

Luis Gustavo Perpetuo Costa Marques

**METODOLOGIA DE DESENVOLVIMENTO DE VHDL
SINTETIZÁVEL COM USO DE MODEL CHECKING**

Dissertação submetida ao Programa
de Pós-Graduação em Engenharia de
Automação e Sistemas para a obtenção
do Grau de Mestre em Engenharia de
Automação e Sistemas.

Orientador: Prof. Dr. Max Hering de
Queiroz

Coorientador: Prof. Dr. Jean-Marie
Farines

Florianópolis

2016

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

Marques, Luis Gustavo Perpetuo Costa
Metodologia de desenvolvimento de VHDL sintetizável com
uso de model checking / Luis Gustavo Perpetuo Costa
Marques ; orientador, Max Hering de Queiroz ;
coorientador, Jean-Marie Farines. - Florianópolis, SC, 2016.
185 p.

Dissertação (mestrado profissional) - Universidade
Federal de Santa Catarina, Centro Tecnológico. Programa de
Pós-Graduação em Engenharia de Automação e Sistemas.

Inclui referências

1. Engenharia de Automação e Sistemas. 2. VHDL. 3. Model
Checking. 4. OVL. 5. FIACRE. I. Queiroz, Max Hering de .
II. Farines, Jean-Marie. III. Universidade Federal de
Santa Catarina. Programa de Pós-Graduação em Engenharia de
Automação e Sistemas. IV. Título.

Luis Gustavo Perpetuo Costa Marques

METODOLOGIA DE DESENVOLVIMENTO DE VHDL SINTETIZÁVEL COM USO DE MODEL CHECKING

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

Florianópolis, 9 de Março de 2016.

Prof. Dr. Rômulo Silva de Oliveira
Coordenador do Curso

Prof. Dr. Max Hering de Queiroz
DAS-UFSC
Orientador

Prof. Dr. Jean-Marie Farines
DAS-UFSC
Coorientador

Banca Examinadora:

Prof. Dr. Max Hering de Queiroz
DAS-UFSC
Presidente

Prof. Dr. Djones Vinicius Lettnin
EEL-UFSC

Prof. Dr. Fabio Baldissera
DAS-UFSC

Prof. Dr. Leandro Buss Becker
DAS-UFSC

Eng. Celso Luis de Souza
Reason Tecnologia - General Eletric

Este trabalho é dedicado a minha família,
cujo suporte me permitiu chegar até aqui
e completar mais essa etapa da minha
formação.

AGRADECIMENTOS

À minha família pelo eterno apoio, em especial aos meus pais que se mudaram para Florianópolis para me auxiliar nessa etapa.

Aos meus amigos, pelos momentos sempre agradáveis de descontração que passamos juntos.

Aos meus orientadores Max e Jean-Marie, pela excelente orientação e contribuições fundamentais para este trabalho.

Aos professores do DAS, que cuidaram de toda minha formação acadêmica.

À Reason Tecnologia pelo completo apoio a pós-graduação, liberando as horas necessárias para realização de aulas e atividades relacionadas a dissertação, além do auxílio financeiro.

RESUMO

Essa dissertação foi elaborada em uma companhia que desenvolve equipamentos para proteção e automação de subestações, sendo que a maior parte deles possui um FPGA programado em VHDL como unidade principal de processamento. O código VHDL sintetizável é validado através de simulação e testes em equipamento, método bastante comum mas que não é suficiente para garantir a satisfação de propriedades tanto gerais quanto orientadas à aplicação, devido ao fato de não ser exaustivo. Na direção de aumentar a confiabilidade do circuito projetado para o FPGA, o objetivo principal da dissertação é apresentar uma metodologia de desenvolvimento de código VHDL sintetizável que aprimore as atuais técnicas utilizadas, ao incorporar métodos formais para verificação de propriedades, sendo que o método formal utilizado é o *model checking*. A metodologia é construída de um modo que o uso do *model checking* seja transparente ao desenvolvedor VHDL, mantendo a interface com o processo de verificação formal em linguagem de usuário, evitando a necessidade de aprendizado de novas linguagens. Para atingir esse objetivo específico, é proposto que as propriedades sejam representadas através de padrões orientados a VHDL que são baseados na biblioteca OVL. Além disso, os contraexemplos gerados no processo de *model checking* retornam como *testbench* VHDL, permitindo ao usuário identificar o comportamento indesejado através de simulação. O ambiente de verificação adotado utiliza modelos em linguagem intermediária FIACRE como *front-end* e por isso são propostas regras de tradução VHDL-FIACRE para que a transformação possa ocorrer no contexto de engenharia dirigida a modelos e assim evitar erros no processo de tradução. O uso da linguagem intermediária é vantajoso, pois permite a utilização das ferramentas de verificação, as quais são de código aberto, sem que seja necessária a tradução direta do VHDL para os formalismos matemáticos em que essas ferramentas se baseiam. A metodologia é validada com a aplicação em quatro exemplos de código VHDL, sendo dois deles utilizados em projetos desenvolvidos na empresa: uma função de proteção e um controlador de acesso a um barramento de transferência de dados. Os resultados da aplicação indicam que a proposta é viável, pois foi possível fazer a verificação dos exemplos, sendo que em um deles foi identificado um erro que havia passado despercebido por simulação, sinalizando que a proposta contribui no aumento da confiabilidade do código desenvolvido.

Palavras-chave: VHDL; *Model Checking*; OVL; *Model Driven Engineering*; FIACRE; FPGA

ABSTRACT

This dissertation was elaborated in a company that develops equipment for substation protection and automation, most of them having an FPGA programmed in VHDL as the main processing unit. The synthesizable VHDL code is validated through simulation and tests on equipment, a fairly common method that is not enough to ensure the satisfaction of both general and application-oriented properties, due to the fact of being non exhaustive. In the direction of increasing the reliability of the designed FPGA circuit, the main objective of this work is to present a synthesizable VHDL code development methodology that enhances the current techniques by incorporating formal methods for verification of properties, with model checking being the selected method. The methodology is constructed in such a way that the use of model checking procedure should be transparent to VHDL designers, keeping the interface with the formal verification process in user language, avoiding the need to learn new languages. To achieve this specific objective, it is proposed that the properties are represented by VHDL oriented patterns based on OVL library. In addition, the counterexamples generated in the model checking process for properties that failed, return as VHDL testbench, allowing the user to identify the undesired behavior through simulation. The verification environment used in the methodology requires models described with the intermediate language FIACRE as front-end and so VHDL-FIACRE translation rules are proposed to allow the transformation to occur in the context of model driven engineering, and thus prevent errors in the translation process. The use of an intermediate language is advantageous because it allows the use of the verification tools, which are open source, without the need of translating VHDL directly to the mathematical formalism in which these tools are based. The methodology is validated by the application in four examples of VHDL code, two of them are used in designs developed by the company: a protection function and a controller to access a data transfer bus. The application results indicate that the proposal is viable because it was possible to verify the examples, and for one of them was identified an error that had passed unnoticed by simulation, showing that the proposal contributes to increase the reliability of the created code.

Keywords: VHDL; Model Checking; OVL; Model Driven Engineering; FIACRE; FPGA

LISTA DE FIGURAS

Figura 1	Execução do código VHDL em simulação	31
Figura 2	Máquina de estados do controlador de memória	33
Figura 3	Metodologia atual de desenvolvimento VHDL	36
Figura 4	Metodologia atual de desenvolvimento VHDL com o uso de observadores	40
Figura 5	Esquemático da abordagem por <i>model checking</i>	45
Figura 6	Metodologia de Desenvolvimento VHDL proposta (visão geral)	49
Figura 7	Metodologia de Desenvolvimento VHDL proposta (FIACRE)	51
Figura 8	Cadeia de verificação genérica (SOUZA, 2010)	52
Figura 9	Cadeia de verificação proposta	53
Figura 10	Processo de Tradução de Contraexemplo	56
Figura 11	Transformação M2M (MDA, 2003)	63
Figura 12	Modelo Geral de Transformação VHDL-FIACRE	68
Figura 13	Esqueleto do processo FIACRE <i>delta_cycle</i>	69
Figura 14	Diagrama de blocos do VHDL do controlador de memória	79
Figura 15	Exemplo de propriedade com <i>clock</i>	86
Figura 16	Comparação de uso do OVL como <i>assert</i> entre o convencional e a proposta	93
Figura 17	Comparação de uso do OVL como <i>assume</i> entre o convencional e a proposta	94
Figura 18	Contraexemplo para propriedade 3 do controlador de memória (ferramenta ModelSim (MODELSIM, 2015))	113
Figura 19	Diagrama de funcionamento da PTOV	114
Figura 20	Contraexemplo para propriedade 1 da PTOV (figura extraída da ferramenta ModelSim (MODELSIM, 2015))	125
Figura 21	Transmissão de pacotes por meio do barramento Avalon-ST	175
Figura 22	Contraexemplo para propriedade 5 do <i>Goertzel Control</i> (ferramenta ModelSim (MODELSIM, 2015))	185

LISTA DE TABELAS

Tabela 1	Formato dos estímulos de entrada	57
Tabela 2	Observadores OVL traduzidos	106
Tabela 3	Resumo dos exemplos de aplicação	107

LISTA DE ABREVIATURAS E SIGLAS

ASIC	Application Specific Integrated Circuit	21
ATL	Atlas Transformation Language	21
CTL	Computation Tree Logic	21
DUT	Device Under Test	21
FIACRE	Format Intermédiaire pour les Architectures de Composants Répartis Embarqués	21
FPGA	Field Programmable Gate Array	21
HDL	Hardware Description Language	21
LTL	Linear Temporal Logic	21
MDE	Model Driven Engineering	21
OVL	Open Verification Library	21
PSL	Property Specification Language	21
RTL	Register Transfer Level	21
SVA	System Verilog Assertion	21
VHDL	Very High Speed Integrated Circuits HDL	21

SUMÁRIO

1	INTRODUÇÃO	21
1.1	MOTIVAÇÃO	22
1.2	OBJETIVO	24
1.2.1	Objetivos Especificos	24
1.3	ORGANIZAÇÃO DO TRABALHO	25
2	METODOLOGIA DE DESENVOLVIMENTO DE CÓDIGO VHDL	27
2.1	VHDL	27
2.1.1	Estilo de Codificação	31
2.1.2	Análise de Propriedades	35
2.2	METODOLOGIA DE DESENVOLVIMENTO ATUAL ..	36
2.2.1	Desenvolvimento com uso de observadores	39
2.3	CONCLUSÃO	41
3	METODOLOGIA PROPOSTA	43
3.1	MODEL CHECKING E LÓGICAS TEMPORAIS	44
3.2	METODOLOGIA PROPOSTA	47
3.3	AMBIENTE DE VERIFICAÇÃO	52
3.3.1	Compilador FIACRE/TTS e a ferramenta TINA ..	54
3.3.2	Tradução de Contraexemplo	55
3.4	TRABALHOS RELACIONADOS	57
3.5	CONCLUSÃO	60
4	MODELO DE TRADUÇÃO VHDL-FIACRE	61
4.1	ENGENHARIA DIRIGIDA A MODELOS	61
4.2	FIACRE	63
4.3	MODELO DE TRADUÇÃO VHDL-FIACRE	66
4.3.1	Tradução de Declarações Concorrentes	74
4.3.2	Tradução de Tipos e Operadores	77
4.4	EXEMPLO	78
4.4.1	Modelo FIACRE	79
4.5	CONCLUSÃO	83
5	MODELO DE VERIFICAÇÃO	85
5.1	CLOCKED LTL (@LTL)	85
5.1.1	Tradução @LTL-LTL	87
5.2	OVL	88
5.2.1	Aplicação convencional do OVL	90
5.2.2	Aplicação proposta do OVL	92
5.2.3	Restrições da tradução do OVL para a metodologia	95

5.3	TRADUÇÃO OVL-LTL.....	96
5.3.1	ovl_always.....	98
5.3.2	ovl_window.....	98
5.3.3	ovl_next.....	99
5.4	MODELO INPUT_GENERATOR COM RESTRIÇÕES OVL.....	100
5.4.1	Estrutura do input_generator.....	101
5.4.2	ovl_always.....	103
5.4.3	ovl_window.....	103
5.4.4	ovl_next.....	104
5.5	OBSERVADORES TRADUZIDOS.....	105
5.6	CONCLUSÃO.....	105
6	APLICAÇÃO DA METODOLOGIA PROPOSTA	107
6.1	EXEMPLO CONTROLADOR DE MEMÓRIA.....	108
6.1.1	Propriedades do controlador de memória.....	109
6.1.2	Resultados da Verificação.....	111
6.2	EXEMPLO FUNÇÃO DE PROTEÇÃO.....	113
6.2.1	Modelo FIACRE da PTOV.....	116
6.2.2	Propriedades da PTOV.....	120
6.2.3	Resultados da Verificação.....	123
6.3	EXEMPLO GOERTZEL CONTROL.....	126
6.4	CONCLUSÃO.....	128
7	CONCLUSÃO	129
	REFERÊNCIAS	133
	APÊNDICE A – Códigos para o Controlador de Memória	139
	APÊNDICE B – Tradução OVL	147
	APÊNDICE C – Códigos da Função de Proteção PTOV	163
	APÊNDICE D – Exemplo Goertzel Control	175

1 INTRODUÇÃO

As linguagens de descrição de hardware (*Hardware Description Language - HDL*) (MICHELI, 1994) surgiram para suprir a necessidade de se ter uma ferramenta para modelar e especificar hardware, em especial circuitos digitais. O VHDL (*Very High Speed Integrated Circuits HDL*) (VHDL, 2008) é uma HDL que foi desenvolvida pelo Departamento de Defesa (DoD) dos EUA para ser uma linguagem a ser utilizada para descrever o comportamento de ASICs (*Application Specific Integrated Circuit*) a serem fabricados por empresas fornecedoras. Em 1987, o VHDL se tornou um padrão IEEE e desde então seu uso tem se tornado cada vez mais intenso, especialmente devido à utilização para programação de FPGAs (*Field Programmable Gate Arrays*) (BROWN et al., 2012).

Os FPGAs são circuitos integrados programáveis, projetados para serem configurados pelo usuário para realizar os mais diversos tipos de componentes, desde uma simples memória ROM até um processador. Desde seu surgimento, em meados dos anos 80, eles têm se tornado maiores (mais portas lógicas e registradores) e mais rápidos, sendo capazes de reproduzirem circuitos digitais cada vez mais complexos e por isso sendo cada vez mais utilizados. O uso de FPGAs é bastante atraente em sistemas embarcados de tempo real devido ao fato de serem circuitos digitais programáveis, o que permite que sejam criados circuitos específicos para realização das tarefas que requerem determinismo e rapidez na execução, de um modo mais rápido e com menor custo.

O uso de FPGAs, em especial para sistemas de tempo real crítico, exige uma garantia de correção lógica e temporal dos circuitos projetados para que se aumente a confiabilidade dos sistemas. Portanto, é cada vez mais necessário o uso de formalismos e de técnicas matemáticas que permitem analisar, verificar e provar que os modelos dos circuitos, descritos por meio da HDL utilizada na programação do FPGA, estejam corretos. No entanto, os métodos de desenvolvimento de HDL atualmente utilizados, em sua maioria, se baseiam em codificação a partir de especificações muitas vezes descritas de modo informal e em validação por simulação. Esse modelo de validação não consegue garantir propriedades que são comumente demandadas em circuitos digitais como, por exemplo, ausência de *deadlocks*, exclusão mútua, sequência de eventos, limites de valores etc.

Vários estudos na literatura propõem soluções para aprimorar o

desenvolvimento de código HDL através da adição de verificação formal ao processo de desenvolvimento. No entanto, essas metodologias são difíceis de serem incorporadas pelas empresas pelo fato de exigirem que o desenvolvedor tenha que adquirir conhecimento de linguagens e ferramentas, muitas vezes complexas, às quais não estão acostumados. Algumas fabricantes de software de verificação já possuem ferramentas que conseguem fazer verificação formal de HDLs em um ambiente mais amigável para o desenvolvedor. Porém, essas soluções proprietárias são muito caras, com licenças anuais custando mais que 100 mil dólares, o que não é compatível com a realidade financeira de pequenas e médias empresas fazendo com que elas usualmente não utilizem essas ferramentas.

1.1 MOTIVAÇÃO

A ideia para o desenvolvimento desse trabalho partiu de experiências no ambiente de trabalho. No dia-a-dia do trabalho em uma empresa que fabrica equipamentos para proteção e automação de subestações de energia elétrica ¹, vivenciou-se a experiência de desenvolvimento de código VHDL sintetizável para FPGAs em sistemas embarcados de tempo real. A empresa necessita aumentar a confiabilidade dos equipamentos e portanto precisa melhorar o processo de desenvolvimento VHDL, de tal forma que se possa garantir formalmente o atendimento de propriedades.

O atual processo de desenvolvimento de código VHDL adotado na empresa começa com uma especificação informal do circuito a ser implementado. Essa especificação é criada por profissionais do setor de Marketing, contendo textos e diagramas que descrevem o comportamento desejado. A partir dessa especificação, o desenvolvedor gera o código VHDL sintetizável que descreve o circuito. Esse código é validado por simulação, onde o desenvolvedor estimula o circuito com o objetivo de reproduzir cenários de funcionamento. Os resultados da simulação são analisados para concluir se o circuito se comportou corretamente para as situações reproduzidas e se as propriedades foram satisfeitas. Tanto a funcionalidade quanto as propriedades são informações extraídas da especificação inicial. Quando o código VHDL é aprovado em simulação, ele é sintetizado em FPGA para que sejam feitos testes reais com equipamento.

Simulação e testes reais em equipamento podem até serem su-

¹Reason Tecnologia S.A - <http://reason.com.br/>

ficientes para validar se o código VHDL implementa a funcionalidade desejada, no entanto, não são suficientes para garantir o atendimento de propriedades por não fazerem uma verificação exaustiva. A falta de garantia de satisfação de propriedades pode fazer com que muitos problemas sejam detectados apenas após a implementação de protótipo ou até mesmo correndo o risco de aparecer após a instalação para o cliente, o que gera prejuízos.

A incorporação de verificação formal ao atual processo de desenvolvimento aumenta a confiabilidade do código VHDL desenvolvido ao garantir, por meio métodos exaustivos de exploração do espaço de estados do sistema, que as propriedades são satisfeitas. Uma solução para adoção de verificação ao atual processo poderia ser a utilização de ferramentas proprietárias que disponibilizam verificação para VHDL (QUESTA, 2014) (CADENCE, 2014), no entanto, o alto custo desses *softwares* inviabilizam a adoção pela empresa. Além das ferramentas proprietárias, existem outros trabalhos disponíveis na literatura que propõem métodos e até mesmo oferecem ferramentas abertas para verificação de VHDL. Porém, essas soluções costumam exigir conhecimento que os desenvolvedores não possuem, em especial de métodos e linguagens formais. O custo elevado de ferramentas e a necessidade de aprendizado sobre métodos e linguagens formais são problemas comuns no ambiente de desenvolvimento para VHDL em geral e também na empresa onde o trabalho foi realizado. Portanto, uma solução que ofereça uma ferramenta aberta e que necessite o uso apenas de linguagens conhecidas pelo usuário é uma contribuição para indústria.

A pouca utilização de métodos formais também acontece para processos de desenvolvimento em outras linguagens (programação ou de definição de modelos de alto nível) e em outros setores da indústria (aeronáutica, defesa, transportes, nuclear etc.), e por isso alguns trabalhos, desenvolvidos junto ao projeto TOPCASED (TOPCASED, 2014) ou relacionados com resultados do mesmo, propõem soluções para inclusão de verificação formal no desenvolvimento para diversas linguagens (FARINES et al., 2011) (BERTHOMIEU et al., 2009). Nesses trabalhos, o objetivo é propor soluções em que o desenvolvedor possa fazer a verificação, tendo acesso a ferramentas construídas a partir de modelos formais, porém utilizando apenas as linguagens (do usuário, de seu domínio) que ele conhece. Para isso são propostas cadeias de verificação em que a partir de sucessivas transformações, é possível obter um modelo que sirva de entrada para ferramentas de verificação e que represente corretamente o modelo em linguagem de usuário, alvo da verificação. Em comum, essas cadeias utilizam a linguagem intermediária

FIACRE e ferramentas *Open Source*. Esses trabalhos também fornecem meios para que o desenvolvedor descreva as propriedades sem precisar ter contato com a complexidade das ferramentas.

O projeto TOPCASED foi concebido para atender o processo de desenvolvimento de sistemas embarcados que integra diferentes tipos de modelos e ferramentas, levando em consideração as características distintas destes modelos e também o longo tempo de vida dos produtos. O projeto coloca à disposição da comunidade um conjunto de ferramentas de engenharia de sistemas *Open Source* para o desenvolvimento de sistemas críticos de tempo real (SAAD et al., 2008). Embora o projeto tenha se encerrado, as ferramentas e trabalhos continuam no âmbito do projeto PolarSYS.²

As soluções propostas no projeto TOPCASED serviram de inspiração para o trabalho apresentado nessa dissertação, cujo foco é incluir verificação formal no processo de desenvolvimento de VHDL sintetizável. A metodologia proposta e a forma como a verificação é adicionada (através do uso de uma cadeia de verificação baseada na linguagem intermediária FIACRE) geram contribuições também para a academia, ao propor regras de tradução entre linguagens (FIACRE-VHDL) e um método de desenvolvimento de código mais confiável.

1.2 OBJETIVO

O objetivo desse trabalho é apresentar uma metodologia de desenvolvimento de código VHDL sintetizável que aprimore as atuais técnicas utilizadas no desenvolvimento para sistemas embarcados de tempo real de uma empresa do setor elétrico. A metodologia deve incorporar métodos formais de verificação de propriedades para garantir que o código VHDL gerado para o FPGA embarcado atenda os requisitos especificados, em especial, requisitos de segurança. Nesse trabalho são apresentados os conceitos e princípios utilizados na metodologia, mas não são construídas ferramentas.

1.2.1 Objetivos Específicos

A metodologia proposta deve ser complementar à utilizada atualmente, adicionando verificação formal por *model checking* à análise por simulação na qual os desenvolvedores estão acostumados. A escolha

²PolarSys - <https://www.polarsys.org/>

do *model checking* como método de verificação se deve ao fato de ser uma técnica automática e também porque gera contraexemplos para propriedades que falham, tornando mais fácil a identificação de erros.

O processo de *model checking* ocorre através de uma cadeia de verificação que utiliza a linguagem intermediária FIACRE. Para ser possível a utilização da cadeia é necessário que o código VHDL sintetizável seja traduzido para um modelo em FIACRE e, portanto, neste trabalho são apresentadas as regras de tradução VHDL-FIACRE.

Como mencionado, uma das principais barreiras para o uso de métodos formais no desenvolvimento de código VHDL é a complexidade das ferramentas e, por isso, a metodologia proposta deve ser acessível ao usuário. Com essa finalidade, é proposto um método de definição de propriedades que se baseia no uso de padrões comuns na verificação de propriedades para circuitos digitais, sendo esses padrões extraídos da biblioteca de observadores OVL (*Open Verification Library*) (OVL, 2014). Para ser possível o uso de padrões OVL na cadeia de verificação FIACRE, eles devem ser traduzidos para o formalismo de representação de propriedades utilizado pela ferramenta de *model checking* da cadeia, no caso, lógica temporal LTL (*Linear Temporal Logics*) (PNUELI, 1977). Neste trabalho são apresentados os princípios de tradução dos padrões OVL para fórmulas LTL.

Ainda com o objetivo de tornar a verificação acessível ao usuário, os resultados da verificação devem ser apresentados de um modo familiar ao desenvolvedor VHDL. Neste trabalho os contraexemplos gerados no processo de *model checking* para as propriedades que falham são traduzidos para *testbenches* VHDL.

1.3 ORGANIZAÇÃO DO TRABALHO

A dissertação está dividida em sete capítulos, que são descritos a seguir. No capítulo 2 é apresentada a atual metodologia de desenvolvimento de VHDL sintetizável utilizada na empresa onde o trabalho foi desenvolvido e suas limitações.

No capítulo 3 é apresentada a metodologia proposta nesse trabalho. É feita uma descrição de como ocorre o procedimento de *model checking*, apresentando as etapas necessárias, assim como as ferramentas e linguagens utilizadas.

No capítulo 4 é descrita em detalhes uma das principais etapas do processo de verificação que é a transformação do código VHDL sintetizável para a um modelo na linguagem intermediária FIACRE. O

modelo FIACRE é a entrada para as ferramentas de verificação utilizadas nesse trabalho. No capítulo ainda será apresentado as regras de tradução que permitem que a transformação ocorra sob os conceitos de engenharia dirigida a modelos (MDE).

No capítulo 5 é apresentado o método utilizado para definição de propriedades, que se baseia na utilização de padrões de propriedade os quais são derivados de uma biblioteca de observadores para VHDL já existente e bastante conhecida dos desenvolvedores, o OVL. Ainda nesse capítulo é apresentado como a biblioteca OVL é utilizada na metodologia para complementar a criação do modelo FIACRE, ao definir restrições quanto ao comportamento das entradas no modelo de verificação.

No capítulo 6 a metodologia é aplicada a alguns exemplos com o objetivo de validação.

Por fim, são apresentadas as conclusões obtidas com o desenvolvimento deste trabalho, bem como suas contribuições e trabalhos futuros.

2 METODOLOGIA DE DESENVOLVIMENTO DE CÓDIGO VHDL

Nesse capítulo é apresentada a metodologia desenvolvimento VHDL atualmente utilizada na empresa onde o trabalho foi desenvolvido. Inicialmente, uma descrição das principais características do VHDL é feita para auxiliar na compreensão da metodologia.

2.1 VHDL

VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) (VHDL, 2008) é uma linguagem de descrição de hardware (HDL) que foi concebida a pedido do Departamento de Defesa dos Estados Unidos para servir como linguagem para descrição de circuitos integrados a serem fabricados por empresas fornecedoras.

Atualmente a linguagem é um padrão IEEE (*Institute of Electric and Electronic Engineers*), podendo ser utilizada para três finalidades principais: documentação, simulação e síntese. A finalidade de documentação, como mencionado anteriormente, foi o objetivo que levou a concepção da linguagem inicialmente, e para isso o VHDL é utilizado para descrição da estrutura e comportamento de circuitos digitais em diferentes níveis de abstração. A linguagem também pode ser usada para simular circuitos digitais, através da construção de códigos chamados *testbenches* onde, além da descrição do circuito a ser testado (*Device Under Test - DUT*), é apresentada uma descrição dos estímulos que serão aplicados nele. O *testbench* é executado em um simulador, o qual apresenta a resposta do circuito aos estímulos através de formas de onda que representam o comportamento dos sinais internos, entradas e saídas. A terceira utilização possível do VHDL é para síntese de circuitos digitais, especialmente para programação de FPGAs. Na síntese, o circuito que se deseja construir é descrito através de um subconjunto de estruturas da linguagem chamado de VHDL sintetizável, e então uma ferramenta interpreta o código e configura o FPGA de tal forma que execute o comportamento descrito.

O VHDL é uma linguagem baseada em fluxo de dados, com a sintaxe baseada nas linguagens ADA e Pascal. É uma linguagem fortemente tipada e o código é dividido em dois blocos principais: **Entity** e **Architecture**. O **Entity** é a interface do circuito descrito com o ambiente, contendo as declarações das portas de entrada e saída. O bloco

Architecture especifica de modo detalhado o comportamento do circuito e como ele é implementado, com suas associações, comparações, conexões entre componentes, operações, etc.

No código 2.1, por exemplo, temos a descrição em VHDL de um circuito que faz um AND lógico. Na estrutura **Entity** temos a declaração das portas de entrada I1 e I2, e da saída O. Na linha 16 do bloco **Architecture** temos a descrição da operação AND lógica entre as entradas.

```

1  -- import std_logic from the IEEE library
   library IEEE;
   use IEEE.std_logic_1164.all;

5  -- this is the entity
   entity andgate is
     port (
       i1 : in std_logic;
       i2 : in std_logic;
10      o  : out std_logic);
   end entity ANDGATE;

   -- this is the architecture
   architecture RTL of andgate is
15  begin
     o <= i1 and i2;
   end architecture RTL;

```

Código 2.1 – Exemplo de código VHDL para uma porta lógica AND

Uma das principais características que diferenciam o VHDL de linguagens de programação tradicionais é a concorrência. Todas as declarações presentes no bloco **Architecture** ocorrem em paralelo. Por ser uma linguagem baseada em fluxo de dados, o que determina a execução de uma declaração no VHDL são os eventos de mudança de valor nos sinais aos quais a declaração é sensível. Essa execução concorrente baseada em eventos é o modo como a linguagem emula o comportamento de um hardware concorrente, onde a mudança de um sinal gera modificações em todos os outros em seu caminho.

A declaração **process** possui uma execução diferente das outras estruturas. O **process** é executado em paralelo em relação às outras declarações utilizadas no código, porém o código apresentado internamente à estrutura é executado de modo sequencial. Essa execução sequencial é útil para descrição de máquinas de estados e algoritmos. Outra utilização importante do **process** é para descrição de circuitos sequenciais visto que é possível especificar a quais sinais a estrutura será sensível, diferentemente das outras estruturas VHDL que são sensíveis a todos os sinais presentes na declaração.

Como mencionado anteriormente, quando se deseja analisar um

circuito descrito por VHDL, é necessária a construção de um *testbench*. O *testbench* é um código VHDL feito somente para simulação e seu bloco **Entity** é vazio. No bloco **Architecture** está descrito o comportamento dos estímulos a serem aplicados nas entradas do circuito a ser simulado, e essas entradas estão associadas a uma instância do circuito. O *testbench* é executado e normalmente uma forma de onda é apresentada com o comportamento dos sinais ao longo da simulação (para a executar a simulação é necessário o *testbench* e o código VHDL a ser simulado). No Código 2.2 temos um exemplo de um *testbench* para simular a porta lógica AND do Código 2.1.

```

1 library IEEE;
  use IEEE.std_logic_1164.all;

  entity andgate_tb is
5 end entity andgate_tb;

  architecture RTL of andgate_tb is
    signal i1, i2, o : std_logic;
    begin
10   DUT: entity work.andgate
        port map (
            i1 => i1,
            i2 => i2,
            o => o);
15
        stimulus: process
        begin
            i1 <= '1';
            wait 10 ns;
20         i2 <= '1';
            wait 10 ns;
            i1 <= '0';
            i2 <= '0';
            wait 10 ns;
25         end process stimulus;
    end architecture RTL;

```

Código 2.2 – Exemplo de testbench para uma porta lógica AND

Nas linhas 16-25 temos uma estrutura do tipo **process** que descreve uma forma de onda para as entradas do dispositivo testado. Primeiro a entrada **i1** é verdadeira, após 10 ns a entrada **i2** também se torna verdadeira e 10 ns depois ambas as entradas se tornam falsas. Nas linhas 10-14 do *testbench* o código do circuito AND é instanciado e os sinais gerados pelo **process** são associados ao componente.

A execução do código VHDL por um interpretador durante a simulação ocorre de acordo com o fluxograma da Figura 1. A simulação começa com uma inicialização das variáveis presentes no código. Em seguida, o interpretador avança o tempo da simulação até um instante em que houve uma modificação em algum sinal presente na descrição dos

estímulos de entrada para o circuito, para em seguida fazer a execução do código e a atualização do valor de todas as variáveis. Quando todas as variáveis foram atualizadas, o simulador avança o tempo novamente até próxima modificação nos sinais dos estímulos e repete o processo de execução do código. Quando não ocorrem mais modificações nos estímulos a simulação é encerrada.

A ordem em que as declarações são apresentadas no código VHDL não é relevante pois elas são executadas em paralelo. No entanto, um interpretador VHDL processa o código na simulação de modo sequencial por ser executado em um computador, e por isso precisa de um mecanismo para garantir determinismo no processamento e emular o comportamento concorrente de *hardware*. Esse mecanismo é chamado de **delta ciclo** (NAVABI, 1997).

O comportamento do delta ciclo está representado no fluxograma da Figura 1. Durante a simulação, na fase em que o interpretador precisa executar o código devido à mudança nos estímulos, a execução da cadeia de delta ciclos é iniciada. Na primeira etapa do delta ciclo, todas as associações sensíveis à entrada modificada são agendadas para serem executadas. Na segunda etapa essas associações são examinadas. Essas associações podem gerar mudanças no valor de alguns sinais, as quais são agendadas para ocorrer na terceira fase. Na terceira etapa, a mudança no valor dos sinais ocorre e um outro delta ciclo é iniciado. Dessa vez, a primeira etapa irá disparar todas as associações que são sensíveis a alguns dos sinais modificados. Vários deltas ciclos podem ocorrer sequencialmente até que na terceira etapa de um deles não ocorrer a mudança de valor de nenhum sinal, quando então o interpretador termina a cadeia de delta ciclos e avança o tempo até a próxima modificação de uma entrada.

É importante lembrar que nem todas as estruturas existentes no VHDL são possíveis de serem sintetizadas em hardware, e por isso o subconjunto em que a síntese é possível é chamado de VHDL sintetizável. Na linha 19 do *testbench* apresentado no Código 2.2 a diretiva **wait** é utilizada. Esse comando é utilizado para representar atrasos de transporte no VHDL e foi utilizado para dizer que um determinado sinal somente deveria assumir o novo valor após 10 ns. Não é possível configurar um FPGA para gerar esse atraso e, portanto, esse comando é considerado não sintetizável. O VHDL sintetizável é uma linguagem síncrona. Uma linguagem é dita síncrona quando satisfaz a hipótese de sincronismo de que o tempo de computação necessário para atualização de suas variáveis é zero.

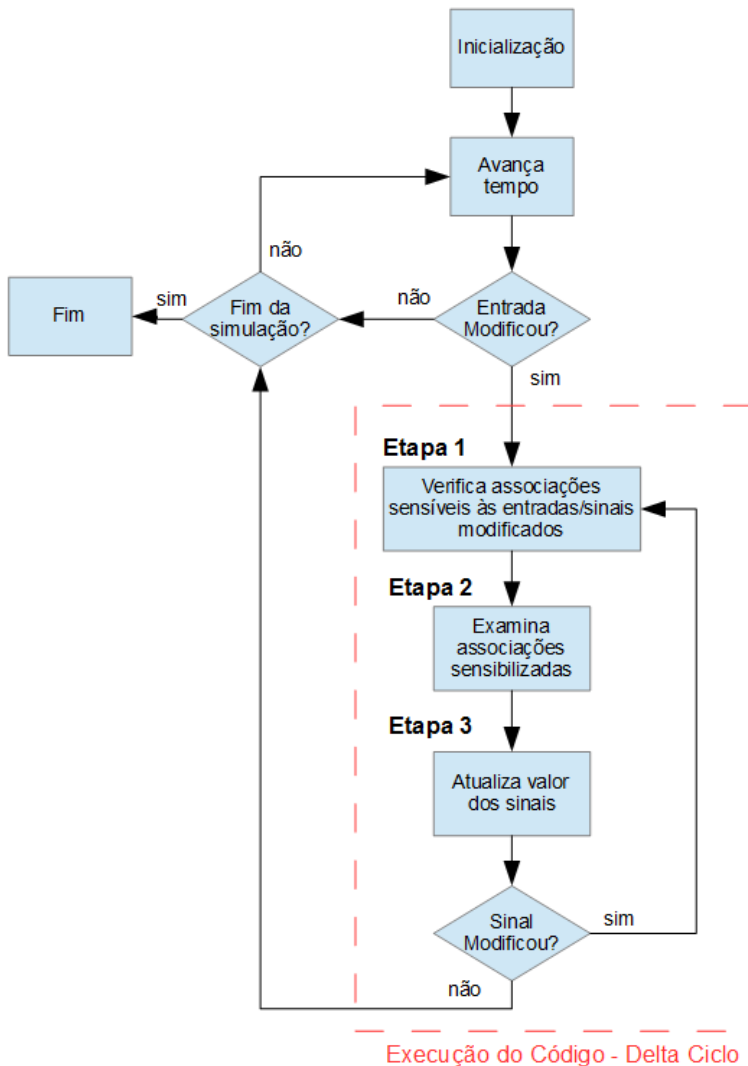


Figura 1 – Execução do código VHDL em simulação

2.1.1 Estilo de Codificação

Um circuito digital pode ser representado em VHDL de várias formas e em vários níveis de abstração. Na empresa onde o trabalho

foi realizado, é considerado como boas práticas que o circuito criado seja sempre síncrono, se possível com um sinal de *clock* apenas, e o nível de abstração adotado nas descrições é o *Register Transfer Level - RTL*. Quando um código VHDL é escrito em nível de abstração RTL, o comportamento do circuito é descrito por meio da representação do fluxo de dados entre registradores e operações sobre esses sinais.

Uma representação RTL em VHDL pode ser feita de várias formas. A empresa costuma seguir as boas práticas mencionadas em Chu (2006). O circuito é dividido em dois blocos principais: o circuito combinacional e o circuito sequencial. O circuito sequencial é a parte que contém os elementos de memória, cujos valores da saída definem qual é o estado do circuito, sendo que esses valores são alterados a cada borda ativa do sinal de *clock*. O circuito combinacional representa a parte que não possui elementos de memória, cujo valor dos sinais depende apenas das entradas conectadas a eles (pode ser uma entrada do circuito ou a saída de um registrador), podendo ser representado por uma tabela verdade. Em Chu (2006) são apresentadas várias maneiras de se realizar a divisão dos circuitos combinacionais e sequenciais, por meio de uma, duas ou três estruturas **process**. A empresa costuma utilizar dois **process**: um **process** para representar o circuito sequencial e um outro para representar o combinacional.

É prática comum de desenvolvedores VHDL representar o comportamento desejado do circuito por meio de máquinas de estado e em seguida codificá-las. Para construir uma máquina de estados em VHDL geralmente se utilizam duas variáveis de tipo enumerado para representar o estado atual e o próximo. A lógica de transição de próximo estado é um circuito combinacional, sendo representado por meio de uma declaração do tipo **case** dentro do **process** da parte combinacional do circuito. A declaração **case** seleciona para execução uma de várias alternativas de código dependendo do valor da variável associada a expressão e, no caso da representação de máquinas de estado, essa variável é a que representa o estado atual.

Para exemplificar o estilo de codificação adotado na empresa, será mostrado um exemplo de código VHDL sintetizável extraído de Chu (2006) (página 334) para um controlador de memória simples com 5 entradas e 3 saídas. A entrada **mem** indica que o usuário deseja acessar a memória, enquanto que a entrada **rw** define se o acesso é uma leitura (verdadeiro, valor 1) ou escrita (falso, valor 0). A entrada **burst** define se o procedimento de leitura será uma sequência de quatro leituras seguidas. As outras entradas são o *clock* do sistema (**clk**) e o sinal de *reset*. A saída **oe** realiza uma leitura na memória enquanto que

a saída **we** realiza uma escrita. A máquina de estados que representa o comportamento do circuito está na Figura 2. Na ocorrência de um sinal de *reset*, a máquina de estados retorna ao estado **idle** independente do valor do sinal de **clk** (*reset* assíncrono), enquanto que as outras transições entre estados somente acontecem na borda de subida de **clk** (e as outras condições necessárias também satisfeitas).

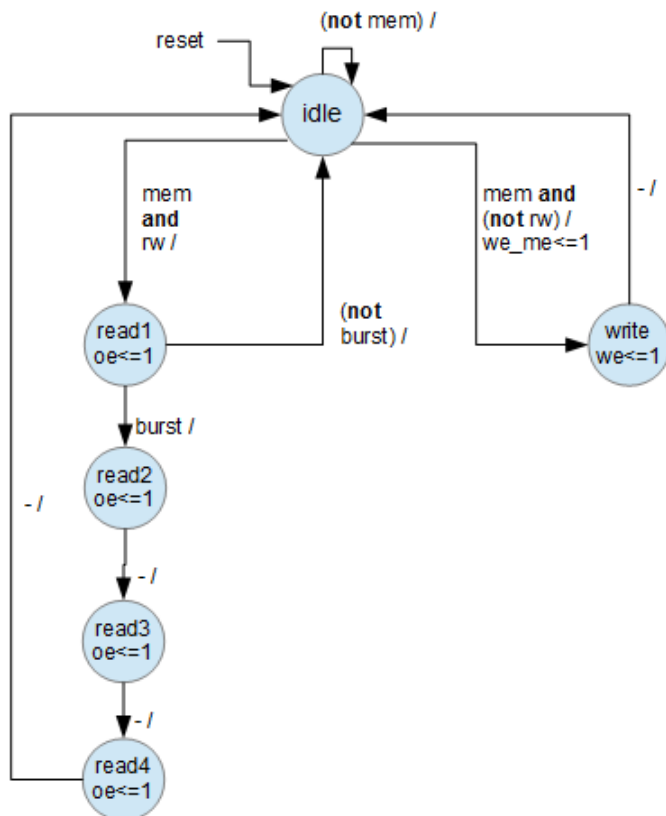


Figura 2 – Máquina de estados do controlador de memória

O Código 2.3 apresenta o VHDL do bloco **Architecture** do controlador de memória. A linha 2 contém a declaração do tipo enumerado que representa os estados, enquanto na linha seguinte temos a declaração dos sinais que representam o estado atual (**state_reg**) e o próximo estado (**state_next**). O código possui somente duas declarações concorrentes do tipo **process**. O **process** apresentado entre

as linhas 7-14, representa a parte sequencial do circuito responsável por atualizar o valor de “**state_next**” com o valor de “**state_reg**” na ocorrência da borda de subida do *clock*. O outro processo, que é descrito entre as linhas 17-55, é a parte combinacional do circuito responsável por atualizar as saídas e definir o próximo estado (atualizar o “**state_next**”). O código VHDL completo do controlador de memória pode ser encontrado no Apêndice A.

```

1 architecture two_seg_arch of mem_ctrl is
    type mc_state_type is (idle, read1, read2, read3,
                           read4, write);
    signal state_reg, state_next : mc_state_type;
5 begin
    -- state register
    process(clk, reset)
    begin
        if (reset = '1') then
10         state_reg <= idle;
            elsif rising_edge(clk) then
                state_reg <= state_next;
            end if;
        end process;

15 -- next-state logic and output logic
    process(state_reg, mem, rw, burst)
    begin
        oe <= '0'; -- default value
20         we <= '0';
            we_me <= '0';

        case state_reg is
25         when idle =>
            if mem = '1' then
                if rw = '1' then
                    state_next <= read1;
                else
30                 state_next <= write;
                    we_me <= '1';
                end if;
            else
                state_next <= idle;
            end if;
35         when write =>
            state_next <= idle;
            we <= '1';
        when read1 =>
            if (burst = '1') then
40             state_next <= read2;
            else
                state_next <= idle;
            end if;
            oe <= '1';
45         when read2 =>
            state_next <= read3;
            oe <= '1';
        when read3 =>
            state_next <= read4;
            oe <= '1';
50         when read4 =>

```

```

        state_next <= idle;
        oe         <= '1';
    end case;
55 end process;
end two_seg_arch;

```

Código 2.3 – Código VHDL do **Architecture** do controlador de memória

2.1.2 Análise de Propriedades

Análise de propriedades em VHDL pode ser feita de dois modos: por meio de observadores ou utilizando linguagens de descrição de propriedades.

