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

Verificação funcional distribuída para projetos de circuitos integrados baseados em arquiteturas heterogêneas

Thiago Werlley Bandeira da Silva

Tese de Doutorado apresentada à Coordenadoria do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande - Campus de Campina Grande como parte dos requisitos necessários para obtenção do grau de Doutor em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Processamento da Informação

Professores Orientadores:
Antonio Marcus Nogueira Lima
Elmar Uwe Kurt Melcher

Campina Grande, Paraíba, Brasil
©Thiago Werlley Bandeira da Silva, 28 de outubro de 2021

Verificação funcional distribuída para projetos de circuitos integrados baseados em arquiteturas heterogêneas

Thiago Werlley Bandeira da Silva

Tese de Doutorado apresentada em 25 de agosto de 2021

Professores Orientadores:
Antonio Marcus Nogueira Lima
Elmar Uwe Kurt Melcher

Campina Grande, Paraíba, Brasil, 28 de outubro de 2021

S586v Silva, Thiago Werlley Bandeira da.
Verificação funcional distribuída para projetos de circuitos integrados baseados em arquiteturas heterogêneas / Thiago Werlley Bandeira da Silva. – Campina Grande, 2021.
125 f. : il. color.

Tese (Doutorado em Engenharia Elétrica) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2019.
"Orientação: Prof. Dr. Antonio Marcus Nogueira Lima, Prof. Dr. Elmar Uwe Kurt Melcher".
Referências.

1. Circuitos Integrados Digitais. 2. Projeto de Hardware. 3. Verificação Funcional. 4. Arquiteturas Heterogêneas. 5. IP-core. 6. Wrapper. 7. Virtual Bus/HLA. 8. Redução do Tempo de Projeto. 9. Processamento da Informação. I. Lima, Antonio Marcus Nogueira. II. Melcher, Elmar Uwe Kurt. III. Título.

CDU 621.38:004.3(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

VERIFICAÇÃO FUNCIONAL DISTRIBUÍDA PARA PROJETOS DE CIRCUITOS INTEGRADOS BASEADOS EM ARQUITETURAS HETEROGÊNEAS

THIAGO WERLLEY BANDEIRA DA SILVA

TESE APROVADA EM 25/08/2021

ANTONIO MARCUS NOGUEIRA LIMA, Dr., UFCG
Orientador(a)

ELMAR UWE KURT MELCHER, Dr., UFCG
Orientador(a)

ANGELO PERKUSICH, D.Sc., UFCG
Examinador(a)

ALISSON VASCONCELOS DE BRITO, D.Sc., UFPB
Examinador(a)

EDNA NATIVIDADE DA SILVA BARROS, Ph.D, UFPE
Examinador(a)

IVAN SARAIVA SILVA, Dr., UFPI
Examinador(a)

CAMPINA GRANDE - PB

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO EM ENGENHARIA ELETRICA
Rua Aprigio Veloso, 882, - Bairro Universitario, Campina Grande/PB, CEP 58429-900

REGISTRO DE PRESENÇA E ASSINATURAS

1. ATA DA DEFESA PARA CONCESSÃO DO GRAU DE DOUTOR EM CIÊNCIAS, NO DOMÍNIO DA ENGENHARIA ELÉTRICA, REALIZADA EM 25 DE AGOSTO DE 2021
(Nº 336)

CANDIDATO: **THIAGO WERLLEY BANDEIRA DA SILVA**. COMISSÃO EXAMINADORA: ANGELO PERKUSICH, D.Sc., UFCG, Presidente da Comissão, ANTONIO MARCUS NOGUEIRA LIMA, Dr., UFCG, ELMAR UWE KURT MELCHER, Dr., UFCG, Orientadores, ALISSON VASCONCELOS DE BRITO, D.Sc., UFPB, EDNA NATIVIDADE DA SILVA BARROS, Ph.D, UFPE, IVAN SARAIVA SILVA, Dr., UFPI. TÍTULO DA TESE: Verificação Funcional Distribuída para Projetos de Circuitos Integrados Baseados em Arquiteturas Heterogêneas. ÁREA DE CONCENTRAÇÃO: Processamento da Informação. HORA DE INÍCIO: **08h00** – LOCAL: **Sala Virtual, em virtude da suspensão de atividades na UFCG decorrente do coronavírus e de conformidade com o Art. 8º da PORTARIA PRPG/GPR Nº 003, DE 18 DE MARÇO DE 2020)**. Em sessão pública, após exposição de cerca de 45 minutos, o candidato foi arguido oralmente pelos membros da Comissão Examinadora, tendo demonstrado suficiência de conhecimento e capacidade de sistematização, no tema de sua tese, obtendo conceito APROVADO com pequenas modificações no texto, de acordo com as exigências da Comissão Examinadora, que deverão ser cumpridas no prazo de 30 dias. Face à aprovação, declara o presidente da Comissão, achar-se o examinado, após o cumprimento das referidas exigências, legalmente habilitado a receber o Grau de Doutor em Ciências, no domínio da Engenharia Elétrica, cabendo a Universidade Federal de Campina Grande, como de direito, providenciar a expedição do Diploma, a que o mesmo faz jus. Na forma regulamentar, foi lavrada a presente ata, que é assinada por mim, ÂNGELA DE LOURDES RIBEIRO MATIAS, e os membros da Comissão Examinadora presentes. Campina Grande, 25 de Agosto de 2021.

ÂNGELA DE LOURDES RIBEIRO MATIAS
Secretária

ANGELO PERKUSICH, D.Sc., UFCG
Presidente da Comissão e Examinador Interno

ANTONIO MARCUS NOGUEIRA LIMA, Dr., UFCG
Orientador

ELMAR UWE KURT MELCHER, Dr., UFCG
Orientador

ALISSON VASCONCELOS DE BRITO, D.Sc., UFPB
Examinador Externo

EDNA NATIVIDADE DA SILVA BARROS, Ph.D, UFPE
Examinador Externo

IVAN SARAIVA SILVA, Dr., UFPI
Examinador Externo

THIAGO WERLLEY BANDEIRA DA SILVA
Candidato

2 - APROVAÇÃO

2.1. Segue a presente Ata de Defesa de Tese de Doutorado da candidata **THIAGO WERLLEY BANDEIRA DA SILVA**, assinada eletronicamente pela Comissão Examinadora acima identificada.

2.2. No caso de examinadores externos que não possuam credenciamento de usuário externo ativo no SEI, para igual assinatura eletrônica, os examinadores internos signatários **certificam** que os examinadores externos acima identificados participaram da defesa da tese e tomaram conhecimento do teor deste documento.

Documento assinado eletronicamente por **ANGELA DE LOURDES RIBEIRO MATIAS**, **SECRETÁRIO (A)**, em 25/08/2021, às 14:31, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **ANTONIO MARCUS NOGUEIRA LIMA**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 25/08/2021, às 14:57, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Alisson Vasconcelos de Brito**, **Usuário Externo**, em 25/08/2021, às 15:47, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **ANGELO PERKUSICH**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 25/08/2021, às 16:22, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **ELMAR UWE KURT MELCHER**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 26/08/2021, às 09:32, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **THIAGO WERLLEY BANDEIRA DA SILVA**, **Usuário Externo**, em 26/10/2021, às 09:56, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **1729085** e o código CRC **BA604084**.

Referência: Processo nº 23096.053020/2021-08 SEI nº 1729085

Resumo

Este trabalho trata da verificação funcional distribuída de projetos em *hardware* baseados em arquiteturas heterogêneas. Um *testbench* foi concebido para permitir a utilização de *open source* IP-*cores*. O *testbench* é baseado na biblioteca SystemC e utiliza o conceito de *Virtual Bus* da especificação da arquitetura de alto nível (HLA). Desse modo, a integração de um IP-*core* demanda o desenvolvimento de dois *wrappers* de comunicação, um externo e outro interno ao *testbench*. O *testbench* foi utilizado na verificação do projeto de um sistema composto de dois subsistemas, um que efetua a conversão de RGB para YC_bC_r (S_1) e outro que calcula a integral da imagem (S_2). Dois cenários de verificação foram considerados, no primeiro os subsistemas são conectados em série (C_1), e no segundo, são conectados em paralelo (C_2). No cenário C_1, os IP-*cores* de S_1 e de S_2 foram disponibilizadas em C++/OpenCL e SystemVerilog, respectivamente. No cenário C_2, os IP-*cores* foram disponibilizadas em C++/OpenCL e SystemVerilog, respectivamente. Nesses dois cenários foram utilizados IP-*cores* de domínio público, e os respectivos *wrappers* de comunicação externos foram implementados. Considerou-se que os "*golden models*" desses IP-*cores* eram disponíveis. No cenário C_1, usou-se o *testbench* para integrar os IP-*cores* usando o *Virtual Bus*, sendo necessário implementar *wrappers* internos para C++/OpenCL e *SystemVerilog*. No cenário C_2, foi necessário implementar *wrappers* internos para C++/OpenCL e *SystemVerilog*. No cenário C_1 a saída gerada por S_2 é comparada a saída do *golden model* de S_2. No cenário C_2 as saídas geradas por S_1 e S_2 são comparadas com os respectivos *golden models*. A utilização do *Virtual Bus*/HLA permite a integração de um *open source* IP-*core* sem a necessidade de re-codificação, elimina uma etapa do fluxo de projeto convencional, e desse modo, reduz o tempo de projeto e elimina erros. Além disso, o *open source* IP-*core* é executado na arquitetura e na infraestrutura em que for disponibilizado, sem ensejar óbice ao processo de verificação.

Palavras-chave: Projeto de *Hardware*, Verificação Funcional, Arquiteturas Heterogêneas, IP-*core*, *Wrapper*, *Virtual Bus*/HLA, Redução do Tempo de Projeto.

Abstract

This work deals with the functional verification of hardware designs based on heterogeneous architectures. One method is designed to reduce design time by excluding and adapting steps from the conventional design flow. These changes aim to use legacy IP-cores through the use of an integration interface of heterogeneous architectures. The testbench of the proposed functional verification method is based on the SystemC library and uses the concept of Virtual Bus from the High-Level Architecture (HLA) specification. Thus, the integration of an IP core requires the development of a communication wrapper with testbench. The proposed testbench was used to verify the design of a system composed of two subsystems, one that converts RGB to YC_bC_r (S_1) and another that calculates the integral of the image (S_2). Two verification scenarios were considered. In the first the two subsystems are connected in series, and in the second, they are connected in parallel. In the first scenario, the IP cores of S_1 and S_2 were made available in C++/OpenCL and SystemVerilog, respectively. In the second scenario, the IP cores of S_1 and S_2 were made available in OpenCL and SystemVerilog, respectively. In these two scenarios, the public domain IP-cores were used, and the respective external communication wrappers were implemented. The "golden models" of these IP colors were considered to be available. In the first scenario, the testbench was used to integrate the available implementations using Virtual Bus, being necessary to implement specific wrappers for C++/OpenCL and SystemVerilog in the scope of the testbench. In the second verification scenario, the testbench was used to integrate the available implementations using Virtual Bus, being necessary to implement specific wrappers for OpenCL and C++ with SystemVerilog, in the scope of the testbench. Tests are applied in usage scenarios with $10,000$ samples generated from an image. The serial scenario compares the output generated from S_1 to S_2 with a golden model. The parallel scenario compares the generated outputs of S_1 and S_2 with golden models. Therefore, the method reduced design time without recoding, adding steps to the design flow.

Keywords: Hardware Design, Functional Verification, Implementations, Heterogeneous Architectures, IP-core, Wrapper, Virtual Bus/HLA, Reduce Design Time.

Sumário

1 Introdução **1**

1.1 Motivação e contexto . . . 1

1.2 Definição do problema . . . 5

1.3 Relevância . . . 6

1.4 Problema abordado . . . 7

1.5 Hipótese . . . 7

1.6 Objetivo geral . . . 8

1.7 Objetivos específicos . . . 8

1.8 Organização do texto . . . 8

2 Terminologias e conceitos do fluxo de projetos e da verificação funcional **10**

2.1 Introdução . . . 11

2.2 Fluxo de desenvolvimento de um projeto de hardware . . . 12

2.3 Ciclo de Verificação . . . 13

2.4 Desafios na verificação . . . 15

2.5 Etapa de verificação funcional . . . 17

2.6 Metodologias de verificação funcional . . . 19

2.7 Hierarquia do *testbench* . . . 20

2.7.1 *Testbench* . . . 20

2.7.2 *Test* . . . 21

2.7.3 *environment* . . . 21

2.7.4 *Agent* . . . 21

2.7.5 *Driver* . . . 23

2.7.6 *Monitor* . . . 23
2.7.7 *Scoreboard* . . . 24
2.7.8 *Reference Model* . . . 24
2.7.9 *Checker* . . . 24
2.7.10 *Design Under Verification* . . . 24
2.8 Considerações finais . . . 25

3 Interoperabilidade na computação distribuída 26
3.1 Simulação Paralela e Simulação Distribuída . . . 26
3.1.1 Simulação distribuída entre arquiteturas heterogêneas . . . 27
3.2 High Level Architecture . . . 28
3.2.1 Template de modelo de objetos . . . 29
3.2.2 Regras do HLA . . . 31
3.2.3 Especificação da interface do HLA . . . 32
3.2.4 Estrutura Geral . . . 32
3.3 Considerações finais . . . 37

4 Trabalhos relacionados 38
4.1 Considerações finais . . . 41

5 Implementação da infraestrutura de interoperação 45
5.1 *Virtual Bus* . . . 45
5.2 Interoperabilidade para computação heterogênea distribuída . . . 48
5.3 Resultados experimentais . . . 54
5.3.1 Equipamentos . . . 55
5.3.2 Cenários . . . 55
5.3.3 Configuração e troca de dados . . . 56
5.3.4 Sender Federate . . . 58
5.3.5 Integração SoC com *Virtual Bus* . . . 60
5.3.6 Resultados . . . 64
5.4 Considerações finais . . . 70

6 Ambiente de verificação funcional distribuída e heterogênea 72

6.1 Etapas abordadas para verificação funcional distribuída 72

6.2 Ambiente de verificação funcional distribuído e heterogêneo proposto 76

6.3 Resultados experimentais 78

6.3.1 Equipamentos 79

6.3.2 Cenários 80

6.3.3 Estudos de caso 80

6.3.4 Resultados 105

6.4 Análise dos Resultados 115

6.5 Considerações finais 116

7 Conclusões 117

7.1 Contribuições da pesquisa 118

7.2 Limitações e sugestões para pesquisas futuras 119

Referências bibliográficas 120

Lista de abreviaturas, símbolos, siglas e acrônimos

Abreviaturas

IEEE	Electrical and Electronics Engineers	29
RTIA	RTI Ambassador	34
RTIG	RTI Gateway	34
SCV	SystemC Verification	5
SoC	System on a Chip	1

Símbolos

$*$	Convolução	81
$II(x, y)$	Fórmula para calcular a integral da imagem no ponto da imagem em x e y	81
i	Linha da operação de máscara de imagem	63
$i(x, y)$	Valor do *pixel* na posição (x, y)	81
$I)$	Matriz da imagem da função	81
$I(i, j)$	Vetor da máscara de imagem	63
j	Coluna da operação de máscara de imagem	63
M	Matriz da máscara de imagem	63
n	número de *pixels* enviados em variáveis	57
RGB	Cores: Red, Green e Blue	94
$S)$	Matriz da integral da função	81

xp Coordenada do *pixel* enviada em x 57

Y Sinal de luma armazenado com alta resolução 94

yp Coordenada do *pixel* enviada em y 57

(x, y) Região no ponto da imagem em x e y 81

C_b Componentes de crominância 94

C_r Componentes de crominância 94

Siglas

ABV Assertion-Based Verification 4

API Application Programming Interface 46

BVM Brazil-IP Verification Methodology 19

CPU Central Process Unit 2

DDS Data Distribution Service 37

DMSO Defence Modelling and Simulation Office 28

ESL Electronic System Level 6

FPGA Field Programmable Gate Array 2

GPU Graphics Processing Unit 2

HDL Hardware Description Language 2

HLA High Level Architecture 5

IP-*core* Intellectual Property Core 2

IVM Interoperable Verification Methodology 19

OMT Object Model Template 28

OVM Open Verification Methodology 19

PLI-I Programming Language Interface for Interoperation 39

RGB Red, Gree e Blue 94

RTI Run Time Infrastructure 28

RTL Register Transfer Level 6

TCP/IP Transmission Control Protocol/Internet Protocol 34

TLM Transaction Level Modeling 16

UPF Unified Power Format 19

URM Universal Reuse Methodology 18

UVM Universal Verification Metodology 5

VMM Verification Methodology Manual 18

YC_bC_r Luminance and Chrominance Components 94

Acrônimos

DUV Design Under Verification 13

FOM Federation Object Model 29

GALT Greatest Available Logical Time 36

LAN Local Area Network 27

ONERA Office National d'Etudes et de Recherches Aerospatiales 47

SISO Simulation Interoperability Standards Organization 29

SOM Simulation Object Model 29

WAN Wide Area Network 27

Lista de Tabelas

3.1 Grupo de Serviços oferecidos pela RTI.Grupo de Serviços oferecidos pela RTI. 35

4.1 Comparação entre diferentes trabalhos com o trabalho proposto. 42

5.1 Descrição das funções básicas do *Virtual Bus*. 47

5.2 Especificação do equipamento. 55

5.3 Cenários usados nos experimentos. 55

5.4 Tempos de transferência da imagem de Lena com 786.432 elementos. 65

5.5 Tempo de processamento e tempo total da imagem de Lena com 786.432 elementos. 66

6.1 Fluxo de dados entre os federados. 78

6.2 Especificações dos equipamentos. 79

6.3 Cenários usados nas experiências. 80

6.4 Comparação entre o ambiente de verificação funcional proposto e o tradicional.116

Lista de Figuras

1.1 IP-*core* para *plug and play* no projeto de SoC. 3

1.2 Ciclo de vida do IP-*core*. 3

2.1 O custo de verificação aumenta à medida que o nó de tecnologia diminui. . . 11

2.2 Etapas do fluxo de desenvolvimento de um IP-*core*. 12

2.3 Ciclo de vida da verificação. 14

2.4 Metodologia comum do ciclo de projeto de verificação 16

2.5 Ambiente de verificação funcional 20

2.6 Agente UVM 22

3.1 Simulação entre arquiteturas heterogêneas. (a) abordagem da implementação *ad hoc*. (b) implementação proposta com abordagem de interoperação. . . . 27

3.2 Estrutura genérica de elementos da arquitetura HLA. 33

3.3 Estrutura genérica do federado com *time-regulator* e um federado com *time-constrained*. 37

5.1 Adaptação do HLA como base para criação do *Virtual Bus*. 46

5.2 Etapas de implementação para criação do *Virtual Bus*. (a) Implementação de uma interface *Virtual Bus*. (b) Integração e interoperabilidade. 46

5.3 Componentes do pacote *Virtual Bus*. 47

5.4 Arquitetura Geral do *Virtual Bus*. 48

5.5 Fluxo de execução. (a) HLA. (b) *Virtual Bus*. 52

5.6 *Virtual Bus* com sistemas heterogêneos. 56

5.7 Estrutura das mensagens. 57

5.8 Estrutura para mensagens *one-by-one*. 57

5.9 Exemplo de mensagem *multi-pixel*, transportando cinco *pixels* de três canais. 58
5.10 A imagem de Lena usada nos experimentos. 65
5.11 Atividade *Sender Federate* durante a transição para o SoC *Federate*. 67
5.12 Atividade de FPGA (no SoC *federate*) durante a comunicação com o *Sender Federate*. 68
5.13 Atividade de transmissão entre *Sender* e *multi-core*. 68
5.14 Atividade de transmissão entre *Sender* e GPU. 69
5.15 Atividade de transmissão entre *Sender*, FPGA e *multi-core*. 70
5.16 Atividade de transmissão entre *Sender*, FPGA e GPU. 70

6.1 Etapas de validação da abordagem de verificação funcional distribuída proposta. (a) Especificação dos componentes de teste. (b) Desenvolvimento do modelo de teste e do modelo ideal. (c) Adaptação de um modelo de verificação funcional tradicional para um modelo distribuído. (d) Simulação e teste com o ambiente para verificação funcional distribuída e heterogêneo. 73
6.2 Etapas necessárias para adaptar a abordagem proposta. 74
6.3 Adaptação do ambiente de verificação do projeto para verificação funcional distribuída. 75
6.4 Ambiente de verificação funcional distribuída e heterogênea. 77
6.5 Fluxo de implementação para o ambiente de verificação. 88
6.6 Teste de integração dos *wrappers* na verificação funcional com os dispositivos propostos. 90
6.7 Fluxo do tempo de comunicação HLA. 92
6.8 Modelo para verificação funcional distribuída com *testbench*, FPGA e *reference model*. 93
6.9 Fluxo do tempo de comunicação HLA. 98
6.10 *testbench*, *reference model* e GPU integrados e sincronizando. 99
6.11 Teste de verificação funcional com arquitetura heterogênea GPU. 101
6.12 Quatro federados comunicando e sincronizando com o federado *testbench*. . . 102
6.13 Quatro federados interagindo e sincronizando no tempo. 103
6.14 Cinco federados comunicando e sincronizando com o federado *testbench*. . . . 104

6.15 Cinco federados interagindo e sincronizando no tempo. 105
6.16 Análise do sincronismo dos dados 106
6.17 Análise de detecção de erros fixos 107
6.18 Análise de detecção de erros aleatórios 108
6.19 Uso do processador pelo ambiente de verificação com *Virtual Bus*. 109
6.20 Comparação entre o DUV e o *reference model*. 109
6.21 Uso do processador ARM e PC (porcentagem) por segundos. 110
6.22 Tráfego de rede por segundo para o ARM e o PC. 111
6.23 Uso de processamento da *Jetson* TX1. 112
6.24 Análise do tempo de leitura, escrita e processamento com 4 federados 113
6.25 Análise do tempo de leitura, escrita e processamento com 5 federados 114
6.26 Comparação to tempo de leitura, escrita e processamento dos cenários C_1 e C_2.115

Capítulo 1

Introdução

Este trabalho aborda uma contribuição para a área de circuitos integrados digitais. O foco do trabalho consiste em modificar etapas do fluxo de projeto para reduzir o tempo de desenvolvimento de projetos de sistemas complexos na etapa verificação funcional, pois é nessa etapa que apresenta o maior custo do projeto. Dessa forma, o trabalho aqui descrito trata de um método que permite realizar a etapa de verificação funcional do projeto sem a necessidade de re-codificação implementados em diferentes dispositivos heterogêneos, de forma que passe pela verificação considerando cruciais algumas etapas para essa finalidade.

Organizou-se a apresentação deste capítulo da seguinte forma; na Seção 1.1 introduz-se a motivação deste trabalho de doutorado; na Seção 1.2 trata-se da definição do problema e seus desafios; na Seção 1.3 descreve-se a relevância deste trabalho de doutorado; na Seção 1.4 aborda-se o problema que deve ser tratado; na Seção 1.5 apresenta-se a proposição para a hipótese; na Seção 1.6 comenta-se o objetivo proposto a ser desenvolvido; na Seção 1.7 abordam-se os objetivos específicos; na Seção 1.8 trata-se uma visão geral de como se encontra estruturado o restante do documento.

1.1 Motivação e contexto

O avanço das técnicas de integração em silício demandam em produtos cada vez mais complexos e variados que influenciam na mudança diretamente no desenvolvimento dos projetos de sistema em um *chip* (SoC, do inglês *System on a Chip*), em que todo sistema

embarcado possa ser integrado e implementado por apenas um *chip* [1].

Um SoC é composto por vários componentes sendo alguns: memória, processador de uso geral (CPU, do inglês *Central Process Unit*), unidade de processamento gráfico (GPU, do inglês *Graphics Processing Unit*), arranjo de portas programáveis em campo (FPGA, do inglês *Field Programmable Gate Array*), circuito de aplicação específica, entre outros. O desenvolvimento desse circuito eletrônico exige um determinado tempo, no qual os projetistas enfrentam o desafio de reduzir ao máximo esse tempo para introduzir o produto no mercado [2]. A velocidade com que novos SoCs são introduzidos no mercado não acompanham a complexidade de integrar os componentes no circuito. Com a proposta de integração de componentes mais complexos, os projetos têm uma abordagem baseada no reuso de componentes pré-existentes e pré-verificados. Esses componentes são conhecidos como propriedade intelectual (IP-*core*, do inglês *Intellectual Property Core*) [1].

Os IP-*cores* são componentes reutilizáveis, geralmente, sintetizados e descritos em uma linguagem de descrição de *hardware* (HDL, do inglês *Hardware Description Language*) ou podem iniciar em um nível mais alto de abstração. Nesse sentido, quando se deseja iniciar em um nível mais alto de abstração, são reutilizados *open core*[1] que foram implementados e estão funcionando isoladamente em GPU, CPU, e/ou FPGA. Essas implementações são partes de um SoC, que podem ser compostas por aplicações em *hardware*, *software* e partes reutilizadas de IP-*cores* digitais e analógicos [3].

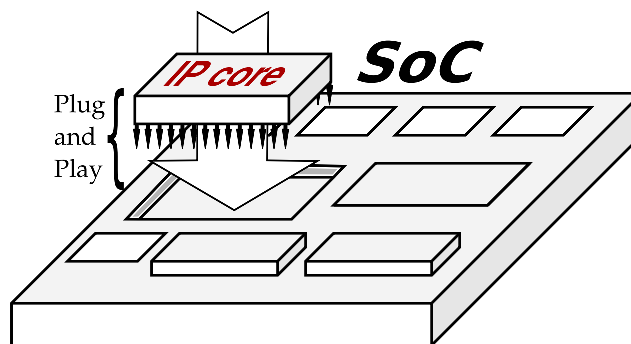
Os IP-*cores* se enquadram em três categorias: *hard cores*, não permitem qualquer tipo de modificação ou personalização, sendo otimizados para uma tecnologia específica; *firm cores*, a combinação do código-fonte e sua lista de conexões (do inglês *netlist*) que pode variar conforme a tecnologia utilizada, além de serem configuráveis para várias aplicações, com a dificuldade em seu uso devido aos componentes serem desenvolvidos por diferentes fabricantes; ou *soft cores*, descritos em linguagens de descrição de *hardware*, sendo um *open core* que oferece flexibilidade e independência de tecnologia. Este trabalho será utilizado a nomenclatura *open source* IP-*core* para representa os IP-*cores* iniciando em um nível de abstração mais alto.

A construção de um IP-*core* passa por diversas etapas como, por exemplo, simulação,

[1]Um *open cores* é uma implementação de código-fonte aberto que pode iniciar em um nível de abstração alto ou baixo, sendo de domínio público e revisados pela comunidade.

verificação funcional, síntese, prototipação, proteção de propriedade intelectual, especificação funcional e implementação [4]. Contudo, existe a necessidade de integrar todas as partes e que funcionem com seus determinados sincronismos. Isso demanda tempo e vários testes de compatibilidade. Em Seok, Kim e Park[5], os autores propõem um mecanismo de comunicação entre partes diferentes de um protótipo para funcionarem em conjunto.

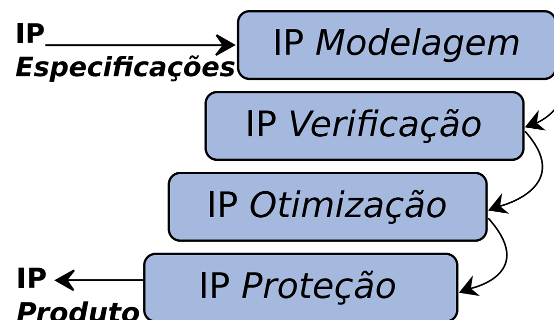
Figura 1.1: IP-*core* para *plug and play* no projeto de SoC.



Fonte: adaptada de Mohamed[3].

A indústria reutiliza os IP-*cores* por *plug and play*, como ilustrado na Figura 1.1. O processo do ciclo de vida dos IP-*cores* (ver Figura 1.2), inicia na especificação passando por diferentes etapas para desenvolver o produto, são realizadas em quatro etapas principais: IP modelagem, IP verificação, IP otimização e IP proteção [3].

Figura 1.2: Ciclo de vida do IP-*core*.



Fonte: adaptada de Mohamed[3].

Na etapa de verificação do IP-*core* pode ser realizada por: verificação funcional ou verificação formal. De acordo com Mohamed[3], a verificação funcional de IP-*cores* é a chave para reduzir o custo de desenvolvimento e o tempo de inserir o produto no mercado (do inglês *time-to-market*). Para tal, o tempo de simulação é uma questão relevante para sistemas

complexos, visto que, em alguns casos, uma simulação lenta demanda de tempo para cobrir um número suficiente de casos de teste na etapa de verificação. Enquanto, a proposta de utilizar a verificação formal é provar que as funções do projeto estão implementadas corretamente, em vez de simular e aplicar vetores de teste, analisando o comportamento possível do IP-*core* para detectar quaisquer estados de erro alcançável usando metodologia de verificação baseada em asserção (ABV, do inglês *Assertion-Based Verification*) [3].

Na verificação funcional aborda-se um ciclo com maior investimento e tempo de projeto. Isso se deve ao conjunto de tarefas destinadas a descobrir erros gerados durante o desenvolvimento do projeto. Como soluções, são utilizadas metodologias que possam integrar e validar um conjunto de componentes que trabalham em conjunto para realizar os testes [6]. Esses componentes podem cooperar entre si e com um ambiente de simulação com a intenção de verificar implementações antes mesmo de obter o modelo sintetizável, reduzindo o tempo total do projeto e a possibilidade de inserir novos erros na escolha de componentes ainda não testados.

Em sistemas cada vez mais complexos, a etapa de verificação é um recurso essencial. Diante que em alguns projetos se tornam insuficientes quando desenvolvidos com apenas uma linguagem ou até mesmo com um único nível de abstração, pois os sistemas computacionais quase sempre são compostos por módulos desenvolvidos em *hardware*, *software*, e ainda podendo conter outras partes mecânicas e módulos analógicos. Essa combinação de componentes de diferentes domínios que precisam cooperar são denominados sistemas heterogêneos. Para realizar a verificação desse tipo de sistema é comum validar cada módulo desenvolvido separadamente [7].

Neste trabalho objetiva-se adaptar etapas no fluxo de projeto de *hardware* que cooperem com a etapa de verificação funcional, na perspectiva de reduzir o tempo de desenvolvimento de projetos de circuitos integrados. Essas etapas adicionadas propõem um método com base na verificação funcional para validar e verificar a combinação de diferentes *open source* IP-*cores* em diferentes dispositivos, que serão verificados ao serem integrados mesmo em diversos níveis de abstrações, linguagens e arquiteturas heterogêneas. Contudo, alguns desafios desses *open source* IP-*cores* devem ser considerados no processo de verificação funcional, como portabilidade e reusabilidade [1]. Com base nesses desafios mencionados, este trabalho realiza a intercomunicação dos dispositivos heterogêneos com um ambiente de simulação

(do inglês *testbench*), em que o ambiente conecta o vetor de teste com o componente em desenvolvimento realizando a verificação. Essa integração é realizada por uma interface de comunicação denominada barramento virtual (do inglês *Virtual Bus*) que foi desenvolvida com base na especificação da arquitetura de alto nível (HLA, do inglês *High Level Architecture*). Silva et al.[8] desenvolveram *Virtual Bus* como uma interface de comunicação entre as arquiteturas heterogêneas, realizando a sincronização durante a simulação. Além disso, deve-se implementar uma versão do ambiente de simulação que se comunique com os sistemas heterogêneas para proporcionar o reuso e a interoperabilidade. Esse ambiente é implementado nas bibliotecas de verificação *SystemC* (SCV, do inglês *SystemC Verification*) [9] e na metodologia de verificação universal (UVM, do inglês *Universal Verification Metodology*). Assim, o método é proposto para realizar verificação funcional ao integrar o *testbench* com os componentes em teste e/ou já testados, para isso se deve implementar o *Virtual Bus*.

Alguns trabalhos tratam somente da verificação funcional sem envolver a validação das arquiteturas heterogêneas e sem englobar os aspectos necessários para atenderem as especificações do projeto. Diversas pesquisas, Seok et al.[10], Brito et al.[11], Tran et al.[12], Hekmatpour et al.[13], são dedicadas as técnicas para simular e emular, permitindo coordenar o experimento de forma distribuída. Muhr, Holler e Horauer[14], Barnasconi et al.[15] vêm realizando pesquisas com verificação funcional e simulação de forma local.

1.2 Definição do problema

Conforme o trabalho de Chen et al.[6], 57% do tempo total do projeto ocorre na fase de verificação funcional, que requer o maior esforço no projeto. Associado aos esforços para realizar a verificação funcional, o trabalho de Duenas[16] trata que 65% dos IP-*cores* falham em sua primeira prototipação em silício e 70% destes casos são devidos a problemas na verificação funcional.

O reuso de IP-*cores*, em outros projetos, pode apresentar problemas quanto a compatibilidade. Geralmente, as empresas compram IP-*cores* prontos para serem integrados nos projetos de SoC. No reuso de IP-*cores* disponíveis e em alto nível de abstração é necessário a verificação, no entanto, várias partes podem estar com linguagens que ainda não foram

re-codificadas é um grande desafio. Outro desafio, surge devido à confiabilidade para incorporar partes de outros projetos que não se conheçam [17]. Nos casos de implementações disponíveis, funcionando isoladamente em GPU, ou CPU, ou FPGA, pode não existir nem modelo ao nível de sistema eletrônico (ESL, do inglês *Electronic System Level*) nem modelo ao nível de transferência de registro (RTL, do inglês *Register Transfer Level*). Quando não existem tais modelos, a criação desses modelos para fins de verificação levaria um certo tempo e abre brecha para erros nesses modelos [6].

Outro desafio, seria integrar ao processo de verificação funcional diferentes placas de prototipação tais como FPGA, GPU ou CPU, com o intuito de verificar o funcionamento de diferentes partes em conjunto, eliminando a necessidade da criação de novos modelos dessas implementações [5].

Um grande desafio surge quando se decide adaptar novas etapas ao fluxo de desenvolvimento de *hardware*, permitindo combinar o *testbench* aos sistemas heterogêneas com *open source* IP-*cores* no processo de verificação funcional. Para isso ser realizado, a implementação a ser testada em um dispositivo precisa apresentar uma interface de fácil integração ao *testbench*.

1.3 Relevância

A necessidade de inclui novos produtos ao mercado não segue o avanço da capacidade de integração de sistemas complexos em *chip* [1]. Dessa forma, um método na etapa de verificação funcional permitindo a integração de *open source* IP-*cores* é inicialmente desconhecido, verifica-se a necessidade de adaptar durante a simulação do ambiente de verificação funcional com os dispositivos heterogêneos um *middleware* para prover a interoperabilidade, reuso e sincronização, isso com o propósito de verificar partes separadas em conjunto, permitindo reduzir o tempo de projeto sem a necessita de re-codificação.

Segundo Seok et al.[10], é empregado um método baseado na abordagem de co-simulação, para isso é utilizado a especificação do HLA para combinar IP-*cores* heterogêneos, com objetivo de validar um método de otimização que possibilite uma prototipagem rápida. Na proposta do trabalho de [1] é tratada uma abordagem para reuso de IP-*cores*, em um processo de desenvolvimento denominado ipPROCESS, visando facilitar e conduzir o projeto de IP-

core de alta qualidade, assim, promovendo a identificação de erros durante as primeiras fases do projeto. Neste trabalho é tratado um método de verificação funcional que intercomunica *open source* IP-*cores* em sistemas heterogêneos, de modo a serem utilizados na etapa de verificação, promovendo heterogeneidade dos IP-*cores* combinados [2].

1.4 Problema abordado

Na definição do problema tratado nessa tese, consideram-se as seguintes premissas básicas: *a) A verificação funcional é a etapa que consome o maior esforço e o tempo no fluxo de projeto convencional; b) Os ambientes de verificação funcional requerem o modelo RTL do projeto do sistema; c) Re-codificação de modelos ESL para modelos RTL demanda tempo e enseja a possibilidade de erros de codificação; d) Os ambientes de verificação funcional, geralmente, não são projetados para integrar IP-cores que não sejam disponibilizados em RTL e não sejam executáveis na infraestrutura computacional do próprio testbench.*

Nesse contexto, o problema tratado nessa tese é: "*Como realizar a verificação funcional de projetos baseados em sistemas heterogêneos sem a necessidade de re-codificação e integrando IP-cores que não sejam disponibilizados em RTL e não sejam executáveis na infraestrutura computacional do próprio testbench?*"

1.5 Hipótese

Conforme o problema exposto, consideram-se os desafios devido à heterogeneidade dos *open source* IP-*cores* combinados [5]. Para integrar implementações em *hardware* com o ambiente de verificação funcional, é necessário a aplicação de técnicas para as quais não há o conhecimento a-priori da integração dos módulos. Dessa forma, para o problema abordado neste trabalho é considerada a hipótese de que: "*O método de verificação funcional integrando módulos com implementações permite reduzir o tempo de projetos de SoCs digitais que empregam open source* IP-*cores*".

