UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT
PROGRAMA DE PÓS-GRADUAÇÃO STRICTO SENSU EM COMPUTAÇÃO APLICADA

DISSERTAÇÃO DE MESTRADO

TRATAMENTO DE REQUISITOS NÃO-FUNCIONAIS EM SISTEMAS DE TEMPO-REAL EMBARCADOS IMPLEMENTADOS EM VHDL/FPGA

MARCELA LEITE

JOINVILLE, 2014

MARCELA LEITE

TRATAMENTO DE REQUISITOS NÃO-FUNCIONAIS EM SISTEMAS DE TEMPO-REAL EMBARCADOS IMPLEMENTADOS EM VHDL/FPGA

Dissertação apresentada ao Curso de Pós-Graduação Stricto Sensu em Computação Aplicada, na Universidade do Estado de Santa Catarina, como requisito parcial para obtenção do grau de Mestre em Computação Aplicada.

Orientador: Dr. Marco A. Wehrmeister

Coorientador: Dr. Cristiano D. Vasconcellos

JOINVILLE, SC
2014

L533t Leite, Marcela
Tratamento de requisitos não-funcionais em sistemas de tempo-real embarcados implementados em FPGA / Marcela Leite. – 2014.
178 p. : il. ; 21 cm

Orientador: Marco Aurélio Wehrmeister
Coorientador: Cristiano Damiani Vasconcellos
Bibliografia: p. 143-150
Dissertação (mestrado) – Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada, Joinville, 2014.

1. VHDL. 2. Aspectos. 3. Requisitos não-funcionais. I. Wehrmeister, Marco Aurélio. II. Vasconcellos, Cristiano Damiani. III. Universidade do Estado de Santa Catarina. Programa de Pós-Graduação em Computação Aplicada. IV. Título.

CDD: 005.1 – 20.ed.

MARCELA LEITE

TRATAMENTO DE REQUISITOS NÃO-FUNCIONAIS EM SISTEMAS DE TEMPO-REAL EMBARCADOS IMPLEMENTADOS EM VHDL/FPGA

Dissertação apresentada ao Curso de Mestrado Acadêmico Computação Aplicada como requisito parcial para obtenção do título de Mestre em Computação Aplicada na área de concentração "Ciência da Computação".

Banca Examinadora

Orientador:

Prof. Dr. Marco Aurélio Wehrmeister
UTFPR

Coorientador:

Prof. Dr. Cristiano Damiani Vasconcellos
CCT/UDESC

Membros

Prof. Dr. Gian Ricardo Berkenbrock
UFSC

Prof. Dr. Carlos Eduardo Pereira
UFRGS

Prof. Dr. Fabiano Baldo
CCT/UDESC

Joinville,SC, 21 de março de 2014.

Aos meus pais Ivonete Senem Leite e Germano Leite.

AGRADECIMENTOS

Primeiramente gostaria de agradecer ao meu orientador Dr. Marco A. Wehrmeister por seu empenho, paciência e apoio, tanto ao ministrar as disciplinas do mestrado quanto ao orientar minha dissertação.

Especial agradecimento aos professores que ministraram as disciplinas do mestrado pois o conhecimento repassado em suas disciplinas foi muito importante para o amadurecimento científico desse aluno. Agradeço também a todos os colegas de metrado que compartilharam seus conhecimentos e aos colegas do grupo BDES Diogo e Johnny que participaram desse projeto.

Agradeço especialmente ao meu namorado Antônio J. Fidélis, por seu apoio e companheirismo durante toda a realização do mestrado e a toda minha família por seu apoio e compreensão pelos momentos ausentes. Acima de tudo, agradeço a Deus por proporcionar esse momento.

RESUMO

LEITE, Marcela. **Tratamento de Requisitos Não-Funcionais em Sistemas de Tempo-Real Embarcados Implementados em VHDL/FPGA.** 2014. 178f. Dissertação (Mestrado Acadêmico em Computação Aplicada - Área: Engenharia de Software) - Universidade do Estado de Santa Catarina. Programa de Pós-Graduação em Computação Aplicada, Joinville, 2014.

Este trabalho apresenta uma abordagem de desenvolvimento de sistemas embarcados implementados em FPGA, que agrega técnicas de MDE e AOSD com o objetivo de sistematizar e automatizar o processo de desenvolvimento. Propõe-se o tratamento e gerenciamento dos requisitos não funcionais para sistemas embarcados desenvolvidos na plataforma FPGA, com o uso do paradigma orientado a aspectos e de métricas que possibilitem o controle no cumprimento das restrições do projeto. Para tanto, a geração do código VHDL a partir do modelo especificado na UML foi implementada neste trabalho. Para essa transformação, um conjunto de regras de mapeamento dos elementos da UML para VHDL foi criado. A partir da análise da literatura foi detalhado um conjunto de requisitos não funcionais para projetos implementados em FPGA, que constituem o *framework* de aspectos para essa plataforma. Novos aspectos foram incluídos no DERAF e implementadas regras de mapeamento para esses. Foram desenvolvidos três estudos de caso utilizando a abordagem e o conjunto de regras de mapeamento criado, nos quais foram implementados três aspectos que tratam requisitos não funcionais dessas aplicações. Com as regras de mapeamento implementadas, foi possível a geração completa do código VHDL, funcional e sintetizável. O uso das métricas identificadas permitiu uma avaliação mais precisa da eficácia da utilização da abordagem proposta. Os resultados encontrados, mostram que a utilização da orientação a aspectos para o tratamento de requisitos não funcionais na descrição de hardware em VHDL, melhora o desempenho do sistema, tem alto impacto sobre o sistema final e contribui para o atendimento de requisitos de projeto como *time-to-market*, reusabilidade e manutenabilidade.

Palavras-chave: VHDL. Aspectos. Requisitos não funcionais.

ABSTRACT

This work proposes a design approach for FPGA-based embedded system. Such an approach integrates concepts and techniques from Model Driven Engineering and Aspect Oriented Software Development approaches, in order to systematize and automate the design process. AOSD concepts are used to handle non-functional requirements in FPGA-based embedded system development, in conjunction with a set of metrics to control the project constraints. A subset of relevant non-functional requirements for FPGA-based embedded system, as well as a set of metrics to evaluate these requirements, have been identified through a survey of literature. This subset of non-functional requirements composes an aspect-oriented framework for the FPGA platform, including aspects and their implementation in VHDL. In this sense, new aspects have been included in the Distributed Real-time Aspects Framework (DERAF). Moreover, to support the proposed approach, a code generation tool was enhanced to enable the generation of VHDL descriptions from UML models. A set of mapping rules have been defined to perform this UML-to-VHDL transformation. In order to validate the proposed approach, three case studies have been performed. The obtained results demonstrate the feasibility of combining AOSD and MDE in order to handle non-functional requirements in the design of systems through hardware description languages, such as VHDL. In addition, the modularization achieved by using AOSD affects positively the final embedded system, improving the overall system performance, as well as it contributes to the fulfillment of system requirements, *time-to-market*, reusability and manutenability of design artifacts, such as models and component descriptions.

Key-words:VHDL. Aspects. Non-Functional Requirements.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AB	Aspectual Bloat
ACOD	Aspects Crosscutting Overview Diagram
ADH	Aspect Described Hardware-Description-Language
ADL	Architecture Description Languages
AMoDE-RT	Aspect-oriented Model-Driven Engineering for Real-Time systems
ANOffM	Access Number Off-Chip Memory
ANOnM	Access Number On-Chip Memory
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
ASIP	Application-Specific Instruction-Set Processors
CDLOC	Concern Diffusion over Lines of Code
CI	Circuitos Integrados
CPT	Critical Path Time
DERAF	Distributed Embedded Real-time Aspects Framework
DERCS	Distributed Embedded Real-time Compact Specification
ECF	Energy Consumption/Functionality
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
FR	Fault Rate
FRIDA	From RequIrements to Design using Aspects
GenERTiCA	Generation of Embedded Real-Time Code based on Aspects
HDL	Hardware Description Language
HRM	Hardware Resource Modeling
IEEE	Institute of Electrical and Electronics Engineers
IOBs	Input/Output Blocks
JPDD	Join Point Designation Diagrams
LOC	Lines of Code
LOAC	Lines of Adaptation Code

LOWC	Lines of Woven Code
LUT	Look-Up Table
MARTE	Modeling and Analysis of Real-Time and Embedded systems
MDE	Model Driven Engineering
MF	Maximum Frequency
MS	Memory Size
NoC	Network on Chip
NPU	Number of Processing Units
OS	Occupied Slices
PIM	Platform Independet Model
POO	Programação Orientada a Objetos
PSM	Platform Specific Model
PWM	Pulse Width Modulation
SoC	System-on-Chip
RF	Risk Factor
RSM	Repetitive Structure Modeling
TR	Tangling Ratio
UML	Unified Model Language
VHDL	Very high speed integrated circuit HDL
VSL	Value Specification Language
WCD	Worst Case Delay
WCET	Worst Case Execution Time
WS	Word Size
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

A computação está cada vez mais presente em nosso cotidiano como, por exemplo, em dispositivos eletrônicos, eletrodomésticos, veículos ou automação predial, entre outros. Esses dispositivos, constituídos por hardware e software, são desenvolvidos para um conjunto restrito de funções e, em geral, estão embutidos em algum equipamento ou sistema maior, sendo denominados sistemas embarcados (WOLF, 2008). A *engenharia de sistemas* trata do processo de desenvolvimento dos sistemas embarcados, envolvendo o hardware e software (SOMMERVILLE, 2007). A complexidade do processo de desenvolvimento desses sistemas tem aumentado devido ao aumento da complexidade das funcionalidades e do tamanho das aplicações fornecidas. Por essa razão, a *engenharia de software* vem emprestando vários conceitos e abstrações tais como diagramas de comportamento e estrutura de objetos, para lidar com esses desafios e agilizar o processo de desenvolvimento.

Sistemas embarcados são compostos por microprocessadores complexos denominados *System-on-Chip* (SoC)[1], que requerem dos desenvolvedores muito mais conhecimentos e habilidades do que a programação de softwares convencionais. É necessário conhecer o hardware em detalhe, ter o domínio dos componentes da plataforma e do tratamento dos requisitos do projeto que será desenvolvido para obter o melhor desempenho a um baixo custo. Um exemplo são os dispositivos *Field-Programmable Gate Array* (FPGA) onde é o desenvolvedor que implementa a arquitetura e comportamento do hardware, e não apenas a codificação de um software.

Os sistemas embarcados estão sujeitos a rígidas restrições devido as suas características de tempo-real, consumo de energia e tamanho. Tais requisitos impactam fortemente no projeto desses sistemas. Esses requisitos se referem aos aspectos arquiteturais, de qualidade de projeto e ao comportamento do sistema. Há uma crescente necessidade no tratamento dos requisitos do projeto, principalmente os relacionados aos recursos de hardware e restrições temporais, devido a grande utilização dos sistemas embarcados em aplicações de tempo-real (MONMASSON; CIRSTEA, 2007; SALEWSKI; TAYLOR, 2008).

O gerenciamento dos requisitos não funcionais é outro fator que diferencia o desenvolvimento de sistemas embarcados dos sistemas convencionais. Para gerenciar é necessário medir os indicadores do atendimento dos requisitos não funcionais. Para isso é importante a definição de métricas e sua aplicação. Assim, outro ramo de pesquisa é a validação dos requisitos não funcionais do projeto com o uso de métricas no processo de desenvolvimento (REDIN et al., 2008; CORREA et al., 2010).

[1] Sistemas embarcados desenvolvidos dentro de um único chip.

Nesse contexto, pesquisadores em engenharia de software vem desenvolvendo várias ferramentas e metodologias de suporte ao desenvolvimento de sistemas embarcados. Uma de suas vertentes é a automatização do processo de desenvolvimento com o uso da engenharia guiada por modelos, do inglês *Model Driven Engineering* (MDE) (WEHRMEISTER, 2009). A abordagem MDE propõe o desenvolvimento de projetos a partir da transformação de modelos e geração automática de código (SELIC, 2003). Nesta, o principal artefato é o modelo, que pode ser, por exemplo, descrito por meio de uma especificação UML, do inglês *Unified Modeling Language*, assim como outros modelos utilizados de acordo com o domínio da aplicação.

Metodologias para o tratamento e gerenciamento dos requisitos não funcionais em sistemas embarcados também vem surgindo. Abordagens de desenvolvimento orientado a objetos e a aspectos vem sendo propostas como solução para esse problema. O desenvolvimento orientado a aspectos já é comum no desenvolvimento de software devido aos benefícios que traz ao projeto como manutenabilidade e reutilização de código (RASHID et al., 2010). A orientação a aspectos agora é proposta para o desenvolvimento de sistemas embarcados (WEHRMEISTER; PEREIRA; RAMMIG, 2013).

Esta dissertação explora os desafios de desenvolvimento de sistemas embarcados de tempo-real, buscando uma solução para o tratamento e gerenciamento dos requisitos não funcionais. Nesta dissertação é apresentada uma metodologia que possibilita a especificação em alto-nível dos requisitos não funcionais e a automatização do processo de codificação desses sistemas.

1.1 CONTEXTUALIZAÇÃO DO TRABALHO

O desenvolvimento de sistemas embarcados, simples ou complexos, está cada vez mais ágil, requerendo plataformas eficientes e automatizadas para reduzir o tempo de projeto e seu custo (SHIMIZU et al., 2004). Além disso, a busca por componentes de hardware mais baratos e adaptáveis tem motivado a utilização de FPGAs. O aperfeiçoamento de circuitos FPGA os torna uma opção interessante de hardware para sistemas embarcados complexos por serem flexíveis e possuírem custo/benefício economicamente atraente (MONMASSON; CIRSTEA, 2007; SALEWSKI; TAYLOR, 2008).

Muitos pesquisadores avaliaram a utilização das FPGAs na indústria e verificaram seu potencial para a aplicação em sistemas embarcados (MONMASSON; CIRSTEA, 2007; SALEWSKI; TAYLOR, 2008; TAHOORI et al., 2009). Entretanto, aqueles pesquisadores observaram o avanço do hardware em contrapartida a defasagem das ferramentas de desenvolvimento e técnicas para o uso desta tecnologia. De fato, a limitação das ferramentas para o desenvolvimento de sistemas implementados em FPGAs tem dificultado sua

adoção pela indústria (MONMASSON et al., 2011). Além disso, as FPGAs possuem características especiais em relação às demais plataformas de sistemas embarcados. Um exemplo é a descrição textual especificada em Linguagem de Descrição de Hardware (*Hardware Description Language - HDL*), que, para as FPGAs, define a arquitetura e comportamento do hardware e não apenas um conjunto de instruções que será executado por um processador tradicional.

Alguns autores sugerem introduzir níveis de abstração mais altos ao projeto para lidar com a complexidade no desenvolvimento de sistemas embarcados (GOKHALE; BALASUBRAMANIAN; LU, 2004; DENG; SCHMIDT; GOKHALE, 2006; ANNE et al., 2009; DRIVER et al., 2010; QIU; ZHANG, 2011). Uma abordagem semelhante vem sendo proposta para o desenvolvimento de projetos implementados em FPGAs (QUADRI; MEFTALI; DEKEYSER, 2008; MUCK et al., 2011; CARDOSO et al., 2012). Segundo Sommerville (2007), *abstrair* significa utilizar modelos ou linguagens de programação de alto-nível, que removam ou ocultem detalhes de implementação, tornando o processo de compreensão e desenvolvimento mais simples. Isso implica em ocultar detalhes de implementação como, por exemplo, os requisitos não funcionais nas fases iniciais de desenvolvimento. No entanto, abstrair detalhes de implementação é um desafio para o desenvolvimento de sistemas embarcados que lidam com várias restrições de projeto, tais como consumo de energia, memória e área, mesmo nas fases iniciais do projeto (MONMASSON; CIRSTEA, 2007; SALEWSKI; TAYLOR, 2008). Esse desafio é uma brecha para o desenvolvimento de sistemas embarcados que ainda não foi tratado e explorado o suficiente. Esse estudo pode aperfeiçoar os processos e metodologias de desenvolvimento de sistemas embarcados e avançar o estado-da-arte da *engenharia de sistemas*.

A MDE vem sendo proposta como uma solução para agilizar o processo de desenvolvimento de sistemas embarcados, melhorando a produtividade e o *time-to-market* do produto (QUADRI; MEFTALI; DEKEYSER, 2008; WEHRMEISTER, 2009; QUADRI; MEFTALI; DEKEYSER, 2010; MOREIRA, 2012). A abordagem MDE utiliza-se de níveis mais abstratos para a modelagem dos sistemas como, por exemplo, modelos especificados na linguagem UML. A partir de mecanismos de transformação, um Modelo Independente de Plataforma (*Platform Independet Model ou PIM*) de execução é transformado em um Modelo Específico à Plataforma (*Platform Specific Model ou PSM*) para uma arquitetura escolhida e, neste momento, são adicionados os detalhes e restrições relativos à plataforma escolhida. A UML possui uma extensão específica para a modelagem de sistemas embarcados chamada MARTE (*Modeling and Analysis of Real-Time and Embedded systems*), que fornece pacotes para modelar tanto o software quanto o hardware. Com isso,

torna-se viável o processo de automatização do desenvolvimento de sistemas embarcados, contando com ferramentas e linguagens apropriadas para cada nível de abstração. Contudo, é necessário um mapeamento entre o artefato representado pelo modelo e o artefato que será gerado no código.

