00 00 00 13 00 00 00 00 00 00 13 11 00 00 00 00 13 00
16 16 00 00 00 00 13 00 00 00 00 00 00 11 14 00 00 00 00 17 00
17 11 00 00 00 00 11 00 00 00 00 00 00 11 10 00 00 00 00 12 00
18
19 03 01 03 01 02 03 02 01 03 03 01 01 01 02 02 01 01 01 03 02
20 00 10 00 10 20 00 21 11 00 00 10 10 10 21 21 11 11 11 00 21
21 00 11 00 11 00 00 00 10 00 00 10 11 10 00 00 10 11 10 00 00
22 00 11 00 11 00 00 00 11 00 00 10 10 10 00 00 10 11 11 00 00
23 00 11 00 13 00 00 00 16 00 00 15 12 12 00 00 16 14 13 00 00
24 00 13 00 12 00 00 00 13 00 00 12 12 10 00 00 12 11 11 00 00
25 00 12 00 14 00 00 00 11 00 00 10 11 11 00 00 12 12 12 00 00
26 00 10 00 10 00 00 00 12 00 00 13 10 10 00 00 10 10 11 00 00
```

ao código todos os sinais necessários pelo *testbench*, realizar suas conexões, implementar os protocolos de BFM e Monitores e adicionar ao *Driver* as restrições de randomização definidas anteriormente durante o planejamento do AG. Com base nas métricas de cobertura criadas durante o planejamento da verificação, inicia-se a criação de testes (no caso deste trabalho, inicializa-se o AG) e realizam-se simulações até alcançar os objetivos planejados.

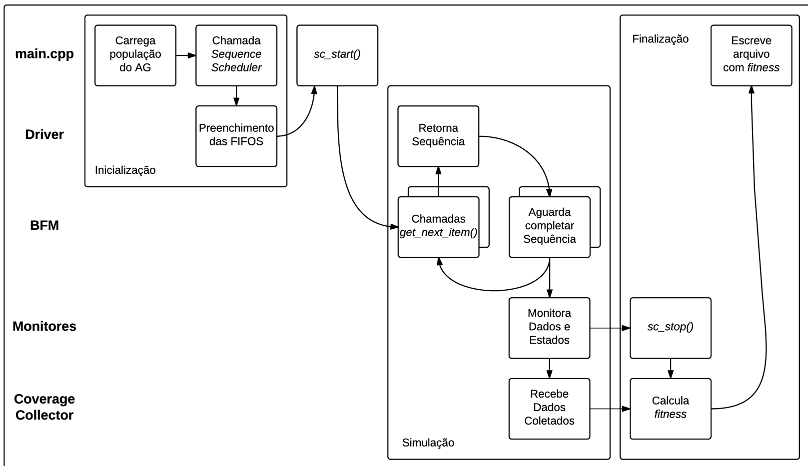
Por fim, a Figura 30 mostra o Diagrama de sequência referente ao funcionamento do *testbench* completo, dividido em três etapas: inicialização, simulação e finalização. Na primeira, inicia-se a função *main*, que carrega a população inicial do AG, e chama a função *Sequence Scheduler*, do *Driver*. Nesta, são preenchidas as FIFOS com as Sequências necessárias, possibilitando então o início do teste, pela função *sc_start()*. Inicia-se a simulação, na qual o BFM realiza chamadas da função *get_next_item()* do *Driver*, retornando uma sequência ao BFM. Este aguarda a Sequência ser executada para que o Monitor possa coletar informações, e enviar ao *Coverage Collector*. Este procedimento se repete para cada Sequência, até que não hajam mais Sequências a serem executadas, iniciando a etapa de finalização. Na última etapa, é chamada a função *sc_stop()* com o objetivo de encer-

Figura 29 – Fluxo de implementação do *testbench* proposto.



rar a simulação e, os dados coletados são processados pela biblioteca de cobertura, escrevendo um arquivo contendo o *fitness* deste indivíduo.

Figura 30 – Diagrama de sequência do funcionamento do *testbench UVM-Lite*.



5.4 RESUMO

Este capítulo detalhou as bibliotecas para criação de ambientes de teste e para coletar cobertura desenvolvidas para este trabalho. Embora existam disponíveis metodologias e bibliotecas prontas para uso, dificuldades relativas à sua utilização, e integração com o AG já desenvolvido levaram à decisão da criação de bibliotecas próprias.

Baseando-se na metodologia UVM, foi derivado um padrão simplificado UVM-Lite, mantendo a estrutura da UVM tradicional porém, abstraindo conceitos desnecessários ao funcionamento deste trabalho, considerando-se que a UVM ainda se encontra em fase de padronização para a linguagem *SystemC*. A biblioteca de cobertura desenvolvida implementa os principais conceitos necessários para a aquisição de dados e cálculo de *fitness* para o AG, com o objetivo de ser simples e prática para utilização pelo usuário.

No próximo capítulo, serão abordados os modelos escolhidos como DUT para testar a eficácia do AG, assim como o funcionamento do *testbench*, e os resultados obtidos na verificação.

6 RESULTADOS

Para comprovar o correto funcionamento do Algoritmo Genético, testes da implementação foram realizados utilizando diferentes modelos e técnicas dentro do próprio AG, para assegurar seu correto funcionamento.

Todos os testes aqui apresentados foram realizados em uma máquina virtual com sistema operacional *Scientific Linux 6.0*, em um computador com processador $i5 - 4200M$ $2.5GHz$, com 2 processadores e $4GB$ de memória dedicados à máquina virtual.

6.1 CASO DE ESTUDO: MÓDULO DE COMUNICAÇÃO DE SATÉLITE

6.1.1 Primeiro módulo: *ACK/NACK Layer*

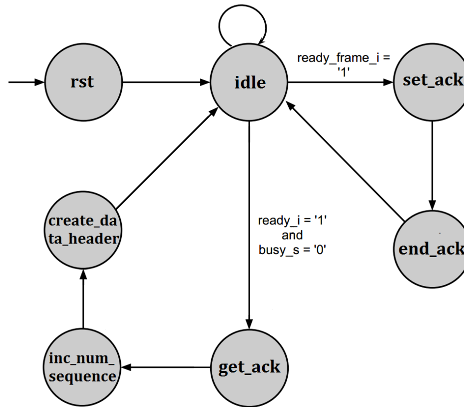
Para realização dos testes, foi escolhido como caso de estudo a Unidade de Telecomando e Telemetria (UTMC) (BEZERRA, E.; SILVA, E.; ROCHAT, D., Outubro 2009) do satélite Amazônia I. A UTMC é uma interface de comunicação de bordo para plataformas orbitais (satélites) concebida e desenvolvida de acordo com requisitos de missão fornecidos pelo Instituto Nacional de Pesquisas Espaciais (INPE). Esse sistema é responsável pela comunicação entre a estação terrestre e o computador de bordo do satélite, realizando a decodificação dos pacotes enviados pela estação, e recebendo pacotes do computador de bordo, codificando-os e enviando novamente ao controle de missão.