1.6 Objetivo geral

O objetivo deste trabalho é propor um método de verificação funcional para projetos de sistemas digitais baseados em arquiteturas heterogêneas, mediante o uso do qual contribuirá para redução do tempo de projetos de sistemas heterogêneos, sem a necessidade de recodificação de *open source* IP-*cores*.

1.7 Objetivos específicos

Os objetivos específicos deste trabalho são os seguintes:

- Adaptar etapas no fluxo de projeto que permita integrar *open source* IP-*cores* ao ambiente de verificação funcional;
- Desenvolver uma interface que permita que módulos desenvolvidos em *hardware* possam ser integrados por interoperação ou reuso no mesmo ambiente de verificação funcional.

1.8 Organização do texto

Esse documento está organizado da seguinte forma:

No Capítulo 2 é apresentada a base conceitual para desenvolver um projeto de verificação. É necessário seguir um fluxo de desenvolvimento que contém algumas etapas de gerenciamento das atividades. Assim como, são descritos as terminologias e conceitos adotados pela verificação funcional convencional.

No Capítulo 3 introduzem-se os principais conceitos da arquitetura de alto nível, que será a base para implementação da interface de comunicação durante a simulação das arquiteturas heterogêneas.

No capítulo 4 contextualiza-se uma comparação dos trabalhos relacionados com o trabalho aqui proposto, tratando as técnicas, os padrões de intercomunicações, os prós e os contras.

No capítulo 5 apresentam-se os principais conceitos da abordagem proposta para integrar diferentes arquiteturas heterogêneas durante a verificação funcional.

No capítulo 6 tratam-se os passos necessários para a verificação funcional distribuída, assim como experimentos e resultados para validar as arquiteturas heterogêneas.

No Capítulo 7 abordam-se as considerações finais sobre a pesquisa e sugestões para pesquisas futuras.

Capítulo 2

Terminologias e conceitos do fluxo de projetos e da verificação funcional

Neste capítulo tratam-se das etapas que precedem e sucedem à etapa de verificação funcional durante o fluxo de desenvolvimento de *hardware*. Assim, apontando os principais desafios e a complexidade do fluxo de projeto de circuitos digitais. O entendimento dos desafios propiciam o entendimento de quanto tempo um projeto de SoC pode levar para ser desenvolvido. Além disso, tratam-se os conceitos e as nomenclaturas de verificação funcional. Neste capítulo abordam-se os conceitos e as nomenclaturas de verificação funcional em que se baseia este trabalho.

Organizou-se a apresentação deste capítulo da seguinte forma: na Seção 2.1 introduz-se sobre o projeto de circuito integrado; na Seção 2.2 apresentam-se as etapas de gerenciamento com suas atividades do fluxo de desenvolvimento de projeto de *hardware*; na Seção 2.3 conceituam-se as etapas do ciclo de verificação; na Seção 2.4 descrevem-se os desafios do projeto de verificação; na Seção 2.5 apresenta-se uma visão introdutória da etapa de verificação funcional; na Seção 2.6 tratam-se algumas metodologias utilizadas para verificação funcional; na Seção 2.7 apresentam-se as terminologias utilizada na metodologia de verificação funcional e que são comumente abordadas em trabalhos de outros pesquisadores da área; na Seção 2.8 abordam-se as considerações finais deste capítulo.

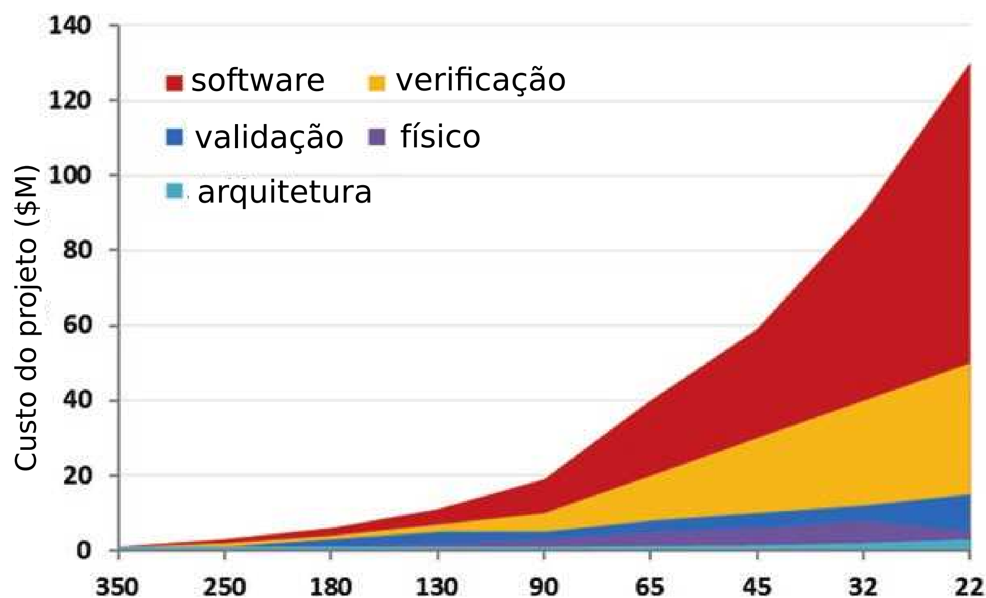
2.1 Introdução

Durante o projeto de circuito integrado, a intensidade do desenvolvimento recai sobre a verificação funcional para garantir a comparação entre modelos e, assim, no domínio funcional corresponda com a especificação. A verificação funcional tem um ciclo importante desde a verificação do RTL até depois de gerar o produto na etapa de validação pós-silício [2].

Segundo Mehta[2], dentre os vários desafios que as empresas enfrentam, o maior é a velocidade para fornecer silício funcional. Com base em estudos realizados e no aperfeiçoamento do projeto na etapa de verificação funcional, relata-se que a maior parte do recurso do projeto é nessa etapa sendo em torno de 40% e 50%. Assim como, verifica-se que mais de 50% dos projetos requerem reformulação devido a erros funcionais.

O gráfico da Figura 2.1 ilustra o custo do projeto para diversas partes de um ciclo de projeto. Nesse gráfico, o custo de verificação do projeto tem percentual de 40%, em que se refere a uma parte considerável do projeto. Em outras palavras, esse problema afirma que amplia o rendimento da verificação funcional do projeto e reduzir outras etapas de modo a aumentar produtividade na etapa de verificação [2].

Figura 2.1: O custo de verificação aumenta à medida que o nó de tecnologia diminui.



Fonte: adaptado de Mehta[2]

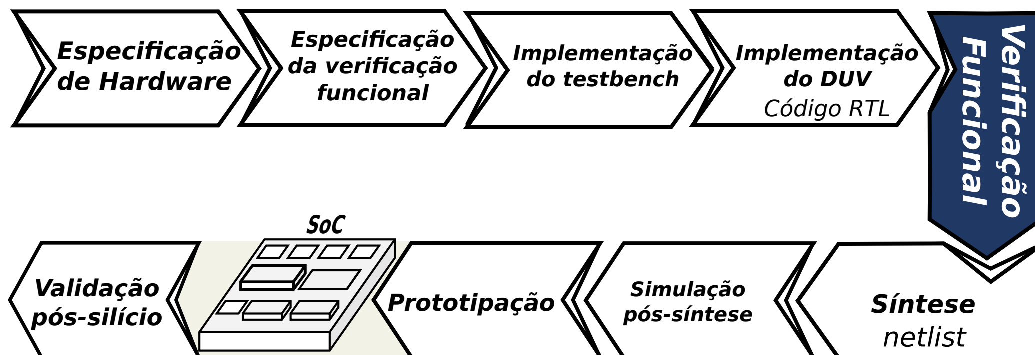
Devido a erros (do inglês *bugs*) funcionais as grandes empresas investem na etapa de

verificação funcional, assim evitando comprometer o tempo determinado do projeto a ser lançado no mercado. Neste capítulo, será apresentado metodologias que lidam com os desafios de verificação funcional de modo a apontar as falhas em suas etapas.

2.2 Fluxo de desenvolvimento de um projeto de hardware

Em pesquisas realizadas na área de circuito integrados, não existe um modelo padrão com as etapas para a construção de um IP-*core* digital ou analógico. Assim como, não existem elementos conceituais e a delimitação de cada fase. Os trabalhos são breves quanto a explicação de cada etapa do processo de desenvolvimento de um *hardware* [2] [3] [4].

Figura 2.2: Etapas do fluxo de desenvolvimento de um IP-*core*.



Fonte: adaptada de Silva[4].

O fluxo convencional de desenvolvimento de um *hardware* é ilustrado na Figura 2.2, contém as etapas de gerenciamento de atividades, conforme resumido a seguir:

- Especificação do *hardware*: deve ter um alto nível de abstração. Ela contém as funcionalidades a serem executadas, as informações de baixo nível, as especificações e as descrições dos pinos a serem usados.

- Especificação da verificação funcional: é nesta etapa que são documentados os aspectos importantes que na etapa de verificação funcional devem ser verificados em um determinado dispositivo. A documentação deve ser em formato de texto definida pelo engenheiro de verificação e por todos os participantes do projeto.

- Implementação *testbench*: nesta etapa desenvolve-se o ambiente de simulação, o responsável pela verificação dos componentes de *hardware* que são utilizados em testes, chamado de verificação sob o projeto (DUV, do inglês *Design Under Verification*).

- Implementação do DUV: é um código escrito em um nível baixo de abstração. O qual as operações são controladas pelo *clock*. O código RTL é implementado em linguagem HDL.

- Verificação funcional: é realizada através da comparação de dois modelos, o modelo desenvolvido e o modelo ideal que reflete a especificação, ocorrendo por simulação.

- Síntese: é realizada através da conversão de uma descrição RTL em um conjunto de registradores e em um conjunto de lógica combinacional. Assim que o código RTL é sintetizado gera-se uma *netlist* ao nível de portas lógicas.

- Simulação pós-síntese: é essencial para determinar se os requisitos de tempo são respeitados e se pode ser alcançado um desempenho aceitável pelo dispositivo, usando circuitos mais otimizados.

- Prototipação: é a fabricação de uma pequena quantidade de protótipos em silício.

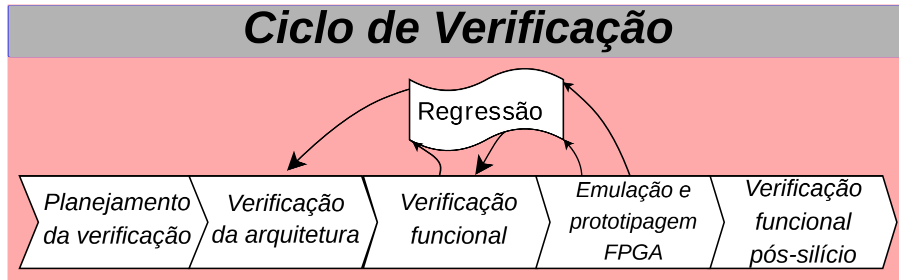
- Validação pós-silício: tem-se um artefato já elaborado e real em vez de um modelo RTL. Pode ser indicado a um laboratório específico para depuração, passado por vários testes para averiguar alguns parâmetros, como, funcionalidade, tempo, potência, desempenho, características elétricas, entre outros.

2.3 Ciclo de Verificação

Dada a complexidade dos dispositivos, o fluxo de verificação de IP-*core* é complexo e exige um planejamento inicial e abrangente, levando quase todo o ciclo de vida do projeto. No desenvolvimento do projeto, o ciclo de verificação pode ser abordado como uma estratégia potencial para reduzir o tempo de projeto. Para isso, é analisado as dificuldades embarcadas na computação dos dispositivos durante o ciclo de verificação, conforme mostrado na Figura 2.3. De certa forma, o fluxo captura a essência básica das etapas do ciclo de verificação [6].

Essas etapas tratam tanto o desenvolvimento, simulação e emulação, assim como detecção e correção de erros que venha a ser testados nas diferentes etapas do projeto.

Figura 2.3: Ciclo de vida da verificação.



Fonte: adaptada de [6].

A Figura 2.3 ilustra o ciclo de vida da verificação que consiste em etapas importantes na etapa durante a verificação funcional, baseada em [6], descritas adiante:

- Planejamento da verificação: inicia quase ao mesmo tempo que o planejamento da especificação do produto e continua durante a etapa de desenvolvimento do sistema. Planejar o produto requer definição, decomposição, interfaces de conexão e comunicação, consumo de energia, desempenho, segurança dos IP-*cores*. Enquanto, o planejamento da verificação inclui a criação de planos de teste apropriados para os IP-*cores* [6].

- Verificação da arquitetura: há duas atividades relevantes: a primeira é uma das mais importantes, sendo ela a verificação funcional dos vários protocolos de comunicação; a segunda é a parte primordial e o início do desenvolvimento dos modelos de prototipagem para dispositivos de *hardware*, suprimindo as necessidades que segue nas outras etapas que envolve a verificação de *software* e *firmware* [6].

- Verificação funcional: os protótipos de SoCs são formados por IP-*cores*, sejam eles reaproveitados de terceiro ou criados internamente, para assegurar que a integração ocorra como planejado, é estabelecido uma equipe responsável por realizar uma verificação de cada IP-*core* que está sendo integrado [6]. Nessa etapa é uma das principais atividades presente no ciclo de verificação, pois envolve os recursos necessários na construção e implementação do *hardware*. De maneira geral, os protótipos de SoCs

industriais são reutilizados de terceiros. Uma equipe de verificação é responsável por realizar uma verificação de cada IP-*core* confiado, garantindo que funcionem conforme foi planejado [6]. Essa equipe envolvida na verificação, são capazes de integrar os IP-*cores* em um modelo de SoC durante sua construção e execução da verificação ao nível de sistema, a verificação tem como intuito assegurar que estejam funcionando corretamente juntos, como um sistema integrado. A grande parte da verificação e da integração dos IPs está inserida ao nível de sistema [6].

- Emulação e prototipagem FPGA: são inseridos os modelos RTL do *hardware* para uma arquitetura reconfigurável, a FPGA, ou aceleradores e emuladores que são especializados [18] [6]. A utilização dessas arquiteturas deve-se pelo fato de rodarem bem mais rápido quando comparado a um simulador RTL [6].

- Verificação funcional pós-silício: faz parte da validação pós-silício e tipicamente usa cenários de teste reais que teriam demorado tempo demais para serem simulados numa etapa pré-silício.

- Regressão: propõe detectar a reintrodução de erros que levem o projeto a retroceder a um estado menos avançado do projeto. Isso consiste em continuar reexecutando os testes definidos no plano de verificação [19] [20].

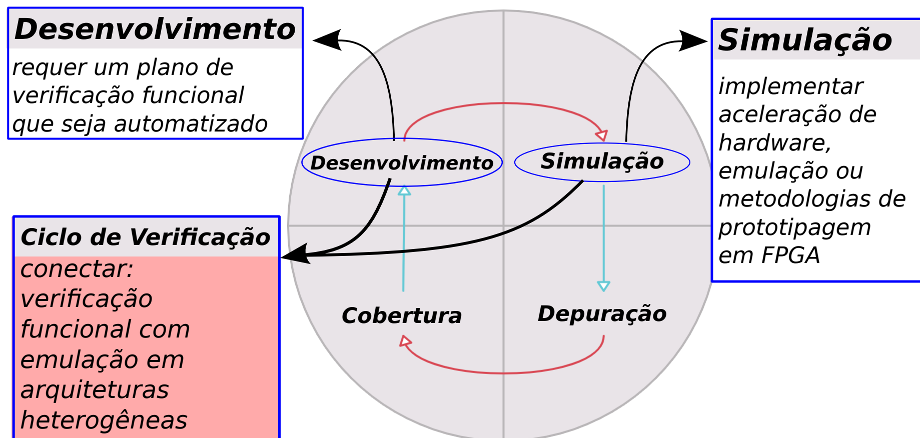
No ciclo de verificação foi descrito as diferentes funções de cada etapa. Com o intuito de avaliar e realizar uma execução de diferentes etapas em conjunto para verificar e testar arquiteturas heterogêneas que venham a compor um SoC.

2.4 Desafios na verificação

Os desafios do projeto de verificação funcional torna-se uma grande responsabilidade que requer algumas soluções específicas. Esses desafios surgem em algumas etapas destacadas no ciclo de verificação, que podem ser solucionados a partir da análise detalhada em cada etapa [2]. No projeto de verificação, conforme ilustrado na Figura 2.4, as etapas de desenvolvimento e de simulação propõem desafios que são tratados durante o ciclo de verificação. Dessa forma, é necessário compreender os desafios de cada etapa do projeto de verificação.

Durante o ciclo existem os desafios com a abordagem do reuso de componentes, que envolve vários níveis de abstração. Isso é uma maneira encontrada para reduzir ainda mais o tempo de projeto [21]. No entanto, o projeto tem a dificuldade de como verificar e integrar IP-*cores* independentes em desenvolvimento ou já desenvolvidos, para validar sua integridade.

Figura 2.4: Metodologia comum do ciclo de projeto de verificação



Fonte: adaptada de Mehta[2].

A Figura 2.4 ilustra a metodologia comum do ciclo de verificação de um projeto de SoC que consiste em quatro fases, baseada em [2], descritas adiante:

- Reduzir o tempo de desenvolvimento: que requer um plano de verificação funcional que seja elaborado de forma automatizada, para que não haja preocupação com a perda de tempo durante outras etapas. Nessa etapa do ciclo o ambiente de verificação deve ser criado de forma cuidadosa para que o andamento dos testes sejam obtidos com mais eficácia.

- Reduzir o tempo de simulação: que o tempo de simulação dos modelos de alto nível é bem menor que o teste RTL, modelado no nível de sinal. Assim, como, utilizar o *test-bench* de nível de transação. Como utilizar modelagem de nível de transação (TLM, do inglês *Transaction Level Modeling*) para reduzir o tempo de desenvolvimento, depuração e simulação. Além disso, pode-se implementar aceleração de *hardware* bem planejada, emulação ou metodologias de protótipo FPGA. Outro meio de reduzir o tempo de simulação, seria Desenvolver *testbench* de nível de transação que interage diretamente com o projeto acelerado ou emulado.

- Reduzir o tempo de depuração: utilizando metodologias de verificação baseada em *SystemVerilog* para encontrar rapidamente o erro. Essas metodologias no nível de transação reduzem o esforço de depuração. Assim como, utilizar verificação aleatória, pois isso permite reduzir o número de testes necessários.

- Reduzir o tempo de cobertura: para atender a esse quesito se pode utilizar *SystemVerilog* para cobertura funcional; especificar o domínio de seu projeto; utilizar cobertura de código para garantir uma cobertura estrutural.

O escopo deste trabalho não é voltado especificamente para o desenvolvimento das etapas em um projeto de *hardware*. Dentre as etapas de desenvolvimento de um *hardware*, é explorado a etapa de verificação funcional, assim como o fluxo do ciclo de verificação. Nesse sentido, a simulação distribuída surge como uma grande contribuição, tanto para a aceleração de desempenho, quanto para a co-simulação. Então se faz necessário prover algum mecanismo de integração rápido entre os IP-*cores* que já possuem implementações em arquiteturas diversas.

2.5 Etapa de verificação funcional

Em Bergeron[22], o conceito de verificação funcional é entendido como um processo que possibilita preservar em sua implementação o objetivo do projeto. Durante o processo de teste dos componentes, o projeto tem foco em cada fase, monitorando o andamento do projeto. Esse conceito é bastante analisado em cada etapa do *hardware*, permitindo analisar se um modelo em teste atende as especificações de um modelo ideal. Conforme, um grupo de entrada conhecido como vetores de testes, são os estímulos com a intensão de avaliar o modelo em simulação.

Para realizar o processo de verificação funcional, deve-se desenvolver um ambiente de propósito específico (*testbench*), essencial a simulação do modelo em teste. O problema de verificação funcional varia de sua causa, que pode ocorrer por: erro de implementação do *testbench*; ou algum erro na etapa de verificação do projeto. No caso do erro de fase, pode não ser visto nas etapas de prototipação, ou seja, o erro só é descoberto depois que o silício estiver integrado no sistema [23].

Com relação ao componente a ser verificado, a terminologia adotada neste trabalho é DUV, que pode ser implementado em diferentes linguagens com diferentes níveis de abstração. A metodologia aqui apresentada pode ser usada pelos demais níveis de abstração, da mesma forma.

Com a necessidade de criar o *testbench* que seja convencional para indústria, surge a UVM compartilhando sua utilidade com vários usuários. Na indústria era predominante três metodologias, sendo elas: manual de metodologia de verificação (VMM, do inglês *Verification Methodology Manual*) da Synopsys, OVM da empresa Mentor e metodologia de reutilização universal (URM, do inglês *Universal Reuse Methodology*) da Cadence [2].

Cada uma dessas metodologias foram capazes de oferecer uma biblioteca de classes básicas e recursos de TLM, permitindo implementar código reutilizável. De forma geral, acabou que no final houve uma confusão, com relação a qual metodologia os clientes poderiam adotar. A dúvida acabou depois do ingresso da UVM, permitindo que os clientes/usuários se sentissem mais confortáveis referente a essa metodologia e não ficassem submetidos a um só fornecedor. Atualmente, todos os fornecedores fornecem suporte a essa metodologia [2].

A UVM tornou-se um padrão para a verificação de projetos de circuito integrado, sua lógica de conexão das classes simplifica a implementação do *testbench* [24]. Desse modo, a configuração dos códigos se torna fácil e reutilizável. Com a implementação do ambiente de verificação funcional, ele ser reutilizado para diferentes componentes de teste realizando poucas modificações [2].

Vários utilitários genéricos são fornecidos pela biblioteca de classes UVM, que permitem aos usuários criarem diferentes estruturas para *testbench*, como banco de dados de configuração, modelo de biblioteca TLM, hierarquia de componentes, entre outros. Além disso, o UVM disponibiliza inúmeras interfaces e canais de comunicação no nível da transação, isso pode ser utilizado para conectar componentes no nível da transação. Ainda, é fundamental destacar que o uso de interfaces TLM isola os componentes individualmente, das alterações em outros componentes do ambiente completo [2].

O TLM proporciona a reutilização ao permitir que outro componente seja substituído, desde que possua a mesma interface. Isso é possível quando o TLM é acoplado à infraestrutura em fases de construção flexível no UVM. Por fim, o TLM disponibiliza o necessário para encapsular, simplificadamente em componentes reutilizáveis, também nomeados de compo-

nentes de verificação, para potencializar o reaproveitamento e diminuir o tempo e esforço obrigatório para montar um ambiente de verificação funcional [2].

2.6 Metodologias de verificação funcional

O maior desafio que as empresas enfrentam é o curto tempo para inserir um produto funcional no mercado, isso devido à complexidade na produção e na verificação de componentes de *hardware* cada vez menores. Com a importância dada para a verificação funcional, surgiram metodologias para facilitar a melhoria contínua com alta qualidade, dentre essas metodologias se destacam a UVM e a formato de energia unificado (UPF, do inglês *Unified Power Format*) para baixo consumo de energia (do inglês, *low power*). Como afirmado em Mehta[2], essas duas metodologias se tornam agora os pilares de quase todas as metodologias de projetos de verificação funcional [2].

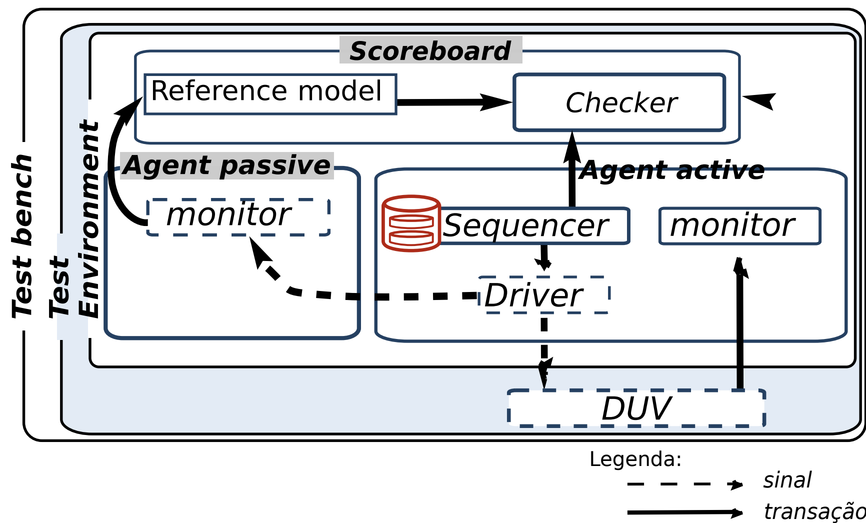
A partir da importância da UVM foram criadas metodologias com intuito de otimizar o esforço empreendido, fornecendo uma cobertura de testes mais ampla. Com a finalidade de conferir mais agilidade ao desenvolvimento do ambiente de verificação funcional. Quatro metodologias de verificação funcional são pesquisadas na literatura: VeriSC, metodologia de verificação aberta (OVM, do inglês *Open Verification Methodology*), metodologia de verificação interoperável (IVM, do inglês *Interoperable Verification Methodology*) e metodologia de verificação Brazil-IP (BVM, do inglês *Brazil-IP Verification Methodology*) [25].

A metodologia VeriSC [4] criada para o projeto de circuitos digitais síncronos que é caracterizada por um fluxo de atividades em que todo o modelo de referência (do inglês *reference model*) do circuito é construído antes do início da construção do ambiente de verificação funcional com abordagem de projeto de cima para baixo (do inglês *top-down*). Diferente da metodologia OVM [26], que se caracteriza por um fluxo de desenvolvimento de projeto, em que o circuito é implementado por partes com abordagem de baixo para cima (do inglês *bottom-up*). A metodologia IVM [27] destaca-se por sua arquitetura de *testbench* concebida com vistas às virtudes e defeitos das metodologias VeriSC e principalmente a metodologia OVM, buscando a obtenção de uma metodologia melhorada. Por fim, a metodologia BVM [28] é utilizada no Brazil-IP, que utiliza biblioteca de OVM como base para o desenvolvimento do ambiente de verificação e foi baseada na metodologia VeriSC.

2.7 Hierarquia do *testbench*

Na Figura 2.5 apresenta a hierarquia simples UVM. A hierarquia é composta pelos seguintes componentes: *testbench*, *test*, *environment* e *agents* (*active* e *passive*). O *testbench* é componente de nível superior que contém um ou mais *environment*, sendo que cada um inclui um *agent* com uma interface de comunicação para o DUV. Cada *agent* é composto por um *sequencer*, um *driver* e/ou um *monitor*, além disso, ainda pode existir no *environment* um componente *scoreboard*. O *scoreboard* é composto por um *checker* e um *reference model*. No *scoreboard* são recebidos os estímulos gerados do DUV e do *reference model*. As saídas desses componentes são comparadas no *checker*. O *reference model* é o modelo ideal que já é testado e serve de modelo para a implementação do DUV. Segundo Bergeron[22], os componentes da UVM que são originados de uma classe da UVM são definidos a seguir [2].

Figura 2.5: Ambiente de verificação funcional



Fonte: próprio autor.

2.7.1 *Testbench*

O *testbench*, normalmente, é responsável por instanciar o módulo em verificação DUV e a classe *test*, configurando as conexões entre eles. Caso tenha componentes que são baseados em módulos, eles também serão instanciados no *testbench*. As interfaces TLM utilizadas no UVM dispõem de um conjunto consistente de métodos de comunicação para enviar e receber transações entre os componentes. Os próprios componentes quando são instanciados

e conectados no *testbench* conseguem realizar diferentes operações que são necessárias para verificar um projeto. Uma grande observação é que o *test* é instanciado dinamicamente em tempo de execução, assim possibilitando que o *testbench* seja compilado uma vez e executado com vários testes diferentes [2].

2.7.2 *Test*

O *test* é o componente de nível superior que está contido no *testbench*. Na Figura 2.5 parece que o *testbench* é o componente de maior nível hierárquico, no entanto, sua função é instanciar o DUV e o *test* e configurar a conexão entre eles. Um *test* é o componente que encapsula instruções de escrita para realizar vetores de teste [2]. Os testes são, normalmente, instanciados pelo *testbench*, permitindo inicializar e enviar a sequência de estímulos para o DUV.

A utilização de classes permite usar a herança e consegue obter a reutilização de testes. Geralmente, uma classe de teste de base é definida para instância e configuração do ambiente, então estendida para estabelecer as configurações específicas do local, como quais sequências executar, parâmetros de cobertura, entre outros.

2.7.3 *environment*

O *environment* é um componente que juntam outros componentes interligando-os. Os componentes que são instanciados dentro do *environment* são os *agents*, *scoreboard* ou até mesmos outros *environments*. Ele encapsula todos os componentes no processo de verificação do DUV [2]. O *environment* instância e configura o IP-*core* de verificação reutilizável e define a configuração padrão desse IP. Vários testes podem instanciar o componente *environment* e determinar a sequência de estímulos gerados e enviados para a configuração selecionada.

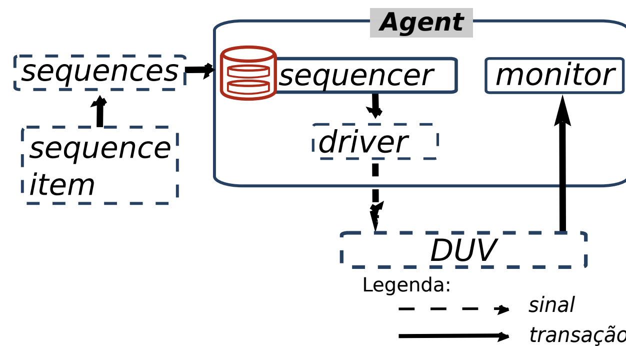
2.7.4 *Agent*

O *agent* é um componente hierárquico capaz de agrupar outros componentes de verificação que trabalham com o DUV. O *agent* é composto pelos seguintes componentes, contém um *sequencer* para administrar o fluxo de estímulos, um *driver* que possa aplicar estímulos pela interface do DUV e um *monitor* com o objetivo de monitorar a interface do DUV.

Em alguns casos, os *agents* podem incluir outros componentes, por exemplo, coletores de cobertura, verificadores de protocolo e um modelo TLM Mehta[2].

Dessa forma, esse componente conduz os sinais pela interface de nível de sinal para o DUV. As transações são geradas por um *sequences* que as transmite para o *sequencer* como transações, em seguida, são enviadas para o *driver*, que consegue converter uma transação em sinais. Isso é ilustrado na Figura 2.6.

Figura 2.6: Agente UVM



Fonte: próprio autor.

Existe dois modos que o *agent* pode operar, como *agent active* e o outro como *agent passive*, conforme ilustrado na Figura 2.5. No modo ativo, ele pode produzir o estímulo (ou seja, o *driver* aciona a entrada do DUV e detecta as saídas do DUV. Enquanto, no modo passivo, o *driver* e o *sequencer* permanecem desabilitados e apenas o *monitor* é ativo para monitora as saídas do DUV. Esses dois modos podem ser configurados dinamicamente.

2.7.4.1 *Sequence item*

O *sequence item* é o componente de menor nível hierarquia. É definido como transação básica que será utilizada para gerar sequências. Esse componente define os itens de dados da transação básica e/ou limitações impostas a eles. O *driver* lida com atividades de sinal no nível de bit, deixando transação mais rápida, já que não faz sentido manter esse nível de abstração conforme se distância do DUV. As transações são definidas como as menores transferências de dados que podem ser realizadas em um modelo de verificação. Elas conseguem inserir variáveis, restrições e até métodos para atuar sobre si mesmos [2].

2.7.4.2 *Sequences*

Após *sequence item* ser criado, o *testbench* deve gerar as *sequences* utilizando o *sequence item*, que envia os estímulos para o *sequencer*. As sequências de estímulos enviadas são uma junção ordenada de transações, em que as transações são moldadas conforme a necessidade dos testes. A finalidade do componente *sequences* é criar várias transações. Após criar essas transações, tem outra classe que as leva para o *sequencer* [2].

2.7.4.3 *Sequencer*

O *sequencer* é um componente responsável por controlar o fluxo criado pelo *sequence item* de requisição e de resposta na comunicação entre os componentes *sequencer* e *driver* através da interface TLM. O *sequencer* é usado como árbitro para controlar o fluxo de transações de várias sequências de estímulos.

2.7.5 *Driver*

O *driver* é o componente responsável pela conversão de transações em sinais. Esse componente consegue receber as sequências ao nível de transação do *sequencer*, convertê-las ao nível de sinal e as coordená-las para interface do DUV. Essa funcionalidade *driver* deve ser restrito apenas para envio de dados necessários para o DUV. Repare que nada empata que o *driver* consiga monitorar os dados transmitidos ou recebidos do DUV [2]. O *driver* contém uma porta TLM que serve para receber as transações do *sequencer* e enviá-las para o DUV, isso após convertê-las em sinais, assim são conduzidas por uma interface de acesso no DUV.

2.7.6 *Monitor*

Ao contrário do *driver*, o *monitor* é um componente que adquiri as atividades de nível de sinal e as converte de volta ao nível de transação para serem encaminhadas para o resto do *testbench*. Esse componente transmite as transações criadas por sua porta TLM. Perceba que a comparação da saída recebida pelo DUV com aquela com a saída já esperada é normalmente feita no componente *scoreboard*, e não exatamente no componente *monitor*

[2].

A utilidade de um *monitor* é monitorar (receber) os sinais enviados do DUV e os converter para transações. Com isso, a função do *scoreboard* é receber as transações transmitidas do *monitor* e fazer a comparação com os resultados coletados. O *monitor* executa alguns procedimentos internos nas transações geradas como, por exemplo, coleta de cobertura, verificação, registro, gravação, entre outros, ou pode ceder isso a componentes dedicados conectados à porta de análise do *monitor*.

2.7.7 *Scoreboard*

O *scoreboard* é o componente responsável pela verificação. Ele verifica a resposta do DUV em relação à resposta esperada (de um modelo ideal testado). O *scoreboard* recebe transações vindas do DUV passando pelo *monitor* por portas de análise do *agent active* e vindas do *reference model* por meio de portas de análise do *agent passive*, então compara ambos os modelos no componente *checker* [2].

2.7.8 *Reference Model*

O *reference model* é o componente implementado com um código testado e funcional, que deve produzir a resposta certa. O *reference model* pode ser um modelo em que utiliza linguagens de programação C, C ++ um modelo *SystemC* TLM2.0 ou simplesmente outro modelo *SystemVerilog*, são diversos modelos que podem ser utilizados.

2.7.9 *Checker*

O *checker* é um componente com função de comparar dados que chegam do *reference model* e do DUV. Essa comparação é em nível de transação das respostas geradas na saída do *reference model* e do DUV

2.7.10 *Design Under Verification*

O DUV é o componente antes da manufaturação, ou seja, antes de o *chip* ser fabricado. Ele é implementado sem considerar a verificação funcional [22], é utilizado quando é realizada

uma verificação de uma implementação, ou seja, não verifica dispositivos concluídos. Ele deve ser implementado para facilitar o processo de verificação funcional. Assim, pode-se validar ou não um DUV, confirmando ou não sua conformidade da implementação com a especificação.

Existe uma certa confusão quanto a nomenclatura a ser adotada para verificação da implementação rodando em um dispositivo a ser verificado. Em alguns trabalhos é encontrado a terminologia de verificação sob o projeto (DUT, do inglês *Design Under Test*), enquanto este trabalho adota a terminologia denominada DUV. O DUT é o RTL depois da validação que já foi manufaturado, enquanto o DUV é o DUT antes da manufaturação, ou seja, antes de o *chip* ser fabricado. Dessa forma, o DUV a ser verificado pode ser implementado em diferentes níveis de abstração. Este trabalho inicia a implantação do DUV a partir de um modelo comportamental, que depois pode ser refinado até chegar em sua *netlist* ao nível de portas lógicas [29].

2.8 Considerações finais

Neste capítulo foi apresentado as etapas que precedem e sucedem à implementação do RTL. Assim como é descrito o ciclo de verificação e o fluxo de projeto de desenvolvimento de *hardware*. Isso permite analisar o fluxo de projeto e aonde se situa os principais desafios para desenvolver um IP-*core* com a menor quantidade de falhas possíveis. Assim, surge a necessidade de abordar os conceitos do desenvolvimento de um ambiente de verificação com base na metodologia UVM que utilizando uma técnica pode integrar IP-*cores* implementados em arquiteturas heterogêneas. Isso permite acelerar o processo de validação e verificação sem a necessidade de re-codificação que levaria um tempo considerável de projeto. Outras metodologias poderiam ser testadas proporcionando uma redução de tempo. Foram explicados os principais conceitos da metodologia UVM, antes de descrever as nomenclaturas do ambiente de verificação funcional. A parte de maior impacto em um projeto de componentes digitais é a etapa verificação funcional, e em muitos trabalhos ela não seja priorizada no momento da implementação do DUV. No próximo capítulo serão apresentados os conceitos envolvendo a especificação de um padrão de comunicação que será a base para realizar a simulação distribuída deste trabalho.