A UML foi desenvolvida com base no paradigma orientado a objetos. Assim a conversão de modelos para código orientado a objetos se torna natural. Entretanto quando se trata de hardware existem outros paradigmas envolvidos, devido a sua natureza concorrente e estrutural. Esse problema é mais claro em projetos implementados em FPGAs, onde a descrição do hardware resultará em sua estrutura lógica e comportamental. A linguagem VHDL (do inglês *Very high speed integrated circuit HDL*) é um dos padrões para descrição de hardware e se assemelha com uma linguagem estruturada. VHDL possui construções e elementos que não estão diretamente associados a nenhum artefato ou abstração da UML. Tais características tornam mais complexa a transformação de um modelo UML orientado a objetos para uma descrição de hardware que pode ser implementada em FPGA, sendo o mapeamento destas regras de implementação um desafio no projeto de sistemas embarcados com FPGA.

A falta de ferramentas apropriadas para lidar com tais requisitos em um nível mais abstrato do projeto é um desafio para a utilização de abordagens MDE em projetos implementados em FPGA. Os requisitos não funcionais influenciam durante todo o ciclo de desenvolvimento do projeto podendo estar ou não vinculados a alguma funcionalidade. Alguns autores propõem a utilização do paradigma orientado a aspectos para separar e modularizar a implementação dos requisitos não funcionais dos funcionais, possibilitando melhor gerenciamento e manutenção desses (ENGEL; SPINCZYK, 2008; MUCK et al., 2011; CARDOSO et al., 2012; MEIER; HANENBERG; SPINCZYK, 2012). O paradigma de Desenvolvimento de Software Orientado a Aspectos (do inglês *Aspect-Oriented Software Development, AOSD*) visa a separação de preocupações transversais com o uso de um tipo de abstração chamada *aspecto* (ELRAD; FILMAN; BADER, 2001). AOSD busca separar o tratamento de aspectos que estão presentes em todo o projeto, influenciando diversas funcionalidades, em módulos específicos, que são entrelaçados durante a compilação do código. O principal benefício apontado nesta abordagem é a reutilização do tratamento desses requisitos não funcionais, assim como a sua manutenção.

Esse trabalho aborda dois problemas: (i) o mapeamento dos elementos e comportamentos da UML para o código VHDL e (ii) o tratamento dos requisitos não funcionais em um nível mais alto de abstração com o emprego do paradigma orientado a aspectos. Assim, este trabalho apresenta uma abordagem para o desenvolvimento automatizado de projetos implementados em

FPGAs. Esta abordagem permite o desenvolvimento dos sistemas com a aplicação de técnicas da MDE, utilizando a UML como linguagem de modelagem para a geração automática de código VHDL. Os requisitos não funcionais do projeto são tratados pela aplicação de conceitos e técnicas utilizadas no paradigma de AOSD, permitindo separar o tratamento/gerenciamento de requisitos funcionais dos não funcionais. Esta proposta visa tornar o processo de desenvolvimento mais sistemático e controlado por meio de métricas. Foi desenvolvido um novo conjunto de regras de mapeamento dos elementos da UML para VHDL, com base nos conceitos apresentados em (MOREIRA, 2012). Esse conjunto de regras permite a transformação da especificação em alto nível do sistema embarcado para uma descrição do hardware em VHDL.

Em uma segunda etapa, foram identificados e implementados requisitos não funcionais com a abordagem orientada a aspectos. Os aspectos foram incluídos nas regras de mapeamento, permitindo a geração do código que instrumenta o projeto para o tratamento das restrições dos requisitos não funcionais. Por fim, para validar a abordagem, as regras de mapeamento e os aspectos desenvolvidos, foram desenvolvidos três estudos de caso de sistemas embarcados na plataforma FPGA. Esses estudos de caso permitiram avaliar a aplicabilidade e eficiência dos elementos desenvolvidos neste trabalho.

1.2 OBJETIVOS E ESCOPO DO TRABALHO

Considerando a problemática apresentada, o objetivo principal deste trabalho é: *Propor uma abordagem para o tratamento dos requisitos não funcionais para a plataforma FPGA com o uso de técnicas da MDE e AOSD, implementando o mapeamento de elementos de um modelo UML especificando um sistema embarcado, para construções na linguagem de descrição de hardware VHDL.*

O objetivo principal deste trabalho pode ser dividido nos seguintes objetivos específicos:

- Definir o mapeamento dos elementos da UML para construções da linguagem VHDL;
- Criar o conjunto de regras de mapeamento na ferramenta GenERTiCA para a geração do código VHDL a partir do modelo UML;
- Estender o *framework* DERAF, incluindo novos requisitos não funcionais, identificados na revisão da literatura;
- Criar um conjunto de regras de mapeamento para os requisitos não funcionais implementados;

- Realizar estudos de caso para avaliação da abordagem e das regras propostas;
- Avaliar o tratamento dos requisitos não funcionais utilizando as métricas identificadas.

1.3 CONTRIBUIÇÕES DO TRABALHO

A principal contribuição desse trabalho é a evidência de que é possível e benéfico a especificação de sistemas embarcados baseados em FPGA em alto nível de abstração a partir de modelos UML. Assim como o tratamento e gerenciamento dos requisitos não funcionais em projetos de sistemas embarcados na plataforma FPGA com o uso de AOSD, é possível e benéfico para o projeto. Este trabalho também fornece as seguintes contribuições:

- Definição de conceitos para a transformação dos modelos UML para construções da linguagem VHDL e suas limitações;
- Identificação de requisitos não funcionais importantes para a plataforma FPGA;
- Definição de conceitos para o tratamento de requisitos não funcionais na plataforma FPGA e seu mapeamento em aspectos que os tratam;
- Definição de *Pointcuts* para a linguagem VHDL e seus impactos;
- Extensão do DERAF e da ferramenta GenERTiCA para implementar a transformação proposta do modelo UML para VHDL;
- Implementar o tratamento dos requisitos não identificados para a linguagem VHDL com o uso de conceitos da orientação a aspectos;
- Identificação de métricas para a avaliação de projetos implementados em FPGA;
- Análise do impacto da utilização da abordagem proposta na plataforma FPGA.

1.4 ORGANIZAÇÃO DO TEXTO

O texto desta dissertação encontra-se organizado como indicado nos parágrafos a seguir.

No Capítulo 2 são apresentados os principais conceitos para o entendimento e elaboração deste trabalho, enquanto o Capítulo 3 apresenta uma revisão da literatura sobre a temática deste trabalho.

O Capítulo 4 descreve a implementação da transformação UML para VHDL proposta neste trabalho. Este capítulo foi dividido da seguinte maneira: a Seção 4.1 discute os principais requisitos não funcionais identificados para a plataforma FPGA, assim como as métricas para avaliação de projetos nesta plataforma; a Seção 4.3, por sua vez, apresenta as regras de mapeamento para a geração automática do código VHDL a partir dos modelos UML. Adicionalmente, a Seção 4.3 apresenta a definição dos *pointcuts* e aspectos identificados e desenvolvidos para a plataforma FPGA/VHDL.

O Capítulo 5 apresenta os estudos de caso desenvolvidos e os resultados dos experimentos realizados, assim como uma análise do impacto da abordagem proposta.

Por fim, o Capítulo 6 apresenta as conclusões deste trabalho e propostas de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta os principais conceitos necessários para a execução deste projeto. Esses conceitos auxiliam no entendimento da problemática da pesquisa e servem como base para o desenvolvimento da abordagem proposta.

2.1 SISTEMAS EMBARCADOS

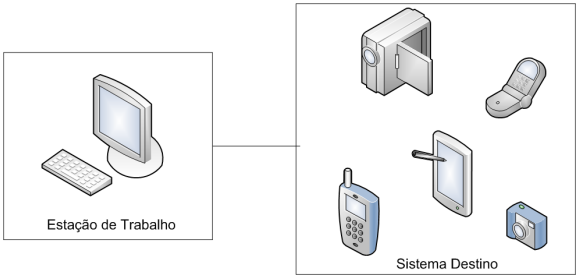
Segundo Wolf (2008), sistemas embarcados são dispositivos compostos por hardware e software que possuem um processador programável desenvolvidos para um propósito específico, diferentemente de computadores pessoais que são sistemas de uso geral. Os sistemas embarcados podem ser aplicados em diversas áreas desde eletrônicos de consumo até automação residencial, aplicações automobilística e aeroespaciais.

A utilização de sistemas embarcados iniciou-se por volta de 1970, com o desenvolvimento de processadores simples, em um único chip, para o uso em calculadoras. Observou-se que esses chips poderiam ser reprogramados para utilização com outros propósitos, dando origem ao termo *computer-on-a-chip* (WOLF, 2008). Mais tarde passou-se a usar o termo *System-on-a-Chip* ou SoC, ou seja, um sistema completo em um único chip (CARRO; WAGNER, 2003).

Os sistemas embarcados atuais tem como desafio implementar algoritmos complexos, fornecendo interfaces sofisticadas, além de lidar com restrições rígidas de desenvolvimento e projeto como, por exemplo, o *time-to-market* (CARRO; WAGNER, 2003). Estas restrições levaram a adotação de microprocessadores para o desenvolvimento de sistemas embarcados, ao invés de computadores de uso geral, pois os microprocessadores são mais simples e podem ser facilmente customizados para atender estas restrições de projeto, além de possuírem custo mais baixo do que processadores. Outro aspecto importante em sistemas embarcados são as características de tempo-real, ou seja, os sistemas precisam responder corretamente aos estímulos e dados de entrada e ainda fornecer estas respostas no momento certo. O projetista precisa lidar com essas restrições levando em conta as limitações de poder de processamento, memória e consumo de energia das plataformas de sistemas embarcados. Essa característica torna o desenvolvimento desse tipo de sistema muito peculiar, trazendo limitações de hardware e software para o projeto (WOLF, 2008). Carro e Wagner (2003) apontam essas características como cruciais na definição pela tecnologia para o desenvolvimento de sistemas embarcados, que podem ser desde FPGAs até microprocessadores e microcontroladores.

Wolf (2008) destaca como uma das dificuldades do desenvolvimento dos sistemas embarcados a adaptação do sistema no ambiente destino e a sua validação nesse ambiente, pois esse último possui características diferentes do ambiente de produção, conforme apresentado na Figura 1. Segundo Wolf (2008) é importante a utilização de ferramentas apropriadas de depuração em ambos os ambientes, assim como ferramentas de suporte na plataforma destino para a validação final e verificação de eventuais erros, tais como LEDs, *breakpoints*, portas de conexão serial ou USB.

Figura 1 – Ambiente de desenvolvimento e produção.



Fonte: Produção do próprio autor. Adaptado de (WOLF, 2008).

Carro e Wagner (2003) destacam a automação de projetos e testes como um fator chave para o sucesso no desenvolvimento de sistemas embarcados. A automação permite a reutilização de plataformas e componentes, além de agilizar o desenvolvimento dos sistemas, aumentando a viabilidade dos projetos, além de diminuir os custos e o *time-to-market*.

2.2 SISTEMAS DE TEMPO-REAL

Uma aplicação é dita de tempo-real quando é sensível ao cumprimento dos prazos (*deadlines*) de execução das tarefas (TSAI et al., 1996). Os sistemas embarcados podem possuir restrições de tempo, que dizem respeito ao prazo para execução das tarefas, assim como o tempo de validade dos dados após o prazo determinado (*freshness*) (FARINES; FRAGA; OLIVEIRA, 2000). Segundo Tsai et al. (1996) os sistemas de tempo-real podem ser classificados como:

- **Hard:** são aplicações altamente sensíveis aos prazos para a execução de tarefas. Se esses prazos não forem cumpridos, uma grande perda ou desastre pode acontecer. Exemplos de sistemas de tempo-real *hard* são

componentes de aeronaves, carros ou componentes utilizados na área médica, entre outros;

- **Soft:** são aplicações onde uma distribuição estatística de atrasos é aceitável, podendo até ocorrer perdas e danos devido a esse atraso, porém o impacto não é drástico. Geralmente os prazos *soft* podem degradar o desempenho do sistema, uma vez que precisam aguardar o término das tarefas. Exemplos de sistemas de tempo-real *soft* são aplicações de multimídia, como o YouTube ou TV digital.

Outra característica importante dos sistemas de tempo-real é a *previsibilidade*, pois é necessário saber exatamente quando uma tarefa começa e termina, além de vários outros fatores que afetam no processamento, como o escalonamento das tarefas, atrasos, preempção, entre outros. Dessa forma, uma das métricas mais importantes para sistemas de tempo-real é conhecer o Pior Caso do Tempo de Execução (do inglês *Worst Case Execution Time - WCET*) de uma tarefa, pois permite controlar o cumprimento de prazos de execução.

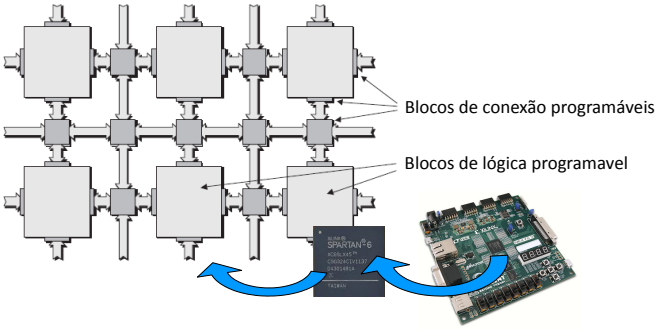
2.3 FIELD-PROGRAMMABLE GATE ARRAY

Field-Programmable Gate Array ou FPGA são Circuitos Integrados (CI) formados por vetores de portas lógicas interconectadas que podem ser configuradas pelo usuário final, daí o nome *field-programmable* (MAXFIELD, 2004).

Os circuitos FPGAs são constituídos de unidades de lógica combinacional, que são compostas por blocos de lógica e blocos de conexão programáveis. Os blocos de lógica permitem a implementação de vários tipos de portas lógicas, como “AND”, “OR” ou “XOR”. Os blocos de conexão permitem conectar os blocos de lógica, criando circuitos mais complexos. As FPGAs possuem fios conectando todos os blocos, denominados canais de comunicação, que podem ter diversos tamanhos, dependendo da distância entre os elementos conectados. Os blocos de conexão permitem selecionar quais canais serão utilizados para a lógica programada. As interconexões criadas entre os elementos da FPGA geram uma rede lógica de sinais, onde cada conexão é denominada *netlist*. As FPGAs também possuem os blocos de entrada e saída (*Input/Output Blocks - IOBs*) para comunicação externa (WOLF, 2004). Na Figura 2 é apresentada a arquitetura genérica de uma FPGA.

FPGAs são reconfiguráveis devido a tecnologia de fabricação. Algumas FPGAs são programadas uma única vez. Esse tipo de FPGA é constituída por links de fusíveis ou anti-fusíveis microscópicos. Todos os blocos de lógica vem conectados de fábrica por esses links. A “programação” da FPGA se

Figura 2 – Arquitetura genérica de uma FPGA.



Fonte: Produção do próprio autor. Adaptado de (MAXFIELD, 2004).

baseia em remover fusíveis ou acionar os anti-fusíveis para gerar a lógica desejada. Tal processo não pode ser revertido. Já as FPGAs reprogramáveis, que são o objeto de estudo deste trabalho, são constituídas por memórias SRAM (*Static RAM*) composta de transistores e *flip-flops* (FF). Contudo as memórias SRAM perdem a informação ao desligar a fonte de energia, necessitando a sua reprogramação sempre que são religadas. Para resolver esse problema, são utilizadas memórias externas, que mantém os dados de configuração para serem carregados no momento em que a FPGA é ligada. O dados de configuração da FPGA são armazenados em um arquivo denominado *bitstream* (MAXFIELD, 2004).

O desenvolvimento de projetos implementados em FPGAs segue um processo hierárquico, em diferentes níveis de abstração. Desde o modelo esquemático, onde a definição é em nível de portas lógicas, até o nível de descrição de hardware por meio de uma linguagem HDL (ver Seção 2.4) (WOLF, 2004). Após a especificação do projeto é efetuada a síntese do código, por uma ferramenta dedicada a plataforma FPGA utilizada como, por exemplo, a *ISE Web Pack* da empresa Xilinx[1]. Na etapa de síntese são efetuados a localização e roteamento dos componentes do projeto no circuito FPGA, além de otimizações que permitem melhorar o desempenho e consumo de energia do projeto e, então, é gerado o arquivo *bitstream* com os bits de configuração da FPGA.