Na validação de propriedades por observadores, o desenvolvedor deve criar no *testbench* um circuito gerador de estímulos para o DUT e um circuito que observa as entradas e saídas dele (chamados também de monitores), sinalizando quando da ocorrência de uma situação indesejada. Geralmente essa sinalização é feita através da diretiva *assert* da linguagem, que gera uma mensagem ou encerra a simulação quando a expressão booleana que está observando é falsa.

As duas principais linguagens de descrição de propriedades utilizadas em HDLs são: PSL (Property Specification Language) (PSL, 2004) e SVA (System Verilog Assertion) (SVA, 2009). A linguagem VHDL adotou o PSL como linguagem padrão para descrição de propriedades em sua versão de 2008. O PSL é uma lógica temporal que estende a lógica LTL (PNUELI, 1977) com a adição de alguns operadores além de incluir uma extensão que utiliza operadores de CTL (CLARKE; EMERSON, 1981). A linguagem PSL foi desenvolvida inicialmente pela Accellera (ACCELLERA, 2015) (posteriormente se tornou padrão IEEE) como uma lógica temporal melhor adaptada para descrição de propriedades para projetos de *hardware*, em especial circuitos digitais. Algumas das funcionalidades do PSL que facilitam a descrição de propriedades para circuitos são: existência de um operador de *clock*, operador de *reset* e uso de expressões regulares.

Com o uso do PSL, o desenvolvedor pode descrever propriedades em forma de lógica temporal que podem estar relacionadas tanto com as entradas e saídas do circuito quanto a sinais internos. A possibilidade de se verificar propriedades quanto a sinais internos é uma grande vantagem em relação aos monitores, os quais ficam restritos a interface do circuito com o ambiente. As expressões PSL podem estar presentes em um arquivo separado ou apresentados como comentários

internamente ao código VHDL (do DUT ou *testbench*). Durante a simulação, se a ferramenta de simulação tem suporte a PSL, toda vez que uma propriedade descrita em PSL falha, uma mensagem é gerada indicando o momento exato. Algumas ferramentas são capazes de fazer verificação formal das propriedades PSL.

2.2 METODOLOGIA DE DESENVOLVIMENTO ATUAL

A empresa utiliza um processo típico de desenvolvimento de VHDL sintetizável, apresentado na Figura 3.

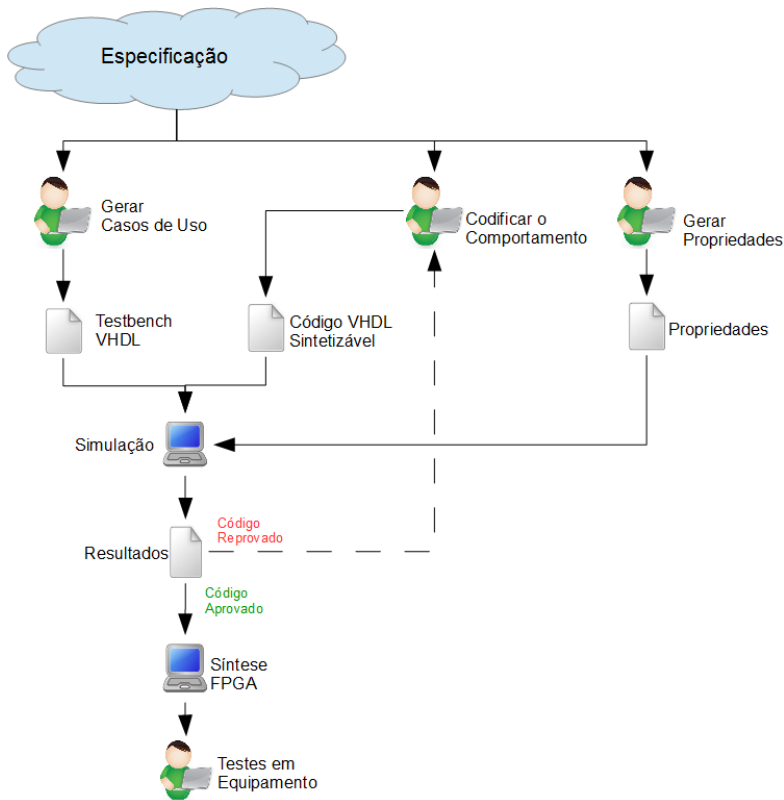


Figura 3 – Metodologia atual de desenvolvimento VHDL

Inicialmente o desenvolvedor VHDL recebe uma especificação informal descrevendo o comportamento desejado para o circuito. Normal-

mente essa especificação tem origem a partir de necessidades avaliadas pelo setor de Marketing (pode ser, por exemplo, o conserto de algum problema ou funcionalidades novas) e é comum que seja baseada em alguma norma utilizada no setor de atuação da empresa. O documento de especificação descreve o comportamento desejado através de textos e diagramas que são comuns para representar sistemas na área de aplicação do equipamento desenvolvido. No caso da empresa, o setor de aplicação é automação de subestações de energia elétrica. Portanto, os equipamentos estão relacionados à proteção, sincronismo, oscilografia e redes em subestações. O desenvolvedor VHDL está acostumado com modelos para especificação de circuitos digitais e, por isso, é comum que tenha dificuldade em compreender a especificação, necessitando muitas vezes do auxílio de engenheiros de aplicação (engenheiros responsáveis pela instalação em campo dos equipamentos), ou dos funcionários do setor de *Marketing* da empresa.

A partir da especificação, o desenvolvedor VHDL gera casos de uso representando a funcionalidade do sistema, máquinas de estado que representam o comportamento do circuito a ser implementado e também as propriedades que o circuito deve respeitar. As propriedades usualmente definem limites de valor para saídas do circuito, condições em que saídas devem ser ativadas/desativadas, exclusão mútua na ativação de saídas, ausência de *deadlocks*, controle de acesso à memórias, eliminação de *loops* ou contadores infinitos etc.

O código VHDL sintetizável para FPGA é criado com base nas máquinas de estados. *Testbenches* VHDL são criados com formas de onda que representam os casos de uso. Um problema que ocorre nessa etapa é que nem sempre o desenvolvedor gera documentos registrando os casos de uso implementados no *testbench* e as máquinas de estado que serviram de modelo para o código. Em alguns casos, são criados observadores nos *testbenches* para representar as propriedades e auxiliar na validação das mesmas. Linguagens de descrição de propriedades não são utilizadas.

O código VHDL e o *testbench* são usados para simular o circuito em operação. A ferramenta de simulação gera uma forma de onda com o comportamento de todas as variáveis (entradas, saídas e sinais internos) do circuito (todas as ferramentas de simulação VHDL disponíveis atualmente geram essas formas de onda), e a partir disso o desenvolvedor verifica se a evolução dos valores das variáveis ocorreu como desejado em relação aos estímulos aplicados. Caso o comportamento não seja como desejado, o desenvolvedor procura a fonte de erro no código usando a forma de onda como auxílio, ao observar o comportamento

das variáveis à medida que os estímulos vão sendo aplicados ao circuito. Após a correção, a simulação é executada novamente para verificar se o comportamento desejado foi atingido e, em hipótese verdadeira, o FPGA é configurado para representar o circuito.

Na Figura 3 foram omitidas as etapas do processo de configuração do FPGA, pois esse trabalho tem o foco em verificação da lógica do circuito. Porém, durante esse processo alguns problemas podem aparecer. O principal deles é o não atendimento dos requisitos temporais dos elementos de memória devido a atrasos de propagação dos sinais no circuito configurado no FPGA. Esse problema pode levar partes do circuito a situações de metaestabilidade (quando o elemento de memória, usualmente um *flip-flop*, entra em um estado em que sua saída se torna imprevisível, oscilando entre 0 e 1) e normalmente para evitar esse tipo de situação é necessário inserir estágios de registradores nos caminhos de maior atraso, o que exige alterações no código VHDL implementado. Outro problema que pode acontecer é a falta de recursos no FPGA para a implementação do circuito, o que pode exigir também alterações no código. Todos esses problemas são reportados pela ferramenta que compila o código VHDL e faz a configuração do FPGA.

Após a configuração do FPGA, o equipamento é repassado para uma equipe de testes (diferente da equipe de desenvolvimento) responsáveis por fazer as verificações finais. Caso algum erro encontrado esteja diretamente relacionado com as tarefas desempenhadas pelo código VHDL criado, o desenvolvedor passa a fazer novas correções, tentando inicialmente reproduzir em simulação o erro encontrado. Após a origem do erro ser descoberta, correções são feitas e o código corrigido é simulado novamente para os casos de uso para poder ser aprovado.

Essa metodologia baseada somente em simulação com *testbenches* apresenta muitas limitações. A principal deles é o fato de não ser uma análise exaustiva e por isso não ser capaz de garantir o atendimento de propriedades. Como uma limitação adicional podemos citar também a dificuldade de visualizar na forma de onda a ocorrência de um erro, ainda mais para códigos relativamente mais complexos e que contenham uma grande quantidade de sinais internos. Um modo que os desenvolvedores utilizam para contornar esse problema é a utilização de observadores que sinalizam por meio de mensagens quando da ocorrência do comportamento indesejado.

O uso de observadores facilita encontrar erros nas formas de onda ao aumentar a observabilidade do circuito. No entanto, ainda é dependente do desenvolvedor escrever um *testbench* que estimule o

circuito ao ponto de chegar a situação em que a falha ocorra, o que não é uma tarefa fácil. O uso de técnicas como *code coverage* e *functional coverage* tentam mitigar esse problema ao gerar estímulos randômicos ou em uma determinada faixa de valores (obedecendo restrições), porém ainda não efetuam uma análise exaustiva e não tem como garantir que uma propriedade não falhará. Na empresa não é utilizado *code coverage* nem *functional coverage* nas simulações.

Como último problema podemos citar a quantidade de tempo necessária para se efetuar uma análise por simulação para uma quantidade de situações de uso bastante abrangente, que em geral acaba sendo muito grande e gerando custos mais altos para as empresas.

2.2.1 Desenvolvimento com uso de observadores

O uso de observadores na simulação ajuda o desenvolvedor a visualizar com mais facilidade a ocorrência de um erro. Um modo de facilitar a análise de atendimento de propriedades durante a simulação dos casos de uso seria convertê-las em observadores e incluir no código do *testbench*, o que faria com que a atual metodologia fosse modificada para aquela apresentada na Figura 4.

Após a definição das propriedades a partir da especificação, elas são convertidas para observadores. Esses observadores são incluídos no *testbench* e a simulação executada. O observador sinaliza durante a simulação quando a propriedade não é satisfeita, podendo até encerrar a simulação se for configurado para tal. O uso de observadores facilita bastante pois o usuário não precisa visualizar toda a forma de onda para saber se houve alguma falha de propriedade, no entanto, os erros somente serão encontrados se o *testbench* excitar o circuito a tal ponto que ele chegue ao estado indesejado. Um outro problema quanto ao uso de observadores é que a definição deles para representação de uma propriedade é uma tarefa muitas vezes custosa e sujeita à erros. Um erro na criação do observador inviabilizaria a validação da propriedade analisada.

A biblioteca OVL (*Open Verification Library*), concebida pelo comitê de pesquisa *Accellera Initiative*¹, possui uma grande quantidade de observadores (chamados de *assertions*) escritos em VHDL ou Verilog, que tem como foco facilitar o trabalho do desenvolvedor VHDL/Verilog ao oferecer uma biblioteca de propriedades mais frequentemente observadas nessas linguagens (evitando erros na criação de obser-

¹<http://accellera.org/>

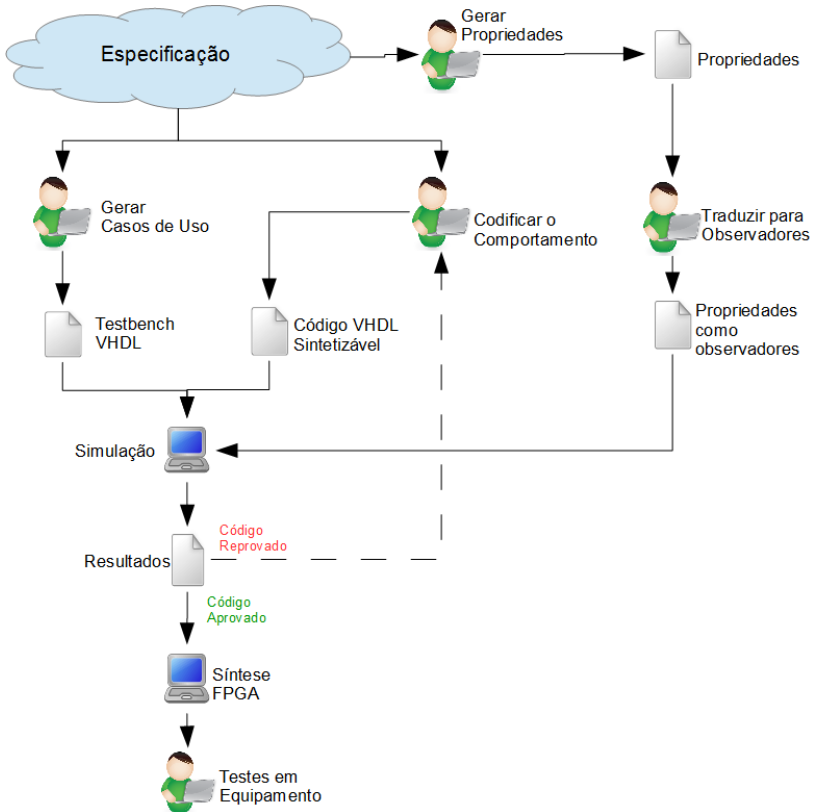


Figura 4 – Metodologia atual de desenvolvimento VHDL com o uso de observadores

vadores): exclusão mútua, janelas de eventos, implicação, sequência de eventos, etc. Um observador OVL é utilizado ao instanciá-lo no código do *testbench* e fazer as conexões dos sinais relacionados à propriedade a ser verificada. O observador é geralmente utilizado em simulação, mas também pode ser usado em verificação formal por uma ferramenta que dê suporte.

O OVL tem uma importância relevante neste trabalho por servir como base na definição de propriedades para verificação conforme será explicado posteriormente.

2.3 CONCLUSÃO

Análise somente por simulação pode até ser suficiente para observar se o circuito é capaz de atender os casos de uso, porém não é capaz de garantir o atendimento de propriedades, pois naturalmente exige verificação exaustiva. Na continuação desse documento será apresentada a proposta de metodologia de desenvolvimento que inclui a verificação formal para checagem de propriedades, no intuito de contornar as limitações descritas da abordagem atual.

3 METODOLOGIA PROPOSTA

A metodologia de desenvolvimento atual, por utilizar apenas simulação para validação do código VHDL, não é capaz de garantir formalmente o atendimento de propriedades já que esta prova depende de uma verificação exaustiva do espaço de estados do circuito, o que não é obtido através de simulação, por mais abrangente que seja. A metodologia proposta nesse capítulo, tem como finalidade resolver essa limitação ao adicionar verificação por *model checking* das propriedades ao processo de desenvolvimento. O *model checking* é uma técnica de verificação que faz uma exploração de todo espaço de estados do sistema para provar matematicamente que uma determinada propriedade é atendida em todas as situações.

A ideia de usar *model checking* para validação de VHDL não é nova, muitos trabalhos na literatura apresentaram propostas para isso e também já existem ferramentas proprietárias que fornecem soluções. Apesar de essas soluções existirem, elas acabaram não sendo amplamente usadas na indústria principalmente por necessitarem que o usuário tenha que ter conhecimento aprofundado de métodos formais (WOODCOCK et al., 2009), tendo que muitas vezes aprender a definir propriedades com o uso de lógica temporal ou descrever o modelo do circuito a ser verificado usando algum formalismo matemático específico de uma ferramenta, algo bem complexo e muito diferente para o desenvolvedor VHDL. Além da complexidade de ter que lidar com métodos formais, muitas vezes as ferramentas disponíveis são proprietárias e de alto custo, fazendo com que o uso por empresas de pequeno porte se torne inviável.

A metodologia proposta nesse capítulo tem como objetivo específico manter a interface com as ferramentas de verificação em nível de usuário, tornando o uso dos métodos formais transparente ao usuário e facilitando seu uso. Outro ponto importante da proposta é que as ferramentas utilizadas no processo de verificação são *open source*.

A seguir será apresentado com mais detalhes o método de *model checking*, descrevendo como funciona e suas vantagens. Em seguida será apresentada a proposta de metodologia e os trabalhos relacionados.

3.1 MODEL CHECKING E LÓGICAS TEMPORAIS

Model Checking (BAIER; KATOEN, 2008) é uma técnica de verificação que explora todos os possíveis cenários de execução de um sistema, de um modo sistemático. Dessa maneira, é possível mostrar que um sistema verdadeiramente satisfaz certas propriedades, potencialmente revelando erros que passam despercebidos em emulação, testes e simulação, devido ao fato de essas técnicas não serem exaustivas.

O esquema da verificação por abordagem de *model checking* está representado na Figura 5. O *model checking* é uma técnica de verificação automática e a ferramenta de verificação necessita que o sistema seja descrito em um modelo formal que ela possa interpretar. Em geral, o modelo formal do sistema é construído manualmente usando o formalismo exigido pelas ferramentas, porém algumas delas podem disponibilizar geração automática do modelo formal a partir de modelos descritos em linguagem de usuário, podendo ser linguagens de programação (C, Java etc.), descrição de *hardware* ou linguagens de modelagem em alto nível (UML, AADL etc.). As propriedades a serem verificadas também precisam ser descritas em um modelo formal, preciso e sem ambiguidades. O *model checker* (ferramenta de verificação) examina todos os estados relevantes do sistema para checar se uma propriedade desejada é satisfeita. Se um estado é encontrado que viola a propriedade em consideração, a ferramenta fornece um contra-exemplo o qual descreve um caminho de execução que leva do estado inicial ao estado indesejado. Com a ajuda de um simulador, o usuário pode visualizar o cenário de erro, obtendo desse modo uma informação importante para encontrar a origem do erro e consertá-la.

Ao aplicar o *model checking* em um projeto, as seguintes fases podem ser identificadas:

- **Modelagem:** Modelar o sistema em consideração usando a linguagem de descrição da ferramenta verificadora. Formalizar a propriedade a ser verificada usando a linguagem de descrição de propriedades que a ferramenta é capaz de interpretar.
- **Verificação:** Executar a verificação para checar a validade das propriedades sobre o modelo.
- **Análise:** Caso a propriedade seja satisfeita, checar a próxima. Se a propriedade falhar, analisar o contra-exemplo para encontrar a origem do erro para, em seguida, fazer modificações necessárias no projeto e executar novamente a verificação.

Um dos principais problemas do *model checking* é a situação de explosão de estados. Esse problema acontece quando o espaço de estados do sistema possui um tamanho maior do que é possível representar com a memória disponível nos sistemas computacionais atuais. Várias alternativas podem ser utilizadas para contornar o problema, que podem ser desde alternativas de representação de estados que explorem a regularidade entre eles, até a abstração dos modelos, no intuito de reduzi-los somente à parte que é relevante a propriedade verificada (CLARKE et al., 2001).

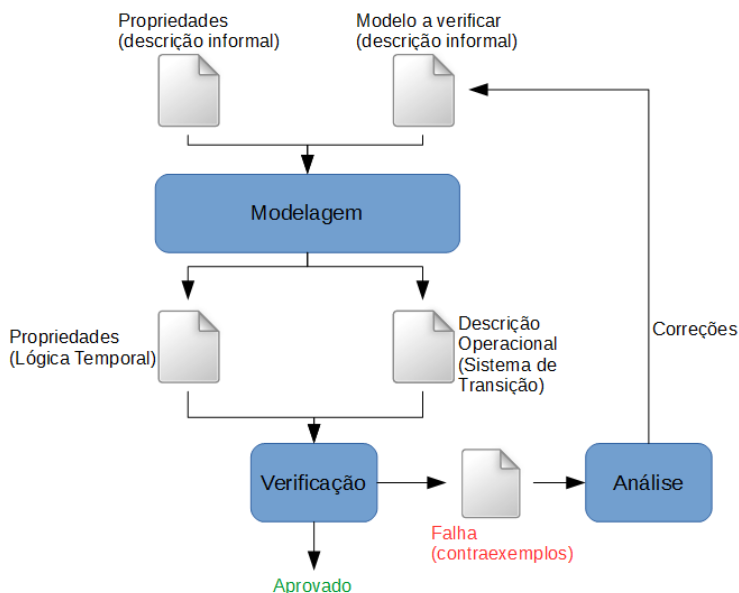


Figura 5 – Esquemático da abordagem por *model checking*

O modelo formal mais utilizado em *model checking* para descrição do sistema a ser verificado é o **sistema de transição**. Um sistema de transição é, basicamente, um grafo direcionado onde nós representam estados, e arcos representam transições, isto é, mudança de estados. Um estado descreve alguma informação sobre o sistema em algum momento de seu comportamento. Em um circuito digital síncrono, um estado tipicamente representa o valor atual dos registradores e das entradas. As transições especificam como o sistema pode evoluir de um estado para outro. Para um circuito digital, a transição pode representar as mudanças de valores nos registradores e saídas para

um novo conjunto de entradas.

O *model checking* é bastante aplicado para verificação de sistemas reativos, que se caracterizam por uma interação contínua com o ambiente no qual estão inseridos. Os sistemas desta natureza tipicamente recebem estímulos do ambiente e quase que instantaneamente reagem às entradas recebidas. Tradicionalmente, eles são distribuídos, concorrentes e não possuem um término de execução, isto é, eles estão constantemente prontos para interagir com o usuário ou outros sistemas. Para sistemas reativos, a validação de um correto funcionamento depende também da análise das execuções, isto é, a ordem em que os eventos ocorrem, e não somente da avaliação dos valores de saída resultantes para um determinado conjunto de valores de entrada.

Lógica temporal é um formalismo adequado para representar propriedades para sistemas reativos e, portanto, especificar seu funcionamento correto para fins de verificação. Ela estende a lógica proposicional ou de predicado, por modalidades que permitem se referir ao comportamento infinito desses sistemas. As lógicas temporais fornecem uma notação bem intuitiva porém matematicamente precisa para expressar propriedades sobre relações entre estados em uma execução.

A natureza subjacente de tempo em lógica temporal pode ser tanto linear quanto ramificada. Na perspectiva linear, a cada momento no tempo existe somente um momento sucessor, enquanto que na visão ramificada cada momento possui uma ramificação onde o tempo pode se dividir em percursos alternativos. A lógica temporal CTL (*Computation Tree Logic*) (CLARKE; EMERSON, 1981) adota a abordagem ramificada, enquanto que a lógica LTL (*Linear Temporal Logics*) (PNU-ELI, 1977) adota a perspectiva linear. A ferramenta de *model checking* utilizada nesse trabalho aceita propriedades descritas em lógica LTL e, portanto, essa linguagem terá destaque.

Apesar do termo “temporal” em LTL sugerir uma relação com o comportamento em tempo real de sistemas reativos, isso é verdade apenas em sentido abstrato. Lógica temporal permite a especificação da ordem relativa entre eventos. Ela não suporta nenhum modo de se referir ao tempo preciso de acontecimento dos eventos. Em termos de sistemas de transição, nem o tempo para a realização de uma transição ou o tempo de permanência em um estado podem ser especificados usando as modalidades elementares da lógica temporal LTL. Ao invés disso, essas modalidades permitem especificar a ordem na qual determinadas proposições atômicas são verdadeiras durante uma execução, ou afirmar que certas proposições são verdadeiras com frequência infinita em uma (ou todas) execuções do sistema.

Uma propriedade descrita com fórmula LTL é formada pela combinação de proposições atômicas, constantes (verdadeiro ou falso), conectores booleanos e operadores temporais. As proposições atômicas fazem afirmações sobre os estados, em que estas proposições são relações elementares, as quais, em um dado estado, possuem um valor verdadeiro bem definido. Os conectores booleanos são conjunção, disjunção, negação, implicação e dupla implicação. Os operadores temporais permitem construir expressões relacionadas ao sequenciamento dos estados ao longo de uma execução e não apenas aos estados individualmente. Os operadores temporais presentes no LTL são (considerando p e q como proposições atômicas):

- **Next (X):** $\bigcirc p$ (define que p é válido no próximo estado)
- **Until (U):** $p \text{ U } q$ (define que p é válido até q ser)
- **Globally (G):** $\Box p$ (define que p é válido sempre)
- **Future (F):** $\Diamond p$ (define que q é eventualmente válido no futuro ou no estado atual)

A sintaxe do LTL é definida como:

- Se p é uma proposição atômica, então p é uma fórmula LTL.
- Se f , f_1 e f_2 são fórmulas LTL, então as seguintes expressões são fórmulas LTL: $\neg f$, $f_1 \wedge f_2$, $\bigcirc f$, $[f_1 \text{ U } f_2]$.

Os operadores **Next** e **Until** são os operadores elementares, todos os outros operadores podem ser obtidos a partir de expressões contendo somente eles.

3.2 METODOLOGIA PROPOSTA

A metodologia proposta tem como objetivo adicionar verificação de propriedades ao atual processo, complementando o que já é utilizado. A parte que é mantida do atual método de desenvolvimento pode ser visualizada na parte esquerda da Figura 6. Na proposta o usuário deve, a partir da especificação recebida com a descrição do comportamento do circuito a ser implementado, continuar a gerar casos de uso representando a funcionalidade do circuito, máquinas de estado representando o comportamento do circuito a ser implementado, e propriedades que o circuito deve satisfazer. Como já descrito no Capítulo 2, os casos de uso

dão origem a *testbenches* que são usados em simulação para verificar se o circuito cumpre a funcionalidade esperada, e as máquinas de estado originam o código VHDL sintetizável que será usado para configurar o FPGA.

A diferença da proposta em relação ao método atual é que, ao invés de o usuário validar propriedades do circuito por simulação, ele faz a verificação de propriedades paralelamente a simulação dos casos de uso. Um circuito somente é aprovado quando a simulação dos casos de uso não apontar erros e todas as propriedades forem aprovadas na verificação.

As duas principais abordagens de verificação são o *model checking* e prova de teorema (CLARKE; WING, 1996). Para a metodologia proposta foi escolhida a abordagem por *model checking*, porque ela retorna contraexemplos contendo condições que levam ao erro nos casos de propriedades que falham, facilitando muito o trabalho do desenvolvedor para encontrar os erros no código. Além disso, o *model checking* é um método de verificação completamente automático e mais rápido que os demais. A abordagem por prova de teorema normalmente é lenta e parcialmente automática, precisando de interferência do usuário em algumas etapas e por isso sendo mais suscetível a erros e exigindo um conhecimento muito mais aprofundado do método.

A parte da direita da Figura 6, destacado em tracejado, representa o que a metodologia propõe de novo. Como o método de verificação escolhido é o *model checking*, é necessário que o código VHDL seja traduzido para um modelo formal e as propriedades sejam descritas por fórmulas de lógica temporal. Os formalismos tanto do modelo quanto da lógica dependem da ferramenta de *model checking*.

Como um dos objetivos da proposta é tornar o método de verificação transparente ao desenvolvedor, mantendo as interfaces com o processo em nível de usuário, a metodologia propõe que ao invés do desenvolvedor VHDL descrever as propriedades diretamente em lógica temporal, elas devem ser representadas por meio de padrões de propriedade.

A proposta de utilizar padrões de propriedade se inspira no trabalho de Dwyer (DWYER; AVRUNIN; CORBETT, 1999). Em seu trabalho, Dwyer mostrou que a maioria das especificações de propriedades para verificação de máquinas de estados finitos são pequenas variações de um conjunto de padrões. Ele conclui que, em geral, não é necessário o desenvolvedor conhecer as particularidades de uma determinada linguagem, mas somente conhecer como esses padrões são representados. Com um objetivo similar ao de Dwyer, o comitê de pesquisa *Accellera*

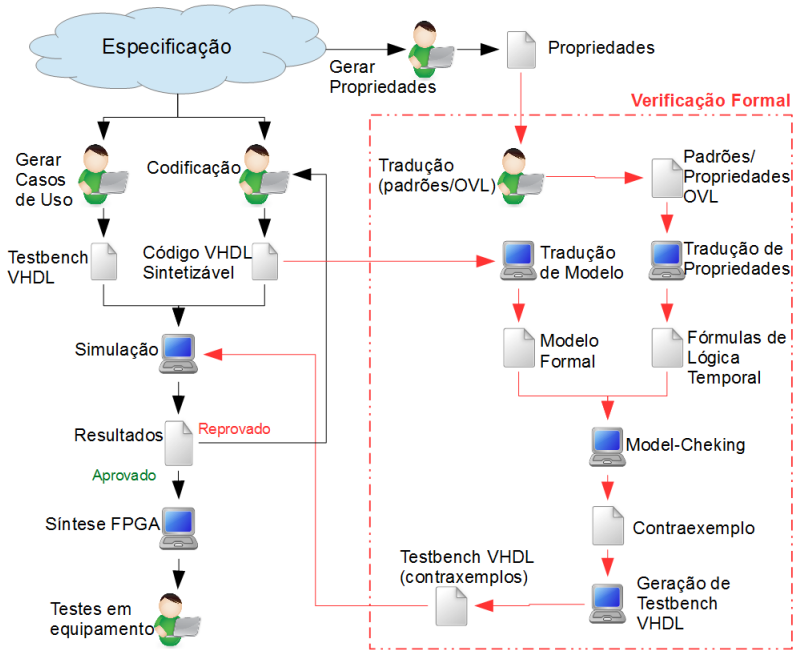


Figura 6 – Metodologia de Desenvolvimento VHDL proposta (visão geral)

Initiative criou o OVL (*Open Verification Library*) (OVL, 2014).

Inicialmente a ideia é traduzir todos os observadores da biblioteca OVL para a lógica temporal utilizada pela ferramenta de *model checking* na metodologia. Quando o usuário deseja definir uma propriedade, ele apenas escolhe um dos padrões e define quais sinais do circuito serão utilizados na propriedade, e então a propriedade é traduzida automaticamente para lógica temporal (uma ferramenta de tradução automática deve ser criada). Novos padrões de propriedade, não disponíveis na OVL, podem ser disponibilizados quando houver necessidade.

O modelo formal correspondente ao código VHDL e as propriedades descritas em lógica temporal são aplicadas ao processo de *model checking*, o qual retorna os resultados da verificação. Quando as propriedades falham, o processo retorna um contraexemplo contendo a sequência de entradas que levam a situação indesejada. Os contraexemplos gerados representam uma sequência de transições de estados que

levam à condição de falha a partir do estado inicial. No entanto, esse tipo de representação é de difícil entendimento para um desenvolvedor VHDL devido à complexidade de se associar as alterações de valores das variáveis do código VHDL com as transições de estado do modelo formal. Para manter a saída do processo de verificação em nível de usuário, é proposto que o contraexemplo seja traduzido para um *testbench* VHDL, pois o usuário pode executar o *testbench* em qualquer simulador que preferir, e assim visualizar a forma de onda que leva ao erro para conseguir detectar no código o problema.

Após a correção dos erros encontrados, a nova versão do código VHDL é submetida novamente à verificação. Caso nenhum erro seja detectado (nenhum contraexemplo gerado), então as propriedades são todas satisfeitas. Se o código for também aprovado na simulação de casos de uso, então o código está pronto para ser sintetizado em FPGA e, em seguida, submetido a testes em equipamento.

A tradução do código VHDL para um modelo formal é complicada devido ao fato da linguagem VHDL estar muito distante dos formalismos matemáticos adotados pelas ferramentas de *model checking*, com grandes diferenças de sintaxe e semântica que dificultam muito a tradução. Para resolver esse problema, na metodologia proposta o processo de *model checking* ocorre através de uma cadeia de verificação. No contexto desse trabalho, é chamada “cadeia de verificação” uma sequência de transformações e ferramentas necessárias para realizar o processo de verificação formal de um sistema em linguagem de usuário. O primeiro passo da cadeia é a transformação do modelo em linguagem de usuário para um modelo correspondente em uma linguagem intermediária, o qual serve de entrada para ferramentas de verificação. Nesse trabalho a linguagem de usuário é o VHDL enquanto a linguagem intermediária é o FIACRE. O uso de uma linguagem intermediária é vantajosa porque serve como um ponto de convergência, permitindo que se possa utilizar qualquer ferramenta de verificação construída para aquela linguagem.

A linguagem FIACRE foi desenvolvida como parte de projetos relacionados à engenharia dirigida a modelos. Ela foi construída como sendo uma linguagem intermediária formal entre linguagens de modelagem de alto nível, e ferramentas de verificação que trabalham com linguagens baseadas em formalismos matemáticos (autômatos, redes de Petri, sistemas de transição temporizados). FIACRE é um modelo intermediário formal para representar tanto aspectos comportamentais como temporais de sistemas, com a finalidade de verificação e simulação. No âmbito do projeto TOPCASED, a linguagem FIACRE

já foi utilizada na construção de cadeias de verificação para diversas linguagens de alto nível, além disso, a linguagem já tem disponível modelos de tradução para diversos formalismos matemáticos, permitindo que o FIACRE seja passo intermediário para diversas ferramentas de verificação.

A Figura 7 mostra a metodologia como ela é de fato proposta nesse trabalho: com o uso da cadeia de verificação FIACRE. A ferramenta de *model checking* da cadeia de verificação necessita que as propriedades sejam descritas em lógica temporal LTL, portanto, as propriedades descritas em OVL precisam ser traduzidas para essa lógica. Para ser possível a utilização da cadeia de modo transparente, o código VHDL sintetizável criado é automaticamente traduzido para um modelo FIACRE correspondente.

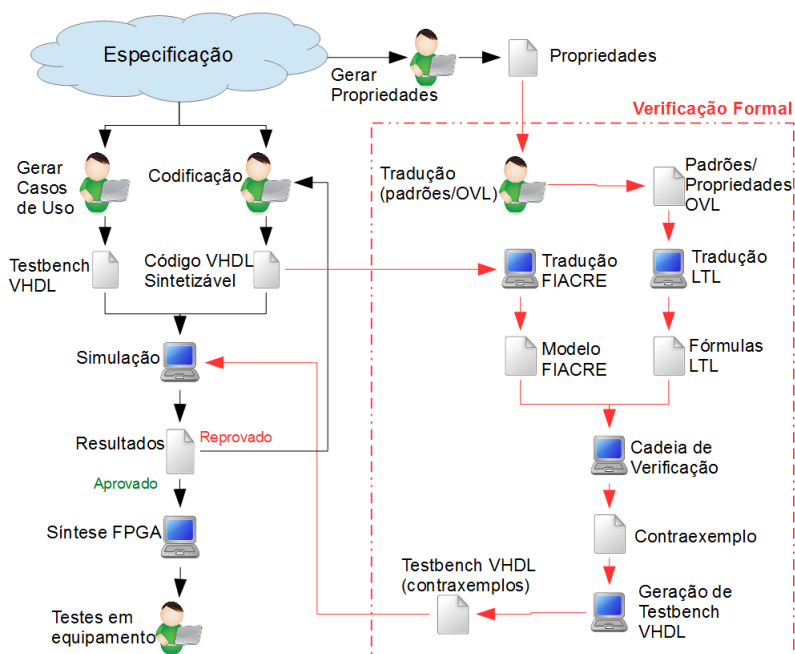


Figura 7 – Metodologia de Desenvolvimento VHDL proposta (FIACRE)

3.3 AMBIENTE DE VERIFICAÇÃO

O projeto TOPCASED (*Toolkit in OPen-source for Critical Application and SystEms Development*) (TOPCASED, 2014), teve o intuito de criar um ambiente de desenvolvimento com um conjunto de ferramentas de engenharia de sistemas para o desenvolvimento de sistemas críticos de tempo real. No projeto TOPCASED foram propostas cadeias de verificação que utilizassem a linguagem intermediária FIA-CRE. A Figura 8 apresenta a estrutura de uma cadeia de verificação genérica.

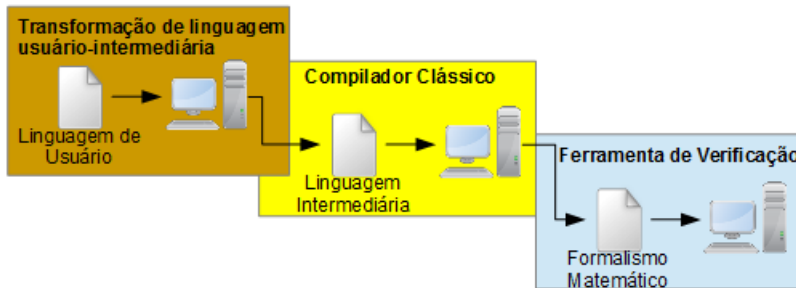


Figura 8 – Cadeia de verificação genérica (SOUZA, 2010)

No exemplo da Figura 8, a cadeia de verificação possui duas transformações de modelo e uma ferramenta de verificação. A primeira transformação é a tradução do modelo do sistema descrito em linguagem de usuário para uma linguagem intermediária. A segunda é a tradução do modelo em linguagem intermediária para o formalismo matemático selecionado, dependendo da ferramenta de verificação escolhida.

Como mencionado na seção anterior, o processo de verificação formal realizado na metodologia proposta acontece através de uma cadeia de verificação, que segue a estrutura clássica apresentada na Figura 8. A cadeia é inspirada no trabalho de Farines et al. (2011), onde foi proposta uma cadeia de verificação com base na linguagem intermediária FIACRE para realizar *model checking* em programas *ladder* para CLPs, e está ilustrada na Figura 9. As etapas destacadas em vermelho são objetivos desse trabalho.

O processo de verificação começa com a obtenção do modelo FIACRE (linguagem intermediária) do VHDL sintetizável (linguagem de usuário), que corresponde a primeira transformação de uma cadeia

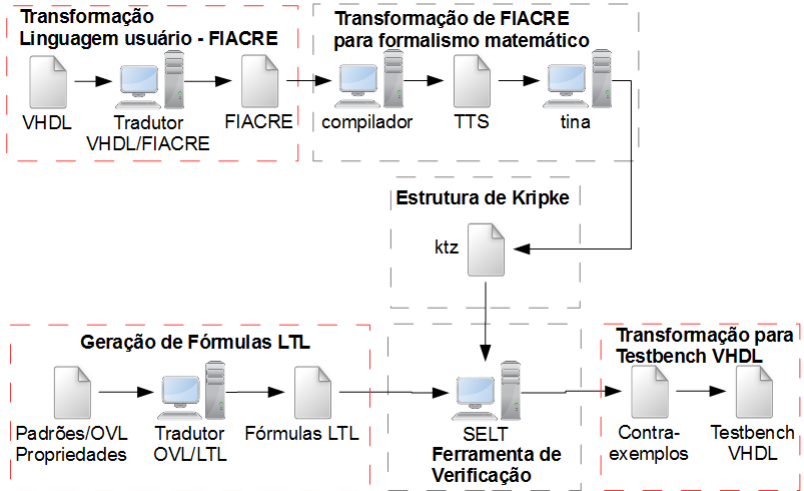


Figura 9 – Cadeia de verificação proposta

genérica e que será apresentada em detalhes no Capítulo 4. Na primeira etapa também são obtidas as fórmulas LTL para as propriedades a serem verificadas, partir de padrões OVL, conforme será detalhado no Capítulo 5. Na segunda etapa, o modelo FIACRE é compilado em um modelo TTS (arquivo .tts), e então a partir do TTS a ferramenta TINA (*Time Petri Net Analyzer*) (LAAS/CNRS, 2014) é utilizada para construir uma abstração adequada do grafo de alcançabilidade do sistema como um sistema de transição Kripke (arquivo .ktz) (formalismo matemático), finalizando a segunda transformação da cadeia tradicional. A estrutura de Kripke é um modelo muito utilizado para a descrição de sistemas concorrentes ou reativos, que representam sistemas com comportamento infinito. A estrutura de Kripke e as fórmulas LTL servem como entradas para a ferramenta de verificação SELT. O SELT retorna um arquivo com os resultados do *model checking*, incluindo contraexemplos para as propriedades que falharam.

Visto que já existem ferramentas para a compilação de FIACRE para TTS/Kripke, é necessário criar uma ferramenta para fazer a transformação VHDL sintetizável para modelo FIACRE, e uma ferramenta para fazer a tradução das propriedades descritas com observadores OVL para fórmulas LTL, linguagem de descrição de propriedades aceito pela ferramenta de verificação. Neste trabalho serão apresentadas regras de tradução VHDL-FIACRE (Capítulo 4) as quais permitem que a

transformação de modelos ocorra segundo o paradigma MDE (*Model Driven Engineering*), garantindo que o modelo FIACRE seja correto e represente o mesmo comportamento descrito no modelo VHDL, algo fundamental pois um erro na tradução invalida o processo de verificação formal.