A UTMC é composta de diversos módulos responsáveis por funções específicas em seu fluxo de funcionamento. O sistema como um todo foi desenvolvido em linguagem VHDL (IEEE..., 2002), para implementação em FPGA, tornando este um projeto de *hardware* apenas.

Como primeiro caso de estudo foi escolhido o módulo *ACK/NACK Layer*, o menor dos módulos deste sistema. Este módulo é responsável por montar pacotes de *acknowledge* a partir dos telecomandos enviados pela estação terrestre, e passá-los para a camada de transferência, onde serão posteriormente codificados e enviados de volta à Terra. Uma FSM com 7 estados, mostrada na Figura 31, coordena seu funcionamento. O módulo *ACK/NACK Layer* também possui um total de 42 bits em sinais de entrada e 186 bits em saídas.

Originalmente desenvolvido em VHDL, o sistema foi convertido

Figura 31 – FSM do módulo *ACK/NACK Layer* da UTMC. Adaptado de (BEZERRA, E.; SILVA, E.; ROCHAT, D., Outubro 2009).



para *SystemC* através da ferramenta HIFSuite (BOMBIERI et al., 2010), da empresa *EDALab s.r.l.*. Esta ferramenta faz conversão automática entre múltiplas linguagens de descrição de *hardware* (VHDL, Verilog, *SystemC* RTL e *SystemC* TLM) através de um formato de arquivo intermediário HIF (*Heterogeneous Intermediate Format*) capaz de ser inferido a partir de qualquer destas linguagens, e capaz de gerar uma implementação também em quaisquer delas.

O *ACK/NACK Layer* recebe informações de três diferentes módulos do sistema da UTMC, definindo assim o número de interfaces para a implementação do Algoritmo Genético. Estas interfaces e seus respectivos sinais são:

- *DATA*, que recebe um pacote de dados de 40 bits, possui 7 parâmetros para randomização:

  num_ver: 3 bits;

  tipo: 1 bit;

  flag_header_dado: 1 bit;

  apid: 11 bits;

  flag_seq: 2 bits;

  cont_seq: 14 bits;

  id_error: 8 bits;

- *TC*, que recebe apenas um sinal, e um parâmetro para randomização:

  type: 1 bit;

- *FRAME*, que recebe apenas um sinal, e nenhum parâmetro de randomização:

  frame_p, um sinal de 1 bit, sendo um pulso com um ciclo de clock de duração. Este sinal não é configurado como parâmetro por apresentar sempre o mesmo comportamento.

Para a interface *DATA*, os campos do pacote recebem, respectivamente 2, 1, 1, 8, 2, 8 e 4 Genes de Parâmetros, sendo que os dados de cada campo são divididos em intervalos de mesmo tamanho para cada gene. Por exemplo, em relação ao campo *apid*, que possui 11 bits e portanto, $2^{11} = 2048$ valores divididos em 8 genes:

- *D0*: $0 - 255$
- *D1*: $256 - 511$
- *D2*: $512 - 767$
- *D3*: $768 - 1023$
- *D4*: $1024 - 1279$
- *D5*: $1280 - 1535$
- *D6*: $1536 - 1791$
- *D7*: $1792 - 2047$

Em *TC*, apenas um Gene de Parâmetro é utilizado para *type*, por ser uma variável de 1 bit. Em *FRAME*, nenhum Gene de Parâmetro é necessário.

Para definição dos parâmetros do AG, foram realizados testes de até 100 gerações, onde se observou uma evolução da cobertura principalmente entre as primeiras 20 a 30 gerações, sendo 20 o número de gerações escolhidas para realização dos primeiros testes aqui apresentados. Por se tratar de um módulo simples, cujo funcionamento é conhecido pelo autor deste trabalho, foi escolhido um número pequeno de Sequências para cada teste (14), de forma a possibilitar um pequeno grupo de soluções conhecidas capazes de atingir cobertura total da máquina de estados deste módulo. Para definição do número de indivíduos

por geração, também foram realizados testes com quantidades variadas, de 8 a 40, tendo sido observado que com poucos indivíduos (menos de 20), a solução converge rapidamente em mínimos locais, devido à falta de diversidade e, com muitos indivíduos (mais de 30) a variação prejudica a evolução de uma solução. Foram definidos assim 20 indivíduos por geração, totalizando 400 testes por semente de randomização gerada para o AG.

De forma a possibilitar um *fitness* adequado às configurações de teste propostas e com o conhecimento do *hardware* a ser verificado, a cobertura de FSM foi definida como "atingir 7 vezes cada estado possível, excluindo estados *idle* e *reset*". Neste módulo, um pacote na interface *DATA* avança em 3 dos 5 estados restantes, e um pacote *FRAME* os outros 2. Como pacotes *TC* não avançam diretamente nenhum estado, espera-se que com o passar de várias gerações estes sejam excluídos dos indivíduos.

Outra observação é que pacotes *DATA* necessitam ser seguidos de um pacote *FRAME* para processamento dos dados. Caso ocorram dois *DATA* data seguidos, o tesbench fica aguardando um *FRAME* indefinidamente, causando um *deadlock* enquanto espera. Para alcançar 100% de cobertura nessa situação, é necessário que todos os pacotes sejam uma sequência de *DATA* seguido de *FRAME*, havendo apenas duas possibilidades dentre a combinação de Genes Básicos: uma que inicie com uma Sequência *DATA* e outra com *FRAME*.

6.1.2 Cobertura de FSM do primeiro módulo

As Figuras 32 e 33 mostram os resultados para cinco processos de verificação utilizando o Algoritmo Genético. Para cada processo foi utilizada uma semente de randomização diferente, cada semente representada por uma linha no gráfico.

Na Figura 32 são mostrados os resultados para cobertura de FSM do módulo ACK/NACK Layer utilizando seleção por método de torneio. O gráfico mostra o melhor resultado de cobertura a cada geração de vinte indivíduos. Já, na Figura 33, o mesmo processo é realizado utilizando seleção por método de roleta.

Um melhor desempenho do AG ocorre quando utilizado o método de Seleção por Torneio. Para fins de comparação, 400 testes randômicos com restrições foram realizados, e a distribuição de seus resultados de cobertura é mostrada na Figura 34. Nota-se que a maior parte dos testes ficaram na faixa de 40% a 50%, sem que nenhum teste tenha

Figura 32 – Resultados para cobertura de FSM do módulo ACK/NACK Layer utilizando seleção por método de torneio.