[1] www.xilinx.com

2.4 VHDL

Devido à complexidade no desenvolvimento de projetos eletrônicos e do aumento do número de componentes em um único CI, a atividade de definição do hardware em nível de portas lógicas e FFs tornou-se difícil e sujeita a erros. Assim, linguagens de descrição de hardware ganharam espaço devido à facilidade e agilidade que trazem para esta tarefa (ROTH, 1998). A liguagem VHDL foi proposta em 1980 pelo departamento de defesa dos Estados Unidos da América com o objetivo de padronizar a descrição de circuitos integrados, facilitando seu entendimento e desenvolvimento. Atualmente a linguagem VHDL é padronizada pela *Institute of Electrical and Electronics Engineers* (IEEE) nas normas internacionais 1076 e 1164, cuja última atualização é de 2009 (IEEE, 2009). Esta linguagem permite descrever tanto a estrutura de circuitos eletrônicos, como também o seu comportamento (PEDRONI, 2004).

Outra linguagem muito popular para descrição de hardware é o Verilog. Verilog segue uma metodologia *"bottom-up"*. O Verilog é utilizado em geral para a simulação do comportamento de circuitos. Uma de suas limitações é não permitir a definição de diferentes comportamentos em um único módulo (ROTH, 1998). Já a linguagem VHDL segue uma metodologia *"top-down"* e permite utilizar múltiplos níveis de modelos com diferentes arquiteturas, além de também permitir a simulação de circuitos (ROTH, 1998).

VHDL permite descrever as portas lógicas e as conexões que serão configuradas no hardware por meio de expressões booleanas. Outra característica importante é que os comandos VHDL executam em paralelo, i.e., cada linha ou elemento do código corresponde a um circuito que processará sinais paralelamente com outros circuitos ou componentes. Entretanto VHDL também possui estruturas que permitem implementar máquinas de estado para processamentos sequenciais. A seguir são apresentados alguns conceitos básicos da linguagem VHDL com base em (PEDRONI, 2004; IEEE, 2009).

1. **Entity**: Define a estrutura física do circuito. Uma *entity* pode definir as seguintes estruturas:
   - **Port**: esse comando define as entradas e saídas do circuito com tipo e direção.
   - **Generics**: esse comando é utilizado para definir parâmetros para o circuito.
   - **Constants**: permite definir constantes para o circuito.

   Na Listagem 2.1 é apresentado um exemplo de definição de uma *entity*.

Listagem 2.1 – Exemplo de Entity

```
1 -- Exemplo completo de uma Entity:
```

```
2  entity exemplo is
3
4    Port ( sw : in  STD_LOGIC_VECTOR (2 downto 0);
5           led : out  STD_LOGIC_VECTOR (7 downto 0)  );
6
7    generic ( gain : integer := 4;
8              timeDelay: time := 10 ns );
9
10   constant : rpu : real := 100.0;
11
12 end exemplo;
```

2. **Variables e Signals**: São duas formas de armazenar e transmitir valores no circuito na linguagem VHDL. *Variables* armazenam valores como em um software, porém esses valores são acessíveis somente no escopo em que a variável foi definida. Variáveis podem ser declaradas somente dentro de códigos sequenciais. *Signals* permitem a troca de valores por meio de sinais e estão acessíveis fora do escopo de onde foram definidos. Os sinais podem ser definidos em *entity*, *architecture* ou dentro de blocos de código sequencial. A principal diferença entre *signals* e *variables* é que, ao atualizar um valor em um *signal*, a atualização estará disponível somente no próximo ciclo de clock, enquanto que a atualização em *variables* é automática.

3. **Architecture**: Define o comportamento do circuito que pode ser do tipo *Behavioral*, *Data Flow* ou *Structural*. Nesta seção do código será definido efetivamente a estrutura lógica do circuito, i.e., como o circuito se comportará com base nas entradas e conexões. Variáveis locais e sinais podem ser declarados dentro do escopo da *architecture*, antes do comando *begin*, conforme o exemplo da Listagem 2.2. Esse local é chamado de "seção de declarações da arquitetura". Essas variáveis poderão ser utilizadas somente dentro do escopo da arquitetura que as definiu.

Listagem 2.2 – Exemplo de Architecture

```
1 architecture behaviour exemplo is
2         -- declaracao de variaveis locais
3         signal sig1, sig2 : bit;
4 begin
5   -- '<=' e um comendo de atribuicao para sinais
6   saida <= sig1 and sig2;
7 end exemplo;
```

4. **Process, Function e Procedure**: São comandos que permitem definir máquinas de estados para execução sequencial. É importante observar

que esses blocos de código executam em paralelo com outros comandos do código. *Process* são utilizados dentro de um *architecture* para criar um bloco de lógica sequencial. Esses blocos são locais do módulo implementado e não podem ser acessados por outras rotinas. Na Listagem 2.3 é apresentado um exemplo de utilização do comando *Process*.

Listagem 2.3 – Exemplo de Architecture

```
1  architecture behaviour exemplo is
2          -- declaracao de variaveis de toda a aquitetura
3          signal port1, port2 : bit;
4  begin
5
6    nome:Process (port2) -- lista de variaveis sensitivas
7      -- declaracao de variaveis locais do processo
8      signal sig1, sig2 : bit;
9    begin
10     -- aqui vai o codigo sequencial
11     port2 <= sig1 or sig2;
12   end nome;
13
14   -- esse comando executa em paralelo com o processo
15   port <= port2;
16
17 end exemplo;
```

Procedures e *functions* são utilizados em sub-programas para implementar um bloco de lógica sequencial que pode ser chamado por outras rotinas, permitindo o seu reaproveitamento em várias rotinas. *Procedures* não retornam nenhum valor após o processamento, já *functions* retornam algum valor. Não é permitido a declaração de componentes em *procedures* e *functions*. Sinais somente podem ser declarados em *procedures* e *function* quando esses forem declarados dentro de um *process* (PEDRONI, 2004). As ferramentas de síntese podem sintetizar as estruturas de *procedures* e *function* de forma *inline* onde as chamadas dos *procedures* e *function* são substituídas pelo corpo destas estruturas, replicando o hardware gerado, ou com a criação de uma componente independente para execução das *procedures* e *function* para o qual será necessário a definição de um protocolo de comunicação e regras de escalonamento (RAMACHANDRAN et al., 1993).

2.5 REQUISITOS NÃO-FUNCIONAIS

Segundo Sommerville (2007) os requisitos de sistemas podem ser divididos em requisitos funcionais, não funcionais e de domínio. Requisitos funcionais são aqueles que definem as funcionalidades do sistema de forma completa e concisa. Já os requisitos não funcionais se referem as restrições do

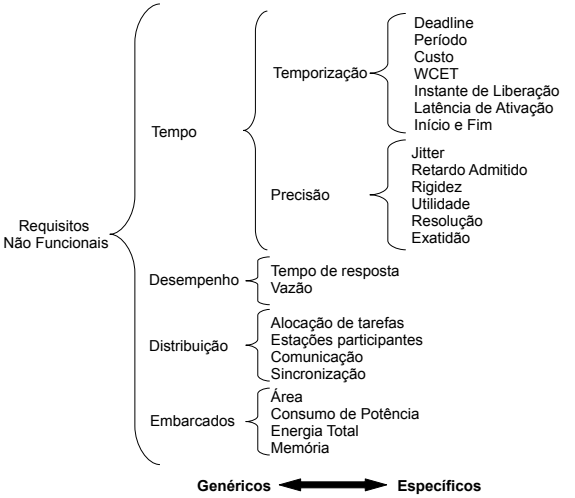
projeto que afetam uma ou mais de suas funcionalidades, tais como restrições de tempo, utilização de recursos ou padrões de desenvolvimento. São também requisitos não funcionais restrições como confiabilidade e integridade, não relacionados diretamente a uma funcionalidade. Enquanto que os requisitos de domínio estão relacionados ao domínio de aplicação, como tipos e tamanho de dados (SOMMERVILLE, 2007).

Sommerville (2007) divide os requisitos não funcionais em três tipos: (i) requisitos de produto, que definem o comportamento e qualidade do produto; (ii) requisitos organizacionais, relacionados à qualidade do projeto; e (iii) requisitos externos, que abrangem os comportamentos emergentes com a integração com outros sistemas. Bertagnolli (2004) propõem outra classificação dos requisitos não funcionais visando facilitar seu entendimento e a sua identificação no projeto, na qual os requisitos são descritos de forma genérica e então são subdivididos em requisitos específicos, permitindo um nível maior de abstração. Utilizando tal classificação, Bertagnolli (2004) propõem o *framework* FRIDA (*From RequIrements to Design using Aspects*) para gerenciar os requisitos do projeto e auxiliar na identificação e tratamento de requisitos não funcionais aplicando o paradigma orientado a aspectos. Baseado na proposta de Bertagnolli (2004), Freitas (2007) adaptou o FRIDA para uma classificação dos requisitos não funcionais para sistemas de tempo-real embarcados e distribuídos, criando a extensão RT-FRIDA. A classificação proposta por Freitas (2007) é apresentada na Figura 3. A classificação dos requisitos não funcionais proposta por Freitas (2007) foi utilizada por Wehrmeister (2009) para a construção do *framework* DERAF que serve como base para a ferramenta GenERTiCA.

Existem vários trabalhos que abordam o tratamento de requisitos não funcionais em sistemas embarcados. Entretanto, poucos trabalhos abordam o tratamento dos requisitos não funcionais no contexto de sistemas implementados em FPGA. As características especiais das FPGAs, tais como reconfigurabilidade e paralelismo, não são propriamente abordadas no projeto como um todo. Salewski e Taylor (2008) destacaram o impacto dos requisitos não funcionais em aplicações industriais baseadas em FPGA.

Com o objetivo de tratar os requisitos não funcionais em projetos implementados em FPGA, propôs-se um relacionamento entre os requisitos descritos em Freitas (2007) e as características das FPGAs, visando identificar os requisitos que tem maior impacto sobre essa plataforma. Essa relação e os requisitos não funcionais identificados a partir da análise da literatura são apresentados na Seção 4.1.

Figura 3 – Classificação dos requisitos não funcionais.



Fonte: (FREITAS, 2007).

2.6 ORIENTAÇÃO A ASPECTOS

A Programação Orientada a Aspectos (*Aspect-Oriented Programming, AOP*) foi proposta como uma solução para uma lacuna do paradigma orientado a objetos. Segundo Kiczales et al. (1997) observou-se que alguns requisitos podem influenciar diversas funcionalidades do projeto e eles não podem ser implementados como o comportamento de um único objeto. Esses requisitos, que estão presentes em todo o projeto, são denominados requisitos transversais como, por exemplo, o *log* do sistema. Em outras palavras esses requisitos atravessam (do inglês *cross-cutting*) as funcionalidades básicas do sistema (KICZALES et al., 1997). Como consequência, o código para tratamento desses requisitos acaba sendo espalhado por todo o projeto, o que leva a um código disperso e confuso (KICZALES et al., 1997).

A partir dessa ideia surgiu o paradigma de Desenvolvimento de Software Orientado a Aspectos (AOSD). Esse paradigma se baseia no conceito de *separação de preocupações*, que se referem a alguma característica ou área de interesse do projeto (ELRAD; FILMAN; BADER, 2001). Segundo Sommerville (2007), o AOSD se baseia na introdução da abstração *aspecto* ao paradigma orientado a objetos, que representam as *preocupações transversais* do projeto, ou seja, os requisitos que são implementados em várias funcionalidades ao mesmo tempo. As preocupações transversais estão geralmente

associadas ao tratamento de requisitos não-funcionais.

Kiczales et al. (1997) diferenciam aspectos de componentes, onde componentes são as funcionalidades que podem ser implementadas de forma geral em um único procedimento; já aspectos não podem ser encapsulados em um único procedimento. A implementação dos requisitos transversais utilizando abordagens tradicionais (e.g. orientação a objetos ou o paradigma estruturado) acaba levando ao espalhamento do código de tratamento por várias funcionalidades (do inglês, *Scattering*) e na ocorrência de diversos requisitos sendo tratados e implementados em uma mesma funcionalidade (do inglês *Tangling*). Esse espalhamento e o emaranhamento produzem um código com baixa coesão e alto acoplamento, reduzindo a possibilidade do seu reuso.

O objetivo da abordagem AOSD é separar o tratamento das preocupações transversais das outras funcionalidades do projeto, sendo que o programa final será a composição dos objetos, métodos e aspectos (SOMMERVILLE, 2007). Além disso, os aspectos indicam onde o tratamento das preocupações transversais será introduzido nas funcionalidades. O processo de unir os aspectos com as demais funcionalidades do projeto é denominada *entrelaçamento*. Essa separação permite melhorar a manutenção do código e diminuir erros de programação (ELRAD; FILMAN; BADER, 2001).

Na Listagem 2.4 são apresentados dois exemplos de requisitos entrelaçados em um código VHDL, nas linhas 1-5 e 21-25. Esse trecho de código implementa a funcionalidade do controle de velocidade e direção em um controlador para um motor elétrico. A linha 2 verifica o sinal de clock, para que o processamento execute somente quando o sinal estiver alto. Essa linha trata de um requisito transversal presente em todas as funcionalidades do sistema que operam em sincronia com o sinal de clock (alto ou baixo). Já a linha 4 verifica se o motor está ativo para efetuar o processamento. Esta validação é feita em todas as funcionalidades que executam somente quando o motor está ligado.

Listagem 2.4 – Exemplo de Código com requisitos transversais.

```
1  -- Requisito A
2  if ((Clk'event) and (Clk = '1')) then
3  -- Requisito B
4    if (Enable = '1') then
5    --
6       sensor_diff := conv_integer(ADdifRead);
7       if(sensor_diff > dif_theshold) then
8         if(ADdifSignal = '0') then
9           Left_Right <= left;
10        else
11          Left_Right <= right;
12        end if;
13        Walk_Turn <= turn;
```

```
14          Speed <= speed_turn;
15          MotorEn <= '1';
16        else
17         Walk_Turn <= walk;
18         Speed <= speed_walk;
19         MotorEn <= '1';
20        end if;
21     --
22     else
23        MotorEn <= '0';
24     end if;
25     --
26  end if;
```

Os requisitos implementados nas linhas 2 e 4 da Listagem 2.4 podem ser tratados em módulos independentes das funcionalidades do sistema, o que facilitaria sua manutenção e reutilização em outros projetos semelhantes. Devido ao espalhamento de código desses requisitos transversais pelas funcionalidades do sistema, se torna difícil a separação desses tratamentos. É nestas situações que a AOSD se aplica, permitindo a separação desses requisitos, sem perder a confiabilidade e características do sistema.

Segundo Rashid et al. (2010) a indústria vem adotando a utilização de AOSD em projetos de média e grande escala. Porém, a maioria dos projetos se concentra na utilização das características básicas de AOSD e na aplicação em requisitos transversais bem conhecidos como monitoramento do código, validação da arquitetura do sistema e *log* de erros (RASHID et al., 2010). Rashid et al. (2010) destacam que o maior desafio na adoção de AOSD é na sua aplicação em processos de desenvolvimento atuais, o que inclui no preparo e treinamento da equipe de desenvolvimento e gestores para a nova metodologia. Aqueles autores citam como principais benefícios na utilização de AOSD a simplificação e diminuição do modelo de dados e a estabilidade sobre as manutenções e mudanças do projeto. Entretanto, também destacam fragilidades na abordagem em consequência de problemas emergentes, relacionados aos *pointcuts* que são definidos de acordo com os nomes de classes, métodos, campos, entre outros, que podem sofrer alterações invalidando a definição dos *pointcuts* (RASHID et al., 2010). Além disso, Rashid et al. (2010) observaram uma falha na aplicação de *pointcuts* em projetos baseados em camadas, onde definições de uma camada afetaram outras que não deveriam ser afetadas.

A seguir são apresentados os principais conceitos da abordagem AOSD que serviram como base para construção da solução proposta. Esses conceitos são baseados nas definições apresentadas em (KICZALES et al., 1997; SOMMERVILLE, 2007; WEHRMEISTER, 2009).

- **Aspecto:** É abstração do requisito transversal que será implementado

por *adaptações* e *pointcuts*; A Listagem 2.5 apresenta a definição de um aspecto na linguagem AspectJ (KICZALES et al., 1997).

Listagem 2.5 – Exemplo de Aspecto definido na linguagem AspectJ (KICZALES et al., 1997).

```
1 public aspect AspectoExemplo{
2         // define o pointcut
3         pointcut point1(): execution(**());
4         //define a adaptacao
5         before():point1(){
6                 system.out.println("Adaptacao point1");
7         }
8 }
```

- **Adaptação:** Especifica as modificações que devem ser realizadas em um ou mais elementos do sistema para tratar um requisito transversal, encapsulando tal tratamento em um único componente. A adaptação pode ser de duas formas: (i) estrutural, se refere a adaptações na estrutura de um elemento; (ii) comportamental, que se refere a alteração no comportamento de um elemento; As linhas 5-7 (Listagem 2.5) apresentam um exemplo de definição de uma adaptação comportamental que adiciona uma mensagem aos elementos afetados. O termo "before" indica que a adaptação será introduzida antes do elemento selecionado.