Os contraexemplos gerados pelo SELT são apresentados como uma sequência de transições de estados na estrutura de Kripke que levam a situação de falha. Esse formato é de difícil leitura por um desenvolvedor VHDL, pois não é simples a associação de transições de estado no modelo formal para mudança de valor das variáveis do código VHDL. Portanto, os contraexemplos precisam ser convertidos em algo que os desenvolvedores estejam acostumados a trabalhar e, por isso, eles são traduzidos para uma sequência de conjuntos de entradas as quais formam uma onda de estímulo que geram o comportamento indesejado. A sequência de entradas é convertida para um *testbench* VHDL, similar ao que foi feito por Clarke (DÉHARBE; SHANKAR; CLARKE, 1998). O *testbench* pode ser executado em qualquer simulador VHDL para o que o desenvolvedor visualize a forma de onda do contraexemplo e descubra a fonte do erro.

3.3.1 Compilador FIACRE/TTS e a ferramenta TINA

O compilador FIACRE/TTS, pertencente à cadeia de verificação, desenvolvido inicialmente por Saad (SAAD et al., 2008) e constantemente aprimorado pelo grupo de pesquisa do LAAS da França, gera um modelo no formalismo matemático TTS com o objetivo de ser utilizado na ferramenta TINA. Este formalismo foi selecionado porque é muito utilizado para verificação formal de sistemas com as mesmas características dos representados pela linguagem FIACRE: concorrentes, assíncronos, distribuídos, paralelos ou não determinísticos. O TTS pode ser considerado como uma generalização do sistema de transições de base através da associação de tempos mínimos e máximos para as transições.

É possível interpretar o TTS como uma extensão da Rede de Petri Temporal (TPN) estruturada em duas partes: controle e dados. A parte de controle é descrita pela TPN comum e descreve os encadeamentos de eventos e atividades. Assim, as variáveis que são importantes para o controle do sistema são representadas como lugares da TPN empregada. A parte de dados descreve as variáveis que não pertencem à estrutura de controle do sistema. Os cálculos, que são realizados sobre a estrutura de dados, são representados associando expressões condi-

onais e ações às transições.

O compilador FIACRE/TTS, da cadeia de verificação, gera como saída um diretório (.tts) com dois arquivos (.net e .c). Após a tradução, se o modelo utiliza estruturas de dados externas à TPN (*unions, queues, booleans, etc.*), é necessário compilar o arquivo .c para ser utilizado pela ferramenta TINA. Depois de compilado, é possível executar a verificação formal, acompanhado da ferramenta de verificação SELT, pertencente ao ambiente TINA (BERTHOMIEU; VERNADAT, 2006).

O TINA é uma ferramenta para edição e análise de redes de Petri e TPN, desenvolvido no grupo de pesquisa do LAAS/CNRS da França. O TINA tem uma grande importância pelo fato de utilizar e ter suporte para ferramentas de verificação, além de ser capaz de editar e analisar outras extensões, como o TTS. As diferentes ferramentas que constituem o ambiente podem ser utilizadas isoladamente ou em conjunto. A ferramenta de verificação que integra o TINA é o SELT. A ferramenta SELT é um verificador de modelo para uma versão enriquecida de Estado/Evento LTL. No caso de não satisfazer a propriedade, o SELT é capaz de gerar uma sequência de contraexemplo em uma ou mais formas utilizáveis pelo simulador do TINA, a fim de executá-lo passo a passo (BERTHOMIEU; VERNADAT, 2006). O simulador do TINA não é utilizado nesse trabalho por não ser algo familiar a um desenvolvedor VHDL e também porque a estrutura de Kripke gerada para o modelo FIACRE não ser de fácil associação com as variáveis do código VHDL.

Para expressar as propriedades a serem verificadas através de fórmulas de lógica utilizadas no SELT, é utilizado um subconjunto da lógica temporal LTL. Estas fórmulas são construídas a partir de um conjunto de variáveis proposicionais e operadores lógicos usuais como os seguintes: implicação, negação, conjunção, disjunção e constantes (true e false).

Além desses, o subconjunto também possui operadores temporais do LTL: **always (G)**, **eventually (F)**, **next (X)**, **until (U)** e **release (R)**.

3.3.2 Tradução de Contraexemplo

O processo de tradução do contraexemplo gerado pelo SELT é apresentado na Figura 10.

Internamente, o tradutor de contraexemplos possui três ferramentas: **gerador de *testbench***, **filtro de contraexemplo** e **gerador de estímulos**. Essas ferramentas são *parsers* construídos com o

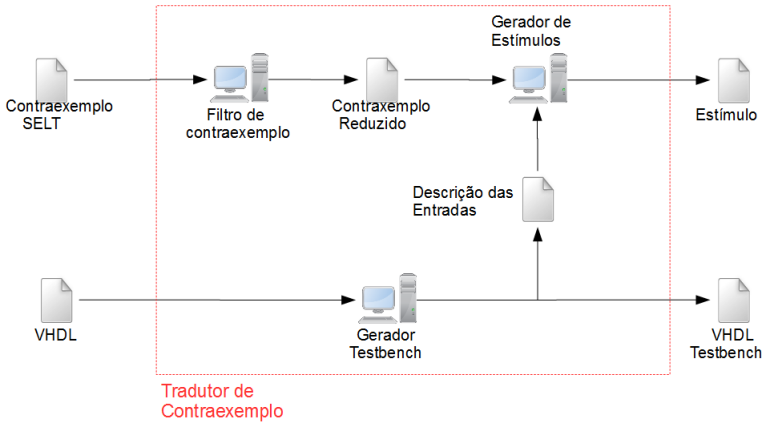


Figura 10 – Processo de Tradução de Contraexemplo

auxílio da biblioteca **Lex&Yacc** (LEVINE; MASON; BROWN, 1992).

O **gerador de *testbench*** recebe como entrada o arquivo VHDL com o circuito a ser verificado, observa as entradas declaradas no bloco **entity**, e cria um arquivo *testbench* VHDL e um arquivo contendo a descrição das entradas (nomes e tipos) para repassar para ferramenta **gerador de estímulos**. Esse *testbench* faz a leitura do arquivo contendo o estímulo de entrada, cria uma instância do circuito VHDL verificado e gera os estímulos em suas entradas. O **gerador de *testbench*** só precisa ser executado uma vez, pois os arquivos gerados por ele são os mesmos utilizados para leitura de contraexemplos para todas as propriedades verificadas. Somente se houver alteração nas portas de entrada do circuito que a ferramenta precisa ser executada novamente.

O **filtro de contraexemplo** recebe o contraexemplo no formato gerado pelo SELT e faz uma limpeza, removendo do *trace* todos os estados e eventos não relevantes para a geração de estímulo, textos não relacionados ao contraexemplo, etc., além de organizar o arquivo em um formato para ser processado pela ferramenta **gerador de estímulos**.

O **gerador de estímulos** recebe o contraexemplo reduzido e a descrição das entradas para gerar um arquivo contendo os estímulos de entrada que são lidos pelo *testbench*. A descrição das entradas é importante porque indica para a ferramenta quais eventos são relevantes no *trace* e também informa qual é o tipo de cada entrada, informação necessária para interpretar o contraexemplo do SELT e extrair dados sobre valores em cada estado. O arquivo de estímulos é uma reorga-

nização do contraexemplo em vários conjuntos de valores de entradas que devem ser aplicados sequencialmente ao circuito, de tal forma que ele alcance a situação onde a propriedade é violada. Ele pode ser interpretado como a Tabela 1, onde cada coluna representa uma entrada e as linhas são os valores que devem ser aplicados ao mesmo tempo ao circuito, sendo que as linhas são aplicadas sequencialmente.

Tabela 1 – Formato dos estímulos de entrada

	Entrada 1	Entrada 2	...	Entrada n
Estado 0	valor(1,1)	valor(2,1)	...	valor(n,1)
Estado 1	valor(1,2)	valor(2,2)	...	valor(n,2)
...
Estado (m-1)	valor(1,m-1)	valor(2,m-1)	...	valor(n,m-1)
Estado m	valor(1,m)	valor(2,m)	...	valor(n,m)

Para visualizar o contraexemplo, o usuário deve executar o *testbench* VHDL em um simulador para gerar uma forma de onda. O *testbench* somente é executável com a presença do arquivo com os estímulos e do código VHDL a ser verificado.

3.4 TRABALHOS RELACIONADOS

Vários trabalhos foram realizados com o objetivo de formalizar o processo de desenvolvimento de código VHDL para garantir que o código gerado esteja correto, seja com geração automática de código a partir de modelos alto nível ou na aplicação de verificação formal ao código.

Wood et al. (2008) propuseram uma ferramenta em que o código VHDL sintetizável é gerado a partir de diagramas de estado UML, utilizando o perfil MARTE (TAHA et al., 2007). O perfil MARTE é uma adaptação do UML para sistemas embarcados de tempo real. Apesar de o trabalho tentar trazer certo formalismo para geração do VHDL, o maior problema é que o UML não possui semântica bem definida, tornando muito difícil garantir formalmente que o modelo base para geração do VHDL respeite propriedades desejáveis.

Gomes et al. (2007) propuseram uma ferramenta em que código VHDL é gerado a partir de modelos em redes de Petri. A vantagem de se utilizar redes de Petri como modelo de alto nível é a possibilidade de se garantir por construção que o código gerado respeita as propriedades

desejáveis e verificadas para o circuito. A desvantagem desse método é a necessidade de o usuário ter conhecimento de redes de Petri além de que a construção de modelos complexos nesse formalismo é bastante trabalhosa e de difícil manutenção.

Wang et al. (2008) propuseram que o código VHDL seja gerado a partir de um modelo alto nível descrito em linguagem síncrona COLA. Essa linguagem possui sintaxe e semântica bem definidas, possibilitando verificação formal de propriedades em cima do modelo de alto nível, no entanto é uma linguagem pouco conhecida e difundida no meio acadêmico, o que poderia dificultar bastante seu uso.

Berry, Kishinevsky e Singh (2003) propuseram algo semelhante ao colocado por Wang, porém é utilizada a linguagem síncrona ESTEREL (BERRY; GONTHIER, 1992). ESTEREL, ao contrário do COLA, é uma linguagem síncrona bem conhecida e utilizada no meio acadêmico e na indústria, sendo bastante utilizada em trabalhos para representação de modelos de sistemas de tempo real crítico. A grande desvantagem do trabalho é o fato de ferramenta utilizada para geração do código VHDL ser proprietária e com preço elevado.

York et al. (1995) propuseram um ambiente de verificação formal de código VHDL e Verilog sintetizável, onde as propriedades a serem verificadas são descritas em lógica temporal CTL. A ferramenta se baseia na tradução do código VHDL para uma linguagem intermediária chamada IF, a qual é uma extensão da linguagem de descrição utilizada no SMV ¹. O problema dessa abordagem também é o fato da ferramenta ser proprietária.

Déharbe, Shankar e Clarke (1998) descreveram nesse trabalho uma ferramenta *open source* para *model checking* de código VHDL sintetizável. Nesse trabalho também é utilizada uma linguagem intermediária como *front-end* para a ferramenta de verificação. No entanto, a descrição das propriedades ainda necessita ser feita diretamente em lógica temporal CTL, exigindo ainda que o usuário possua um conhecimento avançado em *model checking*, diferentemente da metodologia proposta nesse trabalho que faz o uso de padrões de propriedade para tornar a complexidade das ferramentas transparente para o usuário. Embora mais informações sobre a ferramenta criada por Clarke et al. não estarem disponíveis, a ideia proposta de gerar contraexemplos como *testbench* VHDL é bastante interessante e foi utilizada na metodologia proposta nesse documento.

Ayav, Tuglular e Belli (2010) propuseram utilizar a ferramenta

¹<http://www.cs.cmu.edu/modelcheck/smv.html>

UPPAAL² para fazer *model checking* de código VHDL sintetizável. O trabalho apresenta uma metodologia de como traduzir as estruturas de VHDL para autômatos temporizados. No entanto, o UPPAAL também exige que as propriedades sejam representadas em lógica temporal (CTL no caso da ferramenta). No trabalho, os contraexemplos são gerados como autômatos temporizados, o que torna a tarefa de identificação do erro muito difícil para o desenvolvedor VHDL, já que a associação das transições de estados do autômato com a evolução dos valores das variáveis do código VHDL ser uma tarefa complicada. Além disso, esse tipo de contraexemplo exige que o desenvolvedor tenha conhecimento avançado do formalismo autômato temporizado.

A maior desvantagem da geração automática de código é a necessidade de aprender uma linguagem para modelagem em alto nível, o que impactaria bastante no processo de desenvolvimento atual, contrariando um dos objetivos desse trabalho. Além disso, a própria tradução dos modelos alto nível para códigos VHDL pode ser problemática, ainda mais quando os modelos alto nível utilizados não possuem uma sintaxe e semântica bem definida, como no caso do UML. A inclusão de verificação formal de propriedades sobre o código gerado não exige alteração no processo de como ele é construído, e por isso se tornou a melhor alternativa.

Em geral, os trabalhos sobre verificação de código VHDL utilizam uma linguagem intermediária para servir como entrada para as ferramentas de verificação, o que também é utilizado nesse trabalho, no entanto, não há um esforço em facilitar o processo de descrição de propriedades e se assume que o desenvolvedor deva ter conhecimento de lógica temporal, ou então o método exige que o usuário conheça a linguagem de descrição de comportamento utilizada. A utilização de padrões de propriedade e a da cadeia de verificação (não exigindo o conhecimento de outra linguagem a não ser VHDL) como propostos nesse trabalho, têm como objetivo corrigir essas dificuldades e também fazer com que a adoção da metodologia seja mais suave.

Além dos trabalhos acadêmicos citados, existem muitas ferramentas proprietárias desenvolvidas por grandes empresas da área de verificação que realizam verificação formal de código VHDL através de *model checking*, utilizando a linguagem PSL (PSL, 2004) ou SVA (SVA, 2009) para descrição das propriedades a serem verificadas. PSL e SVA são linguagens de descrição de propriedades, que além de possuir os operadores temporais presentes em linguagens de lógica temporal mais tradicionais (CTL e LTL), também possuem outros operadores que fa-

²<http://www.uppaal.org/>

cilitam a definição de propriedades para circuitos digitais, como operadores de *clock*, *reset* e outros. Muitas dessas ferramentas proprietárias também possibilitam geração automática de código VHDL a partir de modelos formais de alto nível. A grande desvantagem dessas ferramentas são os preços excessivamente altos (licenças anuais possuem valores acima de 100 mil dólares), os quais impossibilitam que empresas de médio porte possam utilizar. Como exemplo dessas ferramentas podemos citar o “*Questa Formal Verification*” (QUESTA, 2014) da Mentor Graphics, e o “*Cadence Incisive Formal Verifier*” (CADENCE, 2014) da Cadence.

A cadeia de verificação FIACRE é toda construída com ferramentas *open source*, o que torna a metodologia proposta vantajosa em relação ao uso de ferramentas proprietárias.

3.5 CONCLUSÃO

Neste capítulo foi apresentada a metodologia proposta neste trabalho. A metodologia adiciona a verificação de propriedades por *model checking* ao atual processo. As propriedades são definidas com o uso de padrões de propriedade derivados da biblioteca OVL, que são traduzidas para lógica LTL. O processo de verificação ocorre por meio de uma cadeia de verificação, onde o código VHDL é transformado em um modelo FIACRE que servirá de entrada para as ferramentas de verificação. Os contraexemplos gerados são transformados em *testbenches* VHDL.

A tradução de contraexemplos foi apresentada em detalhes e nos capítulos seguintes serão apresentadas as demais transformações necessárias para que o processo de verificação possa ocorrer: tradução VHDL-FIACRE e tradução OVL-LTL.

4 MODELO DE TRADUÇÃO VHDL-FIACRE

Neste capítulo são apresentadas as regras de tradução que possibilitam a transformação de um código VHDL sintetizável para um modelo em linguagem FIACRE. Essas regras são importantes para permitir que a tradução ocorra sob os princípios de engenharia dirigida a modelos. O capítulo começa apresentando o conceito de engenharia dirigida à modelos (MDE), para em seguida apresentar as regras de tradução e finalizar com a aplicação das regras a um pequeno exemplo.

4.1 ENGENHARIA DIRIGIDA A MODELOS

O termo MDE (*Model Driven Engineering*) é normalmente usado para descrever formas de desenvolvimento de sistemas em que modelos abstratos são criados e sistematicamente transformados em implementações concretas. No paradigma MDE os modelos são considerados como as principais entidades de todo o ciclo de vida do software. O objetivo principal do MDE é aumentar o nível de abstração no processo de desenvolvimento, ao focar na utilização de modelos que expressem melhor os conceitos dos domínios de aplicação e escondam as complexidades da plataforma alvo, cuja implementação seria obtida automaticamente através de transformações de modelos, partindo do modelo inicial em alto nível (SCHMIDT, 2006).

A transformação de modelos pode ocorrer entre níveis iguais e diferentes de abstração, como também pode ocorrer entre mesmos domínios e domínios diferentes. A transformação é a geração automática de um modelo destino a partir de um modelo origem. Essa transformação é definida por um conjunto de regras, que juntas, descrevem como um modelo na linguagem origem pode ser transformado em um ou mais modelos na linguagem destino. As transformações de modelos podem ser classificadas de transformação Modelo-Modelo (M2M) e transformação Modelo-Texto (M2T). As transformações M2T são aplicadas para gerar códigos em uma determinada linguagem a partir de um modelo dependente de plataforma e também podem ser utilizadas para gerar documentação. As transformações M2M são importantes quando se deseja obter um modelo em uma linguagem mais interessante para o usuário, seja por ser uma linguagem que ele tenha mais experiência ou por ser uma linguagem mais adaptada a uma aplicação específica, e também quando se deseja alterar o nível de abstração do

modelo, para aumentar ou diminuir o nível de detalhamento.

A transformação M2T consiste em definir regras que geram texto (código) a partir de elementos do modelo de entrada. A transformação M2T pode ser diferenciada entre as seguintes abordagens: manipulação direta, orientada a estrutura, operacional, baseada em *template*, relacional, baseada em grafos e híbrida.

Para que aconteça a transformação M2M, é preciso definir as regras de transformação do modelo que representa o transformador. Essas regras são baseadas nas estruturas dos elementos dos metamodelos de origem e de destino. O modelo de transformação recebe como entrada o modelo origem e o transforma no modelo destino. De acordo com os domínios dos metamodelos de origem e destino, o transformador de modelos pode gerar novos modelos no mesmo domínio, aumentando o detalhamento do modelo, e novos modelos em domínios diferentes (MDA, 2003). A Figura 11 ilustra, de forma geral, a estrutura base para a transformação de modelos. O modelo de entrada ‘Ma’ deve estar conforme o seu metamodelo de entrada ‘MMA’. O mesmo ocorre com o modelo de saída ‘Mb’, conforme ao metamodelo ‘MMb’, e o modelo de transformação ‘Mt’, conforme ao metamodelo ‘MMt’. Todos estes metamodelos, por sua vez, devem estar conforme a um meta-metamodelo ‘MMM’, que testa sua conformidade em relação a ele próprio. Para garantir que uma transformação de um modelo em outro seja correta, é necessário que os metamodelos de entrada, de saída e de transformação, tenham o mesmo meta-metamodelo. Desse modo, será gerado, a partir do modelo de entrada Ma, um modelo de saída Mb, de acordo com as regras de transformação escritas no modelo de transformação Mt.

Para a transformação de modelos, é necessário utilizar uma linguagem para expressar as regras de transformação. Uma linguagem bastante utilizada é a ATL (ATLAS, 2005). A ATL (*Atlas Transformation Language*) é uma linguagem híbrida, que possui construções imperativas e declarativas, feita para expressar transformações M2M como requeridas pela abordagem de MDE. Para que a transformação de modelos ocorra sistematicamente e com garantias de que o modelo na linguagem alvo represente o mesmo comportamento do modelo inicial, o metamodelo da linguagem de transformação deve estar conforme ao mesmo meta-metamodelo de ambos.

Em resumo, para ser possível traduzir o código VHDL para um modelo em FIACRE, tendo como base o MDE, será necessário definir um metamodelo VHDL, um metamodelo FIACRE e regras de transformação entre modelos VHDL e FIACRE. Os metamodelos do VHDL,

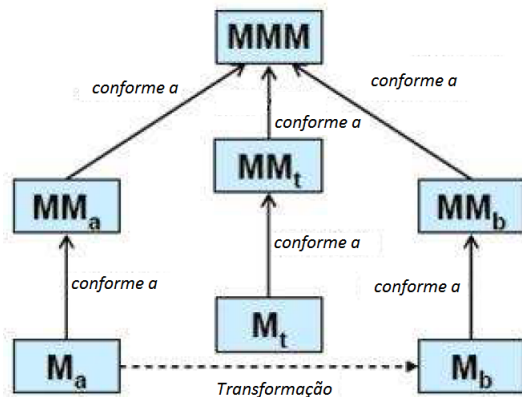


Figura 11 – Transformação M2M (MDA, 2003)

FIACRE devem estar conformes ao mesmo meta-metamodelo. Nessa dissertação serão apresentadas apenas as regras de transformação, sem estarem escritos em uma linguagem específica. O metamodelo do FIACRE já está disponível, resultado dos trabalhos do projeto TOPCASED. Alguns trabalhos de MDE relacionados com VHDL (WOOD et al., 2008) (MOREIRA et al., 2010) criaram metamodelos para a linguagem os quais podem ser utilizados, em trabalhos futuros, para a construção da ferramenta de tradução VHDL-FIACRE.

4.2 FIACRE

A linguagem FIACRE (*Format Intermédiaire pour les Architectures de Composants Répartis Embarqués* - Formato Intermediário para Arquiteturas de Componentes Distribuídos Embarcados) é uma linguagem intermediária formal para representar tanto os aspectos comportamentais quanto temporais de sistemas, em particular os sistemas distribuídos e embarcados, com a finalidade de verificação formal e simulação. FIACRE é uma linguagem assíncrona, orientada a processos que contém as seguintes noções:

- **Processos** descrevem o comportamento de componentes sequenciais. Um processo é definido como um conjunto de estados de controle, cada um associado com um pedaço de programa formado por construções determinísticas clássicas de linguagens de pro-

gramação (atribuições de variáveis, estruturas if-then-else, while, composições sequenciais), construções não-determinísticas (escolhas e atribuições não determinísticas), eventos de comunicação por portas e transições para outros estados.

- **Componentes** descrevem a composição de processos. Um componente é definido como uma composição paralela de componentes e/ou processos que se comunicam através de portas e variáveis compartilhadas. A noção de componente também permite restringir o modo de acesso e a visibilidade de variáveis compartilhadas e portas, associar restrições temporais à comunicação, e definir prioridades nos eventos de comunicação.

FIACRE foi desenvolvido conjuntamente por vários parceiros, tanto da academia quanto da indústria, no contexto do projeto TOP-CASED, como um *framework* para projetos envolvendo engenharia dirigida a modelos. FIACRE foi projetada para ser uma linguagem intermediária formal entre as linguagens de modelagem de alto nível e as ferramentas de verificação que trabalham com linguagens baseadas em formalismos matemáticos (autômatos, redes de Petri, sistemas de transição temporizados) (BERTHOMIEU et al., 2012).

A sintaxe da linguagem FIACRE é descrita da seguinte forma (SOUZA, 2010):

- **Tipos:** os tipos de dados aceitos são divididos em dois grupos - base e construídos. Os tipos de base são: inteiro (int), natural (nat) e booleano (bool). Tipos específicos podem ser criados utilizando os tipos nativos: matrizes, uniões, pilhas, enumerações etc.
- **Portas e canais de comunicação:** as portas (port) fazem parte da interface de um processo FIACRE. Elas são responsáveis pela comunicação, que pode ser síncrona ou não, e podem ser utilizadas para a troca de dados. Os canais (*channel*) são usados para definir um conjunto de tipos de dados aceitos por uma porta. Um perfil do tipo *none* sinaliza que a comunicação em questão é uma sincronização sem troca de valor.
- **Processos:** um processo FIACRE é uma máquina de estados composta por um conjunto finito de portas (sincronização com outros componentes ou processos), um conjunto finito de parâmetros, um conjunto finito de estados para controle interno, um conjunto

finito de variáveis locais, um conjunto de transições atômicas entre os estados. As transições definem o comportamento do processo, sendo que cada transição possui um estado de partida e outro de chegada, e seus corpos são formados por uma estrutura de controle.

- **Componente:** Um componente é composto por um conjunto finito de portas, um conjunto finito de parâmetros, um conjunto finito de variáveis locais, um conjunto finito de portais locais associadas a restrições temporais (são os canais de comunicação utilizados para interligar as instâncias que compõem o componente), um conjunto finito de prioridades, uma composição paralela de instâncias de processos (descreve a interação entre as instâncias que compõem o componente).
- **Composição:** a comunicação síncrona presente em FIACRE é resultado da composição paralela de um conjunto de instâncias. A composição promove um determinado tipo de comunicação dependendo do operador utilizado. A comunicação entre os processos pode ser totalmente em paralelo, totalmente sincronizada ou com algumas ações sincronizadas.

Um programa FIACRE, para ser completo, necessita de um conjunto de tipos, canais de comunicação, processos e componentes. O Código 4.1¹ apresenta um código de exclusão mútua entre 2 processos chamados de "proc". O processo inicia no estado "waitlock" e verifica se a seção crítica está liberada, o que é representado pela variável compartilhada "lock" possuindo valor 0. Se a seção crítica está liberada, o processo passa para o estado "waitlock2", onde espera um tempo, definido como o intervalo [0, 2], e tenta acessar a região crítica escrevendo sua identificação na variável compartilhada "lock" e passando para o estado "setlock". No estado "setlock" o processo espera um tempo definido no intervalo]2, ... [para então passar para o estado "testlock". No estado "testlock" o processo verifica se conseguiu acesso a região crítica (variável "lock" com sua identificação). Caso não foi tenha conseguido, o processo retorna ao estado "waitlock", do contrário passa ao estado "criticalSection" para realizar suas funções. No componente "main" é feita a comunicação entre os processos por meio da variável compartilhada "lock" e da composição "par" entre dois processos do tipo "proc".

¹<http://projects.laas.fr/fiacre/examples/fischer.fcr>

```

1 type id is 1..2
  type lock is 0..2

  process Proc (pid : id, &lock : lock) is
5     states WaitLock, WaitLock2, SetLock, TestLock, CriticalSection
      from WaitLock
          on (lock = 0);
          to WaitLock2
10     from WaitLock2
          wait [0, 2];
          lock := pid;
          to SetLock
15     from SetLock
          wait ]2, ...[;
          to TestLock
          from TestLock
          if lock = pid then
          to CriticalSection
          else
20     to WaitLock
          end
          from CriticalSection
          lock := 0;
          to WaitLock
25 component Main is
      var lock : lock := 0
      par
          Proc (1, &lock)
30     || Proc (2, &lock)
      end

Main

```

Código 4.1 – Exemplo de código FIACRE para exclusão mútua

4.3 MODELO DE TRADUÇÃO VHDL-FIACRE

Como mencionado na Seção 2.1, o VHDL pode ser utilizado para três atividades principais: documentação, simulação e síntese, sendo essas duas últimas de interesse para esse trabalho. Quando o código VHDL é utilizado para síntese, a ferramenta que faz essa atividade não executa o código, mas apenas avalia a estrutura dele e como estão conectados os sinais, com a finalidade de configurar o FPGA para realizar o circuito descrito. Somente em simulação é possível, antes da configuração do FPGA propriamente dito e testes em equipamento, visualizar se o VHDL sintetizável criado está descrevendo o circuito esperado. Na simulação é onde o VHDL é executado, e por isso é importante que o modelo de tradução para o FIACRE inclua além da tradução das estruturas da linguagem, o modo como um simulador interpreta o código.

Para que um interpretador VHDL consiga emular o modo concorrente de execução de um *hardware*, e com isso simular o comportamento do circuito digital real que será sintetizado no FPGA, ele utiliza o conceito de delta ciclo. Como no modelo FIACRE o objetivo é reproduzir o comportamento do circuito na simulação, é fundamental que o modelo do código emule esse conceito conforme realizado pelo interpretador. O delta ciclo é o conceito que diferencia a execução de um VHDL em relação a linguagens de programação de execução sequencial, e nada mais é que uma ordenação de como o código é executado e quando as variáveis têm seus valores atualizados, de maneira que se emule uma execução concorrente sem perder determinismo.

- **Visão Geral:**

A Figura 12 mostra o conceito geral da tradução de um código VHDL sintetizável para um modelo FIACRE correspondente. O modelo FIACRE correspondente é um **componente** que faz uma composição síncrona de dois **processos**: *input_generator* e *delta_cycle*. O processo *delta_cycle* é responsável por executar o código VHDL: declarações ou partes de código presentes no bloco **architecture** que são executados em paralelo, também chamados de *concurrent statements*. A execução das declarações concorrentes pelo processo FIACRE *delta_cycle* emula o comportamento ordenado por um delta ciclo, e é obtido no processo FIACRE através da coordenação de passagem de valores entre variáveis da linguagem. No entanto, para que as declarações concorrentes sejam processadas é preciso que sejam gerados valores de entradas que estimulem o circuito, sendo que essa geração de entradas ocorre no processo *input_generator*. O processo *input_generator* é responsável por gerar todos os valores possíveis para as entradas declaradas no bloco **entity** do código VHDL. As entradas são geradas por meio de uma máquina de estados (todo processo FIACRE é uma máquina de estados) e são repassadas para o processo *delta_cycle* por meio de um *channel* de sincronização com passagem de valores. A passagem de valores de entrada é sincronizada entre os processos, para que não aconteça no espaço de estados gerado para o sistema de valores serem gerados e não serem processados pelo código VHDL interpretado pelo processo *delta_cycle*.

É importante ressaltar que as declarações concorrentes traduzidas pelo atual modelo de transformação são apenas códigos ou estruturas VHDL do tipo **process**, não estão sendo traduzidas instâncias de outros códigos VHDL internamente ao que está sendo verificado. A tradução de declarações que são instâncias pode ser feita em trabalhos futuros. É importante lembrar também que a estrutura **process**

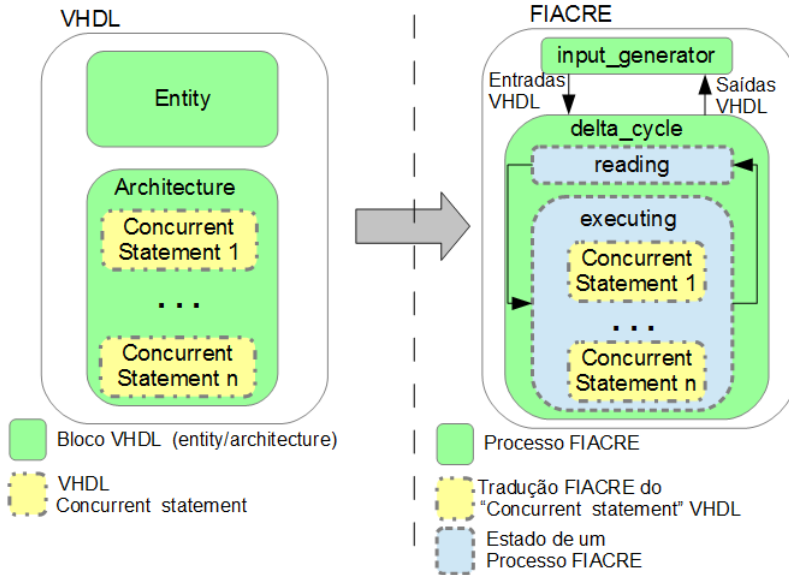


Figura 12 – Modelo Geral de Transformação VHDL-FIACRE

da linguagem VHDL é bem diferente de um processo FIACRE, sendo apenas uma declaração concorrente cujo código interno é executado sequencialmente, enquanto que o processo FIACRE é uma máquina de estados.

- **Processo `delta_cycle`:**

O processo `delta_cycle` tem uma estrutura fixa composta por uma máquina de estados com dois estados: **reading** e **executing**. No estado **reading**, o processo recebe os valores das entradas VHDL gerados pelo processo `input_generator`, através de um *channel* de sincronização com passagem de valores. A passagem sincronizada é importante, porque impede que o processo `delta_cycle` passe para o estado **executing** sem que novos valores de entrada tenham sido gerados ou atualizados. O processo `delta_cycle` possui dois estados devido a uma limitação da própria linguagem FIACRE, pois não é possível em uma mesma transição ocorrer uma sincronização e uma atualização de variável compartilhada. A comunicação entre o `input_generator` e o `delta_cycle` é por sincronização com passagem de valor para as entradas, e atualização de variável compartilhada para as saídas (calculadas

quando o código é processado no estado **executing** do **delta_cycle**). O processo de passagem de valores de entrada e saída são em métodos diferentes para evitar a necessidade de dois passos de sincronização, reduzindo o número de estados. A Figura 13 mostra a estrutura do processo **delta_cycle**, comparando com a estrutura de um código VHDL.

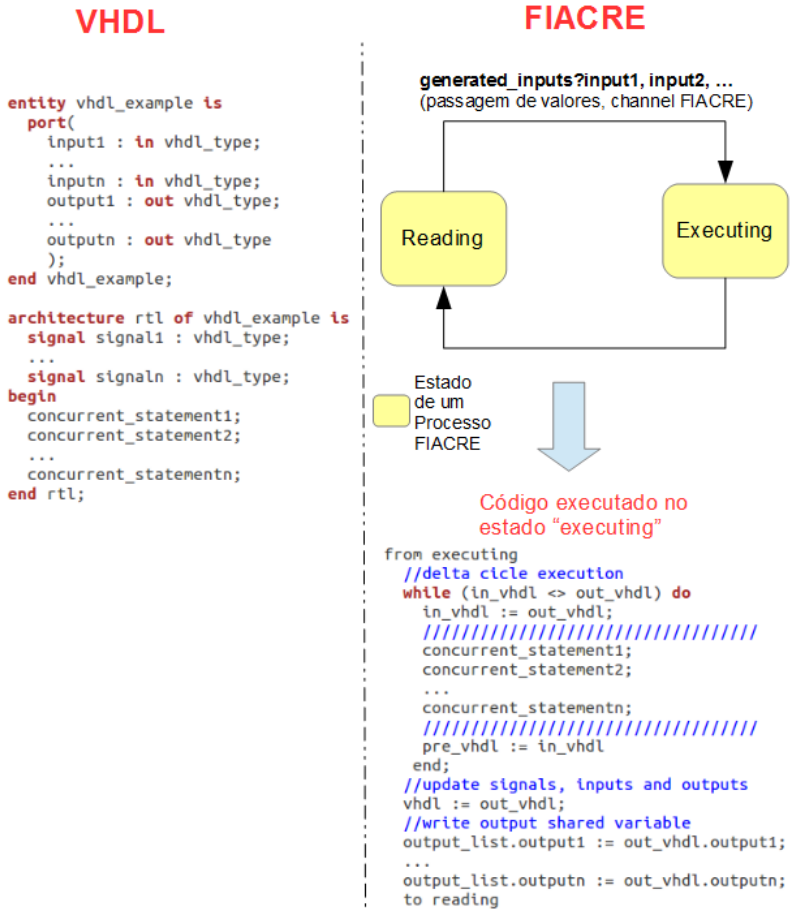


Figura 13 – Esqueleto do processo FIACRE *delta_cycle*

O código do bloco **Architecture** do VHDL está encapsulado no estado **executing** do processo FIACRE **delta_cycle**. Todas declarações concorrentes do código VHDL (não incluído constantes e declarações de sinais) estão contidas dentro de uma estrutura *while* no

estado **executing** do modelo FIACRE, com as conversões de tipos VHDL para tipos FIACRE assim como as típicas declarações do VHDL (*when..else*, *select..with*, *process*, etc, detalhados na próxima seção). Um *loop while* em FIACRE se comporta da mesma maneira que em linguagens de programação convencionais.

Como a ideia é traduzir o código VHDL e executar como um interpretador faz em simulação, o *while* é utilizado para reproduzir o conceito de delta ciclo. Cada execução da estrutura *while* funciona como a execução de um delta ciclo, e para garantir o determinismo na execução é utilizado um conjunto de variáveis para auxiliar o processamento. Um tipo FIACRE *record* (similar a um *struct* na linguagem C) é criado contendo todas as variáveis do código VHDL (entradas, saídas e *signals*). São geradas quatro instâncias do *record*: **vhdl**, **in_vhdl**, **out_vhdl** e **pre_vhdl**. A instância **out_vhdl** representa os valores calculados no delta ciclo atual, a instância **in_vhdl** representa os valores calculados no delta ciclo anterior e a instância **pre_vhdl** representa os valores calculados a 2 delta ciclos anteriores. O Código 4.2 apresenta a estrutura do processo **delta_cycle**, que é a mesma para todos os VHDL traduzidos. O critério para término do loop *while* é os *records in_vhdl* e *out_vhdl* terem os mesmos valores para todas as variáveis (linha X), o qual é o mesmo critério de parada para a cadeia de delta ciclos em uma execução do código VHDL por um interpretador: a não ocorrência de eventos de mudança de valor para nenhuma variável (entrada, saída ou *signal*), discutido na Seção 2.1. Quando o critério de parada é atingido, o processo FIACRE atualiza o *record vhdl* com os valores de **out_vhdl** (linha 28), atualiza a variável compartilhada **output_list** também com os mesmos valores (linhas 30-32) e retorna ao estado **reading** para receber novos valores gerados para as entradas (linha 33). Essa atualização da variável compartilhada é importante pois é o modo como o processo **delta_cycle** compartilha com o processo **input_generator** os valores de saída calculados.

```

1 process delta_cycle [input_list : in input_list]
  (&output_list : read write output_list) is
  states reading, executing
  var //signals+inputs+outputs
5   vhdl : vhdl_variables := INIT_VHDL_VARIABLES,
     in_vhdl : vhdl_variables := INIT_VHDL_VARIABLES,
     pre_vhdl : vhdl_variables := INIT_VHDL_VARIABLES,
     out_vhdl : vhdl_variables := INIT_VHDL_VARIABLES,

10  init to reading

  from reading
     input_list?out_vhdl.input1, ..., out_vhdl.inputn;
  to executing

```

```

from executing
  //delta cycle execution
  while (in_vhdl <> out_vhdl) do
    in_vhdl := out_vhdl;
20
    //concurrent statements
    //////////////////////////////////////
    ...
    //////////////////////////////////////
25    pre_vhdl := in_vhdl
  end;
  //update signals, inputs and outputs
  vhdl := out_vhdl;
  //write output shared variable
30  output_list.output1 := out_vhdl.output1;
  ...
  output_list.outputn := out_vhdl.outputn;
  to reading

```

Código 4.2 – Código do proceso *delta_cycle*

A transição do estado **reading** para o **executing** é equivalente a passagem do tempo na execução do simulador até a próxima alteração nas entradas, porém no modelo FIACRE não ocorre referência explícita a tempo. Como estamos trabalhando com verificação lógica em cima de VHDL sintetizável, o conceito de tempo é irrelevante pois estamos trabalhando com a hipótese de sincronismo (mudança de valores não demanda tempo) e também porque a evolução do estado do sistema depende unicamente da sequência de alteração dos valores das entradas, e não do tempo entre essas modificações.

- **Processo `input_generator`:**

O processo **input_generator** é responsável por gerar valores para as entradas definidas no bloco **Entity** do código VHDL. Em geral, como o objetivo é a verificação exaustiva, o modelo é criado para gerar qualquer valor para as entradas, sendo esses valores repassados para o processo **delta_cycle** através do *channel* de sincronização que compartilham.

A estrutura do processo **input_generator** no modelo FIACRE segue o formato mostrado no Código 4.3. Para o código apresentado é considerado que o código VHDL de origem possui “n” entradas.

```

1  process input_generator [generated_inputs : out input_list]
   (&output_list : read output_list) is
   states generating
   var input1 : fiacre_type := init_value,
5     ...
     inputn : fiacre_type := init_value
   init to generating
   from generating
     input1, ..., inputn := any;
10  generated_inputs!input1, ..., inputn;

```

 Código 4.3 – Estrutura do processo **input_generator**

O modelo do **input_generator** é um processo com apenas um estado em *loop* infinito. As linhas 4-6 do Código 4.3 apresentam a declaração de “n” variáveis FIACRE, uma para cada entrada declarada no bloco **Entity** do código VHDL a ser verificado. As variáveis devem ter o tipo FIACRE correspondente ao tipo VHDL da entrada que está representando, e mesmo nome. A linha 9 é a parte do código que gera qualquer valor para cada uma das entradas, sendo que o valor gerado está dentro do intervalo de valores possíveis para cada variável. A linha 10 é sincronização com o processo **delta_cycle** para passagem de valores e a linha seguinte é a transição (para o mesmo estado).

O modelo permite a definição de entradas com valor constante. Nesse caso a variável que representa a entrada não terá seu valor gerado pela expressão “any”, mas sim por uma expressão que associa o valor desejado. Para entradas de tipo natural ou inteiro também é permitido ao usuário restringir o intervalo de valores possíveis para a mesma. Nesse caso a restrição deve ser feita na declaração do tipo da variável que representará a entrada. Por exemplo, se uma entrada for restrita ao intervalo [1,10], a declaração do tipo para variável que representa essa entrada será uma expressão do tipo “1..10”. O valor gerado para entrada continua sendo através da expressão “any”.