Capítulo 3

Interoperabilidade na computação distribuída

No capítulo anterior foram apresentados os conceitos de verificação funcional. Enquanto, neste capítulo apresentam-se os principais conceitos sobre uma especificação para intercomunicação de simuladores. Dessa forma, serão definidos os principais conceitos relacionados à simulação paralela e simulação distribuída, assim como HLA.

Organizou-se a apresentação deste capítulo da seguinte forma: na Seção 3.1 apresentam-se os conceitos da simulação paralela e da simulação distribuída; na Seção 3.2 conceitua-se o HLA com suas nomenclaturas e definições baseadas na documentação; na Seção 3.3 abordam-se as considerações finais deste capítulo.

3.1 Simulação Paralela e Simulação Distribuída

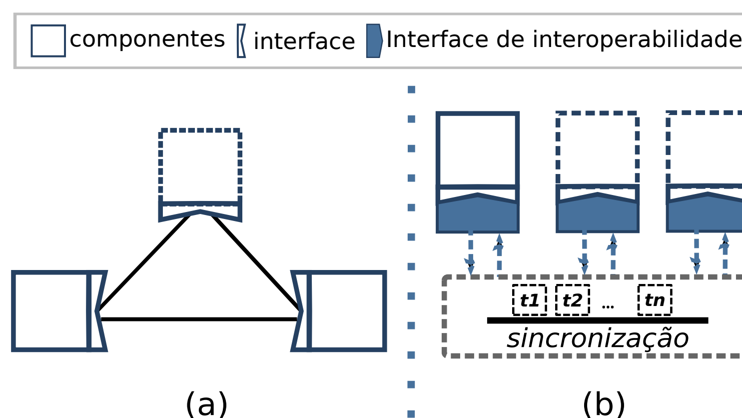
Na parte de computação paralela, existe um grupo de computadores agrupados fisicamente, que devem estar juntos por um único compartimento como, por exemplo, gabinete ou separado de forma que esteja um em cada compartimento, mas tendo o mesmo propósito que é fazer com que os processadores fiquem de maneira próxima compartilhando outros componentes. Dessa forma, os computadores próximos e que utilizam computação paralela, normalmente, contém *hardware* de comutação adequada para a função, em que possibilita a transmissão ou compartilhamento de mensagens de um computador para o outro com

um curto tempo de resposta [30]. Enquanto isso, a simulação distribuída os computadores estão distribuídos em ambientes físicos. Na simulação distribuída o tempo de resposta na troca de mensagens é maior em comparação a simulação paralela, já que com o uso de rede de área local (LAN, do inglês *Local Area Network*) ou rede de longa distância (WAN, do inglês *Wide Area Network*) exige uma transmissão mais lenta comparada a computadores acoplados que tem um poder de respostar maior. Na simulação tratada com a execução de programas de simulação em computadores paralelos, com diversos processadores que usam memória compartilhada, são definidas como simulação paralela. Na simulação tratada com a execução de simulações com computadores geograficamente dispersos, conectados através de algum tipo de rede, são definidas como simulações distribuídas.

3.1.1 Simulação distribuída entre arquiteturas heterogêneas

Para realizar a conexão entre diferentes dispositivos, são utilizadas as interfaces de comunicação. Em redes convencionais, há um ponto de acesso pelo qual todos os dispositivos trocam as informações. Nas redes *ad hoc* os dispositivos podem se comunicar diretamente entre si, o que pode permitir que haja maior flexibilidade na rede, conforme ilustrado na Figura 3.1 (a). Essa abordagem causa problemas de escalabilidade devido ao aumento do número de módulos de conexão e à extensão do módulo para sincronização distribuída.

Figura 3.1: Simulação entre arquiteturas heterogêneas. (a) abordagem da implementação *ad hoc*. (b) implementação proposta com abordagem de interoperação.



Fonte: Próprio autor.

Na Figura 3.1(a) ilustra-se o sistema nas redes *ad hoc* que não há centralização com dificuldade devido à heterogeneidade das arquiteturas com uma sincronização mais complexa

[31]. Na Figura 3.1(b) ilustra-se o sistema centralizado em que se perde desempenho e é vulnerável a falhas, permitindo simular partes de código em dispositivos. Este trabalho, com base na pesquisa em Silva et al.[17], implementa a integração de diferentes arquiteturas heterogêneas por meio da interoperação.

3.2 High Level Architecture

A HLA foi originalmente desenvolvida pelo Escritório de Modelagem e Simulação (DMSO, do inglês *Defence Modelling and Simulation Office*), é uma arquitetura utilizada como *middleware* de comunicação e sincronização para diferentes ambientes, com diferentes funcionalidades, sendo algumas: simulação com restrições em tempo real [32]; integração de simuladores heterogêneos [11]; simuladores com *hardware-in-the-loop* [33].

Essa arquitetura é especificada por três documentos [34]:

- HLA template de modelo de objetos do HLA (OMT, do inglês *Object Model Template*): uma descrição para estabelecer o formato e sintaxe de modelos de objetos HLA. A definição de objeto HLA é semelhante à ideia de objetos em programação orientada objetos conforme os padrões [35];

- *Framework* e Regras HLA (do inglês HLA *rules*): contém uma definição da HLA, seus componentes e dez regras fundamentais que em conjunto descrevem os conceitos gerais desta arquitetura de alto nível seguindo os padrões [34];

- Interface de especificação HLA (do inglês HLA *interface specification*): uma definição da interface funcional entre simulações (federados) e RTI conforme os padrões [36]. Sendo assim a infraestrutura de tempo de execução (RTI, do inglês *Run Time Infrastructure*) fornece serviços para federados, que comparando é como um sistema operacional (SO) distribuído fornece serviços para suas aplicações. O documento de Interface de especificação HLA contém bases para que sejam desenvolvidas aplicações que possam ser implementadas com código aberto utilizando uma arquitetura de alto nível.

A principal característica do HLA é suportar a reutilização e a interoperabilidade. A

interoperabilidade é um termo que abrange mais do que apenas enviar e receber dados, entende-se que é o processo de comunicação de dois ou mais sistemas sem, necessariamente, a geração de uma dependência tecnológica entre os mesmos, permitindo que diversos sistemas e organizações trabalharem em conjunto, de modo a garantir que sistemas computacionais interajam para trocar informações de maneira eficaz e eficiente. Os sistemas devem operar de tal forma que eles consigam atingir um objetivo conjunto através da colaboração.

O HLA é um padrão definido pelos engenheiros eletrotécnicos e eletrônicos (IEEE, do inglês *Electrical and Electronics Engineers*) [34], e desenvolvido pela organização padrões de interoperabilidade de simulação (SISO, do inglês *Simulation Interoperability Standards Organization*). Inicialmente, não era um padrão aberto, mas depois foi reconhecido e adotado pelo OMG e IEEE. Esse padrão foi desenvolvido para ser independente de qualquer linguagem ou plataforma, é considerado um protocolo de transporte [37], utilizado com sistemas distribuídos e heterogêneos.

O padrão introduz uma estrutura comum para a arquitetura do sistema, definindo os principais os elementos funcionais, interfaces e regras de projeto. Os elementos funcionais especificam o conjunto de recursos que serão fornecidos. As interfaces definem como o usuário irá usá-las. As regras de projeto introduzem as práticas de como essas funções utilizarão as interfaces para a criação de um sistema [38].

3.2.1 Template de modelo de objetos

O OMT é responsável por definir o formato e a sintaxe para gravação de informações nos modelos de objetos HLA, que seja capaz de inserir objetos, atributos, interações e os parâmetros. O padrão HLA especifica dois tipos de modelos de objetos: o HLA modelo de objeto da federação (FOM, do inglês *Federation Object Model*) e o HLA modelo de objetos de simulação (SOM, do inglês *Simulation Object Model*) [35]. A documentação do FOM e SOM é estabelecida de acordo com OMT segundo a descrição em [35]. Sendo assim o OMT estabelece um formato e uma sintaxe apropriado para suas documentações e não os conteúdos de um SOM ou FOM. O documento OMT é a descrição dos elementos essenciais compartilhados pela simulação ou federação em termos de objetos.

O padrão HLA não coloca obstáculos no conteúdo dos modelos de objetos. O OMT

é um template geral para especificação das tabelas que precisam ser documentadas [39]. Esses templates são os meios para compartilhamento aberto de informações através da comunidade. Para facilitar o reúso, esses templates completos são disponibilizados para que ferramentas automatizadas possam realizar buscas sobre os dados do modelo de objetos.

3.2.1.1 Federation Object Model

A HLA proporciona algumas vantagens, uma delas é que além dela suportar uma diversidade de políticas de gerenciamento de tempo, também concede a interoperabilidade entre os federados com diversas políticas [36]. Especificada por dois modelos de objetos: FOM que é o conjunto de objetos, atributos e interações, os quais são compartilhados através da federação; e SOM que é a simulação como tipos de objetos, atributos e interações, que ele pode oferecer para futuras federações [35].

O FOM é composto por: conjunto classes de objetos selecionados com o objetivo de representar a informação trocada em uma federação; conjunto de classes de interação selecionados para retratar a ação combinada entre os objetos criados; atributos de classes de objetos e os parâmetros de classes de interação, também outras informações que são bem relevantes [35].

No FOM, os componentes nele presente determinam o conjunto de SOM dos federados necessário que participam da federação, garantindo a comunicação transparente das informações entre os federados. Em uma HLA, todo objeto é uma instância de uma classe de objeto que está estabelecida no FOM [40].

3.2.1.2 Simulation Object Model

O SOM descreve os tipos de informações do federado em que pode prover para as federações, sendo elas distintas de outras informações referentes ao federado. Pode-se identificar um conjunto estabelecido de objetos e interações suportados concedidos no SOM do federado responsável por definir o FOM [40].

3.2.2 Regras do HLA

As regras são divididas em dois grupos: regras de federação e de federado. Nesta seção, explicam-se, de forma resumida, as 10 regras que resumem o padrão HLA, de modo que são cinco para federação e cinco para federado [34].

As regras para federação são as seguintes [34]:

1. De acordo com o OMT, as federações necessitam ter documentado um FOM;
2. Na federação, deve-se ter todas as representações de instâncias de objetos em simulação no federado e não no RTI;
3. Na execução da simulação, toda a troca de dados entre federados devem ocorrer através do RTI;
4. A especificação de interface HLA dos federados devem interagir com o RTI;
5. Durante a execução da federação, um atributo de uma instância deve ser propriedade de no máximo um federado em um dado instante de tempo.

As regras para federado são as seguintes [34]:

1. Devem ter um SOM, em que deve ser documentado conforme o OMT;
2. Como especificado em seus SOMs, os federados devem estar aptos a atualizar quaisquer atributos, como refletir, enviar, receber interações;
3. Como especificado em seus SOMs, os federados devem estar aptos a transferir e/ou aceitar propriedade sobre atributos dinamicamente durante a execução da federação;
4. Como especificado nos seus SOMs, os federados devem ter as condições de atualização dos seus atributos, assim como são aptos a variar;
5. Os federados devem estar aptos a gerenciar o tempo local, de modo que permita a coordenação da troca de dados com outros membros da federação.

3.2.3 Especificação da interface do HLA

O documento da especificação da interface do HLA descreve os serviços que o federado pode usar para se comunicar com outros federados pelo RTI. Além de descrever os serviços que podem ser utilizados por um federado e os que não são fornecidos.

As seis classes de serviços são as que descrevem [36]:

1. Gerenciamento de Federação (do inglês *federation management*) – os serviços que oferecem funções básicas para criar e operar uma federação;

2. Gerenciamento de declaração (do inglês *declaration management*) – as que suportam gerenciamento eficiente de troca de dados, por meio das informações fornecidas pelos federados, por sua vez, em que os dados fornecem e solicitam durante a execução da federação;

3. Gerenciamento de objetos (do inglês *object management*) – oferecem os serviços para criação, exclusão, identificação, além de outros serviços ao nível de objeto;

4. Gestão de propriedade (do inglês *ownership management*) – que oferece o suporte para transferência dinâmica de propriedade de objetos/atributos durante a execução;

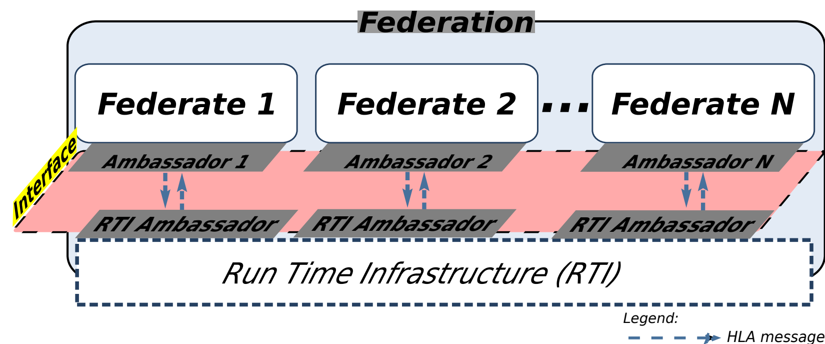
5. Gerenciamento de tempo (do inglês *time management*) – com suporte à sincronização nas trocas de dados durante a simulação;

6. Gerenciamento de distribuição de dados (do inglês *data distribution management*) – os serviços com suporte ao roteamento eficiente de dados entre os federados durante a execução da federação.

3.2.4 Estrutura Geral

Dentre as funcionalidades, o HLA permite separar uma funcionalidade específica do sistema em vários componentes usando uma infraestrutura de propósito geral [11]. Essa infraestrutura, inicializa a realização da comunicação entre os componentes. A Figura 3.2 ilustra um cenário em que cada integrante está conectado ao RTI, os integrantes são denotados como federado (do inglês, *federate*) e o conjunto composto pelos federados e o RTI é chamado de federação (do inglês, *federation*).

Figura 3.2: Estrutura genérica de elementos da arquitetura HLA.



Fonte: próprio autor

Os principais elementos da HLA serão melhor descritos de maneiras mais específicas a seguir, sendo apontadas algumas definições que tem como base os padrões de conceitos que definem a HLA pela documentação em [34] [36] [35].

3.2.4.1 Federado

O federado é um membro da simulação com a arquitetura HLA [41]. Com o auxílio do FOM, FOM Document Data (FDD)/dados de execução da federação (FED, do inglês *Federation Exectuion Data*) e RTI pode estar conectado com outra aplicação. O federado é compreendido como uma aplicação ou está conectado a outra aplicação [34]. Existem alguns itens em destaque que estão nesse grupo, são eles gerenciadores de federações, sistemas do mundo real, simulações, coletores de dados, e outros utilitários.

3.2.4.2 Federação

A federação é formada por um conjunto de *federados* que se comunicam por meio dos serviços da RTI, operando com um FOM [41]. Assim, qualquer federado que esteja participando da efetuação da federação é definido como federado afiliado (do inglês *joined federate*) de acordo com as definições estabelecida pela documentação em [42].

3.2.4.3 Infraestrutura de Tempo de Execução

A RTI é descrita como um *software* que provê uma interface de serviços gerais para suportar sincronização e uma troca de dados enquanto acontece a execução de uma federação. Ainda tem como funcionalidade a sincronização de tempo, passagem de eventos e

troca de dados entre os componentes. Para isso, fornece para cada federado uma interface chamada embaixador do RTI local (RTIA, do inglês RTI *Ambassador*) e cada federado possui uma interface chamada embaixador (do inglês, *Ambassador*) do federado para exercer comunicação inversa. Para ocorrer as trocas de mensagens, um único componente global que funciona como um gateway central é responsável por entregar e realizar o *broadcast* das mensagens para todas as interfaces RTIA, denominado *gateway* do RTI global (RTIG, do inglês RTI *Gateway*). Cada federado está localmente associado a um processo único no RTI, que realiza a troca de dados através de protocolo de controle de transmissão/protocolo *internet* (TCP/IP, do inglês *Transmission Control Protocol/Internet Protocol*) para troca de dados com outros componentes [40].

Existem alguns serviços que podem ser iniciados pelo federado e outros pela RTI, alguns exemplos que são iniciados pelo federado são: *subscribe, publish, update, register*, entre outros. Alguns exemplos de serviços iniciados pela RTI (do inglês *Initiated Services*): *reflect, time advanced grant*, entre outros. Na Tabela 3.1 ilustra-se um breve resumo da finalidade de cada grupo de serviços que são ofertados pela RTI. Esse conjunto de serviços são definidos pela documentação em [36].

3.2.4.4 FOM *Document Data* (FDD)

O FOM *Document Data* ou FDD contém classes, atributos, parâmetros e outras informações que são derivadas do FOM, são utilizadas pelo RTI no decorrer do tempo de execução. Cada vez que uma federação é executada, necessita de um arquivo FDD [43].

3.2.4.5 Ordenação de mensagens

A ordenação das mensagens durante a execução da federação, algumas mensagens devem ser trocadas entre os federados por meio da RTI. Enquanto ocorre as trocas de mensagens entre os federados, elas podem conter um rótulo de tempo (do inglês *timerstamp*) como não conter. No caso que contem, pode estar individualmente associado a cada uma das mensagens. O *timerstamp* pode ser aplicado para ordenar o modo como essas mensagens são recebidas, existem dois tipos de ordem de recebimento de mensagens, são eles: receber pedido (RO, do inglês *Receive Order*) que descreve como uma mensagem que não tem

Tabela 3.1: Grupo de Serviços oferecidos pela RTI.Grupo de Serviços oferecidos pela RTI.

Grupo de Serviços	Definição
Federation management	Grupo de serviços para criação, controle dinâmico, modificação e finalização de uma execução de federação.
Declaration management (DM)	Federados afiliados utilizam esses serviços para declarar sua intenção em gerar informação.
Object management	Grupo de serviços para lidar com o registro, modificação, e finalização de instâncias de objetos e envio e recebimento de interações.
Ownership management	Grupo de serviços usado pelos federados afiliados e RTI para a transferência de posse de atributos de instância de objetos entre esses federados.
Time management	Esses serviços e outros mecanismos associados fornecem meios para ordenar a entrega de mensagens durante a execução da federação. Os serviços de gerenciamento de tempo também são utilizados para controlar o avanço de federados ao longo do eixo de tempo da federação durante sua execução.
Data distributionmanagement (DDM)	Federados afiliados utilizam esses serviços para reduzir a transmissão e recepção de dados irrelevantes. Enquanto que serviços DM fornecem informação na relevância de dados no nivel de atributo de classe, serviços DDM adicionam capacidade para refinar ainda mais os requisitos de dados no nivel de instância.
Support service	Definem vários serviços para recuperar informação sobre a federação, tais como classes e interações.

Fonte: Adaptado de [35]

garantia de ordem de entrega; e ordem marcada com tempo (TSO, do inglês *Time Stamped Order*) que associa um *timerstamp* associado as mensagens para entregar na ordem correta a um federado destino [40].

3.2.4.6 Tipos de federados

Os tipos de federados são definidos para federados, onde cada um pode ser, regulador de tempo (do inglês *time-regulating*), restrito de tempo (do inglês *time-constrained*), regulador e restrito de tempo, como também nem regulador e nem restrito de tempo. Os federados reguladores de tempo são aqueles que restringem o progresso de avanço do tempo de outro federado. De forma contrária, federados que detêm seu progresso de tempo definido por federados reguladores são denominados federados restritos de tempo. Quando um federado é gerado, em sua estrutura padrão, ele tem que ser definido como nem regulador e nem restrito de tempo [40].

3.2.4.7 Sincronização

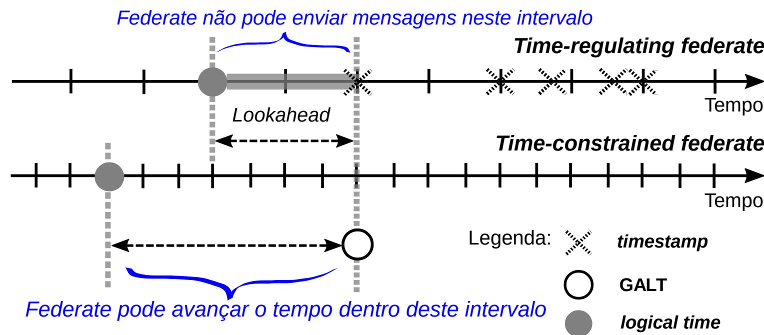
Ocorre por um serviço denominado de concessão de adiantamento de tempo (do inglês *Time Advance Grant*) (*callback*), em que a RTI concede o avanço do tempo. Assim, garante que nenhuma mensagem TSO será entregue com o rótulo de tempo (do inglês *timestamp*) menor do que o avanço do tempo fornecido. Dessa forma, a RTI assegura que os federados jamais receberão mensagens com *timestamp* menor do que seu tempo lógico atual [44].

A Figura 3.3 ilustra os tipos de federados em suas *timestamp*. Durante a sincronização é definido um limite e posto em cada um dos federados de tempo restrito. Esse limite delimita o quanto eles conseguem avançar no tempo, certificando que não avançaram o tempo e ultrapassaram um ponto em que as mensagens TSO conseguiriam ainda ser enviadas por outros federados afiliados. Se esses federados solicitarem um avanço além do limite seu tempo lógico (do inglês *logical time*), o avanço de tempo não será fornecido até que esse limite seja aumentado além do tempo lógico a ser oferecido [36]. O limite no avanço é representado por um valor intitulado de maior tempo lógico disponível (GALT, do inglês *Greatest Available Logical Time*) [40].

Qualquer federado detêm um tempo lógico. Esse tempo de um federado deve ser definido como o ponto atual no eixo do tempo HLA [35]. Enquanto os *timestamps* são os pontos do eixo do tempo HLA conforme as atividades que estão associadas [40].

O termo lookahead é definido como um valor não negativo que determina o menor valor dos *timestamps*, para os quais podem ser enviadas mensagens TSO pelo federado

Figura 3.3: Estrutura genérica do federado com *time-regulator* e um federado com *time-constrained*.



Fonte: próprio autor.

estabelecido como regulador de tempo. Um federado regulador de tempo não consegue enviar mensagens TSO que contenham *timestamp* com valor abaixo do seu tempo lógico somado ao valor do seu *lookahead* (ver Figura 3.3) [40].

3.3 Considerações finais

Existem vários padrões baseados em computação distribuída, tais como, SIMNET, simulação interativa distribuída (DIS, do inglês *Distributed Interactive Simulation*), arquitetura orientada a serviços (SOA, do inglês *Service-Oriented Architecture*), serviço de distribuição de dados (DDS, do inglês *Data Distribution Service*), HLA, entre outros. É preciso escolher aquela que apresente vantagens para reutilizar IP-*cores* e realize a intercomunicação com o ambiente de verificação. Dentre as tecnologias que existem, a escolha da arquitetura é baseada no trabalho Silva et al.[8] que adaptou HLA como meio de comunicação para as arquiteturas heterogêneas. Essa escolha é justificada pelo fato de facilitar a interoperabilidade e a composição da maior gama possível de plataformas.

Neste capítulo tratou-se um dos principais conceitos da fundamentação teórica para a contextualização da pesquisa. No capítulo definem-se os principais conceitos relacionados a simulação paralela e a simulação distribuída, assim como HLA, apresentando os principais conceitos dos elementos que a compõem e noções básicas de gerenciamento de tempo e sincronização. No próximo capítulo analisam-se os trabalhos relacionados para o desenvolvimento da abordagem proposta, destacando suas principais contribuições e características.

Capítulo 4

Trabalhos relacionados

Neste capítulo tratam-se os trabalhos relacionados de verificação funcional, de simulação distribuída e de sistemas heterogêneos.

Duas questões de pesquisa relevantes foram propostas:

1. Como realizar verificação funcional incluindo componentes implementados em diferentes arquiteturas, auxiliando a exploração arquitetural e evitando a inclusão de IP-*cores* inapropriados para o projeto?

2. Como verificar IP-*cores* em sistemas heterogêneos com diferentes níveis de abstração, sem prejuízo e reduzindo o tempo de projeto de SoC?

As questões de pesquisa pretendem levar a respostas de como reduzir o tempo de projeto ao verificar diferentes sistemas heterogêneos separados que são partes de um projeto. A partir das questões de pesquisa, uma *string* de busca foi selecionada com as seguintes palavras *("Functional verification" OR "verification") AND ("distributed") AND ("heterogeneous" AND ("System-on-a-Chip" OR "SoC")*, para realizar uma busca automatizada nas principais bases de dados de publicações científicas (*IEEEXPlore, ACM Digital Biblioteca, ScienceDirect.com, Scopus e ISI Web of Science*).

Os seguintes critérios de seleção dos trabalhos foram considerados:

a) O trabalho apresenta o contexto relacionado com verificação funcional;

b) O estudo envolve verificação funcional e apresenta alguma abordagem com arquiteturas heterogêneas;

c) *O trabalho apresenta o contexto relacionado com projetos de SoC.*

d) *O estudo apresenta estratégias para integração de simuladores e arquiteturas diversas, assim como métodos ou ferramentas;*

Com base nesses critérios, uma revisão bibliográfica é apresentada com trabalhos que envolvam verificação funcional e/ou simulação e com integração de arquiteturas heterogêneas.

Em Seok et al.[10], os autores propõem a implementação da Interface de linguagem de programação para interoperação (PLI-I, do inglês *Programming Language Interface for Interoperation*) como uma interface formal para integrar simuladores de *hardware*. Para isso, utiliza-se a arquitetura HLA que promove a interoperabilidade entre os componentes que compõem o sistema. Nesse contexto, um framework é proposto para simular uma combinação dos modelos de *hardware* heterogêneos de uma rede de sensores sem fio.

O HLA é utilizado para conseguir conectar sistemas para co-simulação. Em Brito et al.[11], o foco é a integração de cinco ferramentas de simulação distintas em que o HLA consegue integrar, sendo elas: Ptolemy II, SystemC, Omnet++, Veins, Stage e robôs físicos. Com as ferramentas citadas, foi implementado uma plataforma capaz de utilizar simulação distribuída com alguns simuladores heterogêneos.

A proposta do trabalho de Lima[1] visa trazer uma abordagem para reúso de IP-*cores*, em um processo de desenvolvimento denominado ipPROCESS, visando facilitar e conduzir o projeto de IP-*core* de alta qualidade, assim, promovendo a identificação de erros durante as primeiras fases do projeto. Baseado nas análises de IP-*cores*, o foco deste trabalho é fomentar e apoiar a produção de novos componentes com o objetivo de aplicar o reúso em projetos de SoCs, já que a reusabilidade é um dos principais desafios dos projetos.

Uma simulação mista é introduzida no trabalho de Tran et al.[12] para coordenar várias simulações paralelas como um sistema de simulação distribuído. As simulações paralelas são conduzidas de acordo com HLA, funcionando como uma ponte de co-simulação.

Tratando de verificação de integração de sistemas, os autores de KrÜger, Meisinger e Menarini[45] abordam metodologias para verificação em tempo de execução. Nesses trabalhos, os autores buscam soluções diferentes para verificar sistemas já implementados, sem a necessidade de desconectar o sistema para realizar a verificação.

Segundo Rashmi, Somayaji e Bhamidipathi[46], trata uma forma eficaz e transparente de reutilizar arquitetura completa de verificação funcional de IP-*core* para SoC. Os autores resolvem o problema de reutilização de um produto customizado, utilizando a Metodologia de verificação de IP-*cores* em uma verificação SoC direcionada a redução em todo ciclo de desenvolvimento de testcase SoC.

O trabalho de Brodtkorb et al.[47] realiza uma explanação de trabalhos que envolvem sistemas heterogêneos para apontar o que se tem produzido de melhor. Essa revisão surgiu devido à necessidade de uma visão geral e compreensão do paralelismo de alta precisão que vinha evoluindo. Em especial, o trabalho contribui com uma revisão do *hardware*, ferramentas disponíveis de *software* e uma visão geral das técnicas e algoritmos de ponta. Além disso, descrevem-se uma comparação qualitativa e quantitativa das arquiteturas e uma visão sobre o futuro da computação heterogênea. Este trabalho em comparação ao dos autores, tem o propósito de realizar uma revisão literária de trabalhos que envolva sistemas heterogêneas.

Na literatura é difícil encontrar trabalhos que se dedicam à verificação funcional de arquiteturas heterogênea. O trabalho de Hekmatpour et al.[13] aborda uma plataforma de verificação distribuída com suporte para um ambiente de: simulações heterogêneas, testes, projetos de HDL, linguagens e ferramentas de análise. A plataforma é baseada em *web* e fornece verificação funcional integrada para projetos de circuitos integrados, tais como processadores que incluem unidades em VHDL, Verilog ou *SystemC*. O que é importante ressaltar é a interface de simulação flexível, o ambiente dinâmico eficiente de gerenciamento baseado na *web*. Enquanto, este projeto deve usar uma técnica que gerencie todos os integrantes, assim, verificando a escalabilidade e desempenho.

Ao utilizar vários sistemas heterogêneos é preciso de uma rede mais robusta. Em Muhr, Holler e Horauer[14], realiza-se uma pesquisa dedicada a questão de verificação (*hardware* e *software*), apresentando um novo e eficiente ambiente heterogêneo de simulação para sistemas embarcados distribuídos centrados em rede. Os autores abordam pela primeira vez a integração da implementação com um Kernel do sistema operacional, permitindo testar o *software* do driver e as pilhas de protocolos. Essa integração permite desconectar de forma inteligente as simulações desnecessárias. Em comparação com este trabalho, busca-se verificação eficiente de sistemas heterogêneos.

Dentre os desafios em verificação, o trabalho Barnasconi et al.[15], utilizando o UVM, tanto é utilizado em sistemas digitais quanto pode ser usado em sistemas analógicos e mistos. Os autores abordam verificação funcional com a estrutura UVM-*SystemC*-AMS que é baseado em *SystemC* e AMS. Para demostrar a flexibilidade e desempenho da abordagem, os autores utilizam verificação e validação através de simulação *hardware-in-the-loop* (HIL). O diferencial dos autores é o AMS, que é uma extensão do *SystemC* para modelagem de sistemas analógicos, em questão de realizar simulação conjunta a verificação com HIL.

O trabalho de Wetter e Haves[48] propõe a modelagem e simulação de um sistema heterogêneo. A maior contribuição desse trabalho se deve à integração dos diversos simuladores a partir do Ptolemy. No entanto, os autores não se preocuparam em realizar a simulação distribuída desses sistemas.

O trabalho Neema[49] expõe os padrões, *frameworks* e métodos existentes para co-simulações distribuídas, desenvolvidos em um conjunto de requisitos essenciais para co-simulações distribuídas no mundo real e descritos em uma abordagem de integração baseada em modelo para integração de simulações de SoS para arquiteturas heterogêneas.

A Tabela 4.1 ilustra uma análise comparativa entre o trabalho proposto e os trabalhos mais relacionados encontrados na literatura, focando nas principais contribuições deste trabalho. Essa tabela tem como principais pontos: comunicação e sincronização, aplicação, prós e contras.

4.1 Considerações finais

Neste capítulo foi apresentado o estado da arte relacionado aos trabalhos que contribuem com verificação funcional e simulação distribuída, de modo que podem envolver arquiteturas heterogêneas. Com a *string* de busca foram encontrados trabalhos na literatura, sendo selecionados os trabalhos relevantes ao tema desta pesquisa. De maneira geral, diversos critérios de seleção permitiram relacionar trabalhos com verificação funcional e/ou simulação distribuída, além de que são voltados para área de projetos com SoC. Essa busca reforça quais trabalhos realizam verificação funcional e quais realizam integração de IP-*cores*, além daqueles que utilizam arquiteturas heterogêneas com IP-*cores*, valorizando as técnicas e métodos adotados e distingue pelo fato que permite escolher o IP-*core* ao realizar validação

Tabela 4.1: Comparação entre diferentes trabalhos com o trabalho proposto.

Trabalhos	Comum. & Sinc.	Aplicação	Prós	Contras
Seok et al. [10]	HLA	Co-simulação	Integração formal de simuladores HW, incluindo emuladores.	Não se preocupa em verificar diferentes sistemas desenvolvidos.
Brito et al. [11]	HLA	Simulação	Desenvolvimento de uma plataforma de simulação distribuída para simuladores heterogêneos	Atraso na sincronização e não realiza verificação funcional
Souto et al. [1]	UML, RTL e RMM	Simulação e Verificação	Verificação funcional para especificação e implementação de maior qualidade de IP-cores, reduzindo assim o número de erros no projeto.	Não cria um SoC com base em IP-cores, apenas modela.
Van Tran et al. [12]	HLA	Simulação	Permite coordenar várias simulações paralelas como um sistema de simulação distribuído.	Concentre-se na simulação, não integre arquiteturas heterogêneas.
Kruger et al. [45]	Enterprise Service Buses (ESB)	Verificação distribuída	Verificação em tempo de execução na integração de sistemas.	Não realiza verificação distribuída de arquitetura heterogêneos.
Gervais et al. [50]	HLA	Simulação	Aborda a simulação para analisar o desempenho em tempo real com o HLA.	Não realiza verificação funcional.
Hekmatpour et al. [13]	HLA	Simulação e Verificação	Uma interface de simulação flexível e com um ambiente dinâmico eficiente de gerenciamento baseado na web.	Não usa técnica que gerencie todos os integrantes.

Muhr et al. [14]	Ethernet	Simulação e Verificação	Verificação (hardware e software) com um ambiente heterogêneo de simulação para sistemas embarcados distribuídos centrados em rede.	A verificação não é distribuída e realizada entre arquiteturas heterogêneas.
Barnasconi et al. [15]	Ethernet	Simulação e Verificação	Verificação funcional com a estrutura UVM-SystemC-AMS, com validação por simulação HIL e extensão do SystemC para modelagem de sistemas analógicos	A verificação não ocorre distribuída entre arquiteturas heterogêneas.
Wetter e Haves [48]	Local	Simulação	Integração dos diversos simuladores a partir do Ptolemy	Não realiza a simulação distribuída.
Rashmi et al. [46]	UVM	Verification	Aborda o desafio de reutilização de IP-*core* ao propor um fluxo que permite a reutilização de dados aleatórios e estímulos de IP-*core* para o ambiente de verificação SoC	Comprovada apenas para um IP específico
Vineeth et al. [51]	SPI	Simulation and Verification	Verificação funcional com o SPI utilizando valores aleatórios.	A verificação não é realizada entre arquiteturas heterogêneas.
Trabalho Proposto	Virtual Bus implementando HLA/RTI	Computação Distribuída	Verificar partes já implementadas permitem acelerar o processo de desenvolvimento.	Em algumas aplicações, um grande número de federados pode possivelmente diminuir o desempenho.

e verificação funcional do RTL em arquiteturas heterogêneas.

Capítulo 5

Implementação da infraestrutura de interoperação

Neste capítulo serão apresentados os conceitos utilizados para construção da interface de comunicação. Assim como, a proposta para computação distribuída heterogênea responsável por enviar e receber dados na rede e por permitir a interoperabilidade entre múltiplas plataformas heterogêneas.

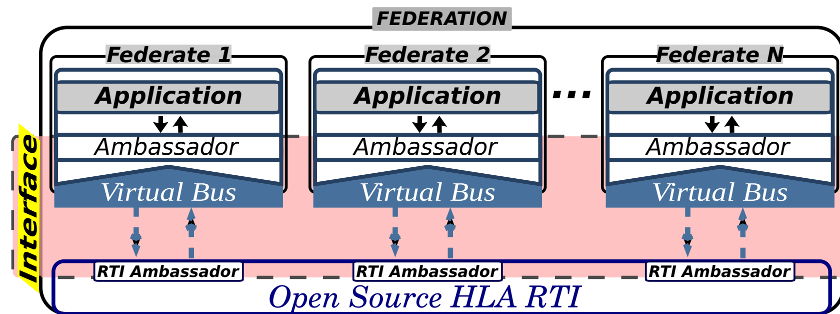
Organizou-se a apresentação deste capítulo da seguinte forma: na Seção 5.1 apresentam-se os conceitos e a estrutura da interface de comunicação baseada na especificação HLA; na Seção 5.2 trata-se a proposta de interoperação para computação heterogênea; na Seção 5.3 abordam-se os experimentos e os resultados de simulação para validar a interface de comunicação; na Seção 5.4 tratam-se as considerações deste capítulo.

5.1 *Virtual Bus*

No Capítulo 3 foram mostrados os conceitos e a estrutura da especificação HLA, como base para ser adaptada para implementação de uma interface transparente de comunicação. A partir dessa especificação é desenvolvida uma infraestrutura de intercomunicação, denominado *Virtual Bus*, como mostrado na Figura 5.1. Essa infraestrutura deve ser transparente para o usuário enviar dados durante a comunicação.