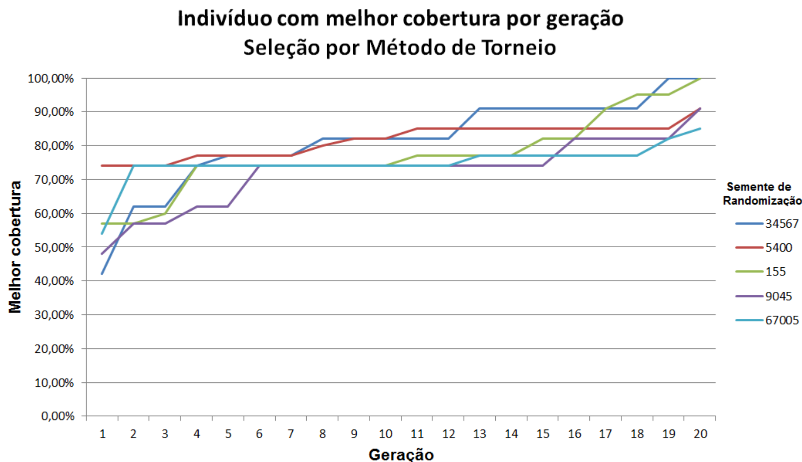
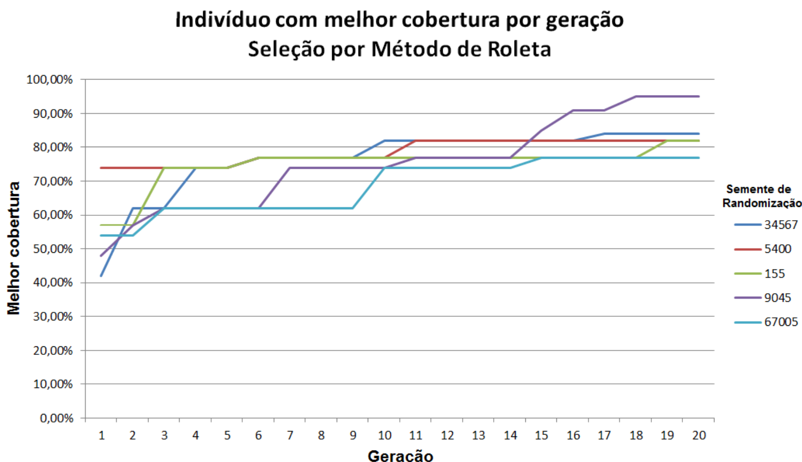


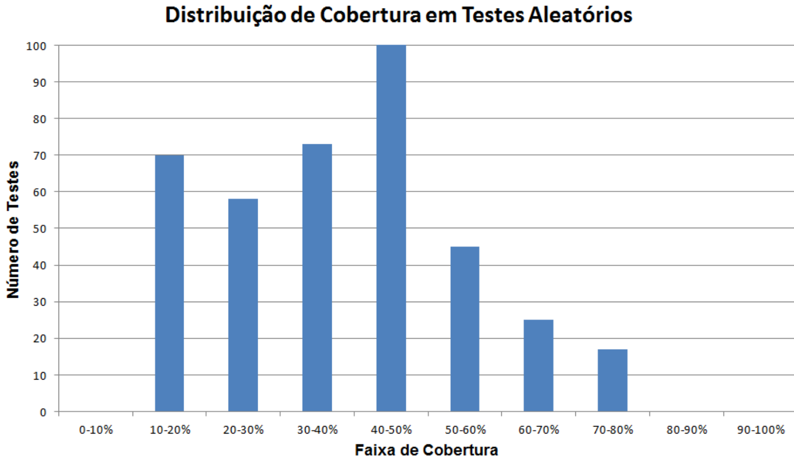
Figura 33 – Resultados para cobertura de FSM do módulo ACK/NACK Layer utilizando seleção por método de roleta.



alcançado mais de 80% de cobertura.

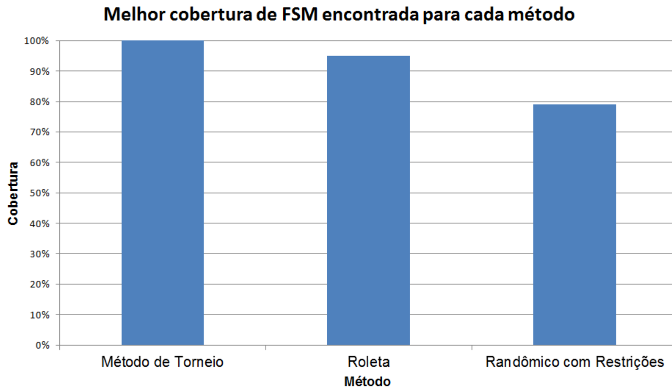
A Figura 35 compara o melhor resultado de cobertura encontrado nos testes mencionados anteriormente entre os métodos de Seleção por

Figura 34 – Distribuição de resultados para cobertura de FSM do módulo ACK/NACK Layer com testes aleatórios com restrições.



Torneio, Seleção por Roleta, e Testes randômicos com restrições. O melhor resultado foi encontrado pelo método de torneio, sendo o único que atingiu 100% das métricas de cobertura definidas.

Figura 35 – Comparação de melhor cobertura de FSM encontrada entre os métodos de seleção e testes randômicos.



6.1.3 Tempo de Simulação do primeiro módulo

Para comparação da eficiência desta implementação da metodologia em questão de tempo de simulação, foram medidos separadamente os tempos de execução de testes pelo *testbench*, tempo de geração da população aleatória e tempo de execução do Algoritmo Genético para os dois métodos de seleção utilizados, e também para testes randômicos com e sem restrições. Esses testes randômicos com restrições não utilizam o AG porém, a randomização se aproveita das mesmas restrições definidas pelos genes do AG para criar os testes. A Tabela 2 apresenta os resultados:

Tabela 2 – Tempo de execução para Cobertura de FSM do módulo ACK/NACK Layer.

Método	Seleção por Torneio	Seleção por Roleta	Randômico c/ restrições	Randômico
Geração	0,005s	0,005s	0,1s	0,5s
Teste	0,35s	0,35s	0,35s	0,06s
AG	0,007s	0,007s	-	-
Total	**181s - 201s**	**182s - 201s**	**163s - 195s**	**40s - 55s**

Como ocorrem variações no tempo de execução dos testes, do AG e da geração da população inicial, a Tabela 2 mostra o tempo médio entre os testes realizados. Para o tempo total de execução, é mostrado a variação total entre os cinco processos de verificação realizados. Nota-se aqui que o AG tem uma influência bastante pequena no tempo total de execução da verificação. A maior parte do tempo é gasta nos testes em si, independente se é utilizada a metodologia de aprendizado de máquina ou não. Para testes completamente randômicos, o tempo total de execução é bastante reduzido, já que este método não utiliza os Genes Básicos e de Parâmetros como restrições para a randomização. Assim, pode-se concluir que a maior parte do tempo é gasto no processamento dos vetores e estruturas responsáveis pela codificação dos genes, e sua interpretação no *testbench*.