Um modelo de entradas em que quaisquer valores são gerados para as mesmas, conforme apresentado no Código 4.3, pode levar a um espaço de estados muito grande, ocasionando um problema comum no *model checking* que é a explosão de estados. Além disso, gerar qualquer valor pode não ser muito realístico visto que algumas entradas modificam seus valores de modo ordenado (um sinal de *clock* por exemplo está sempre alternando seu estado periodicamente) ou dependendo do comportamento das saídas, ou então determinados valores de entrada não são de interesse para a propriedade que se deseja verificar. Ao aplicar essas restrições, é possível obter um modelo que melhor representa as entradas, contendo apenas comportamentos possíveis ou de interesse, e que também gere um espaço de estados muito menor e diminua o risco de explosão de estados.

Para ser possível o usuário fazer restrições quanto ao modelo em que as entradas são geradas, sem que o mesmo precise alterar o código FIACRE, nesse trabalho é proposto que se utilize também a biblioteca OVL para essa finalidade. Mais detalhes sobre a geração do **input_generator** com restrições OVL serão apresentados no Capítulo

seguinte.

- **Regras de Tradução:**

A tradução de VHDL sintetizável para FIACRE pode ser resumida nas seguintes regras:

1. Identificar as entradas e saídas (nome e tipo) presentes no bloco **Entity** do código VHDL.
2. Identificar os sinais internos (*signals*), nome e tipo, declarados no bloco **Architecture** do código VHDL.
3. Criar um *channel* de sincronização de nome **input_list**. O *channel* deve repassar uma quantidade de dados equivalente ao número de entradas. Para cada entrada deve existir um campo no *channel* de tipo FIACRE equivalente ao tipo VHDL da mesma.
4. Criar um *record* de nome **output_list** cuja quantidade de elementos é o número de saídas do código VHDL. Para cada saída do VHDL deve existir um elemento de mesmo nome no *record* e tipo FIACRE equivalente ao tipo VHDL (mais detalhes da tradução de tipos será discutido posteriormente).
5. Criar um tipo *record* de nome **vhdl_variables**, cuja quantidade de elementos é a soma do número de entradas, saídas e *signals* do VHDL. Para cada entrada, saída ou sinal interno do VHDL deve existir um elemento de mesmo nome no *record* e tipo FIACRE equivalente ao tipo VHDL.
6. Criar o processo **input_generator**. Ele deve ser criado para gerar qualquer valor para as entradas conforme o formato mostrado no Código 4.3. Caso seja utilizadas restrições definidas por observadores OVL, o formato será o apresentado em detalhes no Capítulo 5.
7. Criar o processo **delta_cycle**. Ele deve possuir a estrutura mostrada no Código 4.2. As variáveis **in_vhdl**, **out_vhdl**, **pre_vhdl** e **vhdl** devem ser do tipo *record* **vhdl_variables**. O processo deve possuir dois estados: **reading** e **executing**. No estado **reading** ocorre a sincronização com o processo **input_generator** por meio do *channel* **input_list** para recepção dos valores de entradas. No estado **executing** o código FIACRE correspondente às declarações concorrentes do bloco **Architecture** são executados.

8. Toda declaração concorrente presente no bloco **Architecture** do código VHDL está presente dentro do *loop while* no estado **executing** do processo **delta_cycle**. A tradução de cada tipo de declaração concorrente para um código FIACRE correspondente será detalhada a seguir.
9. Criar um **Componente** FIACRE de nome idêntico ao do circuito descrito no código VHDL (nome declarado no bloco **Entity** do VHDL). O componente é uma composição paralela entre os processos **delta_cycle** e **input_generator**. Os processos devem compartilhar um *channel* de sincronização de nome “**input_list**” (tipo **input_list**, regra 3), e uma variável compartilhada de nome “**output_list**” (tipo **output_list**, regra 4).

4.3.1 Tradução de Declarações Concorrentes

Nessa seção é apresentado o princípio de tradução das declarações mais comuns e relevantes do VHDL, mas ainda falta a tradução de declarações como instâncias de código, *generate*, associações de **variables**, etc. As estruturas traduzidas foram suficientes para resolução dos exemplos utilizados na dissertação e também para maior parte dos códigos desenvolvidos na empresa.

É importante lembrar que todo código do bloco **Architecture** está encapsulado dentro do *loop while* no estado **executing** do processo **delta_cycle**.

- **Atribuição de Valores:**

No VHDL, a atribuição de valor para um determinado sinal (*signal*) é representado pela expressão “<=”. No FIACRE, a atribuição de valor para uma variável é representado pela expressão “:=”. Todo *signal*, entrada ou saída do código VHDL é traduzido no modelo FIACRE para uma variável com tipo equivalente.

- **when..else**

A estrutura **when..else** é uma associação condicional. A decisão de qual valor será atribuído ao sinal *signal* depende da satisfação de uma determinada condição. A ordem de verificação das condições define a prioridade (ordem decrescente) de qual valor será atribuído. Em VHDL, a declaração **when..else** ocorre do seguinte modo:

VHDL:

```

2  signal <= value1 when condition1 else
      value2 when condition2 else
      value3;

```

Em FIACRE, o **when..else** é traduzido como:

FIACRE:

```

      if condition1 then
2   signal := value1
      else
4   if condition2 then
      signal := value2
6   else
      signal:= value3
8   end if
end if

```

- **select..when**

A estrutura **select..when** é uma associação condicional. A decisão de qual valor será atribuído ao sinal *signal_out* depende do valor de um segundo sinal *signal_in*. Em VHDL, a declaração **select..when** ocorre do seguinte modo:

VHDL:

```

with signal_in select
2  signal_out <= value1 when condition1,
      value2 when condition2,
4  value3 when condition3,
      value4 when others;

```

Em FIACRE, o **select..when** é traduzido como:

FIACRE:

```

      if signal.in = condition1 then
2   signal.out := value1
      end if;
4   if signal.in = condition2 then
      signal.out := value2
6   end if;
      if signal.in = condition3 then
8   signal.out := value3
      end if
10  if (signal.in <> condition1)
      and (signal.in <> condition2)
12  and (signal.in <> condition3) then
      signal.out := value4
14  end if

```

- **Process**

O **Process** é uma estrutura especial do VHDL onde o código interno é executado somente se um dos sinais presentes na lista de sensibilidade

tiver seu valor modificado. A execução do código interno ao **Process** é sequencial. Algumas estruturas do VHDL somente podem ser usados dentro de um **Process**: **if..else**, **while**, **for**, **case**. As expressões **if..else**, **while** e **for** possuem equivalentes no FIACRE e, portanto, a tradução é automática.

Todo **Process** VHDL é convertido em uma estrutura **if** FIACRE. A estrutura **if** analisa se algum dos sinais presentes na lista de sensibilidade alterou valor em relação a última execução e, em caso positivo, o código é executado. No modelo FIACRE, um sinal teve seu valor modificado quando a variável FIACRE que o representa possui valores diferentes nos *records* **in_vhdl** e **pre_vhdl**. Em VHDL, a declaração **process** ocorre do seguinte modo:

VHDL:

```

1 process(a, b, c)
2 begin
  -- código interno ao process
4 end process;
```

Em FIACRE, o **process** é traduzido como:

FIACRE:

```

1 if (pre_vhdl.a<>in_vhdl.a) or
2   (pre_vhdl.b<>in_vhdl.b) or
   (pre_vhdl.c<>in_vhdl.c) then
4 /*código interno ao process*/
end if
```

• Case

No VHDL, a declaração **case** só pode existir dentro de um **process**. Por estar dentro de **process**, o **case** é de execução sequencial. Em VHDL, a declaração do **case** é do seguinte modo:

VHDL:

```

1 case signal.in is
2   when condition1 =>
   signal_out <= value1;
4   when condition2 =>
   signal_out <= value2;
6   when condition3 =>
   signal_out <= value3;
8   when others =>
   signal_out <= value4;
10 end case;
```

Em FIACRE, o **case** precisa ser convertida em um grupo de expressões do tipo **if**, conforme mostrado abaixo:

FIACRE:

```

    if signal_in = condition1 then
2   signal_out := value1
    end if;
4   if signal_in = condition2 then
        signal_out := value2
    end if;
6   if signal_in = condition3 then
8       signal_out := value3
    end if
10  if (signal_in <> condition1)
        and (signal_in <> condition2)
12      and (signal_in <> condition3) then
        signal_out := value4
14  end if

```

4.3.2 Tradução de Tipos e Operadores

A linguagem FIACRE possui tipos naturais (**nat**) e inteiros (**int**), com os operadores usuais: soma (+), subtração (-), negação (-), multiplicação (*), divisão (/), módulo (%), igualdade (=), diferença (<>), maior (>), menor (<), maior ou igual (>=) e menor ou igual (<=). A tradução dos tipos VHDL **integer** e **natural** para esses tipos FIACRE é automática.

Os tipos VHDL **boolean** e **bit** podem ser traduzidos para o tipo FIACRE **bool**. Esse tipo FIACRE é um booleano que possui os operadores lógicos básicos **and**, **not** e **or**. As operações lógicas **nand**, **xor** e **nor** (presentes para os tipos **bit** e **boolean** do VHDL) precisam ser construídos a partir dos operadores básicos.

O FIACRE permite a construção de *arrays* de qualquer tipo nativo da linguagem, no entanto, tem suporte apenas para operação de igualdade e diferença. É possível representar o tipo VHDL **bit_vector** através de um *array* de **bool** em FIACRE, porém as operações de *shift* lógico e aritmético, operadores lógicos e comparações não possuem suporte. Para os exemplos utilizados nesse trabalho, as operações sobre *array* foram traduzidas manualmente, caso a caso, quando necessário. No entanto, para que o processo de tradução VHDL-FIACRE seja automático, no futuro será necessário a definição desses operadores. A linguagem FIACRE permite a adição de primitivas que utilizem seus tipos nativos, sendo essas funções declaradas em um arquivo separado (com o código escrito em linguagem C) e importadas ao modelo. Essa funcionalidade será utilizada para a criação das operações necessárias sobre *arrays*.

Além dos tipos nativos, grande parte dos códigos VHDL sintetizáveis desenvolvidos fazem uso de quatro tipos adicionais presen-

tes nas bibliotecas *std_logic_1164* e *numeric_std*: **signed**, **unsigned**, **std_logic** e **std_logic_vector**. Os tipos **std_logic** e **std_logic_vector** podem ser traduzidos para tipos FIACRE **bool** e *array* de **bool** respectivamente, visto que para VHDL sintetizável somente são usados os valores 0 e 1 para esses tipos (a diferença de um **std_logic** para um **bit** é que o primeiro apresenta mais 7 tipos de valores diferentes além de 1 ou 0, porém as boas práticas para código VHDL sintetizável não fazem uso dos outros valores). Os tipos VHDL **signed** e **unsigned** são uma interpretação do tipo **std_logic_vector** como inteiros com sinal ou sem, e nos exemplos utilizados nessa dissertação foram traduzidos para tipos **int** FIACRE, no entanto o ideal é criar esses tipos na linguagem FIACRE (também com auxílio de funções em C) para que a tradução seja automática e que tenha suporte para operações a níveis de bit (deslocamento e rotação).

Os tipos enumerados em VHDL são traduzidos em FIACRE como **union**, com comportamento idêntico. A criação de tipos escalares e compostos em FIACRE ocorre de modo similar ao VHDL através do uso da diretiva **type** (no VHDL também é o mesmo comando), e a tradução é automática.

4.4 EXEMPLO

Para exemplificar o uso das regras de tradução, elas serão aplicadas para converter o código VHDL do controlador de memória apresentado na Seção 2.1.1.

Na estrutura **Entity** são declaradas 5 entradas (**clk**, **reset**, **mem**, **rw** e **burst**) e 3 saídas (**oe**, **we** e **we_me**). A Figura 14 apresenta um diagrama de blocos representando o código da estrutura VHDL **Architecture**. O VHDL desse exemplo possui somente duas declarações concorrentes do tipo **process**, que compartilham dois sinais: “**state_reg**” e “**state_next**”. Um processo (chamado de “*sequential process*”) é um circuito sequencial responsável por atualizar o valor de “**state_next**” com o valor de “**state_reg**” na ocorrência da borda de subida do *clock*. O outro processo (chamado de “*combinational process*”) é um circuito combinacional responsável por atualizar as saídas e definir o próximo estado (atualizar o “**state_next**”). O código VHDL completo do controlador de memória pode ser encontrado no Apêndice A.

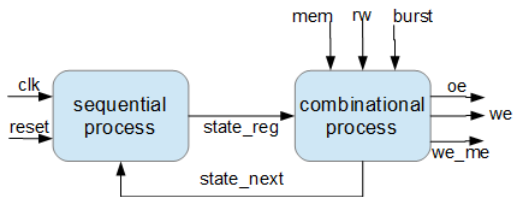


Figura 14 – Diagrama de blocos do VHDL do controlador de memória

4.4.1 Modelo FIACRE

As regras 1 e 2 definem que o tradutor deve identificar quais são as entradas, saídas e sinais do código VHDL, que para o controlador de memória são os seguintes:

- Entradas:
 - Nome: clk. Tipo: std_logic.
 - Nome: reset. Tipo: std_logic.
 - Nome: mem. Tipo: std_logic.
 - Nome: rw. Tipo: std_logic.
 - Nome: burst. Tipo: std_logic.
- Saídas:
 - Nome: oe. Tipo: std_logic.
 - Nome: we. Tipo: std_logic.
 - Nome: we_me. Tipo: std_logic.
- Sinais
 - Nome: state_reg. Tipo: enumerado.
 - Nome: state_next. Tipo: enumerado.

A regra 3 define a criação do *channel* de sincronização com base nas entradas do VHDL. A linha 5 do Código 4.4 apresenta a definição do *channel* para o exemplo. Os tipos **std_logic** das entradas foram convertidas em FIACRE para tipos **bool**, e são passados 5 valores desse tipo no *channel*, um para cada entrada. As linhas 7-9 mostram a

aplicação da regra 4, criando um *record* de nome **output_list** com base nas saídas declaradas no VHDL. Os tipos **std_logic** das saídas também foram convertidas em FIACRE para tipos **bool**. As linhas 11-15 são geradas devido a regra 5 (além das entradas e saídas, o controlador só possui dois sinais internos para os estados). No VHDL do exemplo é usado um tipo enumerado para os sinais “**state_reg**” e “**state_next**”, que no modelo FIACRE foi traduzido para um *union*, mostrado nas linhas 1-3.

```

1 type mc_state_type is union
    idle | read1 | read2 | read3 | read4 | writel
end

5 channel input_list is bool#bool#bool#bool#bool

type output_list is record //record com as saidas
    oe : bool, we : bool, we_me : bool
end

10 type vhdl_variables is record
    state_reg : mc_state_type, state_next : mc_state_type,
    oe : bool, we : bool, we_me : bool,
    clk : bool, reset : bool, mem : bool, rw : bool, burst : bool
15 end

```

Código 4.4 – Declaração de tipos e *records* para o exemplo

No Código 4.5 é apresentado o modelo do processo FIACRE “**input_generator**” para o exemplo (regra 6). As linhas 1-2 apresentam a declaração do processo, definindo que o mesmo tem como saída um *channel* de sincronização de nome “**generated_inputs**” (tipo **input_list**, regra 3) e processa uma variável compartilhada de nome “**output_list**” (tipo **output_list**, regra 4), sendo somente leitura. Na linha 4 são declaradas as variáveis FIACRE que representam as entradas do VHDL e, comparando com o código no Apêndice A, vemos que novamente os tipos **std_logic** foram traduzidos em FIACRE como **bool**. O processo foi traduzido para gerar qualquer valor para as entradas e, por isso, a linha 7 temos um exemplo do uso da diretiva **any**. A expressão mostrada na linha 8 é passagem sincronizada de valor para o processo “**delta_cycle**”, ocorrendo somente quando este último está no estado “**reading**”.

```

1 process input_generator [generated_inputs : out input_list]
    (&output_list : read output_list) is
    states generating
    var clk, reset, mem, rw, burst : bool := false
5    init to generating
    from generating
        clk, reset, mem, rw, burst := any;
        generated_inputs!clk, reset, mem, rw, burst;
    loop

```

Código 4.5 – Processo FIACRE “input_generator” do controlador de memória

O Código 4.6 apresenta a estrutura resumida do processo “**delta_cycle**”, seguindo a regra 7 de tradução. O código FIACRE contém a tradução somente do **process** sequencial responsável pela atualização do sinal “**state_reg**” (código disponível no Apêndice A. A tradução do outro **process** foi omitida no modelo. No FIACRE, todas as sentenças entre */* e */* são comentários.

```

1 process delta_cycle [input_list : in input_list]
    (&output_list : read write output_list) is
    states reading, executing
    var vhdl : vhdl_variables := INIT_VHDL_VARIABLES,
5     in_vhdl : vhdl_variables := INIT_VHDL_VARIABLES,
        pre_vhdl : vhdl_variables := INIT_VHDL_VARIABLES,
        out_vhdl : vhdl_variables := INIT_VHDL_VARIABLES,

    init to reading
10    from reading
        input_list?out_vhdl.clk, out_vhdl.reset, out_vhdl.mem,
            out_vhdl.rw, out_vhdl.burst;
        to executing

15    from executing
        /*execução do delta ciclo*/
        while (in_vhdl <> out_vhdl) do
            /*atualiza sinais de in_vhdl list (valores do último
                delta ciclo)*/
20            in_vhdl := out_vhdl;
            /*process 1, sequential circuit*/
            if (pre_vhdl.clk<>in_vhdl.clk) or
                (pre_vhdl.reset<>in_vhdl.reset) then
                if in_vhdl.reset then
25                    out_vhdl.state_reg := idle
                else
                    if (not pre_vhdl.clk) and in_vhdl.clk then
                        out_vhdl.state_reg := in_vhdl.state_next
                    end
30                end
            end;
            /*process 2, combinational circuit*/
            /*código FIACRE gerado para o processo combinacional*/
            If ...
35            ...
            end;
            /*atualiza sinais do record pre_vhdl*/
            pre_vhdl := in_vhdl
        end;
40    /*atualiza sinais, entradas e saídas*/
        vhdl := out_vhdl;
        /*atualiza variável compartilhada*/
        output_list.oe := out_vhdl.oe;
        output_list.we := out_vhdl.we;
45    output_list.we_me := out_vhdl.we_me;
    to reading

```

Código 4.6 – Código FIACRE do “delta_cycle” para o controlador de memória

A linha 1 do Código 4.6 apresenta a declaração do processo, definindo que o mesmo recebe de entrada um *channel* de sincronização de nome “input_list” (tipo **input_list**, regra 3) e processa uma variável compartilhada de nome “output_list” (tipo **output_list**, regra 4), podendo ler e escrever na mesma. As linhas 10-13 descrevem o estado “reading”, sendo que na linha 11 ocorre a sincronização com processo “input_generator” através do *channel* “input_list”. Os novos valores de entrada são gravados no *record* “out_vhdl”, e a transição para o estado “executing” somente ocorre de modo sincronizado com o processo gerador das mesmas.

Como definido na regra 8, o código da estrutura **Architecture** do VHDL do exemplo está encapsulado no *loop while* do estado “executing”. O VHDL do controlador de memória possui as seguintes entradas e sinais “state_reg”, “state_next”, “clk”, “mem”, “reset”, “rw” e “burst”. O *loop while* (linha 17) observa mudanças nessas variáveis ao comparar os *records* FIACRE “in_vhdl” e “out_vhdl”. Quando o critério de parada do *loop* é atingido, os valores das entradas, saídas e sinais são atualizados (linha 41), a variável compartilhada é atualizada com os novos valores das saídas (linhas 43-45) e o processo retorna ao estado “reading” para receber novos valores para as entradas (linha 46).

A linha 20 do Código 4.6 é a atualização do *record* “in_vhdl” com os valores calculados no último delta ciclo (última execução do *loop*). Os valores presentes no *record* “in_vhdl” são os utilizados para o processamento do código VHDL. As linhas 22-23 representam a estrutura *if* que corresponde a tradução do **process** VHDL do circuito sequencial. As linhas 22-23 fazem a verificação se algum sinal da lista de sensibilidade (somente “clk” e “reset” nesse caso) foi modificado e, em caso positivo, o código é executado. A comparação da lista de sensibilidade é feita entre os valores da variável presentes nos *records* “in_vhdl” e “pre_vhdl”. Todo valor calculado no processamento do VHDL (traduzido em código FIACRE) é guardado no *record* “out_vhdl”. As linhas 34-36 representam a tradução do circuito combinacional que foi omitido. A regra 9 de tradução define que o modelo FIACRE deve ser uma composição entre os processos “input_generator” e “delta_cycle”, sendo que eles compartilham um *channel* de sincronização (“input_list”) e uma variável (“output_list”). O Código 4.7

representa a aplicação da regra 9 ao controlador de memória.

```

1 component mem_ctrl is
    var output_list : output_list := INIT_OUTPUT_LIST
    port input_list : input_list in [0,0]
    par * in
5     delta_cycle [input_list] (&output_list)
      || input_generator [input_list] (&output_list)
    end

```

Código 4.7 – Declaração de componente para o controlador de memória

4.5 CONCLUSÃO

Nesse capítulo foram apresentadas regras de tradução que possibilitam a criação do modelo FIACRE, que serve de entrada para a cadeia de verificação, a partir do código VHDL sintetizável a ser analisado. A estrutura geral do modelo FIACRE contém dois processos, um para gerar as entradas (**input_generator**) e outro para executar o código VHDL (**delta_cycle**). Nesse capítulo foram apresentadas regras para a criação do **input_generator**, para geração de qualquer valor para todas as entradas. Porém em verificação formal pode ser importante que as entradas sejam geradas com restrições para evitar situações impossíveis ou explosão de estados. Nesse trabalho, as restrições para as entradas serão modeladas a partir de observadores OVL que serão traduzidos para processos **input_generator** adequados, sendo que as regras de tradução serão apresentadas no próximo capítulo.

Além do modelo FIACRE, o processo de *model checking* através da cadeia de verificação necessita que as propriedades sejam representadas por meio de fórmulas LTL. Na metodologia de desenvolvimento proposta, o usuário descreve as propriedades por meio de padrões da biblioteca OVL, sendo as propriedades traduzidas para LTL. As regras de tradução OVL-LTL também serão apresentadas no próximo capítulo.

5 MODELO DE VERIFICAÇÃO

Neste capítulo é apresentado o modelo de verificação das propriedades. Entende-se por modelo de verificação a definição das propriedades que serão verificadas e também as restrições quanto à geração do espaço de estados para a verificação.

Como definido na metodologia apresentada no Capítulo 3, as propriedades devem ser especificadas a partir de padrões de propriedade derivados da biblioteca OVL e convertidos para fórmulas LTL, que é o modelo formal de especificação de propriedades adotado pela ferramenta de *model checking*. Devido à dificuldade de se representar comportamento de *clock* em LTL convencional, a lógica @LTL é utilizada como um passo intermediário da tradução. A lógica temporal @LTL possui um operador de *clock* adicional, o que facilita a tradução do OVL, e sua tradução para LTL é automática com o uso de algumas regras.

A redução do espaço de estados gerados é realizada ao restringir o comportamento das entradas aos estímulos de interesse para a propriedade a ser verificada. Nesse trabalho é proposta a utilização de padrões da biblioteca OVL também para definição de restrições nas entradas do modelo FIACRE, sendo os padrões OVL automaticamente traduzidos para um processo **input_generator** adequado.

O capítulo inicia fazendo uma apresentação da lógica @LTL, para em seguida apresentar a biblioteca OVL, mostrando como ela é usada normalmente e como será aplicada na metodologia. Posteriormente são apresentados os princípios de tradução do OVL para lógica LTL e as regras de tradução para gerar o processo **input_generator** do modelo FIACRE. O capítulo finaliza com um comentário sobre os observadores traduzidos nesse trabalho.

5.1 CLOCKED LTL (@LTL)

Circuitos digitais são, em geral, projetos síncronos que por sua vez são baseados na noção de um tempo discreto em que um *flip-flop* leva o sistema do estado atual ao próximo. Um *flip-flop* é um elemento de memória que altera o valor de suas saídas em função das entradas, mas somente quando a entrada de *clock* está ativa, ou seja, a cada ciclo de *clock* do *hardware*. Normalmente quando se define propriedades a serem verificadas em circuitos síncronos, o que é levado em consideração

é a evolução do sistema em relação aos ciclos de *clock* do *hardware*, e a fórmula LTL:

$$\Box(p \rightarrow \bigcirc q)$$

pode ser interpretada por exemplo como “sempre que p for verdadeiro, q é verdadeiro no próximo ciclo de *clock*”. O mapeamento entre modelo de estados da lógica temporal LTL para o modelo de ciclos de *clock* não é automático e, portanto, se a fórmula acima fosse utilizada para verificar uma propriedade, os resultados provavelmente não seriam os esperados. Considerando o evento “clk” como a ocorrência de um *tick* no sinal de *clock* (quando *clock* está ativo, em geral em circuito síncronos pode ser a borda de subida ou descida), a fórmula deveria ser reescrita em LTL para o formato:

$$\Box((clk \wedge p) \rightarrow \bigcirc[\neg clk W(clk \wedge q)])$$

para ser interpretada como desejado. Nesse caso é utilizado o operador **weak until (W)** pois é considerada possível a existência de caminhos de execução onde eventos de *clock* parem de ocorrer (a expressão à direita do operador **W** nunca será verdadeira, mesmo com **q** verdadeiro), do contrário deveria ser utilizado **strong until (U)**, o qual exige que a expressão à direita do operador seja verdadeira em algum momento para a propriedade ser satisfeita. A aplicação dessa propriedade está ilustrada na Figura 15, considerando a borda de subida do sinal **clk**, como evento de *clock*. Na figura, durante a segunda borda de subida do sinal **clk** a proposição **p** é verdadeira portanto, para a propriedade ser satisfeita, é necessário que na próxima borda de subida o sinal **q** deva ser verdadeiro. Na terceira borda de subida do **clk** a proposição **q** é verdadeira, satisfazendo a propriedade. Na sexta borda de subida do **clk**, a proposição **p** é amostrada como verdadeira novamente. No entanto, na borda seguinte o sinal **q** é falso e a propriedade não é satisfeita.

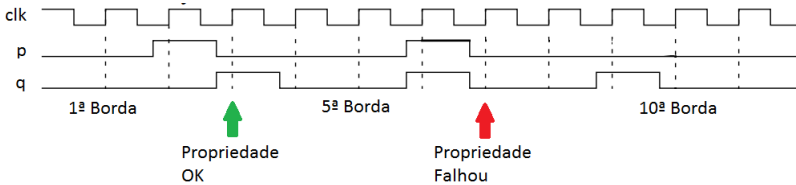


Figura 15 – Exemplo de propriedade com *clock*

Com um pequeno exemplo é possível perceber a dificuldade de se representar propriedades que se baseiam em ciclos de *clock*, utilizando apenas LTL. Com o objetivo de tornar o LTL mais adaptado à representação de propriedades de *hardware*, Eisner et al. (2003) propuseram

estender o LTL tradicional com a adição de um operador de *clock* @, a lógica @LTL (*clocked LTL*). Esse operador faz uma projeção do espaço de estados do sistema, englobando somente os estados em que ocorre o evento de *clock*. Eisner também definiu o operador proposicional forte $p!$, que só pode ser verdadeiro em um caminho não vazio, enquanto que uma proposição convencional (chamada em @LTL de proposição fraca) pode ser satisfeita em um caminho vazio. O operador @, ao fazer a projeção dos estados, pode criar um caminho vazio se o evento de *clock* nunca ocorrer e, por isso, é preciso criar operadores (proposição forte e fraca) para definir se uma fórmula é verdadeira nesse caso, o qual não existe em LTL convencional (sempre existe um estado atual no LTL convencional).

Outro problema introduzido pela projeção do operador @ é a possibilidade de transformar caminhos infinitos em finitos, o que traz problemas para o operador LTL **Next** (\bigcirc) o qual necessita da existência de um próximo estado para ter significado. Para resolver esse problema foi introduzido o operador **Next forte** ($\bigcirc!$), que somente pode ser verdadeiro em um caminho em que existe o próximo estado, e o operador **Next fraco** (\bigcirc) que é verdadeiro em caso de não existir um próximo estado. A sintaxe do @LTL é definida como:

- Se p é uma proposição atômica, então p e $p!$ são fórmulas @LTL
- Se clk é uma expressão booleana e f , f_1 e f_2 são fórmulas @LTL, então as seguintes expressões são fórmulas @LTL: $\neg f$, $f_1 \wedge f_2$, $\bigcirc!f$, $[f_1 U f_2]$, $f@clk$.

O LTL convencional é um subconjunto do @LTL, consistindo das fórmulas sem o operador de *clock* @ e sem sub-fórmulas na forma $p!$, para alguma proposição atômica p .

5.1.1 Tradução @LTL-LTL

O operador @ não adiciona nenhum poder de expressão adicional ao LTL tradicional, tanto que qualquer expressão que utilize esse operador pode ser traduzida para o LTL convencional. O valor verdade de qualquer fórmula @LTL sob contexto clk (evento de *clock* sendo representado por clk) é o mesmo que de uma fórmula LTL $f' = \tau^{clk}(f)$, onde τ^{clk} é definido como (considerando p uma proposição):

- $\tau^{clk}(p!) = [\neg clk U (clk \wedge p)]$
- $\tau^{clk}(\neg f) = \neg \tau^{clk}(f)$

- $\tau^{clk}(f_1 \wedge f_2) = \tau^{clk}(f_1) \wedge \tau^{clk}(f_2)$
- $\tau^{clk}(\bigcirc! f) = [\neg clk U (clk \wedge \bigcirc! [\neg clk U (clk \wedge \tau^{clk}(f))])]$
- $\tau^{clk}([f_1 U f_2]) = [(clk \rightarrow \tau^{clk}(f_1)) U (clk \wedge \tau^{clk}(f_2))]$
- $\tau^{clk}(f@clk_1) = \tau^{clk_1}(f)$

A partir dessas regras, também é possível ampliar a definição de τ^{clk} para outros operadores mais complexos como o **always** (\square) e **weak until** (W):

- $\tau^{clk}([\square f]) = \square[clk \rightarrow \tau^{clk}(f)]$
- $\tau^{clk}([f_1 W f_2]) = \neg[(clk \rightarrow \tau^{clk}(\neg f_2)) U (clk \wedge \tau^{clk}(\neg(f_1 \vee f_2)))]$

Além das fórmulas citadas, o trabalho de Eisner ainda apresenta uma definição de $\tau^{clk}(p)$, porém como nesse trabalho as traduções OVL-LTL serão realizadas considerando um *clock* que está sempre alternando seu valor, somente a proposição forte $p!$ será adotada. Nos princípios de tradução apresentados posteriormente nesse Capítulo, a expressão “!” não é utilizada nas fórmulas por simplificação, já que todos os operadores usados são fortes.

5.2 OVL

Bening e Foster (2000) descrevem que, no final da década de 90, eles tinham grande dificuldade de lidar no local de trabalho com as várias ferramentas de verificação formal para HDLs que estavam aparecendo na época. Cada ferramenta tinha uma linguagem própria para a descrição de propriedades e, por isso, eles tinham muito retrabalho ao ficarem rescrevendo os observadores utilizados para a verificação dos circuitos desenvolvidos. A partir desse problema que tiveram a ideia de criar uma biblioteca de observadores/monitores para HDL que escondessem os detalhes de implementação. Logo eles perceberam que os desenvolvedores HDL preferiam utilizar a biblioteca em vez de ter que aprender novas linguagens, além dessa biblioteca acelerar o processo de verificação ao oferecer padrões de propriedades mais frequentemente utilizados. A partir dessa biblioteca surgiu a ideia da criação do OVL.

O OVL (*Open Verification Library*) (OVL, 2014) é uma biblioteca de módulos observadores que são instanciados no projeto, de preferência em *testbenches*, com o propósito de analisar propriedades específicas do circuito projetado por meio de simulação ou verificação formal. O OVL

oferece um modelo de interface para validação de propriedades que é independente de linguagens e ferramentas. Atualmente, os monitores OVL têm sua estrutura interna implementada em cinco linguagens diferentes (PSL para VHDL, PSL para Verilog, SVA, VHDL e System Verilog), o que permite que sejam compiláveis e utilizáveis por uma grande quantidade de ferramentas.

A biblioteca OVL disponibiliza apenas 10 tipos de monitores implementados em VHDL puro. Em PSL e Verilog são 33 tipos de monitores diferentes enquanto em SVA estão disponíveis todos os 55 monitores da biblioteca.

Um monitor OVL funciona como uma instância de um circuito, possuindo um conjunto de parâmetros de configuração em tempo de compilação (chamado *generics* quando usado com VHDL) e um conjunto de portas de entrada e saída. O circuito interno ao observador OVL é responsável pela análise de uma ou mais propriedades.

As portas comuns a todos observadores da biblioteca são:

- **clock:** Evento de *clock* para o observador.
- **reset:** Sinal de *reset* síncrono para o observador. Quando ativo, esse sinal cancela/reinicia a verificação da propriedade.
- **enable:** Habilita/desabilita a verificação da propriedade. Dependendo da configuração do atributo **gating type**, essa entrada pode ser utilizada para somente pausar a verificação em vez de desabilitar.
- **fire:** Porta de saída que sinaliza quando a propriedade foi violada.

Além dessas portas, cada observador pode ter outras que estejam relacionadas à propriedade que ele analisa. Além da conexão das portas, é necessário definir os parâmetros de configuração *generics*. sendo que os parâmetros comuns a todas as propriedades são:

- **severity_level:** Define a severidade da falha. Dependendo da configuração, uma falha pode apenas gerar um aviso ou até mesmo encerrar a simulação.
- **property_type:** Define o tipo de propriedade. Pode ser do tipo *assert*, definindo que é uma propriedade a ser verificada, ou pode ser do tipo *assume*, definindo quais sinais de estímulo são válidos.
- **msg:** Define a mensagem mostrada no simulador quando a propriedade falhar.

- **coverage_level:** Define quais estatísticas relacionadas a propriedade serão informadas ao simulador. A estatística pode ser a contagem da quantidade de vezes que a propriedade falha (em simulação) ou se algumas condições de contorno acontecem.
- **clock_edge:** Define qual a borda ativa (descida ou subida) do sinal de *clock*.
- **reset_polarity:** Define se o sinal de *reset* será ativo baixo ou alto.
- **gating_type:** Define como será o funcionamento da porta *enable*. Ela pode ser configurada para ser ignorada, desabilitar a propriedade quando tiver em falso ou pausar a verificação quando em falso.

Além desses parâmetros, cada observador pode ter outros que estejam relacionados a propriedade que ele analisa.

5.2.1 Aplicação convencional do OVL

Ao utilizar um observador OVL para análise de uma propriedade, é necessário fazer uma instância do mesmo no *testbench* e fazer a conexão dos sinais com as portas do observador, sendo que os sinais estão relacionados com a propriedade.

Um *testbench* VHDL em que se faz o uso de OVL pode ser dividido em três partes: a declaração dos observadores OVL, código gerador de estímulos para o circuito a ser simulado, instância do circuito a ser simulado/analísado. O Código 5.1 mostra um exemplo de *testbench* VHDL onde é instanciado o monitor **ovl_always**. Esse monitor verifica se a expressão conectada a porta *test_expr* é sempre verdadeira. No exemplo abaixo é verificado se as saídas **o1** e **o2** do DUT nunca são verdadeiras ao mesmo tempo.

```

1 library IEEE;
  use IEEE.std_logic_1164.all;

  library accellera_ovl_vhdl;
5 use accellera_ovl_vhdl.std_ovl_components.all;

  entity ovl_example_tb is
  end entity ovl_example_tb;

10 architecture RTL of ovl_example_tb is
    signal clk : std_logic := '0';
    signal rst_n: std_logic := '0';

```

```

    signal i1, i2, o1, o2 : std_logic;
    signal nand_o1_o2 : std_logic;
15 begin

    OVLINST: ovl_always port map
        port map (
            clock => clk,
20         reset => rst_n,
            enable => '1',
            test_expr => nand_o1_o2;
            fire => open);

25     nand_o1_o2 <= not (o1 and o2);
        clk <= not clk after 10 ns;
        rst_n <= '0', '1' after 200 ns;

    DUT: entity work.dut_example
30     port map (
        i1 => i1,
        i2 => i2,
        o1 => o1,
        o2 => o2);

35     stimulus: process
        begin
            -- stimulus generation
        end process stimulus;
40 end architecture RTL;

```

Código 5.1 – Exemplo de uso de observador OVL

As linhas 17-25 são a parte do *testbench* em que o observador OVL é instanciado e as conexões de sinais as suas portas de entrada são realizadas. As linhas 29-34 formam a instância do circuito a ser simulado e analisado. As linhas 36-39 contêm o código do estímulo para as entradas do DUT, sendo que o código foi omitido. As linhas 26-27 também fazem parte do estímulo, definindo sinais de *reset* e *clock*.

O observador OVL utilizado no exemplo do Código 5.1 possui as seguintes portas: **clock**, **reset**, **enable**, **test_expr** e **fire**. Somente a porta **test_expr** é específica do **ovl_always**, enquanto as demais são comuns a todos os observadores.

Quando utilizado em simulação, o observador OVL faz uma análise da propriedade restrita aos estímulos de entrada gerados no *testbench*. Quando o OVL é utilizado para verificação formal (depende do suporte da ferramenta), a ferramenta estende a análise do observador para todos os casos possíveis de estímulos de entrada.

O parâmetro **property_type** tem uma importância especial porque define qual a finalidade em que será utilizado o observador. Quando esse parâmetro é definido como *assert* (OVL_ASSERT), o observador é utilizado como uma propriedade a ser testada. Quando o parâmetro é definido como *assume* (OVL_ASSUME), o observador é utilizado para

definir quais estímulos são válidos no caso de uma verificação formal (para simulação não tem uso essa configuração).

No exemplo do Código 5.1, o observador foi usado em sua configuração padrão que é *assert*. Se o usuário desejasse fazer uma verificação formal do circuito, porém assumindo que as entradas *i1* e *i2* nunca pudessem ser verdadeiras ao mesmo tempo, o usuário poderia utilizar uma instância do observador **ovl_always**, configurado como *assume* e com a porta **test_expr** do observador conectada a um sinal que fizesse a expressão $\neg(i1 \wedge i2)$. A instância do observador configurado como *assume* indicaria a ferramenta de verificação que todos os estados em que as entradas não respeitem aquela regra devam ser descartados para verificação das propriedades.

5.2.2 Aplicação proposta do OVL

Na metodologia proposta nesse trabalho, os observadores OVL não são instanciados no *testbench*, mas sim traduzidos para fórmulas LTL ou modelos de processo **input_generator**, dependendo da configuração utilizada no parâmetro **property_type**. A Figura 16 faz uma comparação do uso do observador **ovl_always** entre o mostrado no exemplo do Código 5.1 e na metodologia proposta. Em ambos os casos o parâmetro **property_type** está configurado como *assert*.

Após a definição dos sinais conectados as portas e o valor dos parâmetros de configuração, no uso habitual isso é convertido em um bloco de código VHDL que instancia o observador conectando os sinais as portas e definindo os valores dos parâmetros *generics*. É importante lembrar que na utilização habitual, a instância do observador pode ser usada tanto para simulação quanto para a verificação, enquanto que na proposta o uso é somente para verificação, onde o observador é convertido em uma fórmula LTL, que será utilizada pela ferramenta de *model checking* junto com o modelo FIACRE do código VHDL para a verificação da propriedade (conforme descrito no Capítulo 3). Na Figura 16, a fórmula LTL gerada é $\Box(\text{clk} \rightarrow \neg(o1 \wedge o2))$.

No Código 5.1, se considerarmos que “i1” e “i2” são entradas do circuito a ser testado e que fosse necessário criar uma restrição na qual as entradas nunca fossem verdadeiras ao mesmo tempo, um observador **ovl_always** poderia ser utilizado em que *test_expr* = $\neg(i1 \wedge i2)$, e o parâmetro **property_type** fosse do tipo *assume*. A Figura 17 faz uma comparação do uso do observador como *assume* em uma utilização convencional e como ele é aplicado na metodologia.

OVL_ALWAYS

Portas	Expressões conectadas	Generics	Valor
clock	clk	property_type	OVL_ASSERT
reset	rst_n		
enable	true		
test_expr	not(o1 and o2)		
fire	-		

```

entity ovl_example_tb is
end entity ovl_example_tb;

architecture RTL of ovl_example_tb
-- signals declaration
begin
-- ovl instance
OVL_INST: ovl_always port map
generic map (
property_type => OVL_ASSERT);
port map (
clock => clk,
reset => rst_n,
enable => '1',
test_expr => nand_o1_o2;
fire => open);
nand_o1_o2 <= not (o1 and o2);

-- DUT instance
DUT: entity work.dut_example
port map (
i1 => i1,
i2 => i2,
o1 => o1,
o2 => o2);

stimulus: process
begin
-- stimulus generation
end process stimulus;
end architecture RTL;

```

$$LTL : \square(\text{clk} \rightarrow \neg(\text{o1} \wedge \text{o2}))$$

Uso Convencional

Proposta

Figura 16 – Comparação de uso do OVL como *assert* entre o convencional e a proposta

Na aplicação habitual do OVL, o uso é quase idêntico tanto para o *assert* quanto para o *assume*, somente diferenciando o valor do parâmetro **property_type** no momento da instância do observador.

como *assert* ou *assume*. Em seguida ele deve definir os valores dos parâmetros de configuração do observador. Para finalizar, o usuário deve definir as expressões envolvendo variáveis do circuito verificado (entradas, saídas ou sinais internos) que estarão associadas as portas. Após essas definições, o observador é traduzido automaticamente e o *model checking* é realizado através da cadeia FIACRE.

5.2.3 Restrições da tradução do OVL para a metodologia

Antes de apresentar os princípios de tradução é necessário fazer algumas observações sobre portas e parâmetros presentes nos observadores OVL que não serão representados.