Nas Figuras 5.2(a) e 5.2(b) é ilustrado a interface implementada para a integração dos

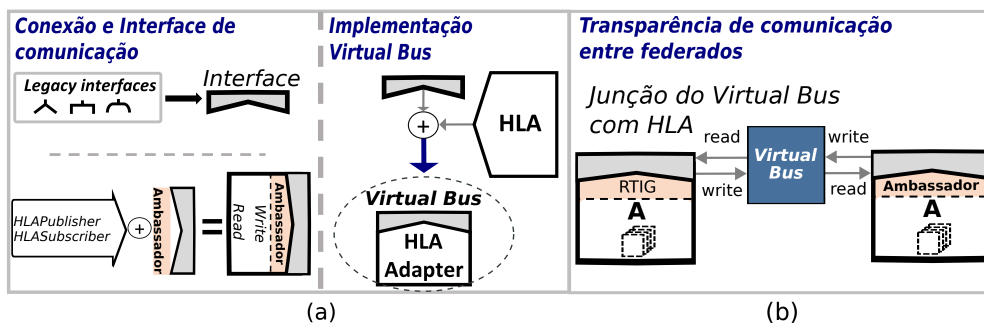
Figura 5.1: Adaptação do HLA como base para criação do *Virtual Bus*.



Fonte: próprio autor

módulos dos IP-*cores* via *Virtual Bus*. O *Virtual Bus* simplifica a comunicação e melhora a reutilização de componentes feitos nas mais diversas linguagens ou que pode ser executado em máquinas diferentes. Primeiramente, uma interface de comunicação é moldada para métodos simples de leitura e escrita determinados pelo *Virtual Bus* (Figura 5.2(a). Esses métodos encapsulam o mecanismo de publicação/assinatura (do inglês *publisher/subscriber*) e o procedimento de sincronização utilizado pelo HLA. Desta forma, cada elemento independente pode adentrar na federação e se comunicar de maneira síncrona (Figura 5.2(b)).

Figura 5.2: Etapas de implementação para criação do *Virtual Bus*. (a) Implementação de uma interface *Virtual Bus*. (b) Integração e interoperabilidade.



Fonte: próprio autor.

A implementação dessa interface de programação de aplicativo (API, do inglês *Application Programming Interface*) é baseada no trabalho de Silva et al.[8], que abordam a implementação e testes com arquiteturas heterogêneas. Ela é oculta para o usuário, já que, não há necessidade dele saber como acontece a transmissão dos dados durante a simulação.

No ambiente de verificação funcional, os novos integrantes devem ser estendidos para suportar o *Virtual Bus*. No ambiente, o *Virtual Bus* é composto por uma interface para

escrever e ler o tráfego de dados através de funções que são rotinas para acessar e modificar os dados entre os integrantes. As rotinas chamadas são descritas na Tabela 5.1.

Tabela 5.1: Descrição das funções básicas do *Virtual Bus*.

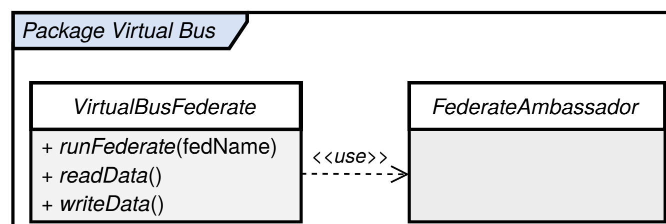
Funções	**Descrição**
write	Envia dados
read	Verifica e atualiza os dados recebidos
advanceTime	Avanço de tempo

O *Virtual Bus* é definido por três funções básicas:

- A função escrever (do inglês *write*) é responsável por enviar os dados para o RTI de um federado para outro, utilizando um "*id*" de identificação de destino;
- A função ler (do inglês *read*) retorna um valor booleano para indicar se algum dado foi recebido para aquele "*id*" de federado específico;
- A função avançar o tempo (do inglês *advanceTime*) executa a rotina que realiza o avanço de tempo.

Para construção do *Virtual Bus*, utilizou-se uma implementação de código *Open Source* CERTI/HLA [52], que implementa a especificação HLA, desenvolvido pelo escritório Nacional de Estudos e Pesquisas Aerospatiales (ONERA, do inglês *Office National d'Etudes et de Recherches Aerospatiales*) [53].

Figura 5.3: Componentes do pacote *Virtual Bus*.



Fonte: próprio autor

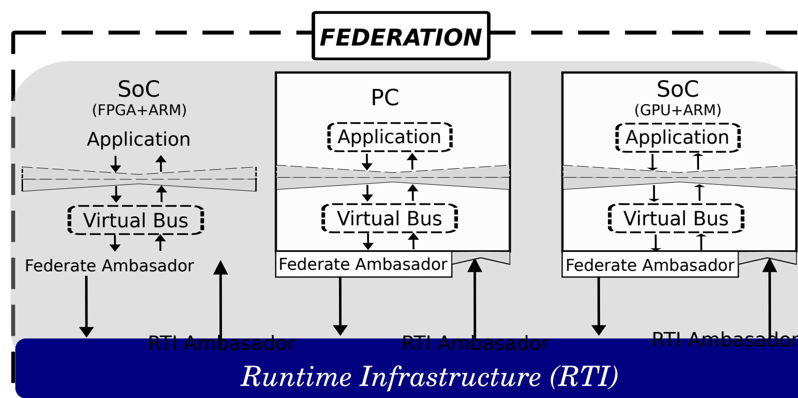
O *Virtual Bus* é uma API encapsulada para formar um *wrapper*, implementado em integrantes para sistemas distribuídos, e seu uso deve ser fácil e simples. Assim, as únicas

mudanças que são necessárias para integrar IP-*cores*, ou *open source* IP-*cores*, são adicionar algumas bibliotecas do CERTI/HLA e incluir o pacote *Virtual Bus* (do inglês *Package Virtual Bus*). Conforme mostrado na Figura 5.3, o *Package Virtual Bus* contém basicamente o *VirtualBusFederate* e o *FederateAmbassador* com suas classes e interfaces. A partir desse ponto, para usar o *Virtual Bus*, as únicas funções necessárias a serem chamadas são: *runFederate*, *readData* e *writeData*. O primeiro é para inicializar o federado, o segundo para enviar e o último para receber os dados. O *FederateAmbassador* é como um código de caixa-preta (do inglês *black box*), usado pelo *VirtualBusFederate* e não precisa ser chamado diretamente.

5.2 Interoperabilidade para computação heterogênea distribuída

A Figura 5.4 mostra diferentes federados se comunicando pela proposta do *Virtual Bus*, em que cada federado o implementa para tornar a computação distribuída mais transparente. Por exemplo, diferentes arquiteturas como CPU, GPU, ARM e FPGA podem ser conectadas usando a API implementada em cima do ambiente CERTI/HLA.

Figura 5.4: Arquitetura Geral do *Virtual Bus*.



Fonte: próprio autor

Para integrar os dispositivos em um mesmo meio, deve-se implementar em cada dispositivo um federado. Para isso, em linguagem C++, pode-se criar a estrutura de um federado, como ilustrado no Código 5.1, instanciando as linhas 2 e 3, para executar um federado a

linha 4 e para finalizar a linha 6.

Código 5.1: Pseudocódigo para criar, executar e finalizar um federado.

```
1  ...
2  VirtualBusFederate *federate;
3  federate = new VirtualBusFederate();
4  federate->runFederate( federateName );
5  ...
6  federate->finalize();
7  ...
```

Para ingressar em uma federação, um federado deve chamar a função *runFederate* da API do *Virtual Bus*, descrita em Código 5.2. Essa função cria uma instância de um RTI *Ambassador*, solicitando que o RTI crie a federação se ela não existir, e cria o *Federate Ambassador*. Esse federado se junta à federação e sinaliza ao RTI que está pronto para ser executado assim que todos os outros federados alcançarem o ponto de sincronização denominado READY_TO_RUN (Código 5.2, linha 9). Finalmente, chamando o método *publishAndSubscribe()* (Código 5.2, linha 11) ele configura a política de tempo e registra todos os objetos com interesse em receber e enviar atualizações.

Código 5.2: Pseudocódigo da função *runFederate*.

```
1  runFederate( char* federateName ){
2     //Create Federation if doesn't exist
3     rtiamb = new RTI::RTIambassador();
4     rtiamb->createFederationExecution();
5
6     //Create the FederateAmbassador
7     fedamb = new FederateAmbassador(federateName);
8     rtiamb->joinFederationExecution( federateName, fedamb );
9     rtiamb->synchronizationPoint( READY_TO_RUN );
10    ...
11    publishAndSubscribe();
12    oHandle = registerObject();
13 }
```

A função *writeData* é responsável pelo envio de dados através do *Virtual Bus*, a lógica da implementação é mostrada no Código 5.3, criando um objeto para manipular seus atributos.

Esses atributos se referem, respectivamente, linhas 3 – 8 do Código 5.3. Estes atributos precisam ser definidos dentro do arquivo *.fed* (Código 5.6, linhas 6 – 11, respectivamente). Todos os novos valores são configurados para esses atributos, enviados ao RTI com o tempo local HLA (do inglês *HLA local time*) desse federado. Também configura todos os federados com as políticas de *time-constrained* e *time-regulating*. Isso garante que um federado avança seu tempo (Código 5.3, linha 17), local para um tempo global especificado somente quando todos os outros federados também atingirem um tempo igual ou superior.

Código 5.3: Pseudocódigo *writeData*.

```
1  writeData(id, data){
2    attributes = new RTI::Attribute();
3    attributes->add(id);
4    attributes->add(data[0]);
5    attributes->add(data[1]);
6    attributes->add(data[2]);
7    ...
8    attributes->add(data[N]);
9
10   //Get HLA time from Federate
11   time = fedamb->federateTime();
12
13   //Updata value in RTI
14   rtiamb->updateValues(oHandle, *attributes, time);
15
16   //Advance time
17   fedamb->advanceTime();
18 }
```

A função *advanceTime* permite que cada federado avance seu tempo local após cada atualização dos valores dos atributos registrados nele, a lógica da implementação é lustrada no Código 5.4. Quando cada federado avança seu tempo local, o RTI avança o tempo global e um ciclo é concluído. Caso os IP-*cores* iniciam em um nível de abstração mais alto e seja necessário sincronizar os diferentes módulos, considera-se que o sincronismo é garantido através da implementação *Virtual Bus*/HLA nos diferentes federados. Caso inicie em um nível de abstração mais baixo é necessário acrescentar uma implementação na sincronização

dos módulos como, por exemplo, *handshake*. Para solicitar explicitamente o avanço do tempo, os federados devem implementar a função *advanceTime* e aguardar uma mensagem de concessão do RTI. Somente quando todos os federados são concedidos, o tempo global da federação é avançado. Enquanto isso, o federado está bloqueado aguardando essa concessão. Na interface *Virtual Bus*, o *advanceTime* é sempre chamado quando a função *writeData* do *Virtual Bus* é chamada.

Código 5.4: Pseudocódigo *advanceTime*.

```
1  advanceTime( double timestep ){
2    // request the advance
3    fedamb->isAdvancing = true;
4    RTIfedTime newTime = (fedamb->federateTime + timestep);
5    rtiamb->timeAdvanceRequest( newTime );
6    // wait for the time advance to be granted. ticking will tell the
7    // LRC to start delivering callbacks to the federate
8    while( fedamb->isAdvancing ){
9      rtiamb->tick();
10   }
11 }
```

Para receber dados, o federado deve usar a função *readData* que retorna os últimos dados recebidos do RTI, o pseudocódigo da função *readData* é mostrado em Código 5.5. A função funciona da seguinte maneira: se algum dado foi recebido, o federado atualiza um sinalizador para verdadeiro. Este sinalizador pode ser verificado pela função *hasReceivedData* (Código 5.5, linha 2). Se houver dados disponíveis, eles serão retornados (Código 5.5, linha 4); caso contrário, um valor nulo será retornado (Código 5.5, linha 7).

Código 5.5: Pseudocódigo *readData*.

```
1  Object readData(id, data){
2    if(fedamb->hasReceivedData(id)){
3      data = fedamb->getReceivedData(id, data);
4      return data;
5    }
6    else
7      return null;
8  }
```

A Figura 5.5(a) ilustra o fluxo de execução convencional do HLA, em que é necessário realizar adaptações para a interface de comunicação transparente. Enquanto, na interface *Virtual Bus* funciona com a interface HLA adaptada, conforme ilustrado na Figura 5.5(b). Pode-se visualizar nas figuras a abordagem com *Virtual Bus* implementando a interface comum para todas as aplicações, que encapsulam funções que eram chamadas separadamente para se comunicar em duas funções: de escrita e de leitura dos dados, somente com essas duas funções é possível realizar a comunicação dos federados, sendo adaptadas as funções do HLA para uso do *Virtual Bus*.

Figura 5.5: Fluxo de execução. (a) HLA. (b) *Virtual Bus*.

Fonte: próprio autor.

O fluxo de execução das funções do *Virtual Bus* funciona conforme ilustrado na Figura 5.5. No lado do remetente, a função *writeData* é usada para enviar dados através de RTI. Depois que essa função é chamada, os valores são atualizados chamando a função *updateValues*, que chama *updateAttributes*. Nessa etapa, cada componente de um item do *array* é reorganizado e passado para o RTI *Ambassador*. O controle de tempo e de distribuição de dados é realizado pelo RTI, chamando as funções para sincronizar todos os federados (pela função *waitSync*) e distribuir os dados entre todos os federados registrados (pela função *distributeData*).

No lado do receptor, os dados são refletidos em variáveis internas pela função *reflectAttribute*, chamada pelo *Federate Ambassador* do federado. Este método chama *receivedData* para armazenar dados internamente e define o sinalizador *hasReceivedData* como verdadeiro. Um federado com *Virtual Bus* é configurado para verificar a cada ciclo interno se algum novo dado foi recebido pelo método *readData*. Esse método verifica o sinalizador e,

se for verdadeiro, os dados são retornados a aplicação.

O formato dos dados trocados pelos embaixadores é definido segundo a documentação do OMT do HLA [36] sendo especificado em um arquivo comum a todos os federados. No *Virtual Bus*, cada objeto possui atributos para identificar o destino e a origem da mensagem, além de atributos de N valores de dados. Os N atributos são definidos, inicialmente, para troca de mensagens pela interface da aplicação, depois devem ser definidos os N atributos dentro do arquivo *.fed*.

Código 5.6: Modelo de objeto de dados para *Virtual Bus*.

```
1  (FED
2    (Federation Test) (class VirtualBus
3     (attribute privilege)
4         (class RTIprivate)
5         (class port
6           (attribute id)
7           (attribute data0)
8           (attribute data1)
9           (attribute data2)
10          ...
11          (attribute dataN)
12      ) ) ) )
```

O arquivo de descrição do modelo de objeto de dados para *Virtual Bus* é apresentado no Código 5.6. Para realizar o mapeamento dos pacotes de dados entre os federados, descreve-se o arquivo *.fed* usado pela federação. Esse arquivo tem descrito nele, atributos (*attribute*) com o campo de identificação (*id*) e o campo de dados (*data*). O tamanho dos atributos dos dados depende da sua natureza, em que N é o número de atributos necessários para troca de dados.

Conforme é inserido um novo federado a simulação tem que acrescentar uma classe no arquivo *.fed* para ocorrer o mapeamento desse novo membro. No Código 5.7 são implementadas as várias classes para cada federado integrante que implementa o modelo de dados do Código 5.6, criando canais que não interferem na troca de mensagem durante a simulação.

Código 5.7: Modelo base do arquivo *.fed* que deve ser implementado para cada sistema heterogênea com *Virtual Bus*.

```
1  (FED
2    (federacao Test) (class VirtualBus
3        (attribute privilege)
4        (class RTIprivate)
5        (class all
6                 (attribute 1) ... (attribute N)
7        )
8        (class duv 1
9                 (attribute 1) ... (attribute N)
10       )
11       (class duv N
12                (attribute 1) ... (attribute N)
13       ) ) ) )
```

5.3 Resultados experimentais

Estes experimentos foram executados com alguns federados que trocam dados de diferentes tipos e tamanhos. Portanto, o experimento foi montado com quatro federados: o *Sender Federate* (rodando em um PC), o SoC *Federate* (ARM + FPGA), o *multi-core Federate* e o GPU *Federate*. O *Sender Federate* envia imagens para os outros federados, que ao receber processam alguma operação nas imagens e retornam o resultado de volta ao *Sender Federate*. Todos os federados usam a mesma implementação do *Virtual Bus*.

Para facilitar a manipulação das imagens, o OpenCV foi usado nos federados *Sender* e *multi-core*. OpenCV é uma biblioteca usada para manipular e processar imagens, desenvolvida originalmente pela Intel (<http://opencv.org>), utilizado neste trabalho apenas para o manuseio básico de píxeis (do inglês *pixels*) através de suas funções e estruturas de dados.

Organizou-se a apresentação desta seção da seguinte forma: na Subseção 5.3.1 trata-se a lista de especificações dos equipamentos e como eles estão conectados; na Subseção 5.3.2 abordam-se os cenários configurados; na Subseção 5.3.3 descrevem-se os formatos de dados testados; nas Subseções 5.3.4 e 5.3.5 trata-se o funcionamento de cada federado com mais detalhes.

5.3.1 Equipamentos

Os equipamentos utilizados são representados por federados que compõe uma federação. A federação é composta por quatro federados: o *Sender Federate*, o *multi-core Federate*, o GPU *Federate* e o SoC (ARM + FPGA) *Federate*. A configuração básica de cada um deles é descrita na Tabela 5.2. O federado responsável por iniciar a comunicação é o *Sender Federate*, esse federado é um computador *desktop* que executa o Ubuntu 14.04 LTS. O SoC (ARM + FPGA) tem um FPGA *Cyclone* V integrado com um processador ARM *Cortex* A9 *dual-core* em um único *chip*, rodando o Ubuntu 12.04 LTS. O GPU *federate* usa uma *GeForce* GT da *NVidea* e estava executando o Ubuntu 16.04 LTS. O *multi-core Federate* executa o *OpenSuse* 13.2 *Harlequin*.

Tabela 5.2: Especificação do equipamento.

Dispositivos	**Configurações**
Sender	*Intel Celeron* 430 com 2GB SDRAM
SoC (ARM+FPGA)	Placa DE1-SoC da Terasic
multi-core	Intel i3-4005U com 4GB DDR3
GPU	*GeForce* GT 740M

5.3.2 Cenários

O experimento foi dividido em cinco cenários, conforme listado na Tabela 5.3. No primeiro cenário, o *Sender Federate* se comunica apenas com o SoC. A seguir, ele se comunica separadamente com os Federados *multi-core* e GPU no cenário 2 e no cenário 3, respectivamente. No cenário 4, a comunicação é feita entre o *Sender Federate*, SoC e *multi-core*. No último cenário, o *multi-core Federate* é substituído pelo GPU *Federate*.

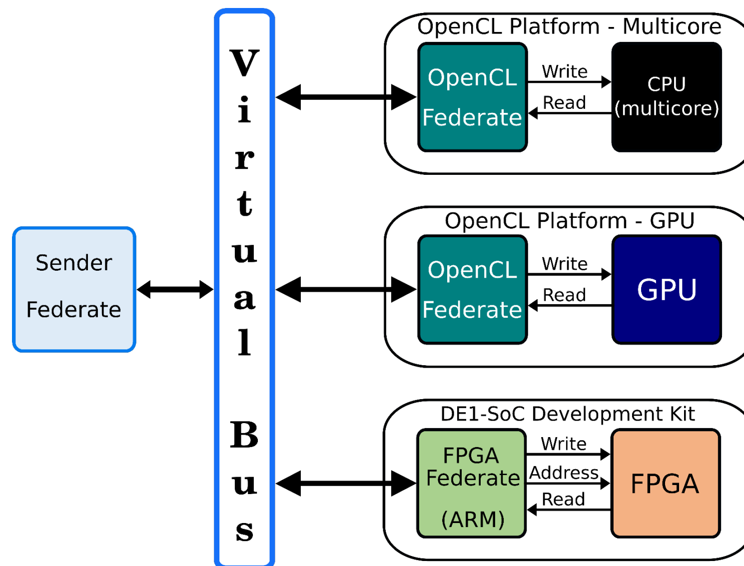
Tabela 5.3: Cenários usados nos experimentos.

Cenários	Sender	SoC (ARM+FPGA)	*multi-core*	GPU
1	X	X	-	-
2	X	-	X	-
3	X	-	-	X
4	X	X	X	-
5	X	X	-	X

Esses cenários se testam separadamente cada federado com o *Sender Federate* nos cenários 1 a 3 e, posteriormente, integram dois federados por experimento nos 4 e 5. Com isso, é possível analisar o comportamento do *Virtual Bus* em casos separados.

A Figura 5.6 ilustra uma visão geral de como os dispositivos estão conectados. No lado esquerdo, o *Sender Federate* está conectado e é responsável por gerar dados para todos os outros federados, assim como coletar os resultados deles. No lado direito estão os outros federados: SoC, em que a ARM faz a ponte entre o *Virtual Bus* com a FPGA, e os federados *multi-core* e GPU, que usam o OpenCL para interagir com o *Virtual Bus* com a arquitetura paralela.

Figura 5.6: *Virtual Bus* com sistemas heterogêneos.



Fonte: próprio autor

5.3.3 Configuração e troca de dados

Um dos desafios é melhorar a transferência de dados para uma taxa aceitável. Na fase inicial do experimento, foram utilizadas as seguintes estratégias de troca de dados:

- um por um (do inglês *one-by-one*): os *pixels* são enviados um por um em cada mensagem HLA;

- *multi-pixel*: um grupo de n *pixels* são enviados em n variáveis, uma variável para cada *pixel*;

- *multi-pixel* em um atributo: um grupo de n *pixels* são enviados em uma matriz n x n.

Um formato das mensagens para melhorar a troca de dados foi definido para a interface *Virtual Bus*. Esse formato geral das mensagens definidas para esses experimentos no *Virtual Bus* é mostrada na Figura 5.7.

Figura 5.7: Estrutura das mensagens.

Fonte: próprio autor

O campo chamado *data* contém parte da imagem que segue em cada mensagem. Durante os experimentos, tentou-se uma variação de tamanhos dos dados nas mensagens, o que resultou em diferentes metodologias de transferência de dados. Eles diferem essencialmente no número de *pixels* por mensagem e na maneira como as informações dos *pixels* são organizadas em variáveis.

A primeira metodologia, daqui por diante referida como *one-by-one*, foi implementada para enviar um *pixel* por mensagem. Isto é, para enviar uma imagem, cada mensagem tem a informação da fonte, mais a posição do *pixel* (x e y) e os dados de *pixel* correspondentes. Assim, a quantidade de mensagens é igual ao número de *pixels* na imagem. As mensagens foram estruturadas como mostrada na Figura 5.8.

Figura 5.8: Estrutura para mensagens *one-by-one*.

Fonte: próprio autor

Em que o campo de *source* é o *id* do *Sender Federate*, o *address* é o *id* do federado de destino que deve receber a mensagem, as posições xp e yp são as coordenadas de *pixel* sendo enviadas e o campo *pixel_ info* é o conteúdo de um único *pixel*.

Na segunda estratégia de envio do experimento, denominada *multi-pixels*, adotou-se a estratégia de enviar apenas informações da imagem na primeira mensagem, como resolução e números de canais.

Lembrando que os números dos elementos são iguais aos números dos *pixels* multiplicado pelos canais. Por exemplo, cinco *pixels* em uma imagem de três canais (RGB) significa quinze campos de dados por mensagem. A estrutura das mensagens é mostrada na Figura 5.9.

Figura 5.9: Exemplo de mensagem *multi-pixel*, transportando cinco *pixels* de três canais.

Fonte: próprio autor

Com base na primeira mensagem enviada com as informações de resolução e o número de canais, é possível gerenciar o recebimento de *pixels*. Assim, para enviar uma imagem completa, essa estratégia gerou o número de mensagens correspondentes ao número aproximado de elementos na matriz da imagem dividido pelo número de elementos enviados por mensagem, adicionando ainda a primeira mensagem.

A última estratégia usada para transferir as imagens, chamada *multi-pixel* em um atributo, é uma variação da segunda implementação. A estrutura da mensagem é a mesma apresentada na Figura 5.9, porém a HLA é usado de maneira diferente. Agora adicionam vários conteúdos de *pixel* em apenas um campo do atributo de mensagem HLA. Esse campo é gerenciado pelo HLA como um tipo de matriz, portanto, os dados de todos os *pixels* são encapsulados em um tipo de matriz exclusivo. No HLA, isso significa que o RTI tentará enviar o máximo de dados por pacote TCP, em vez de ter sido limitado pelo número de campos de dados.

Como é apresentado na próxima seção, essa metodologia chamada *multi-pixel* em um atributo obteve o melhor desempenho. Portanto, essa foi a abordagem escolhida para ser usada nos experimentos aqui apresentados.

5.3.4 Sender Federate

Como o nome sugere, este federado é responsável por enviar dados para serem processados pelos outros federados. Ele abre um arquivo de imagem com resolução de 512x512 *pixels* e envia todos os *pixels*. Na prática, essa imagem é uma matriz de elementos *char* não

assinados. Por ser uma imagem colorida, cada *pixel* é representado em três canais, isso é devido à representação RGB.

A transmissão de dados usando o *Virtual Bus* torna transparente as interações entre os federados envolvidos com a execução. Uma vez com a imagem carregada na memória, cada *pixel* foi fragmentado em um formato escalar para ser enviado em rajadas em um *loop* de envio principal. A primeira ação é iniciar o RTI (Código 5.8, linhas 2–4). A seguir, o valor do *pixel* em cada canal é armazenado em uma matriz (Código 5.8, linhas 7–10). As próximas linhas (Código 5.8, linhas 14–21) verificam se todos os elementos da imagem foram enviados. Finalmente, a matriz de dados é enviado para o RTI (Código 5.8, linha 26). As últimas linhas (Código 5.8, linhas 28–35) recebem dados do RTI e o processam dependendo de qual é o federado de origem.

O *Sender Federate* está em *loop* enviando dados até o limite definido pelo valor da variável denominada *NUMBER_OF_ELEMENTS_BURST* (Código 5.8, linha 6). Por exemplo, a estratégia escolhida usa um *burst* de 900 elementos. Isso significa enviar 300 *pixels* por mensagem. Portanto, a estrutura do laço de repetição *for* tem essa condição de parada porque ela armazena os 300 *pixels* na variável *data*, para enviá-la subsequentemente. As variáveis x e y representam as coordenadas do *pixel* que está sendo acessado.

Código 5.8: Loop principal no *Sender Federate* para enviar imagens.

```
1  ...
2  VirtualBusFederate *federate;
3  federate = new VirtualBusFederate();
4  federate->runFederate( federateName );
5  ...
6  for(i=0; i<NUMBER_OF_ELEMENTS_BURST; i++) {
7    Vec3b s = image.at<Vec3b>(Point(x, y));
8    data[position++] = s.val[0];
9    data[position++] = s.val[1];
10   data[position++] = s.val[2];
11
12   numberElemSent += 3;
13
14   if(++x == resolutionX) {
15     if(numberElemSent == totalElements) {
```

```
16          sentAllElements = true;
17          break;
18        } else {
19          //jump to next line
20          x = 0;
21          y++;
22        }
23    }
24  }
25
26  federate->writeData(Sender_ID, data);
27
28  if(federate->readData(src, data)) {
29    switch(src) {
30            case FPGA_ID :
31            //treats data from FPGA
32            ...
33        case PROCESSOR_ID :
34            //treats data from Multicore/GPU
35            ...
36        }
37  }
```

5.3.5 Integração SoC com *Virtual Bus*

Como prova de conceito, o algoritmo MD5 foi implementado e executado em FPGA. É uma função *hash* amplamente utilizada como soma de verificação para verificar a integridade dos dados [54], que recebe como entrada uma mensagem de tamanho arbitrário com máximo de 512 *bits* e produz como saída uma impressão digital de 128 *bits*. A mensagem de entrada pode ser um inteiro arbitrário e não negativo.

O Código 5.9 mostra como o federado é implementado no processador ARM. Como uma solução inicial, um *loop* é proposto em vez de usar interrupções do processador para verificar se algum dado é recebido. Uma vez recebida, a função para calcular o MD5 é chamada (Código 5.9, linha 9). O resultado é enviado somente ao *Virtual Bus* quando o cálculo é concluído. Isso é controlado por um sinalizador chamado *received* (Código 5.9,

linha 11). Quando o resultado do cálculo é enviado pelo FPGA, então às quatro palavras são enviadas para o *Virtual Bus* (Código 5.9, linha 12) e recebidas pelo *Sender Federate*.

Código 5.9: Lógica do federado em execução no ARM.

```
1  ...
2  // criar e executar o federado
3  VirtualBusFederate *federate;
4  federate = new VirtualBusFederate();
5  federate->runFederate( federateName );
6
7  while(1){
8    if(federate->readData(source, data)){
9       calculate_md5(data, a, b, c, d);
10   }
11   if(received) {
12     federate->writeData(federateName,a,b,c,d);
13     receive = false;
14   }
15 }
16 federate->finalize();
17 ...
```

A comunicação entre o ARM e o FPGA na camada de *software* é feita pela função *calculate_md5* (Código 5.9, linha 9). No ARM, a comunicação é feita através de registros escritos. Isso é configurado com a ferramenta *Qsys*, que mapeia a FPGA como um dispositivo periférico do processador ARM. O MD5 foi implementado para recebe uma sequência de 512 *bits*, separados em 16 blocos de 4 *bytes* cada.

O *Cyclone* V SE SoC tem uma limitação física que não permite a transmissão de 512 *bits* em um ciclo de *clock*. Então, foi criado um *wrapper* no *Verilog* para conectar o código MD5 (em FPGA) com o ARM. Essa lógica divide a transferência entre FPGA e ARM em transferências de 32 *bits*, até que os 512 *bits* sejam transferidos (ver Código 5.10). Assim, os sinais de entrada *in_wdata* e *in_addr* e sinal de saída *out_rdata* são mapeados nos registradores ARM e podem ser facilmente acessados a partir da camada de *software*.

Código 5.10: Bloco *Verilog* MD5: um *wrapper* para conectar MD5 ao ARM.

```
module md5_wr ( clk ,
                reset ,
                in_wdata ,
                in_addr ,
                out_rdata ) ;

//Define Input/Output
input wire         clk ;
input wire         reset ;
input wire [31:0]  in_wdata ;
input wire [63:0]  in_addr ;
output reg [31:0]  out_rdata ;
...
```

O *wrapper* recebe dados do ARM através da *in_wdata* e os armazena em um banco de registradores no endereço fornecido por *in_addr*. Para ler o resultado da conta do MD5, é necessário esperar 63 pulsos de *clock* e, em seguida, configurar *in_addr* para o endereço que contém os resultados e ler *out_rdata*. Para ler e escrever são usadas palavras de 32 *bits*, enquanto para endereçamento são de 64 *bits*.

Os blocos de 0 a 15 (cada com 4 *bytes*) são usados para transmitir partes da mensagem. Após enviar a mensagem completamente, o *bit* menos significativo do bloco 16 é configurado para ativar o bloco MD5 disponível. Assim, após 64 ciclos de *clock*, o resultado é armazenado nos blocos de 32 a 35. Finalmente, o *in_addr* é configurado para indicar o endereço dos registradores e seus valores são retornados via *out_rdata*.

5.3.5.1 Federados baseados no OpenCL

Nesta implementação, dois federados são baseados em OpenCL, o *multi-core Federate* e o GPU *Federate*. Ambos trabalham de maneira semelhante. Eles recebem todos os dados da imagem do *Sender Federate* da mesma forma que os outros federados, então eles constroem a matriz da imagem na memória do dispositivo e um *kernel* OpenCL é inicializado. Então, eles executam uma operação de máscara na imagem. Consiste em recalcular todos os *pixels* da imagem aplicando a equação 5.1.

$$
\begin{aligned}
I(i,j) = 5 * I(i,j) - [I(i-1,j) + I(i+1,j) + \\
I(i,j-1) + I(i,j+1)] \Leftrightarrow I(i,j) * M,
\end{aligned}
\tag{5.1}
$$

$$
I(i,j) * \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}
\tag{5.2}
$$

A equação 5.1 foi obtida pela multiplicação de cada elemento da imagem pela matriz da máscara, conforme mostrado na equação 5.2. Esse cálculo ajusta cada valor de *pixel* com base na influência que os *pixels* atuais e vizinhos têm.

Para alcançar uma portabilidade satisfatória do desempenho do *kernel* entre os diversos dispositivos de *hardware*, alguns cálculos foram realizados para ajustar o tempo de execução do OpenCL. Como o trabalho de Stone, Gohara e Shi[55] também descreve, ao usar o OpenCL, o cálculo do número de grupos de trabalho [56]. A configuração de alguns parâmetros é necessária para ser possível rodar um *kernel* como, por exemplo, a quantia de *threads* que irão executar o programa, então a configuração de como essas *threads* estarão organizadas em grupos. Essas configurações mencionadas são necessárias, conforme o *hardware* será executado se pode mudar a configuração, a quantidade de *kernels* tende a variar bastante [55]. Para esta implementação, foi considerado o número de elementos a serem calculados com base na resolução da imagem, no número de núcleos da arquitetura atual e no número de unidades computacionais do processador. Esses dois últimos parâmetros são baseados nos valores retornados por algumas funções do OpenCL apropriadas para consultar atributos de *hardware*.

Dado o baixo grau de complexidade do *kernel* neste experimento, apenas esta informação é necessária para calcular o número necessário de *threads*. Dividindo-os em grupos de trabalho com quantidades apropriadas conforme o número de unidades computacionais.

Sobre o experimento envolvendo *multi-core Federate* e GPU *Federate*, as seguintes etapas

foram executadas:

1. O *Sender Federate* lê uma imagem e mostra na tela;
2. O *Sender Federate* envia a imagem para o Federado *multi-core*/GPU;
3. O *multi-core*/GPU recebe a imagem e a exibe na tela para uma finalidade de verificação visual da integridade;
4. O *multi-core*/GPU executa o processamento do *kernel OpenCL*;
5. O *multi-core*/GPU mostra a imagem processada resultante;
6. O *multi-core*/GPU envia uma resposta ao *Sender Federate*;
7. O *Sender Federate* recebe a resposta e a exibe na tela.

5.3.6 Resultados

Aqui os resultados são apresentados separados em uma análise da transferência de dados com base nos cenários explicados nas seções anteriores.

5.3.6.1 Análise de troca de dados

A tabela 5.4 apresenta o tempo e uma taxa de transmissão para transmitir uma imagem do *Sender Federate* para outros federados através de uma rede LAN *Ethernet*. A imagem da Lena utilizada nos experimentos é exibida na Figura 5.10, com uma resolução de 512x512 (corresponde a uma matriz de 786.432 elementos *char* não assinados).

Pode-se notar que o aumento do número de atributos de um-*pixel* para 15-*pixels* (Tabela 5.4, experimentos 1 e 2) por mensagem, respectivamente, considera um *speedup* de 9,5 vezes. Ao transferir 100-*pixels* por mensagem (Tabela 5.4, experimento 3), o *speedup* foi de 16,5 vezes, demonstrando um aumento suave. Os maiores *speedups* foram alcançados quando os múltiplos dados foram encapsulados em um único atributo de *array*, atingindo *speedups* de 20 e 317,7 vezes (Tabela 5.4, experimentos 4 e 5), demonstrando um aumento exponencial. O *speedup* é tratado comparando as diferentes configurações, isso fornece dados para comparações que auxiliam na escolha de qual configuração HLA é mais apropriada

ao usar o *Virtual Bus*. Nesses experimentos, foram demostrados que a abordagem mais apropriada é transferir vários dados em um único tipo de *array*, como nos experimentos 4 e 5.

Figura 5.10: A imagem de Lena usada nos experimentos.

Fonte: Adaptado de Gonzalez e Woods[57].

Tabela 5.4: Tempos de transferência da imagem de Lena com 786.432 elementos.

Linha	Experimentos	Tempo	*Throughput* avg.	*Speedup*
1	One-by-one	72 s	87 Kbps	–
2	15-pixels	7.8 s	800 Kbps	9.5 X
3	100-pixels	4.4 s	1.4 Mbps	16.5 X
4	100-px/one attrib.	1.7 s	3.7 Mbps	20.0 X
5	300-px/one attrib.	234 ms	27 Mbps	317.7 X

É importante observar que essa média da taxa de transferência (do inglês *throughput*) é baseada apenas nos dados de imagem enviados e recebidos, não incluindo o tráfego das mensagens de controle enviadas pelo *gateway* do RTI global (RTIG). A primeira linha contém os valores do experimento *one-by-one*), a linha 2 e a linha 3 referem-se aos resultados nos experimentos *multi-pixel*, em que há um atributo para cada elemento a ser enviado. Por fim, as linhas 4 e 5 são os resultados para vários *pixels* em um atributo, que envia vários *pixels* em um único atributo.