6.1.4 Cobertura de FSM e *Datapath* do primeiro módulo

Para avaliar o desempenho envolvendo o *Datapath* do módulo além da FSM, novas métricas de cobertura e parâmetros para as simulações foram utilizados, de forma a tentar criar um pequeno número de possibilidades que levem a cobertura a 100%. Neste caso, os testes foram modificados de 14 para 20 sequências por indivíduo, com os seguintes parâmetros para cobertura:

- Cobertura de todos os estados da FSM exceto *idle* e *reset*, nove vezes cada. Como apenas duas das três Sequências avança a FSM, são necessários 9 Sequências *DATA* e 9 *FRAME*, restando espaço para 2 Sequências *TC*, necessárias para atingir of valores de *datapath* desejados;

- Cobertura do sinal *cont_seq* do pacote de dados de saída. Este sinal realiza uma contagem de quantos pacotes de entrada foram recebidos, iniciando a contagem em 49152 e incrementando em 1 a cada Sequência *DATA*. A cobertura desse valor exige uma contagem até 49160 (9 Sequências *DATA*), sendo cada valor atingido uma vez;

- Cobertura do sinal *pkt_length*, que representa o tamanho do pacote de dados de saída. Este sinal pode assumir dois valores, 15 ou 16, dependendo se o pacote é do tipo *ACK* ou *NACK*, o que é definido pelo valor do sinal da Sequência *TC*. A cobertura exige que cada tamanho de pacote seja atingido ao menos uma vez;

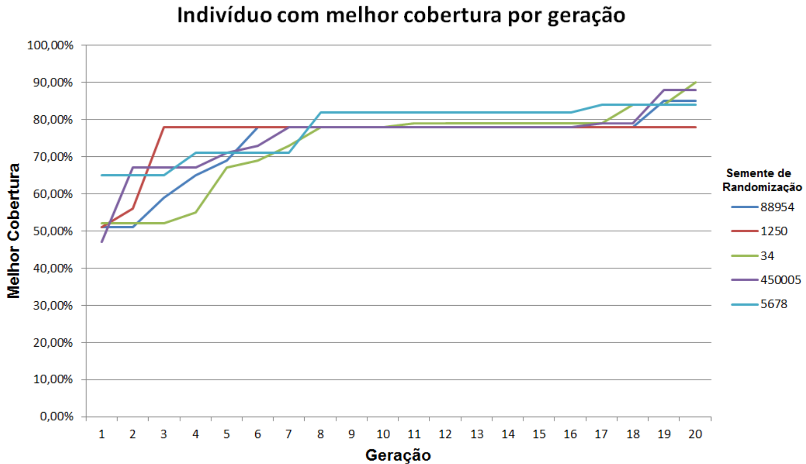
- Cobertura do sinal *num_ver* do pacote de entrada, que possui 3 bits, com cobertura em dois intervalos (0 a 3 e 4 a 7), com frequência de um *hit* em cada intervalo.

Para definir o *fitness* desta cobertura com múltiplos parâmetros, foi associado um peso a cada parâmetro, representando a sua importância no valor final. Estes pesos foram de, respectivamente 50%, 20%, 20% e 10%. Para a FSM foi escolhido o peso maior, por ser esta métrica que guia o caminho de exploração da simulação. O sinal *num_ver* recebeu o menor peso, por se tratar de um sinal que não influencia diretamente o funcionamento do sistema, sendo repassado para a saída apenas. Assim, o objetivo desta métrica é apenas assegurar que diferentes valores serão passados.

A Figura 36 mostra o resultado de melhor indivíduo por geração, para vinte gerações de vinte indivíduos. Os testes foram também

realizados com cinco sementes de randomização diferentes, seleção por torneio e recombinação de parâmetros através de cruzamento baseado no *fitness* do indivíduo (Seção 4.3.3). Na Figura 37, os resultados para o mesmo teste, realizado de forma totalmente aleatória.

Figura 36 – Resultados para cobertura de FSM e *datapath* do módulo *ACK/NACK Layer* através de Seleção por Torneio.



É possível notar que, nos testes aleatórios, a cobertura caiu bastante, devido ao fato da função *fitness* ter uma maior complexidade, necessitando de combinações de Sequências mais específicas, incluindo os Genes de Parâmetro, para obter um resultado elevado. Da mesma forma, comparando-se com o modelo de cobertura utilizado anteriormente, o AG não descobriu nenhuma sequência capaz de satisfazer um *fitness* de 100%, mas ainda assim, obteve resultados superiores à randomização.

A Figura 38 mostra os resultados para uma comparação direta entre os três métodos de recombinação de Genes de Parâmetros apresentados no Capítulo 4: cruzamento baseado em *fitness*, cruzamento aleatório de parâmetros e cruzamento sem mudança de parâmetros. A linha correspondente ao primeiro desses métodos é retirada diretamente da Figura 36 (semente de randomização 34). A mesma quantidade de testes foi realizada para os outros dois métodos.

Foi obtida uma melhor cobertura para o método baseado no *fitness* do indivíduo, enquanto o pior resultado ocorreu no cruzamento aleatório. A razão deste pior desempenho comparado ao método sem

Figura 37 – Distribuição de resultados para cobertura de FSM e *data-path* do módulo *ACK/NACK Layer* com testes aleatórios.

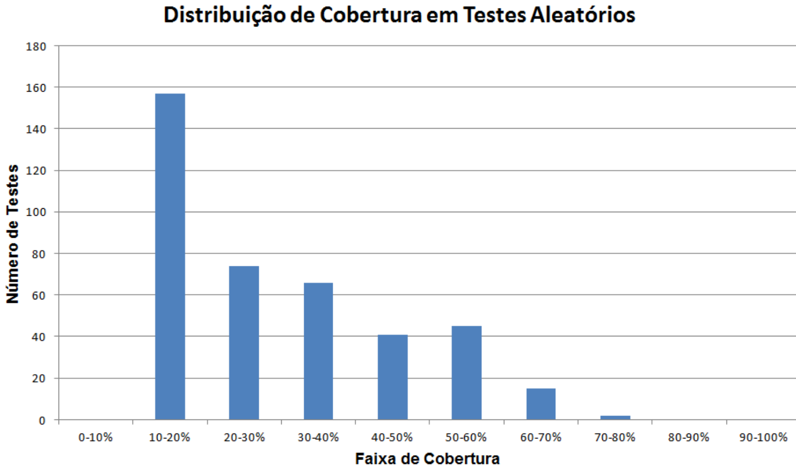
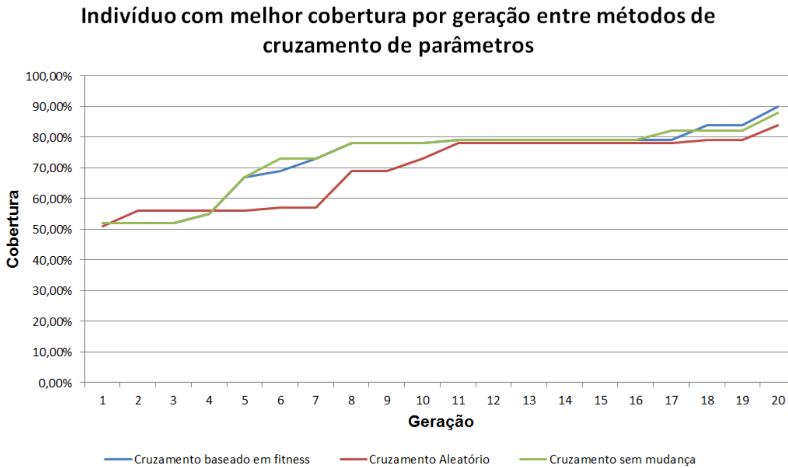


Figura 38 – Comparação de resultados entre os métodos de recombinação de Genes de Parâmetros.