A porta **enable** dos observadores será ignorada nas traduções porque quando a propriedade for utilizada (seja como *assert* ou *assume*) ela será sempre considerada como habilitada para o verificador. A porta **fire** não será representada também, porque a ferramenta de verificação gera os resultados em um arquivo indicando se a propriedade passou ou não passou. O sinal de **reset** não será representado porque o @LTL não possui esse conceito e porque o comportamento de *reset* é complicado de representar em lógica temporal. Como o *reset* de um circuito digital é uma situação de exceção que força o sistema a retornar ao seu estado inicial, não existem muitas perdas para a verificação formal se ele não for considerado. É importante observar que o modelo FIACRE (gerado para o código VHDL a ser verificado) deve conter uma restrição na geração de entradas (processo **input_generator**) que mantenha a entrada de *reset* em valor constante que habilita o circuito, senão as propriedades verificadas falharão por não levar em consideração o *reset* do circuito.

Quanto à tradução dos parâmetros dos observadores, os *generics* **severity level**, **msg** e **coverage level** não serão representados na tradução por apenas definirem como o simulador deverá se comportar em relação ao observador. Os parâmetros **gating type** e **reset polarity** também serão ignorados porque as portas a que estão relacionados não serão representadas. O parâmetro **property type** é traduzido como a possibilidade da propriedade ser usada como *assert*, indicando que deve ser verificada e por isso traduzida para fórmula LTL, ou *assume*, indicando que é uma restrição para as entradas e por isso deve ser traduzida para código adicional no **input_generator**.

A borda ativa utilizada para o sinal de *clock* nas traduções foi a de subida: a mais usada nas aplicações da empresa e também consi-

derada como boa prática no projeto de circuitos digitais. A utilização da borda de descida não traz alterações na tradução para LTL, mas na tradução de restrições para entradas implicaria uma troca do estado em que os valores são gerados no processo **input_generator** (as entradas seriam geradas na borda de descida). A ferramenta que fará a tradução automática deve ter uma opção para o usuário escolher qual borda utilizar (subida ou descida).

Mais detalhes sobre os princípios de tradução do OVL para as aplicações na metodologia serão apresentadas a seguir.

5.3 TRADUÇÃO OVL-LTL

A metodologia proposta nesse trabalho tem como um dos principais objetivos ser acessível ao desenvolvedor VHDL, fornecendo uma interface que seja familiar a eles. A definição de propriedades a serem verificadas e a descrição delas em um modelo formal são etapas fundamentais do processo de *model checking*. Essas etapas costumam ser as mais complicadas para usuários não acostumados com verificação formal, visto que o modelo formal mais utilizado são lógicas temporais, linguagens com que pouquíssimos desenvolvedores VHDL já tiveram contato.

A ferramenta de *model checking* utilizada na cadeia de verificação necessita que as propriedades sejam descritas em lógica LTL tradicional. Porém, forçar o usuário da metodologia a utilizar LTL não é uma alternativa atraente visto que, além dos desenvolvedores nunca terem usado a linguagem, o LTL não é adequado para representar propriedades com *clock*. Como mencionado anteriormente, a maior parte dos circuitos digitais são síncronos e, portanto, as propriedades devem, na maior parte dos casos, descrever comportamentos que evoluem em relação ao sinal de *clock*.

Alguns trabalhos já abordaram o problema de tornar a descrição de propriedades algo mais fácil. Dwyer (DWYER; AVRUNIN; CORBETT, 1999) em especial, mostrou em seu trabalho que a maior parte das propriedades especificadas em verificações de estados finitos são pequenas variações de um conjunto de padrões. Ele concluiu que, em geral, não é necessário que o usuário tenha conhecimento das particularidades de uma determina linguagem de descrição de propriedades, mas apenas conhecer como os padrões são representados. Em seu trabalho, Dwyer também apresentou uma lista de padrões de propriedade mais utilizados, a descrição deles em lógica LTL e CTL, e estatísticas quanto ao uso

desses padrões baseado em observações realizadas em vários trabalhos de verificação.

Abid (ABID, 2012) se baseou no trabalho de Dwyer para propor, entre outros tópicos abordados, padrões de propriedade que são descritos utilizando termos em linguagem natural como *absence*, *within*, *present*, *lasting*, *interval*, além de descrever o comportamento formal de cada padrão e apresentar a tradução deles para observadores na linguagem FIACRE. O trabalho de Abid foi incorporado ao compilador do FIACRE, permitindo que na verificação de propriedades para modelos descritos com a linguagem, o usuário não necessite mais utilizar a lógica temporal LTL (suportada pelo compilador). O compilador faz a tradução do modelo em FIACRE para o formalismo matemático utilizado pelas ferramentas de verificação.

A linguagem FIACRE serve como *front-end* para a cadeia de verificação utilizada na metodologia proposta. Seria natural o uso dos padrões de Abid para auxiliar na descrição de propriedades, já que o trabalho envolve a tradução VHDL-FIACRE. No entanto, eles não são muito aplicáveis a circuitos digitais por não considerarem a evolução do comportamento em relação ao sinal de *clock*.

Como mencionado anteriormente, a biblioteca de observadores OVL foi concebida com base em uma pesquisa em que se foram verificados os padrões de propriedade mais comuns de serem utilizados em circuitos digitais, um trabalho semelhante ao realizado por Dwyer. A proposta desse trabalho é fazer o uso dessa biblioteca para auxiliar o usuário na descrição de propriedades, onde estas seriam especificadas por meio dos observadores OVL e traduzidas automaticamente para LTL (utilizada pela ferramenta de verificação) evitando, desse modo, que o desenvolvedor seja forçado a utilizar lógicas temporais.

Para auxiliar na tradução OVL-LTL, a lógica @LTL é utilizada como passo intermediário pois seu operador de *clock* permite uma tradução mais simples das propriedades OVL. A tradução do @LTL para LTL tradicional é simples, com o uso recursivo das regras apresentadas na Seção 5.1.

Uma ferramenta em que o usuário possa gerar propriedades com base na biblioteca OVL, e essas propriedades serem traduzidas automaticamente para o LTL, é bastante útil para evitar erros na formalização das propriedades, o que invalidaria o processo de verificação. Essa ferramenta não necessariamente precisa conter somente observadores OVL pois, em trabalhos futuros, novos padrões que sejam considerados úteis podem ser adicionados. A seguir serão apresentados os princípios de tradução dos observadores **ovl_always**, **ovl_window** e **ovl_next**. Os

outros observadores traduzidos estão disponíveis no Apêndice B.

5.3.1 ovl_always

O observador *ovl_always* verifica uma expressão de bit único *test_expr* a cada borda ativa de *clock*. Se *test_expr* não é verdadeiro, a propriedade é considerada como violada. Além das portas comuns a todos os *assertions*, esse módulo possui adicionalmente a porta **test expr**, cuja expressão deve ser verdadeira a cada borda ativa do *clock*.

Quando esse observador OVL é utilizado como propriedade para verificação, ele pode ser traduzido em @LTL e LTL com as seguintes expressões:

- @LTL : $\Box(\text{test_expr})@clock$
- LTL : $\Box(\text{clock} \rightarrow \text{test_expr})$

5.3.2 ovl_window

O observador *ovl_window* examina a expressão **start event** a cada borda ativa do *clock* para determinar se deve abrir uma janela de eventos no próximo ciclo. Se **start event** é amostrado como verdadeiro, nos ciclos seguintes o observador passa a acompanhar os valores de **end event** e **test expr**. Caso **test expr** seja falso, a propriedade falhou. Se a expressão **end event** for verdadeira, o observador fecha a janela de eventos e retorna a monitorar a **start event** no próximo ciclo de *clock*. Esse observador pode ser útil, por exemplo, para verificar condições de sincronismo do circuito que necessitam que um dado permaneça estável após um evento de disparo.

Além das portas comuns, o observador possui as seguintes portas adicionais:

- **start_event**: Expressão que abre a janela de eventos.
- **test_expr**: Expressão que deve ser verdadeira na janela de eventos.
- **end_event**: Expressão que encerra a janela de evento.

O observador OVL pode ser traduzido em @LTL e LTL com as seguintes expressões:

- @LTL : $\Box(((\neg \text{start_event}) \wedge \bigcirc (\text{start_event} \wedge \neg \text{end_event})) \rightarrow \bigcirc \bigcirc (\text{test_expr} \text{ W } (\text{test_expr} \wedge \text{end_event})))@clock$

- $LTL : \Box(\text{clock} \rightarrow (((\neg \text{start_event}) \wedge \bigcirc (\neg \text{clock} U (\text{clock} \wedge \text{start_event} \wedge \neg \text{end_event})))) \rightarrow \bigcirc (\neg \text{clock} U (\text{clock} \wedge \bigcirc (\neg \text{clock} U (\text{clock} \wedge (\neg ((\text{clock} \rightarrow \neg \text{end_event}) U (\text{clock} \wedge \neg (\text{test_expr} \vee (\text{test_expr} \wedge \text{end_event})))))))))))))$

5.3.3 ovl_next

O observador *ovl_next* verifica a expressão **start event** a cada borda ativa do *clock*. Se **start event** é verdadeiro, uma verificação é iniciada. A verificação aguarda **num cks** (parâmetro *generics*) ciclos de *clock* para então examinar o valor de **test expr** e, caso a expressão seja falsa, a propriedade foi violada, do contrário, ela é válida. Esse observador é útil para verificar situações de causalidade no circuito, onde a ocorrência de um evento leva a alguma resposta após um tempo fixo.

Além das portas comuns, esse observador possui as seguintes portas adicionais:

- **start_event**: Expressão que identifica (juntamente com **num cks**) quando verificar **test_expr**.
- **test_expr**: Expressão que deve ser verdadeira **num cks** ciclos de *clock* depois de **start event** ser verdadeiro.

O observador OVL pode ser traduzido em @LTL e LTL com as seguintes expressões (**num cks=2**):

- $@LTL : \Box(\text{start_event} \rightarrow \bigcirc \bigcirc (\text{test_expr}))@clock$
- $LTL : \Box(\text{clock} \rightarrow (\text{start_event} \rightarrow \bigcirc (\neg \text{clock} U (\text{clock} \wedge \bigcirc (\neg \text{clock} U (\text{clock} \wedge \text{test_expr}))))))$

Esse observador pode ser configurado para fazer verificações sobrepostas ou não dependendo do valor do parâmetro *generic check overlapping*. Quando esse parâmetro possui valor 0, a verificação sobreposta não é permitida, o que significa uma vez que **start expr** se tornou verdadeiro, no próximo ciclo ele deve ser falso e assim permanecer até que **test expr** seja verdadeiro. Para fazer essa verificação, a seguinte propriedade LTL é necessária (considerando **num cks=3**):

- $@LTL : \Box(\text{start_event} \wedge \bigcirc (\neg \text{start_event} \wedge \bigcirc (\neg \text{start_event})))@clock$
- $LTL : \Box(\text{clock} \rightarrow (\text{start_event} \wedge \bigcirc (\neg \text{clock} U (\text{clock} \wedge (\neg \text{start_event} \wedge \bigcirc (\neg \text{clock} U (\text{clock} \wedge (\neg \text{start_event}))))))))$

Outra configuração possível, quando o parâmetro *generic check missing start* possui valor 1, é a verificação se a expressão *test expr* se torna verdadeira sem **start event** ter sido verdadeiro anteriormente. Para essa checagem, a seguinte expressão LTL é necessária (considerando **num cks=2**):

- $@LTL : \Box(\neg(\neg start_event \rightarrow \bigcirc \bigcirc test_expr))@clock$
- $LTL : \Box(clock \rightarrow \neg(\neg start_event \rightarrow \bigcirc(\neg clock U (clock \wedge \bigcirc(\neg clock U (clock \wedge test_expr)))))$

Generalizando as três propriedades para **num cks=n**, temos as seguintes expressões em @LTL:

1. $@LTL : \Box(start_event \rightarrow \bigcirc^n(test_expr))@clock$
2. $@LTL : \Box(start_event \wedge [\bigcirc(\neg start_event \wedge)^{n-2} \bigcirc(\neg start_event)...])@clock$
3. $@LTL : \Box(\neg(\neg start_event \rightarrow \bigcirc^n test_expr))@clock$

5.4 MODELO INPUT_GENERATOR COM RESTRIÇÕES OVL

Em verificação formal, procura-se garantir que as propriedades são satisfeitas em todas as situações possíveis do sistema e, por isso, costuma-se utilizar um modelo de entradas que gere qualquer valor a qualquer momento. Esse tipo de modelo de entradas muitas vezes pode levar a um espaço de estados muito grande para o sistema, fazendo com que as ferramentas computacionais não sejam capazes de processar, levando a uma situação conhecida de explosão de estados.

Para evitar a explosão de estados, é comum em *model checking* construir modelos de geração de estímulos de entrada que sejam mais restritos, porém sem prejudicar a verificação das propriedades. As restrições aplicadas costumam eliminar comportamentos de entrada impossíveis para o sistema, ou então manter apenas os estímulos que são relevantes para a propriedade que está sendo verificada.

No Capítulo 4 foram apresentadas as regras de tradução VHDL-FIACRE, sendo que o modelo FIACRE gerado é formado por dois processos: um para execução do código (**delta_cycle**) e outro para geração das entradas (**input_generator**). A construção do **input_generator** para gerar qualquer valor é bastante simples e automática conforme descrito no Capítulo anterior, porém é desejável que seja possível gerar automaticamente um modelo com restrições para as entradas e evitar a explosão de estados.

O objetivo específico da metodologia é manter as interfaces com o processo de verificação em linguagem de usuário, logo, exigir que o usuário descreva o modelo de entradas em FIACRE está descartado. Uma solução possível seria permitir que o usuário descreva o comportamento desejado por meio de máquinas de estado ou até mesmo código VHDL. Porém, a tradução seria bastante complicada e a possibilidade de usuário cometer erros na definição das restrições se tornaria muito grande.

A solução proposta é utilizar a mesma ideia da definição de propriedades para verificação, ao usar padrões de restrição de entradas derivados da biblioteca OVL. A biblioteca OVL já prevê o uso dos observadores para definição de estímulos válidos através da configuração do parâmetro **property_type** como *assume*. Na metodologia são traduzidos alguns observadores OVL para serem usados também como *assume*, definindo restrições para as entradas que serão traduzidas automaticamente para um processo **input_generator** adequado.

O uso de observadores OVL leva a algumas limitações no formato do **input_generator** e, por isso, na metodologia são propostos dois tipos de processo (não usados em conjunto): um para gerar valores sem OVL (Capítulo 4) e outro com restrições OVL.

5.4.1 Estrutura do **input_generator**

Quando o desenvolvedor deseja criar um modelo de entradas com restrições geradas por propriedades OVL configuradas como *assume*, o processo **input_generator** deve ter o formato mostrado no Código 5.2 (considerando que o código VHDL de origem possui “n” entradas). Os observadores OVL, em sua grande maioria, descrevem propriedades sobre sinais que evoluem em relação a um sinal de *clock* e, por isso, o modelo **input_generator** gerado quando se usa o OVL necessita que seja definido uma das entradas para realizar esse papel. Outra restrição considerada é que o circuito é síncrono, sendo que as entradas possuem seus valores alterados na borda de subida do *clock*. No exemplo mostrado é considerado a entrada “input1” como sinal de *clock*.

```

1  process input_generator [generated_inputs : out input_list]
   (&output_list : read output_list) is
   states clk_false, clk_true
   var input1 : bool := false,
5     ...
     inputn : fiacre_type := init_value,
     //////////
     //ovl restrictions auxiliary variables
     ...

```

```

10      ///////
      init to clk_false
      from clk_false
        input1 := false;
        generated_inputs!input1, ..., inputn;
15      to clk_true
      from clk_true
        input1 := true;
        //ovl restrictions auxiliary code
        ...
20      ///////////////////////////////////////////////////////////////////
      input2, ..., inputn := any where ovl_restriction1
                                          and
                                          ovl_restriction2
                                          ...
25                                          and
                                          ovl_restrictionn;
      generated_inputs!input1, ..., inputn;
      to clk_false

```

Código 5.2 – Estrutura do **input_generator** com restrições OVL

O modelo do **input_generator** é um processo com dois estados que ficam se alternando: **clk_false** e **clk_true**. No estado **clk_false** a entrada de *clock* deve ser falsa, enquanto que no outro estado deve ser verdadeira. O fato do sinal de *clock* estar sempre alternando seu valor, além de reduzir o espaço de estados, simplifica a definição de propriedades pois as mesmas não precisam levar em consideração caminhos de execução em que o *clock* não mude seu valor.

As linhas 4-6 do Código 5.2 contêm a declaração de n variáveis FIACRE, uma para cada entrada declarada no bloco **Entity** do código VHDL a ser verificado. As variáveis devem ter o tipo FIACRE correspondente ao tipo VHDL da entrada que está representando, e mesmo nome. As linhas 7-10 representam a declaração de variáveis auxiliares usadas para a tradução das propriedades OVL utilizadas para restringir as entradas. Cada propriedade OVL precisa de um conjunto específico de variáveis auxiliares que serão detalhadas nas próximas seções. As linhas 12-15 formam o estado **clk_false**, que contém apenas a associação de valor falso para a *clock* e a sincronização para passagem de valores para o processo **delta_cycle**.

O estado **clk_true** (linhas 16-28) é onde o valor das entradas que não são *clock* possuem seus valores alterados. Além das variáveis auxiliares, algumas propriedades OVL necessitam de um código adicional que é representado nas linhas 18-20 (mais detalhes quando for apresentado a tradução do OVL). Esse código adicional sempre é posicionado antes da expressão “any” que atribui qualquer valor para as entradas (linha 21). A diferença fundamental da expressão “any” nesse caso, é a presença da expressão “where” que define condições as quais os

valores gerados devem satisfazer para que sejam considerados válidos. Na expressão “where” se encontra todas as restrições provenientes das propriedades OVL utilizadas como *assume*. Após a geração de valores para as entradas, elas são repassadas para o **delta_cycle** e o processo retorna ao estado **clk_true**.

As restrições de valores constantes para uma determinada entrada ou intervalo de valores também são possíveis nesse tipo de modelo de entrada, sendo realizadas da mesma maneira que no modelo sem OVL.

A seguir são apresentados os princípios de tradução dos observadores **ovl_always**, **ovl_window** e **ovl_next**. Os outros observadores traduzidos estão disponíveis no Apêndice B.

5.4.2 ovl_always

Quando esse observador OVL é utilizado como restrição para entradas, o processo **input_generator** tem o seguinte formato:

```

1  process input_generator [generated_inputs : out input_list]
   (&output_list : read output_list) is
   states clk_false, clk_true
   //variables declaration
5  ...
   init to clk_false
   from clk_false
     clock := false
   //clk_false code
10 from clk_true
     clock := true;
     input2, ..., inputn := any where test_expr;
     generated_inputs!input1, ..., inputn;
     to clk_false

```

Somente a expressão *test_expr* é adicionada a cláusula *where*.

5.4.3 ovl_window

Quando esse observador OVL é utilizado como restrição para entradas, o processo **input_generator** tem o seguinte formato:

```

1  process input_generator [generated_inputs : out input_list]
   (&output_list : read output_list) is
   states clk_false, clk_true
   //variables declaration
5  var ovl_window_open : bool := false,
     pre_ovl_window : bool := false,
     ...
   init to clk_false

```

```

10   from clk_false
      clock := false
      //clk_false code
  from clk_true
      clock := true;
      //ovl_window code
15   if end_event then
      ovl_window_open := false
      else
        if (not pre_ovl_window) and start_event then
          ovl_window_open := true
20     end if;
      end if;
      pre_ovl_window := start_event;
      //////////////////////////////////////
      input2, ..., inputn := any where
25   (not ovl_window_open)
      or
      (ovl_window_open and test_expr);
      generated_inputs!input1, ..., inputn;
      to clk_false

```

Para utilizar o **ovl_window** como *assume*, além da adição da expressão mostrada nas linhas 25-27 à cláusula *where*, é necessário o código auxiliar mostrado nas linhas 15-22. Duas variáveis extras são criadas para auxiliar no processamento do observador, destacadas nas linhas 5-6.

5.4.4 ovl_next

Quando esse observador OVL é utilizado como restrição para entradas, o processo **input_generator** tem o seguinte formato:

```

1   process input_generator [generated_inputs : out input_list]
      (&output_list : read output_list) is
      states clk_false, clk_true
      //variables declaration
5   var ovl_next_fifo : queue 1 of nat := {||},
      ovl_next_cnt : nat := 0,
      ...
      init to clk_false
      from clk_false
10   clock := false
      //clk_false code
      from clk_true
      clock := true;
      //ovl_next code
15   if (ovl_next_cnt > 0) and (start_event = true) then
      ovl_next_fifo := enqueue (ovl_next_fifo, ovl_next_cnt)
      end if;
      if (ovl_next_cnt = num_cks) then
      if empty(ovl_next_fifo) then
20   ovl_next_cnt := 0
      else
      ovl_next_cnt := ovl_next_cnt - first(ovl_next_fifo) + 1;
      ovl_next_fifo := dequeue(ovl_next_fifo)

```



```

    end if
25  else
    if ovl_next_cnt > 0 then
        ovl_next_cnt := ovl_next_cnt + 1
    else
    30  if start_event = true then
        ovl_next_cnt := 1
        end if
    end if
end if;
////////////////////////////////////
35  input2, ..., inputn := any where
    (test_expr and (ovl_next_cnt = num_cks)) or
    (ovl_next_cnt <> num_cks);
    generated_inputs!input1, ..., inputn;
to clk_false

```

Para utilizar o **ovl_next** como *assume*, além da adição da expressão mostrada nas linhas 36-37 a cláusula *where*, é necessário o código auxiliar mostrado nas linhas 15-33. Se a opção **check missing start** for usada, a expressão $\neg(\text{test_expr} \wedge (\text{ovl_next_cnt} = 0))$ deve ser adicionada ao *where*. Quando o parâmetro **check overlapping** for 0, a expressão $\neg(\text{start_event} \wedge (\text{ovl_next_cnt} > 0))$ deve ser adicionada ao *where*.

5.5 OBSERVADORES TRADUZIDOS

A Tabela 2 faz um resumo de todos os observadores OVL traduzidos, indicando se teve tradução como *assert* ou *assume*, e quais são as portas e parâmetros usados. Os observadores traduzidos são os disponíveis em VHDL puro, o restante da biblioteca só está disponível em PSL, SVA ou Verilog e devem ser traduzidos posteriormente.

As traduções podem ser encontradas no Apêndice B.

5.6 CONCLUSÃO

Nesse capítulo foram definidas as regras de tradução de observadores OVL para fórmulas LTL e para modelos de processo **input.generator**. A biblioteca OVL não foi traduzida por inteiro, apenas os observadores descritos em VHDL. Em trabalhos futuros serão traduzidos outros padrões OVL (adição de padrões que não estejam na biblioteca mas que sejam relevante para as aplicações da empresa) para permitir que a metodologia seja aplicada a um número maior de circuitos e também ampliar a quantidade de padrões de propriedades possíveis de verificação.

Tabela 2 – Observadores OVL traduzidos

	Assert	Assume	Parâmetros	Portas
ovl_always	sim	sim	-	clock test_expr
ovl_cycle_sequence	sim	não	num_cks necessary_condition	clock event_sequence
ovl_implication	sim	sim	-	clock antecedent_expr consequent_expr
ovl_never	sim	sim	-	clock test_expr
ovl_next	sim	sim	num_cks check_overlapping check_missing_start	clock start_event test_expr
ovl_one_hot	sim	não	-	clock test_expr
ovl_range	sim	sim	min max	clock test_expr
ovl_zero_one_hot	sim	não	-	clock test_expr
ovl_width	sim	sim	min_cks max_cks	clock test_expr
ovl_window	sim	sim	-	clock start_event end_event test_expr
ovl_unchange	não	sim	num_cks action_on_new_start	clock start_event test_expr

Com a definição das regras para criação do modelo FIACRE e para obtenção das fórmulas LTL, a cadeia de verificação já pode ser utilizada. A seguir, serão apresentados alguns exemplos de aplicação da metodologia.

6 APLICAÇÃO DA METODOLOGIA PROPOSTA

A metodologia proposta foi aplicada a quatro exemplos, com o objetivo de fazer a validação do trabalho desenvolvido. A Tabela 3 faz um resumo das características dos exemplos. A tabela fornece as seguintes informações: quantidade de linhas no código VHDL, a quantidade de propriedades verificadas, a quantidade de restrições para as entradas no modelo FIACRE utilizando padrões OVL, o número de estados na estrutura de Kripke gerada para o modelo FIACRE, e o tempo médio que a ferramenta de verificação precisou para fazer o *model checking* das propriedades.

Tabela 3 – Resumo dos exemplos de aplicação

Exemplo	Linhas de Código	Propriedades Verificadas	Restrições OVL (entradas)	Estados	Tempo Verificação (médio)
Controlador de Memória	72	4	0	1745	1s
Controlador FIFO (256 posições)	90	2	0	1844234	10s
PTOV	140	5	3	12833	1s
Goertzel Control	400	5	3	383845	40s

O exemplo do controlador de memória é a verificação do código apresentado na seção 2.1.1, cujo modelo FIACRE foi descrito em detalhes na seção 4.4. Tanto o código VHDL quanto o modelo FIACRE estão presentes no Apêndice A.

O controlador FIFO, assim como controlador de memória, é um exemplo de código VHDL presente em Chu (2006). Esse código faz uma descrição de um circuito que controla o acesso à uma memória FIFO, recebendo como entradas as requisições de leitura e escrita. Ele possui como saídas os sinais que indicam se a memória está cheia ou vazia, e os ponteiros para leitura ou escrita na memória. Este exemplo foi importante para validar a metodologia proposta por ser um código mais complexo e que gera um espaço de estados maior, porém o procedimento de verificação foi semelhante ao do controlador de memória e por isso o exemplo não será apresentado em detalhes nesse documento.

O exemplo da função de proteção de sobretensão (PTOV) é um código VHDL desenvolvido na empresa onde o trabalho foi realizado

e que já foi utilizado em equipamentos. O *Goertzel Control* também é um exemplo prático de código desenvolvido na empresa, porém ele ainda será utilizado.

A Tabela 3 mostra que os exemplos do controlador de memória e controlador FIFO não utilizaram restrições OVL para as entradas. A ausência dessas restrições significa que o processo **input_generator** no modelo FIACRE apresenta a estrutura com apenas um estado em *loop*, gerando qualquer valor para as entradas. Os outros dois exemplos (PTOV e *Goertzel Control*) utilizaram restrições e, conseqüentemente, o processo **input_generator** possui dois estados conforme o apresentado no capítulo 5.

O tempo de verificação médio para o controlador FIFO é menor que para o *Goertzel Control*, apesar de o primeiro apresentar um espaço de estados muito maior. Esse comportamento é justificado pelo fato das propriedades especificadas para o controlador FIFO serem mais simples, apenas precisando verificar dois ciclos de *clock* em sequência, enquanto que no *Goertzel Control* algumas propriedades precisavam avaliar caminhos de execução maiores.

A seguir será apresentado em detalhes o processo de verificação para os exemplos do controlador de memória e da função de proteção. Ao final do capítulo são feitas considerações importantes sobre o exemplo *Goertzel Control*, porém os detalhes da verificação estão no Apêndice.

6.1 EXEMPLO CONTROLADOR DE MEMÓRIA

A descrição do funcionamento do controlador de memória está presente na seção 2.1.1. O código VHDL e o modelo FIACRE estão disponíveis no Apêndice A.

O modelo FIACRE não utilizou restrições OVL para as entradas, porém foi necessário manter a entrada de *reset* com valor constante que mantivesse o circuito ativo. Essa restrição é condição necessária para possibilitar o uso das fórmulas LTL geradas a partir dos padrões OVL, visto que na tradução não foi incluída a porta de *reset* (dificuldade relatada no Capítulo 5). O modelo do processo **input_generator** utilizado para a verificação é apresentado no Código 6.1.

```

1  process input_generator [generated_inputs : out input_list]
   (&output_list : read output_list) is
   states generating
   var clk, reset, mem, rw, burst : bool := false
5  init to generating
   from generating
     reset := false;
```

```

10      clk, mem, rw, burst := any;
      generated_inputs!clk, reset, mem, rw, burst;
      loop

```

Código 6.1 – Processo **input_generator** para o controlador de memória

Uma estrutura de Kripke com 1745 estados foi obtida a partir do modelo FIACRE.

6.1.1 Propriedades do controlador de memória

Uma série de propriedades podem ser descritas com os atuais padrões OVL traduzidos. Nesta seção são apresentadas quatro propriedades relevantes que foram verificadas com o uso da cadeia de verificação, tendo como base o modelo FIACRE descrito na seção anterior. Esse exemplo foi utilizado como um cenário de teste controlado para a metodologia, algumas propriedades deveriam ser aprovadas (propriedades 1,2 e 4) e uma deveria ser rejeitada (propriedade 3) pelo procedimento de verificação adotado.

Propriedade 1

Com a essa propriedade, o objetivo é garantir que quando o controlador está no estado **idle**, as saídas de leitura (**oe**) e escrita (**we**) são falsas. O estado **idle** representa que o controlador está sem atividade, portanto ele não deve executar operações de leitura ou escrita.

A propriedade pode ser representada com o uso do padrão OVL **ovl_implication**, com as seguintes configurações:

- **antecedent_expr:** *state_reg = idle*
- **consequent_expr:** $\neg oe \wedge \neg we$
- **clock:** *clk*

A propriedade em @LTL é descrita como:

- @LTL : $\Box((state_reg = idle) \rightarrow (\neg oe \wedge \neg we))@clk$

Propriedade 2

Essa propriedade tem como objetivo provar que o controlador não executa leitura e escrita simultaneamente na memória. Isso significa que as saídas **oe** e **we** não podem ser verdadeiras ao mesmo tempo.

A propriedade pode ser representada com o uso do padrão OVL `ovl_always`, com as seguintes configurações:

- **test_expr:** $\neg(oe \wedge we)$
- **clock:** `clk`

A propriedade em @LTL é descrita como:

- **@LTL :** $\Box(\neg(oe \wedge we))@clk$

Propriedade 3

O objetivo da propriedade é verificar que toda requisição de leitura é atendida, ou seja, toda vez que as entradas **mem** e **rw** são verdadeiras no mesmo ciclo de *clock*, a saída **oe** deve ser verdadeira no próximo ciclo.

A propriedade pode ser representada com o uso do padrão OVL `ovl_next`, com as seguintes configurações:

- **num_cks:** 1
- **start_event:** $mem \wedge rw$
- **test_expr:** `oe`
- **clock:** `clk`

A propriedade em @LTL é descrita como:

- **@LTL :** $\Box((mem \wedge rw) \rightarrow \bigcirc oe)@clk$

Propriedade 4

Essa propriedade busca comprovar que quando o controlador efetuar uma leitura do tipo *burst*, a quantidade de operações de leitura em sequência não é superior a quatro. Essa condição pode ser considerada como satisfeita ao provar que a saída **oe** não é verdadeira por mais de quatro ciclos de *clock* seguidos.

A propriedade pode ser representada com o uso do padrão OVL `ovl_width`, com as seguintes configurações:

- **min_cks:** 1
- **max_cks:** 4

- **test_expr:** *oe*
- **clock:** *clk*

Como o parâmetro **min_cks** é igual a um, não é necessário verificar a condição de mínimo visto que quando a saída *oe* se torna verdadeira em um ciclo de *clock*, a condição de mínimo já é satisfeita. A propriedade em @LTL para verificar a condição de máximo é descrita como:

- @LTL : $\Box \neg ((\neg oe) \wedge \bigcirc (oe \wedge \bigcirc (oe \wedge \bigcirc (oe \wedge \bigcirc (oe \wedge \bigcirc oe)))))) @clk$

6.1.2 Resultados da Verificação

Após obter a fórmula LTL da propriedade e o modelo FIACRE do código VHDL, o processo de verificação é executado através das seguintes etapas:

1. Compilação do modelo FIACRE em um sistema de transição TTS (compilador FIACRE/TTS).
2. Conversão do modelo TTS em uma estrutura de Kripke (TINA).
3. Estrutura de Kripke e a fórmula LTL são executadas na ferramenta SELT (*model checking*). A ferramenta SELT pode gerar dois resultados: propriedade aprovada ou contraexemplo (propriedade falhou). Caso a propriedade seja aprovada, a verificação é encerrada, do contrário o processo continua na etapa seguinte.
4. Conversão do contraexemplo em um arquivo de estímulos utilizando o tradutor de contraexemplo. Nessa etapa também é gerado o arquivo *testbench* VHDL que faz a leitura do arquivo com os estímulos e os aplicam ao código VHDL sob análise na simulação.
5. Execução do arquivo *testbench* em um simulador para gerar a forma de onda do contraexemplo.
6. Com auxílio da forma de onda, investigar e encontrar a fonte do erro no código VHDL.
7. Após feitas as correções, gerar o modelo FIACRE para o VHDL corrigido e retornar a etapa 1.

O *testbench* VHDL é o mesmo para a verificação de todas as propriedades do exemplo e está disponível no Apêndice A. A seguir serão apresentados os resultados da verificação.

Propriedade 1, 2 e 4:

Essas propriedades foram aprovadas no *model checking* realizado pelo SELT. Esses resultados eram esperados ao analisar o código VHDL do controlador, mostrando que a cadeia foi capaz de apontar propriedades que são satisfeitas de maneira correta.

Para essas propriedades, o tempo necessário para obter cada um dos resultados da verificação foi menor que 1s.

Propriedade 3:

Essa propriedade falhou, como esperado, no *model checking* e um contraexemplo foi gerado pela ferramenta SELT. O contraexemplo foi convertido em um arquivo de estímulos que foi executado no *testbench* VHDL. O tempo necessário em todo processo, desde conversão do modelo FIACRE para TTS até o *model checking*, foi de menos de 1s. A Figura 18 mostra a forma de onda do contraexemplo, gerada com o uso da ferramenta ModelSim (MODELSIM, 2015).

A propriedade falhou em mais de um momento sendo que um deles está marcado pela linha amarela na Figura 18. Nesse instante, as entradas **rw** e **mem** são verdadeiras na borda de subida do *clock*. Portanto, no próximo ciclo de *clock*, a saída **oe** deveria ser verdadeira, no entanto a saída se manteve falsa e a propriedade falhou.

Essa falha era esperada porque não foram feitas restrições para impedir que as entradas (a exceção do *clock*) mudassem seu valor mais de uma vez no mesmo ciclo. No contraexemplo, quando ocorre a borda de subida destaca pela linha amarela, os sinais **rw** e **mem** são verdadeiros no estado **idle**, o que implicaria que o próximo estado seria **read1** e, conseqüentemente, a saída **oe** seria verdadeira. No entanto, antes da próxima borda de subida do *clock*, o sinal **rw** passou a ser falso, o que fez com que a transição fosse para o estado **write** onde a saída **oe** é falsa.

Esse resultado mostrou que a cadeia de verificação foi capaz de indicar um erro que era esperado. Não foram feitas modificações no código VHDL ou no modelo FIACRE para corrigir a fonte do erro.

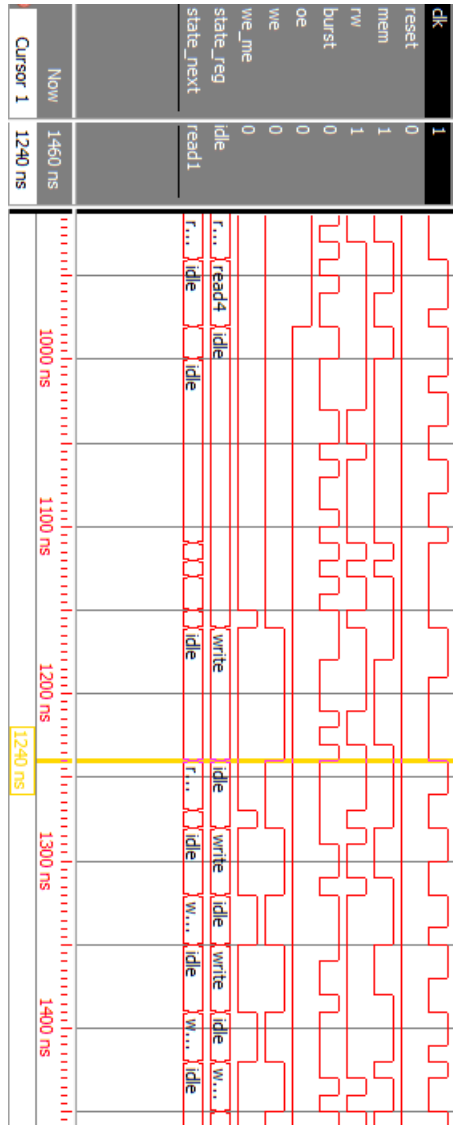


Figura 18 – Contraexemplo para propriedade 3 do controlador de memória (ferramenta ModelSim (MODELSIM, 2015))

6.2 EXEMPLO FUNÇÃO DE PROTEÇÃO

Um dos equipamentos desenvolvidos pela empresa onde o trabalho foi realizado é um relé de proteção digital para subestações de

energia elétrica. Os relés de proteção são equipamentos que estão constantemente monitorando as grandezas elétricas (corrente, tensão e frequência) das linhas conectadas a subestação para detectar a ocorrência de faltas (curto-circuitos por exemplo). Quando faltas são detectadas, os relés geram sinais de controle para disjuntores e outros equipamentos de proteção para que estes desconectem linhas ou equipamentos para evitar que ocorram danos.

A detecção das faltas em um relé é feita através das chamadas funções de proteção. Existem diversos tipos de funções de proteção que verificam os mais variados tipos de faltas, sendo uma delas a função de proteção de sobretensão (PTOV). A PTOV, em sua essência, verifica se o nível de tensão está acima de um limite configurável, e sinaliza quando isso ocorre. A Figura 19 apresenta um diagrama do funcionamento da PTOV. Na especificação da PTOV implementada no relé estava

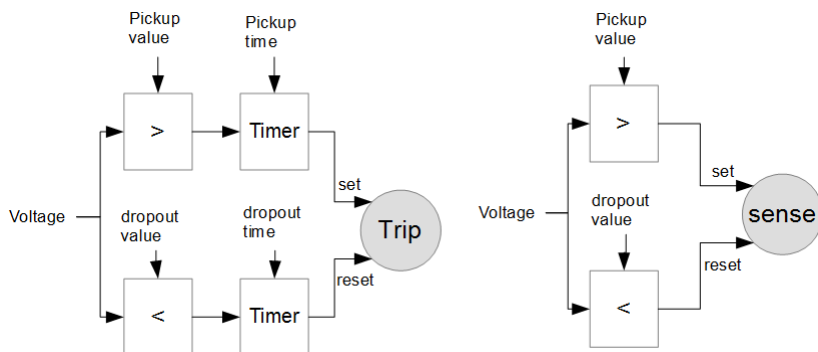


Figura 19 – Diagrama de funcionamento da PTOV

previsto duas saídas para a função: *sense* e *trip*. A saída *sense* é a sensibilização da função e deve ser verdadeira sempre que o valor da tensão monitorada for maior que o valor limite, também chamado de valor de *pickup*. Quando o valor é menor, essa saída deve ser falsa. A saída *trip* é a atuação da função, e deve ser verdadeira se a tensão foi maior que o valor limite por um tempo definido chamado de *pickup time*. Uma vez verdadeira, a saída *trip* só volta a ser falsa se o valor de tensão for menor que um valor configurável, chamado de *dropout*, por um tempo também configurável, chamado de *dropout time*. O valor de *dropout* deve ser sempre configurado como menor que o valor de *pickup*. Em geral, o valor de *dropout* costuma ser configurado como setenta por cento do valor de *pickup*. Os temporizadores utilizados na função são não retentivos, ou seja, quando a condição que habilitou a contagem

deixar de ser verdadeira, a contagem deve ser zerada.

No relé de proteção desenvolvido, todas as funções de proteção são implementadas em *hardware*, com um FPGA configurado com os circuitos digitais que devem realizá-las. A empresa tomou essa decisão porque essas funções são sistemas de tempo real crítico, onde um travamento ou atraso no processamento (perda de *deadline*) pode acarretar em grandes prejuízos (no caso das subestações, equipamentos danificados e cortes de luz). Uma implementação em *hardware* consegue manter determinismo na execução, além de maior rapidez e evitar erros devido a compartilhamento de memórias. Na empresa, os FPGAs são configurados com circuitos descritos em VHDL.

Para a função PTOV foi criado um código VHDL que executasse o comportamento descrito na Figura 19. O código VHDL deve ser capaz de fazer a checagem de sobretensão para as três fases do sistema (A, B e C) em paralelo. A estrutura **Entity** do código, contendo a declaração das portas está mostrado no Código 6.2. O restante do código pode ser encontrado no Apêndice C.

```

1 entity ptov_ptof is
  port (
    -- system signals
    sysclk   : in  std_logic;
5    reset_n : in  std_logic;

    -- interface signals
    reset_op : in  std_logic;
    enable   : in  std_logic;           -- module enable
10   sync_a  : in  std_logic;
    sync_b  : in  std_logic;
    sync_c  : in  std_logic;
    f_v_a   : in  std_logic_vector(31 downto 0); -- mag A
    f_v_b   : in  std_logic_vector(31 downto 0); -- mag B
15   f_v_c   : in  std_logic_vector(31 downto 0); -- mag C
    comp_data : in  std_logic_vector(31 downto 0); -- pickup
    reset_data : in  std_logic_vector(31 downto 0);
    dropout_data : in  std_logic_vector(31 downto 0); -- dropout
    comp_counter : in  std_logic_vector(31 downto 0);
20   sense_o  : out std_logic_vector(2 downto 0); -- sense
    op_o     : out std_logic_vector(2 downto 0) -- trip
  );
end ptov_ptof;

```

Código 6.2 – Estrutura **Entity** do código da PTOV

A entrada **sysclk** é o sinal de *clock* do circuito. A entrada **reset_n** é um *reset* assíncrono (ativo baixo). A porta **reset_op** é a entrada de um *reset* síncrono para a saída de *Trip* (ativo alto). O sinal **enable** habilita a função.