Comparando as linhas 3 e 4, o mesmo número de *pixels* por mensagem foi enviado, mas com diferentes estratégias de transmissão. Neste caso ocorreu uma redução de tempo e um

aumento na taxa de transferência quando mais *pixels* foram transmitidos por mensagem.

Com o OpenCL foi possível implementar o projeto como um componente de simulação que permite o uso de *hardware* heterogêneo, o que possibilita o uso de ambas as CPUs com suporte total a *multi-core*, como a GPU. Também foi possível fazer um gerenciamento adaptativo do número de grupos de trabalho. Esse número é calculado dinamicamente conforme a resolução da imagem e com o número de núcleos, entre outros recursos específicos do dispositivo. Este cálculo possibilitou alcançar um melhor resultado no processamento remoto das imagens.

De acordo com estes tempos de processamento e dados da Tabela 5.4, o tempo de transferência nessa experiência é muito semelhante ao tempo de execução do *kernel*. No entanto, este *kernel* executado na plataforma OpenCL foi bastante simples e tem baixo custo computacional.

O aumento do número de atributos de um a quinze *pixels* por mensagem, representados respectivamente pela linha 1 e pela linha 2 na Tabela 5.4, permitiu reduzir o tempo de transmissão da imagem em 10% do tempo inicial. Em seguida, o tempo foi reduzido para 5,2% em comparação com cem *pixels* por mensagem, ao transmitir novecentos *pixels*, respectivamente linha 5 e 3.

Tabela 5.5: Tempo de processamento e tempo total da imagem de Lena com 786.432 elementos.

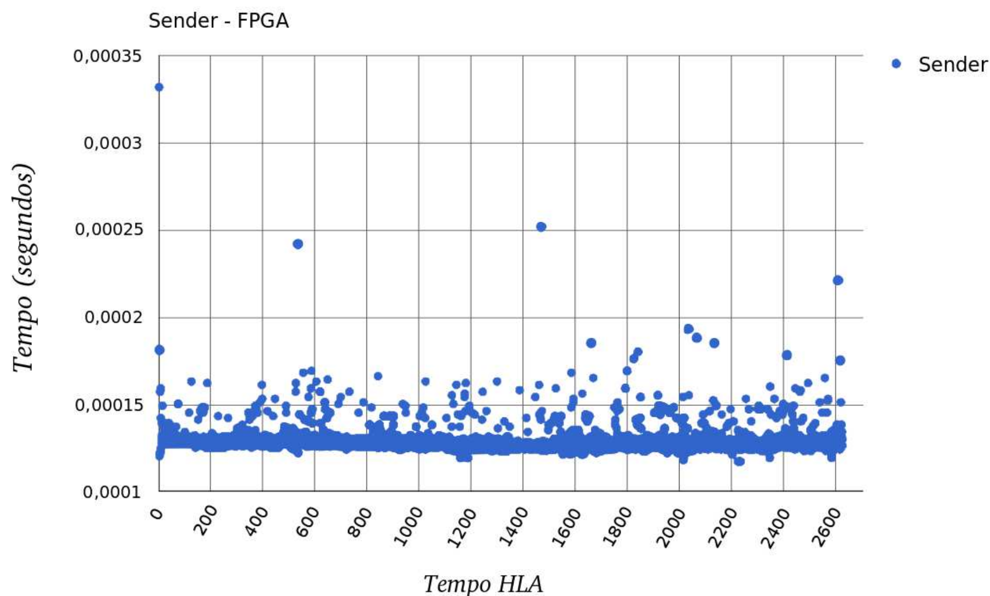
Linha	Experimento	Tempo de processamento	Tempo total
1	One-by-one	281 ms	72 s
2	15-pixels	281 ms	7.8 s
3	100-pixels	281 ms	4.4 s
4	100-px/one attrib.	281 ms	1.7 s
5	300-px/one attrib.	281 ms	515 ms

A Tabela 5.5 mostra o tempo para processar a operação da máscara sobre a imagem Lena pelo *kernel* OpenCL em um federado da GPU. Para todos os experimentos, o tempo de processamento foi o mesmo, 281 milissegundos para processar a máscara, por esse tempo ser independente da estratégia de transmissão. Essa tabela demonstra que a transferência de dados é o maior gargalo. Embora, os resultados demonstraram que ao transmitir 300 *pixels* por atributo, a transmissão diminui para 45% do tempo total.

5.3.6.2 Cenário 1: SoC (ARM + FPGA)

Nas Figuras 5.11 e 5.12 são mostradas as atividades de processamento do *Sender* e da FPGA. O tempo é representado como uma abstração do tempo em termos do tempo próprio do HLA, em relação ao processamento é dado em segundos, representando o tempo que o federado leva para processar os dados recebidos. Ambos os gráficos têm a mesma forma e a média é de 129μs para cada um. Isso ocorre porque a FPGA retorna os resultados dos *hashes* no próximo tempo de HLA que o remetente envia os dados de entrada. Após receber os últimos dados a serem processados, a FPGA é o único dispositivo a transmitir via *Virtual Bus*.

Figura 5.11: Atividade *Sender Federate* durante a transição para o SoC *Federate*.

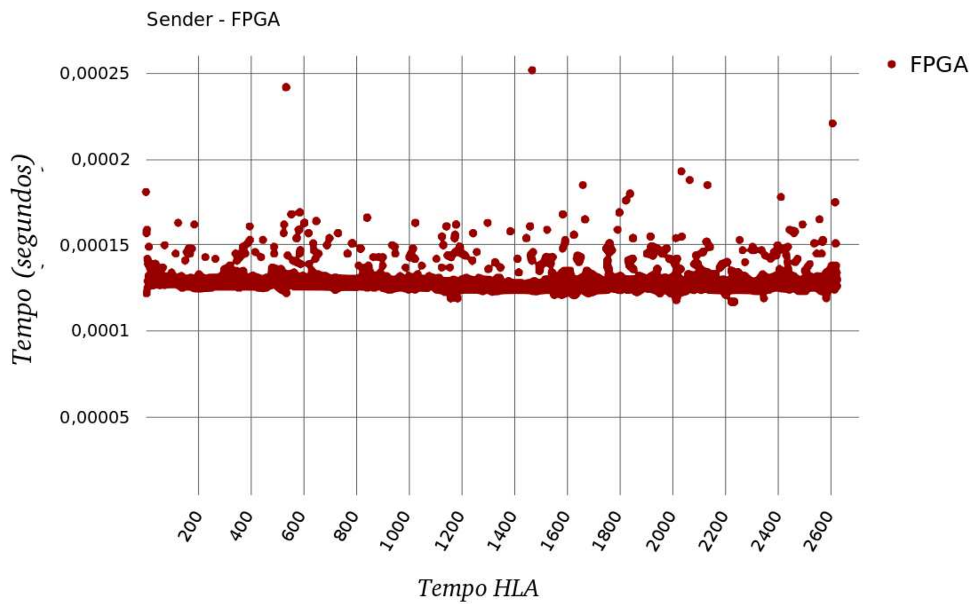


Fonte: próprio autor.

5.3.6.3 Cenário 2: vários *cores*

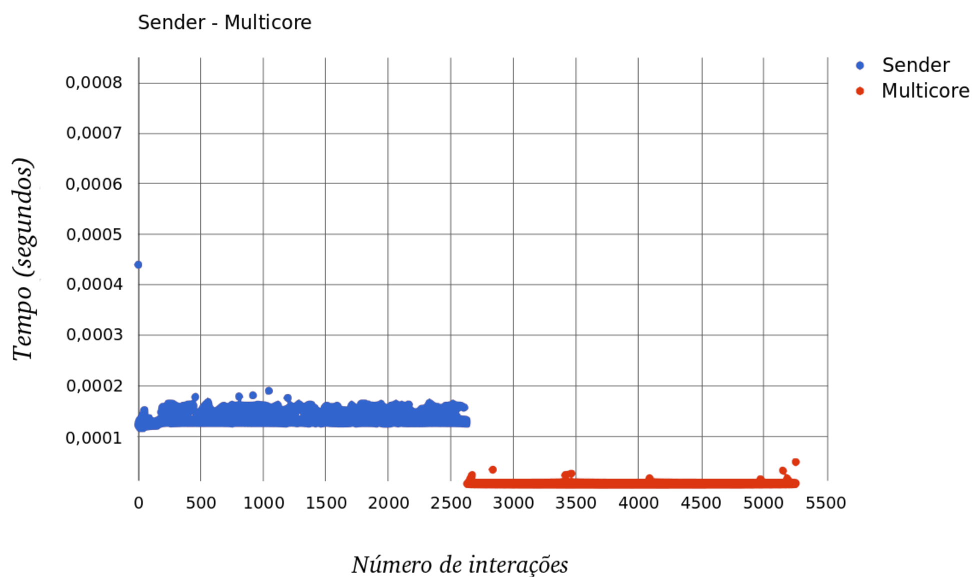
A Figura 5.13 ilustra a comunicação entre *Sender* e *multi-core*. O *Sender* leva em média 131μs para enviar cada mensagem ao *multi-core Federate*, e demora em média 6μs para devolver cada resultado. Após receber todos os dados, o *multi-core Federate* recebe 811μs para aplicar a máscara e retornar a primeira mensagem ao *Sender Federate*, o que ocorre em torno de 2600 interações.

Figura 5.12: Atividade de FPGA (no SoC *federate*) durante a comunicação com o *Sender Federate*.



Fonte: próprio autor.

Figura 5.13: Atividade de transmissão entre *Sender* e *multi-core*.



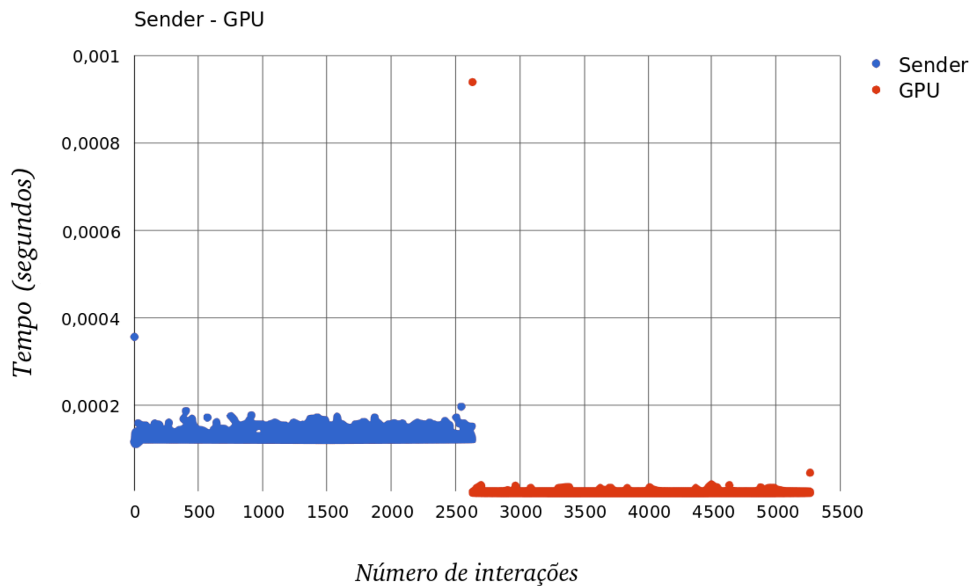
Fonte: próprio autor.

5.3.6.4 Cenário 3: GPU

A Figura 5.14 ilustra a atividade durante a comunicação entre os federados *Sender* e GPU. Os resultados são semelhantes à comunicação entre federados *Sender* e *multi-core*. A

única diferença é que a GPU leva 940μs para aplicar o filtro e retornar a primeira mensagem ao *Sender*. O foco é a comunicação entre dispositivos heterogêneos, então o código não foi otimizado para a GPU, resultando nessa discrepância.

Figura 5.14: Atividade de transmissão entre *Sender* e GPU.



Fonte: próprio autor.

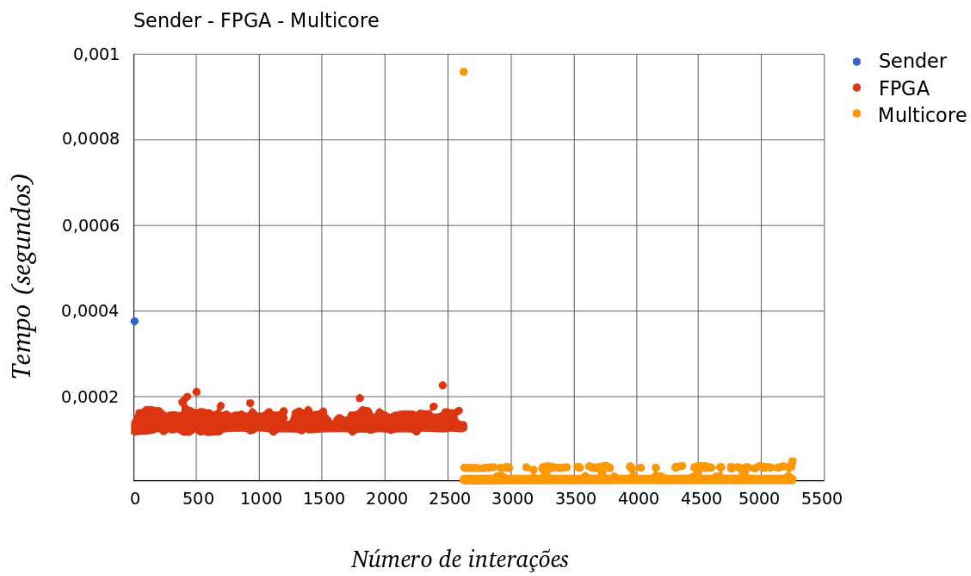
6.3.6.5 Cenário 4: *multi-core* e SoC (ARM + FPGA)

Para avaliar três dispositivos conectados via *Virtual Bus*, o *Sender*, FPGA e *multi-core* foram conectados. A Figura 5.15 ilustra a atividade durante esta comunicação. A integração desses três dispositivos não interferiu nos resultados obtidos quando dois dispositivos estavam trocando dados. Na Figura 5.15 ilustra a atividade da FPGA que sobrepõe o *Sender* como descrito anteriormente.

5.3.6.6 Cenário 5: SoC (ARM + FPGA) e GPU

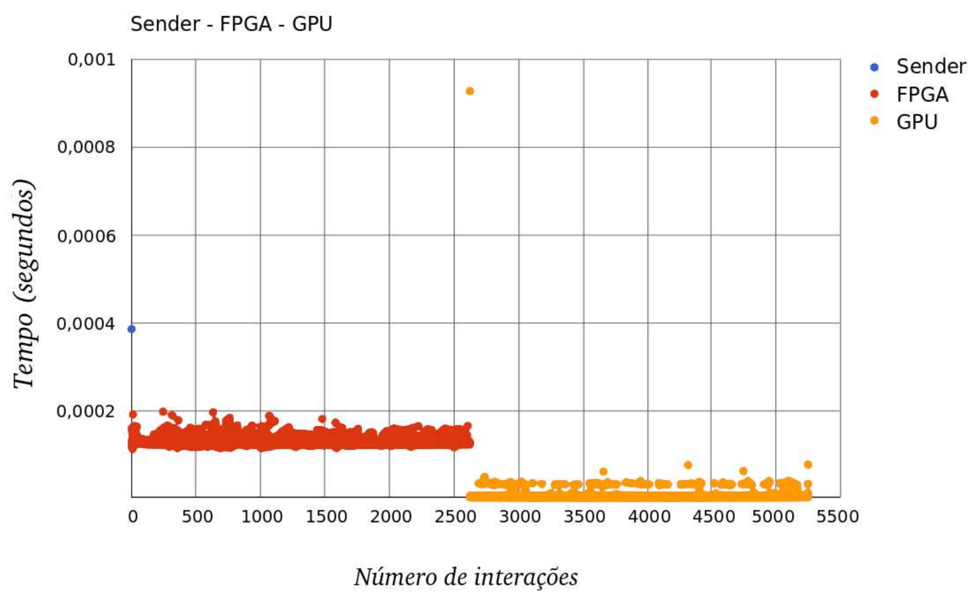
O mesmo experimento foi substituído o *multi-core* por GPU, e o resultado foi repetido como visto na Figura 5.16. Há uma semelhança entre este resultado e o cenário 4, dado que ambos usam o mesmo código OpenCL, e o gargalo de comunicação contínua na mesma quantidade.

Figura 5.15: Atividade de transmissão entre *Sender*, FPGA e *multi-core*.



Fonte: próprio autor.

Figura 5.16: Atividade de transmissão entre *Sender*, FPGA e GPU.



Fonte: próprio autor.

5.4 Considerações finais

Neste capítulo apresentou-se a viabilidade de utilizar a interface *Virtual Bus* para realizar transferências de dados por meio de funções implementadas a partir da especificação HLA, permitindo uma comunicação transparente para o usuário, assim preservando a con-

sistência de tempo e conteúdo das mensagens, além de viabilizar a infraestrutura necessária para o processamento paralelo em cada dispositivo conectado. Nos experimentos para suportar o processamento paralelo massivo de imagens, um federado OpenCL foi desenvolvido para gerenciar várias unidades de computação em GPU e CPU *multi-core*. Além disso, os experimentos demonstraram a comunicação entre diferentes dispositivos usando *Virtual Bus*. Alguns dispositivos diferentes foram integrados em um ambiente de execução único, eles são: PC, DE1-SoC com ARM e FPGA Altera, GPU e processador *multi-core*. Visto que o *Virtual Bus* foi implementado nos dispositivos, a comunicação e a sincronização entre eles eram transparentes e apenas três funções eram necessárias para qualquer aplicação lidar com a comunicação.

Capítulo 6

Ambiente de verificação funcional distribuída e heterogênea

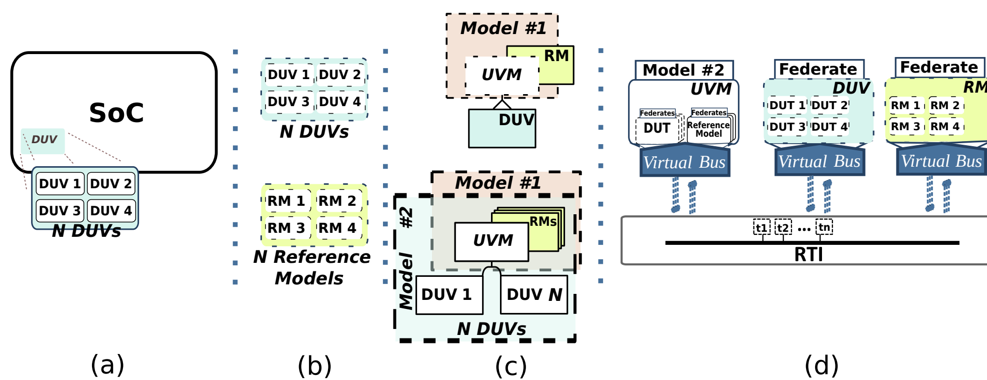
Neste capítulo trata-se a proposta de verificação funcional distribuída com sistemas heterogêneas, em que etapas são inseridas ao fluxo de projeto de *hardware* convencional. Organizou-se a apresentação deste capítulo da seguinte forma: na Seção 6.1 detalham-se as etapas necessárias para desenvolver o ambiente proposto de verificação funcional distribuída; na Seção 6.2 apresenta-se a estrutura do ambiente de verificação funcional com a implementação da interface de comunicação para diferentes aplicações desenvolvidas em arquiteturas heterogêneos, validando assim o método proposto; na Seção 6.3 abordam-se os experimentos aplicando o método proposto em diferentes cenários e, em seguida, avaliam-se os resultados para questões de processamento e funcionamento da sincronização dos integrantes durante a simulação; na Seção 6.4 apresenta-se uma comparação entre as etapas realizadas no ambiente de verificação convencional e as etapas no ambiente proposto; na Seção 6.5 abordam-se as considerações deste capítulo.

6.1 Etapas abordadas para verificação funcional distribuída

A Figura 6.1 ilustra uma visão geral das etapas do método proposto de verificação funcional para *open source* IP-*cores*, considerando projetos de sistemas heterogêneos para

SoC. O *testbench* é o federado responsável por conectar os testes com os demais federados, enviando os vetores de testes para todos os federados, assim como receber os dados resultados e, por fim, compará-los. Os outros federados têm implementações que fazem parte de um projeto heterogêneo, sendo: CPU, FPGA e GPU.

Figura 6.1: Etapas de validação da abordagem de verificação funcional distribuída proposta. (a) Especificação dos componentes de teste. (b) Desenvolvimento do modelo de teste e do modelo ideal. (c) Adaptação de um modelo de verificação funcional tradicional para um modelo distribuído. (d) Simulação e teste com o ambiente para verificação funcional distribuída e heterogêneo.



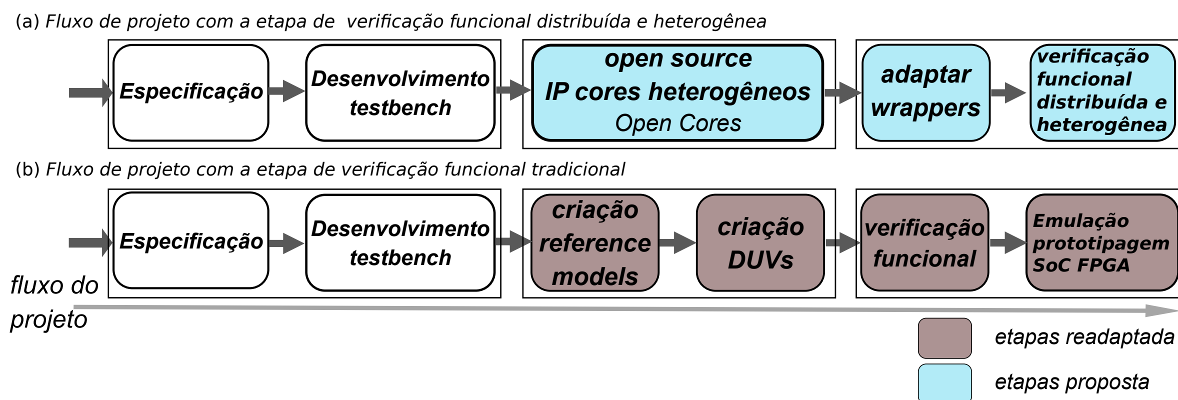
Fonte: próprio autor.

Para realizar os testes foram seguidas as etapas do fluxo de projeto de sistemas heterogêneos. A Figura 6.1(a) ilustra partes implementadas que venham a compor um SoC, dividida em N subsistemas que compõem um DUV, essas partes muitas vezes podem estar disponíveis sem a necessidade de re-codificação, mas em alguns casos empresas compram IP-*cores* para serem adotados no projeto. A Figura 6.1(b) mostra que as partes que devem ser integradas no projeto sejam disponíveis ou implementadas, devem ter suas implementações dos *reference models* (RMs), respectivamente. A Figura 6.1(c) é a etapa de desenvolvimento do ambiente de verificação funcional com base na metodologia de verificação UVM, que permite integrar os módulos desenvolvidos, no caso do *model* #1 necessita de re-codificação, pois o ambiente não suporta as diferentes interfaces dos subsistemas que compõem o DUV, no caso do *model* #2 é necessário implementar *wrappers* de comunicação interno e externo ao *testbench*. A Figura 6.1(d) mostra a etapa de simulação e teste com o ambiente de verificação funcional integrando e os diferentes subsistemas implementando os *wrappers* de comunicação utilizando na comunicação a interface *Virtual Bus*.

A partir da visão geral são adaptadas etapas ao fluxo de projeto. A Figura 6.2 ilustra os

dois fluxos de projeto: o tradicional (ver Figura 6.2(b)) e o proposto (ver Figura 6.2(a)). As figuras mostram as mudanças necessárias ao fluxo de projeto com os passos que o projetista possa seguir. Na Figura 6.2(b) mostra as etapas que devem ser excluídas da nova abordagem e as que devem ser readaptadas para formarem as novas etapas propostas. Na Figura 6.2(a) são inseridas novas etapas como utilizar *open source* IP-*cores*, adaptar *wrappers* para integração dos IPs. Então são necessárias algumas modificações no fluxo de projeto da Figura 6.2(b), para que o projeto siga o fluxo com as etapas de verificação funcional distribuída.

Figura 6.2: Etapas necessárias para adaptar a abordagem proposta.

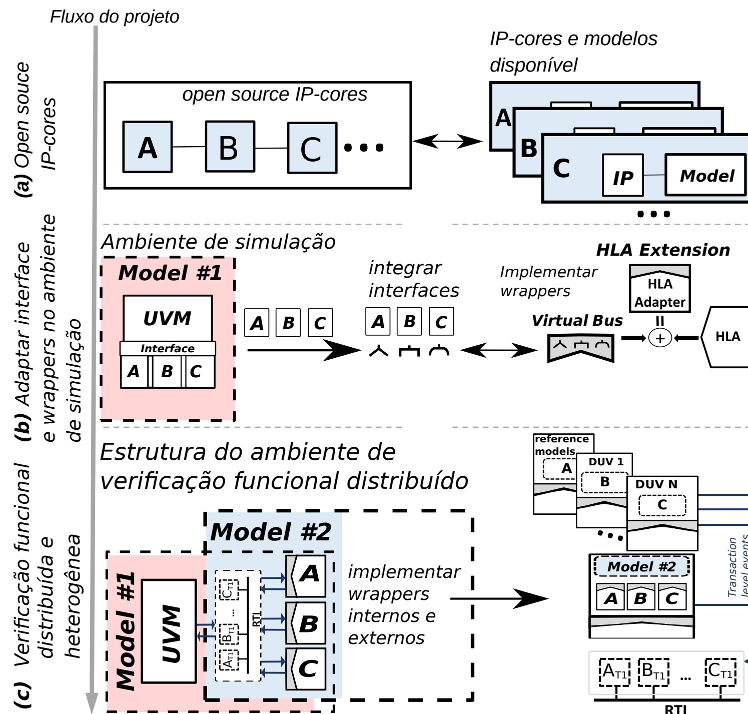


Fonte: próprio autor.

A proposta do método de verificação funcional distribuída de sistemas heterogêneos permite integrar e verificar *open source* IP-*cores* sem a necessidade de re-codificação para realizar a etapa de verificação no fluxo de projeto. Este ponto do trabalho apresenta uma abordagem geral para verificação funcional distribuída para alcançar a extensibilidade e a integração de IPs. Para aplicar o método é necessário passar por algumas etapas, conforme ilustrado na Figura 6.3. Primeiramente, o fluxo do projeto começa na especificação, passando pelo desenvolvimento do RTL até chegar na verificação de cada RTL. Nesse caso, são escolhidas as implementações disponíveis que atendem as necessidades do projeto, tais como IP-*cores* e modelos. Depois, desenvolve-se o ambiente de simulação proposto em cima da metodologia UVM, e nele são implementados componentes *wrappers* para cada IP, respectivamente. Em seguida, adapta-se a biblioteca dos dispositivos heterogêneos para realizar a integração com o ambiente de simulação. Esse processo com as etapas desenvolvidas, do método de verificação funcional distribuída e heterogênea, é descrito nas seguidas etapas:

(a) Especificação e desenvolvimento: na primeira etapa são especificados os subsistemas

Figura 6.3: Adaptação do ambiente de verificação do projeto para verificação funcional distribuída.



Fonte: próprio autor

necessários para o desenvolvimento de um sistema heterogêneo completo, essas especificações são descritas em um documento. A Figura 6.3(a) ilustra as etapas da escolha dos subsistemas que podem estar disponíveis ou não, considerando seus respectivos modelos;

(b) Desenvolvimento do *testbench*: a segunda etapa estabelece diversos parâmetros funcionais do projeto, alguns desses parâmetros são definidos como protocolos de comunicação entre *open source* IP-*cores* e o modelo de verificação funcional proposto, dentre outros fatores. É importante mencionar que cada caso de uso necessita de uma comunicação entre os diferentes IP-*cores*. Por isso, é de suma importância também que, durante a especificação, as conexões dos *open source* IP-*cores* sejam bem definidas para ocorrer uma verificação do SoC de forma coesa ao nível de sistema. A Figura 6.3(b) ilustra o modelo e a linguagem proposta para o ambiente de verificação funcional distribuída, em que foi construída com base na metodologia UVM, utilizando a biblioteca *SystemC*, além de mostrar a biblioteca de baixo custo desenvolvida

para facilitar a troca de mensagens entre diferentes interfaces. Assim como, reduz a sobrecarga de desenvolvimento dos projetistas;

(c) *open source* IP-*cores* heterogêneos (*open cores*): são utilizados IP-*cores* implementados ou disponíveis de código aberto, ou fechado em sistemas heterogêneos;

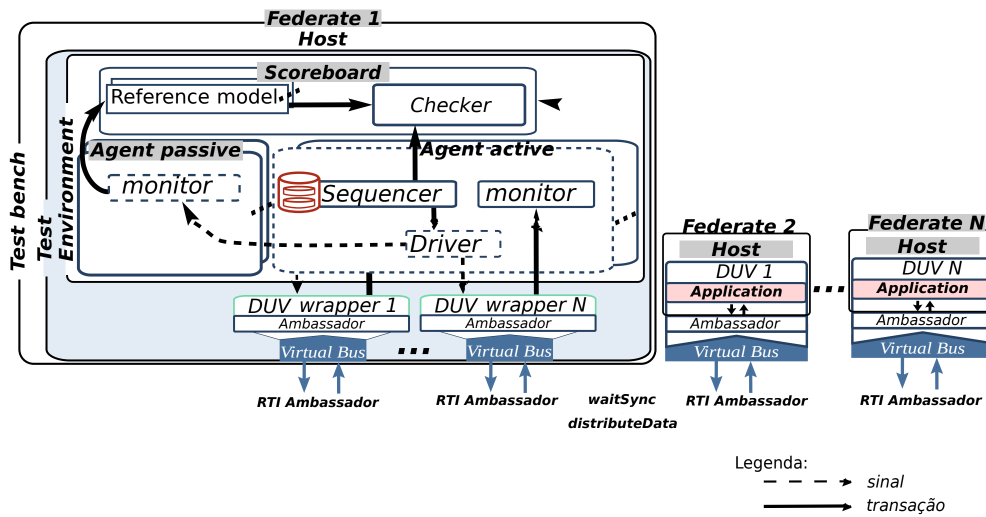
(d) Adaptar *wrappers*: nos *open source* IP-*cores* são implementados *wrappers* da interface de comunicação, enquanto no *testbench* são implementados *wrappers* interno;

(e) Verificação funcional distribuída e heterogênea: a equipe envolvida na verificação dos IP-*cores* consegue integrá-los durante sua construção e, em seguida, verificar ao nível de sistema, assegurando que os IP-*cores* estejam funcionando em conjunto conforme o esperado. A restrição dos IP-*cores* entregues a uma equipe é relevante, pois essa entrega pode ser de IP-*cores* fechados, ou seja, entregue como código que não pode ser alterado, que no caso deste trabalho não é tratado, ou entregues como IP-*cores* de código aberto, que estejam como códigos disponíveis para alteração. A Figura 6.3(c) ilustra o ambiente de verificação funcional distribuída, o qual os blocos a serem trabalhados podem ser integrados e requer a implementação dos *wrappers* que instância a biblioteca desenvolvida do *Virtual Bus*/HLA, um dispositivo ao implementar um *wrapper* ele se torna um federado que pode utilizar os serviços do RTI para trocar mensagens dentro de uma federação, assim como realizar a sincronização com os demais membros, o *testbench* tem que implementar os *wrappers* para cada dispositivo, assim como gerenciar as mensagens enviadas e recebidas de diferentes subsistemas durante o processo de verificação funcional.

6.2 Ambiente de verificação funcional distribuído e heterogêneo proposto

A Figura 6.4 ilustra o ambiente de verificação proposto neste trabalho. Esse ambiente é composto pelos seguintes elementos: *sequencer*, *driver*, *monitor*, *checker*, *reference model*, DUV *wrapper* (dependendo do número de dispositivos será configurado o ambiente com a quantia necessária de implementações dos componentes).

Figura 6.4: Ambiente de verificação funcional distribuída e heterogênea.



Fonte: próprio autor.

A Tabela 6.1 descreve o caminho de envio e de recebimento dos dados durante a simulação, ocorrendo troca de dados entre os elementos do ambiente. Esses dados são um conjunto de transações $(t_1, ..., t_n)$ ou um conjunto de sinais $(s_1, ..., s_n)$. A explicação é dada para o *testbench* (*Federate 1*) com um dispositivo (*Federate 2*). Inicialmente, o *sequencer* gera um conjunto de transações enviadas para o *driver* (etapa #1). Essas transações são convertidas no *driver* para sinais, enviadas para o DUV através da comunicação dos *wrappers*, assim como enviada para o *monitor* (etapa #2). Depois a implementação no dispositivo a ser verificado envia e recebe os estímulos pelos *wrappers* interno e externo (etapas #3 #4 #5). Esses estímulos são enviados do *wrapper* para o DUV, via mensagens HLA (etapa #3). O resultado recebido pelo dispositivo é tratado pelo *testbench* (etapa #4). As mensagens recebidas pelo *testbench* são enviadas para o *monitor* que encaminha para o *reference model* e *checker* (etapa #6). O resultado processado pelo *reference model* é encaminhado para o *checker* (etapa #7). Por fim, realiza-se uma comparação do DUV com *reference Model* no *checker* ao receber os estímulos (etapa #8). Conforme esse processo é replicado para N dispositivos, é necessário implementar os respectivos N *wrappers* internos e externos para ocorrer a troca de mensagens, permitindo integrá-los na federação.

Após o arquivo *.fed* ser configurado para os N federados, a comunicação pelos *wrappers* com *Virtual Bus* é iniciada com o envio dos estímulos pelo (*Federate 1*), quando o valor de algum atributo, via serviço *Update Values*, for atualizado pelo ambiente em execução,

Tabela 6.1: Fluxo de dados entre os federados.

# etapa	origem			destino
1	Sequencer	$(t_1, ..., t_n)$	$\longmapsto$	Drive
2	Drive	$(s_1, ..., s_n)$	$\longmapsto$	DUV wrapper
		$(s_1, ..., s_n)$	$\longmapsto$	Monitor
3	DUV wrapper	$(s_1, ..., s_n)$	$\longmapsto$	DUV
4	DUV	$(s_1, ..., s_n)$	$\longmapsto$	DUV wrapper
5	DUV wrapper	$(s_{D1}, ..., s_{Dn})$	$\longmapsto$	Monitor
6	Monitor	$(t_{R1}, ..., t_{Rn})$	$\longmapsto$	Refmod
		$(t_{D1}, ..., t_{Dn})$	$\longmapsto$	Checker
7	Reference model	$(t_{R1}, ..., t_{Rn})$	$\longmapsto$	Checker
8	Checker		*(match)* or	*(mismatch)*

via serviço *Update Attribute*, em algum momento do tempo o (*Federate N*) receberá uma chamada de retorno relativa a essa atualização. Essa chamada retorno é implementada no Ambassador do federado, via serviço do *Reflect Attribute Values*, cujo objetivo é repassar os valores atualizados para RTI *Ambassador*. Para que o (*Federate 2*) possa receber os valores enviados pelo (*Federate 1*), deve-se assinar previamente tais atributos da classe de objeto pela API do *Virtual Bus*. A interface realiza a troca de dados com a assinatura feita pelos serviços do *write* e *read*. Esses valores são sincronizados no RTI pelos serviços *waitSync* para esperar uma resposta de um federado em um determinado intervalo de tempo e, em seguida, repassar a mensagem pelo serviço *distributeData* para o federado destino. O rótulo do federado origem e destino é atribuído pelo atributo *id* implementado no *.fed*. Com a identificação, o processo de envio de um federado para outro é realizado de forma transparente para o usuário pela interface *Virtual Bus*. Nesse sentido, a troca de mensagens ocorre com o mesmo processo para qualquer federado na federação.

6.3 Resultados experimentais

Os experimentos foram montados com diferentes dispositivos implementando *wrappers*, isso permitiu considerar esses dispositivos como federados, sendo eles: PC (*testbench* e *reference model*), FPGA (DUV), e GPU (DUV). O *testbench* realiza a conexão dos *wrappers*

internos com os estímulos gerados, que enviam imagens para os dispositivos com *wrappers* externos, processando operações nos *open source* IP-*cores*, depois retorna o resultado de volta para comparação.