- **Join Points:** Definem onde as adaptações devem ser aplicadas. Os *join points* são expressões de seleção de funcionalidades, métodos, classes, etc., sobre os quais serão aplicadas as adaptações, funcionando como filtros sobre os elementos da especificação do sistema. Alguns termos e curingas são utilizados para a definição de *join points*. Na Listagem 2.5 a linha 3 apresenta um exemplo de definição de um *join point*: "execution(**())". Esse *join point* seleciona a execução de todos os métodos do projeto. O caractere "*" funciona como um curinga indicando que qualquer outro caractere passará pelo filtro na posição em que foi utilizado.

- **Pointcuts:** Ligam as *adaptações* aos *join points*, i.e. definem qual adaptação ocorrerá para um determinado *join point*. Na Listagem 2.5 a linha 3 representa a definição de um *pointcut* que vincula a adaptação *point1* ao *join point* que seleciona a execução de todos os métodos.

- **Entrelaçamento do Aspecto:** (do inglês *Aspect Weaver*) é o processo de entrelaçamento dos aspectos com as funcionalidades do sistema, i.e., a aplicação das adaptações dos aspectos sobre as funcionalidades afetadas nos *join points* definidos.

2.7 A METODOLOGIA AMODE-RT

A metodologia AMoDE-RT (*Aspect-oriented Model-Driven Engineering for Real-Time systems*) proposta por Wehrmeister (2009), e as ferramentas de suporte a essa metodologia, foram estendidas neste trabalho para dar suporte a geração de código para a linguagem VHDL. Esta nova versão da metodologia foi utilizada para o desenvolvimento dos estudos de caso. Na metodologia AMoDE-RT, a modelagem do sistema alvo é realizada em um alto-nível de especificação utilizando a UML em conjunto com o perfil MARTE. O tratamento dos requisitos não funcionais é realizado usando conceitos de AOSD por meio dos diagramas ACOD (*Aspects Crosscutting Overview Diagram*) e JPDD (*Join Point Designation Diagrams*) e do *framework* DERAF (*Distributed Embedded Real-Time Aspects Framework*). MARTE possui estereótipos para a definição da estrutura e comportamento temporal dos sistemas de tempo-real embarcados e distribuídos. Esses mesmos conceitos podem ser utilizados na plataforma FPGA.

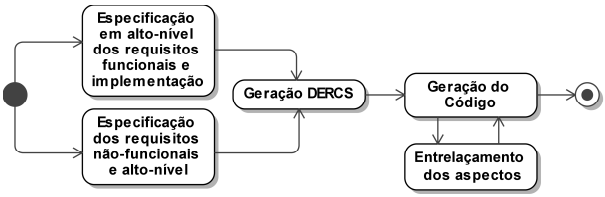
Os modelos especificados em alto-nível são transformados em um modelo intermediário denominado *Distributed Embedded Real-time Compact Specification (DERCS)* que é um PIM. O modelo DERCS é livre de ambiguidades e detalhado o suficiente para a geração de código a partir de seus elementos. Os elementos do modelo DERCS são transformados em código na linguagem alvo a partir de regras que mapeiam os elementos DERCS para elementos da linguagem alvo.

Na Figura 4 é apresentado o fluxo do processo de geração de código seguindo a metodologia AMoDE-RT. A atividade de "Geração do Código" é realizada com o uso de regras de mapeamento definidas e estruturadas em um arquivo de *scripts* no formato *eXtensible Markup Language (XML)*. O resultado desse processo é o código fonte na linguagem selecionada para a plataforma alvo. A ferramenta GenERTiCA (*Generation of Embedded Real-Time Code based on Aspects*) utiliza a metodologia AMoDE-RT para a geração do código. Na Figura 5 é apresentado um resumo das principais etapas no processo da ferramenta GenERTiCA.

A ferramenta GenERTiCA foi construída como um *plugin* para a ferramenta de modelagem Magic Draw[2]. O sistema é especificado na ferramenta Magic Draw, e, após, o *plugin* é acessado para a geração do código. A ferramenta GenERTiCA percorre todos os meta-elementos presentes no projeto selecionado, gerando o modelo DERCS. A partir desse ponto, a ferramenta GenERTiCA executa a leitura do arquivo XML que contém as regras de mapeamento para a linguagem alvo e executa o processo de geração de código baseado nestas regras.

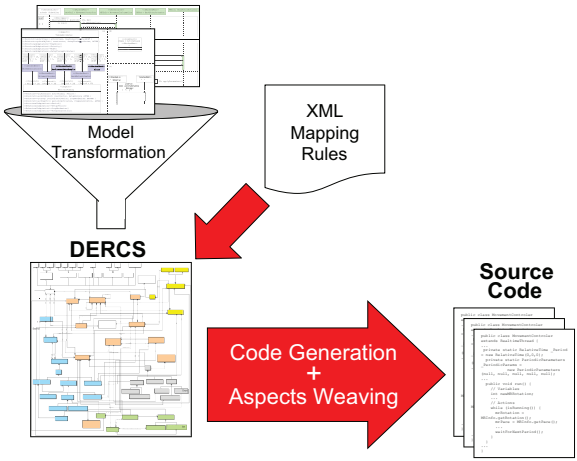
[2] http://www.nomagic.com/

Figura 4 – Fluxo do processo de geração de código.



Fonte: Produção do próprio autor.

Figura 5 – Etapas da ferramenta GenERTiCA.



Fonte: (WEHRMEISTER, 2009).

As regras para conversão dos elementos da UML para os elemento da linguagem alvo são especificadas em um arquivo de *scripts* no formato XML utilizando a linguagem *Template Velocity*[3]. Os *scripts* definem trechos de código VHDL que devem substituir os meta-elementos do modelo DERCS, gerados pela ferramenta GenERTiCA. As Listagens 2.6 e 2.7 apresentam um exemplo de regras de mapeamento e de código gerado. O *script* da Listagem 2.6 define as regras para o tratamento das definições dos métodos da classe. Esse bloco é utilizado na linguagem VHDL para a geração

[3] http://velocity.apache.org

das portas da entidade, selecionando todos os atributos que possuem métodos *get/set*. A Listagem 2.7 apresenta o resultado do processamento para uma classe chamada *Display*.

Listagem 2.6 – Exemplo de regra de mapeamento.

```
1 ...
2 #if ($Message.Name != $Class.Name )
3  #if($Message.isGetSetMethod())
4   #foreach ($msgGer in $Class.getMethods())
5    #if (!$msgGer.isGetSetMethod() and $msgGer.Name != $
         Class.Name)
6     #set($add = $lreadMes.add($DERCSHelper.
          getListOfAttributesUsed($Class, $msgGer.Name, 0)))
7     #set($add = $lwriteMes.add($DERCSHelper.
          getListOfAttributesUsed($Class, $msgGer.Name, 1)))
8    #end
9   #end
10  #set ($attr = $Message.getAssociatedAttribute())
11  #if ($attr.getDataType().getSize() > 1)
12   #set ($dtType = $attr.getDataType().getDataType())
13  #else
14   #set ($dtType = $attr.getDataType())
15  #end
16  #if (!$dtType.getRepresent())
17   #if($Message.getReturnType() == "Void")
18    #set($newDirection = "IN")
19    #foreach ($desLista in $lwriteMes)
20     #if ($desLista.contains($attr))
21      #set($newDirection = "INOUT")
22     #end
23    #end
24    #foreach($MessageGet in $Class.getMethods())
25     #if($MessageGet.isGetSetMethod() and $MessageGet.
           getReturnType() != "Void" and $MessageGet.
           getAssociatedAttribute().Name == $attr.Name)
26      #set($newDirection = "INOUT")
27     #end
28    #end
29    ;
30    \n $attr.Name : $newDirection $CodeGenerator.
          getDataTypeStr($attr.getDataType())
31    ...
```

Listagem 2.7 – Exemplo de código gerado para a classe *Display*.

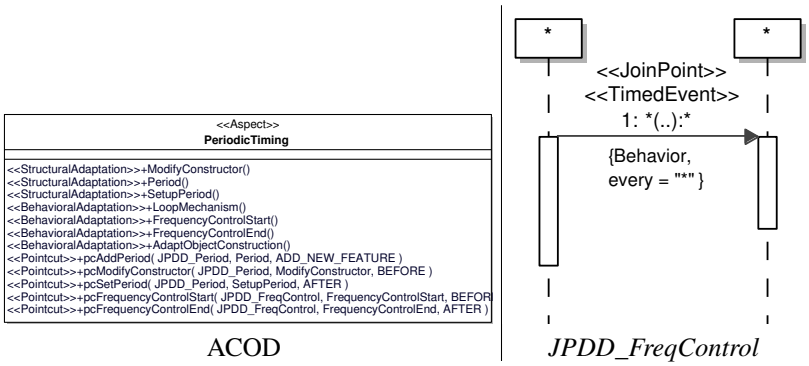
```
1 addsegmentosEnable : INOUT BIT;
2 digito : IN INTEGER;
3 segmentosled_0 : OUT BIT;
4 segmentosled_1 : OUT BIT;
5 segmentosled_2 : OUT BIT;
6 segmentosled_3 : OUT BIT;
```

```
7 segmentosled_4 : OUT BIT;
8 segmentosled_5 : OUT BIT;
9 segmentosled_6 : OUT BIT
```

As regras de mapeamento possuem validações com base no modelo DERCS que determinam o que será gerado pela ferramenta GenERTiCA na linguagem alvo. Assim, para cada linguagem alvo haverá um conjunto de regras de mapeamento. A ferramenta GenERTiCA também permite a definição de regras para a geração de arquivos de configuração para a plataforma alvo, com base nos aspectos especificados no modelo UML. Mais detalhes sobre a ferramenta GenERTiCA são apresentados em Wehrmeister (2009).

Wehrmeister (2009) propõe o diagrama ACOD como proposta para a especificação em alto nível de requisitos transversais com base na abordagem AOSD. O ACOD tem como base o diagrama JPDD (STEIN; HANENBERG; UNLAND, 2006), que define uma semântica para a construção de *joint points* de forma visual usando construções da UML (WEHRMEISTER, 2009). O ACOD possui dois níveis de definição: (i) uma visão macro com todos os aspectos que afetam uma determinada funcionalidade, sem detalhamento dos aspectos; e (ii) visão detalhada, onde são informados os detalhes de cada aspecto (i.e., os *pointcuts* e adaptações). O ACOD permite a definição de adaptações estruturais e comportamentais, por meio de diagramas de classe, enquanto que o diagrama JPDD permite a definição de *join points* por meio de diagramas de classe ou sequência (WEHRMEISTER, 2009). Na Figura 6 são apresentados exemplos de diagramas ACOD e JPDD.

Figura 6 – Exemplos de diagramas ACOD e JPDD.



Fonte: Produção do próprio autor.

No fragmento do diagrama ACOD da Figura 6 é apresentado o aspecto *PeriodicTiming* que especifica uma adaptação estrutural pelos métodos

Period, *ModifyConstructor* e *SetupPerid*, que são marcados com o estereótipo *StructuralAdaptation*. Os métodos *LoopMechanism*, *FrequencyControlStart* e *FrequencyControlEnd* são adaptações comportamentais, marcadas com o estereótipo *BehavioralAdaptation*. Os métodos marcados com o estereótipo *Pointcut* são os *pointcuts* que conectam as adaptações aos *join points*. No exemplo da Figura 6, o *pointcuts pcFrequencyControlStart* associa o *join point JPDD_FreqControl* a adaptação *FrequencyControlStart*. O *join point JPDD_FreqControl* seleciona todas as mensagens trocadas entre qualquer objeto (com o uso de curingas "*(..).*") que sejam marcadas como *TimedEvent*. Outros exemplos de *join points* são apresentados em (WEHRMEISTER, 2009).

3 REVISÃO DA LITERATURA

Este Capítulo apresenta uma revisão bibliográfica sobre o desenvolvimento de sistemas embarcados utilizando a plataforma FPGA, seus desafios e metodologias para lidar com a complexidade desse processo. Esta revisão busca identificar "como" foram solucionados os problemas encontrados e as "vantagens" e "desvantagens" dos trabalhos apresentados.

3.1 TRATAMENTO DE REQUISITOS NÃO-FUNCIONAIS

As FPGAs eram vistas inicialmente como ferramentas de prototipagem devido ao baixo desempenho em relação aos dispositivos convencionais (SHIMIZU et al., 2004; GHODRAT; LAHIRI; RAGHUNATHAN, 2007). Contudo, as FPGAs passaram a ser utilizadas como plataforma final de desenvolvimento (MONMASSON; CIRSTEA, 2007; SALEWSKI; TAYLOR, 2008; HUFFMIRE et al., 2008; MONMASSON et al., 2011; MALINOWSKI; YU, 2011). Huffmire et al. (2008) afirmam que, desde 2005 houve um grande aumento na utilização de FPGAs pela indústria como plataforma final para projetos de sistemas embarcados, incluindo sistemas críticos, como na indústria aeronáutica e militar. Salewski e Taylor (2008) citam as FPGAs como uma alternativa aos ASICs para projetos pequenos e médios devido ao aprimoramento do desempenho e a diminuição do custo ocasionado pela sua popularização. Entretanto, Monmasson et al. (2011) afirmam que ainda há um receio na utilização das FPGAs como hardware final pela indústria porque as ferramentas de desenvolvimento não estão maduras o suficiente.

Com a nova abordagem de utilização das FPGAs como plataforma de desenvolvimento, novas pesquisas surgiram sobre o tratamento de suas características intrínsecas. Monmasson e Cirstea (2007) destacam a importância do tratamento dos requisitos não funcionais, como consumo de energia, gerenciamento de temperatura e confiabilidade em aplicações industriais de sistemas embarcados. A utilização de FPGAs permite um aumento do grau de liberdade para os projetistas, tornando assim, a atividade arquitetural mais intuitiva e, consequentemente, mais dependente das experiências do projetista. Assim, Monmasson e Cirstea (2007) propõem o uso da metodologia A^3 para o tratamento dos requisitos não funcionais, auxiliando na exploração de espaço de projeto, sem infringir nenhuma de suas restrições. A metodologia A^3 (*Algorithm Architecture Adequation*), possibilita ao projetista encontrar uma combinação que atenda a todos os requisitos não funcionais do projeto a partir da geração de um grafo de fluxo de dados (DFG) (do inglês, *Data Flow Graph*), no qual os nós representam as operações e as folhas representam as variáveis do algoritmo (CHARAABI; MONMASSON; SLAMA-

BELKHODJA, 2002). Grandpierre, Lavarenne e Sorel (1999) e Charaabi, Monmasson e Slama-Belkhodja (2002) aperfeiçoaram a metodologia A^3 visando a predição de comportamento em sistemas de tempo-real embarcados e distribuídos. Entretanto, essa metodologia está focada no balanceamento das restrições de projeto de forma eficiente, não possuindo ferramentas que possibilitem gerenciar e tratar essas restrições em fases mais iniciais do projeto.

Mei et al. (2001) abordam o tratamento de requisitos não funcionais em sistemas embarcados parcial e dinamicamente reconfiguráveis implementados com FPGAs. Sua proposta é lidar com os requisitos não funcionais em duas fases, antes e depois do particionamento dos componentes. Uma otimização é executada em um alto-nível de especificação, sendo esta na linguagem C e, após, é gerado o código VHDL já otimizado (MEI et al., 2001).

Papadimitriou, Dollas e Hauck (2011) apresentam um estudo sobre os fatores que impactam no tempo de reconfiguração das FPGAs. Para tanto, foi proposto um modelo para a estimativa do tempo de reconfiguração da FPGA de acordo com a arquitetura adotada para o projeto. Aqueles autores realizaram um comparativo entre o tempo de reconfiguração estimado e o real. O tempo de reconfiguração real foi obtido a partir da realização de experimentos com diferentes abordagens de implementação, validando assim o modelo proposto (PAPADIMITRIOU; DOLLAS; HAUCK, 2011).

Outro aspecto que vem recebendo atenção dos pesquisadores é a segurança dos dados e da arquitetura de hardware (i.e. o *bitstream*) nas FPGAs. Segundo Huffmire et al. (2008), essa preocupação está relacionada justamente com a flexibilidade de reconfiguração das FPGAs e também com as ferramentas utilizadas em seu projeto, que podem introduzir brechas de segurança. Tais brechas podem ser exploradas por usuários mal intencionados para comprometer o sistema. Essa preocupação se estende também aos componentes reutilizados no projeto, principalmente aqueles obtidos de fontes que não possuem um comprometimento com a segurança (HUFFMIRE et al., 2008). Já Liu et al. (2009) destacam o desafio em balancear o atendimento dos requisitos não funcionais de segurança com os demais requisitos não funcionais do projeto. O atendimento dos requisitos de segurança podem impactar severamente em restrições, como consumo de energia e área de chip, além do aumento do custo de desenvolvimento. Uma das dificuldades encontradas é na validação da segurança do sistema, principalmente quando o sistema possui vários componentes de diferentes fontes, devido ao número exponencial de testes necessários para cobrir todas as possibilidades de erros (LIU et al., 2009).