O circuito descrito no VHDL é síncrono ao sinal **sysclk** (estado do sistema evolui a cada borda ativa desse sinal), porém a verificação de

sobretensão não acontece a todo momento, mas somente quando novos valores de magnitude de tensão são calculados. Um pulso (duração de um ciclo de *clock*) nas entradas **sync_a**, **sync_b** e **sync_c** indicam que novos valores foram calculados para as fases A, B e C respectivamente, e então o circuito realiza a verificação de sobretensão para esses valores. Os valores de magnitude estão disponíveis nas entradas **f_v_a** (fase A), **f_v_b** (fase B) e **f_v_c** (fase C).

As entradas **comp_data**, **reset_data**, **dropout_data** e **comp_counter** são parâmetros de configuração da função, cujos valores não são alterados enquanto ela está habilitada (entrada **enable** com valor 1). O parâmetro **comp_data** é o valor de *pickup* para as três fases, enquanto a entrada **dropout_data** é o valor de *dropout*. O parâmetro **comp_counter** define quantos pulsos do sinal de **sync** (**sync_a**, **sync_b** e **sync_c**) consecutivos com a magnitude acima do valor de *pickup* são necessários para disparar o *Trip*. O parâmetro **reset_data** tem comportamento análogo ao **comp_counter**, porém definindo a quantidade de pulsos consecutivos necessários com a magnitude menor que o *dropout* para que a saída *Trip* seja zerada.

A saída **sense_o** são os sinais de *sense* da função para cada fase. O bit menos significativo do vetor é o *sense* da fase A, enquanto o mais significativo é o *sense* da fase C. A saída **op_o** contém os sinais de *Trip* de cada fase, sendo o bit menos significativo para a fase A e o mais significativo para a fase C.

6.2.1 Modelo FIACRE da PTOV

Para ser possível fazer a verificação do VHDL da PTOV usando a cadeia de verificação FIACRE, é necessária a criação do modelo FIACRE correspondente ao código, utilizando as regras de tradução apresentadas no capítulo 4.

O código VHDL como concebido originalmente continha 3 blocos idênticos do tipo VHDL **process**, funcionando de modo assíncrono e independente entre eles, sendo um para cada fase do sistema. A existência de sistemas assíncronos é muito ruim em um sistema de transição, pois permite a existência de uma quantidade muito grande de estados no sistema completo, podendo levar facilmente a uma explosão de estados e inviabilizando a verificação por *model checking*. Como a verificação de sobretensão é independente para cada fase (cada uma possui seu sinal indicando novo valor de magnitude, sua própria entrada de magnitude, sua saída de *sense* e *trip*, sinais internos e contadores também

próprios) e o código é idêntico, não se faz necessário fazer a tradução e verificação do VHDL correspondente a todas as fases, mas sim apenas de uma. O modelo FIACRE criado traduziu somente a parte do VHDL relacionado a fase A para evitar explosão de estados.

Em geral, as regras de tradução apresentadas no capítulo 4 são suficientes para a tradução do VHDL da PTOV, sendo que os sinais cujo tipo eram **std_logic_vector** (inclusive as portas de entrada) foram traduzidos para tipos inteiros em FIACRE, visto que eram utilizados no código somente para comparações de valores e não havia a presença de operações com *arrays*. Os sinais e entradas que utilizavam tipo **std_logic** foram traduzidos para tipos booleanos em FIACRE. O código do processo FIACRE **delta_cycle** está no Apêndice C.

O processo FIACRE **input_generator** criado para a PTOV não foi concebido para gerar todas as entradas possíveis. O circuito possui alguns vetores de entrada com tamanho de 32 bits (valor de magnitude das tensões, parâmetros de configuração), logo, se o modelo do **input_generator** fosse gerar qualquer valor para elas, a quantidade de estados do sistema ficaria gigantesca. Portanto, o modelo de entradas foi concebido para gerar um comportamento mais realístico (menos aleatório) para as entradas, sendo que os valores de **f.v.a** (convertido para inteiro no modelo FIACRE) foram limitados.

Outras restrições foram aplicadas aos valores da entrada, porém sempre com o objetivo de apenas eliminar verificação de situações impossíveis ou repetitivas (quando a verificação para uma faixa de valores já é suficiente para verificar o atendimento de propriedades). Como a tradução dos padrões OVL para fórmulas LTL não contempla os sinais de *reset*, as entradas **reset_n**, **reset_op** e **enable** foram mantidas em valores constantes que habilitam a função. Os parâmetros de configuração (**reset_data**, **comp_counter**, **dropout_data** e **comp counter**) da função não devem ter seus valores alterados enquanto ela estiver habilitada, por isso eles foram mantidos em valores constantes, sendo que $dropout_data \leq comp_data < vector_size$, $reset_data \leq 4$ e $comp_counter \leq 4$, para limitar o espaço de estados.

No relé de proteção, os sistemas de aquisição são sincronizados e geram novos valores de magnitude em um intervalo fixo de ciclos de *clock*. Em geral, esse intervalo é de 100 mil ciclos (1ms), portanto, o pulso na entrada *sync_a* ocorre obedecendo esse intervalo. Para não gerar um espaço de estados excessivamente grande de modo desnecessário, o pulso foi modelado a ocorrer a cada 10 ciclos, tempo suficiente para a execução da função, e para isso foi necessário utilizar os observadores OVL **ovl_next**, **ovl_width** e **ovl_unchange** com as seguintes

configurações:

- **ovl_next:**

- *num_cks*: 1
- *check_overlapping*: 1
- *check_missing_start*: 0
- *start_event*: sync_a
- *test_expr*: not sync_a

- **ovl_width:**

- *min_cks*: 9
- *max_cks*: 9
- *test_expr*: not sync_a

- **ovl_unchange:**

- *num_cks*: 10
- *action_on_new_start*: OVL_RESET_ON_NEW_START
- *start_event*: sync_a
- *test_expr*: f_v_a

O Código 6.3 apresenta o modelo do **input_generator** definido com os seguintes parâmetros de configuração: **comp_data = 12**, **reset_data=2**, **dropout_data=9** e **comp_counter = 4**.

```

1  process input_generator [generated_inputs : out input_list]
   (&output_list : read output_list) is
   states clk_false, clk_true
   var sysclk, reset_n, reset_op, enable, sync_a : bool := false,
5   f_v_a : 0..vector_size := 0,
   comp_data, reset_data : nat := 0,
   dropout_data, comp_counter : nat := 0,
   //ovl_next
   ovl_next_fifo : queue 1 of nat := {},
10  ovl_next_cnt : nat := 0,
   //ovl_width
   pre_ovl_width : bool := false,
   ovl_width_cnt : nat := 0,
   //ovl_unchange
15  pre_ovl_unchange : 0..vector_size := 0,
   ovl_unchange_cnt : nat := 0
   init to clk_false
   from clk_false
   sysclk := false;
20  reset_n := true;
   reset_op := false;
```

```

enable := true;
comp_data := 12; //pickup value
reset_data := 2; //reset time
25 dropout_data := 9; //dropout value
comp_counter := 4; //pickup time
generated_inputs!sysclk, reset_n, reset_op, enable, sync_a,
f_v_a, comp_data, reset_data, dropout_data, comp_counter;
to clk_true
30 from clk_true
sysclk := true;
// ovl_unchange (start_event = sync_a, test_expr = f_v_a)
if sync_a then
    ovl_unchange_cnt := 1
35 else
    if ovl_unchange_cnt > 0 then
        if ovl_unchange_cnt = 10 then
            ovl_unchange_cnt := 0
40         else
            ovl_unchange_cnt := ovl_unchange_cnt + 1
        end if
    end if;
pre_ovl_unchange := f_v_a;
45 // ovl_width (min_cks=max_cks=9, test_expr = not sync_a)
if (pre_ovl_width = false) and (not sync_a) then
    ovl_width_cnt := 1
else
50     if ovl_width_cnt > 0 then
        if (ovl_width_cnt = 10) or (pre_ovl_width = false) then
            ovl_width_cnt := 0
        else
55             ovl_width_cnt := ovl_width_cnt + 1
        end if
    end if;
pre_ovl_width := not sync_a;
/* ovl_next (start_event = sync_a, test_expr = not sync_a,
num_cks = 1) */
60 if (ovl_next_cnt > 0) and sync_a then
    ovl_next_fifo := enqueue(ovl_next_fifo, ovl_next_cnt)
end if;
if (ovl_next_cnt = 1) then
    if empty(ovl_next_fifo) then
65         ovl_next_cnt := 0
    else
        ovl_next_cnt := ovl_next_cnt - first(ovl_next_fifo) + 1;
        ovl_next_fifo := dequeue(ovl_next_fifo)
    end if
70 else
    if ovl_next_cnt > 0 then
        ovl_next_cnt := ovl_next_cnt + 1
    else
        if sync_a then
75             ovl_next_cnt := 1
        end if
    end if;
end if;
////////////////////////////////////
80 sync_a, f_v_a := any where (
((not ((pre_ovl_unchange <> f_v_a) and
(ovl_unchange_cnt > 0))) or sync_a)

```

```

and
85 (not ((ovl_width_cnt <= 9) and (ovl_width_cnt > 0)
and sync_a))
and
(not ((ovl_width_cnt > 9) and (not sync_a)))
and
90 (((not sync_a) and (ovl_next_cnt = 1))
or (ovl_next_cnt <> 1))
);
generated_inputs!sysclk, reset_n, reset_op, enable, sync_a,
f_v_a, comp_data, reset_data, dropout_data, comp_counter;
to clk_false

```

Código 6.3 – `input_generator` da PTOV

A linha 5 do Código 6.3 define a limitação da entrada `f_v_a` como no intervalo 0 a `VECTOR_SIZE`, sendo que este último é uma constante que foi definida como 15. A entrada `sysclk` foi definida como *clock*. Uma estrutura de Kripke com 12833 estados foi obtida a partir do modelo FIACRE com essas configurações para o `input_generator`.

A verificação das propriedades não ficou restrita ao modelo com os parâmetros de configuração acima, ela também foi aplicada a todos os modelos possíveis para `reset_data` e `comp_counter` dentro do intervalo delimitado, o que é importante, pois modificações na relação entre eles altera significativamente o comportamento do circuito. Os parâmetros `comp_data` e `dropout_data` não precisam ter seus valores alterados porque só são utilizados para comparar em que região a magnitude de tensão se situa: acima do *pickup*, abaixo do *dropout* ou entre os valores.

Algumas propriedades tinham a finalidade de verificar o comportamento do circuito quanto ao ativamento dos sinais de *reset* e, para esses casos, a verificação foi realizada em cima de um modelo onde as entradas `reset_n`, `reset_op` e `enable` poderiam possuir qualquer valor.

6.2.2 Propriedades da PTOV

Uma série de propriedades da PTOV podem ser descritas com os atuais padrões OVL traduzidos. Nessa seção são apresentadas cinco propriedades relevantes que foram verificadas com o uso da cadeia de verificação FIACRE, tendo como base o modelo FIACRE descrito na seção anterior. As propriedades verificadas são requisitos de funcionamento presentes na especificação da PTOV que precisam serem satisfeitas.

Propriedade 1

O objetivo da propriedade é verificar que se tensão for maior que o valor de *pickup* quando ocorre o pulso de sincronização **sync_a** (pulso que sinaliza novo valor calculado), no próximo pulso a saída **sense_o** deve ser verdadeira. Com essa propriedade validada, garante-se que o circuito sempre sensibiliza quando detecta níveis de sobretensão. Essa é uma funcionalidade essencial que se não for satisfeita, a PTOV não é considerada como implementada.

A propriedade pode ser representada com o uso do padrão OVL **ovl_next**, com as seguintes configurações:

- **num_cks:** 1
- **start_event:** $f_v_a > comp_data$
- **test_expr:** `sense_o`
- **clock:** `sync_a`

A propriedade em @LTL é descrita como:

- @LTL : $\Box((f_v_a > comp_data) \rightarrow \bigcirc sense_o)@sync_a$

Propriedade 2

Essa propriedade tem como objetivo garantir que a função seja desativada quando o sinal **enable** for falso, o que significa que suas saídas (**sense_o** e **op_o** devem ser falsas. É fundamental que a função seja possível de ser desabilitada, porque o relé pode ser configurado para efetuar a proteção da subestação utilizando outras funções. Para essa verificação, o modelo FIACRE teve que ser um pouco alterado, com o processo **input_generator** gerando qualquer valor para o sinal **enable**, em vez de manter sempre verdadeiro.

A propriedade pode ser representada com o uso do padrão OVL **ovl_implication**, com as seguintes configurações:

- **antecedent_expr:** $\neg enable$
- **consequent_expr:** $(\neg sense_o) \wedge (\neg op_o)$
- **clock:** `sysclk`

A propriedade em @LTL é descrita como:

- $@LTL : \Box(\neg enable \rightarrow (\neg sense_o \wedge \neg op_o))@sysclk$

Propriedade 3

Essa verificação procura provar que uma vez que a função esteja sensibilizada (**sense_o=1**), ela somente deixa de estar nesse estado se o valor de tensão for menor que *dropout*. É muito importante que a função não fique oscilando sua sensibilização sem que o valor de tensão indique essa necessidade, pois pode levar a erros em dispositivos que estejam dependendo dessa informação.

A propriedade pode ser representada com o uso do padrão OVL **ovl_window**, com as seguintes configurações:

- **start_event:** sense_o
- **end_event:** $f_v_a < dropout_data$
- **test_expr:** sense_o
- **clock:** sysclk

A propriedade em @LTL é descrita abaixo. Primeiro é apresentado a tradução usando **weak until** (W) e em seguida o correspondente utilizando **strong until** (U):

1. $@LTL : \Box(((\neg sense_o) \wedge \bigcirc (sense_o \wedge \neg(f_v_a < dropout_data))) \rightarrow \bigcirc \bigcirc (sense_o W (f_v_a < dropout_data)))@sysclk$
2. $@LTL : \Box(((\neg sense_o) \wedge \bigcirc (sense_o \wedge \neg(f_v_a < dropout_data))) \rightarrow \bigcirc \bigcirc (\neg(\neg(f_v_a < dropout_data) U \neg(sense_o \vee (f_v_a < dropout_data))))@sysclk$

Propriedade 4

Essa propriedade tem como objetivo garantir que a saída de *trip* (**op_o**) é desativada quando seu *reset* síncrono é ativado. Para essa verificação, o modelo FIACRE teve que ser um pouco alterado, com o processo **input_generator** gerando qualquer valor para o sinal **reset_op**, em vez de manter sempre falso.

A propriedade pode ser representada com o uso do padrão OVL **ovl_next**, com as seguintes configurações:

- **num_cks:** 1
- **start_event:** reset_op

- **test_expr:** $\neg op_o$
- **clock:** `sysclk`

A propriedade em @LTL é descrita como:

- @LTL : $\square(reset_op \rightarrow \bigcirc(\neg op_o))@sysclk$

Propriedade 5

Após a verificação da correta ativação do sinal de *sense* na propriedade 1, é importante também verificar que saída de *trip* (**op_o**) é ativada no momento certo. Essa saída deve ser ativada sempre que acontecer uma quantidade **comp_counter** de pulsos de **sync_a** consecutivos em que a tensão é maior que o valor de *pickup*. Essa propriedade também define um requisito fundamental da PTOV porque é o procedimento de detecção de falta de sobretensão. Caso essa propriedade não seja respeitada, a PTOV não é considerada como implementada.

Para ser possível essa verificação, será utilizado um contador presente no código da PTOV chamado de **timer_reg**. Esse contador incrementa seu valor toda vez que acontece o pulso de sincronização em situação de sobretensão. A propriedade pode ser representada com o uso do padrão OVL **ovl_next**, com as seguintes configurações:

- **num_cks:** 1
- **start_event:** $(f_v_a > comp_data) \wedge (timer_reg = comp_counter)$
- **test_expr:** `op_o`
- **clock:** `sync_a`

A propriedade em @LTL é descrita como:

- @LTL : $\square(((f_v_a > comp_data) \wedge (timer_reg = comp_counter)) \rightarrow \bigcirc op_o)@sync_a$

6.2.3 Resultados da Verificação

Após obter a fórmula LTL da propriedade e o modelo FIACRE do código VHDL, o processo de verificação é executado com as mesmas etapas descritas para o exemplo do controlador de memória. O *test-bench* VHDL é o mesmo para a verificação de todas as propriedades da PTOV e está disponível no Apêndice C. A seguir serão apresentados

os resultados da verificação.

Propriedade 1:

Para essa propriedade, o processo de verificação foi realizado em mais de um modelo FIACRE do código da PTOV. O processo **delta_cycle** dos modelos FIACRE é o mesmo, pois todos tiveram como origem o mesmo código VHDL, e as modificações ocorrem somente no processo **input_generator**, responsável pela geração das entradas. A diferença entre os modelos eram os valores fixados para os parâmetros de configuração **comp_counter** e **reset_data**, que definem contagem para ativação e desativação do sinal de *trip*. Os demais parâmetros mantiveram as seguintes configurações:

- **comp_data:** 12
- **dropout_data:** 9

Os sinais de *reset* e *enable* foram mantidos em valores que habilitavam a função.

Essa propriedade falhou para os modelos em que a contagem para desativação da saída **op_o** (**reset_data**) foi definido como diferente de 0. Um contraexemplo foi gerado pela ferramenta SELT e convertido em um arquivo de estímulos que foi executado no *testbench* VHDL. O tempo gasto em todo processo, desde conversão do modelo FIACRE para TTS até o *model checking*, foi de menos de 1s. A Figura 20 mostra a forma de onda do contraexemplo, gerada com o uso da ferramenta ModelSim (MODELSIM, 2015).

A propriedade falhou em mais de um momento sendo que um deles está marcado pela linha amarela na Figura 20. O valor da entrada **f_v_a** (valor era 13) era maior que o *pickup* (configurado como valor 12), porém no pulso seguinte de **sync_a** a saída **sense_o** ainda se manteve em 0 (falso). Observando a evolução dos sinais internos na forma de onda e o código, observa-se que a origem da falha era uma condição de transição para o estado onde o **sense_o** era ativado. A condição exigia que **op_o** estivesse desativado para o circuito fazer a transição para um estado onde a saída **sense_o** era ativada.

Correções foram feitas no código VHDL (o código VHDL corrigido e o modelo FIACRE estão no Apêndice C), o processo de verificação foi repetido e a propriedade não falhou mais. O mais interessante quanto a falha encontrada é que ela havia passado despercebida em simulações e em testes realizados em equipamento, no entanto,

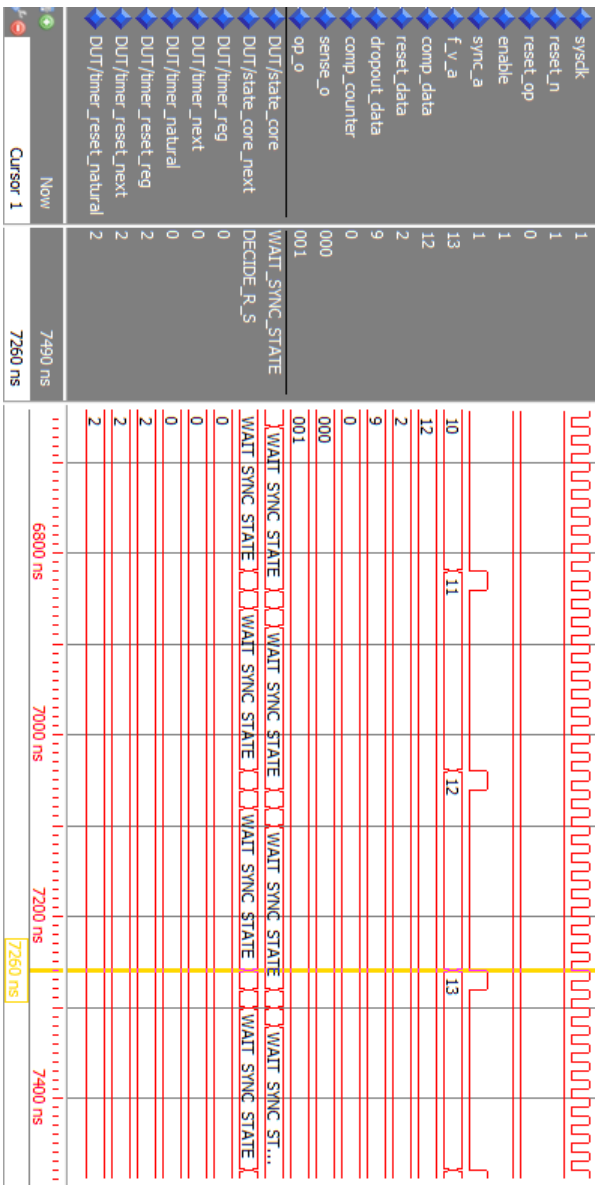


Figura 20 – Contraexemplo para propriedade 1 da PTOV (figura extraída da ferramenta ModelSim (MODELSIM, 2015))

através do processo de *model checking* ela foi revelada sem grandes esforços do desenvolvedor já que um contraexemplo foi gerado automaticamente representando o cenário de erro, evitando assim que o usuário tivesse que avaliar, em busca do erro, as formas de onda geradas para cada cenário de simulação.

Propriedade 2-5:

As demais propriedades verificadas para PTOV foram aprovadas no *model checking* realizado pelo SELT. Como mencionado anteriormente, para as propriedades 2 e 4, o modelo FIACRE utilizado tinha o processo **input_generator** com os sinais de *reset* e habilitação podendo ser qualquer valor. Para as propriedades 3 e 5, os modelos FIACRE utilizados foram os mesmos da propriedade 1, ou seja, modificação nos valores fixados para os parâmetros de configuração.

Assim como na propriedade 1, o tempo necessário para obter os resultados da verificação para cada propriedade foi menor que 1s.

6.3 EXEMPLO GOERTZEL CONTROL

A metodologia também foi aplicada a outro exemplo de código VHDL desenvolvido na empresa, porém com a diferença de ser um código que ainda será utilizado nos equipamentos enquanto que o exemplo da função de proteção era um código que já foi usado.

Dentro do relé de proteção, os fasores para as medições de corrente e tensão são calculados por meio de um módulo que executa o algoritmo de Goertzel com as amostras obtidas na aquisição. Esse circuito que executa o Goertzel está conectado ao restante dos módulos VHDL do equipamento através de um barramento de transferência de dados chamado de **Avalon Streaming (Avalon-ST)** (AVALON-ST, 2011). O módulo deve receber, por meio do barramento, um pacote de dados, extrair a informação necessária para os cálculos (número de amostras para cálculo e as amostras) do pacote e repassar o mesmo para o módulo seguinte, adicionando ao final os fasores calculados.

O circuito VHDL criado para o Goertzel é dividido em dois módulos: um para fazer o controle dos sinais no barramento **Avalon**, determinado de **Goertel Control**, e outro para executar o algoritmo de Goertzel propriamente dito. A metodologia foi aplicada para verificar propriedades do circuito que faz o controle de acesso ao barramento. O módulo que o executa o algoritmo, por ser tratar um circuito essencial-

mente orientado a manipulação de dados, é uma aplicação complicada para *model checking* por levar facilmente a uma situação de explosão de estados devido a uma quantidade muito grande de valores possíveis para as entradas, no entanto ele pode ser validado por simulação com alto grau de confiabilidade por se tratar apenas de um sequenciamento fixo de operações lógicas/aritméticas sobre os vetores, sem caminhos de execução opcionais.

Para esse exemplo não foi necessário criar vários modelos FIACRE, sendo a diferença entre eles era a estrutura do **input_generator**. Apenas um modelo foi criado, porém a diferença fundamental entre o **input_generator** desse exemplo para a função de proteção foi a necessidade de definir restrições quanto ao comportamento das entradas que estavam relacionados com os valores das saídas. O comportamento correto de alguns sinais do barramento Avalon-ST depende de como o circuito que está recebendo os dados está respondendo, levando necessidade de atrelar o comportamento das entradas a algumas saídas do circuito. Outra restrição importante das entradas foi manter a parte de dados do barramento em valores constantes, o que não traz impacto para o *model checking*, visto que os laços de controle do circuito não dependem dos dados.

O código VHDL do *goertzel control* possui cerca de 400 linhas, e o modelo FIACRE gerado possui aproximadamente 650 linhas. A compilação do modelo FIACRE com a ferramenta TINA gerou uma estrutura de Kripke bem maior que no exemplo da função de proteção, com 383845 estados.

As propriedades verificadas para o circuito tinham como objetivo garantir que as regras de correto funcionamento do barramento Avalon-ST são respeitadas. Vários módulos desenvolvidos na empresa precisam fazer interface com esse tipo de barramento, portanto as propriedades criadas para esse exemplo podem servir como padrões para verificação juntamente com as traduções OVL.

O tempo necessário para verificação de cada uma das propriedades (geração da estrutura de Kripke mais a verificação da ferramenta SELT) foi bem maior que no exemplo da função de proteção, em virtude principalmente do espaço de estados ser bem maior. Enquanto para a PTOV o tempo necessário foi de aproximadamente 1s por propriedade, nesse exemplo foi preciso 40s.

Uma das propriedades verificadas falhou e observando a forma de onda do contraexemplo foi possível identificar o erro no código que levou ao problema. Após a realização das correções o processo de verificação foi repetido para a propriedade e ficou comprovado que o erro

foi corrigido. O importante da identificação desse erro é que ele ocorreu para um código em desenvolvimento, contribuindo no aumento da confiabilidade, além de reduzir o processo de verificação pelo fato do *model checking* já entregar um contraexemplo que conduzia ao erro.

A apresentação completa do exemplo pode ser encontrada no Apêndice D. Para esse exemplo não é apresentado o código VHDL completo.

6.4 CONCLUSÃO

Neste capítulo a metodologia foi aplicada a três exemplos práticos. Esses exemplos mostraram que a proposta é viável e que pode contribuir bastante não só para aumentar a confiabilidade dos equipamentos ao identificar erros difíceis de serem encontrados em simulação, como no caso da função de proteção, mas também para agilizar o processo de verificação ao gerar contraexemplos automaticamente para propriedades que falham.

Os exemplos já permitiram visualizar que a restrição no comportamento das entradas é importante para evitar explosão de estados, principalmente nos caminhos de dados, como no caso da função de proteção em que a entrada de dados era originalmente vetores de 32 bits (as ferramentas de verificação não foram capazes de funcionar para entradas com faixa de valores tão grandes), mas na verificação foi reduzida para 4. Ainda é necessário executar, em trabalhos futuros, uma avaliação da escalabilidade das ferramentas de verificação utilizadas na metodologia, aplicando-a em outros exemplos.

7 CONCLUSÃO

Este trabalho foi desenvolvido em um contexto empresarial, onde a companhia buscava aperfeiçoar o processo de desenvolvimento de código VHDL sintetizável utilizado. A empresa desenvolve equipamentos para subestações de energia elétrica, que são sistemas de tempo real crítico e por isso seus produtos precisam ter elevada confiabilidade. Grande parte de seus produtos utilizam FPGAs como módulo principal de processamento, sendo programados com linguagem VHDL. No método de desenvolvimento usado, o processo de verificação do VHDL era baseado em validação por simulação e testes em equipamentos, técnicas mais comumente utilizadas na indústria mas que não são capazes de garantir o atendimento de propriedades por não serem verificações exaustivas.

Esse trabalho propôs o aperfeiçoamento da metodologia de desenvolvimento de código VHDL sintetizável utilizada ao incluir verificação formal por *model checking*. A proposta tinha como objetivo específico evitar que o desenvolvedor VHDL necessitasse de conhecimentos adicionais para utilização do *model checking*, procurando manter a interface com o processo de verificação em linguagem de usuário, isto é, VHDL ou ferramentas orientadas a essa linguagem.

O processo de verificação apresentado na metodologia ocorre através de uma cadeia de verificação cujo *front-end* são modelos descritos na linguagem intermediária FIACRE e propriedades descritas em lógica temporal LTL. Para evitar o contato direto com lógicas temporais pelo usuário, a metodologia propôs o uso de padrões de propriedade baseados na biblioteca de observadores para VHDL/Verilog chamada OVL, fazendo a tradução dessa biblioteca para fórmulas em lógica temporal LTL a fim de serem usadas pela ferramenta de *model checking*. A linguagem FIACRE também não é de conhecimento dos desenvolvedores VHDL e, por isso, este trabalho propôs regras de tradução VHDL-FIACRE para que essa transformação possa ocorrer de modo automático através de uma ferramenta. Nessa etapa de tradução, os observadores OVL foram utilizados também, porém dessa vez para auxiliar o usuário a definir restrições quanto ao comportamento das entradas no modelo para verificação sem que o mesmo precisasse fazer alterações diretamente utilizando a linguagem FIACRE. Os contraexemplos gerados pela cadeia, para as propriedades que falharam, são convertidos em *testbenches* VHDL para que o usuário possa executar na ferramenta de simulação preferida e visualizar a forma de onda que

leva a situação de erro.

A metodologia proposta foi validada, com sucesso, ao ser aplicada em quatro exemplos de código VHDL, sendo dois deles desenvolvidos na empresa. Um código já foi utilizado em equipamentos da empresa (exemplo PTOV) e outro que ainda será aplicado (exemplo *Goertzel Control*). No exemplo da PTOV, uma falha foi encontrada que havia passado despercebida em simulações e testes em equipamentos realizados anteriormente. Para o exemplo do *Goertzel Control*, a metodologia também foi útil para a encontrar erros mais rapidamente, reduzindo o tempo de verificação. Além disso, as propriedades criadas para esse exemplo já podem até se tornarem um padrões para verificação de barramento **Avalon-ST**.

Os exemplos foram importantes para apresentar alguns problemas da metodologia que precisam ser resolvidos. Os exemplos de código da empresa mostraram que as restrições para as entradas (processo **input_generator**) do modelo FIACRE são de fundamental importância para o *model checking*, tendo grande impacto no espaço de estados e nos resultados da verificação. Porém, o processo de definição dessas restrições não é simples para um desenvolvedor não acostumado com verificação, visto que cada restrição pode acontecer por diversas necessidades (redução de espaço de estados, propriedade relacionada a um comportamento específico, evitar situações que o circuito não está preparado para responder, etc.). É preciso propor um método que torne o processo de definição de restrições menos exigente do desenvolvedor VHDL, não necessitando que ele tenha conhecimento avançado de verificação.

A metodologia ainda apresenta poucos observadores traduzidos para restrições quanto às entradas e é importante oferecer mais traduções para aumentar as opções do usuário quanto ao modelo de verificação. Outro problema importante é a tradução dos observadores OVL sem o sinal de *reset*. A inclusão desse sinal nas traduções retiraria a necessidade da criação de modelos FIACRE específicos, que possuam **input_generator** com o *reset* ativo, somente para verificar propriedades que avaliam reposta de circuitos para situações de *reset* (como foi feito para os exemplos).

Para automatizar o uso da metodologia e evitar a erros devido a traduções manuais, ainda é necessária a criação de três ferramentas: transformação VHDL-FIACRE, tradutor OVL-LTL, tradutor de contraxemplos para *testbench* VHDL. Nos exemplos mostrados as traduções foram feitas manualmente, porém para o tradutor de contraxemplos já foi utilizada uma ferramenta para auxílio. A tradução

VHDL-FIACRE, por ser uma tradução mais complexa, será realizada em uma ferramenta desenvolvida sob os conceitos MDE para trazer rigor ao processo, utilizando regras de tradução e metamodelos para evitar erros que possam comprometer o processo de verificação. As regras de tradução utilizadas serão as propostas nesse trabalho.

Dois artigos foram criados como resultados deste trabalho. Um artigo foi aprovado para o evento *Simpósio Brasileiro de Métodos Formais 2015*¹, porém não foi publicado porque o autor não pôde comparecer ao evento para a realização da apresentação. O outro artigo, com o título *Improving a Design Methodology of Synthesizable VHDL With Formal Verification*, foi aprovado para o evento *VII IEEE Latin American Symposium On Circuits and Systems*² e foi publicado.

Com a conclusão deste trabalho, novas perspectivas e projeções para trabalhos futuros surgiram. A complementação deste trabalho, a partir dos resultados obtidos, pode ser feita através dos seguintes acréscimos e atividades futuras:

- Criação da ferramenta de tradução VHDL-FIACRE. Para finalizar as regras de tradução e facilitar o processo, é preciso criar tipos FIACRE para os tipos vetoriais VHDL. É interessante incluir tradução de módulos VHDL hierárquicos (código VHDL que possui instância de outros códigos), não contemplado no modelo atual de tradução.
- Criação da ferramenta que traduz observadores OVL para fórmulas LTL ou restrições para entradas.
- Finalizar a ferramenta de tradução de contraexemplos.
- Propor um método para definição de restrições para as entradas no modelo de verificação.
- Tradução de mais observadores OVL para lógica LTL, de preferência incluir uma solução que inclua as entradas de *reset*
- Tradução de mais observadores OVL para restrição das entradas (modelos de processo **input_generator**).
- Aplicar a metodologia em mais exemplos práticos para avaliar a escalabilidade do método.

¹<http://cbsoft.org/sbmf2015>

²<http://gse.ufsc.br/lascas2016/>

REFERÊNCIAS

- ABID, N. **Verification of real time properties in fiacre language**. Tese (Doutorado) — Toulouse, INSA, 2012.
- ACCELLERA. **Accellera Systems Initiative - Home**. 2015. <http://accellera.org/>.
- ATLAS. **Atlas Transformation Language. ATL Starter's Guide - version 0.1**. Nantes, França, 2005.
- AVALON-ST. **Avalon Interface Specifications**. [S.l.], Maio 2011.
- AYAV, T.; TUGLULAR, T.; BELLI, F. Towards test case generation for synthesizable VHDL programs using model checker. In: **IEEE. Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on**. [S.l.], 2010. p. 46–53.
- BAIER, C.; KATOEN, J.-P. **Principles of Model Checking**. [S.l.]: MIT Press, 2008.
- BENING, L.; FOSTER, H. **Principles of verifiable RTL design**. [S.l.]: Springer, 2000.
- BERRY, G.; GONTHIER, G. The Esterel synchronous programming language: Design, semantics, implementation. **Science of computer programming**, Elsevier, v. 19, n. 2, p. 87–152, 1992.
- BERRY, G.; KISHINEVSKY, M.; SINGH, S. System level design and verification using a synchronous language. In: **IEEE COMPUTER SOCIETY. Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design**. [S.l.], 2003. p. 433.
- BERTHOMIEU, B. et al. Formal verification of AADL specifications in the TOPCASED environment. In: **Reliable Software Technologies–Ada-Europe 2009**. [S.l.]: Springer, 2009. p. 207–221.
- BERTHOMIEU, B. et al. **The Syntax and Semantics of FIACRE**. [S.l.], 2012. Disponível em: <http://projects.laas.fr/fiacre//doc/fiacre.pdf>.

BERTHOMIEU, B.; VERNADAT, F. Réseaux de Petri temporels: méthodes d'analyse et vérification avec TINA. **Traité IC2**, p. 101, 2006.

BROWN, S. D. et al. **Field-programmable gate arrays**. [S.l.]: Springer Science & Business Media, 2012.

CADENCE. **Cadence Incisive Formal Verifier**. November 2014. http://www.cadence.com/products/fv/formal_verifier/pages/default.aspx.

CHU, P. P. **RTL Hardware Design Using VHDL**. [S.l.]: Wiley-Interscience, 2006.

CLARKE, E. et al. Progress on the state explosion problem in model checking. In: SPRINGER. **Informatics**. [S.l.], 2001. p. 176–194.

CLARKE, E. M.; EMERSON, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. **Logics of Programs**, 1981.

CLARKE, E. M.; WING, J. M. Formal methods: State of the art and future directions. **ACM Computing Surveys (CSUR)**, ACM, v. 28, n. 4, p. 626–643, 1996.

DÉHARBE, D.; SHANKAR, S.; CLARKE, E. M. Formal verification of VHDL: the model checker CV. In: IEEE. **Integrated Circuit Design, 1998. Proceedings. XI Brazilian Symposium on**. [S.l.], 1998. p. 95–98.

DWYER, M. B.; AVRUNIN, G. S.; CORBETT, J. C. Patterns in property specifications for finite-state verification. In: **ICSE**. [S.l.: s.n.], 1999.

EISNER, C. et al. The definition of a temporal clock operator. In: **Automata, Languages and Programming**. [S.l.]: Springer, 2003. p. 857–870.

FARINES, J. M. et al. A model-driven engineering approach to formal verification of PLC programs. In: IEEE. **Emerging Technologies & Factory Automation (ETFA)**. [S.l.], 2011. p. 1–8.

GOMES, L. et al. From Petri net models to VHDL implementation of digital controllers. In: IEEE. **Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE**. [S.l.], 2007. p. 94–99.

LAAS/CNRS. **The TINA Toolbox Home Page**. November 2014.
<http://projects.laas.fr/tina/>.

LEVINE, J. R.; MASON, T.; BROWN, D. **Lex & yacc**. [S.l.]:
"O'Reilly Media, Inc.", 1992.

MDA. **The Model-Driven Architecture - Guide Version 1.0.1**.
[S.l.], 2003.

MICHELLI, G. D. **Synthesis and optimization of digital circuits**.
[S.l.]: McGraw-Hill Higher Education, 1994.

MODELSIM. **ModelSim ASIC and FPGA Design - Mentor Graphics**. Maio 2015.
<http://www.mentor.com/products/fv/modelsim/>.

MOREIRA, T. G. et al. Automatic code generation for embedded systems: From uml specifications to vhdl code. In: IEEE. **Industrial Informatics (INDIN), 2010 8th IEEE International Conference on**. [S.l.], 2010. p. 1085–1090.

NAVABI, Z. **VHDL: Analysis and Modeling of Digital Systems**. [S.l.]: McGraw-Hill, 1997.

OVL. **Accellera Standard OVL V2: Library Reference Manual**. [S.l.], 2014.

PNUELI, A. The temporal logic of programs. In: **18th Annual Symposium on Foundations of Computer Science**. Providence, RI, USA: [s.n.], 1977.

PSL. **Property Specification Language Reference Manual**. [S.l.], 2004.

QUESTA. **Questa Formal Verification - Mentor Graphics**.
Novembro 2014.
<http://www.mentor.com/products/fv/questa-formal/>.

SAAD, R. T. et al. Elementos para a construção de uma cadeia de verificação para o projeto TOPCASED. Florianópolis, SC, 2008.

SCHMIDT, D. C. Guest editor's introduction: Model-driven engineering. **Computer**, IEEE Computer Society, v. 39, n. 2, p. 0025–31, 2006.

SOUZA, M. F. d. **Modelagem e Verificação De Programas De CLP Escritos Em Diagrama Ladder**. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, Departamento de Automação e Sistemas, 2010.

SVA, I. **IEEE Standard for System Verilog-Unified Hardware Design**. [S.l.], 2009.

TAHA, S. et al. Marte: UML-based Hardware Design from Modelling to Simulation. In: **FDL**. [S.l.: s.n.], 2007. p. 274–279.

TOPCASED. **TOPCASED: The Open-Source Toolkit for Critical Systems**. November 2014. <http://www.topcased.org>.

VHDL. **1076-2008 - IEEE Standard VHDL Language Reference Manual**. [S.l.], 2008.

WANG, Z. et al. A model driven development approach for implementing reactive systems in hardware. In: IEEE. **Specification, Verification and Design Languages, 2008. FDL 2008. Forum on**. [S.l.], 2008. p. 197–202.

WOOD, S. K. et al. A model-driven development approach to mapping uml state diagrams to synthesizable vhdl. **Computers, IEEE Transactions on**, IEEE, v. 57, n. 10, p. 1357–1371, 2008.

WOODCOCK, J. et al. Formal methods: Practice and experience. **ACM Computing Surveys (CSUR)**, ACM, v. 41, n. 4, p. 19, 2009.

YORK, G. et al. An integrated environment for hdl verification. In: IEEE. **Verilog HDL Conference, 1995. Proceedings., 1995 IEEE International**. [S.l.], 1995. p. 9–18.

APÊNDICE A – Códigos para o Controlador de Memória

Neste anexo é apresentado o código VHDL e o modelo FIACRE correspondente para o exemplo do controlador de memória apresentado na Seção 2 (verificação apresentada no Capítulo 6).