Os estímulos são processados e calculados em OpenCV a partir de uma imagem no *testbench*. OpenCV é usado neste trabalho apenas para o manuseio básico de *pixels* através de suas funções. Para realizar os testes de verificação, uma implementação do algoritmo da integral da imagem foi desenvolvida usando *SystemC*, que realiza os cálculos em uma matriz $4x4$ com 19 *clocks* de interação. Dessa forma, o ambiente gera um conjunto de 16 instâncias aleatórias e os envia para o DUV a cada 20 *clocks*. Um total de 10.000 transações são transmitidas e seus resultados são coletados e comparados com o *reference model*.

Os experimentos e resultados apresentados nesta seção referem-se a uma análise de processamento, leitura, escrita e tempo, considerando o ambiente de verificação funcional distribuída apresentada na Subseção 6.2.

6.3.1 Equipamentos

Os equipamentos utilizados para compor os cenários são: PC, FPGA e GPU. A configuração básica de cada um deles é descrita na Tabela 6.2. O PC é um computador de *desktop* que executa o Ubuntu 16.04 LTS responsável pelo *testbench* e *reference model*. A FPGA tem um *Cyclone* V SE SoC, que tem um FPGA *Cyclone* V integrado com um processador ARM *Cortex* A9 *dual-core* em um único *chip*, rodando o Ubuntu 12.04 LTS. A GPU é uma NVIDIA *Jetson* TX1 com uma CPU ARM de 64 bits integrada com 4 GB de memória DRAM compartilhada e estava executando o Ubuntu 16.04 LTS.

Tabela 6.2: Especificações dos equipamentos.

Dispositivos	**Configurações**
PC	Intel i3-4005U com 4GB DDR3
FPGA	Placa DE1-SoC da Terasic
GPU	NVIDIA *Jetson* TX1

6.3.2 Cenários

Os experimentos foram divididos em dois cenários de verificação, conforme é mostrado na Tabela 6.3, no primeiro os subsistemas são conectados em série (C_1), e no segundo, são conectados em paralelo (C_2). O *testbench* é utilizado na verificação do projeto de um sistema composto de dois subsistemas, um que efetua a conversão de RGB para YC_bC_r implementado em GPU (S_1) e outro que calcula a integral da imagem implementado em FPGA (S_2). Nesses dois cenários foram utilizados *open source* IP-*cores*, e os respectivos *wrappers* de comunicação externos foram implementados. Considerou-se que os "*golden models*" desses IP-*cores* estavam disponíveis, um referente a implementação da FPGA (RF_1) e outro referente a implementação da GPU (RF_2). No cenário 1 a saída gerada S_1 é enviada para S_2 que gera a saída enviada para comparação com a saída do *reference model* (RF_1+RF_2). No cenário 2 as saídas geradas por S_1 e S_2 são comparadas com os respectivos (RF_1) e (RF_2).

Tabela 6.3: Cenários usados nas experiências.

	testbench	S_1	S_2	RF_1	RF_2	RF_1+RF_2
C_1	X	X	X	–	–	X
C_2	X	X	X	X	X	–

6.3.3 Estudos de caso

Nesta subseção são detalhados os estudos de caso empregando o método propostos, aplicado nos cenários com os subsistemas conectados em série e paralelo. Os subsistemas foram implementados em FPGA e GPU separadamente e depois tratados nos cenários propostos. Este trabalho realizou os experimentos com os subsistemas de código aberto implementando *wrappers* internos ao *testbench* e externos (ambos integrando o *Virtual Bus*/HLA). Os dispositivos executam o sistema operacional Linux para configuração da simulação.

6.3.3.1 Desenvolvimento do subsistema S_2 na FPGA

Especificação da integral da imagem

Primeiro é realizada a verificação funcional da implementação da integral da imagem na arquitetura heterogênea FPGA. A integral da imagem é uma técnica desenvolvida para

calcular as características em imagens digitais. Essa técnica foi introduzida por Crow[58] para acelerar o cálculo do mapeamento de texturas e ganhou destaque com a sua aplicação na detecção de faces usando o algoritmo de Viola Jones [59].

O desenvolvimento do código se baseia nos seguintes conceitos matemáticos, tal que a equação 6.1 ilustra a fórmula para calcular a integral da imagem $II(x, y)$ de uma região no ponto (x, y), realizando a soma de todos os *pixels* acima e à esquerda de (x, y) incluindo $i(x, y)$, em que $i(x, y)$ é o valor do *pixel* na posição (x, y) [58].

$$II(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y') \tag{6.1}$$

O cálculo da integral da imagem tem a possibilidade de ser realizado recursivamente, como pode ser visto na equação 6.2.

$$II(x, y) = I(x, y) - II(x - 1, y - 1) + II(x, y - 1) + II(x - 1, y) \tag{6.2}$$

Para realizar a implementação no *hardware*, a equação de definição é baseada como uma convolução da matriz S com a imagem I, conforme mostrado na equação 6.3. A integral da imagem utiliza o processo de cálculo semelhante ao Filtro Sobel, com uma estrutura de janela deslizante com uma matriz de tamanho $2x2$, assim o *buffer* circular precisa armazenar apenas uma linha da imagem.

$$II = S * I = \begin{bmatrix} -1 & +1 \\ +1 & +1 \end{bmatrix} * I \tag{6.3}$$

Implementação concebida para *reference model* da FPGA

Nesta etapa do projeto o *reference model* é concebido como base para verificação da implementação na FPGA, conforme mostrado no Código 6.1. A implementação do *open core* da aplicação é disponível em Campos[60]. O código desenvolvido implementa o *wrapper* externo que comunica com um interno ao *testbench*, ambos estão executando na mesma máquina em processos separados, mas podem ser simulados em máquinas fisicamente sepa-

radas. Quando um *wrapper* é desenvolvido ele pode ser reutilizado novamente em um mesmo sistema heterogêneo, então a implementação do *wrapper* é armazenado para uso futuro, esse conjunto de *wrappers* reutilizáveis compõem uma biblioteca de *wrappers* reaproveitados para diferentes sistemas. Com o *reference model* disponível é possível reutilizá-lo no processo de verificação para testar o componente em teste (DUV). O *reference model* recebe os *pixels* de uma imagem no processo para realizar a integral da imagem, realiza o cálculo necessário e envia 16 valores referentes a largura de transmissão definida no *Virtual Bus* (Código 6.1, linha 22–29).

Código 6.1: Implementação do *reference model* como modelo base. Adaptado de Campos[60]

```
1  int main( int argc, char **argv ){
2  char federateName[20];
3  sprintf(federateName, "%d", 1);
4  // create and run the federate
5  VirtualBusFederate *federate;
6  federate = new VirtualBusFederate();
7  federate->runFederate( federateName );
8  //Data to receive
9  unsigned src, addr, readsrc, cntrl_write, a[IMAGE_HEIGHT][IMAGE_WIDTH];
10 while(1){
11 readsrc = 3;
12 if(federate->readData(readsrc, addr, size, data)){
13     if(readsrc == 3){
14         std::cout << ``data from src'' << readsrc << ``:'';
15         for (size_t i = 0; i < VIRTUALBUS_SIZE; i++){
16             std::cout << `` '' << data[i];
17         }
18         std::cout << std::endl;
19         data[aux] << std::endl;
20         //send data by HLA
21         addr = SENDER_ID; //Sender address
22         a[0][0] = data[0] ; a[0][1] = data[1] ;
23         a[0][2] = data[2] ; a[0][3] = data[3] ;
24         a[1][0] = data[4] ; a[1][1] = data[5] ;
25         a[1][2] = data[6] ; a[1][3] = data[7] ;
```

```
26          a[2][0] = data[8] ; a[2][1] = data[9] ;
27          a[2][2] = data[10]; a[2][3] = data[11];
28          a[3][0] = data[12]; a[3][1] = data[13];
29          a[3][2] = data[14]; a[3][3] = data[15];
30          //First compute the first horizontal line
31          for(int j = 1; j < IMAGE_WIDTH; j++)
32              a[0][j] += a[0][j-1];
33          //Then compute the first vertical line
34          for(int i = 1; i < IMAGE_HEIGHT; i++)
35              a[i][0] += a[i-1][0];
36          //And finally compute the rest of the values
37          for(int i = 1; i < IMAGE_HEIGHT; i++)
38              for(int j = 1; j < IMAGE_WIDTH; j++)
39                  a[i][j] += a[i-1][j]+a[i][j-1]-a[i-1][j-1];
40
41          data[0]  = a[0][0]; data[1]  = a[0][1];
42          data[2]  = a[0][2]; data[3]  = a[0][3];
43          data[4]  = a[1][0]; data[5]  = a[1][1];
44          data[6]  = a[1][2]; data[7]  = a[1][3];
45          data[8]  = a[2][0]; data[9]  = a[2][1];
46          data[10] = a[2][2]; data[11] = a[2][3];
47          data[12] = a[3][0]; data[13] = a[3][1];
48          data[14] = a[3][2]; data[15] = a[3][3];
49          src = 1;
50          federate->writeData(src, 0, size, data);
51          std::cout << std::endl; } }
52      federate->advanceTime(1.0); } }
```

Implementação do subsistema S_2 em *hardware*

Nesta etapa foi utilizado a implementação da integral da imagem em linguagem de descrição de *hardware* [60], conforme é descrito no Código 6.2. A implementação do subsistema S_2 executa em uma plataforma de prototipação em FPGA. Essa plataforma de desenvolvimento tem uma estrutura composta por um kit de desenvolvimento em FPGA DE1-SoC da Terasic TERASIC[61], que contém cerca de 85 mil elementos lógicos acoplados a um processador ARM *dual-core* de 800M Hz [61].

Código 6.2: Implementação *SystemVerilog* da integral da imagem. Adaptado de Campos[60]

```
1  module integral_image #(parameter W = 8,parameter BUFFER_SIZE = 2,
     parameter ROW_SIZE = 4, parameter W_SUM = $clog2(ROW_SIZE*ROW_SIZE
     *(1<<W)))(input logic clock, reset, input logic [W-1:0]new_sample,
     output logic [W_SUM-1:0] S);
2    logic [W_SUM-1:0] s[BUFFER_SIZE][BUFFER_SIZE];
3    logic [W-1:0] prevS;
4    logic [W_SUM-1:0] buffer[ROW_SIZE];
5    logic [$clog2(ROW_SIZE)-1:0] ptr, next_ptr;
6    logic [$clog2(ROW_SIZE):0] row_counter;
7    always_comb begin
8      if(reset) S <= 0;
9      else S <= prevS+s[1][0]+s[0][1]-s[0][0];
10   end
11   always_comb begin
12     if(ptr+1 < ROW_SIZE) next_ptr <= ptr+1;
13     else next_ptr <= 0;
14   end
15   always_comb prevS <= new_sample;
16   always_ff @(posedge clock) begin
17     if(reset)begin
18       ptr <= 0;
19       row_counter <= 0;
20       s[0][0] <= 0; s[0][1] <= 0; s[1][0] <= 0; s[1][1] <= 0;
21       for(int i = 0; i < ROW_SIZE; i++)
22         buffer[i] <= 0;
23     end
24     else begin
25       s[1][0] <= S;
26       buffer[ptr] <= S;
27       s[0][1] <= buffer[next_ptr];
28       s[0][0] <= s[0][1];
29           ptr <= next_ptr;
30       if (ptr == 0) begin
31         if(row_counter < ROW_SIZE)
32           row_counter <= row_counter+1;
```

```
33          else row_counter <= 0;
34
35          s[0][0] <= '0;    s[1][0] <= '0;
36        end
37      end
38    end
39    initial begin
40      $monitor(``dut => new_sample = %d prevS = %d s[1][0] = %d s[0][1] =
            %d s[0][0] = %d buffer[ptr] = %d S = %d at time = %d (ptr=%d,
          next_ptr = %d, row_counter = %d)'', new_sample, prevS, s[1][0],
           s[0][1], s[0][0], buffer[ptr], S, $time, ptr, next_ptr,
          row_counter);
41    end
42  endmodule: integral_image
```

Com o IP-*core* disponível do subsistema S_2, é implementado na plataforma de prototipação, sem a necessidade de re-codificação. Inicialmente, é necessário utilizar a ferramenta Altera *Qsys*, permitindo projetar as conexões do IP-*core* desenvolvido. O projeto é composto por um *clock* de 50 MHz, pelo processador ARM e pelo bloco para realizar a conexão com o bloco da integral da imagem, entre outras conexões que vão auxiliar o processo de envio do cálculo processado pela FPGA. As conexões no *Qsys* utilizam a interface *avalon*, para executar a escrita, a leitura e o endereçamento para calcular a integral da imagem. As conexões entre o ARM e as portas de dados acontecem pelo canal AXI *Master*, com a porta *h2f_lw_axi_master* no processador, enquanto os escravos com a porta *Avalon Memory Mapped Slave*. Essa porta do processador tem a função de ativar ou desativar a interface de comunicação *lightweight* do ARM para FPGA. Quando ativado permite uma transmissão de até 32 bits para *avalon_writedata* e *avalon_readdata*.

Adaptação dos *wrappers* para comunicação com S_2 na FPGA

Com o *reference model*, o *testbench* e o subsistema S_2 (DUV) disponíveis, é preciso resolver o problema de integração dos dispositivos com o ambiente de verificação funcional. Dessa forma, foram desenvolvidos *wrappers* para integrar *open source* IP-*cores* sem a necessidade de re-codificação, acelerando o processo de validação nos estágios iniciais do projeto,

antes que a arquitetura do sistema seja definida, permitindo explorar o conceito de reuso, avaliando se um *open source* IP-*cores* é utilizado na integração dos demais subsistemas.

O subsistema S_2 é implementado na FPGA, necessitando implementar *wrappers* que permitam a comunicação durante o processo de verificação funcional. O *wrapper* externo recebe os dados através da *avs_s0_writedata* e os armazena em um banco de registradores no endereço fornecido por *avs_s0_address*. Para ler o resultado da integral da imagem é configurado *avs_s0_address* para o endereço que contém os resultados e usado o *avs_s0_readdata* para ler o resultado. Para ler e escrever são usadas palavras de 32 *bits*, enquanto para endereçamento são de 6 *bits*. Essa lógica realiza a transferência entre FPGA e ARM (ver Código 6.3). Assim, os sinais de *avs_s0_writedata*, *avs_s0_address* e *avs_s0_readdata* são mapeados nos registradores ARM e podem ser facilmente acessados a partir da camada de *software*.

Código 6.3: Bloco em *SystemVerilog*: para conectar a aplicação da plataforma de prototipada ao ARM.

```
1  ...
2  module avs_wrapper ( //interface para clock e reset
3    input               csi_clk, input rsi_reset_n,
4    input        [5:0]  avs_s0_address,
5    input        [31:0] avs_s0_writedata,
6    output reg   [31:0] avs_s0_readdata );
7    // Para se comunicar com o DUT
8    logic enable_ii, valid_ii;
9    logic [W-1:0] X, Y;
10   logic [W_SUM-1:0] mem[ROW_SIZE][ROW_SIZE], sum_array[ROW_SIZE][
         ROW_SIZE];
11 ...
```

Para realizar a comunicação do subsistema S_2 (DUV) implementado na FPGA com o *testbench*, são necessários os *wrappers* para comunicação. Dessa forma, a comunicação usando o *Virtual Bus* deve acontecer entre a comunicação do *wrapper* no ARM com o *wrapper* criado no *testbench*, conforme descrito no Código 6.4. A conexão dos federados permite que a troca de mensagens seja transparente para o usuário, criando uma comunicação do DUV *wrapper* que utiliza as funções de leitura, escrita e avanço de tempo para sincronismos

na federação.

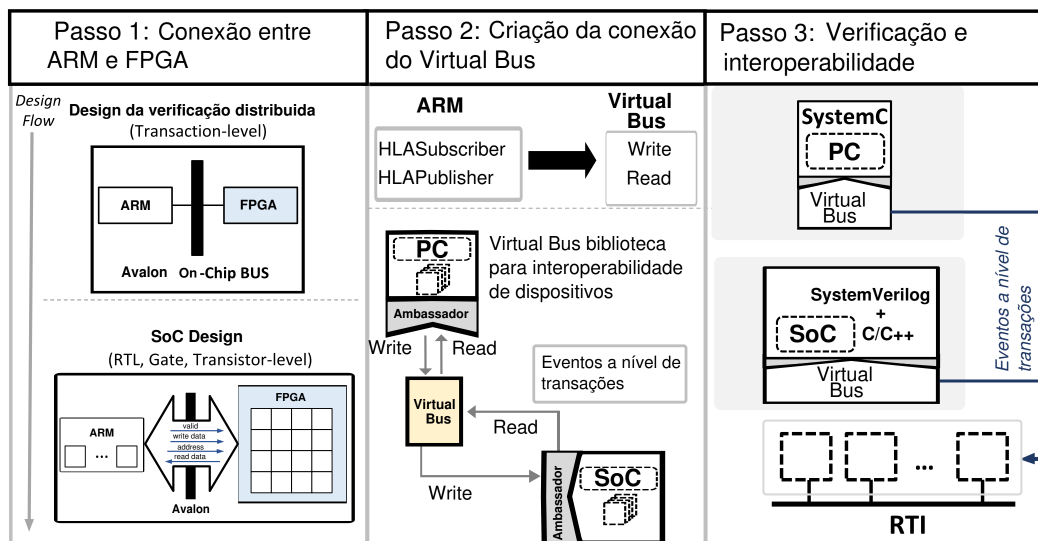
Código 6.4: Bloco *SystemC* no DUV *wrapper*: para conectar o *testbench* com a aplicação no dispositivo.

```
1  ...
2  void ii::func()
3  {
4      data_t temp_out_data;
5      bool  temp_carryout;
6      int temp_sel;
7      // reset
8      while (true)
9      {
10         if(reset_n.read() == false) {
11             out_data_valid.write(0);
12             for (int i = 0; i < DATA_SIZE; ++i) {
13                 out_data[i].write(0);
14             }
15         }
16         else if (in_data_en.read() == 1){
17             for (int i = 0; i < DATA_SIZE; ++i) {
18                 data[i] = in_data[i].read();
19 
20             }
21             for(int i =DATA_SIZE ; i<16; i++)
22                 data[i] = 9;
23 
24             src = 1;
25             addr = 1;
26             federate->writeData(src, addr, size,  data);
27             federate->advanceTime(1.0);
28             if(federate->readData(src,addr,size, data)){
29                 for (int i = 0; i < DATA_SIZE; ++i) {
30                     temp_out_data = data[i];
31                     out_data[i].write(temp_out_data);
32                 }
33                 out_data_valid.write(1);
```

```
34                }
35            }
36            wait(1);
37        }
38    }
39    ...
```

O fluxo seguido para implementar o ambiente de verificação funcional com as arquiteturas heterogêneas é ilustrado na Figura 6.5, dividido em três etapas. Na primeira etapa é realizada a conexão do ARM com a FPGA, através da criação de uma interface *Avalon*. A segunda etapa é realizada a conexão do SoC (ARM + FPGA) com o HLA utilizando o Virtual Bus. Por fim, a terceira etapa é realizada a simulação do ambiente de verificação com SoC, usando o RTI para a troca de transação dos elementos.

Figura 6.5: Fluxo de implementação para o ambiente de verificação.



Fonte: próprio autor

No primeiro passo, o ARM executa os códigos desenvolvidos em linguagem C++, não utilizando as bibliotecas de *SystemC*. Portanto, a comunicação do ARM com a FPGA é realizada por rotinas de leituras e escritas em registradores, como ocorre em comunicações com periféricos de um processador.

No lado da FPGA, uma interface é criada em *Verilog* conforme as especificações da interface de comunicação Avalon [62]. Dado que os dados são atualizados nos registradores, o subsistema implementado na FPGA pode acessar esses dados. Após o processamento, os

resultados são gravados novamente nos registradores e lidos pelo ARM.

Para o ambiente de verificação coletar os dados processados pela FPGA, é utilizado um *wrapper* externo, implementando a interface de comunicação *Virtual Bus*/HLA. Na comunicação são utilizadas três funções básicas como mostra a Tabela 5.1. A função *write* é responsável por enviar os dados para o RTI de um federado para outro, utilizando um *id* de identificação de destino. A função *read* retorna um valor booleano para indicar se algum dado foi recebido para aquele *id* do federado específico. Em caso afirmativo, os dados são devolvidos por referência. Por último, *advanceTime* executa a rotina que realiza o avanço de tempo do HLA.

Neste projeto a versão 3.5.1 do CERTI [52] foi escolhida como implementação do padrão HLA, desenvolvido pela ONERA [53]. A descrição do arquivo *.fed* usado pela federação é apresentado no Código 6.5, são inseridos os campos de identificação(*source*, *address* e *size*) e os campos de dados (*data*[0...15]). O tamanho dos atributos dos dados depende da natureza dos dados, onde N é o número de atributos necessários para troca dos dados. Esses campos são definidos pelo projetista.

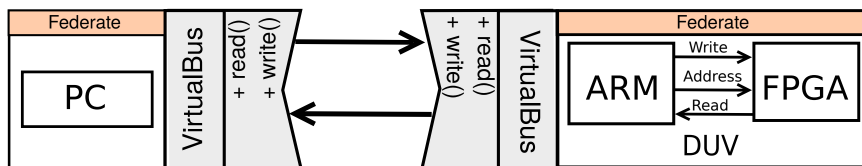
Código 6.5: Classes do objeto de dados do arquivo *.fed* para comunicação entre os dispositivos.

```
1  (FED
2    (Federation Test) (class VirtualBus
3     (attribute privilege)
4        (class RTIprivate)
5        (class all
6          (attribute source reliable timestamp)
7          (attribute address reliable timestamp)
8          (attribute size reliable timestamp)
9          (attribute data0 reliable timestamp)
10                     ...
11         (attribute data15 reliable timestamp) )
12       (class duv_FPGA
13         (attribute source reliable timestamp)
14         (attribute address reliable timestamp)
15         (attribute size reliable timestamp)
16         (attribute data0 reliable timestamp)
```

```
17                          ...
18          (attribute data15 reliable timestamp) ) ) ) )
```

Segundo Silva et al.[17], foi implementado um *wrapper* para integrar diferentes módulos em sistemas heterogêneos. Nesta subseção trata-se a evolução desse trabalho que contribuiu para a integração do *Virtual Bus*/HLA na intercomunicação dos integrantes no ambiente de verificação funcional, permitindo verificar projetos em arquiteturas heterogêneas.

Figura 6.6: Teste de integração dos *wrappers* na verificação funcional com os dispositivos propostos.



Fonte: próprio autor

Para testar a integração dos *wrappers* com *testbench* em arquiteturas heterogêneas, um cenário foi configurado como mostrado na Figura 6.6. A placa FPGA é selecionada para ser o DUV, enquanto o PC é responsável em executar o *testbench* e o *reference model*. Com a configuração do ambiente são realizados os testes para verificar o componente da integral da imagem. Após a prototipação da implementação da integral da imagem na FPGA, é implementado o modelo de referência que será executado no PC em um processo separado do *testbench*, conectado-se pelo *Virtual Bus* aos demais componentes da verificação.

Código 6.6: *Thread* principal do ARM, que lida com recebimento dos dados.

```
1   federate = VirtualBusFederate();
2   federate->runFederate();
3   while (true) then
4     if (federate->read(data)) then
5       send_to_fpga(data);
6       read_from_fpga(data);
7       id = PC_id;
8       federate->write(id, data);
9     end if
10    federate->advanceTime(1.0);
11  end while
```

O pseudocódigo da lógica de implementação do ARM está apresentado no Código 6.6. Após o *Virtual Bus* ser instanciado, a função $read(data)$ (Código 6.6, linha 4) é chamada para verificar se algum dado novo foi recebido. Se sim, ele é enviado para a FPGA (Código 6.6, linha 5) e o resultado da operação é lido de volta (Código 6.6, linha 6). A resposta é retornada informando o id (Código 6.6, linha 7) do federado de destino (neste caso, o PC) e com o uso da função $write(id, data)$ (Código 6.6, linha 8), o dado é enviado utilizando o RTI. Por fim, o ARM chama a função $advanceTime$ (Código 6.6, linha 10) para manter a sincronização entre os federados.

Para ser possível utilizar o *testbench* em conjunto com a FPGA, um componente DUV *wrapper* implementado e integrado com o *Virtual Bus* dentro do *testbench*. Assim, para o *testbench* o DUV é uma caixa-preta podendo ser conectada com qualquer federado que utilize o *Virtual Bus*. A criação e a inicialização do federado é implementada no construtor do módulo *SystemC*, assim, conectando o DUV dos *wrappers* interno e externo para realizar as transações no processo de verificação da FPGA, na parte da máquina de estados *reset* definida pelo *testbench*.

Na *thread* principal do DUV, conforme mostrado no Código 6.7, o dado de entrada é enviado para FPGA usando a função *write()* do *Virtual Bus* e, só então, o tempo HLA é avançado. Com o avanço de tempo, o federado aguarda um retorno do RTI para poder continuar.

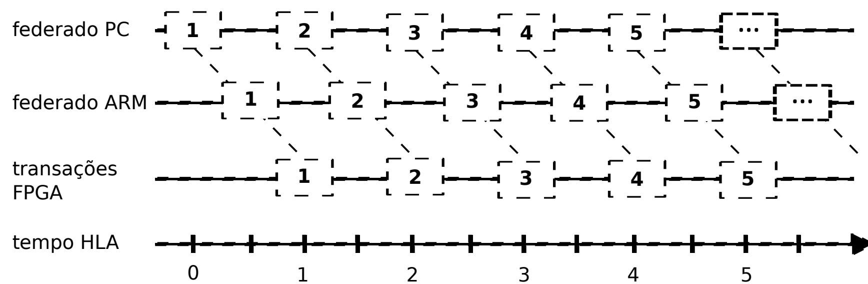
Código 6.7: *Thread* principal do ARM, que lida com envio dos dados.

```
1  while (true) then
2    data = in_data;
3    id = SOC_id;
4    federate->write(id, data);
5    federate->advanceTime(1.0);
6    if (federate->read(data)) then
7      out_data = data;
8      out_valid = 1;
9    end if
10   wait (posedge clock);
11 end while
```

Com o retorno do RTI, o DUV verifica se um dado foi recebido pelo *Virtual Bus*, através

da função *read()* (Código 6.7, linha 6), e atualiza os valores de saída em caso de positivo (Código 6.7, linha 7). Após isto, ele fica aguardando uma borda positiva do *clock* (Código 6.7, linha 10).

Figura 6.7: Fluxo do tempo de comunicação HLA.



Fonte: próprio autor

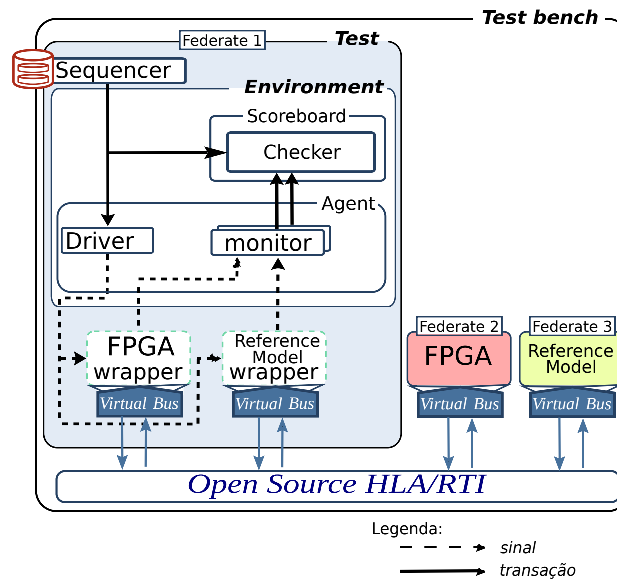
Na Figura 6.7 ilustra-se o fluxo de comunicação, onde a linha federado PC representa o momento em que os pacotes são enviados pelo PC; federado ARM é o momento que o ARM recebe o pacote e envia-o para a FPGA; o fluxo inverso ocorre no federado FPGA para o PC, quando os dados da FPGA chegam no PC. Todo esse processo ocorre em cada ciclo HLA.

Comunicação durante a verificação funcional entre PC e FPGA

Com a implementação e os *wrappers* concluídos, um cenário foi criado para este experimento para realizar a verificação funcional com interações e sincronização do subsistema S_2, conforme ilustrado na Figura 6.8. Esse cenário verifica a implementação S_2 do FPGA (DUV) com o PC (*testbench*) que também tem implementado o modelo base (*reference model*). Nos dispositivos são implementados *wrappers*, isso permite que o ambiente realize a intercomunicação com os dispositivos por meio da interface *Virtual Bus*/HLA. A aplicação da integral da imagem foi implementada em *SystemVerilog* no federado SoC, enquanto no *reference model* foi implementa em C++ no federado PC. Com a simulação do ambiente de verificação funcional, o resultado da FPGA e do *reference model* são comparados.

A comunicação entre *driver*, *monitor* e *checker* é no nível TLM. O *subscriber* realiza o tratamento das portas TLM, colocando os sinais recebidos em uma FIFO. O Código 6.8 descreve o recebimento de uma transação com a função *write*, que guarda o sinal em uma *tlm_fifo*. As FIFOs garantem a sincronização entre DUV e *reference model*.

Figura 6.8: Modelo para verificação funcional distribuída com *testbench*, FPGA e *reference model*.



Fonte: próprio autor.

Código 6.8: Pseudocódigo do *Subscriber*.

```
1 struct subscriber{
2   tlm_fifo<sequence_item>  fifo;
3   void write( sequence_item& t ) {
4     if(fifo.nb_put(t) == 0) {
5         ERROR(name(), ''[write]: It was not possible to put the item in
              the FIFO'');
6     }
7   }
8 };
```

Ao iniciar a execução, a máquina de estados é definida com as seguintes fases: na (1) a fase de *reset* é executada, em que todos os componentes são reiniciados, incluindo o DUV e o *reference model*, isso para garantir que cada componente tenha os mesmos valores iniciais; na (2) a fase de configuração é executada, em que todas as configurações iniciais são iniciadas; na (3) são realizados os testes, que estimulam a entrada do DUV; na (4) acontece o fim da fase de execução, quando são aguardadas algumas transações para finalizar o processo; na fase (5) é verificado se algum contador de erros é incrementado e termina a verificação imprimindo o número de erros, quando houver.

6.3.3.2 Desenvolvimento do subsistema S_1 na GPU

Especificação do conversor de cores

Em seguida, foi desenvolvido o subsistema do conversor de cores de vermelho, verde e azul (RGB, do inglês *Red, Gree e Blue*) para componente luminância (Y) e componentes de crominância (C_b e C_r) (YC_bC_r, do inglês *Luminance and Chrominance Components*) na GPU. O conversor de cores é uma tarefa importante em aplicativos de processamento de imagem. As cores RGB são tratadas tanto em imagens quanto em vídeos em tempo real, sendo a conversão de RGB para YC_bC_r amplamente utilizada em processamento de imagem e vídeo. Sua contribuição cientifica é baseada na sensibilidade das células de detecção de cores no sistema visual humano.

Dado um *pixel* digital representado no formato RGB, com 8 bits por amostra tem um total de $2^8 = 256$ valores, em que a faixa vai de 0 e 255 e esses valores representam os limites de menor e maior intensidade de cor, sendo um limite branco e o outro preto, respectivamente. Os formatos de YC_bC_r são mostrados nas equações (1), (2) e (3) em 6.4, respectivamente.

$$\begin{cases} Y = 16 + \frac{65.738R}{256} + \frac{129.057G}{256} + \frac{25.064B}{256} & (1) \\ Cb = 128 - \frac{37.945R}{256} - \frac{74.494G}{256} + \frac{112.439B}{256} & (2) \\ Cr = 128 + \frac{112.439R}{256} - \frac{94.154G}{256} - \frac{18.285B}{256} & (3) \end{cases} \tag{6.4}$$

Aproximando as equações 6.4 para o número inteiro mais próximo e substituindo a multiplicação e divisão por registradores de deslocamento são obtidos nas equações mostradas em 6.5 [60].

$$\begin{cases} Y = 16 + (((R << 6) + (R << 1) + (G << 7) + G + (B << 4) \\ +(B << 3) + B) >> 8) \\ Cb = 128 + ((-((R << 5) + (R << 2) + (R << 1)) - ((G << 6) \\ +(G << 3) + (G << 1)) + (B << 7) - (B << 4)) >> 8) \\ Cr = 128 + (((R << 7) - (R << 4) - ((G << 6) + (G << 5) \\ -(G << 1)) - ((B << 4) + (B << 1))) >> 8) \end{cases} \tag{6.5}$$

Implementação concebida para GPU

Para o conversor YC_bC_r foi considerado um *reference model* para ser o modelo de base (modelo em validação). Este estudo de caso com a GPU foi aplicado para validar um modelo de base escrito em C++, utilizando funções da biblioteca OpenCL, o Código 6.9 descreve a implementação do *reference model* que será o modelo baseado durante o processo de verificação funcional. O componente Y filtra apenas a luminância (brilho) da imagem (Código 6.9, linha 28), enquanto os componentes C_b e C_r subtraem as cores vermelha e azul, respectivamente, da imagem (Código 6.9, linhas 29–30).

Código 6.9: Implementação em OpenCL do conversor de RGB para YC_bC_r.

```
1  int main( int argc, char **argv ){
2  char federateName[20];
3  sprintf(federateName, "%d", 3);
4  // create and run the federate
5  VirtualBusFederate *federate;
6  federate = new VirtualBusFederate();
7  federate->runFederate( federateName );
8  //Data to receive
9  unsigned src;
10 unsigned addr;
11 unsigned a[IMAGE_HEIGHT][IMAGE_WIDTH];
12 unsigned readsrc;
13 unsigned cntrl_write;
14 while(1){
15 readsrc = 0;
16 if(federate->readData(readsrc,addr,size, data)){
17 if(readsrc == 0){
18  std::cout << ``data from src'' << readsrc << ``:'';
19  for (size_t i = 0; i < VIRTUALBUS_SIZE; i++){
20      std::cout << `` '' << data[i];
21  }
22  std::cout << std::endl;
23  addr = SENDER_ID; //Sender address
24  for(int i = 0; i < VIRTUALBUS_SIZE; i++){
25   unsigned b=((data[i] >> 0) & MASK);
```

```
26      unsigned g=((data[i] >> 8) & MASK);
27      unsigned r=((data[i] >> 16) & MASK);
28      unsigned y=16+((65.738*r)/256)+((129.057*g)/256)+((25.064*b)/256);
29      unsigned cb=128-((37.945*r)/256)-((74.494*g)/256)+((112.439*b)/256);
30      unsigned cr=128+((112.439*r)/256)-((94.154*g)/256)-((18.285*b)/256);
31      a[i/IMAGE_HEIGHT][i % IMAGE_WIDTH]=(y<<16)+(b<<8)+(cr<<0);
32    }
33        data[0]=a[0][0];    data[1]=a[0][1];
34        data[2]=a[0][2];    data[3]=a[0][3];
35        data[4]=a[1][0];    data[5]=a[1][1];
36        data[6]=a[1][2];    data[7]=a[1][3];
37        data[8]=a[2][0];    data[9]=a[2][1];
38        data[10]=a[2][2];   data[11]=a[2][3];
39        data[12]=a[3][0];   data[13]=a[3][1];
40        data[14]=a[3][2];   data[15]=a[3][3];
41        federate->writeData(src, 1, size, data);
42        std::cout << ``New data (GPU):'';
43        for(int aux = 0; aux < VIRTUALBUS_SIZE; aux++)
44            std::cout << `` '' << data[aux];
45        std::cout << std::endl;
46  } }   federate->advanceTime(1.0); } }
```

Adaptação dos *wrappers* para comunicação com subsistema S_1 na GPU

Para realizar a verificação funcional do código executado na GPU, é preciso criar um componente *wrapper* interno ao *testbench*, assim como deve ser implementado um *wrapper* externo na GPU. Esses *wrappers* realizam a comunicação do código do conversor YC_bC_r através da troca de mensagens pelo *Virtual Bus*/HLA. O *wrapper* interno troca mensagens pela interface de comunicação do *Virtual Bus*, utilizando as funções de escrita, leitura e avanço de tempo para sincronizar a comunicação.

Para realizar a comunicação do subsistema S_1 (DUV) implementada na GPU com o *testbench*, são necessários os *wrappers* para comunicação. Dessa forma, a comunicação usando o *Virtual Bus* deve acontecer entre a comunicação do *wrapper* na GPU com o *wrapper* criado no *testbench*, conforme descrito no Código 6.10. A implementação do *wrapper* permite facilitar a comunicação, para isso acontecer deve implementar os *wrappers*, permitindo que

os subsistemas se tornem federados, assim os federados trocam mensagens na federação.