A confiabilidade do sistema é outro requisito não funcional destacado. Um dos fatores que impactam nesse aspecto são os *soft error*. Os *soft error* são ocasionados por interferências eletromagnéticas ou por partículas

do meio ambiente que podem levar à alteração e corrupção de dados. Esses erros podem ser identificados e corrigidos em tempo de execução. Porém, os *soft error* podem levar a um mal funcionamento do sistema, comprometendo a sua confiabilidade. Sistemas embarcados baseados em FPGAs são mais suscetíveis a *soft error* do que sistemas baseados em microcontroladores ou ASICs (TAHOORI et al., 2009; GHOLAMIPOUR et al., 2011). Tahoori et al. (2009) apresentam uma ferramenta para estimar a taxa de *soft error* em um projeto com FPGA, permitindo uma análise de sua confiabilidade. Essa ferramenta analisa as *netlists* e bits sensitivos, i.e., os bits de memória usados, além do tipo da memória, para determinar o tipo, origem e o impacto do erro sobre o projeto (TAHOORI et al., 2009). Gholamipour et al. (2011) avaliaram o tratamento dos *soft error* por meio de técnicas de tolerância a falhas e apresentaram uma abordagem dividida em duas etapas para o desenvolvimento desses projetos. Primeiro são explorados os valores adequados para os parâmetros da área de reconfiguração e de duplicação e, após, a aplicação desses valores na configuração do sistema (GHOLAMIPOUR et al., 2011).

Liu et al. (2009) apresentam uma abordagem para o tratamento de requisitos não funcionais de consumo de energia e tamanho de memória em FPGAs por meio da reutilização de dados. O objetivo é diminuir a quantidade de acessos a memória externa da FPGA com a reutilização de dados, pois o acesso a memória externa consome mais energia que o acesso a sua memória interna. Liu et al. (2009) propõem a utilização de um modelo de consumo de energia simples que fornece uma estimativa aproximada para os valores ideais de reutilização de dados e tamanho da memória, sem perder a fidelidade da informação.

Meyer-Baese et al. (2012) apresentam uma abordagem para o desenvolvimento de processadores específicos para sistemas embarcados (do inglês *Application-Specific Instruction-Set Processors - ASIP*) em plataformas configuráveis. Em seu trabalho, Meyer-Baese et al. (2012) comparam o desempenho, tamanho, taxa de vazão e a eficiência no consumo de energia entre as plataformas ASIC e FPGA. Para isso, utilizou-se como estudo de caso um circuito para compressão de sinais implementado em três diferentes arquiteturas para cada uma das plataformas. Segundo Meyer-Baese et al. (2012) é possível aumentar o desempenho do sistema com a otimização da utilização dos recursos, como o tipo de memória, e a redução na latência das operações executadas pelo hardware, sem ter como consequência o aumento do consumo de energia. Menos recursos como a memória e menor latência diminuem o consumo de energia, proporcionando um desempenho melhor. Devido a complexidade no desenvolvimento de processadores específicos, Meyer-Baese et al. (2012) propõem a utilização de ADLs (*Architecture Description Languages*), no qual o hardware é modelado em um alto nível

diminuindo a complexidade de desenvolvimento. Projeta-se a arquitetura do processador em um nível mais abstrato e de fácil entendimento por meio de ADLs, enquanto uma ferramenta se encarrega da geração do código e sua síntese. Para o desenvolvimento dos experimentos realizados, aqueles autores utilizaram a ferramenta LISA, que gera o código em Verilog ou VHDL (MEYER-BAESE et al., 2012).

Beltran, Guzmán e Sevillano (2010) propõem três métricas para avaliar a qualidade e requisitos não funcionais de projetos implementados em FPGAs. Estas métricas avaliam a eficiência, escalabilidade e robustez do projeto. A eficiência é analisada com base em duas métricas de projeto: (i) o custo em termos de taxa de utilização de recursos da FPGA, e (ii) a velocidade, calculada a partir do tempo de resposta e o número de unidades de processamento em um projeto (BELTRAN; GUZMáN; SEVILLANO, 2010). Para a análise da escalabilidade a métrica da taxa de vazão dos dados é utilizada, além da eficiência calculada dos requisitos não funcionais. Por último, para avaliar a robustez, são utilizadas as métricas de número de tarefas e a taxa de alocação das tarefas entre as unidades de processamento do projeto.

3.2 METODOLOGIAS DE DESENVOLVIMENTO

No surgimento das FPGAs comerciais em 1984 a 1990 (MAXFIELD, 2004), o modelo de desenvolvimento em prática era o modelo em cascata. Seguindo essa prática, Gajski e Vahid (1995) propuseram um modelo hierárquico para o desenvolvimento de sistemas embarcados. Segundo Gajski e Vahid (1995), sua metodologia proporcionaria alta produtividade no desenvolvimento, mantendo a consistência por todas as fases do projeto, por meio da verificação parcial e simulação, além da identificação dos requisitos não funcionais em um nível mais abstrato do projeto. Nesta abordagem são utilizadas métricas para avaliação dos requisitos não funcionais. Essas métricas são formadas a partir dos requisitos não funcionais como desempenho ou consumo de área, e, principalmente, pelas experiências do projetista, o que dificultava a sua obtenção.

Outro fator é a complexidade do projeto. Quanto maior o nível de abstração da modelagem, mais difícil se torna criar estimativas, devido as otimizações executadas por ferramentas e compiladores (GAJSKI; VAHID, 1995). Apesar desta metodologia contar com uma verificação parcial dos requisitos não funcionais em todas as fases, a validação do software em conjunto com o hardware pode ser feita somente ao final do projeto, pois o software é desenvolvido após o hardware de forma sequencial. Nesse caso, um erro no tratamento de um requisito não funcional pode ocasionar um grande impacto sobre o projeto. Gajski e Vahid (1995) propõem a simulação do hard-

ware para auxiliar a validação do software, entretanto tal simulação não é capaz de fornecer todas as variações ocasionadas em um sistema real.

Uma das alternativas para o tratamento dos requisitos não funcionais é a utilização de ferramentas de automação das atividades de desenvolvimento baseadas em técnicas de MDE e reutilização de componentes. Essa abordagem vem sendo cada vez mais utilizada em projetos de sistemas embarcados em diferentes plataformas (GOKHALE; BALASUBRAMANIAN; LU, 2004; DENG; SCHMIDT; GOKHALE, 2006; ANNE et al., 2009; CANCILA et al., 2010; DRIVER et al., 2010; QIU; ZHANG, 2011). Anne et al. (2009) observam que a engenharia de software baseada em componentes é uma necessidade atual para melhorar a produtividade e diminuir o tempo de projeto de sistemas embarcados. Quadri et al. (2012) destacam a vantagem na utilização da MDE quanto ao ganho de tempo obtido no projeto devido a geração automática de código, reutilização de componentes e aumento de produtividade.

Abordagens baseadas em MDE e componentes também vem sendo propostas para o projeto de sistemas embarcados implementados em FPGAs. Quadri, Meftali e Dekeyser (2008) apresentam a ferramenta GASPARD, que implementa técnicas da abordagem MDE, para o desenvolvimento de sistemas embarcados com FPGAs. Essa ferramenta utiliza os conceitos da visão lógica do pacote HRM (*Hardware Resource Modeling*) do perfil MARTE, com a inclusão de dois novos atributos nesse pacote, para suporte a funcionalidade de reconfiguração parcial dinâmica. O objetivo é eliminar o *gap* de produtividade entre o desenvolvimento do hardware e software. A ferramenta GASPARD gera o código VHDL automaticamente a partir da modelagem do sistema em alto nível (QUADRI; MEFTALI; DEKEYSER, 2008; QUADRI; MEFTALI; DEKEYSER, 2010). Outro aspecto destacado por Quadri, Meftali e Dekeyser (2010), é o foco na modularização do sistema, sua comunicação e controle, permitindo a integração entre componentes definidos pelos usuários e componentes de terceiros. Assim, a ferramenta permite que o projetista explore as diversas possibilidades de componentes que melhor atendam os requisitos não funcionais do projeto.

Elhaji et al. (2012) apresentam uma metodologia para a modelagem de sistemas embarcados com estruturas repetitivas com base no pacote RSM (do inglês *Repetitive Structure Modeling*) da UML/MARTE. Naquele trabalho também apresentam uma proposta de mapeamento de elementos da UML para VHDL. Elhaji et al. (2012) utilizam o projeto GASPARD para o desenvolvimento e aplicação da metodologia proposta. Para a geração do código VHDL, a ferramenta GASPARD utiliza modelos de código VHDL pré-definidos, como modelos de definição de uma entidade e arquitetura (ELHAJI et al., 2012). O código gerado compreende a estrutura da entidade e mapea-

mento de componentes. Não possui tratamento para a geração de comportamento para as arquiteturas geradas. A modelagem das portas é feita com a adição de portas as classes. A abordagem proposta é aplicada na modelagem de topologias de redes em chip (NoC, do inglês *Network on Chip*). Elhaji et al. (2012) utilizaram as métricas de velocidade e área para avaliar os estudos de caso desenvolvidos. Para a análise da área, os autores comparam o número de *slices*, *Look-UP Tabels* (LUTs) e *Flip-Flops* (FFs) da síntese do código manual e do código gerado pela ferramenta GASPARD. Elhaji et al. (2012) observaram a mesma quantidade de recursos utilizado por ambas as versões.

Wang et al. (2008) apresentam a ferramenta COLA (*COmponent LAnguage*), baseada na MDE, que possui uma linguagem com semântica formal para definição de fluxos de dados em paralelo. A ferramenta COLA possui uma interface gráfica, na qual é possível a modelagem em alto-nível e a geração de código VHDL automaticamente. COLA permite descrever tanto a estrutura quanto o comportamento do hardware por meio de máquina de estados. No entanto, COLA não possui tratamento para os requisitos não funcionais e não há suporte para a modelagem com a UML (WANG et al., 2008).

Ebeid, Quaglia e Fummi (2012) propõem uma metodologia baseada em UML/MARTE para a geração automática de código VHDL a partir de diagramas de sequência para a validação de restrições de projeto. Aquela metodologia inclui a geração de verificadores de restrições de projeto no código VHDL. Restrições definidas por estereótipos do MARTE nos diagramas de sequência são utilizadas para a geração de componentes de verificação do projeto. Essa definição é feita na linguagem VSL (do inglês *Value Specification Language*) fornecida pelo perfil MARTE para a especificação das restrições temporais (EBEID; QUAGLIA; FUMMI, 2012).

Naquele trabalho os autores apresentam a validação de restrições de tempo sobre o projeto. Ebeid, Quaglia e Fummi (2012) não geram o código completo com sua metodologia. A proposta é gerar somente os arquivos de validação das restrições do projeto. Esses arquivos são conectados com componentes, previamente definidos em uma biblioteca, que implementam as funções do sistema. A metodologia segue com a simulação dos arquivos gerados a partir de *Test benches*[1] e a geração de um *log* sobre o cumprimento das restrições. O *log* de simulação permite validar e refinar o projeto. A metodologia proposta por Ebeid, Quaglia e Fummi (2012) não inclui tratamento para as restrições no código que implementa as funcionalidades do projeto. Assim, sempre que uma restrição não é cumprida, é necessário que uma manutenção corretiva seja feita manualmente no projeto.

[1] Test Benche em VHDL, são entidades e processos que geram estímulos para a simulação de projetos.

3.3 PROGRAMAÇÃO ORIENTADA A ASPECTOS EM SISTEMAS EMBARCADOS

Uma das abordagens propostas atualmente é a utilização do paradigma orientado a aspectos para o tratamento de requisitos não funcionais em sistemas embarcados (ENGEL; SPINCZYK, 2008). A programação orientada a aspectos permite separar o tratamento dos requisitos funcionais dos não funcionais melhorando a produtividade e reutilização do código. A programação orientada a aspectos, unida com técnicas da MDE, também vem sendo proposta para projetos baseados em FPGA.

Bainbridge-Smith e Park (2005) propõem uma linguagem para descrição de hardware em FPGAs chamada ADH (*Aspect Described Hardware-Description-Language*), com base na abordagem AOP e na linguagem AspectJ. O objetivo desta linguagem é separar o tratamento dos requisitos não funcionais das funcionalidades do sistema para projetos implementados em FPGAs. Em Park (2006), um compilador capaz de interpretar a especificação em ADH e transformá-la em código VHDL sintetizável foi desenvolvido. Após a realização de testes, Park verificou que o compilador possuía vários defeitos e restrições, não sendo possível a direta implementação do código VHDL nas FPGAs.

Engel e Spinczyk (2008) avaliam a aplicabilidade e adaptação da AOP para desenvolvimento de hardware na linguagem VHDL. Aqueles autores destacam as adaptações necessárias na AOP à natureza concorrente do hardware, observando o contraste de sua utilização nesse domínio de aplicação em relação com a programação orientada a objetos. Engel e Spinczyk (2008) definem que possíveis implementações de *join points* são em *process* e na atribuição de valores a sinais. Esses locais são apropriados para inserção de adaptações *before*, *after* e *around*. Também observaram que para VHDL há uma vantagem na definição de *pointcuts* devido ao paralelismo, quando na aplicação de curingas (ENGEL; SPINCZYK, 2008). Entretanto, não apresentam uma avaliação do impacto da aplicação desta abordagem sobre o projeto.

Driver et al. (2010) propõem uma metodologia MDE para o desenvolvimento de sistemas embarcados utilizando AOP e orientação a objetos. A modelagem é realizada em UML e na extensão Theme. Theme é um perfil que adiciona conceitos de aspectos, separação de requisitos e modularização das funcionalidades e dos requisitos não funcionais com o uso do relacionamento de composição (DRIVER et al., 2010). Driver et al. (2010) apresentam uma cadeia de ferramentas, denominada Theme/UML que implementa essa metodologia. Theme/UML permite o desenvolvimento em três fases: (i) modelagem, onde são especificados os requisitos funcionais e não funcionais do projeto; (ii) composição, onde são unidos (entrelaçados) os componentes do

projeto, inclusive os aspectos que tratam dos requisitos não funcionais, por meio de um plugin; e (iii) transformação, onde acontece a geração do código na linguagem C (DRIVER et al., 2010). Theme/UML também tem suporte aos metamodelos da linguagem de verificação de hardware *e* que implementa conceitos do pacote MARTE (LINEHAN; CLARKE, 2012).

Muck et al. (2011) apresentam a implementação de um escalonador de tarefas em hardware, desenvolvido na linguagem SystemC, com o uso de conceitos da Programação Orientada a Objetos (POO) e da AOP. Para essa implementação, foram utilizadas apenas instruções sintetizáveis do SystemC. A especificação SystemC é usada para gerar o código em VHDL (MUCK et al., 2011). Como SystemC é uma linguagem utilizada para a simulação de sistemas embarcados, Muck et al. (2011) propõem a sua utilização devido a sua semelhança com a UML e ao fato de ambas seguirem o paradigma orientado a objetos. Tal semelhança permite o aumento da produtividade e o uso de ferramentas para a geração de código sintetizável a partir de modelos UML. A AOP foi aplicada com o uso de abstrações da orientação a objetos, como herança e interfaces. Com isso, Muck et al. (2011) conseguiram isolar os mecanismos de tratamentos dos requisitos não funcionais. Para diminuir o acoplamento entre classes, aplicaram-se padrões de projeto como, por exemplo, o *Adapter*, diminuindo as dependências entre as classes e melhorando a sua reutilização e a qualidade do projeto (MUCK et al., 2011). Muck et al. (2011) observaram que a abordagem proposta não prejudicou o desempenho do sistema, entretanto, consumiu 3% a mais de recursos de hardware para implementação.

Petrov et al. (2011) propõem a utilização da AOP e estratégias de implementação do tratamento dos requisitos não funcionais no projeto REFLECT. O objetivo é diminuir a sobrecarga de tempo de projeto causada pela implementação dos requisitos não funcionais e erros de codificação causados pela adaptação manual do código. REFLECT conta com uma ferramenta baseada na MDE que recebe como entrada a especificação do sistema descrita na linguagem C e as definições dos aspectos e estratégias, descritos na linguagem LARA. REFLECT processa o código e une os aspectos às funcionalidades de acordo com as estratégias e padrões de projeto definidos em um processo chamado *entrelaçamento*. Os padrões de projeto determinam a forma de transformação do código de acordo com as necessidades dos aspectos definidos. REFLECT ainda conta com a utilização de *templates* (ou componentes) que são utilizados de acordo com os padrões de projeto aplicados na transformação do código. Para a geração do código final, REFLECT efetua o particionamento das funcionalidades do projeto, definindo o que será implementado em software e o que será implementado em hardware. Para as funcionalidade definidas como hardware, REFLECT gera a descrição em

VHDL (PETROV et al., 2011). Petrov et al. (2011) executou a análise de Markov[2] para avaliar a confiabilidade do sistema com a abordagem proposta.