A.1 CÓDIGO VHDL DO CONTROLADOR DE MEMÓRIA

```

1  library ieee;
   use ieee.std_logic_1164.all;

   entity mem_ctrl is
5   port(
       clk, reset      : in  std_logic;
       mem, rw, burst  : in  std_logic;
       oe, we, we_me   : out std_logic
       );
10  end mem_ctrl;

   architecture two_seg_arch of mem_ctrl is
       type mc_state_type is (idle, read1, read2, read3,
                               read4, write);
15  signal state_reg, state_next : mc_state_type;
   begin
       -- state register
       process(clk, reset)
       begin
20         if (reset = '1') then
             state_reg <= idle;
         elsif rising_edge(clk) then
             state_reg <= state_next;
         end if;
25     end process;

       -- next-state logic and output logic
       process(state_reg, mem, rw, burst)
       begin
30         oe <= '0'; -- default value
         we <= '0';
         we_me <= '0';

         case state_reg is
35             when idle =>
                 if mem = '1' then
                     if rw = '1' then
                         state_next <= read1;
                     else
40                         state_next <= write;
                         we_me <= '1';
                     end if;
                 else
                     state_next <= idle;
45                 end if;
             when write =>
                 state_next <= idle;
                 we <= '1';
             when read1 =>
50                 if (burst = '1') then
                     state_next <= read2;
                 else
                     state_next <= idle;
                     oe <= '1';
55                 when read2 =>
                     state_next <= read3;
                     oe <= '1';
                 when read3 =>
60                 state_next <= read4;
                     oe <= '1';
                 when read4 =>
                     state_next <= idle;
                     oe <= '1';
65             end case;
         end process;
   end two_seg_arch;

```

 Código A.1 – Código VHDL do controlador de memória

 A.2 *TESTBENCH* DO CONTROLADOR DE MEMÓRIA

```

1 library ieee;
  use ieee.std_logic_1164.all;

  entity mem_ctrl_tb is
5 end entity mem_ctrl_tb;

  architecture mem_ctrl_tb_rtl of mem_ctrl_tb is

    — component ports
10  signal clk, reset    : std_logic := '0';
    signal mem, rw, burst : std_logic := '0';
    signal oe, we, we_me : std_logic;

    file stimulus: text open read_mode is "mem_ctrl_tb_stimulus.txt";
15  begin — architecture mem_ctrl_tb_rtl

    — component instantiation
    DUT: entity work.mem_ctrl
20     port map (
        clk => clk,
        reset => reset,
        mem => mem,
        rw => rw,
25     burst => burst,
        oe => oe,
        we => we,
        we_me => we_me);

30  — Stimulus generation
    stimulus_proc : process
        variable l: line;
        variable s: string(1 to 80);
        variable tmp_integer : integer;
35  begin
        while not endfile(stimulus) loop
            — read state
            readline(stimulus, l);

40            — read clk
            readline(stimulus, l);
            read(l, tmp_integer);
            if tmp_integer = 0 then
                clk <= '0';
45            else
                clk <= '1';
            end if;

            — read reset
50            readline(stimulus, l);
            read(l, tmp_integer);
            if tmp_integer = 0 then
                reset <= '0';
            else
55            reset <= '1';
            end if;

            — read mem
            readline(stimulus, l);
60            read(l, tmp_integer);
            if tmp_integer = 0 then
                mem <= '0';
            else
                mem <= '1';
65            end if;

            — read rw
            readline(stimulus, l);
  
```

```

    read(1, tmp_integer);
    if tmp_integer = 0 then
        rw <= '0';
    else
        rw <= '1';
    end if;
75
    -- read burst
    readline(stimulus, 1);
    read(1, tmp_integer);
    if tmp_integer = 0 then
80        burst <= '0';
    else
        burst <= '1';
    end if;

85    wait for 10 ns;
    end loop;

    report "Simulation_completed" severity failure;
    wait;
90    end process stimulus_proc;
end architecture mem_ctrl_tb_rtl;

```

Código A.2 – *Testbench* do controlador de memória

A.3 MODELO FIACRE DO CONTROLADOR DE MEMÓRIA

Modelo FIACRE utilizado para verificação. Esse modelo difere do apresentado na Seção 4.4 por manter o sinal de *reset* em valor constante.

```

1 /* Types */

type mc_state_type is union
  idle | read1 | read2 | read3 | read4 | writel
5 end

channel input_list is bool#bool#bool#bool#bool //channel com as entradas

type output_list is record //record com as saidas
10   oe : bool,
   we : bool,
   we_me : bool
end

15 const INIT_OUTPUT_LIST : output_list is
{
  oe = false,
  we = false,
  we_me = false
20 }

type vhdl_variables is record
  state_reg : mc_state_type,
  state_next : mc_state_type,
25 // outputs
  oe : bool,
  we : bool,
  we_me : bool,
  // inputs
30  clk : bool,
  reset : bool,
  mem : bool,
  rw : bool,
  burst : bool
35 end

const INIT_VHDL_VARIABLES : vhdl_variables is
{
  state_reg = idle,
40  state_next = idle,

```

```

oe = false ,
we = false ,
we_me = false ,
clk = false ,
45  reset = false ,
mem = false ,
rw = false ,
burst = false
}
50
/* ----- */
/* Processes */
/* ----- */
55  /* Processo gerador de evento para entrada booleana */
process input_generator [generated_inputs : out input_list]
(&output_list : read output_list) is
states generating
var clk , reset , mem , rw , burst : bool := false
init to generating
60  from generating
reset := false ;
clk , mem , rw , burst := any ;
generated_inputs!clk , reset , mem , rw , burst ;
loop
65  /* Processo que gerencia o delta ciclo */
process delta_cicle [input_list : in input_list]
(&output_list : read write output_list) is
states reading , executing
70  var vhdl : vhdl_variables := INIT_VHDL_VARIABLES ,
in_vhdl : vhdl_variables := INIT_VHDL_VARIABLES ,
pre_vhdl : vhdl_variables := INIT_VHDL_VARIABLES ,
out_vhdl : vhdl_variables := INIT_VHDL_VARIABLES ,
75  //first execution mandatory
first_time : bool := false
init to reading
from reading
80  input_list?out_vhdl.clk , out_vhdl.reset , out_vhdl.mem , out_vhdl.rw ,
out_vhdl.burst ;
to executing
from executing
85  //delta cicle execution
while (in_vhdl <> out_vhdl) or (not first_time) do
first_time := true ;
//update process input list
in_vhdl := out_vhdl ;
90  //process1
////////////////////////////////////
if (pre_vhdl.clk <> in_vhdl.clk) or
95  (pre_vhdl.reset <> in_vhdl.reset) then
if in_vhdl.reset then
out_vhdl.state_reg := idle
else
100  if (not pre_vhdl.clk) and in_vhdl.clk then
out_vhdl.state_reg := in_vhdl.state_next
end
end ;
// process2
105  //////////////////////////////////////
if (pre_vhdl.state_reg <> in_vhdl.state_reg) or
(pre_vhdl.mem <> in_vhdl.mem) or
(pre_vhdl.rw <> in_vhdl.rw) or
(pre_vhdl.burst <> in_vhdl.burst) then
out_vhdl.oe := false ;
out_vhdl.we := false ;
out_vhdl.we_me := false ;
110  if in_vhdl.state_reg = idle then
if in_vhdl.mem then
if in_vhdl.rw then
115  out_vhdl.state_next := read1
else
out_vhdl.state_next := write1 ;
out_vhdl.we_me := true
end if
120  else
out_vhdl.state_next := idle
end if

```

```

end if;
if in_vhdl.state_reg = writel then
125   out_vhdl.state_next := idle;
      out_vhdl.we       := true
end if;
if in_vhdl.state_reg = read1 then
      if in_vhdl.burst then
130         out_vhdl.state_next := read2
            else
              out_vhdl.state_next := idle
            end;
              out_vhdl.oe := true
135   end if;
      if in_vhdl.state_reg = read2 then
              out_vhdl.state_next := read3;
              out_vhdl.oe       := true
140   end if;
      if in_vhdl.state_reg = read3 then
              out_vhdl.state_next := read4;
              out_vhdl.oe       := true
            end if;
145   if in_vhdl.state_reg = read4 then
              out_vhdl.state_next := idle;
              out_vhdl.oe       := true
            end if;
            //update pre variables
150         pre_vhdl := in_vhdl
            end;
            //update signals, inputs and outputs
            vhdl := out_vhdl;
155         //write output shared variable
            output_list.oe := out_vhdl.oe;
            output_list.we := out_vhdl.we;
            output_list.we_me := out_vhdl.we_me;
            to reading
160
            /* ----- */
            /* Components */
            /* ----- */

165 /* Main component */
component mem_ctrl is
      var output_list : output_list := INIT_OUTPUT_LIST
        port input_list : input_list in [0,0]
        par * in
170          delta_cicle[input_list] (&output_list)
            || input_generator[input_list] (&output_list)
          end
end
mem_ctrl

```

Código A.3 – Modelo FIACRE do controlador de memória

APÊNDICE B - Tradução OVL

Nesta seção são apresentadas as traduções dos observadores OVL mostrados na Tabela 2.

B.1 OVL_ALWAYS

O observador **ovl_always** verifica uma expressão de bit único **test expr** a cada borda ativa de *clock*. Se **test expr** não é verdadeiro, a propriedade é considerada como violada. Além das portas comuns a todos os *assertions*, esse módulo possui adicionalmente a porta **test expr**, cuja expressão deve ser verdadeira a cada borda ativa do *clock*.

Propriedade como *assert*

Quando esse observador OVL é utilizado como propriedade para verificação, ele pode ser traduzido em @LTL e LTL com as seguintes expressões:

- @LTL : $\square(\text{test_expr})@clock$
- LTL : $\square(\text{clock} \rightarrow \text{test_expr})$

Propriedade como *assume*

Quando esse observador OVL é utilizado como restrição para entradas, o processo **input_generator** tem o seguinte formato:

```

1  process input_generator [generated_inputs : out input_list]
   (&output_list : read output_list) is
   states clk_false, clk_true
   //variables declaration
5  ...
   init to clk_false
   from clk_false
     clock := false
     //clk_false code
10  from clk_true
     clock := true;
     input2, ..., inputn := any where test_expr;
     generated_inputs!input1, ..., inputn;
     to clk_false

```

Somente a expressão **test_expr** é adicionado a cláusula *where*.

B.2 OVL_CYCLE_SEQUENCE

O observador **ovl_cycle_sequence** verifica a expressão **event sequence**, que é um vetor de bits de tamanho *num_cks* (maior que

1), a cada borda ativa de *clock* para identificar se os bits de **event sequence** se tornam verdadeiros sequencialmente em sucessivas bordas ativas de *clock*. Essa propriedade é útil para verificar o sequenciamento de eventos em um circuito.

Além das portas comuns a todos os *assertions*, esse módulo possui adicionalmente a porta **event sequence** que é conectada a um vetor de bits, onde cada um representa um evento.

A expressão pode ter dois comportamentos diferentes dependendo da configuração do atributo *generic necessary condition*. Quando esse atributo é configurado como *OVL TRIGGER ON MOST PIPE*, o observador verifica se os *num_cks-1* bits mais significativos do vetor se tornaram verdadeiros sucessivamente e, em caso positivo, na próxima borda de subida do *clock* o bit menos significativo deve ser verdadeiro senão a propriedade é violada. Para essa configuração, os equivalentes @LTL e LTL são (considerando um vetor de bit de tamanho 3, *num_cks=3*)

- @LTL : $\Box(\text{event_sequence}(2) \rightarrow \bigcirc(\neg\text{event_sequence}(1) \vee (\text{event_sequence}(1) \rightarrow X \text{event_sequence}(0))))@clock$
- LTL : $\Box(\text{clock} \rightarrow (\text{event_sequence}(2) \rightarrow \bigcirc(\neg\text{clock} \wedge ((\neg\text{event_sequence}(1)) \vee (\text{event_sequence}(1) \rightarrow \bigcirc(\neg\text{clock} \wedge (\text{clock} \wedge \text{event_sequence}(2))))))))$

Quando **necessary condition** é configurado como *OVL TRIGGER ON FIRST PIPE*, o observador verifica se o bit mais significativo do vetor é verdadeiro e, em caso positivo, nas próximas bordas de subida do *clock* os bits menos significativos devem se tornarem verdadeiros sucessivamente, senão a propriedade é violada. Para essa configuração, os equivalentes @LTL e LTL são (considerando um vetor de bit de tamanho 3, *num_cks=3*)

- @LTL : $\Box(\text{event_sequence}(2) \rightarrow \bigcirc(\text{event_sequence}(1)) \rightarrow \bigcirc(\text{event_sequence}(0)))@clock$
- LTL : $\Box(\text{clock} \rightarrow (\text{event_sequence}(2) \rightarrow \bigcirc(\neg\text{clock} \wedge (\text{clock} \wedge (\text{event_sequence}(1)))) \rightarrow \bigcirc(\neg\text{clock} \wedge (\text{clock} \wedge (\text{event_sequence}(0))))))$

Generalizando para *num_cks=n*, as expressões @LTL para *OVL TRIGGER ON MOST PIPE* e *OVL TRIGGER ON FIRST PIPE* se tornam respectivamente:

- 1.@LTL : $\Box(\text{event_sequence}(n) \rightarrow \bigcirc(\neg\text{event_sequence}(n-1) \vee (\text{event_sequence}(n-1) \rightarrow \bigcirc(\neg\text{event_sequence}(n-2) \vee (\text{event_sequence}(n-2) \rightarrow \dots(\text{event_sequence}(1) \rightarrow \bigcirc \text{event_sequence}(0)\dots))))))@clock$
- 2.@LTL : $\Box(\text{event_sequence}(n) \rightarrow \bigcirc(\text{event_sequence}(n-1)) \rightarrow \bigcirc(\text{event_sequence}(n-2))\dots \rightarrow \bigcirc(\text{event_sequence}(0)))@clock$

B.3 OVL IMPLICATION

O observador **ovl_implication** verifica a expressão de bit único **antecedent expr** a cada borda ativa de *clock*. Se **antecedent expr** é verdadeiro, então o observador verifica se a expressão **consequent expr** também é verdadeira e, em caso negativo, a propriedade é considerada como violada. Se **antecedent expr** é falso, a propriedade é considerada válida independente do valor de **consequent expr**.

Além das portas comuns, esse observador possui as seguintes portas adicionais:

- **antecedent_expr**: Expressão antecedente que é testada a cada evento de *clock*.
- **consequent_expr**: Expressão consequente que deve ser verdadeiro se *antecedent expr* for quando testada.

Propriedade como *assert*

Quando esse observador OVL é utilizado como propriedade para verificação, ele pode ser traduzido em @LTL e LTL com as seguintes expressões:

- **@LTL** : $\square(\textit{antecedent_expr} \rightarrow \textit{consequent_expr})@clock$
- **LTL** : $\square(clock \rightarrow (\textit{antecedent_expr} \rightarrow \textit{consequent_expr}))$

Propriedade como *assume*

Quando esse observador OVL é utilizado como restrição para entradas, o processo **input_generator** tem o seguinte formato:

```

1  process input_generator [generated_inputs : out input_list]
   (&output_list : read output_list) is
   states clk_false, clk_true
   //variables declaration
5  ...
   init to clk_false
   from clk_false
     clock := false
     //clk_false code
10  from clk_true
     clock := true;
     input2, ..., inputn := any where
     (not antecedent_expr) or
     (antecedent_expr and consequent_expr);
15  generated_inputs!input1, ..., inputn;
     to clk_false

```

Somente a expressão $(\neg \textit{antecedent_expr}) \vee (\textit{antecedent_expr} \wedge \textit{consequent_expr})$ é adicionado a cláusula *where*.

B.4 OVL_NEVER

O observador **ovl_never** verifica uma expressão de bit único **test expr** a cada borda ativa de *clock*. Se **test expr** é verdadeiro, a propriedade é considerada como violada.

Além das portas comuns a todos os *assertions*, esse módulo possui adicionalmente a porta **test expr**, cuja entrada deve ser falsa a cada borda ativa do *clock*.

Propriedade como *assert*

Quando esse observador OVL é utilizado como propriedade para verificação, ele pode ser traduzido em @LTL e LTL com as seguintes expressões:

- @LTL : $\square(\neg \text{test_expr})@clock$
- LTL : $\square(clock \rightarrow \neg \text{test_expr})$

Propriedade como *assume*

Quando esse observador OVL é utilizado como restrição para entradas, o processo **input_generator** tem o seguinte formato:

```

1  process input_generator [generated_inputs : out input_list]
  (&output_list : read output_list) is
  states clk_false, clk_true
  //variables declaration
5  ...
  init to clk_false
  from clk_false
    clock := false
    //clk_false code
10 from clk_true
    clock := true;
    input2, ..., inputn := any where not test_expr;
    generated_inputs!input1, ..., inputn;
    to clk_false

```

Somente a expressão $\neg \text{test_expr}$ é adicionado a cláusula *where*.

B.5 OVL_NEXT

O observador **ovl_next** verifica a expressão **start event** a cada borda ativa do *clock*. Se **start event** é verdadeiro, uma verificação é iniciada. A verificação aguarda *num cks* (parâmetro *generics*) ciclos de *clock* para então examinar o valor de **test expr** e, caso a expressão seja falsa, a propriedade foi violada, do contrário, ela é válida. Esse

observador é útil para verificar situações de causalidade no circuito, onde a ocorrência de um evento leva a alguma resposta após um tempo fixo.

Além das portas comuns, esse observador possui as seguintes portas adicionais:

- **start_event**: Expressão que identifica (juntamente com *num cks*) quando verificar **test_expr**.
- **test_expr**: Expressão que deve ser verdadeira *num cks* ciclos de *clock* depois de **start event** ser verdadeiro.

Propriedade como *assert*

O observador OVL pode ser traduzido em @LTL e LTL com as seguintes expressões (*num cks=2*):

$$\bullet @LTL : \square(\text{start_event} \rightarrow \bigcirc \bigcirc (\text{test_expr}))@clock$$

$$\bullet LTL : \square(\text{clock} \rightarrow (\text{start_event} \rightarrow \bigcirc (\neg \text{clock} U (\text{clock} \wedge \bigcirc (\neg \text{clock} U (\text{clock} \wedge \text{test_expr}))))))$$

Esse observador pode ser configurado para fazer verificações sobrepostas ou não dependendo do valor do parâmetro *generic check overlapping*. Quando esse parâmetro possui valor 0, a verificação sobreposta não é permitida, o que significa uma vez que **start expr** se tornou verdadeiro, no próximo ciclo ele deve ser falso e assim permanecer até que **test expr** seja verdadeiro. Para fazer essa verificação, a seguinte propriedade LTL é necessária (considerando *num cks=3*):

$$\bullet @LTL : \square(\text{start_event} \wedge \bigcirc (\neg \text{start_event} \wedge \bigcirc (\neg \text{start_event})))@clock$$

$$\bullet LTL : \square(\text{clock} \rightarrow (\text{start_event} \wedge \bigcirc (\neg \text{clock} U (\text{clock} \wedge (\neg \text{start_event} \wedge \bigcirc (\neg \text{clock} U (\text{clock} \wedge (\neg \text{start_event}))))))))$$

Outra configuração possível, quando o parâmetro *generic check missing start* possui valor 1, é a verificação se a expressão *test expr* se torna verdadeira sem **start event** ter sido verdadeiro anteriormente. Para essa checagem, a seguinte expressão LTL é necessária (considerando *num cks=2*):

$$\bullet @LTL : \square(\neg(\neg \text{start_event} \rightarrow \bigcirc \bigcirc \text{test_expr}))@clock$$

$$\bullet LTL : \square(\text{clock} \rightarrow \neg(\neg \text{start_event} \rightarrow \bigcirc (\neg \text{clock} U (\text{clock} \wedge \bigcirc (\neg \text{clock} U (\text{clock} \wedge \text{test_expr}))))))$$

Generalizando as três propriedades para $num\ cks=n$, temos as seguintes expressões em @LTL:

- 1.@LTL : $\Box(start_event \rightarrow \bigcirc^n(test_expr))@clock$
- 2.@LTL : $\Box(start_event \wedge [\bigcirc(\neg start_event \wedge)^{n-2} \bigcirc(\neg start_event) \dots])@clock$
- 3.@LTL : $\Box(\neg(\neg start_event \rightarrow \bigcirc^n test_expr))@clock$

Propriedade como *assume*

Quando esse observador OVL é utilizado como restrição para entradas, o processo **input_generator** tem o seguinte formato:

```

1  process input_generator [generated_inputs : out input_list]
  (&output_list : read output_list) is
  states clk_false, clk_true
  //variables declaration
5  var ovl_next_fifo : queue 1 of nat := {},
    ovl_next_cnt : nat := 0,
    ...
    init to clk_false
    from clk_false
10   clock := false
    //clk_false code
    from clk_true
    clock := true;
    //ovl_next code
15   if (ovl_next_cnt > 0) and (start_event = true) then
    ovl_next_fifo := enqueue (ovl_next_fifo, ovl_next_cnt)
    end if;
    if (ovl_next_cnt = num_cks) then
    if empty(ovl_next_fifo) then
20     ovl_next_cnt := 0
    else
    ovl_next_cnt := ovl_next_cnt - first(ovl_next_fifo) + 1;
    ovl_next_fifo := dequeue(ovl_next_fifo)
    end if
25   else
    if ovl_next_cnt > 0 then
    ovl_next_cnt := ovl_next_cnt + 1
    else
    if start_event = true then
30     ovl_next_cnt := 1
    end if
    end if
    end if;
    //////////////////////////////////////
35   input2, ..., inputn := any where
    (test_expr and (ovl_next_cnt = num_cks)) or
    (ovl_next_cnt <> num_cks);
    generated_inputs!input1, ..., inputn;
    to clk_false

```

Para utilizar o **ovl_next** como *assume*, além da adição da expressão mostrada nas linhas 36-37 a cláusula *where*, é necessário o

código auxiliar mostrado nas linhas 15-33. Se a opção **check missing start** for usada, a expressão $\neg(\text{test_expr} \wedge (\text{ovl_next_cnt} = 0))$ deve ser adicionado ao *where*. Quando o parâmetro **check overlapping** for 0, a expressão $\neg(\text{start_event} \wedge (\text{ovl_next_cnt} > 0))$ deve ser adicionada ao *where*.

B.6 OVL_ONE_HOT

O observador **ovl_one_hot** verifica o vetor de bits **test expr** a cada borda ativa do *clock* para examinar se a expressão possui apenas 1 bit verdadeiro.

Além das portas comuns a todos os *assertions*, esse módulo possui adicionalmente a porta **test expr**, que deve ser conectada a um vetor de bits onde somente um deles pode ter valor 1 (verdadeiro) a cada borda ativa do *clock*.

O observador OVL pode ser traduzido em @LTL e LTL com as seguintes expressões (considerando um vetor de tamanho 3):

- @LTL : $\square((\text{test_expr}(0) = \text{true} \wedge \text{test_expr}(1) = \text{false} \wedge \text{test_expr}(2) = \text{false}) \vee (\text{test_expr}(0) = \text{false} \wedge \text{test_expr}(1) = \text{true} \wedge \text{test_expr}(2) = \text{false}) \vee (\text{test_expr}(0) = \text{false} \wedge \text{test_expr}(1) = \text{false} \wedge \text{test_expr}(2) = \text{true}))@clock$
- LTL : $\square(\text{clock} \rightarrow ((\text{test_expr}(0) = \text{true} \wedge \text{test_expr}(1) = \text{false} \wedge \text{test_expr}(2) = \text{false}) \vee (\text{test_expr}(0) = \text{false} \wedge \text{test_expr}(1) = \text{true} \wedge \text{test_expr}(2) = \text{false}) \vee (\text{test_expr}(0) = \text{false} \wedge \text{test_expr}(1) = \text{false} \wedge \text{test_expr}(2) = \text{true})))$

B.7 OVL_RANGE

O observador **ovl_range** examina a expressão **test expr** a cada borda ativa do *clock* para verificar se seu valor seja maior ou igual a *min* e menor ou igual a *max*. Caso o valor esteja fora dos limites a propriedade é considerada como violada. Esse observador é útil para garantir se estruturas do circuito (contadores por exemplo) mantêm seus valores em intervalos desejados.

Além das portas comuns a todos os *assertions*, esse módulo possui adicionalmente a porta **test expr**, cuja entrada deve ter valor maior ou igual a *min* e menor ou igual a *max*. Os valores *max* e *min* são parâmetros *generics*.

A expressão **test expr** é um vetor de bit no OVL, mas na atual tradução iremos restringir para inteiros e naturais visto que operadores

de comparação entre *arrays* ainda devem ser criados para o FIACRE. As traduções para @LTL e LTL estão abaixo:

•@LTL : $\Box((test_expr \geq min) \wedge (test_expr \leq max))@clock$

•LTL : $\Box(clock \rightarrow ((test_expr \geq min) \wedge (test_expr \leq max)))$

A definição de restrição de intervalo de valores para entradas de tipo inteiro ou natural não é realizada com esse observador, mas sim do modo explicado na Seção 5.4

B.8 OVL_ZERO_ONE_HOT

O observador **ovl_zero_one_hot** verifica o vetor de bits **test expr** a cada borda ativa do *clock* para examinar se a expressão possui apenas 1 bit verdadeiro ou nenhum. Esse observador é útil para verificar circuitos de controle, lógica de árbitros ou circuitos de habilitação, onde é necessário liberar acesso a um determinado circuito a apenas um circuito por vez.

Além das portas comuns a todos os *assertions*, esse módulo possui adicionalmente a porta **test expr**, que é a entrada para o vetor que deve possuir nenhum ou apenas um bit com valor 1 (verdadeiro) a cada borda ativa do *clock*.

O observador OVL pode ser traduzido em @LTL e LTL com as seguintes expressões (considerando um vetor de tamanho 3):

•@LTL : $\Box((test_expr(0) = true \wedge test_expr(1) = false \wedge test_expr(2) = false) \vee (test_expr(0) = false \wedge test_expr(1) = true \wedge test_expr(2) = false) \vee (test_expr(0) = false \wedge test_expr(1) = false \wedge test_expr(2) = true) \vee (test_expr(0) = false \wedge test_expr(1) = false \wedge test_expr(2) = false))@clock$

•LTL : $\Box(clock \rightarrow ((test_expr(0) = true \wedge test_expr(1) = false \wedge test_expr(2) = false) \vee (test_expr(0) = false \wedge test_expr(1) = true \wedge test_expr(2) = false) \vee (test_expr(0) = false \wedge test_expr(1) = false \wedge test_expr(2) = true) \vee (test_expr(0) = false \wedge test_expr(1) = false \wedge test_expr(2) = false)))$

B.9 OVL_WIDTH

O observador **ovl_width** examina a expressão **test expr** a cada borda ativa do *clock*. Se o valor de **test expr** é verdadeiro, o observador realiza os seguintes passos:

- 1.A não ser que esteja desativado por *min cks* ter valor 0, uma verificação de mínimo é iniciada. O observador examina **test expr** a cada borda ativa do *clock* e se o valor não for verdadeiro, a checagem de mínimo falha e, conseqüentemente, a propriedade também. Após *min_cks-1* ciclos de *clock*, a verificação de mínimo é encerrada.
- 2.A não ser que esteja desativado por *max cks* ter valor 0, uma verificação de máximo é iniciada. O observador examina **test expr** a cada borda ativa do *clock* e se o valor não for falso após *max cks* ciclos (contando do momento que **test expr** se tornou verdadeiro), a checagem de máximo falha e, conseqüentemente, a propriedade também.
- 3.O observador retorna a examinar **test expr** no ciclo seguinte. Em particular, se **test expr** for verdadeiro, uma nova verificação é iniciada.

Além das portas comuns a todos os *assertions*, esse módulo possui adicionalmente a porta **test expr**, que é a entrada para o bit que deve manter valor 1 (verdadeiro) por pelo menos *min cks* ciclos e no máximo *max cks* ciclos após ser amostrado como verdadeiro. *Max cks* e *min cks* são os parâmetros *generics* adicionais desse módulo.

Propriedade como *assert*

A verificação de mínimo pode ser em @LTL e LTL com as seguintes expressões (considerando *min cks=2*):

- @LTL : $\square(((\neg \text{test.expr}) \wedge \bigcirc \text{test.expr}) \rightarrow \bigcirc (\text{test.expr} \wedge \bigcirc \text{test.expr}))@clock$
- LTL : $\square(\text{clock} \rightarrow (((\neg \text{test.expr}) \wedge \bigcirc (\neg \text{clock} U (\text{clock} \wedge \text{test.expr}))) \rightarrow \bigcirc (\neg \text{clock} U (\text{clock} \wedge \text{test.expr} \wedge \bigcirc (\neg \text{clock} U (\text{clock} \wedge \text{test.expr}))))))$

A checagem de mínimo pode ser generalizada (*min cks=n* (*n* > 0)) para a expressão @LTL:

- @LTL : $\square(((\neg \text{test.expr}) \wedge \bigcirc \text{test.expr}) \rightarrow \bigcirc (\text{test.expr} [\wedge \bigcirc (\text{test.expr}]^{n-1} \dots))@clock$

A verificação de máximo pode ser em @LTL e LTL com as seguintes expressões (considerando *max cks=2*):

- @LTL : $\square \neg((\neg \text{test.expr}) \wedge \bigcirc (\text{test.expr} \wedge \bigcirc (\text{test.expr} \wedge \bigcirc \text{test.expr})))@clock$

lizado na comparação da linha 19 deve ser igual a $max_cks + 1$ para $max_cks > 0$, do contrário deve ser $min_cks + 1$.

B.10 OVL_WINDOW

O observador **ovl_window** examina a expressão **start event** a cada borda ativa do *clock* para determinar se deve abrir uma janela de eventos no próximo ciclo. Se **start event** é amostrado como verdadeiro, nos ciclos seguintes o observador passa a acompanhar os valores de **end event** e **test expr**. Caso **test expr** seja falso, a propriedade falhou. Se a expressão **end event** for verdadeira, o observador fecha a janela de eventos e retorna a monitorar a **start event** no próximo ciclo de *clock*. Esse observador pode ser útil por exemplo para verificar condições de sincronismo do circuito que necessitam que uma dado permaneça estável após um evento de disparo.

Além das portas comuns, o observador possui as seguintes portas adicionais:

- **start_event**: Expressão que abre a janela de eventos.
- **test_expr**: Expressão que deve ser verdadeira na janela de eventos.
- **end_event**: Expressão que encerra a janela de evento.

Propriedade como *assert*

O observador OVL pode ser traduzido em @LTL e LTL com as seguintes expressões:

- **@LTL** : $\Box(((\neg start_event) \wedge \bigcirc (start_event \wedge \neg end_event)) \rightarrow \bigcirc \bigcirc (test_expr \ W \ (test_expr \wedge \ end_event)))@clock$
- **LTL** : $\Box(clock \rightarrow (((\neg start_event) \wedge \bigcirc (\neg clock \ U \ (clock \wedge \ start_event \wedge \neg end_event)))) \rightarrow \bigcirc (\neg clock \ U \ (clock \wedge \bigcirc (\neg clock \ U \ (clock \wedge (\neg((clock \rightarrow \neg end_event) \ U \ (clock \wedge \neg(test_expr \vee (test_expr \wedge end_event))))))))))$

Propriedade como *assume*

Quando esse observador OVL é utilizado como restrição para entradas, o processo **input_generator** tem o seguinte formato:

```
1 process input_generator [generated_inputs : out input_list]
  (&output_list : read output_list) is
```

```

states clk_false, clk_true
//variables declaration
5 var ovl_window_open : bool := false,
    pre_ovl_window : bool := false,
    ...
    init to clk_false
    from clk_false
10   clock := false
    //clk_false code
    from clk_true
    clock := true;
    //ovl_window code
15   if end_event then
        ovl_window_open := false
    else
        if (not pre_ovl_window) and start_event then
20     ovl_window_open := true
        end if
    end if;
    pre_ovl_window := start_event;
    //////////////////////////////////////
25   input2, ..., inputn := any where
    (not ovl_window_open)
    or
    (ovl_window_open and test_expr);
    generated_inputs!input1, ..., inputn;
    to clk_false

```

Para utilizar o **ovl_window** como *assume*, além da adição da expressão mostrada nas linhas 25-27 à clausula *where*, é necessário o código auxiliar mostrado nas linhas 15-22. Duas variáveis extras são criadas para auxiliar no processamento do observador, destacadas nas linhas 5-6.

B.11 OVL_UNCHANGE

A tradução desse observador só foi realizada para restrições nas entradas. Ele verifica a expressão **start_event** a cada borda ativa do *clock* para determinar se deve observar mudanças no valor de **test_expr**. Se **start_event** é amostrado como verdadeiro, o observador examina **test_expr** por *num cks* ciclos de *clock* seguidos, verificando se houve mudança de valor nesse período. Se houve mudança, a propriedade falha.

Além das portas comuns, esse observador possui as seguintes portas adicionais:

- **start_event**: Expressão que identifica (juntamente com **action on new start**) quando verificar **test_expr**.
- **test_expr**: Expressão que não deve ter seu valor modificado por **num cks** (parâmetro de configuração) ciclos de *clock* depois de

start event ser verdadeiro, a menos que a verificação seja interrompida devido a um evento de início válido.

Uma vez que a expressão **start_event** se tornou verdadeira, a verificação começa, no entanto esse valor pode ser mantido ou se tornar verdadeiro no meio da verificação. O modo que o observador trata **start_event** durante uma janela de verificação depende do parâmetro **action on new start**. Quando esse parâmetro é configurado como *OVL_IGNORE_ON_NEW_START*, o observador ignora o valor de **start_event** quando estiver em uma janela de verificação. Para essa configuração o **input_generator** tem o seguinte formato:

```

1  process input_generator [generated_inputs : out input_list]
   (&output_list : read output_list) is
   states clk_false, clk_true
   //variables declaration
5  var ovl_unchange_pre : fiacre_type := init_value,
     ovl_unchange_cnt : nat := 0,
     ...
   init to clk_false
   from clk_false
10  clock := false
   //clk_false code
   from clk_true
   clock := true;
   //ovl_unchange code
15  ovl_unchange_pre := test_expr;
   if ovl_unchange_cnt > 0 then
     if ovl_unchange_cnt = num_cks then
       ovl_unchange_cnt := 0
     else
20     ovl_unchange_cnt := ovl_unchange_cnt + 1
     end if
   else
     if start_event = true then
       ovl_unchange_cnt := 1
25     end if
   end if
   //////////////////////////////////////
   input2, ..., inputn := any where
   not ((ovl_unchange_pre <> test_expr) and
30  (ovl_unchange_cnt > 0));
   generated_inputs!input1, ..., inputn;
   to clk_false

```

Além da adição da expressão mostrada nas linhas 29-30 à cláusula *where*, é necessário o código auxiliar mostrado nas linhas 15-26. Duas variáveis extras são criadas para auxiliar no processamento do observador, destacadas nas linhas 5-6.

Outra configuração possível é para **action on new start** como *OVL_RESET_ON_NEW_START*. Nessa configuração, toda vez que **start event** é verdadeiro, a análise de modificação de **test expr** é reiniciada. Para essa configuração, o modelo do **input_generator** é descrito

como:

```

1  process input_generator [generated_inputs : out input_list]
   (&output_list : read output_list) is
   states clk_false, clk_true
   //variables declaration
5  var ovl_unchange_pre : fiacre_type := init_value,
      ovl_unchange_cnt : nat := 0,
      ...
   init to clk_false
   from clk_false
10  clock := false
      //clk_false code
   from clk_true
      clock := true;
      //ovl_unchange code
15  if start_event = true then
      ovl_unchange_cnt := 1
      else
      if ovl_unchange_cnt > 0 then
      if ovl_unchange_cnt = num_cks then
20  ovl_unchange_cnt := 0
          else
          ovl_unchange_cnt := ovl_unchange_cnt + 1
          end if
      end if
25  end if;
      pre_ovl_unchange := test_expr;
      //////////////////////////////////////
      input2, ..., inputn := any where
30  (not ((ovl_unchange_pre <> f_v_a) and
      (ovl_unchange_cnt > 0))) or start_event;
      generated_inputs!input1, ..., inputn;
      to clk_false

```

APÊNDICE C - Códigos da Função de Proteção PTOV

C.1 CÓDIGO VHDL DA FUNÇÃO DE PROTEÇÃO PTOV

O código VHDL da função de proteção de sobretensão (PTOV) está abaixo. A parte referente a proteção para as fases A e B foram omitidas pois não foram utilizadas no exemplo.

```

1 library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
  use ieee.std_logic_arith.all;
5
  entity ptov_ptof is
    port (
      -- system signals
10      sysclk : in std_logic;
      reset_n : in std_logic;

      -- interface signals
      reset_op : in std_logic;
      enable : in std_logic;           -- module enable
15      sync_a : in std_logic;
      sync_b : in std_logic;
      sync_c : in std_logic;
      f_v_a : in std_logic_vector(31 downto 0);
      f_v_b : in std_logic_vector(31 downto 0);
20      f_v_c : in std_logic_vector(31 downto 0);
      comp_data : in std_logic_vector(31 downto 0);
      reset_data : in std_logic_vector(31 downto 0);
      dropout_data : in std_logic_vector(31 downto 0);
      comp_counter : in std_logic_vector(31 downto 0);
25      sense_o : out std_logic_vector(2 downto 0);
      op_o : out std_logic_vector(2 downto 0);
    );
  end ptov_ptof;
30
  architecture ptov_ptof_rtl of ptov_ptof is
    type STATE_PTOVF_TYPE is (WAIT_ENABLE_STATE, WAIT_SYNC_STATE,
      DECIDE_R_S, UPDATE_RESET_STATE, DECIDE_RESET_STATE,
      UPDATE_TIMER_STATE, DECIDE_OP_TIMER_STATE, D1);
35
    attribute ENUMENCODING : string;
    attribute ENUMENCODING of STATE_PTOVF_TYPE : type is
      "000_001_010_011_100_101_110_111";
40
    signal state_core : STATE_PTOVF_TYPE;
    signal state_core_next : STATE_PTOVF_TYPE;

    signal sense_timer_reg : std_logic_vector(2 downto 0);
    signal sense_timer_next : std_logic_vector(2 downto 0);
    signal sense_timer_2_reg : std_logic;
45    signal sense_timer_2_next : std_logic;
    signal timer_reg : natural range 0 to 2147483647;
    signal timer_next : natural range 0 to 2147483647;

    -- signals
50    signal dropout_reg : std_logic_vector(31 downto 0);
    signal reset_data_reg : std_logic_vector(31 downto 0);
    signal op_timer_reg : std_logic_vector(2 downto 0);
    signal op_timer_next : std_logic_vector(2 downto 0);
55
    -- Signals das Fases A e B
    ...

  begin
    op_o <= op_timer_reg;
    sense_o <= sense_timer_reg;
    timer_natural <= CONV_INTEGER(comp_counter);
    timer_reset_natural <= CONV_INTEGER(reset_data);
60

65    process (enable, reset_n, reset_op, sysclk) is
    begin
      if ((reset_n = '0') or (reset_op = '1')
        or (enable = '0')) then
        state_core <= WAIT_ENABLE_STATE;
        timer_reg <= 0;
70        timer_reset_reg <= 0;

```

```

    op_timer_reg(0)    <= '0';
    sense_timer_reg(0) <= '0';
    elsif rising_edge(sysclk) then
75      state_core      <= state_core_next;
        timer_reg       <= timer_next;
        timer_reset_reg <= timer_reset_next;
        op_timer_reg(0) <= op_timer_next(0);
        sense_timer_reg(0) <= sense_timer_next(0);
80    end if;
end process;

process (enable, f_v_a, dropout_data, comp_data,
85      op_timer_reg(0), sense_timer_reg(0), state_core,
        sync_a, timer_reset_natural,
        timer_natural, timer_reg, timer_reset_reg) is
begin
    state_core_next <= state_core;
    timer_next      <= timer_reg;
90    timer_reset_next <= timer_reset_reg;
    sense_timer_next(0) <= sense_timer_reg(0);
    op_timer_next(0) <= op_timer_reg(0);
    case state_core is
    when WAITENABLE_STATE =>
95      if (enable = '1') then
          state_core_next <= WAIT_SYNC_STATE;
          timer_next      <= 0;
        end if;
    when WAIT_SYNC_STATE =>
100     if (sync_a = '1') then
          -- entrada_a_reg <= f_v_a;
          state_core_next <= DECIDE_R_S;
        end if;
    when DECIDE_R_S =>
105     if (f_v_a < dropout_data) then
          timer_next      <= 0;
          sense_timer_next(0) <= '0';
          state_core_next <= UPDATE_RESET_STATE;
        elsif ((f_v_a > comp_data) or (timer_reg > 0))
110         and (op_timer_reg(0) = '0') then
          timer_reset_next <= 0;
          state_core_next <= UPDATE_TIMER_STATE;
        else
          state_core_next <= DECIDE_OP_TIMER_STATE;
115     end if;
    when UPDATE_RESET_STATE =>
          timer_reset_next <= timer_reset_reg + 1;
          state_core_next <= DECIDE_RESET_STATE;
    when DECIDE_RESET_STATE =>
120     if (timer_reset_reg > timer_reset_natural) then
          op_timer_next(0) <= '0';
        end if;
          state_core_next <= WAIT_SYNC_STATE;
    when UPDATE_TIMER_STATE =>
125     sense_timer_next(0) <= '1';
          timer_next      <= timer_reg + 1;
          state_core_next <= DECIDE_OP_TIMER_STATE;
    when DECIDE_OP_TIMER_STATE =>
130     if (timer_reg > timer_natural) then
          op_timer_next(0) <= '1';
        end if;
          state_core_next <= WAIT_SYNC_STATE;
    when others => null;
    end case;
135 end process;

-- Código das Fases B e C
...
140 end pto_v_ptof_rtl;

```

Código C.1 – Código VHDL da PTOV

A verificação do código da PTOV, conforme descrito na Seção 6.2, encontrou um erro na verificação de uma das propriedades. O erro foi corrigido ao alterar o código das linhas 104-115 pelo seguinte:

```

1  when DECIDE_R_S =>
    if (f_v_a < dropout_data) then

```

```

        sense_timer_next(0) <= '0';
    else
5       if (f_v_a > comp_data) then
            sense_timer_next(0) <= '1';
        end if;
    end if;
10      if (f_v_a < dropout_data) then
            timer_next      <= 0;
            state_core_next <= UPDATE_RESET_STATE;
        elsif (((f_v_a > comp_data) or (timer_reg > 0))
15             and (op_timer_reg(0) = '0')) then
            timer_reset_next <= 0;
            state_core_next <= UPDATE_TIMER_STATE;
        else
            state_core_next <= DECIDE_OP_TIMER_STATE;
        end if;