Código 6.10: Bloco *SystemC* no DUV *wrapper*: para conectar o *testbench* com a aplicação no dispositivo.

```
1  ...
2  void ii::func()
3  {
4      data_t temp_out_data;
5      bool  temp_carryout;
6      int temp_sel;
7      // reset
8      while (true)
9      {
10         if(reset_n.read() == false) {
11             out_data_valid.write(0);
12             for (int i = 0; i < DATA_SIZE; ++i) {
13                 out_data[i].write(0);
14             }
15         }
16         else if (in_data_en.read() == 1){
17             for (int i = 0; i < DATA_SIZE; ++i) {
18                 data[i] = in_data[i].read();
19 
20             }
21             for(int i =DATA_SIZE ; i<16; i++)
22                 data[i] = 9;
23 
24             src = 2;
25             addr = 2;
26             federate->writeData(src, addr, size,  data);
27             federate->advanceTime(1.0);
28             if(federate->readData(src,addr,size, data)){
29                 for (int i = 0; i < DATA_SIZE; ++i) {
30                     temp_out_data = data[i];
31                     out_data[i].write(temp_out_data);
32                 }
33                 out_data_valid.write(1);
```

```
34                }
35            }
36            wait(1);
37        }
38    }
39    ...
```

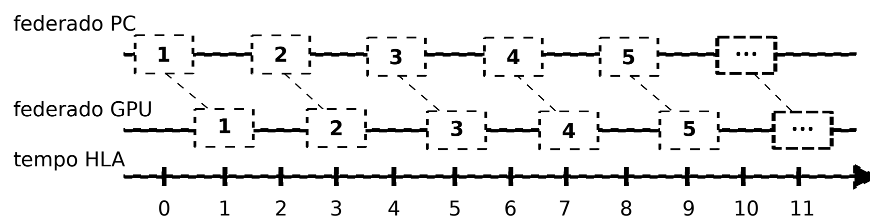
A integração de um novo dispositivo na simulação só acontece ao descrever uma classe no arquivo *.fed*, acrescentando a classe para a GPU, são inseridos os campos de identificação ($source, address$ e $size$) (Código 6.11, linha 3–5) e os campos de dados ($data[0...15]$) (Código 6.11, linha 6–8).

Código 6.11: Classes do objeto de dados do arquivo *fed* para comunicação entre os dispositivos.

```
1     ...
2       (class duv_GPU
3         (attribute source reliable timestamp)
4         (attribute address reliable timestamp)
5         (attribute size reliable timestamp)
6         (attribute data0 reliable timestamp)
7                     ...
8         (attribute data15 reliable timestamp) )
9     ...
```

Na Figura 6.9 é mostrado o fluxo de comunicação, a linha do tempo do federado PC representa o momento em que os pacotes são enviados pelo PC; federado GPU é o momento que o GPU recebe os pacotes do PC; fluxo inverso ocorre da GPU para federado PC, quando os dados da GPU chegam no PC. Todo esse processo ocorre em cada ciclo HLA.

Figura 6.9: Fluxo do tempo de comunicação HLA.

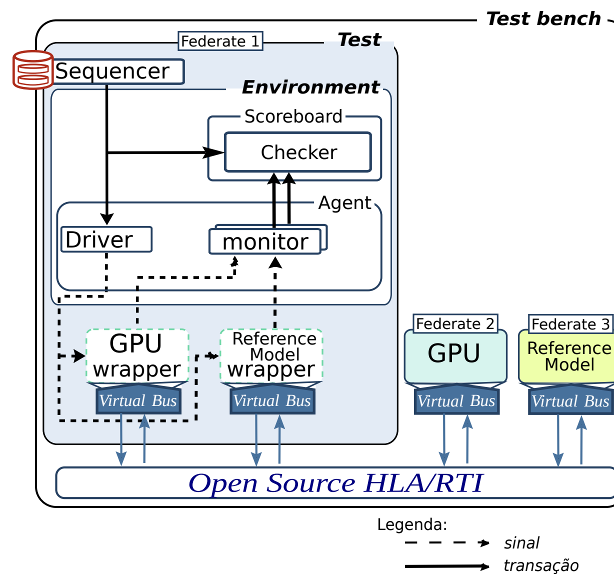


Fonte: próprio autor.

Comunicação durante a verificação funcional entre PC e GPU

Com a implementação e os *wrappers* concluídos, um cenário foi criado para este experimento construindo para realizar a verificação funcional com interações e sincronização do subsistema S_1, conforme ilustrado na Figura 6.10. Esse cenário verifica a implementação S_1 do GPU com o PC (*testbench*) que também tem implementado o modelo base (*reference model*). Nos dispositivos são implementados *wrappers*, isso permite que o ambiente realize a intercomunicação com os dispositivos por meio da interface *Virtual Bus*/HLA. A aplicação do conversor de cores é implementada em OpenCL no federado SoC, enquanto no *reference model* é implementa em C++ no federado PC. Com a simulação do ambiente de verificação funcional, o resultado da GPU e do *reference model* são comparados.

Figura 6.10: *testbench*, *reference model* e GPU integrados e sincronizando.



Fonte: próprio autor.

Desenvolvimento da aplicação em *hardware*

Com base na especificação matemática, a etapa de desenvolvimento no dispositivo é baseada com a utilização do processamento e do potencial da placa. A implementação em OpenCL do conversor permite o uso do dispositivo heterogêneo para o projeto, assim validando durante a verificação. A proposta do trabalho trata a verificação funcional de IP-*cores* pre-existentes ou pre-verificados em diferentes linguagens de programação. Para isso,

nesta etapa foi realizada o desenvolvimento do código do conversor de cores de RGB para YC_bC_r na GPU, com base no Código 6.12. A implementação utiliza a tecnologia OpenCL. O *hardware* de desenvolvimento utilizado foi a plataforma para sistemas embarcados da NVIDIA, Jetson TX1 equipada com GPU NVIDIA Maxwell, 256 núcleos CUDA, CPU Quad ARM® A57 / 2 MB L2 e 4 GB de RAM compartilhada.

Código 6.12: Implementação OpenCL do conversor de RGB para YC_bC_r.

```
1   __kernel void YCbCr(__global float* a, __global float* b, int
        iNumElements) {
2   int iGID = get_global_id(0);
3   if (iGID < iNumElements) {
4           int b_ = ((a[iGID] >> 0) & 255);
5           int g_ = ((a[iGID] >> 8) & 255);
6           int r_ = ((a[iGID] >> 16) & 255);
7           int y = 16 + ((65.738 * r) / 256) + ((129.057 * g) / 256) +
                ((25.064 * b) / 256);
8           int cb = 128 - ((37.945 * r) / 256) - ((74.494 * g) / 256) +
                ((112.439 * b) / 256);
9           int cr = 128 + ((112.439 * r) / 256) - ((94.154 * g) / 256) -
                ((18.285 * b) / 256);
10          b[iGID] = (y << 16) + (b << 8) + (cr << 0);
11  }
12  }
```

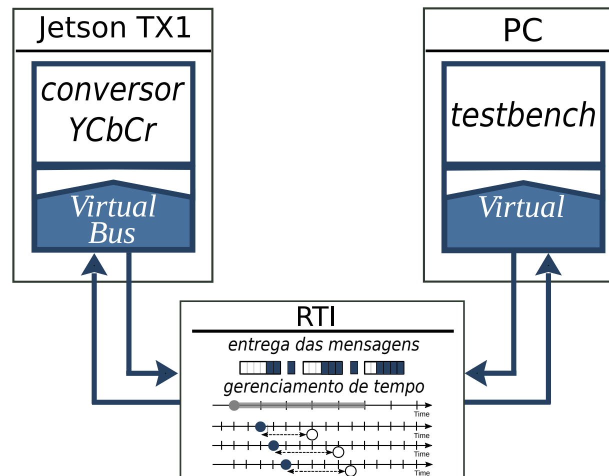
Na Figura 6.11 é mostrado a arquitetura dos experimentos, conectando o PC e a placa Jetson TX1. Assim é possível enviar os estímulos a partir de uma imagem gerada, o conversor calcula e retorna o resultado para a verificação, conforme mostrado na Figura 6.11. A troca de mensagem utiliza os serviços RTI através da interface da API *Virtual Bus*.

6.3.3.3 Cenário 3: subsistemas conectados em série

Com base no desenvolvimento do subsistema S_2 na FPGA e do subsistema S_1 na GPU, este cenário propõe conectar os subsistemas em série para realizar a verificação funcional. No *reference model* implementa-se o código desses dois subsistemas para funcionarem juntos.

Este cenário é composto em sua estrutura de verificação funcional, considerando que são

Figura 6.11: Teste de verificação funcional com arquitetura heterogênea GPU.

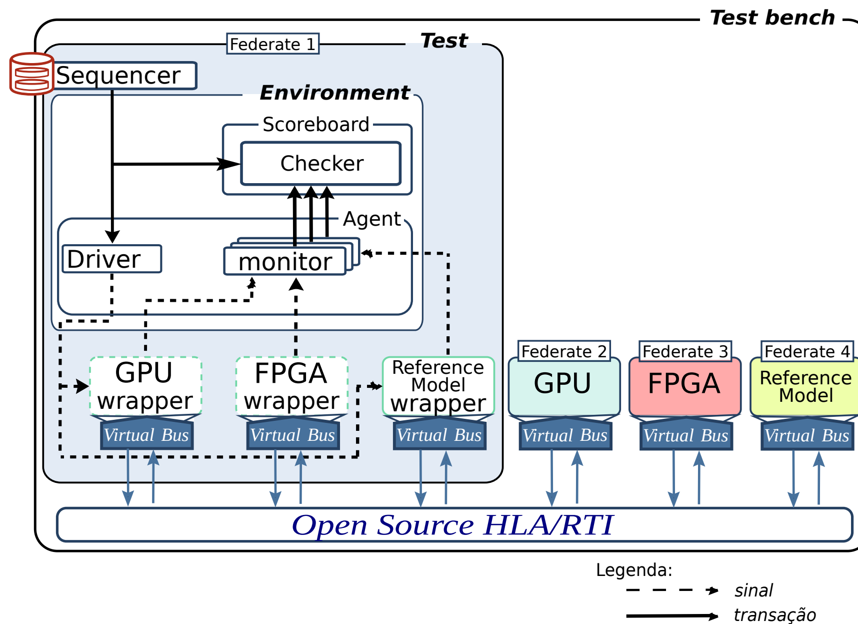


Fonte: próprio autor

cinco federados nos testes o PC (*testbench* e *reference model*), GPU (DUV) e FPGA (DUV), conforme ilustrado na Figura 6.12. O ambiente de verificação funcional distribuída deste trabalho verifica componentes *open source* IP-*cores* em GPU e FPGA. Durante a simulação, a FPGA utiliza os estímulos gerados pela GPU. No entanto, para que aconteça a integração com o *testbench*, cada dispositivo deve implementar um *wrapper* externo, assim como um respectivo *wrapper* interno ao ambiente de verificação funcional, permitindo estabeler as conexões. Os testes de verificação funcional com quatro federados, são realizados com o subsistema S_1 implementado no federado GPU e, em seguida, enviando para o subsistema S_2 implementado no federado FPGA. O *reference model* é disponibilizado em único código que tem implementado S_1 e S_2. Por fim, a saída do subsistema S_2 na FPGA e da saída do *reference model* são comparados.

Na Figura 6.13, as linhas e círculos representam o tempo de sincronização e avanço de tempo lógico concedido pelos serviços da RTI. Isso ocorre quando é solicitada alguma atividade de assinatura pelos federados. Dessa forma, as linhas e os círculos significam um avanço de tempo para cada um dos subsistemas durante a simulação, ou seja, um avanço de tempo lógico concedido pela RTI, caso seja solicitado pelo federado. Cada federado afiliado tem um valor limite para enviar mensagens por assinatura denominado registro de data e hora de menor entrada (LITS, do inglês *Least Incoming Timestamp*). Esse valor expressa o menor *timestamp* em que um federado afiliado pode receber no futuro uma mensagem do

Figura 6.12: Quatro federados comunicando e sincronizando com o federado *testbench*.



Fonte: próprio autor.

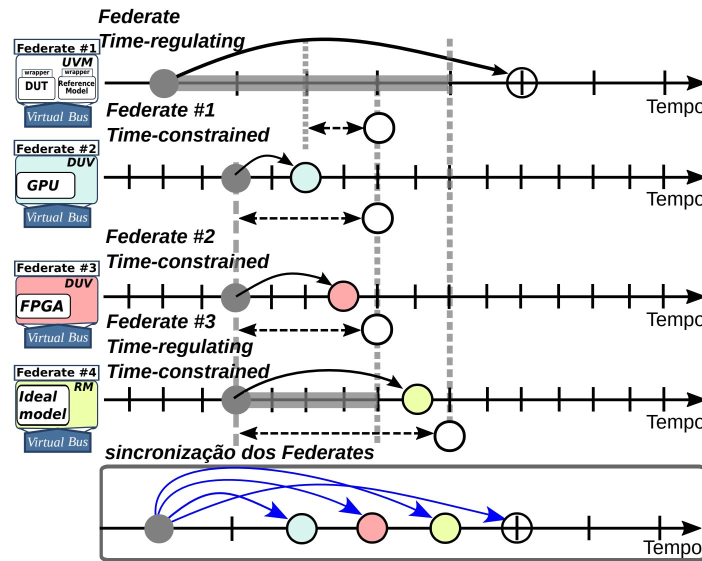
tipo TSO, ou seja, o *timestamp* da próxima mensagem que o federado pode ter que processar. Inicialmente, o ambiente inicia o tempo regulado para troca de mensagens entre os demais federados. Para acontecer o sincronismo, os federados devem realizar o avanço de tempo durante a troca de mensagens como ilustrado no exemplo do *Federate* #1 e do *Federate* #2, isso é baseado na assinatura do *Federate* #1 para o *Federate* #2. Para ocorrer sincronismo no momento da comparação no *checker* é preciso que o *Federate* #3 seja regulador e restrito de tempo para que a assinatura dos federados ocorra no mesmo momento no *checker*. Esse fluxo para interação e sincronização é ilustrado na Figura 6.13.

O Código 6.13 apresenta o código de execução genérico de cada um dos DUV em nossa aplicação. Na Linha 2 espera-se o recebimento de dados a serem processados na Linha 3. O processamento pode ser a execução do algoritmo de integral da imagem ou do filtro YC_bC_r. Depois de processados, os dados são enviados na Linha 5. Por fim, na Linha 6, é realizado um avanço de tempo.

Código 6.13: thread principal do DUV na proposta de implementação.

```
1 while (true) then
2    if (federado->read(data)) then
```

Figura 6.13: Quatro federados interagindo e sincronizando no tempo.



Fonte: próprio autor.

```
3         data = federado->processing(data);
4         id = SOC_id;
5         federado->write(id, data);
6         federado->advanceTime(1.0);
7     end if
8     wait (posedge clock);
9 end while
```

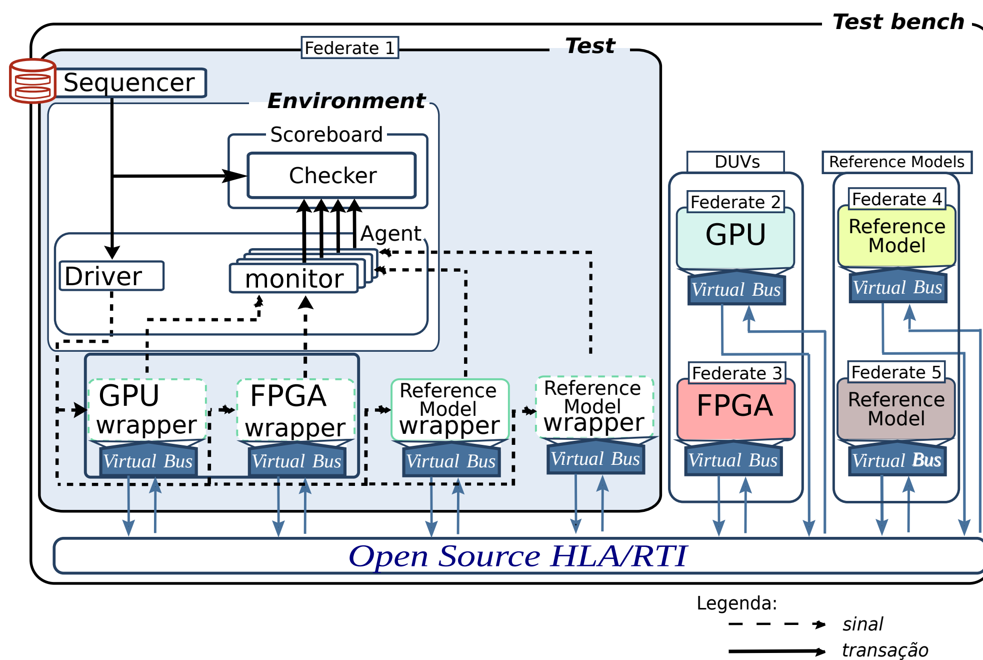
6.3.3.4 Cenário 4: subsistemas conectados em paralelo

Com base no desenvolvimento do subsistema S_2 na FPGA e do subsistema S_1 na GPU, este cenário propõe conectar os subsistemas em paralelo para realizar a verificação funcional. As implementações desses dois códigos estão separados em seus respectivos *reference models*.

Este cenário é composto em sua estrutura de verificação funcional, considerando que são cinco federados nos testes o PC (*testbench*, *reference model* de S_1 e *reference model* de S_2), GPU (DUV) e FPGA (DUV), conforme ilustrado na Figura 6.14. O ambiente de verificação funcional distribuída deste trabalho verifica componentes *open source* IP-*cores* em GPU e FPGA. Durante a simulação, quando os estímulos são gerados, eles são enviados para a FPGA e a GPU. No entanto, para que aconteça a integração com o *testbench*, cada dispositivo deve implementar um *wrapper* externo, assim como um respectivo *wrapper* interno

ao ambiente de verificação funcional, permitindo estabeler as conexões. Os testes de verificação funcional com cinco federados, foram realizados com o subsistema S_1 implementado no federado GPU e o subsistema S_2 implementado no federado FPGA. Cada subsistema é comparado com um modelo base, já implementado e testado, sendo um *reference model* de S_1 e um *reference model* de S_2. Por fim, as saídas dos subsistemas S_1 e S_2 são comparados com seus respectivos *reference models* conectados em paralelo.

Figura 6.14: Cinco federados comunicando e sincronizando com o federado *testbench*.

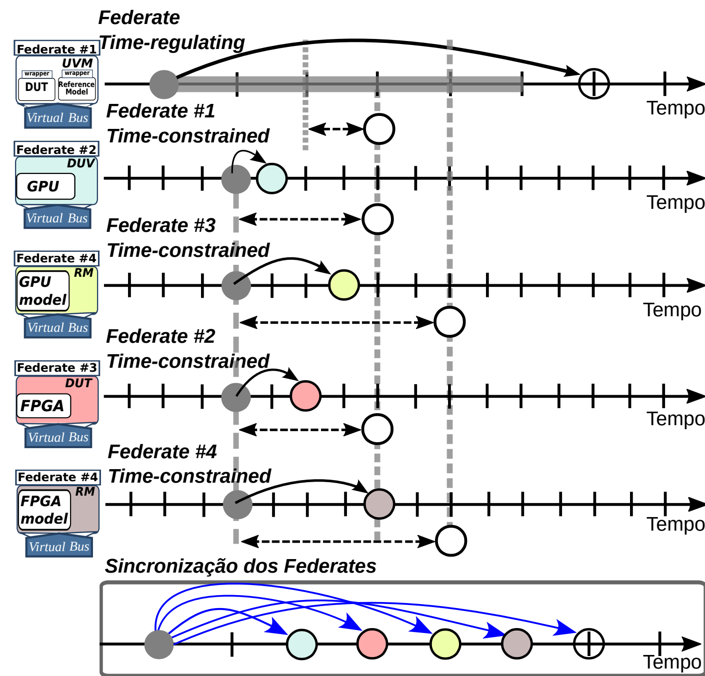


Fonte: próprio autor.

O fluxo para interação e sincronização é ilustrado na Figura 6.15. Na figura com cinco federados é realizando interação e sincronização, esse exemplo utiliza os mesmos princípios em que um federado afiliado pode receber no futuro uma mensagem do tipo TSO. Inicialmente, o ambiente inicia o tempo regulado para troca de mensagens entre os demais federados. Todos os federados recebem os estímulos ao mesmo tempo. Conforme avançam o tempo, enviam para o ambiente de verificação o resultado de cada federado em determinado tempo HLA. O sincronismo ocorre pelo avanço de tempo, conforme realiza a assinatura do federado #1 e #3, eles enviam a mensagem para ser comparada. Depois são realizadas as assinaturas dos federados #2 e #4 para serem comparados. Para ocorrer sincronismo no momento da comparação no *checker* é preciso que o ambiente faça o sincronismo com federados de tempo

restrito e esperem o fim de uma comparação para iniciar outra.

Figura 6.15: Cinco federados interagindo e sincronizando no tempo.



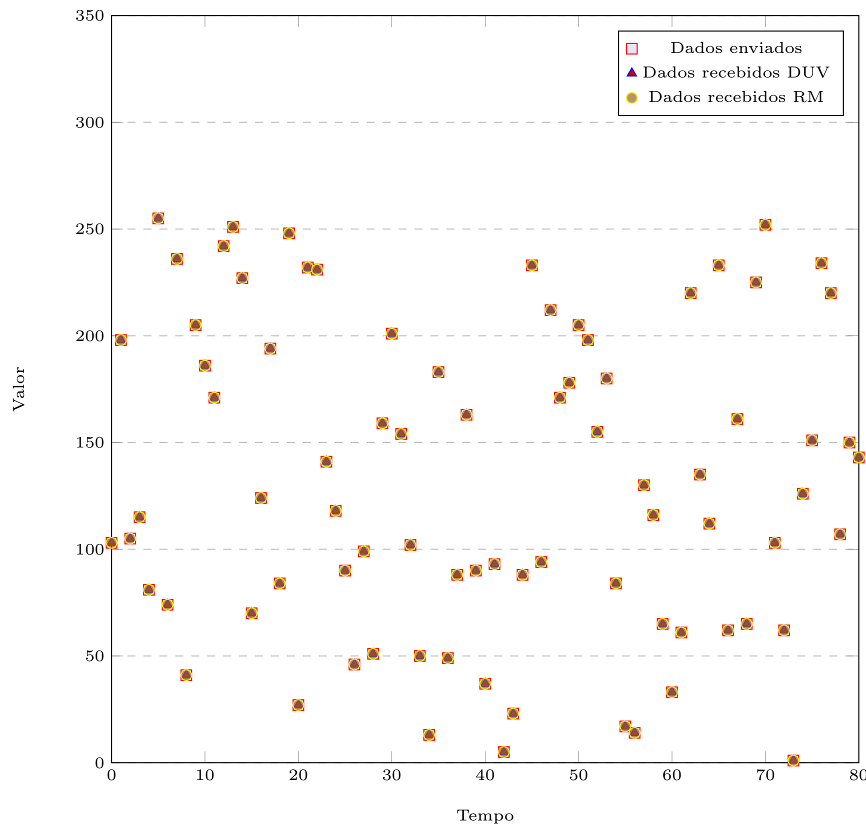
Fonte: próprio autor.

6.3.4 Resultados

Com o *testbench* e o *reference model* implementados em um federado e duas versões dos subsistemas em federados diferentes. Para as implementações dos subsistemas, o algoritmo selecionado a ser verificado foram a integral da imagem, implementada usando *SystemVerilog* na FPGA, e outra implementada do conversor de cores RGB para YC_bC_r em OpenCL na GPU. Inicialmente, cada um deles foram desenvolvidos em um federado separado.

Além da verificação funcional do algoritmo, foram adicionadas medidas de tempo de execução para mostrar o tempo de execução de cada iteração. O objetivo principal demonstrou a capacidade do ambiente proposto para integrar e intercomunicar componentes de diferentes subsistemas interagindo na verificação funcional distribuída. Todos os testes aqui foram realizados após diversas interações, cerca de 10.000, proporcionando verificar se acontece interrupção durante a simulação e verificando possíveis erros, que resultam analisar subsistemas separados no processo de verificação.

Figura 6.16: Análise do sincronismo dos dados



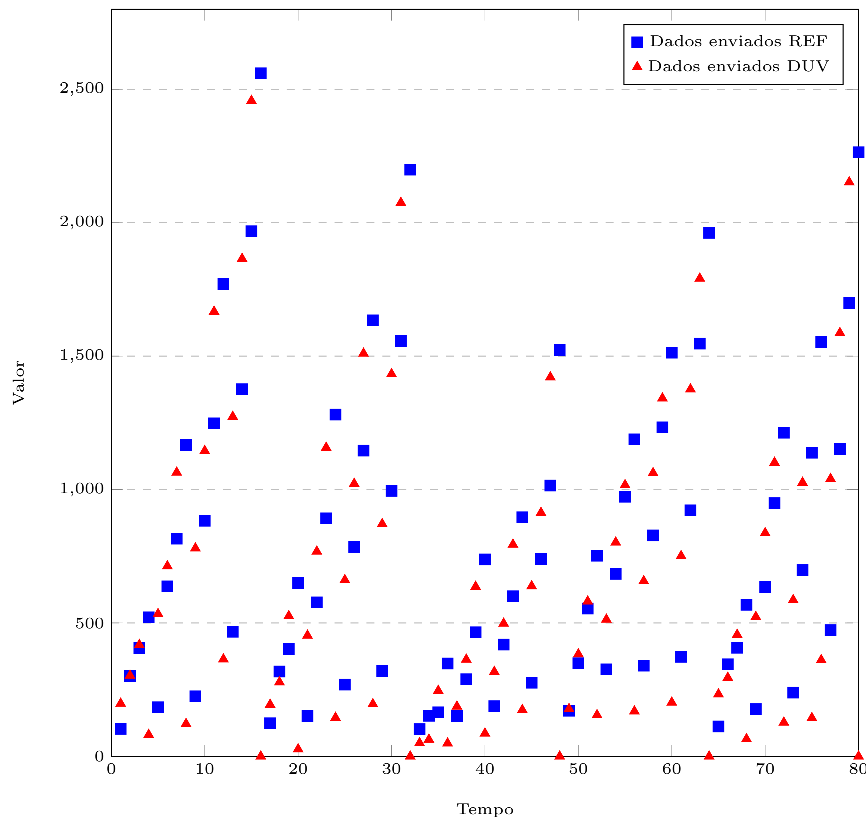
Fonte: próprio autor.

6.3.4.1 Análise do sincronismo e da detecção de erros

O *wrapper* implementado nos subsistemas utiliza os serviços HLA fornecidos para configuração de início e encerramento da federação, gerenciamento de troca de dados, gerenciamento de tempo e assim por diante. Para interoperação, os federados são estendidos para executar procedimentos usando pelo *Virtual Bus*/HLA para invocar os serviços e procedimentos de retorno da chamada que o RTI solicita.

Para avaliar a sincronização no envio das mensagens foi realizado um estudo de caso. A arquitetura de teste utilizada foi o mesmo que o Cenário 1, onde temos uma FPGA realizando o processamento de integral da imagem. Foram coletados os dados enviados pelo *testbench* e os dados recebidos pelo DUV e pelo seu *reference model*. Foram coletados os cinco primeiros ciclos de informação, ou seja, os 80 primeiros dados enviados. A Figura 6.16 apresenta um gráfico com os dados coletados. Pode-se notar que os dados enviados pelo *testbench* coincidem com os dados recebidos pelo DUV e pelo *reference model*. Dessa forma,

Figura 6.17: Análise de detecção de erros fixos



Fonte: próprio autor.

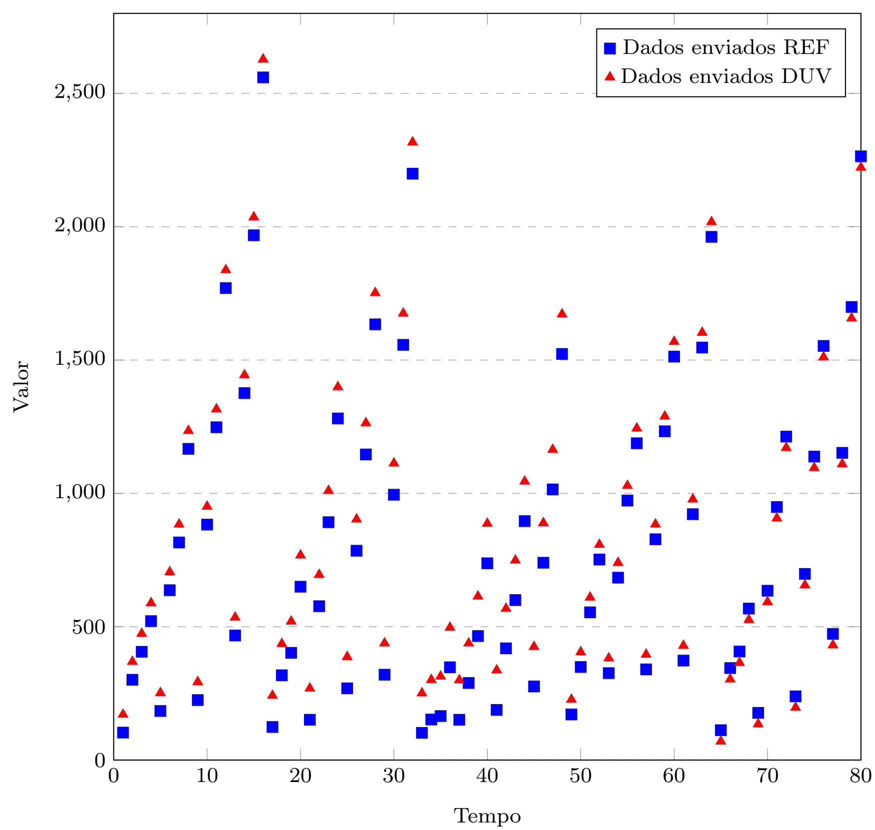
o sincronismo dos dados é mantido nessa amostra coletada.

Para avaliar a detecção de erros, foram implementados dois testes. No primeiro teste é adicionado um erro, fazendo com que se ignore a primeira posição do vetor de entrada e utilize esta posição com valor nulo. O resultado desse primeiro experimento é apresentado na Figura 6.17. No segundo teste, também é ignorado o valor da primeira posição do vetor de entrada, porém, substituí-se o seu valor por um aleatório. O resultado desse segundo experimento é apresentado na Figura 6.18. Vale destacar que em todos os casos, o ambiente de verificação funcional distribuído detectou o erro, gerando mensagem de erro.

6.3.4.2 Validação da comunicação do subsistema S_2 no SoC (ARM + FPGA) interagindo com *testbench*

A configuração da simulação iniciou 10 segundos após o início da coleta de dados, conforme apresentado na Figura 6.19. A simulação leva 97 segundos. Nenhum dado de tráfego

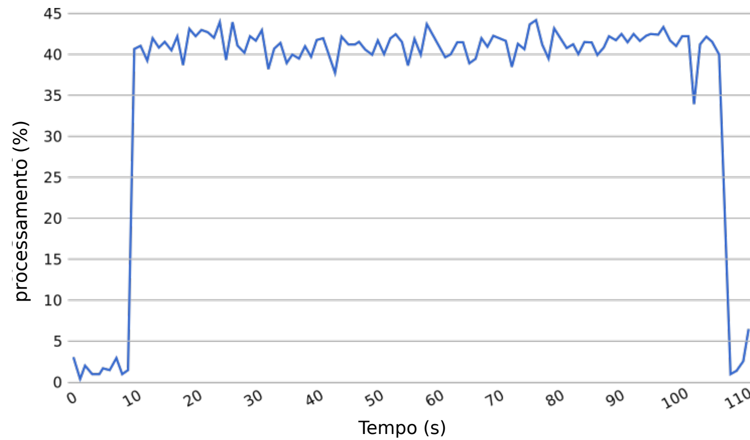
Figura 6.18: Análise de detecção de erros aleatórios



Fonte: próprio autor.

de rede foi coletado, toda simulação é feita em uma única máquina.

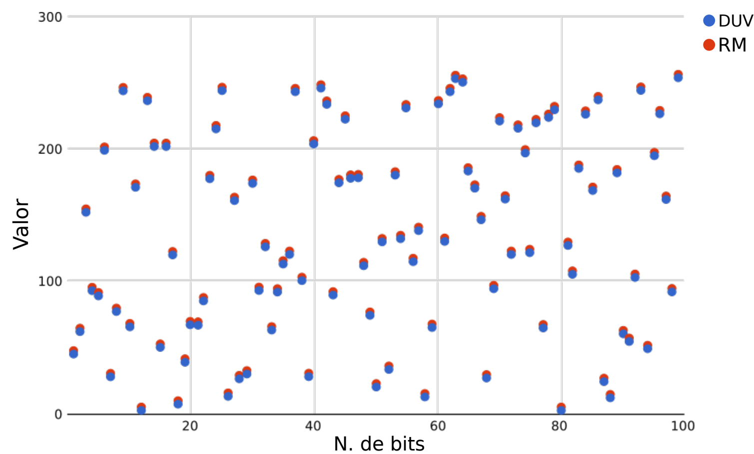
Figura 6.19: Uso do processador pelo ambiente de verificação com *Virtual Bus*.



Fonte: próprio autor

Para verificar a precisão do ambiente de verificação, um pequeno erro foi adicionado intencionalmente nas saídas do *reference model* (RM). Na Figura 6.20 os *pixels* de saída do DUV e do RM são plotados. Assim, é possível ver o casamento entre os 100 *pixels* apresentados.

Figura 6.20: Comparação entre o DUV e o *reference model*.



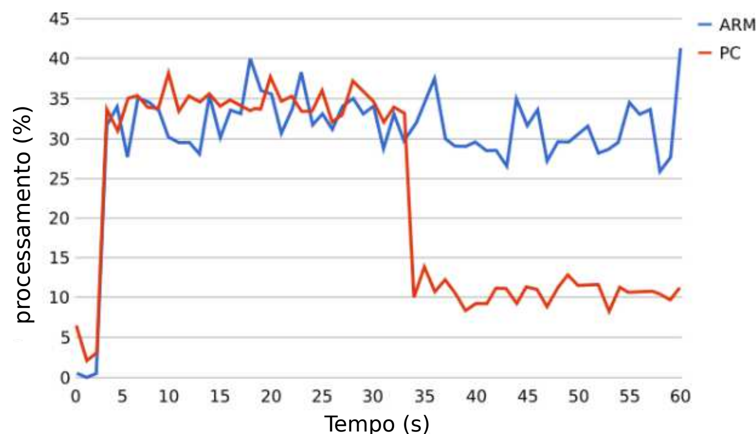
Fonte: próprio autor.

O primeiro passo é a comunicação com o ARM e a FPGA por meio da interface *Avalon* [62]. A última etapa foi a simulação que executa o ambiente de verificação com o *hardware* em *loop* usando o *Virtual Bus*/HLA para trocar os elementos ao nível de transação.

A integral da imagem implementada é semelhante à desenvolvida em *SystemC*. Assim, o *Virtual Bus* foi configurado para transmitir 16 elementos de 32 *bits* em cada tempo HLA. Em cada sinal de *clock*, o ambiente envia 16 valores gerados usando uma função aleatória. O DUV executa as operações e retorna o resultado usando os mesmos sinais do *Virtual Bus*.

Na execução do ambiente de verificação funcional foram utilizados o PC e o ARM durante 10.000 transações do *Virtual Bus*/HLA, como mostrada na Figura 6.21. A transmissão começa pelo PC após 4 segundos, agora é possível ver que o uso do processador aumenta em ambos os lados (PC e ARM). Após 30 segundos, o uso diminui no PC porque o processo de verificação terminou, mas o federado do ARM continua aguardando transações HLA.

Figura 6.21: Uso do processador ARM e PC (porcentagem) por segundos.

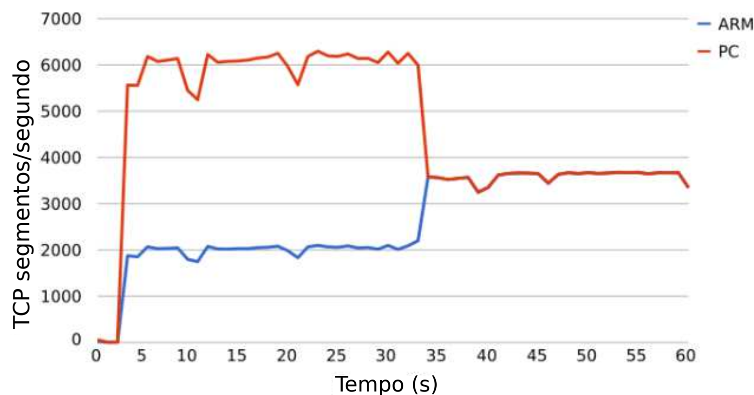


Fonte: próprio autor

O experimento que mostra o tráfego de rede do ARM para o PC é mostrado na Figura 6.22. Pode-se notar o número total de segmentos transferidos por segundo, incluindo aqueles nas conexões atuais e excluindo aqueles que contêm apenas segmentos retransmitidos. Como esperado, ambos os componentes começam a usar a rede após cerca de 4 segundos. É possível observar que o PC gera mais tráfego de rede que um ARM porque transmite e recebe dados a serem processados e também executa o RTI. Após 34 segundos, a simulação termina e o tráfego de rede é gerado pelo processador ARM que está aguardando o processamento dos dados, resultando em uma queda do processamento do PC.