Cardoso et al. (2012) apresentam a linguagem LARA, para o tratamento de requisitos não funcionais em sistemas embarcados implementados com FPGAs e microcontroladores. Segundo Cardoso et al. (2012), entre os principais desafios no projeto de sistemas embarcados está o particionamento das funcionalidades entre os componentes do projeto e a exploração de espaço de projeto. Por meio da definição de estratégias e padrões de projeto, LARA auxilia na exploração de espaço de projeto, onde os desenvolvedores podem experimentar diferentes soluções de forma automatizada, mantendo uma única especificação do sistema (CARDOSO et al., 2012). Para a otimização e adaptação do código aos requisitos não funcionais, a linguagem LARA utiliza a definição de *join points*, *pointcuts* e *join point attributes*. Os *join point attributes* permitem a definição de valores de atributos que servem para a comunicação desses atributos entre diferentes estágios do projeto, além de servirem como filtros para os *join points*. As estratégias, por sua vez, servem para determinar quando e como os aspectos serão aplicados no entrelaçamento do código, junto com os padrões de projeto, buscando a otimização do código e arquitetura do projeto (CARDOSO et al., 2012).

Segundo Coutinho et al. (2012), LARA permite a injeção de código referente ao tratamento dos aspectos, de forma não invasiva, instrumentando e monitorando o sistema. A principal vantagem nesta separação de interesses é a portabilidade para outras plataformas conforme as estratégias de implementação especificadas no LARA (COUTINHO et al., 2012). Para validar o fator de impacto dos aspectos sobre o código final, LARA foi avaliada por algumas métricas, tais como o número de linhas de código (do inglês *Lines of Code (LOC)*), a difusão das preocupações sobre as linhas de código (do inglês *Concern Diffusion over Lines of Code (CDLOC)*) (FIGUEIREDO et al., 2008), inchaço do código com os aspectos (LOPES, 1997), taxa de entrelaçamento (LOPES, 1997) e o percentual de funções que foram afetadas por um dado aspecto. Essas métricas auxiliam a avaliar a portabilidade e reutilização do tratamento de um requisito não funcional.

Meier, Hanenberg e Spinczyk (2012) apresentam o desenvolvimento da linguagem chamada AspectVHDL, que é uma extensão da linguagem VHDL voltada para a abordagem AOP. AspectVHDL busca permitir a separação do tratamento de requisitos não funcionais, os quais levam, muitas vezes, a espalhar trechos repetidos de código por todo o projeto. AspectVHDL permite a definição de *join points* em procedimentos, funções, definição de tipos, arquitetura e lista de variáveis sensitivas de processos (MEIER; HA-

[2] Análise que determina o número de possíveis falhas em um sistema crítico a partir de cálculos estatísticos.

NENBERG; SPINCZYK, 2012). Para utilização eficiente dessa abordagem, é necessário que o código VHDL esteja estruturado como, por exemplo, utilizando funções para códigos recorrentes, o que facilita a introdução de *join points*. Meier, Hanenberg e Spinczyk (2012) apresentam a sintaxe e semântica bem definidas para a linguagem proposta. Aqueles autores preveem o desenvolvimento de um terceiro estágio para a adição de *pointcuts* em processos, permitindo, por exemplo, a alteração de estados em máquinas de estados (MEIER; HANENBERG; SPINCZYK, 2012). Entretanto, Meier, Hanenberg e Spinczyk (2012) não apresentam validação ou experimentação da linguagem proposta, nem ferramenta de suporte para linguagem AspectVHDL.

3.4 TRABALHOS BASE

Wehrmeister (2009) propôs a abordagem AMoDE-RT que utiliza técnicas da MDE e AOSD para o desenvolvimento de sistemas de tempo-real embarcados e distribuídos. O foco dessa abordagem é no tratamento e gerenciamento dos requisitos não funcionais utilizando o *framework* DERAF que classifica e especifica a implementação desses requisitos com base no paradigma orientado a aspectos (WEHRMEISTER, 2009). Naquele trabalho desenvolveu-se um conjunto de regras de mapeamento para a geração de código em linguagens de alto nível como Java e C++. Moreira (2012) estendeu aquele trabalho criando regras de mapeamento para a geração de código VHDL a partir do modelo, entretanto o tratamento para os aspectos relacionados aos requisitos não funcionais do projeto não foi abordado.

Moreira (2012) examinou na literatura os conceitos mais utilizados para a transformação de artefatos da UML para VHDL. A partir desse estudo, aquele autor definiu um conjunto de regras de mapeamento para VHDL a partir de modelos UML (ver Tabela 1) utilizando a abordagem já proposta por Wehrmeister (2009).

Tabela 1 – Conceitos mapeados em Moreira (2012).

Elemento UML	**Elemento VHDL**
Classe	Par Entidade-Arquitetura
Atributos Públicos	Portas da Entidade
Atributos Privados	Sinais
Métodos	Processos
Troca de Mensagens	Portas da Entidade
Associação entre classes	Portas da Entidade

Fonte: (MOREIRA, 2012).

O conjunto de regras proposto por Moreira (2012) considera apenas os diagramas de classe e sequência, relacionados a, respectivamente, arquitetura e ao comportamento do hardware. Naquele trabalho foram implementados apenas relacionamentos de associação 1-para-1. Moreira (2012) propôs uma tradução para VHDL dos relacionamentos de herança e polimorfismo, porém não os implementou. Outros conceitos não implementados são auto-relacionamentos, associações um-para-muitos e atributos complexos como, por exemplo, tipos enumerados e vetores. Além disso, é muito importante para abordagens MDE que a modelagem do sistema em UML siga padrões comuns a outras plataformas, ou seja, o modelo deve ser independente da plataforma. Entretanto, com as regras propostas por Moreira (2012), o modelo fica restrito a elementos e condições presentes unicamente na plataforma FPGA como, por exemplo, a atribuição de sinais em vez de troca de mensagem entre objetos como especificado em modelos UML.

3.5 DISCUSSÃO

Na análise da literatura percebem-se várias propostas que sugerem o uso de FPGAs como uma solução para o desenvolvimento de sistemas embarcados, visando a portabilidade e flexibilidade do hardware. Com isso surgem novos desafios para o desenvolvimento de sistemas embarcados ao considerar as características e comportamento da FPGA. Entre eles, destaca-se o tratamento de requisitos não funcionais em um nível mais abstrato de desenvolvimento, onde não há muitos detalhes de implementação e é necessário a tomada de decisões sobre a arquitetura do projeto. Entre os requisitos não funcionais abordados nas pesquisas analisadas, destacam-se: tempo de reconfiguração para sistemas dinamicamente reconfiguráveis, memória, área, consumo de energia, segurança, confiabilidade e comunicação entre componentes.

Para melhorar o desenvolvimento de sistemas implementados em FPGA, são propostas abordagens e ferramentas baseadas em técnicas de MDE. Essas propostas tem o objetivo de aumentar a produtividade, por meio da automatização do processo de desenvolvimento, além de diminuir o tempo de projeto com a reutilização de componentes. Quadri, Meftali e Dekeyser (2008) propõem a ferramenta GASPARD, que permite a geração do código VHDL a partir de modelos especificados na UML e MARTE. No entanto, a abordagem proposta não trata dos requisitos não funcionais que precisam ser definidos juntamente com as funcionalidades ou na modificação do código VHDL para inserção dos tratamentos. Já Wang et al. (2008) propõem a ferramenta COLA, semelhante a GASPARD, porém COLA não possui suporte a UML, pois se baseia em uma semântica própria além de também não possuir

tratamento para os requisitos não funcionais.

Para o tratamento dos requisitos não funcionais, analisaram-se abordagens baseadas no paradigma orientado a aspectos, que tem como objetivo separar o tratamento dos requisitos não funcionais do desenvolvimento das funcionalidades do sistema. As abordagens orientadas a aspectos possibilitam a reutilização desses tratamentos, melhorando a manutenção do projeto (BAINBRIDGE-SMITH; PARK, 2005; PARK, 2006; ENGEL; SPINCZYK, 2008; MUCK et al., 2011; MEIER; HANENBERG; SPINCZYK, 2012). Petrov et al. (2011) e Cardoso et al. (2012) apresentam o projeto REFLECT, que tem como base a linguagem orientada a aspectos chamada LARA. No entanto, esta abordagem não possui suporte para a especificação do projeto em níveis mais abstratos (e.g. modelos UML), concentrando-se na implementação a nível de linguagem de descrição de hardware. A linguagem AspectVHDL, proposta por (MEIER; HANENBERG; SPINCZYK, 2012) não está bem definida e fica dependente da estrutura do código VHDL para a aplicação dos aspectos.

A metodologia Theme/UML proposta por Driver et al. (2010) possui algumas limitações, indicadas pelos próprios autores, como a falta de suporte para tratar os requisitos não funcionais de tempo e comunicação. Além disso, a ferramenta não possui suporte ao perfil MARTE, padrão definido pela OMG[3] para sistemas embarcados. A semântica proposta permite a definição ambígua de restrições, o que pode prejudicar o projeto e causar erros detectados somente após a implementação física do mesmo, uma vez que as restrições duplicadas são unificadas durante a fase de composição, ficando ocultas ao desenvolvedor. Por fim, a abordagem e ferramenta não possuem suporte para a geração de código VHDL.

Engel e Spinczyk (2008) apresentam possíveis implementações de *pointcuts* para VHDL em processos e na atribuição de valores a sinais. Esses locais são cobertos pela metodologia apresentada neste trabalho. Observa-se neste trabalho, que não há relação entre a definição de *pointcuts* e o paralelismo inerente na linguagem VHDL como mencionando em (ENGEL; SPINCZYK, 2008), uma vez que esta especificação é elaborada em um nível abstrato, independente da plataforma alvo.

Meier, Hanenberg e Spinczyk (2012) apresentam como locais para *pointcuts* procedimentos, funções, definição de tipos, arquitetura e lista de variáveis sensitivas de processos. Conforme discutido na Seção 4.3.2, procedimentos e funções não são tratados na abordagem proposta, por não serem compatíveis com os elementos da orientação a objetos. Os demais itens apresentados estão presentes na abordagem proposta. Meier, Hanenberg e Spinczyk (2012) não possuem *join points* para processos, dessa forma, a

[3] Object Management Grout.

metodologia proposta por aqueles autores requer que o código VHDL esteja estruturado em funções e procedimentos. Esta restrição não existe para a abordagem proposta neste trabalho, uma vez que a especificação do sistema e aspectos é elaborada em um nível abstrato, independente da plataforma alvo.

Os trabalhos apresentados na Seção 3.4 servem de base para esse trabalho que visa estender a abordagem apresentada em Wehrmeister (2009) incluindo suporte a plataforma FPGA e linguagem VHDL. A proposta desse trabalho é implementar o tratamento de requisitos não funcionais em sistemas embarcados desenvolvidos em FPGA com o uso de técnicas AOSD e MDE, a qual não foi desenvolvida no trabalho de Moreira (2012). Além disso, esse trabalho estende a abordagem AMoDE-RT, pela inclusão do pacote *FaultHandling*, com o aspecto *COPMonitoring*, para o tratamento de falhas em sistemas embarcados. Conforme identificado na revisão da literatura, requisitos de segurança são muito importantes para sistemas implementados em FPGA. Adicionalmente, tratamentos para novos estereótipos do perfil MARTE foram adicionados na abordagem, permitindo a criação de *join points* mais genéricos, para o tratamento dos requisitos não funcionais.

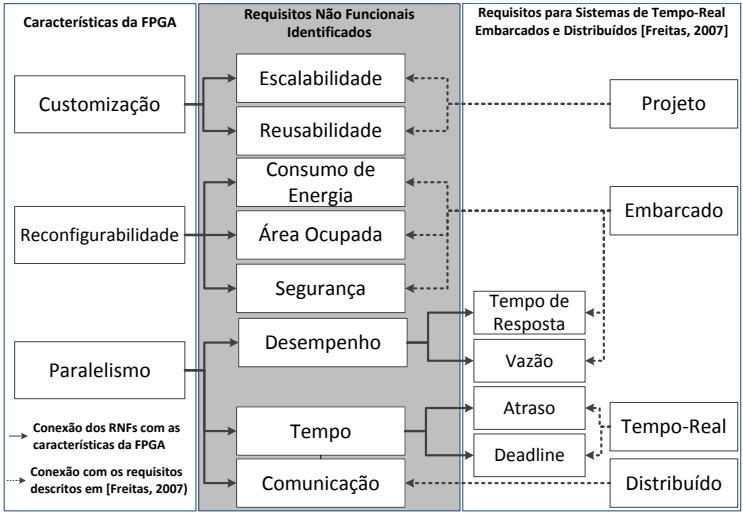
Esse trabalho utiliza como base os conceitos apresentados em Moreira (2012) para a geração de código VHDL. Entretanto, acrescenta e altera esses conceitos e as regras de mapeamento dos elementos da UML para VHDL (apresentados na Seção 4.3.1), visando melhorar o suporte da modelagem em alto-nível pelo uso da orientação a objetos na especificação de sistemas embarcados implementados em FPGA. Além disso, um conjunto de *pointcuts* foi definido, com base na revisão da literatura e da implementação deste projeto.

4 DESENVOLVIMENTO

4.1 ANÁLISE E AVALIAÇÃO DOS REQUISITOS NÃO-FUNCIONAIS PARA PROJETOS BASEADOS EM FPGA

Com base na revisão da literatura realizada (apresentada no Capítulo 3) foram extraídos os principais requisitos não funcionais para projetos de sistemas embarcados implementados em FPGA. Na Figura 7 é apresentado um resumo desses requisitos e o seu relacionamento com alguns requisitos não funcionais de sistemas de tempo-real embarcados e distribuídos (FREITAS, 2007). As linhas sólidas indicam as conexões dos requisitos não funcionais com as características da FPGA. Esses requisitos tem uma influência direta sobre o desempenho do sistema. Do outro lado, as linhas pontilhadas indicam a relação do requisito não funcional selecionado com os requisitos descritos em Freitas (2007).

Figura 7 – Requisitos Não-Funcionais para Projetos baseados em FPGA.



Fonte: Produção do próprio autor.

A identificação desses requisitos não funcionais como importantes para sistemas embarcados implementados em FPGA, levou em consideração as preocupações observadas nos projetos analisados na revisão da literatura (ver Capítulo 3). Estes requisitos são tratados de diferentes formas em projetos diferentes. Entretanto, com a abordagem proposta, o objetivo é criar um

padrão de tratamento para estes requisitos não funcionais que seja genérico o suficiente para serem introduzidos em diferentes plataformas e domínios de aplicação, além de possibilitar a extensão para implementações mais específicas.

Na Seção 4.2 é apresentado um conjunto de métricas para avaliar cada requisito não funcional selecionado. Este conjunto de métricas foi elaborado com base no estudo da revisão da literatura. Estas métricas servirão para auxiliar na implementação dos aspectos que os tratarão, assim como na avaliação desse tratamento.

Pode não ser viável o tratamento de requisitos como escalabilidade ou reusabilidade por meio de aspectos, mas estes requisitos podem ser controlados com o uso das métricas propostas. A reusabilidade e escalabilidade podem ser alcançadas pela separação, tanto das funcionalidades quanto dos requisitos não funcionais, em módulos que poderão executar em diferentes componentes de hardware. Já requisitos não funcionais como, por exemplo, os relacionados ao consumo de energia, podem ser melhores tratados e gerenciados com o uso de aspectos implementados sobre as funcionalidades e, ainda, serem controlados por meio de métricas.

4.1.1 Escalabilidade

Escalabilidade pode ser entendida como a propriedade de um sistema ou aplicação que permite o crescimento contínuo sem afetar seu desempenho (TANENBAUM; RENESSE, 1985). Paralelizar funções ou tarefas é uma técnica que permite que sistemas sejam escaláveis, pois permite aumentar o número de tarefas executadas (HILL, 1990). Devido ao paralelismo intrínseco das FPGAs, onde cada tarefa ou módulo executa concorrentemente, pode-se dizer que projetos implementados em FPGAs são escaláveis (WOLF, 2004).

Segundo Beltran, Guzmán e Sevillano (2010), para analisar a escalabilidade de uma nova configuração de arquitetura em uma FPGA, é necessário comparar o seu desempenho em relação ao atual, permitindo avaliar o seu crescimento em termos de utilização de recursos da FPGA e a possibilidade de futuras expansões de arquitetura. Beltran, Guzmán e Sevillano (2010) propõem como métrica para avaliar a escalabilidade a razão entre duas medições (i.e., o valor novo e o anterior) de um mesmo atributo que representa um requisito não funcional. Esse atributo pode ser a velocidade de processamento, consumo de energia, área ocupada ou qualquer outra restrição que necessite ser controlada. O aumento de desempenho do projeto geralmente impacta em outro requisito não funcional como, por exemplo, no aumento de consumo de energia ou no aumento da área ocupada da FPGA.

Em relação a sistemas embarcados implementados em FPGA, pode-se determinar como métricas para análise da escalabilidade a quantidade de recursos utilizados da FPGA, como os número de *Occupied Slices* (OS), *Input/Output Blocks* (IOB), FFs e LUTs (BELTRAN; GUZMáN; SEVILLANO, 2010), em função da velocidade de processamento da lógica implementada, i.e. a Frequência Máxima (MF) (do inglês, *Maximum Frequency*) daquele circuito.

4.1.2 Reusabilidade

O reuso e a modularização do código de tratamento dos requisitos não funcionais são as principais vantagens da abordagem AOSD, conforme mencionado na literatura (CARDOSO et al., 2012; PETROV et al., 2011; ENGEL; SPINCZYK, 2008). Assim é importante conhecer o impacto dos aspectos sobre o projeto, além da taxa de reuso.