mudanças é que, com a randomização os padrões de parâmetros que contribuíram para uma boa aptidão em um teste anterior podem ser facilmente perdidos com a substituição por parâmetros aleatórios.

6.1.5 Segundo módulo: *Idle Layer*

Como segundo caso de estudo foi escolhido outro módulo que compõe a UTMC. O módulo *Idle Layer* tem como objetivo principal o preenchimento de espaço ocioso dos pacotes de telemetria a serem enviados de volta à Terra, e incluir os dados recebido do *ACK/NACK Layer* nestes pacotes. O módulo permanece verificando se existe espaço em algum de seus seis *buffers* de escrita na memória para armazenar pacotes de *ACK/NACK*, e escreve nestes até que o pacote esteja completo (BEZERRA, E.; SILVA, E.; ROCHAT, D., Outubro 2009). Este módulo possui 243 bits em sinais de entrada, e 31 bits em saídas, além de uma complexa FSM composta por 38 estados.

A separação em interfaces e definição de Sequências para teste baseou-se na documentação do projeto, separando-o em 5 interfaces, de acordo com os módulos externos com os quais este sistema se comunica, e 7 diferentes Sequências:

- Interface *PACKET*, contendo sinais relativos ao início do processo de preenchimento do módulo, e configurações relativas a quais *buffers* de memória serão utilizados. Duas Sequências:

  1) Sinaliza início do processo (apenas um parâmetro para o AG);

  2) Escolha do *buffer* (dois parâmetros no AG);

- Interface *ACK*, recebe dados do *ACK/NACK Layer*. Duas Sequências:

  1) Pacote padrão enviado pelo módulo *ACK/NACK Layer* (nenhum parâmetro para o AG);

  2) Sinaliza se módulo está ocupado ou livre para envio do pacote (um parâmetro para o AG);

- Interface *FRAME*, recebe dados do módulo responsável pela montagem dos pacotes com as informações recebidas dos outros módulos. Uma Sequência:

  1) Sinaliza se módulo está ocupado ou livre para montagem de pacotes (um parâmetro para o AG);

- Interface *CLCW*, recebe a *Command Link Control Word*, que possui informações à respeito da aceitação ou rejeição de pacotes por outros módulos. Uma Sequência:

1) *Command Link Control Word* aleatório, já que esta informação é passada adiante por não ser diretamente utilizada pelo *Idle Layer* (nenhum parâmetro para o AG);

- Interface *MEMORY*, responsável por emitir sinais de controle relativos à gravação de dados nos *buffers* de memória. Uma Sequência:

1) Sequência autorizando o início de uma gravação no *buffer* selecionado pela interface *PACKET* (nenhum parâmetro para o AG).

6.1.6 Cobertura de FSM e *Datapath* do segundo módulo

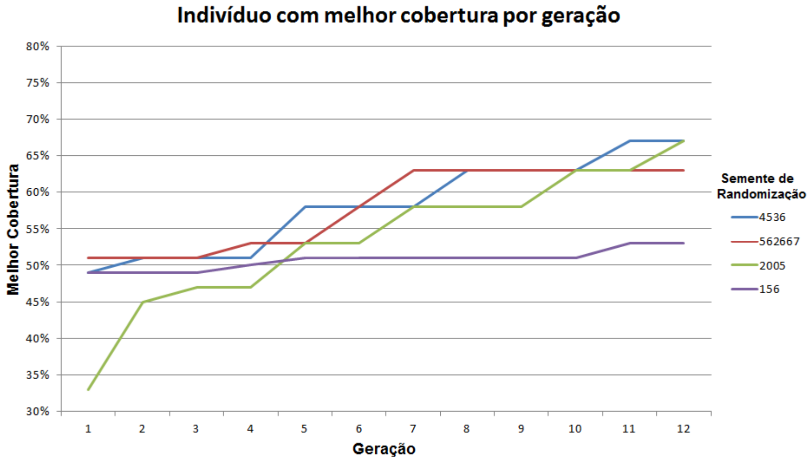
A cobertura desejada para este módulo envolve explorar todas as suas funcionalidades, e realizar operações de escrita em todos os *buffers* de memória disponíveis. Para isso, as seguintes métricas foram definidas:

- Cobertura de todos os estados da FSM com exceção de estados de *reset* e *idle* (totalizando 36 estados), com frequência mínima de uma vez por estado;
- Cobertura de escrita em todos os seis *buffers* da memória, sendo escrito ao menos um byte em cada;
- Cobertura dos sinais de entrada num_buffer e num_buffer_idle, responsáveis por definir quais *buffers* da memória serão utilizados, no mínimo 15 vezes cada.

Foram definidos pesos de 70%, 15% e 15% respectivamente para cada objetivo. Os dois últimos itens receberam o mesmo peso devido ao fato que ambas coberturas relacionam-se ao mesmo objetivo, a seleção de *buffers*. A cobertura de FSM possui o maior peso pelo fato da exploração de todas as funcionalidades do projeto a principal prioridade para esta verificação.

A Figura 39 mostra o resultado de quatro execuções do AG utilizando diferentes sementes de randomização. Cada execução corresponde a 12 gerações de 30 testes, totalizando 360 testes, sendo cada teste composto por 400 Sequências. O melhor resultado de cobertura após as 12 gerações foi de 67%, sendo que nenhuma das três métricas foi atingida totalmente em quaisquer testes.

Figura 39 – Resultados para cobertura de FSM e *datapath* do módulo *Idle Layer* através de Seleção por Torneio



Foi escolhido um número menor de gerações como parâmetro devido ao fato que estes testes seriam muito mais longos e complexos, preferindo-se então uma maior diversidade através de um tamanho de população maior. Foram realizados testes com mais de 12 gerações porém, não houve um aumento significativa da cobertura a partir da geração de número 13.