Em comparação com o resultado quando o *Virtual Bus* não foi utilizado no experimento, a simulação executa mais rápido (60 segundos), visto que não tem o *overhead* de transmissão, do que quando executa a simulação usando *Virtual Bus* e um modelo em *SystemC* (94

Figura 6.22: Tráfego de rede por segundo para o ARM e o PC.



Fonte: próprio autor

segundos). Isso ocorreu porque o FPGA processa os dados em tempo real de *clock* e não no *clock* simulado, compensando assim qualquer atraso possível introduzido pela rede.

6.3.4.2 Validação da comunicação do subsistema S_1 na GPU interagindo com *testbench*

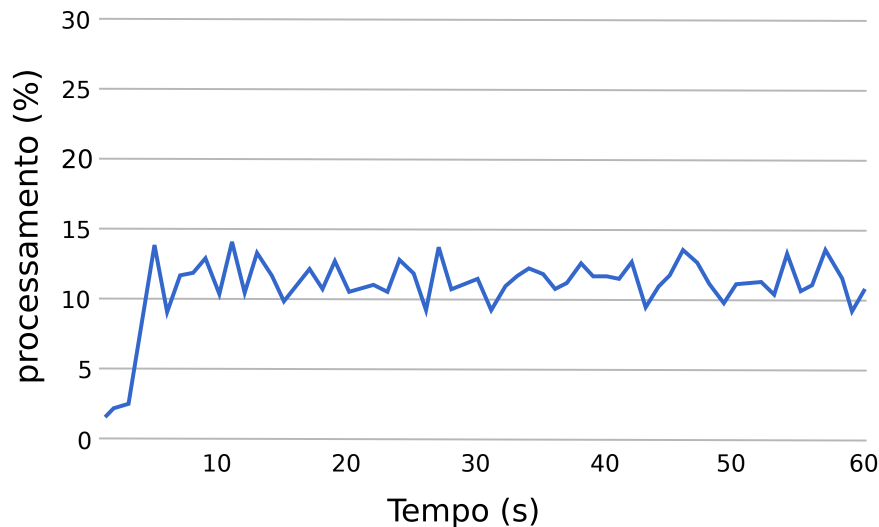
Outro experimento foi executado integrando uma GPU com um PC. Neste cenário, o PC está executando o *testbench* com a implementação do *reference model* e a GPU *Jetson* TX1 executa o DUV, que executa o subsistema S_1 em OpenCL. Para provar que o ambiente poderia enviar mais dados através do *datapath* e feita a verificação, ele foi configurado para utilizar 1024 elementos do tipo *unsigned int* em cada iteração a ser processada pelo DUV e *reference model*.

Na Figura 6.23 ilustra-se o uso do processador ARM da GPU. O tempo médio de resposta entre os estímulos e a saída recebida foi de 870 microssegundos. Com o tempo médio de processamento da GPU de 205 microssegundos. Neste caso, a integração fornecida pelo *Virtual Bus*/HLA permitiu verificar em um tempo viável.

6.3.4.4 Verificação com subsistemas conectados em série com um *reference model*

Nos experimentos, as mensagens HLA são enviadas na mesma quantidade. Esses experimentos foram considerados para as abordagens com quatro e cinco federados, executando simulação de 10.000 interações. Em ambas as abordagens são calculadas as médias totais de interações. Ambas as abordagens usam 15 *pixels* por atributos (HLA), um atributo para

Figura 6.23: Uso de processamento da *Jetson* TX1.



Fonte: próprio autor

cada *pixel*. Assim como, o identificador para troca de mensagens entre os federados na simulação.

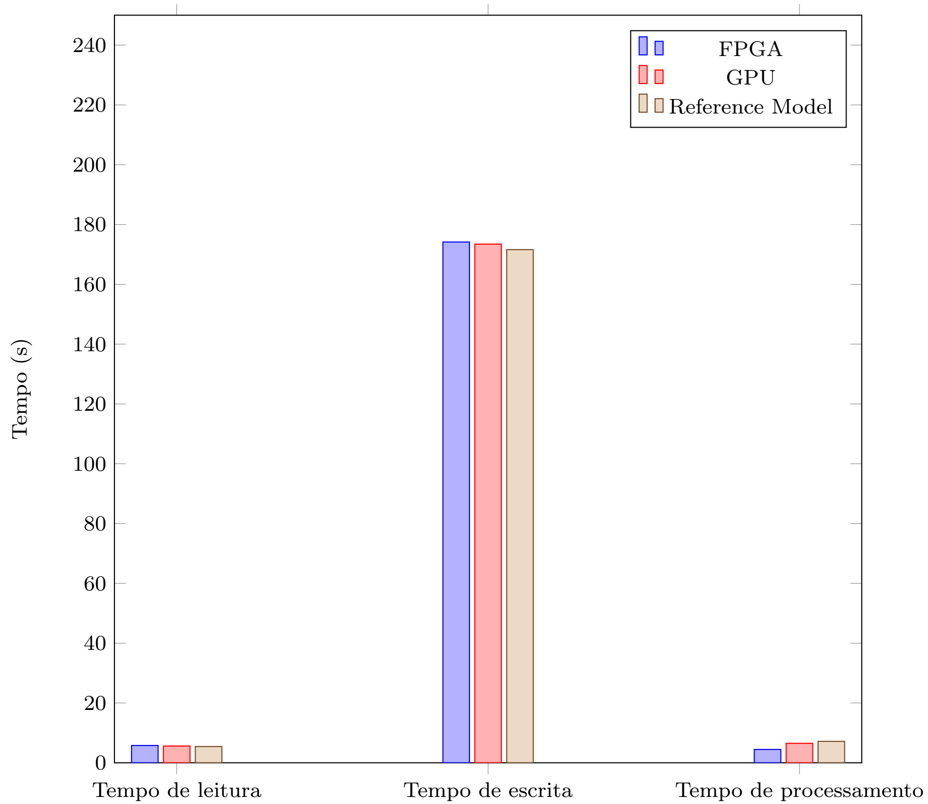
Nas Figuras 6.24 e 6.25 ilustram-se os envios dos diferentes subsistemas com matrizes de dados do mesmo tamanho. Esse experimento foi importante para avaliar o impacto de diferentes abordagens com *Virtual Bus*/HLA mesmo com o aumento dos federados em diferentes arquiteturas heterogêneas.

O experimento do Cenário 1 (C_1) como mostrado na Figura 6.24 foi repetido para o Cenário 2 (C_2) com o acréscimo de um federado Figura 6.25 . Visto que ambos utilizam o mesmo código, a redução na simulação com o aumento de federados é clara, reduzindo a latência de comunicação na troca de mensagens HLA até a comparação. Assim como, os avanços de tempo sincronizados permitem reduzir o gargalo de comunicação na mesma proporção.

6.3.4.5 Verificação com subsistemas conectados em paralelo com dois *reference models*

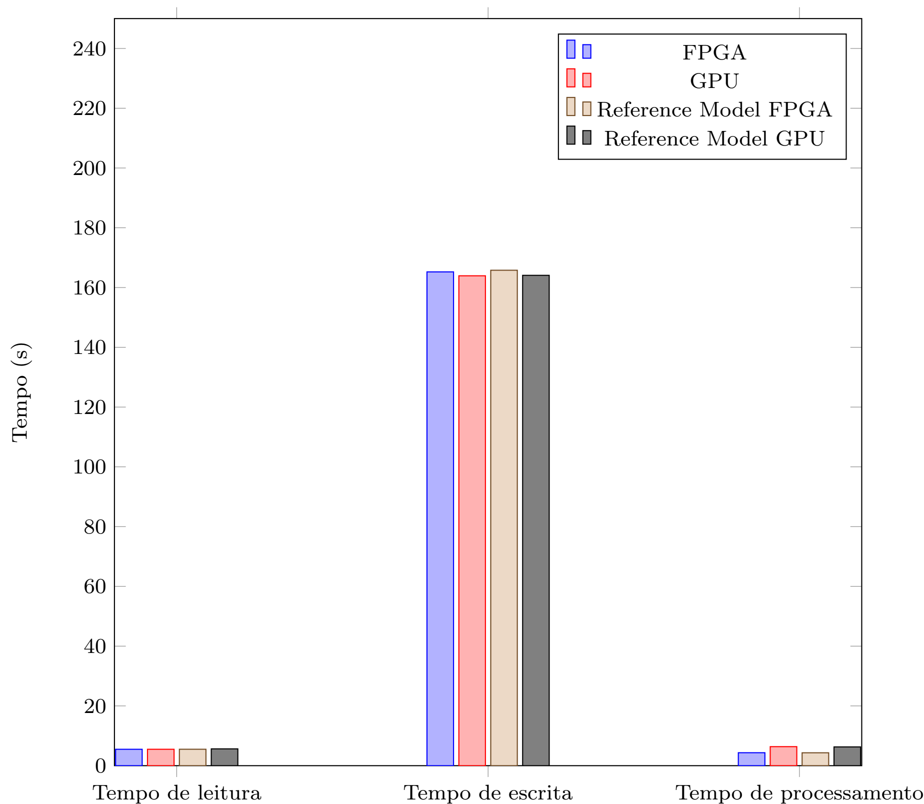
A Figura 6.24 apresenta o tempo de comunicação, escrita e processamento na configuração que utiliza quatro federados. Como esperado, o modelo de referência é o que apresenta o maior tempo de processamento. Porém, é o federado que apresenta os menores tempos de leitura e escrita. Já o federado FPGA apresenta os maiores tempos de leitura e escrita,

Figura 6.24: Análise do tempo de leitura, escrita e processamento com 4 federados



Fonte: próprio autor.

Figura 6.25: Análise do tempo de leitura, escrita e processamento com 5 federados



Fonte: próprio autor.

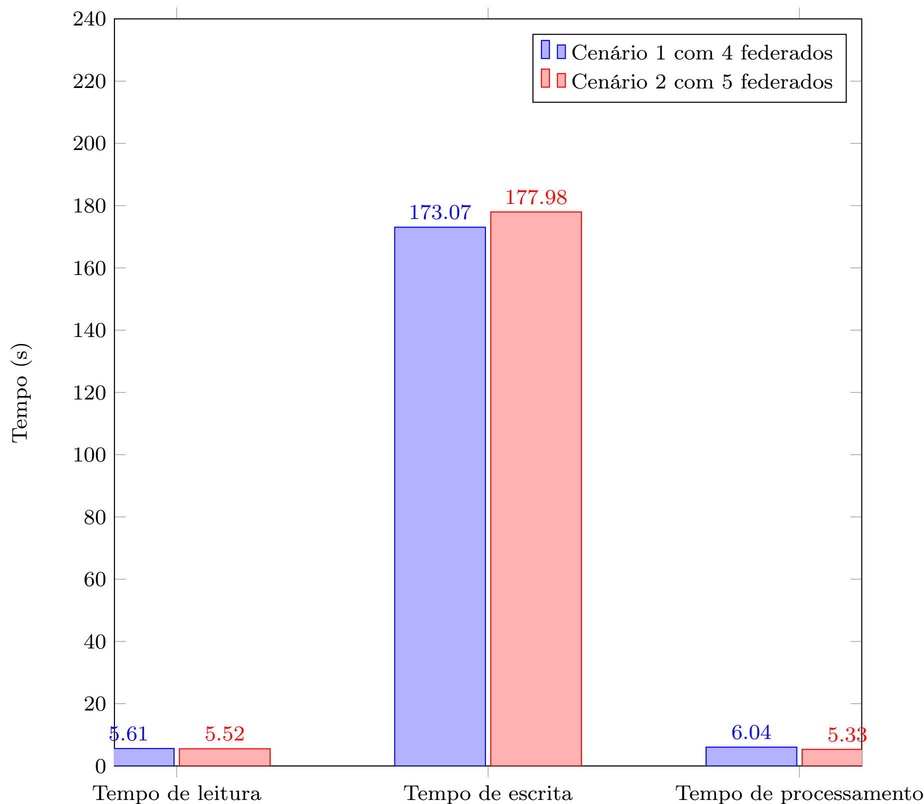
pois o mesmo depende do federado GPU. Já na Figura 6.25, é apresentado o resultado da configuração que utiliza cinco federados. Nesta figura, é possível notar que a variação do tempo entre os federados é menor que na outra configuração.

6.3.4.6 Comparação dos subsistemas em série e paralelo

A Figura 6.24 apresenta o tempo de comunicação, escrita e processamento na configuração que utiliza quatro federados. Como esperado, o modelo de referência é o que apresenta o maior tempo de processamento. Porém, é o federado que apresenta os menores tempos de leitura e escrita. Já o federado FPGA apresenta os maiores tempos de leitura e escrita, pois o mesmo depende do federado GPU. Já na Figura 6.25, é apresentado o resultado da configuração que utiliza cinco federados. Nesta figura, é possível notar que a variação do tempo entre os federados é menor que na outra configuração.

Um resumo da comparação entre às duas configurações é apresentado na Figura 6.26. Nesta figura, pode-se observar que a configuração com cinco federados apresenta um maior

Figura 6.26: Comparação to tempo de leitura, escrita e processamento dos cenários C_1 e C_2.



Fonte: próprio autor.

tempo de comunicação, mas apresenta um menor tempo de processamento.

6.4 Análise dos Resultados

A Tabela 6.4 apresenta uma comparação entre as etapas realizadas no ambiente de verificação tradicional e as etapas no ambiente proposto. Pode-se notar que existe uma mesma quantidade de etapas a serem realizadas nos dois ambientes. Às duas últimas etapas realizadas no ambiente proposto são tidas como o custo que deve ser implementado, mas isso permite integrar subsistemas em sistemas heterogêneos, com a possibilidade de reutilização de *open source* IP-*cores*, diferentemente das duas primeiras etapas do ambiente tradicional. Desta forma, no ambiente proposto pode existir uma diferença significativa no tempo de projeto. Outra vantagem na utilização do ambiente proposto é a interoperabilidade dos federados ao implementar os *wrappers*.

Tabela 6.4: Comparação entre o ambiente de verificação funcional proposto e o tradicional.

Etapas	**Ambiente Tradicional**	**Ambiente Proposto**
Reimplementação do código FPGA	X	
Reimplementação do código GPU	X	
Implementação do *reference model*	X	X
Implementação do *testbench*	X	X
Implementação dos *wrappers*		X
Integração com *Virtual Bus*/HLA		X

Com a implementação dos *wrappers* internos e externos, cria-se uma biblioteca de *wrappers* que possam ser reutilizados em diferentes implementações. Por exemplo, na implementação de um subsistema em FPGA é criado um *wrapper* interno ao *testbench* que venha a ser reutilizado, facilitando a integração de placas que tenham a configuração similar.

6.5 Considerações finais

Neste capítulo introduzem-se as etapas para construção do ambiente de verificação funcional distribuído. Essas etapas são explicadas com a estrutura do ambiente sem a validação de uma arquitetura heterogênea e depois com a validação da arquitetura. O desenvolvido é a próxima etapa, estruturando um ambiente de verificação funcional distribuído e heterogêneo que utiliza dois subsistemas para realizar a verificação de um *open source* IPs, sem a necessidade de re-codificação ao implementar *wrappers* de comunicação. No próximo capítulo apresentam-se as conclusões deste trabalho de doutorado.

Capítulo 7

Conclusões

Neste capítulo tratam-se os aspectos-chave sobre a pesquisa, assim como a principal contribuição, as limitações e sugestões para pesquisas futuras. Inicialmente, uma contextualização do fluxo de desenvolvimento de projetos voltados para circuito integrado é introduzida, isso leva a uma discussão da problemática que envolve o tempo para o desenvolvimento de tais projetos de *hardware*, com o foco voltado para os problemas relacionados na etapa de verificação funcional. Foram introduzidos os desafios na verificação, assim como os conceitos referentes ao fluxo de desenvolvimento de um projeto de *hardware*, bem como onde se localiza a parte de verificação funcional dentro desse projeto, em seguida, são tratados os conceitos necessários para o desenvolvimento deste trabalho, bem como os trabalhos relacionados e relevantes. A proposta foi tratada com explicações sobre as técnicas e passos a serem desenvolvidos. Por fim, foram mostrados os resultados referentes a este trabalho.

No processo de verificação funcional tradicional, quando se verifica um IP-*core* com o proposito de atender a especificação traçada pelo projeto, surgem vários procedimentos que podem identificar falhas, esses processos custam tempo para o projeto, muitos dos quais são necessários rever a especificação. Na proposta desta tese foram adaptadas etapas no fluxo de projeto, surgindo etapas que permitem realizar o processo de verificação funcional de forma distribuída e heterogênea, isso sem obstruir os testes durante a simulação. As etapas adicionais ao fluxo de projeto permitem integrar arquiteturas heterogêneas ao *testbench*, verificando diferentes implementações em níveis de abstrações diferentes, sem necessidade de re-codificação. O ambiente de verificação funcional pode integrar uma variedade de IP-

cores, realizando interoperabilidade ao implementar os *wrappers* de comunicação interno e externo, fazendo a comunicação por meio da API *Virtual Bus*. O experimento demonstrou um cenário onde duas implementações diferentes do algoritmo, uma da integral da imagem e outra do conversor YC_bC_r, são verificadas, uma implementada em FPGA e outra em GPU, respectivamente. Portanto, demostrou-se que a interoperabilidade das implementações com o resto do ambiente de verificação foi realizada com sucesso, mesmo com componentes de verificação em diferentes linguagens implementados em arquiteturas heterogêneas.

A pesquisa desenvolvida contempou algumas publicações de artigo para conferência, SBESC 2016 com os trabalhos Morais et al.[63] e Andrade et al.[64], o SBCCI 2018 com o trabalho Silva et al.[65] e as publicações em periódicos Silva et al.[8] em *Journal of Internet Services and Applications* (JISA) 2018 e Silva et al.[66] em *Multimedia Tools and Applications* 2021.

7.1 Contribuições da pesquisa

Baseado na hipótese deste trabalho o método permitiu intercomunicar e sincronizar diferentes arquiteturas heterogêneas com diferentes implementações mesmo sem a necessidade de re-codificação para realizar a verificação funcional. Esse método aplicado necessita de implementação de *wrappers* de comunicação, que vem a permitir que os projetistas consigam comunicar *open source* IP-*cores* implementados em arquiteturas heterogêneas, permitindo reduzir o tempo de projeto de componentes digitais.

O método proposto oferece vantagens ao integrar dispositivos usando *wrappers* implementando as interfaces para computação distribuída. Nesse caso, pode-se constatar que a API *Virtual Bus*/HLA suporta comunicação entre arquiteturas heterogêneos distribuídas, oferecendo uma forma simples e clara de troca de dados, sem a necessidade de conhecer em detalhes as arquiteturas envolvidas. Esses *wrappers* com a interface implementada podem ser adaptados para vários dispositivos, pois são baseados no padrão consolidado HLA (IEEE 1516), isso propicia na sua implementação, considerando seu principal potencial que é a possibilidade de integrar arquiteturas heterogêneas de forma transparente e síncrona. Além disso, o resultado demostrou que adicionar etapas validam a abordagem proposta ao verificar módulos desenvolvidos em diferentes dispositivos integrados ao ambiente proposto.

Com o método foi possível atender ao objetivo principal deste trabalho, ao adaptar etapas no fluxo de projeto para inserir arquiteturas heterogêneas duranta a verificação funcional. Assim, na etapa de verificação funcional foi integrado *open source* IP-*cores* em dispositivos que não tenha interfaces comuns de comunicação. Isso permite validar e verificar a integração de partes que venham ainda a ser re-codificadas, validando em estágios inicias. Além disso, na simulação não ocorreu obstrução do sincronismo durante o processo de verificação funcional distribuída e heterogênea.

Para os *wrappers* criados é possível reutilizar diferentes implementações sem a necessidade de realizar um esforço para re-codificar o *wrapper* de um determinado subsistema que seja integrado, reduzindo o esforço de desenvolvimento e tempo de projeto.

7.2 Limitações e sugestões para pesquisas futuras

Mesmo com a redução do tempo de projeto ao adaptar novas etapas, as limitações da API tornaram-se evidentes na execução da verificação funcional. Há limitação devido ao *overhead* de transmissão. HLA é uma abordagem centralizada importante para gerenciar a sincronização, mas aumenta o gargalo de comunicação. No entanto, ao demostrar que, ao usar tipos de matriz em HLA, a sobrecarga de transmissão pode ser reduzida.

O sincronismo dos dispositivos, durante a simulação, deve ser configurado conforme o percurso das transações é especificado, isso é configurado manualmente, mas pode ser automatizado para diferentes cenários, evitando processos demorados de configuração e de inserção de erros durante a simulação. Além disso, um avanço de tempo de algum federado pode ser executado mais rápido no tempo HLA podendo obstruir o sincronismo na verificação funcional.

Como trabalhos futuros existem a possibilidade de:

- Empregar o método proposto em IP-*cores* que sejam fechados;
- Além disso, outros *middlewares* de comunicação (por exemplo, DDS) podem substituir o HLA e os resultados em comparação com o método da implementação atual;
- Outra proposta, seria utilizar geradores de estímulos para verificações de módulos específicos de processamento de imagem.

Referências bibliográficas

1 LIMA, Marilia Souto Maior de. *ipProcess: um processo para desenvolvimento de IP-Cores com implementação em FPGA*. Dissertação (Mestrado) — Universidade Federal de Pernambuco, 2005.

2 MEHTA, Ashok B. *ASIC/SoC Functional Design Verification : A Comprehensive Guide to Technologies and Methodologies*. [S.l.: s.n.]. ISBN 978-3-319-59418-7,3319594184,978-3-319-59417-0.

3 MOHAMED, Khaled Salah. *IP Cores Design from Specifications to Production: Modeling, Verification, Optimization, and Protection*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2015. ISBN 3319220349.

4 SILVA, Karina Rocha Gomes da. Uma metodologia de verificação funcional para circuitos digitais. Tese de Doutorado, 2007.

5 SEOK, Moon Gi; KIM, Tag Gon; PARK, Daejin. An hla-based formal co-simulation approach for rapid prototyping of heterogeneous mixed-signal socs. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, v. 100-A, n. 7, p. 1374–1383, 2017.

6 CHEN, W.; RAY, S.; BHADRA, J.; ABADIR, M.; WANG, L. Challenges and trends in modern soc design verification. *IEEE Design Test*, v. 34, n. 5, p. 7–22, Oct 2017. ISSN 2168-2356.

7 JUNIOR, José Cláudio Vieira e Silva. Verificação de projetos de sistemas embarcados através de cossimulação hardware/software. Dissertação de Mestrado, 2015.

8 SILVA, Thiago W. B.; MORAIS, Daniel C.; ANDRADE, Halamo G. R.; LIMA, Antonio M. N.; MELCHER, Elmar U. K.; BRITO, Alisson V. Environment for integration of distributed heterogeneous computing systems. *Journal of Internet Services and Applications*, v. 9, n. 1, p. 4, Jan 2018. ISSN 1869-0238. Disponível em: <https://doi.org/10.1186/s13174-017-0072-1>.

9 IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, p. 1–638, Jan 2012.

10 SEOK, M. G.; KIM, T. G.; CHOI, C. B.; PARK, D. An hla-based distributed cosimulation framework in mixed-signal system-on-chip design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 25, n. 2, p. 760–764, Feb 2017. ISSN 1063-8210.

11 BRITO, A. V.; BUCHER, H.; OLIVEIRA, H.; COSTA, L. F. S.; SANDER, O.; MELCHER, E. U. K.; BECKER, J. A distributed simulation platform using hla for complex embedded systems design. In: *2015 IEEE/ACM 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. [S.l.: s.n.], 2015. p. 195–202. ISSN 1550-6525.

12 TRAN, H. V..; TRUONG, T. P.; NGUYEN, K. T.; HUYNH, H. X.; POTTIER, B. Context-aware systems and applications: 4th international conference, iccasa 2015, vung tau, vietnam, november 26-27, 2015, revised selected papers. In: ______. Cham: Springer International Publishing, 2016. cap. A Federated Approach for Simulations in Cyber-Physical Systems, p. 165–176. ISBN 978-3-319-29236-6.

13 HEKMATPOUR, A.; ALLEY, C.; STEMPEL, B.; COULTER, J.; SALEHI, A.; SHAFIE, A.; PALENCHAR, C. "A heterogeneous functional verification platform". *Proceedings of the IEEE 2005 Custom Integrated Circuits Conference, 2005.*, p. 63–66, Sept 2005. ISSN 0886-5930.

14 MUHR, H.; HOLLER, R.; HORAUER, M. "A Heterogeneous Hardware-Software Co-Simulation Environment Using User Mode Linux and Clock Suppression". *2006 2nd IEEE/ASME International Conference on Mechatronics and Embedded Systems and Applications*, p. 1–6, Aug 2006.

15 BARNASCONI, M.; DIETRICH, M.; EINWICH, K.; VöRTLER, T.; CHAPUT, J. P.; LOUëRAT, M. M.; PêCHEUX, F.; WANG, Z.; CUENOT, P.; NEUMANN, I.; NGUYEN, T.; LUCAS, R.; VAUMORIN, E. "UVM-SystemC-AMS Framework for System-Level Verification and Validation of Automotive Use Cases". *IEEE Design Test*, v. 32, n. 6, p. 76–86, Dec 2015. ISSN 2168-2356.

16 DUENAS, C. A. M. Verification and test challenges in soc designs. *Proceedings. SBCCI 2004. 17th Symposium on Integrated Circuits and Systems Design (IEEE Cat. No.04TH8784)*, p. 9–, Sept 2004.

17 SILVA, T. W. B.; MORAIS, D. C.; ANDRADE, H. G. R.; LIMA, A. M. N.; MELCHER, E. U. K.; BRITO, A. V. Environment for integration of distributed heterogeneous computing systems. *Journal of Internet Services and Applications*, 2018.

18 MANNOS, Tom J; DZIKI, Brian; SHARIF, Moslema. Fault testing a synthesizable embedded processor at gate level using ultrascale fpga emulation. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. [S.l.: s.n.], 2019. p. 116–116.

19 WILE, Bruce; GOSS, John; ROESNER, Wolfgang. *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 0127518037.

20 VASUDEVAN, Srivatsa. *Effective functional verification principles and processes*. 1st ed. 2006.. ed. [S.l.]: Springer US, 2006. 256 p.

21 ALBERTINI, Bruno de Carvalho et al. Metodologias de suporte a verificação e análise de modelos de plataformas em alto nível de abstração. In: . [S.l.: s.n.].

22 BERGERON, J. *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. Norwell, MA, USA: Kluwer Academic Publishers, 2003. ISBN 1402074018.

23 BHADRA, J.; ABADIR, M. S.; WANG, L.; RAY, S. A survey of hybrid technqiues for functional verification. *IEEE Design & Test of Computers*, IEEE Computer Society, v. 24, n. 2, p. 112–122, 2007.

24 ACCELLERA. Universal Verification Methodology (UVM) 1.2 User's Guide. *Acesso em: 22 de Fevereiro de 2018*, p. 61–73, 2015. Disponível em: <http://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf>.

25 CAMARA, Rômulo Calado Pantaleão. Ovm_tpi: Uma metodologia de verificação funcional para circuitos digitais. Dissertação de Mestrado, 2011.

26 OLIVEIRA, H. F. D. A. Reformulação, baseada em ovm, da metodologia de verificação funcional verisc. Dissertação de Mestrado, 2010.

27 Prado, B.; Barros, E.; Silva, L.; Cabrai, A.; Bruno, R.; demente, G. Ivm: An interoperable verification methodology for iterative and incremental digital system design. In: *2009 17th IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*. [S.l.: s.n.], 2009. p. 207–210.

28 OLIVEIRA, Helder Fernando de Araújo. Bvm: Reformulação da metodologia de verificação funcional verisc. Dissertação de Mestrado, 2010.

29 WINDOWS, MS. *MS Windows NT Kernel Description*. Acessado em: 20 de fevereiro de 2020. Disponível em: <https://www.chipverify.com/systemverilog/systemverilog-simple-testbench>.

30 FUJIMOTO, R. M. Parallel and distributed simulation systems. New York: John Wiley & Sons, 2000.

31 GOLDSMITH, Andrea. *Wireless Communications*. 2005.

32 SIMO, R.; SANTOS, L. H. S.; BRITO, A. V. An adaptive approach for real-time communication of multi-robots based on hla. In: *2015 Latin American Network Operations and Management Symposium (LANOMS)*. [S.l.: s.n.], 2015. p. 92–98.

33 JUNIOR, Jose Claudio VS; BRITO, Alisson V; COSTA, Luis Feliphe Silva; NASCIMENTO, Tiago P; MELCHER, Elmar Uwe Kurt. Testing real-time embedded systems using high level architecture. *Design Automation for Embedded Systems*, Springer, v. 20, n. 4, p. 289–309, 2016.

34 SOCIETY, IEEE COMPUTER. Std. 1516-2000. ieee standard for modeling and simulation (m&s) high level architecture (hla) - framework and rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, 1516.ed. New York: IEEE, p. 1–38, Aug 2010.

35 SOCIETY, IEEE COMPUTER. Std. 1516.2-2000. ieee standard for modeling and simulation (m&s) high level architecture (hla) -object model template (omt) specification. *IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000)*, 1516.2. ed. New York: IEEE, p. 1–110, Aug 2010.

36 SOCIETY, IEEE COMPUTER. Std. 1516.1-2000. ieee standard for modeling and simulation (m&s) high level architecture (hla) - federate interface specification. *IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000)*, 1516.1. ed. New York: IEEE, p. 1–378, Aug 2010.

37 AWAIS, M. U.; PALENSKY, P.; ELSHEIKH, A.; WIDL, E.; MATTHIAS, S. The high level architecture rti as a master to the functional mock-up interface components. *Computing, Networking and Communications (ICNC), 2013 International Conference on*, p. 315–320, Jan 2013.

38 TOPÇU, Okan; DURAK, Umut; OĞUZTÜZÜN, Halit; YILMAZ, Levent. Distributed simulation: A model driven engineering approach. In: ______. Cham: Springer International Publishing, 2016. cap. High Level Architecture, p. 39–66. ISBN 978-3-319-03050-0.

39 STRASSBURGER, Steffen. Distributed simultation based on the high level architecture in civilian application domains. In: . [s.n.], 2002. ISBN 1-56555-218-0. Disponível em: <http://d-nb.info/964727544>.

40 OLIVEIRA, Helder Fernando de Araújo. Uma abordagem para estimação do consumo de energia em modelos de simulação distribuída. Universidade Federal de Campina Grande, 2015.

41 KUHL, F.; WEATHERLY, R.; DAHMANN, J. Creating computer simulation systems: An introduction to the high level architecture. In: . [S.l.]: Saddle River: Prentice Hall, 1999. v. 1st. ed.

42 SOCIETY, IEEE COMPUTER. Ieee recommended practice for high level architecture (hla) federation development and execution process (fedep). 1516.3. ed. New York: IEEE, 2003.

43 GRAHAM, P. Ansi common lisp. Saddle River: Prentice Hall, 1995.

44 GUPTA, P. Resource-constraint and scalable data distribution management for high level architecture. Orlando: University of Central Florida, 2007.

45 KRÜGER, I. H.; MEISINGER, M.; MENARINI, M. Interaction-based runtime verification for systems of systems integration. *J. Log. and Comput.*, Oxford University Press, Oxford, UK, v. 20, n. 3, p. 725–742, jun. 2010. ISSN 0955-792X. Disponível em: <http://dx.doi.org/10.1093/logcom/exn079>.

46 RASHMI, V S; SOMAYAJI, Giridhar; BHAMIDIPATHI, Sirisha. A methodology to reuse random ip stimuli in an soc functional verification environment. In: *2015 19th International Symposium on VLSI Design and Test*. [S.l.: s.n.], 2015. p. 1–5.

47 BRODTKORB, Andre R.; DYKEN, Christopher; HAGEN, Trond R.; HJELMERVIK, Jon M.; STORAASLI, Olaf O. State-of-the-art in heterogeneous computing. *Sci. Program.*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 18, n. 1, p. 1–33, jan. 2010. ISSN 1058-9244. Disponível em: <http://dx.doi.org/10.1155/2010/540159>.

48 WETTER, M.; HAVES, P. A modular building controls virtual test bed for the integration of heterogeneous systems. *Proceedings of the 3rd SimBuild Conference*, p. 69–76, 2008. Disponível em: <http://simulationresearch.lbl.gov/wetter/download/SB08-04-2-Wetter.pdf>.

49 NEEMA, H. Large-scale integration of heterogeneous simulations. *Faculdade da Escola de Pós-Graduação da Universidade Vanderbilt*, 2018.

50 GERVAIS, C.; CHAUDRON, J. B.; SIRON, P.; LECONTE, R.; SAUSSIé, D. Real-time distributed aircraft simulation through hla. *Distributed Simulation and Real Time Applications (DS-RT), 2012 IEEE/ACM 16th International Symposium on*, p. 251–254, Oct 2012. ISSN 1550-6525.

51 VINEETH, B.; SUNDARI, B. Bala Tripura. Uvm based testbench architecture for coverage driven functional verification of spi protocol. In: *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. [S.l.: s.n.], 2018. p. 307–310.

52 CERTI. jul. 2016. Disponível em: <http://savannah.nongnu.org/projects/certi>.

53 SIRON, Pierre. Design and implementation of a hla rti prototype at onera. *1998 Fall Simulation Interoperability Workshop, 98F-SIW-036, ftp://ftp. cert. fr/pub/siron/98f-siw-036. ps*, 1998.

54 IETF. Rfc 1321 – the md5 message-digest algorithm. April 1992. Disponível em: <https://tools.ietf.org/html/rfc1321>.

55 STONE, J. E.; GOHARA, D.; SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, v. 12, n. 3, p. 66–72, 2010. ISSN 15219615.

56 CUMMINS, Chris; PETOUMENOS, Pavlos; STEUWER, Michel; LEATHER, Hugh. Autotuning opencl workgroup size for stencil patterns. *ADAPT: International Workshop on Adaptive Self-tuning Computing Systems*, p. 8, 2016. Disponível em: <http://arxiv.org/abs/1511.02490>.

57 GONZALEZ, R. C.; WOODS, R. E. *Digital image processing*. Upper Saddle River, N.J.: Prentice Hall, 2008. ISBN 9780131687288 013168728X 9780135052679 013505267X. Disponível em: <http://www.amazon.com/Digital-Image-Processing-3rd-Edition/dp/013168728X>.

58 CROW, Franklin C. Summed-area tables for texture mapping. In: *SIGGRAPH*. [S.l.]: ACM, 1984. p. 207–212.

59 VIOLA, P.; JONES, M. Rapid object detection using a boosted cascade of simple features. *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, v. 1, p. I–511–I–518 vol.1, 2001. ISSN 1063-6919.

60 CAMPOS, Nelson Carlos de Sousa. Uma metodologia de projeto e validação de sistemas de detecção de faces. Dissertação de Mestrado, 2017.

61 TERASIC. *TERASIC DE1-SoC User Manual*. Acessado em: 22 de junho de 2020. Disponível em: <https://courses.cs.washington.edu/courses/cse467/15wi/docs/DE1_SoC_User_Manual.pdf>.

62 AVALON INTERFACE SPECIFICATIONS. nov 2017. Disponível em: <https://www.altera.com/documentation/nik1412467993397.html>.

63 MORAIS, D. C.; SILVA, T. W. B.; NASCIMENTO, T. P.; MELCHER, E. U. K.; BRITO, A. V. A distributed platform for integration of fpga-based embedded systems. *2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC)*, p. 86–92, Nov 2016. ISSN 2324-7894.

64 ANDRADE, H. G. R.; MORAIS, D.; SILVA, T. W. B.; NASCIMENTO, T. P.; BRITO, A. V. The integration of gpu-based and heterogeneous devices using hla. *2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC)*, p. 162–167, Nov 2016. ISSN 2324-7894.

65 SILVA, T. W. B.; MORAIS, D. C.; ANDRADE, H. G. R.; NUNES, F.; LIMA, A. M. N.; MELCHER, E. U. K.; BRITO, A. V. A distributed functional verification environment for the design of system-on-chip in heterogeneous architectures. *SBCCI 2018 Final Technical Program. Reliability & Verification*, 2018. No prelo.

66 SILVA, Thiago W. B.; ANDRADE, Halamo G. R.; LIMA, Antonio M. N.; MELCHER, Elmar U. K.; BRITO, Alisson V. An image generator based on neural networks in gpu. *Multimedia Tools and Applications*, v. 9, n. 1, p. 1008, Set 2021. ISSN 1573-7721. Disponível em: <https://doi.org/10.1007/s11042-021-11489-5>.