Cardoso et al. (2012) propõem algumas métricas para medir o código como o Número de Linhas de Código (LOC) (do inglês, *Lines of Code*), Número de Linhas da Adaptação (LOAC) (do inglês, *Lines of Adaptation Code*), a Taxa de Entrelaçamento (TR) (do inglês, *Tangling Ratio*) do código e o Número de Linhas do Código Entrelaçado (LOWC) (do inglês, *Lines of Woven Code*), a difusão do aspecto sobre o código (CDLOC) (do inglês, *Concern Diffusion over Lines of Code*,índice de impacto do aspecto (AB) (do inglês, *Aspectual Bloat*) e o percentual de funcionalidades afetadas (FIGUEIREDO et al., 2008). Para as FPGAs estas métricas também indicam consumo de hardware, que pode implicar em mais área ocupada, aumento no consumo de energia e maior atraso do circuito implementado na FPGA. Tal análise e exemplos de como utilizar estas métricas são apresentados no Capítulo 5.

4.1.3 Consumo de Energia

Segundo Wolf (2004) é possível controlar restrições de consumo de energia em projetos baseados em FPGA a partir da área ocupada e o comportamento temporal do sistema. Também é possível gerenciar restrições de consumo de energia por meio de otimizações em blocos de lógica sequencial. Por exemplo, na implementação de uma máquina de estados é possível diminuir o consumo de energia aumentando a velocidade do circuito com o uso de paralelismo. De fato, a velocidade do circuito permanece a mesma. Porém utilizando componentes executando em paralelo, é possível melhorar o desempenho do sistema. Outro fator que impacta no consumo de energia é o *glitch*, presente em máquinas de estado, que pode ser diminuído com o uso de hardware extra (por exemplo, *flip-flops*) (WOLF, 2004)

Além disso, segundo Liu et al. (2009) e Meyer-Baese et al. (2012), o uso de memória *on-chip*[1] e *off-chip* [2] também impacta no consumo de energia. Por isso a definição correta dos tipos de dados e sua forma de armazenamento contribuem para a otimização do consumo de energia do sistema. Pode-se utilizar como métricas o Tamanho da Palavra (WS) (do inglês, *Word Size*), o Número de Acessos da Memória Interna (ANOnM) (do inglês, *Access Number of On-Chip Memory*) e o Número de Acessos da Memória Externa (ANOffM) (do inglês, *Access Number of Off-Chip Memory*).

É proposta a métrica de Consumo de Energia por Funcionalidade (ECF) (do inglês, *Energy Consumption per Functionality*). Uma métrica similar foi utilizada em (CORREA et al., 2010) para medir o consumo de energia por funcionalidades em projetos de software.

4.1.4 Área Ocupada

As restrições sobre a área utilizada nas FPGAs estão geralmente associadas com o desempenho do projeto, consumo de energia e a qualidade da localização e roteamento dos elementos da FPGA (GHOLAMIPOUR et al., 2011; BELTRAN; GUZMáN; SEVILLANO, 2010; WOLF, 2004). A taxa de utilização da área em um projeto de FPGA está relacionada a eficiência na especificação do circuito (SALEWSKI; TAYLOR, 2008). A FPGA consome energia para toda a área disponível, mesmo não sendo completamente ocupada pela especificação do circuito. Assim, quanto mais área utilizada para um projeto, mais eficiente é o uso de recursos como, por exemplo, o consumo de energia. Entretanto, uma maior utilização de área também pode indicar um baixo desempenho e baixa qualidade na localização e roteamento, resultando em um alto consumo de energia e aumento no atraso do circuito. Por isso é necessário medir a área ocupada em termos de quantidade de componentes utilizados da FPGA, como *Occupied Slices* (OS), LUTs e FFs e *Input/Output Blocks* (IOBs). Tais métricas também foram utilizadas em (MEIER; HANENBERG; SPINCZYK, 2012; CARDOSO et al., 2012; ELHAJI et al., 2012) para avaliar a versão do código VHDL com e sem o uso de conceitos de orientação a aspectos implementados. Estas métricas foram utilizadas na análise dos estudos de caso apresentados no Capítulo 5.

[1] On-Chip: Memória interna do circuito FPGA.
[2] Off-Chip: Memória externa do circuito FPGA.

4.1.5 Segurança

O requisito de disponibilidade[3] compreende a capacidade do sistema de continuar operando mesmo após uma falha ocorrer (com o uso de redundâncias ou da correção da falha de forma rápida, de modo que o tempo inoperante tenda a zero). O tratamento de *soft errors* é uma das principais preocupações em projetos implementados em FPGA (TAHOORI et al., 2009; GHOLAMIPOUR et al., 2011). Conforme apresentado no Capítulo 3, alguns indicadores que podem ser utilizados para controlar esse requisito é o Tamanho de Memória (MS) (do inglês, *Memory Size*), pois a memória utilizada para os bits de configuração da FPGA são suscetíveis a *soft errors*, além da quantidade de componentes sensitivos da FPGA como, por exemplo, o número de blocos de lógica e de conexão utilizados. O tratamento para detecção e correção de erros em projetos de FPGA acarreta em hardware extra para processamento e para redundâncias.

O requisito de confiabilidade compreende a perda e corrupção de dados. Segundo Petrov et al. (2011), a análise de *Markov* permite a identificação de possíveis estados de falha, o que pode ser utilizado para estimar a Taxa de Falhas (FR) (do inglês, *Fault Rate*), de modo semelhante ao que ocorre nas métricas de software para estimar o número de casos de teste.

A proteção dos dados e da arquitetura do circuito (i.e. o *bitstream* que define o circuito) são restrições críticas em projetos implementados em FPGA devido à possibilidade de reconfiguração durante a execução do sistema (WANDERLEY et al., 2011). De acordo com Wanderley et al. (2011), pode-se avaliar o nível de tecnologia de segurança disponível na FPGA para auxiliar na determinação do que será necessário implementar para proteger o sistema. Esse nível tecnológico pode ser entendido como uma métrica de Fator de Risco (RF) (do inglês, *Risk Factor*) ao qual o sistema poderá estar exposto. Wanderley et al. (2011) apresenta cinco níveis de classificação, que variam de acordo com os recursos proteção do circuito.

4.1.6 Desempenho

O desempenho de sistemas embarcados é geralmente medido pelo tempo de resposta e taxa de vazão dos dados, que refletem a capacidade de processamento do sistema (BELTRAN; GUZMáN; SEVILLANO, 2010). Porém para as FPGAs a taxa de vazão e tempo de respostas dependem do hardware disponível (quantidade de recursos da FPGA) e da lógica implementada.

Uma métrica que auxilia na análise de desempenho da lógica implementada é o Tempo do Maior Caminho Crítico (CPT) (do inglês, *Critical*

[3] Neste trabalho aspectos de disponibilidade do sistema, confiabilidade e proteção do projeto e dados da FPGA estão relacionados ao termo segurança.

Path Time) que pode ser utilizado para analisar o desempenho da lógica implementada como Wang et al. (2008) utilizou para analisar a versão manual do código VHDL em relação à versão gerada pela ferramenta COLA. Além disso, propõe-se verificar a Máxima Frequência (MF) (do inglês, *Maximum Frequency*) do circuito, dado o tempo de caminho crítico obtido. Estas métricas foram utilizadas para a análise de desempenho dos estudos de caso apresentados no Capítulo 5.

4.1.7 Atraso

Para os sistemas implementados em FPGA é necessário conhecer o atraso nas *netlists*, assim como considerar o potencial de paralelismo fornecido pela arquitetura da FPGA (WOLF, 2004). Em uma FPGA, o atraso varia dependendo da área utilizada no chip e do caminho crítico, que significa a maior distância do sinal de clock para a lógica construída, os quais irão impactar no Pior Caso de Atraso (WCD) (do inglês, *Worst Case Delay*) (WOLF, 2004). O atraso está relacionado com o desempenho do projeto, consumo de energia e *clock skew*[4]. Em projetos de FPGA, é possível controlar o atraso por meio de lógicas otimizadas e bem estruturadas, que diminuam o tamanho dos canais e componentes por quais passam um sinal.

4.1.8 Prazos

Cumprir os prazos de execução das tarefas depende fortemente do escalonamento e políticas de alocação do sistema, além da frequência do clock. Em projetos de ASIP usando FPGA, os projetistas não contam com um sistema operacional para gerenciar o escalonamento e alocação de tarefas. Assim, os projetistas precisam escalonar e alocar manualmente as tarefas, além de definir todo o comportamento temporal do sistema, sem esquecer das prioridades e dependências de dados. O WCET (do inglês, *Worst Case Execution Time*) é uma métrica que auxilia na determinação do cumprimento dos prazos para o circuito e instruções.

4.2 MÉTRICAS DE AVALIAÇÃO

Com base nos requisitos não funcionais apresentados na Seção 4.1, define-se um conjunto de métricas relacionados aos requisitos não funcionais em projetos de sistemas embarcados implementados em FPGA. Este conjunto de métricas (ver Tabela 2) permite uma avaliação dos indicadores dos

[4] Clock Skew é um atraso de chegada do sinal do clock nos diferentes componentes de uma lógica implementada no circuito.

requisitos não funcionais do projeto. Assim é possível gerenciar e controlar as restrições do projeto e seu impacto, assim como a eficiência da metodologia de projeto e soluções adotadas.

Tabela 2 – Métricas dos requisitos não funcionais para projetos de FPGA.

RNF STRE	**RNF FPGA**	**Métricas**
Projeto	Escalabilidade	Occupied Slices (OS) Input/Output Blocks (IOBs) Flip-Flops (FFs) Look-Up Tables (LUTs) Maximum Frequency (MF)
	Reusabilidade	Lines of Code (LOC) Lines of Adaptation Code (LOAC) Tangling Ratio (TR) Aspectual Bloat (AB) Lines of Woven Code (LOWC)
Embar-cado	Consumo de Energia	Access Number On-Chip Memory (ANOnM) Access Number Off-Chip Memory (ANOffM) Word Size (WS) Energy Consumption/Functionality (ECF)
	Área Ocupada	Occupied Slices (OS) Flip-Flops (FFs) Look-Up Tables (LUTs) Input/Output Blocks (IOBs)
	Segurança	Occupied Slices (OS) Input/Output Blocks (IOBs) Memory Size (MS) Risk Factor (RF) Fault Rate (FR)
	Desempenho	Critical Path Time (CPT) Maximum Frequency (MF)
Tempo-Real	Atraso	Worst Case Delay (WCD)
	Prazo	Worst Case Execution Time (WCET)

Fonte: Produção do próprio autor.

Na tabela 2 a coluna “RNF STRE” lista os Requisitos Não-Funcionais comuns para Sistemas de Tempo-Real Embarcados e a coluna “RNF FPGA” lista os requisitos Não-Funcionais identificados para projetos baseados em FPGA. A unidade de medida das métricas CPT, WCET e WCD

é em unidades de tempo (nanosegundos), para as métricas WS e MS a unidade de medida é em bytes. Para a métrica MF a unidade de medida é em MHz. Para o restante das métricas a unidade de medida é em quantidade.

Algumas destas métricas podem ser obtidas a partir de relatórios fornecidos pelas ferramentas de síntese, pois se referem à quantidade de recursos utilizados na FPGA para a implementação da lógica especificada. Essas métricas são: OS, IOBs, FFs e LUTs. Por exemplo, a ferramenta de síntese ISE Web Pack fornece o relatório de utilização dos recursos da FPGA, onde podem ser consultados estes valores. Na Figura 8 é apresentado um exemplo de relatório de utilização dos recursos da FPGA.

Figura 8 – Exemplo de relatório de utilização de recursos da FPGA.

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	104	18,224	1%
Number used as Flip Flops	103		
Number used as Latches	1		
Number used as Latch-thrus	0		
Number used as AND/OR logics	0		
Number of Slice LUTs	183	9,112	2%
Number used as logic	183	9,112	2%
Number using O6 output only	118		
Number using O5 output only	60		
Number using O5 and O6	5		
Number used as ROM	0		
Number used as Memory	0	2,176	0%
Number of occupied Slices	52	2,278	2%
Number of MUXCYs used	64	4,556	1%
Number of LUT Flip Flop pairs used	183		
Number with an unused Flip Flop	79	183	43%
Number with an unused LUT	0	183	0%
Number of fully used LUT-FF pairs	104	183	56%
Number of unique control sets	8		
Number of slice register sites lost to control set restrictions	32	18,224	1%
Number of bonded IOBs	46	232	19%
Number of RAMB16BWERs	0	32	0%

Fonte: Produção do próprio autor.

Informações sobre as métricas de tempo e roteamento do circuito também podem ser obtidas na ferramenta de síntese. A Figura 9 apresenta um exemplo de relatório com informações referente as restrições de tempo e roteamento do circuito. Esse relatório mostra, por exemplo, MF de $307,659$MHz para a lógica implementada e CPT de $3,250$ns.

Figura 9 – Exemplo de relatório de informações de tempo e roteamento do circuito.

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 3.250ns (Maximum Frequency: 307.659MHz)
   Minimum input arrival time before clock: 6.660ns
   Maximum output required time after clock: 3.634ns
   Maximum combinational path delay: No path found

Timing Details:
---------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clock'
  Clock period: 3.250ns (frequency: 307.659MHz)
  Total number of paths / destination ports: 1484 / 169
-------------------------------------------------------------------------
Delay:               3.250ns (Levels of Logic = 4)
  Source:            numberCS_31_C_31 (FF)
  Destination:       numberCS_31_C_31 (FF)
  Source Clock:      clock rising
  Destination Clock: clock rising

  Data Path: numberCS_31_C_31 to numberCS_31_C_31
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDCE:C->Q             3   0.447   0.755  numberCS_31_C_31 (numbe
     LUT3:I1->O            1   0.203   0.580  numberCS_311 (numberCS_
     LUT1:I0->O            0   0.205   0.000  Madd_numberCS[31]_GND_3
     XORCY:LI->O           2   0.136   0.617  Madd_numberCS[31]_GND_3
     LUT3:I2->O            1   0.205   0.000  numberCS_31_C_31_dpot (
     FDCE:D                    0.102          numberCS_31_C_31
    ----------------------------------------
    Total                      3.250ns (1.298ns logic, 1.952ns route)
                                       (39.9% logic, 60.1% route)
```

Fonte: Produção do próprio autor.

As métricas LOC, LOAC e LOWC são obtidas com a contagem do número de linhas de código. A métrica TR é a razão entre as métricas CDLOC e LOWC. A métrica CDLOC indica o número de vezes em que houve alteração de contexto entre o código original e o código do aspecto inserido. Assim, para calcular essa métrica, é necessário contar o número de vezes em que ocorreu a troca de contexto. Por exemplo, na Listagem 4.1, as linhas 3-7 são inseridas por um aspecto, assim ocorrem 2 trocas de contexto entre o aspecto e o código original, nas linhas 2 e 8.

Listagem 4.1 – Exemplo de Código