Em relação à FSM, 8 dos 36 estados não foram atingidos em nenhum dos testes realizados. Uma possível razão para esta falha na cobertura é a necessidade de testes mais longos, já que o módulo necessita de muitos ciclos de *clock* para preencher cada *buffer* completamente. Isto se torna relevante ao se considerar que cada uma das Sequências definidas para este teste são executadas ao longo de um a três ciclos, e que muitas delas podem não ter efeito na exploração da FSM, já que esta possui muitas transições incondicionais, independentes de qualquer sinal de entrada que possa ser fornecido pelas Sequências. Foram realizados testes direcionados com até 2000 Sequências para testar essa possibilidade, porém não houve uma melhora significativa na cobertura de estados.

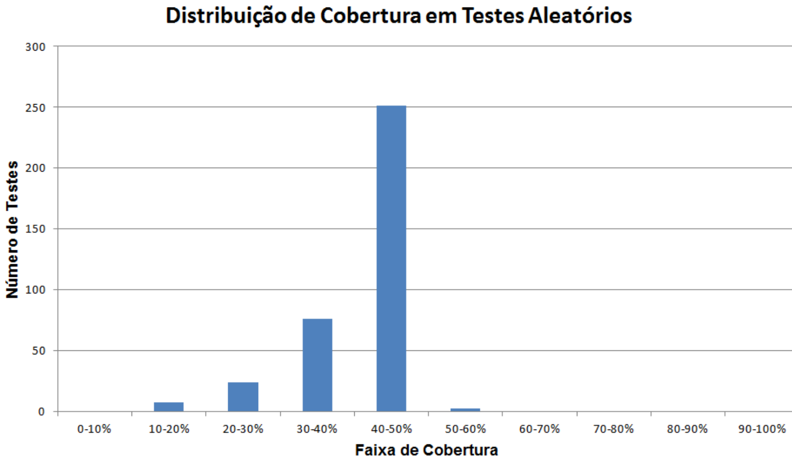
Outra possível razão é a falta de Sequências que explorem corretamente o módulo, já que estas foram criadas a partir da documentação do projeto, sendo essa documentação bastante sucinta e com informações insuficientes acerca do funcionamento do sistema. Ainda assim,

estes testes permitiram descobrir quais estados não foram explorados, possibilitando um estudo detalhado do módulo para criar testes direcionados a explorar essa área, e completar a cobertura.

Apenas um dos *buffers* foi acessado na grande maioria dos testes, mesmo com os sinais responsáveis pela escolha tendo sua cobertura quase completa. Assim, pode-se notar que a relação entre estes dois sinais ()num_buffer e num_buffer_idle) e o acesso aos *buffers* não é direta, dependendo também de outros aspectos como a FSM e o nível de preenchimento do *buffer*.

A Figura 40 mostra os resultados para testes randômicos utilizando restrições no módulo *Idle Layer*, totalizando também 360 testes. O maior valor de cobertura encontrado nestes testes foi de 53% com apenas três ocorrências, e 45% o valor mais comum.

Figura 40 – Distribuição de resultados para cobertura de FSM e *datapath* do módulo *Idle Layer* com testes aleatórios.



Como esses testes consistem de um grande número de Sequências, e a métrica definida exige apenas uma ocorrência de cada estados, a cobertura da FSM foi bastante semelhante aos testes com AG, diferenciando-se por um pequeno grupo de estados que foram atingidos pelo AG, mas não pelos testes randômicos. Além disso, em nenhuma simulação randômica houve acesso a mais de um *buffer* em um único teste.

6.1.7 Tempo de Simulação do segundo módulo

A Tabela 3 mostra os tempos de simulação para este módulo. Embora os testes realizados no *Idle Layer* possuam uma quantidade mais de vinte vezes superior em sequências, em relação ao *ACK/NACK Layer*, o tempo gasto por cada simulação diminuiu consideravelmente. Isso se deve ao fato de que no primeiro caso de estudo um grande número de parâmetros e Genes de Parâmetros foi utilizado em cada Sequência, como por exemplo os oito parâmetros da Sequência *DATA*, possuindo oito Genes de Parâmetros diferentes. Estes são processados em *C++* através de estruturas *switch/case*, consumindo muito tempo de execução, já que cada Sequência executada precisa processar várias dessas estruturas. Das sete Sequências definidas para o *Idle Layer*, três não possuem nenhum parâmetro, e entre as outras no máximo dois parâmetros, com escolhas entre apenas três Genes. Como muitas das Sequências executadas no teste não precisam realizar nenhum processamento de parâmetro, os testes se tornam rápidos.

Tabela 3 – Tempo de execução para simulação do módulo *Idle Layer*.

Método	Seleção por Torneio	Randômico c/ restrições
Geração	0,015s	0,2s
Teste	0,09s - 0,11s	0,06s - 0,08s
AG	0,08s - 0,1s	-
Total	**50s - 70s**	**45s - 60s**

Por outro lado, o tempo de execução de cada iteração do AG aumentou mais de dez vezes, devido ao trabalho repetitivo do algoritmo entre processar conjuntos de 20 (no *ACK/NACK Layer*) ou 400 Sequências, como ocorre agora. Ainda assim, considerando-se o tempo total de verificação, esse aumento de tempo é desprezível, já que mesmo com a complexidade deste módulo, o processo completo leva cerca de um terço do tempo utilizado pelo caso de estudo anterior.

6.2 DISCUSSÃO DOS RESULTADOS

Realizaram-se experimentos para validar a eficiência da metodologia aqui apresentada, utilizando dois módulos implementados em Hardware. Com base nos resultados obtidos, e na experiência durante o desenvolvimento de cada parte deste trabalho, algumas considerações

podem ser feitas a respeito de determinados aspectos deste:

6.2.1 Métodos de seleção

Comparando os dois métodos de seleção para Genes Básicos utilizados neste trabalho, a Seleção por Torneio apresentou um melhor resultado em comparação com a Seleção por Roleta. Isso ocorre devido ao fato que, a primeira possibilita uma maior diversidade de soluções, pelo fato de que indivíduos menos aptos possuem chances de serem escolhidos, devido a seleção inicial para o torneio ser aleatória.

Utilizando a Seleção por Roleta, estes indivíduos tem chance muito pequena de passar seus genes adiante, já que competem contra toda a população da geração atual, fazendo o algoritmo convergir em um mínimo local rapidamente, sem possibilidade de escapar deste, a não ser por uma mutação.

6.2.2 Métodos de cruzamento de Genes de Parâmetros

Os resultados obtidos pela comparação dos três métodos apresentados no Capítulo 4 mostraram uma pequena vantagem para o Cruzamento Baseado em *Fitness*, em relação aos outros dois métodos, sendo o Cruzamento sem Mudanças com o segundo melhor resultado, e o Cruzamento Aleatório apresentando o pior.