```

C.2 TESTBENCH DA FUNÇÃO DE PROTEÇÃO PTOV

Código do *testbench* VHDL utilizado para visualização de contraexemplos do exemplo da PTOV apresentado na Seção 6.2.

```

1  library ieee;
   use ieee.std_logic_1164.all;
   use ieee.numeric_std.all;
   use std.textio.all;
2
   entity ptov_tb is
   end entity ptov_tb;
3
10  architecture ptov_tb_rtl of ptov_tb is
    -- component ports
    signal sysclk : std_logic;
    signal reset_n : std_logic;
15    signal reset_op : std_logic;
    signal enable : std_logic;
    signal sync_a : std_logic;
    signal sync_b : std_logic;
    signal sync_c : std_logic;
20    signal f_v_a : std_logic_vector(31 downto 0);
    signal f_v_b : std_logic_vector(31 downto 0);
    signal f_v_c : std_logic_vector(31 downto 0);
    signal comp_data : std_logic_vector(31 downto 0);
    signal reset_data : std_logic_vector(31 downto 0);
25    signal dropout_data : std_logic_vector(31 downto 0);
    signal comp_counter : std_logic_vector(31 downto 0);
    signal sense_o : std_logic_vector(2 downto 0);
    signal op_o : std_logic_vector(2 downto 0);
30
    -- file with stimulus
    file stimulus: text open read_mode is "ptov_tb_stimulus.txt";
31
   begin
35     sync_b <= '0';
     sync_c <= '0';
     f_v_b <= (others => '0');
     f_v_c <= (others => '0');
40
    -- component instantiation
    DUT: entity work.ptov
        port map (
45         sysclk => sysclk,
         reset_n => reset_n,
         reset_op => reset_op,
         enable => enable,
         sync_a => sync_a,
         sync_b => sync_b,
         sync_c => sync_c,
50         f_v_a => f_v_a,
         f_v_b => f_v_b,
         f_v_c => f_v_c,

```

```

    comp_data => comp_data,
    reset_data => reset_data,
55    dropout_data => dropout_data,
    comp_counter => comp_counter,
    sense_o => sense_o,
    op_o => op_o);

60  -- Stimulus generation
    stimulus_proc : process
        variable l; line;
        variable s; string(1 to 80);
        variable tmp_integer : integer;
65  begin
        while not endfile(stimulus) loop
            -- read state
            readline(stimulus, l);

70            -- read sysclk
            readline(stimulus, l);
            read(l, tmp_integer);
            if tmp_integer = 0 then
                sysclk <= '0';
75            else
                sysclk <= '1';
            end if;

80            -- read reset_n
            readline(stimulus, l);
            read(l, tmp_integer);
            if tmp_integer = 0 then
                reset_n <= '0';
85            else
                reset_n <= '1';
            end if;

            -- read reset_op
            readline(stimulus, l);
90            read(l, tmp_integer);
            if tmp_integer = 0 then
                reset_op <= '0';
            else
                reset_op <= '1';
95            end if;

            -- read enable
            readline(stimulus, l);
100            read(l, tmp_integer);
            if tmp_integer = 0 then
                enable <= '0';
            else
                enable <= '1';
105            end if;

            -- read sync_a
            readline(stimulus, l);
            read(l, tmp_integer);
            if tmp_integer = 0 then
                sync_a <= '0';
110            else
                sync_a <= '1';
            end if;

115            -- read comp_data
            readline(stimulus, l);
            read(l, tmp_integer);
            comp_data <= std_logic_vector(to_unsigned
                (tmp_integer,32));

120            -- read reset_data
            readline(stimulus, l);
            read(l, tmp_integer);
            reset_data <= std_logic_vector(to_unsigned
                (tmp_integer,32));

125            -- read dropout_data
            readline(stimulus, l);
            read(l, tmp_integer);
            dropout_data <= std_logic_vector(to_unsigned
                (tmp_integer,32));

130            -- read comp_counter
            readline(stimulus, l);
            read(l, tmp_integer);
            comp_counter <= std_logic_vector(to_unsigned
                (tmp_integer,32));

```

```

135     — read f_v_a
        readline(stimulus, 1);
        read(1, tmp_integer);
        f_v_a <= std_logic_vector(to_unsigned
140           (tmp_integer, 32));

        wait for 10 ns;
        end loop;

        report "Simulation_completed" severity failure;
145     wait;
    end process stimulus_proc;
end architecture ptov_tb_rtl;

```

Código C.2 – Código VHDL do *testbench* para a PTOV

C.3 MODELO FIACRE DA FUNÇÃO DE PROTEÇÃO PTOV

```

1  /* Types */
   const timer_N : nat is 7

   const vector_size : nat is 15

   channel input_list is bool#bool#bool#bool#bool#nat#nat#nat#nat#nat

   type STATE_PTOVF_TYPE is union
       WAIT_ENABLE_STATE
10  | WAIT_SYNC_STATE
       | DECIDE_R_S
       | UPDATE_RESET_STATE
       | DECIDE_RESET_STATE
       | UPDATE_TIMER_STATE
15  | DECIDE_OP_TIMER_STATE
   end

   type output_list is record //record com as saidas
       sense_o : bool,
20   op_o : bool
   end

   const ZERO_OUTPUT_LIST : output_list is
25   {
       sense_o = false,
       op_o = false
   }

   type vhdl_variables is record
30   state_core : STATE_PTOVF_TYPE,
       state_core_next : STATE_PTOVF_TYPE,
       sense_timer_reg : bool,
       sense_timer_next : bool,
35   timer_reg : nat,
       timer_next : nat,
       timer_natural : nat,
       timer_reset_reg : nat,
       timer_reset_next : nat,
40   timer_reset_natural : nat,
       op_timer_reg : bool,
       op_timer_next : bool,
       //outputs
       sense_o : bool,
       op_o : bool,
45   //inputs
       sysclk : bool,
       reset_n : bool,
       reset_op : bool,
       enable : bool,
50   sync_a : bool,
       f_v_a : nat,
       comp_data : nat,
       reset_data : nat,
       dropout_data : nat,
55   comp_counter : nat
   end

```

```

const INIT_VHDL_VARIABLES : vhdl_variables is
{
60   state_core = WAIT_ENABLE_STATE,
      state_core_next = WAIT_ENABLE_STATE,
      sense_timer_reg = false,
      sense_timer_next = false,
      timer_reg = 0,
65   timer_next = 0,
      timer_natural = 0,
      timer_reset_reg = 0,
      timer_reset_next = 0,
      timer_reset_natural = 0,
70   op_timer_reg = false,
      op_timer_next = false,
      //outputs
      sense_o = false,
      op_o = false,
75   //inputs
      sysclk = false,
      reset_n = false,
      reset_op = false,
      enable = false,
80   sync_a = false,
      f_v_a = 0,
      comp_data = 0,
      reset_data = 0,
      dropout_data = 0,
85   comp_counter = 0
}

/* ----- */
/* Processes */
90 /* ----- */

process input_generator [generated_inputs : out input_list]
(&output_list : read output_list) is
states clk_false, clk_true
var sysclk, reset_n, reset_op, enable, sync_a : bool := false,
95   f_v_a : 0..vector_size := 0,
      comp_data, reset_data, dropout_data, comp_counter : nat := 0,
      //ovl_next
      ovl_next_fifo : queue 1 of nat := {},
100   ovl_next_cnt : nat := 0,
      //ovl_width
      pre_ovl_width : bool := false,
      ovl_width_cnt : nat := 0,
      //ovl_unchange
      pre_ovl_unchange : 0..vector_size := 0,
105   ovl_unchange_cnt : nat := 0
init to clk_false
from clk_false
  sysclk := false;
110   reset_n := true;
      reset_op := false;
      enable := true;
      comp_data := 12; //pickup value
      reset_data := 2; //reset time
      dropout_data := 9; //dropout value
115   comp_counter := 4; //pickup time
      generated_inputs!sysclk, reset_n, reset_op, enable, sync_a,
      f_v_a, comp_data, reset_data, dropout_data, comp_counter;
to clk_true
from clk_true
120   sysclk := true;
      // ovl_unchange (start_event = sync_a, test_expr = f_v_a)
      if sync_a then
        ovl_unchange_cnt := 1
      else
125   if ovl_unchange_cnt > 0 then
        if ovl_unchange_cnt = 10 then
          ovl_unchange_cnt := 0
        else
          ovl_unchange_cnt := ovl_unchange_cnt + 1
        end if
      end if;
      pre_ovl_unchange := f_v_a;
      // ovl_width (min_cks=max_cks=9, test_expr = not sync_a)
135   if (pre_ovl_width = false) and (not sync_a) then
      ovl_width_cnt := 1
    else
      if ovl_width_cnt > 0 then

```



```

140         if (ovl_width_cnt = 10) or (pre_ovl_width = false) then
            ovl_width_cnt := 0
        else
            ovl_width_cnt := ovl_width_cnt + 1
        end if
    end if
145 end if;
pre_ovl_width := not sync_a;
// ovl_next (start_event = sync_a, test_expr = not sync_a, num_cks = 1)
if (ovl_next_cnt > 0) and sync_a then
    ovl_next_fifo := enqueue(ovl_next_fifo, ovl_next_cnt)
150 end if;
if (ovl_next_cnt = 1) then
    if empty(ovl_next_fifo) then
        ovl_next_cnt := 0
    else
155 ovl_next_cnt := ovl_next_cnt - first(ovl_next_fifo) + 1;
        ovl_next_fifo := dequeue(ovl_next_fifo)
    end if
else
    if ovl_next_cnt > 0 then
160 ovl_next_cnt := ovl_next_cnt + 1
    else
        if sync_a then
            ovl_next_cnt := 1
        end if
165 end if
end if;
////////////////////////////////////
sync_a, f_v_a := any where (
170 ((not ((pre_ovl_unchange <> f_v_a) and
    (ovl_unchange_cnt > 0))) or sync_a)
and
    (not ((ovl_width_cnt <= 9) and (ovl_width_cnt > 0) and sync_a))
and
175 (not ((ovl_width_cnt > 9) and (not sync_a)))
and
    (((not sync_a) and (ovl_next_cnt = 1)) or (ovl_next_cnt <> 1))
);
generated_inputs!sysclk, reset_n, reset_op, enable, sync_a,
180 f_v_a, comp_data, reset_data, dropout_data, comp_counter;
to clk_false

/* Proceso que gerencia o delta ciclo */
process delta_cicle [input_list : in input_list]
    (&output_list : read write output_list) is
185 states reading, executing
    var vhdl : vhdl_variables := INIT_VHDL_VARIABLES,
        in_vhdl : vhdl_variables := INIT_VHDL_VARIABLES,
        pre_vhdl : vhdl_variables := INIT_VHDL_VARIABLES,
        out_vhdl : vhdl_variables := INIT_VHDL_VARIABLES,
190
        //first execution mandatory
        first_time : bool := false
    init to reading

195 from reading
        input_list?out_vhdl.sysclk, out_vhdl.reset_n, out_vhdl.reset_op,
        out_vhdl.enable, out_vhdl.sync_a, out_vhdl.f_v_a,
        out_vhdl.comp_data, out_vhdl.reset_data,
        out_vhdl.dropout_data, out_vhdl.comp_counter;
200 to executing

    from executing
        //delta cycle execution
        while (in_vhdl <> out_vhdl) or (not first_time) do
205 first_time := true;
            //update process input list
            in_vhdl := out_vhdl;

            //process1
210 out_vhdl.op_o := in_vhdl.op_timer_reg;
            //process2
            out_vhdl.sense_o := in_vhdl.sense_timer_reg;
            //process3
215 out_vhdl.timer_natural := in_vhdl.comp_counter;
            //process4
            out_vhdl.timer_reset_natural := in_vhdl.reset_data;
220

```

```

// process5
////////////////////////////////////////////////////////////////////
if (pre_vhdl.sysclk <= in_vhdl.sysclk) or
225 (pre_vhdl.reset_n <= in_vhdl.reset_n) or
(pre_vhdl.enable <= in_vhdl.enable) or
(pre_vhdl.reset_op <= in_vhdl.reset_op) then
    if ((in_vhdl.reset_n = false) or
        (in_vhdl.reset_op = true) or
        (in_vhdl.enable = false)) then
230 out_vhdl.state_core := WAIT_ENABLE_STATE;
out_vhdl.timer_reg := 0;
out_vhdl.timer_reset_reg := 0;
out_vhdl.op_timer_reg := false;
out_vhdl.sense_timer_reg := false
235 else
    if (not pre_vhdl.sysclk) and in_vhdl.sysclk then
out_vhdl.state_core := in_vhdl.state_core_next;
out_vhdl.timer_reg := in_vhdl.timer_next;
out_vhdl.timer_reset_reg := in_vhdl.timer_reset_next;
240 out_vhdl.op_timer_reg := in_vhdl.op_timer_next;
out_vhdl.sense_timer_reg := in_vhdl.sense_timer_next
    end if
end if
245 end;
// process6
////////////////////////////////////////////////////////////////////
if (pre_vhdl.enable <= in_vhdl.enable) or
(pre_vhdl.f.v.a <= in_vhdl.f.v.a) or
250 (pre_vhdl.dropout_data <= in_vhdl.dropout_data) or
(pre_vhdl.comp_data <= in_vhdl.comp_data) or
(pre_vhdl.op_timer_reg <= in_vhdl.op_timer_reg) or
(pre_vhdl.sense_timer_reg <= in_vhdl.sense_timer_reg) or
(pre_vhdl.state_core <= in_vhdl.state_core) or
255 (pre_vhdl.sync_a <= in_vhdl.sync_a) or
(pre_vhdl.timer_reset_natural <= in_vhdl.timer_reset_natural)
or (pre_vhdl.timer_natural <= in_vhdl.timer_natural) or
(pre_vhdl.timer_reg <= in_vhdl.timer_reg) or
(pre_vhdl.timer_reset_reg <= in_vhdl.timer_reset_reg) then
260 out_vhdl.state_core_next := in_vhdl.state_core;
out_vhdl.timer_next := in_vhdl.timer_reg;
out_vhdl.timer_reset_next := in_vhdl.timer_reset_reg;
out_vhdl.sense_timer_next := in_vhdl.sense_timer_reg;
out_vhdl.op_timer_next := in_vhdl.op_timer_reg;

265 if in_vhdl.state_core = WAIT_ENABLE_STATE then
    if (in_vhdl.enable = true) then
out_vhdl.state_core_next := WAIT_SYNC_STATE;
out_vhdl.timer_next := 0
    end if
270 end if;

    if in_vhdl.state_core = WAIT_SYNC_STATE then
        if (in_vhdl.sync_a = true) then
275 out_vhdl.state_core_next := DECIDE_R_S
        end if
    end if;

    if in_vhdl.state_core = DECIDE_R_S then
        if (in_vhdl.f.v.a < in_vhdl.dropout_data) then
280 out_vhdl.timer_next := 0;
out_vhdl.sense_timer_next := false;
out_vhdl.state_core_next := UPDATE_RESET_STATE
        else
            if (((in_vhdl.f.v.a > in_vhdl.comp_data) or
                (in_vhdl.timer_reg > 0)) and
                (in_vhdl.op_timer_reg = false)) then
285 out_vhdl.timer_reset_next := 0;
out_vhdl.state_core_next := UPDATE_TIMER_STATE
            else
290 out_vhdl.state_core_next := DECIDE_OP_TIMER_STATE
            end if
        end if
    end if;

295 if in_vhdl.state_core = UPDATE_RESET_STATE then
        if (in_vhdl.timer_reset_reg < timer_N) then
out_vhdl.timer_reset_next := in_vhdl.timer_reset_reg + 1
        end if;
out_vhdl.state_core_next := DECIDE_RESET_STATE
300 end if;

    if in_vhdl.state_core = DECIDE_RESET_STATE then

```

```

        if (in_vhdl.timer_reset_reg > in_vhdl.timer_reset_natural) then
305      out_vhdl.op.timer_next := false
        end if;
        out_vhdl.state_core_next := WAIT_SYNC_STATE
    end if;

    if in_vhdl.state_core = UPDATE_TIMER_STATE then
310      out_vhdl.sense_timer_next := true;
        if (in_vhdl.timer_reg < timer_N) then
            out_vhdl.timer_next := in_vhdl.timer_reg + 1
        end if;
315      out_vhdl.state_core_next := DECIDE_OP_TIMER_STATE
    end if;

    if in_vhdl.state_core = DECIDE_OP_TIMER_STATE then
        if (in_vhdl.timer_reg > in_vhdl.timer_natural) then
320          out_vhdl.op.timer_next := true
        end if;
        out_vhdl.state_core_next := WAIT_SYNC_STATE
    end if
    end if
    //update pre variables
325  pre_vhdl := in_vhdl
end;
//update signals, inputs and outputs
330  vhdl := out_vhdl;
//write output shared variable
output_list.sense_o := out_vhdl.sense_o;
output_list.op_o := out_vhdl.op_o;
to reading

335 /* ----- */
    /* Components */
    /* ----- */

    /* Main component */
340 component ptov_ptof is
        var output_list : output_list := ZERO_OUTPUT_LIST
        port input_list : input_list in [0,0]
        par * in
            delta_cicle[input_list] (&output_list)
345      || input_generator[input_list] (&output_list)
        end
    /* Entry point */
    ptov_ptof

```

Código C.3 – Modelo FIACRE da PTOV

A verificação do código da PTOV, conforme descrito na Seção 6.2, encontrou um erro na verificação de uma das propriedades. O modelo FIACRE para o VHDL com correção é gerado ao alterar o código das linhas 278-293 pelo seguinte:

```

1      if in_vhdl.state_core = DECIDE_R_S then
        if (in_vhdl.f.v.a < in_vhdl.dropout_data) then
            out_vhdl.sense_timer_next := false
        else
5          if (in_vhdl.f.v.a > in_vhdl.comp_data) then
                out_vhdl.sense_timer_next := true
            end if;
        end if;
        if (in_vhdl.f.v.a < in_vhdl.dropout_data) then
10          out_vhdl.timer_next := 0;
            out_vhdl.state_core_next := UPDATE_RESET_STATE
        else
            if (((in_vhdl.f.v.a > in_vhdl.comp_data) or
15          (in_vhdl.timer_reg > 0)) and
                (in_vhdl.op.timer_reg = false)) then
                    out_vhdl.timer_reset_next := 0;
                    out_vhdl.state_core_next := UPDATE_TIMER_STATE
            else
20          out_vhdl.state_core_next := DECIDE_OP_TIMER_STATE
            end if
        end if
    end if;

```

APÊNDICE D – Exemplo Goertzel Control

Dentro do relé de proteção, os fasores para as medições de corrente e tensão são calculados por meio de um módulo que executa o algoritmo de Goertzel com as amostras obtidas na aquisição. Esse circuito que executa o Goertzel está conectado ao restante dos módulos VHDL do equipamento através de um barramento de transferência de dados chamado de **Avalon Streaming (Avalon-ST)** (AVALON-ST, 2011). O módulo deve receber, por meio do barramento, um pacote de dados, extrair a informação necessária para os cálculos (número de amostras para cálculo e as amostras) do pacote e repassar o mesmo para o módulo seguinte, adicionando ao final os fasores calculados.

O barramento **Avalon-ST** pode ter várias configurações de sinais. A configuração utilizada no módulo contém os seguintes sinais: **SOP**, **EOP**, **valid**, **ready**, **data**, **channel**. A Figura D apresenta o comportamento dos sinais citados durante a transmissão de um pacote por meio do barramento **Avalon-ST**.

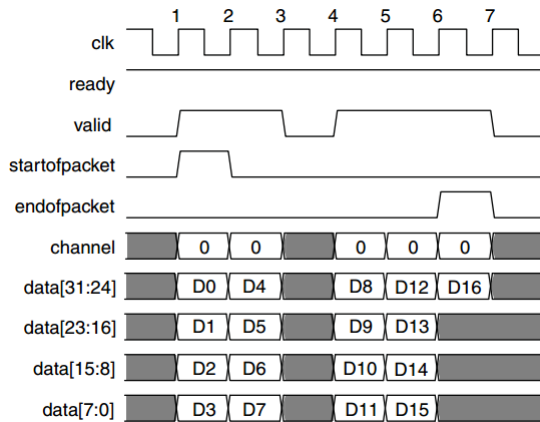


Figura 21 – Transmissão de pacotes por meio do barramento **Avalon-ST**

A transmissão dos dados somente ocorre durante a borda ativa do sinal de *clock* (pode ser configurada como subida ou descida). O sinal **start of package (SOP)** define que os dados transmitidos no atual ciclo são o início do pacote, enquanto o sinal **end of package (EOP)** define que são o fim do pacote. O sinal **valid** define que dados transmitidos no atual ciclo de *clock* são válidos. O sinal **ready** indica que o módulo que está recebendo os dados está pronto. Se acontecer em um ciclo de o sinal de **valid** ser verdadeiro, porém o **ready** ser

negativo, o módulo que está transmitindo os dados deve manter o sinal de **valid** verdadeiro (manter também os dados no sinal **data**, **sop** e **eop** até que o **ready** passe a ser verdadeiro, indicando que os dados foram recebidos. O sinal **channel** é apenas um identificador no pacote, e deve manter o valor durante toda a transmissão do mesmo.

O circuito VHDL criado para o Goertzel é dividido em dois módulos: um para fazer o controle dos sinais no barramento **Avalon**, determinado de **Goertel Control**, e outro para executar o algoritmo de Goertzel propriamente dito. O **Goertzel Control** foi verificado com a metodologia proposta e sua estrutura **Entity** com as entradas e saídas é apresentada no Código D.1.

```

1  entity goertzel_control is
      generic (
          DATA.IN.OFFSET : natural := 4
      );
5
      port (
          -- system signals
          sysclk : in std_logic;
          reset_n : in std_logic;
10
          -- avalon-st sink interface
          asi_ready_o : out std_logic;
          asi_valid_i : in std_logic;
          asi_data_i : in std_logic_vector(15 downto 0);
15          asi_sop_i : in std_logic;
          asi_eop_i : in std_logic;
          asi_channel_i : in std_logic_vector(3 downto 0);

          -- avalon-st source interface
20          aso_ready_i : in std_logic;
          aso_valid_o : out std_logic;
          aso_data_o : out std_logic_vector(15 downto 0);
          aso_sop_o : out std_logic;
          aso_eop_o : out std_logic;
25          aso_channel_o : out std_logic_vector(3 downto 0);

          -- Goertzel Interface
          get_sample : in std_logic;
          resultr : in std_logic_vector(15 downto 0);
30          resulti : in std_logic_vector(15 downto 0);
          harm2_resultr : in std_logic_vector(15 downto 0);
          harm2_resulti : in std_logic_vector(15 downto 0);
          harm3_resultr : in std_logic_vector(15 downto 0);
          harm3_resulti : in std_logic_vector(15 downto 0);
35          harm5_resultr : in std_logic_vector(15 downto 0);
          harm5_resulti : in std_logic_vector(15 downto 0);
          finish_goertzel : in std_logic;
          start_goertzel_o : out std_logic;
          n_points_fft_o : out std_logic_vector(9 downto 0);
40          data_ready_o : out std_logic;
          data_input_o : out std_logic_vector(15 downto 0)
      );
end goertzel_control;

```

Código D.1 – Estrutura **Entity** do código do Goertzel Control

O parâmetro *DATA_OFFSET* é um *generic* do VHDL do Goertzel que determina em que dado dos pacotes o módulo vai encontrar a informação de número de amostras. Por ser um *generic*, esse parâmetro não se modifica durante execução.

A entrada **sysclk** é o sinal de *clock* do circuito. A entrada **reset_n**: e um *reset* assíncrono (ativo baixo). Os sinais de entrada e saída

com prefixo “asi” são sinais da interface **Avalon** para recebimento de pacotes, enquanto que os sinais com prefixo “aso” estão relacionados com a interface de envio de pacotes. As linhas 29-42 do Código D.1 são sinais de entrada e saída relacionados com o módulo de execução do algoritmo, porém não será apresentada descrição dos mesmos porque a verificação apresentada como exemplo está focada nos sinais de interface com o barramento **Avalon**.

D.1 MODELO FIACRE DO GOERTZEL CONTROL

Para ser possível fazer a verificação do VHDL do *Goertzel Control* usando a cadeia de verificação FIACRE é necessário a criação do modelo FIACRE correspondente ao código, utilizando as regras de tradução apresentadas no capítulo 4.

Em geral, as regras de tradução apresentadas no capítulo 4 são suficientes para a tradução do VHDL do *Goertzel Control*, sendo que dessa vez, os sinais cujo tipo eram **std_logic_vector** (inclusive as portas de entrada) foram traduzidos para *arrays* de tipos booleanos em FIACRE, visto que eram utilizados no código para operações com *arrays* e não para comparações de valores e cálculos aritméticos como na função PTOV. Os sinais e entradas que utilizavam tipo **std_logic** foram traduzidos para tipos booleanos em FIACRE. O código do processo FIACRE **delta_cycle** não será apresentado nesse trabalho.

Para evitar um espaço de estados muito grande e também verificações de situações impossíveis, algumas simplificações no modelo de geração de entradas foram feitas. Como objetivo da verificação era garantir propriedades quanto a troca de sinais com o barramento **Avalon-ST**, as entradas relativas aos resultados do algoritmo de Goertzel foram mantidas em valor constante (**result r**, **result i**, **harm2 resultr**, **harm2 resulti**, **harm3 resultr**, **harm3 resulti**, **harm5 resultr** e **harm5 resulti**). Os demais sinais provenientes do módulo de processamento do algoritmo (**get sample** e **finish goertzel**) foram modelados para possuir qualquer valor.

Como a tradução dos padrões OVL para fórmulas LTL não contempla os sinais de *reset*, a entrada **reset_n** é mantida em valor constante que habilita a função.

O objetivo das propriedades era verificar o comportamento dos sinais de controle do barramento, por isso os sinais relativos a dados (**asi_data_i**) e informação de canal (**asi_channel_i**) foram mantidos com valor constante. Os sinais de controle foram modelados para

abranjer o maior número de situações possíveis, obedecendo somente as seguintes regras:

- Uma vez iniciado o recebimento de um pacote, não pode ocorrer o início de outro (**asi_sop_i** e **asi_valid_i** verdadeiros) enquanto não houver a indicação de fim do pacote anterior (**asi_eop_i** e **asi_valid_i** verdadeiros).
- Se a entrada **asi_valid_i** for verdadeira porém o circuito verificado estiver com a saída **asi_ready_o** em falso, indicando que não pode receber os dados no momento, essa entrada deve ser mantida verdadeira no próximo ciclo de *clock*.
- O sinal **aso_ready_o**, que indica que o *Goertzel Control* pode transmitir os dados, pode assumir qualquer valor.

Para satisfazer as restrições acima, foram utilizados duas instâncias do observador **ovl_window** e um do **ovl_next** com as seguintes configurações:

• **ovl_window:**

```

-start_event: asi_valid_i and asi_sop_i and asi_ready_o
-end_event:  asi_valid_i and asi_eop_i and asi_ready_o
-test_expr:  not (asi_valid_i and asi_sop_i)

```

• **ovl_window:**

```

-start_event: asi_valid_i and asi_eop_i and asi_ready_o
-end_event:  asi_valid_i and asi_sop_i and asi_ready_o
-test_expr:  not (asi_valid_i and asi_eop_i)

```

• **ovl_next:**

```

-num_cks: 1
-check_overlapping: 1
-check_missing_start: 0
-start_event: asi_valid_i and (not asi_ready_o)
-test_expr:  asi_valid_i

```

O Código D.2 apresenta o modelo do **input_generator**.

```

1  process input_generator [generated_inputs : out input_list]
   (&output_list : read output_list) is
   states init_state, clk_false, clk_true
   var //inputs declarations
5     ...
     ////////////////////////////////
     //ovl window 1
     ovl_window_open1 : bool := false,
     pre_ovl_window1 : bool := false,
10    //ovl window 2
     ovl_window_open2 : bool := false,
     pre_ovl_window2 : bool := false,
     //ovl_next
     ovl_next_fifo : queue 1 of nat := {},
15    ovl_next_cnt : nat := 0

   init to init_state
   from init_state
     //código init state

20  from clk_false
     sysclk := false;
     reset_n := true;
     asi_channel_i := 0;
     asi_data_i := ZERO_16;
25    resulti[0] := true;
     harm2_resultr [1] := true;
     harm2_resulti [2] := true;
     harm3_resultr [3] := true;
     harm3_resulti [4] := true;
30    harm5_resultr [5] := true;
     harm5_resulti [6] := true;
     generated_inputs!sysclk, reset_n, asi_valid_i, asi_data_i,
     asi_sop_i, asi_eop_i, asi_channel_i, aso_ready_i, get_sample,
     resultr, resulti, harm2_resultr, harm2_resulti, harm3_resultr,
35    harm3_resulti, harm5_resultr, harm5_resulti, finish_goertzel;
     to clk_true

   from clk_true
     sysclk := true;
40    /*ovl.window 1
     (start_event = asi_valid_i and asi_sop_i and asi_ready_o,
     test_expr = not (asi_valid_i and asi_sop_i),
     end_expr = asi_valid_i and asi_eop_i and asi_ready_o)*/
     if (asi_valid_i and asi_eop_i and
45    output_list.asi_ready_o) then
         ovl_window_open1 := false
     else
         if (not pre_ovl_window1) and
             (asi_valid_i and asi_sop_i and
50            output_list.asi_ready_o) then
             ovl_window_open1 := true
         end if;
     end if;
     pre_ovl_window1 := asi_valid_i and asi_sop_i
55    and output_list.asi_ready_o;
     ////////////////////////////////
     /*ovl.window 2
     (start_event = asi_valid_i and asi_eop_i and asi_ready_o,
     test_expr = not (asi_valid_i and asi_eop_i),
60    end_expr = asi_valid_i and asi_sop_i and asi_ready_o)*/
     if (asi_valid_i and asi_sop_i and
         output_list.asi_ready_o) then
         ovl_window_open2 := false
     else
65    if (not pre_ovl_window2) and
         (asi_valid_i and asi_eop_i and
             output_list.asi_ready_o) then
             ovl_window_open2 := true
         end if;
70    end if;
     pre_ovl_window2 := asi_valid_i and asi_eop_i
         and output_list.asi_ready_o;
     ////////////////////////////////
75    /*ovl.next
     (start_event = asi_valid_i and (not asi_ready_o),
     test_expr = asi_valid_i, num_cks = 1)*/
     if (ovl_next_cnt > 0) and (asi_valid_i
         and (not output_list.asi_ready_o)) then
         ovl_next_fifo := enqueue (ovl_next_fifo, ovl_next_cnt)
80    end if;
     if (ovl_next_cnt = 1) then

```

```

    if empty(ovl_next_fifo) then
      ovl_next_cnt := 0
    else
85      ovl_next_cnt := ovl_next_cnt - first(ovl_next_fifo) + 1;
      ovl_next_fifo := dequeue(ovl_next_fifo)
    end if
  else
    if ovl_next_cnt > 0 then
90      ovl_next_cnt := ovl_next_cnt + 1
    else
      if (asi_valid_i and (not output_list.asi_ready_o)) then
        ovl_next_cnt := 1
      end if
95    end if
  end if;
  //////////////////////////////////////
  asi_valid_i, asi_sop_i, asi_eop_i, aso_ready_i,
100  get_sample, finish_goertzel, asi_data_i[14] := any where
  (
    ((not ovl_window_open1) or (ovl_window_open1 and
      (not (asi_valid_i and asi_sop_i))))
    and
    ((not ovl_window_open2) or (ovl_window_open2 and
105    (not (asi_valid_i and asi_eop_i))))
    and
    ((asi_valid_i and (ovl_next_cnt = 1)) or
      (ovl_next_cnt <= 1))
  );
110  generated_inputs!sysclk, reset_n, asi_valid_i, asi_data_i,
  asi_sop_i, asi_eop_i, asi_channel_i, aso_ready_i, get_sample,
  result_r, result_i, harm2_result_r, harm2_result_i, harm3_result_r,
  harm3_result_i, harm5_result_r, harm5_result_i, finish_goertzel;
  to clk_false

```

Código D.2 – **input_generator** do Goertzel Control

O código completo do modelo FIACRE não será apresentado nesse trabalho. Uma estrutura de Kripke com 383845 estados foi obtida a partir dele.

D.2 PROPRIEDADES PARA O GOERTZEL CONTROL

Uma série de propriedades podem ser descritas com os atuais padrões OVL traduzidos. Nessa seção é apresentado cinco propriedades relevantes que foram verificadas com o uso da cadeia de verificação FIACRE, tendo como base o modelo FIACRE descrito na seção anterior.

Propriedade 1

O objetivo da propriedade é verificar que quando o módulo seta o valor da saída **aso_valid_o** para enviar dados, mas o receptor não está disponível (entrada **aso_ready_i** em falso), a saída é mantida em verdadeiro até que o receptor possa receber (entrada **aso_ready_i** com valor verdadeiro).

A propriedade pode ser representada com o uso do padrão OVL **ovl_window**, com as seguintes configurações:

- start_event:** `aso_valid_o`
- end_event:** `aso_ready_i`
- test_expr:** `aso_valid_o`
- clock:** `sysclk`

A propriedade em @LTL é descrita como:

$$\bullet @LTL : \square(((\neg aso_valid_o) \wedge \bigcirc (aso_valid_o \wedge \neg aso_ready_i)) \rightarrow \bigcirc \bigcirc (\neg(\neg aso_ready_i U \neg(aso_valid_o \vee aso_ready_i)))) @sysclk$$

Propriedade 2

O objetivo dessa propriedade é garantir que o módulo não envie um dado de SOP (*start of package*) com duração de mais de dois ciclos. Quando `aso_sop_o`, `aso_valid_o` e `aso_ready_i` são verdadeiros, então no próximo ciclo de *clock* ou `aso_sop_o`, ou `aso_valid_o` devem ser falsos.

A propriedade pode ser representada com o uso do padrão OVL `ovl_next`, com as seguintes configurações:

- num_cks:** 1
- start_event:** `aso_sop_o \wedge aso_valid_o \wedge aso_ready_i`
- test_expr:** `(\neg aso_sop_o) \vee (\neg aso_valid_o)`
- clock:** `sysclk`

A propriedade em @LTL é descrita como:

$$\bullet @LTL : \square((aso_sop_o \wedge aso_valid_o \wedge aso_ready_i) \rightarrow \bigcirc ((\neg aso_sop_o) \vee (\neg aso_valid_o))) @sysclk$$

Propriedade 3

O objetivo dessa propriedade é garantir que o módulo não envie um dado de EOP (*end of package*) com duração de mais de dois ciclos. Quando `aso_eop_o`, `aso_valid_o` e `aso_ready_i` são verdadeiros, então no próximo ciclo de *clock* `aso_valid_o` deve ser falso.

A propriedade pode ser representada com o uso do padrão OVL `ovl_next`, com as seguintes configurações:

- **num_cks:** 1
- **start_event:** $aso_eop_o \wedge aso_valid_o \wedge aso_ready_i$
- **test_expr:** $\neg aso_valid_o$
- **clock:** sysclk

A propriedade em @LTL é descrita como:

- **@LTL :** $\square((aso_eop_o \wedge aso_valid_o \wedge aso_ready_i) \rightarrow \bigcirc (\neg aso_valid_o))@sysclk$

Propriedade 4

O objetivo da propriedade é garantir após o módulo enviar um dado como SOP, ele não envia outro SOP enquanto não acontecer um EOP, ou seja, somente é iniciado a transmissão de um segundo pacote quando o primeiro é finalizado. Quando aso_sop_o , aso_valid_o e aso_ready_i forem verdadeiros, ou aso_sop_o , ou aso_valid_o são falsos até que aconteça um EOP (aso_sop_o , aso_valid_o e aso_ready_i verdadeiros).

A propriedade pode ser representada com o uso do padrão OVL ovl_window , com as seguintes configurações:

- **start_event:** $aso_sop_o \wedge aso_valid_o \wedge aso_ready_i$
- **end_event:** $aso_eop_o \wedge aso_valid_o \wedge aso_ready_i$
- **test_expr:** $(\neg aso_valid_o) \vee (\neg aso_valid_o)$
- **clock:** sysclk

A propriedade em @LTL é descrita como:

- **@LTL :** $\square(\neg(\neg(aso_sop_o \wedge aso_valid_o \wedge aso_ready_i)) \wedge \bigcirc((aso_sop_o \wedge aso_valid_o \wedge aso_ready_i) \wedge \neg(aso_eop_o \wedge aso_valid_o \wedge aso_ready_i))) \rightarrow \bigcirc \bigcirc (\neg(\neg(aso_eop_o \wedge aso_valid_o \wedge aso_ready_i) \wedge \neg(((\neg aso_valid_o) \vee (\neg aso_valid_o)) \vee (aso_eop_o \wedge aso_valid_o \wedge aso_ready_i))))@sysclk$

Propriedade 5

O objetivo da propriedade é garantir que após o módulo começar a receber um pacote, ele não começa a receber outro enquanto não terminar de enviar o primeiro. Caso **asi_sop_i**, **asi_valid_i** e **asi_ready_o** sejam verdadeiros, ou não acontece de **asi_sop_i** e **asi_valid_i** serem verdadeiros, ou **aso_ready_o** é falso, até que o módulo envie um EOP (**aso_eop_o**).

A propriedade pode ser representada com o uso do padrão OVL **ovl_window**, com as seguintes configurações:

- **start_event:** *asi_sop_i* \wedge *asi_valid_i* \wedge *asi_ready_o*
- **end_event:** *aso_eop_o*
- **test_expr:** $(\neg(\textit{asi_sop_i} \wedge \textit{asi_valid_i})) \vee (\neg\textit{asi_ready_o})$
- **clock:** sysclk

A propriedade em @LTL é descrita como:

- **@LTL :** $\square(((\neg(\textit{asi_sop_i} \wedge \textit{asi_valid_i} \wedge \textit{asi_ready_o})) \wedge \bigcirc((\textit{asi_sop_i} \wedge \textit{asi_valid_i} \wedge \textit{asi_ready_o}) \wedge \neg\textit{aso_eop_o})) \rightarrow \bigcirc \bigcirc (\neg(\neg\textit{aso_eop_o} \vee \neg(((\neg(\textit{asi_sop_i} \wedge \textit{asi_valid_i})) \vee (\neg\textit{asi_ready_o}))) \vee \textit{aso_eop_o}))))@sysclk$

D.3 RESULTADOS DA VERIFICAÇÃO

As fórmulas LTL acompanhado do modelo FIACRE foram aplicados a cadeia de verificação para realização do *model-checking*. O *testbench* VHDL utilizado para visualização da forma de onda dos contraexemplos foi construído de modo similar ao da função PTOV, onde os estímulos de entrada são lidos em um arquivo e repassados ao DUT. O *testbench* para esse exemplo não será apresentado.

Propriedade 1-4

As propriedades 1-4 verificadas para o *Goertzel Control* foram aprovadas no *model-checking* realizado pelo SELT. 1, O tempo gasto em todo processo, desde conversão do modelo FIACRE para TTS até a geração dos resultados pelo SELT, foi de aproximadamente 40s para cada propriedade.

Propriedade 5

Essa propriedade falhou e um contraexemplo foi gerado pela ferramenta SELT e convertido em um arquivo de estímulos que foi executado no *testbench* VHDL. O tempo gasto em todo processo, desde conversão do modelo FIACRE para TTS até o *model-checking*, foi de aproximadamente 40s. A Figura D.3 mostra a forma de onda do contraexemplo, gerada com o uso da ferramenta ModelSim (MODELSIM, 2015).

A propriedade falhou em mais de um momento sendo que um deles está marcado pela linha amarela na Figura D.3. O circuito do *Goertzel* recebeu dois sinais SOP (**asi_sop_i**, **asi_valid_i** e **asi_ready_o** verdadeiros) sem ter enviado um sinal de EOP (**aso_eop_o**, **aso_valid_o** e **aso_ready_i** verdadeiros) entre eles. Observando a evolução dos sinais internos na forma de onda e o código, observa-se que a origem da falha era que o circuito só volta a avaliar a chegada de sinais EOP após receber *DATA_OFFSET* dados, antes disso o circuito aceita qualquer dado válido e reenvia.

Correções foram feitas no código VHDL, o processo verificação foi repetido e a propriedade não falhou mais.

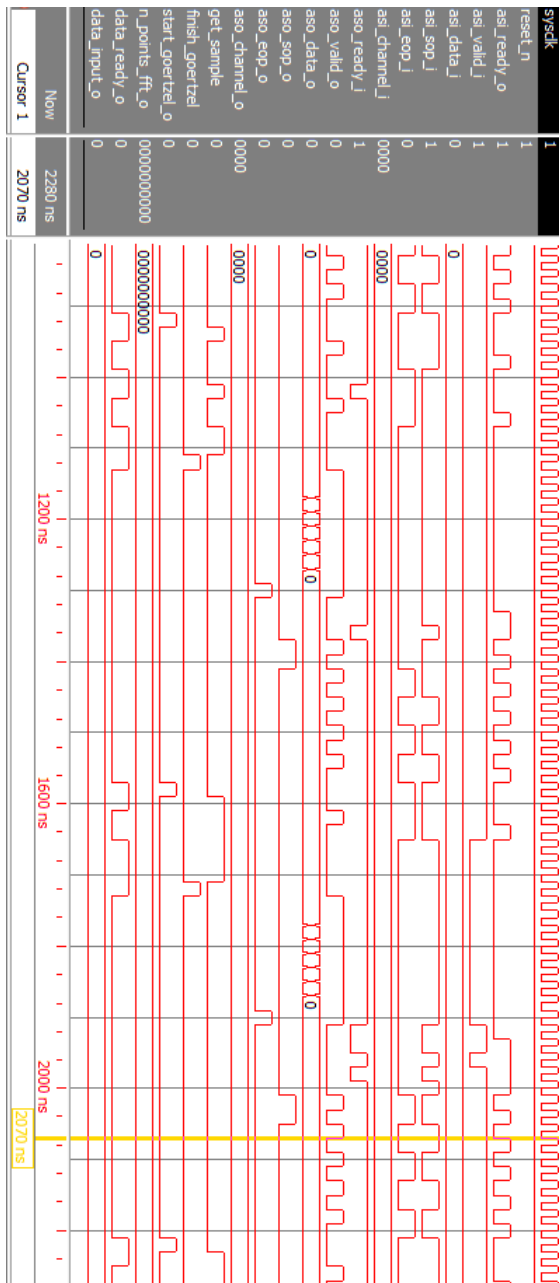


Figura 22 – Contraexemplo para propriedade 5 do *Goertzel Control* (ferramenta ModelSim (MODELSIM, 2015))