Essa pequena diferença deve-se, principalmente, pelo fato de que nos experimentos aqui realizados, os Genes de Parâmetros não possuem um impacto tão significativo na cobertura, visto que FSMs receberam o peso maior na cobertura, e essas são controladas principalmente pelos Genes Básicos. No caso de Genes de Parâmetros terem um impacto maior na cobertura, este método terá uma vantagem significativa.

O método de Cruzamento sem Mudanças também obteve um bom resultado por preservar as Sequências originalmente criadas para o teste, fazendo com que testes com boa cobertura continuem a gerar indivíduos com boa cobertura. Um problema deste método é que não ocorrem variações nos Genes de Parâmetros, exceto por meio de mutação, mantendo o espaço de estados explorado relativamente restrito.

Por fim, o cruzamento aleatório apresentou o pior desempenho por gerar novos padrões de parâmetros de forma randômica. Embora seu resultado tenha sido ruim para os casos aqui estudados, certas sementes de randomização poderiam apresentar resultados melhores.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho abordou a geração automática de testes para verificação por simulação, minimizando a influência humana e otimizando o esforço gasto neste processo. O foco deste foi o uso de Algoritmos Genéticos na análise de cobertura e geração de testes, assim como a criação de uma metodologia para seu uso, durante o estudo teórico realizado para a fundamentação. No entanto, foi necessário o desenvolvimento de um ambiente de testes devido à indisponibilidade de *testbenches* prontos para uso, e de fácil integração com o AG desenvolvido. Com isso, este trabalho abordou uma grande gama de aspectos do processo de verificação, e dos problemas decorrentes da integração entre esses aspectos.

7.1 CONTRIBUIÇÕES

Como contribuição científica, a metodologia desenvolvida para uso de um AG no processo de verificação mostrou um desempenho superior às bastante utilizadas metodologias de testes randômicos, com ou sem restrições. Foi observado um melhor resultado de cobertura em todos os casos testados com a metodologia desenvolvida neste trabalho, provando que a geração automática de testes baseados em cobertura contribui para uma melhora do processo de verificação, permitindo uma maior eficiência da equipe responsável por esta tarefa, já que reduz o esforço em criação de testes, possibilitando que o tempo investido nisso avance outras áreas da verificação.

Embora esta metodologia tenha se mostrado superior, cuidados devem ser tomados com a complexidade da estrutura do AG adotada em um determinado teste. Utilizar um grande número de parâmetros e Genes de Parâmetros pode levar a simulações longas, com grande custo computacional, porém sem grandes diferenças em resultados. Realizar testes randômicos com restrições utilizando o *testbench* aqui apresentado é particularmente desvantajoso, já que de acordo com os resultados obtidos, a maior parte do processamento de dados ainda ocorre na definição de restrições, e não no AG, perdendo todas as vantagens da geração automática, e mantendo o elevado tempo de simulação.

Como contribuição técnica, o ambiente de testes desenvolvido e a biblioteca de cobertura possibilitaram a verificação de modelos em *SystemC* e, mesmo com a ausência de diversos detalhes da metodo-

logia UVM original, são aptos à utilização em projetos de verificação reais, caso sejam acrescentadas certas funcionalidades, especialmente a inclusão de *checkers* no *testbench*, para permitir que este encontre erros.

A Tabela 4 mostra novamente a comparação realizada na Tabela 1, com a inclusão desse trabalho ao final.

Tabela 4 – Comparação deste trabalho com trabalhos correlatos.

Trabalho	Implementação *Testbench*	AG com Metodologia de Impletentação	Padrão UVM
(BRITO; FRANCO; SILVA, 2013), (SILVA; MELCHER; ARAUJO, 2004)	Gerado automaticamente	AG apenas	Padrão próprio
(SAMARAH et al., 2006)	Não informado	AG e Metodologia genérica	Não
(CHENG; LIM, 2009)	Não informado	AG e Metodologia de cobertura apenas	Não
(ELVER; NAGARAJAN, 2016)	Não informado	AG e Metodologia para modelos de memória	Não
(MEFENZA; YONGA; BOBDA, 2014)	Gerado automaticamente, necessita integração entre linguagens	Não	UVM
(S.OLIVEIRA et al., 2012)	Gerado automaticamente	Não	UVM
Este trabalho	Implementação manual	AG e Metodologia Genérica	UVM Parcial

Em relação ao *testbench* desenvolvido, este possui um modelo genérico para implementação em qualquer DUT, porém esta implementação ainda deve ser realizada manualmente, e requer um considerável esforço em escrita de código. Esse problema é causado principalmente pelo fato de que o ambiente não segue totalmente o conceito de encapsulamento proposto por linguagens baseadas em Orientação a Objetos, como *C++*, necessitando de mudanças internas nas classes em diver-

sos momentos de seu desenvolvimento, como por exemplo na separação de Interfaces. Além disso, comparado aos outros trabalhos estudados, este não possui um gerador automático de código para facilitar o uso do *testbench*.

O Algoritmo Genético e sua metodologia de uso proposta neste trabalho são genéricos o suficiente para possibilitar a utilização nos mais variados modelos que se deseje verificar, embora este possua um foco na exploração de FSM, devido às características sequenciais da estrutura proposta para o AG, e da metodologia UVM que este trabalho utilizou.

Embora a UVM tenha sido a principal inspiração para a criação do *testbench*, este provavelmente não será compatível com a padronização UVM para *SystemC* (que no momento de finalização deste trabalho ainda se encontra sob desenvolvimento). Isto significa que um esforço maior para refinamento deste *testbench* não será relevante no cenário de verificação, já que após o padrão ser finalizado, não haverão motivos para manutenção deste, salvo casos em que se deseje esta simplificação do padrão em algum processo de verificação.

Embora ainda haja pontos a serem melhorados, tanto na geração de testes quanto na implementação do *testbench*, este trabalho conseguiu concluir os objetivos pré-definidos, com exceção da fácil adaptação do *testbench* desejada para qualquer modelo a ser verificado.

7.2 TRABALHOS FUTUROS

Entre os pontos que podem ser aprimorados neste trabalho, o *testbench* se destaca pelas funcionalidades não implementadas (*checkers*), e a necessidade de uma implementação mais otimizada de certas partes, como a escolha das restrições (a maior causa do elevado tempo de simulação), assim como na definição das interfaces, que viola os princípios da programação orientada à objetos.

A biblioteca de cobertura possui dois pontos que podem também ser aprimorados: não foi implementada cobertura por transições, apenas de itens únicos e de cruzamento entre itens. Além disso, uma base de dados para cobertura unificada entre vários testes seria interessante para obtenção da cobertura total do processo de verificação, já que a cobertura utilizada atualmente é referente a testes individuais. Isto também tornaria atraente uma otimização de múltiplos objetivos para a definição da função *fitness*, de forma a direcionar o AG tanto para uma cobertura de um único teste, quanto para a cobertura total do processo.

A estrutura para AG proposta por este trabalho também pode ser aplicada em outros casos de verificação, como por exemplo para a geração de programas para verificação de processadores, já que além das instruções é necessário definir parâmetros como valores para cálculos, ou endereços de memória, como Genes de Parâmetros. Além da verificação, esta estrutura para AG pode ser utilizada em problemas cujos parâmetros também possuam suas próprias configurações a serem exploradas, independente da área de aplicação.

REFERÊNCIAS

ACCELLERA SYSTEMS INITIATIVE. **SystemC Verification Library**. 2014. Disponível em: <http://accellera.org/downloads/standards/systemc>.

ACCELLERA SYSTEMS INITIATIVE. **Universal Verification Methodology (UVM) 1.2 User's Guide**. October 2015. Disponível em: <http://www.accellera.org/images//downloads/standards /uvm/uvm_users_guide_1.2.pdf>.

ALVES, V. D. Verificação Formal de Computadores de Bordo (VEROBC): Relatório Final de Bolsa PIBIC, Universidade Federal de Santa Catarina (UFSC). Agosto 2015.

BEZERRA, E.; SILVA, E.; ROCHAT, D. **Relatorio técnico 72709-90000 - DESCRIÇÃO DO PROJETO DA UTMC**. Instituto Nacional de Pesquisas Espaciais (INPE), Coordenacão Geral de Engenharia e Tecnologia Espacial, Divisão de Eletrônica Aeroespacial, São José dos Campos, SP, Brasil, Outubro 2009.

BLACK, D. et al. **SystemC: From the Ground Up, Second Edition**. Springer US, 2011. (H]: [NATO ASI series). ISBN 9780387565217. Disponível em: <https://books.google.com.br/books?id=OUFvI9oi5isC>.

BOMBIERI, N. et al. Hifsuite: Tools for hdl code conversion and manipulation. In: **High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International**. 2010. p. 40–41. ISSN 1552-6674.

BRITO, A. F.; FRANCO, R. A.; SILVA, K. R. G. da. Using genetic algorithm in functional verification to reach high level functional coverage. In: **Simpósio Sul de Microeletrônica, 2013**. 2013.

CHENG, A.; LIM, C.-C. Multi-objective genetic test generation for systems-on-chip hardware verification. **Global Optimization: Theory, Methods & Applications**, p. 1, 2009.

DARWIN, C. On the origins of species by means of natural selection. **London: Murray**, v. 247, 1859.

DELAMARO, M.; MALDONADO, J.; JINO, M. **Introdução ao teste de software**. CAMPUS - RJ, 2007. ISBN 9788535226348. Disponível em: <https://books.google.com.br/books?id=7R6XPgAACAAJ>.

ELVER, M.; NAGARAJAN, V. Mcversi: A test generation framework for fast memory consistency verification in simulation. In: **2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. 2016. p. 618–630.

FRANCO, R. A. P.; SILVA, K. R. G. da. Templates de algoritmos genéticos para a geração de estímulos aplicados à verificação funcional de dispositivos. In: **Conferência de estudos em Engenharia Elétrica, 2014, Uberlândia**. 2014.

GRIFFITHS, A. **Introduction to Genetic Analysis**. W. H. Freeman and Company, 2012. (W. H. Freeman). ISBN 9781429276344. Disponível em: <https://books.google.es/books?id=dyPlTgEACAAJ>.

HOLLAND, J. H. **Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence**. Cambridge, MA, USA: MIT Press, 1992. ISBN 0262082136.

HOLLAND, J. H. Genetic algorithms. **Scientific american**, v. 267, n. 1, p. 66–72, 1992.

IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. **IEEE STD 1800-2009**, p. 1–1285, Dec 2009.

IEEE Standard for the Functional Verification Language e - Redline. **IEEE Std 1647-2011 (Revision of IEEE Std 1647-2008) - Redline**, p. 1–780, Aug 2011.

IEEE Standard VHDL Language Reference Manual. **IEEE Std 1076-2002 (Revision of IEEE Std 1076, 2002 Edn)**, p. 300, 2002.

IMAN, S.; JOSHI, S. **The e Hardware Verification Language**. Springer US, 2004. (Information Technology: Transmission, Processing & Storage). ISBN 9781402080234. Disponível em: <https://books.google.com.br/books?id=Zqqy_bulKogC>.

KUZNIK, C.; MÜLLER, W. Functional Coverage-driven Verification with SystemC on Multiple Level of Abstraction. **Proceedings of DVCON**, mar. 2011.

MARWEDEL, Peter. **Embedded Systems Design: Embedded Systems Foundations of Cyber-Physical Systems**. : Springer, 2011. 400 p. ISBN 978-94-007-0257-8.

MEFENZA, M.; YONGA, F.; BOBDA, C. Automatic uvm environment generation for assertion-based and functional verification of systemc designs. In: **2014 15th International Microprocessor Test and Verification Workshop**. 2014. p. 16–21. ISSN 1550-4093.

MITCHELL, M. **An Introduction to Genetic Algorithms**. Cambridge, MA, USA: MIT Press, 1998. ISBN 0262631857.

PARETO, V. **Cours'd'économie politique**. 1896.

PIZIALI, Andrew. **Functional Verification Coverage Measurement and Analysis**. : Springer, 2008. 216 p. ISBN 978-1-4020-8026-5.

ROSENBERG, S.; MEADE, K. **A Practical Guide to Adopting the Universal Verification Methodology (UVM) Second Edition**. : Raleigh, NC: Cadence Design Systems, 2013. ISBN 9781300535935.

SAMARAH, A. et al. Automated coverage directed test generation using a cell-based genetic algorithm. In: **2006 IEEE International High Level Design Validation and Test Workshop**. 2006. p. 19–26. ISSN 1552-6674.

SEMICO RESEARCH AND CONSULTING GROUP. 2014. Disponível em: <http://www.semico.com/>.

SILVA, K. R. G. da; MELCHER, E. U. K.; ARAUJO, G. An automatic testbench generation tool for a systemc functional verification methodology. In: **Integrated Circuits and Systems Design, 2004. SBCCI 2004. 17th Symposium on**. 2004. p. 66–70.

S.OLIVEIRA, M. F. et al. A SystemC Library for Advanced TLM Verification. In: **Proceeding of Design and Verification Conference (DVCON)**. 2012.

SPEARS, W. M.; JONG, K. A. D. **An analysis of multi-point crossover**. 1990.

SYNOPSYS, INC. **OpenVera Verification Language**. Disponível em: <http://www.open-vera.com/home.html>.

SYSWERDA, G. Uniform crossover in genetic algorithms. In: **Proceedings of the 3rd International Conference on Genetic Algorithms**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989. p. 2–9. ISBN 1-55860-066-3. Disponível em: <http://dl.acm.org/citation.cfm?id=645512.657265>.