```
1 architecture Behavioral of controleSistema is
```

```
2  signal telaDigitodigito : INTEGER RANGE -252 TO +252:= 0;
3  signal watchdogSignal : BIT:= '0';
4  signal watchdogReset : BIT:= '0';
5  constant watchdogTimer : INTEGER RANGE -2147483647 TO
       +2147483647:= -1;
6  constant atualizaDigitoThreshold : INTEGER RANGE
       -2147483647 TO +2147483647:= 25000;
7  signal atualizaDigitoClockdiv : BIT:= '0';
8
9  component relogio IS
10  port (
11    clock : in STD_LOGIC;
12    reset : in STD_LOGIC;
13    ...
```

A métrica AB é obtida pela Equação 4.1:

$$AB = \frac{LOWC - LOC}{LOAC} \tag{4.1}$$

Fonte: (CARDOSO et al., 2012)

Por exemplo, assumindo os valores: LOWC = 15, LOC = 10 e LOAC 3, temos $AB = 15 - 10/3 = 1,66$.

As métricas ANOnM e ANOffM são a quantidade de acessos à memória interna e externa, feitas pelo circuito. A métrica WS indica o maior tamanho de dado a ser armazenado pelo circuito e a métrica MS indica o tamanho total de memória utilizada pelo circuito. A métrica ECF indica a razão da energia consumida pelo circuito em relação à quantidade de funcionalidades (requisitos funcionais) implementadas por este circuito. O consumo de energia pode ser obtido a partir de medições executadas na FPGA.

A análise da escalabilidade do projeto em relação ao seu desempenho e consumo de recursos da FPGA pode ser feita pela Equação 4.2.

$$\psi = \frac{MF_\beta \cdot \mathcal{C}_\alpha}{MF_\alpha \cdot \mathcal{C}_\beta} \tag{4.2}$$

Onde α representa a primeira versão do projeto e β representa a versão seguinte. $\mathcal{C}$ é a média dos recursos utilizados da FPGA (BELTRAN; GUZMáN; SEVILLANO, 2010), determinado segundo a Equação 4.3:

$$\mathcal{C} = \frac{1}{K} \cdot \sum_{j=1}^{K} \frac{r_j}{t_j} \tag{4.3}$$

Sendo K a quantidade de tipos de recursos que a FPGA possui, r o recurso que está sendo avaliado no somatório e t a quantidade total deste

recurso. Por exemplo, utilizando os valores apresentados na Figura 8 das métricas de OS, FFs, LUTs e IOBs tem-se $\mathcal{C} = 0,0616$. Assim, cria-se uma relação entre os recursos utilizados e a velocidade máxima para o circuito implementado, que permite a comparação entre diferentes implementações para um projeto, que fornecerá um índice de escalabilidade de uma versão para outra. A escalabilidade é calculada dividindo os valores da versão atual do projeto pelo versão anterior. Quanto mais próximo ou maior que 1 for ψ, mais escalável é o sistema pois aumenta de tamanho (que pode indicar mais funcionalidades implementadas), sem degradar o desempenho.

Esta seção apresentou possíveis métricas para avaliação de um projeto implementado em FPGA sobre diferentes perspectivas em relação aos requisitos não funcionais do projeto. O objetivo de discutir essas métricas é tornar o processo de avaliação e análise de projetos em FPGA mais pragmático, possibilitando comparações com outros trabalhos semelhantes e fornecendo um ferramenta para a determinação das capacidades do projeto desenvolvido e da metodologia utilizada para o seu desenvolvimento. Por fim, o conjunto de métricas proposto não é definitivo e não abrange todos os aspectos possíveis para um projeto implementado em FPGA, podendo ser aperfeiçoado e estendido.

4.3 GERAÇÃO AUTOMÁTICA DO CÓDIGO VHDL A PARTIR DE MODELOS UML

4.3.1 Regras de Mapeamento para Requisitos Funcionais

Para aperfeiçoar a transformação dos elementos da UML para o modelo DERCS, novas funções foram implementadas na ferramenta GenERTiCA. Foi adicionado tratamento para o estereótipo *TimedEvent* do perfil MARTE para a definição de métodos ativos. Esse estereótipo define que um elemento tem um comportamento periódico orientado ao tempo. Métodos marcados como *TimedEvent* são considerados métodos ativos no modelo DERCS, pois não são escalonados por outros processos, em vez disso, tem sua própria rotina de execução periódica. Outros tratamentos como, por exemplo, indicação de *read only* para atributos de uma classe, tipo de mensagem (síncrona ou assíncrona), entre outros, também foram adicionados à ferramenta GenERTiCA. Além disso, foram adicionadas funções na ferramenta GenERTiCA para avaliar se um determinado atributo é lido ou escrito em um comportamento de algum método (diagrama de sequência). Outra funcionalidade adicionada foi para retornar uma lista de métodos que relacionam classes. Esta lista são métodos de *get/set* de atributos criados a partir de associações, que possibilita conhecer a hierarquia das associações das classes em nível de regras de mapeamento.

O objetivo da metodologia AMoDE-RT é a construção de um modelo em alto-nível que permita a geração automática do código fonte na linguagem alvo, utilizando mecanismos de transformação de um modelo genérico para um modelo específico da plataforma alvo, a partir das regras de mapeamento. As regras propostas por Moreira (2012) infringem este princípio, pois o modelo em alto-nível precisa incluir detalhes específicos da linguagem VHDL, os quais não são usuais em outras plataformas.

Com as regras de mapeamento propostas por Moreira (2012) a modelagem do sistema é restrita à atribuição de valores aos sinais e processos executando isoladamente de forma concorrente, não possibilitando a utilização de troca de mensagens entre processos e objetos. Além disso, não foi identificado um tratamento para a execução de processos de forma síncrona ou assíncrona. Ao instanciar um processo em VHDL, esse passa a executar de forma concorrente com o processo principal e demais processos, sem respeitar o comportamento definido no diagrama de sequência.

No mapeamento realizado em Moreira (2012), a definição das portas das entidades ocorre pela definição de atributos públicos. Essa regra infringe o princípio de encapsulamento da orientação a objetos, onde um atributo de uma classe só deve ser acessado por ela mesma. Seguindo as boas práticas em programação orientada a objetos, para a manipulação dos valores de um atributo de uma classe, deve-se fornecer métodos *get/set* (obter/atribuir).

Dessa forma, é proposto um novo conjunto de regras de mapeamentos, contendo novas definições, que não são suportadas na implementação de Moreira. Na Tabela 3 são apresentados os principais conceitos alterados e acrescentados nas regras de mapeamento para VHDL.

Para facilitar o entendimento das regras de mapeamento para VHDL, serão utilizados alguns exemplos do estudo de caso Relógio. Este estudo de caso consiste em um relógio que apresenta os minutos e segundos em um display de sete segmentos de LED. A Figura 10 apresenta o diagrama de classes do projeto Relógio.

4.3.1.1 Atributos

Atributos com métodos *get/set* são definidos como portas e os demais são definidos como sinais, pois são usados apenas internamente pela classe proprietária. Para determinar a direção da porta, verifica-se um determinado atributo possui somente o método *get* ou *set*, ou possui ambos, determinando, respectivamente, portas OUT, IN ou INOUT. Além disso, caso um determinado atributo possua apenas um dos métodos mencionados e esse atributo é acessado em algum comportamento no diagrama de sequência, o atributo deve ser mapeado para uma porta com a direção adequada à operação executada. Por exemplo, se um atributo possui apenas o método *get*, deve

Tabela 3 – Conceitos mapeados.

Elemento UML		**Conceitos (MOREIRA, 2012)**	**Alterados/Incluídos**
Classes	Estrutura	Entidade-Arquitetura	Entidade-Arquitetura
	Associação	Componentes (1-para-1)	Componentes (1-para-n)
	Herança	-	Componentes para classes concretas e agregação de funcionalidades herdadas para as classes, atributos e métodos abstratos
Atributos	Públicos	Portas	-
	Privados	Sinais	Portas, sinais ou constantes
	Read-Only	-	Constantes
Tipos de Dados	Enumerados	-	Enumerados encapsulados em pacotes
	Compostos	-	Vetores
Métodos	Comportamento	Processos	Processos ou incorporação do código
	Troca de Mensagens	Portas da Entidade	Atribuição de sinais ou ativação de processos
	Estereótipo TimedEvent (MARTE)	-	Método ativo

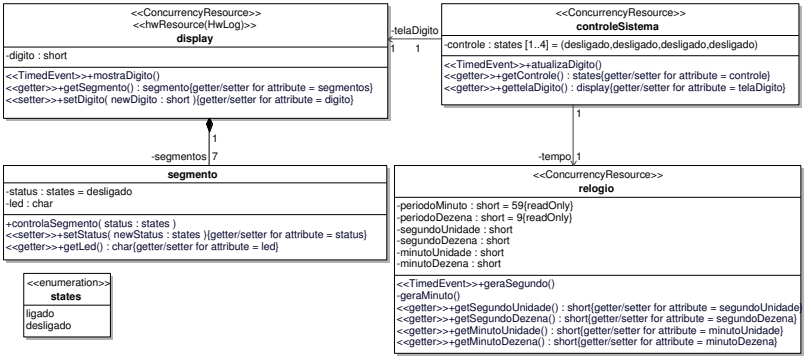
Fonte: Produção do próprio autor.

ser indicado como uma porta OUT, pois os dados serão transmitidos para fora do componente. Entretanto, caso o valor desse atributo seja lido em algum comportamento da classe, é necessário que esse atributo seja declarado como uma porta INOUT, pois na síntese do código VHDL não é permitido a leitura de valores em portas somente OUT.

A Figura 11 apresenta a classe *Segmento*. Essa classe possui dois atributos: *Status* e *Led*. De acordo com as regras definidas, esses atributos serão transformados em portas, pois possuem os métodos *get/set*.

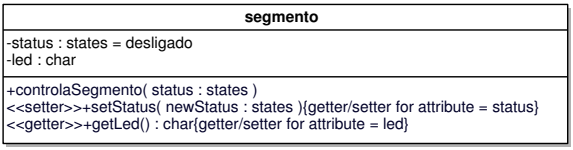
Para a definição de um método como *get/set* são utilizados os estereótipos ≪getter≫ e ≪setter≫ da UML, aplicados aos métodos. Esses estereótipos possuem a propriedade *attribute* que indica qual o atributo é acessado

Figura 10 – Diagrama de classes do projeto Display.



Fonte: Produção do próprio autor.

Figura 11 – Classe *Segmento*.



Fonte: Produção do próprio autor.

pelo método.

A Listagem 4.2 apresenta o código gerado para a classe *Segmento*. As portas *Clock* e *Reset* sempre são incluídas na entidade.

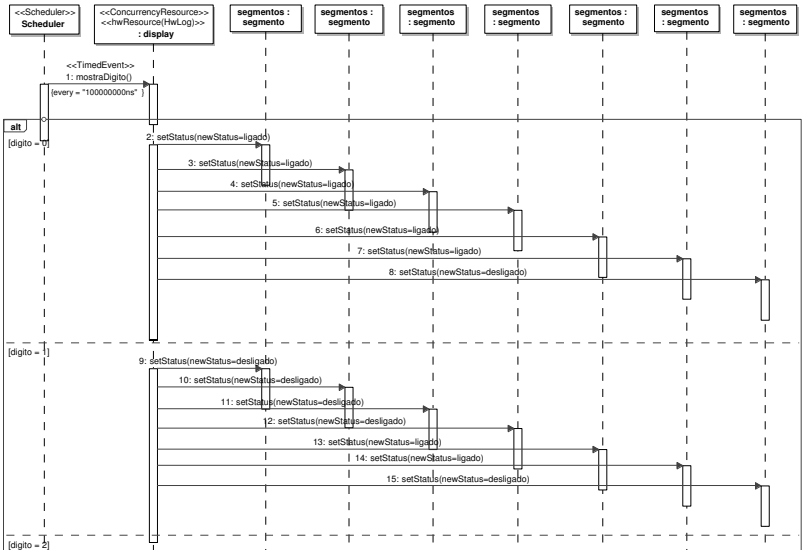
Listagem 4.2 – Descrição da Entidade Segmento.

```
1  entity segmento is
2         Port (
3         clock : in STD_LOGIC;
4         reset : in STD_LOGIC;
5         status : IN states;
6         led : OUT BIT );
7 end segmento;
```

Para que outra classe tenha acesso aos atributos da classe *Segmento*, deve-se utilizar os métodos *get/set* fornecidos. A chamada desses métodos é transformada em uma atribuição de valores aos sinais conectados às portas (atributos) na classe em que são utilizados. A Figura 12 apresenta parte do

diagrama de sequência do método *mostraDigito* da classe *Display* que utiliza o método *setStatus* da classe *Segmento*, seguindo o encapsulamento dos atributos de acordo com as boas práticas de um projeto orientado a objetos.

Figura 12 – Extrato do diagrama de sequência do método *mostraDigito* da classe *Display*.



Fonte: Produção do próprio autor.

Na Listagem 4.3 é apresentado o extrato do processo *mostraDigito* gerado a partir do diagrama apresentado na Figura 12. A entidade *Segmento* é incorporada à entidade *Display* como um componente. São definidos sinais na entidade *Display* que são conectados as portas do componente *Segmento* como apresentado nas linhas 7 e 13. Quando ocorre a chamada do método *setStatus*, as regras convertem para a atribuição de valor ao sinal local, como na linha 45, responsável por transmitir a informação para o componente *Segmento*, mantendo assim, o encapsulamento das informações.

Listagem 4.3 – Extrato da descrição da entidade *Display* e do processo *mostraDigito*.

```
1 ...
2 entity display is
3  Port (
4   clock : in STD_LOGIC;
```

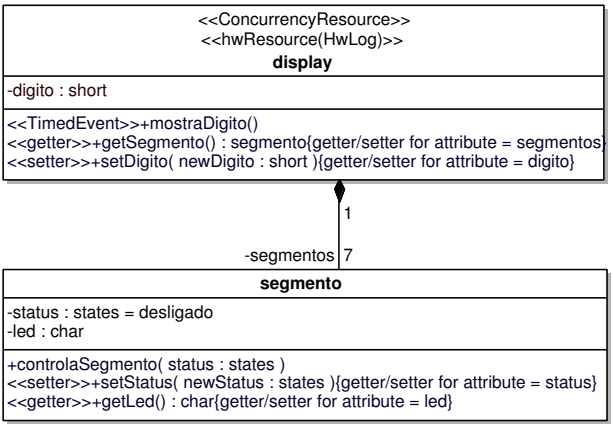
```
5   reset : in STD_LOGIC;
6   digito : IN INTEGER RANGE -252 TO +252 ;
7   segmentosled_0 : OUT BIT;
8   segmentosled_1 : OUT BIT;
9   ...
10  segmentosled_6 : OUT BIT);
11 end display;
12 architecture Behavioral of display is
13   signal segmentosstatus_0 : states;
14   signal segmentosstatus_1 : states := desligado;
15   signal segmentosstatus_2 : states := desligado;
16   ...
17   signal segmentosstatus_6 : states := desligado;
18   component segmento IS
19     port (
20           clock : in STD_LOGIC;
21           reset : in STD_LOGIC;
22           status : IN states;
23           led : OUT BIT );
24     end component;
25 begin
26   segmentos_0: segmento port map(
27      clock => clock,
28      reset => reset,
29      status  =>  segmentosstatus_0,
30      led  =>  segmentosled_0  );
31    segmentos_1: segmento port map(
32      clock => clock,
33      reset => reset,
34      status  =>  segmentosstatus_1,
35      led  =>  segmentosled_1  );
36    ...
37    segmentos_6: segmento port map(
38      clock => clock,
39      reset => reset,
40      status  =>  segmentosstatus_6,
41      led  =>  segmentosled_6  );
42    mostraDigito: process( clock, reset, digito )
43    begin
44     if ( digito = 0 ) then
45      segmentosstatus_0 <= ligado;
46      segmentosstatus_1 <= ligado;
47      segmentosstatus_2 <= ligado;
48      segmentosstatus_3 <= ligado;
49      segmentosstatus_4 <= ligado;
50      segmentosstatus_5 <= ligado;
51      segmentosstatus_6 <= desligado;
52    ...
```

4.3.1.2 Associações

A associação 1-para-1 define a instância de um componente e seu mapeamento na classe que realiza esta associação. Este trabalho incluiu o tratamento para associações 1-para-n. Ambas as associações resultam em um mesmo código, sendo o mapeamento de um componente para a classe associada e sua instanciação. A diferença entre as associações é o número de vezes que o componente instanciado é mapeado.

O componente é instanciado uma única vez na classe que realiza a associação e *n* mapeamentos desse componente, conforme definido na cardinalidade do relacionamento. Na Figura 13 é apresentado um exemplo de associação de agregação entre as classe *Segmento* e *Display*, indicando que cada display possui sete segmentos.

Figura 13 – Associação das classes *Segmento* e *Display*.



Fonte: Produção do próprio autor.

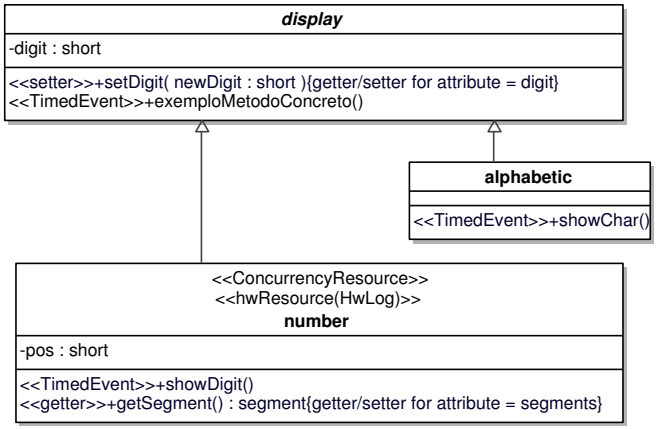
A Listagem 4.3 apresenta o extrato da descrição gerada a partir desse relacionamento. O componente *Segmento* é instanciado um única vez, linhas 18-24, e *n* mapeamentos para o componente nas linhas 26-41. Como a classe *Display* possui o método *getSegmento*, os atributos que possuem métodos *get* ou que possuem atributos escritos na classe *Segmento*, também são incluídos como portas na classe *Display*, garantindo o princípio de encapsulamento da orientação a objetos. Os atributos que possuem apenas método *set* ou não são escritos no componente instanciado, não são declarados como portas, sendo declarados como sinais. Esses atributos serão apenas manipulados pela entidade que instancia o componente (neste caso, o *Display*), encapsulando esse

comportamento.

4.3.1.3 Herança

A associação de generalização define que uma classe herda o comportamento e estrutura da classe associada, especializando-a. Essa associação é conhecida como herança, onde uma classe é chamada de superclasse e a classe que herda desta, é chamada de subclasse (DEITEL; DEITEL, 2010). Aqui o projeto relógio foi alterado para incluir um exemplo de herança. A Figura 14 apresenta o exemplo de generalização criado no projeto Relógio. Neste, as classes *number* e *alphabetic* herdam da classe *display*, que é uma classe abstrata.

Figura 14 – Associação de generalização da classe *Display*.

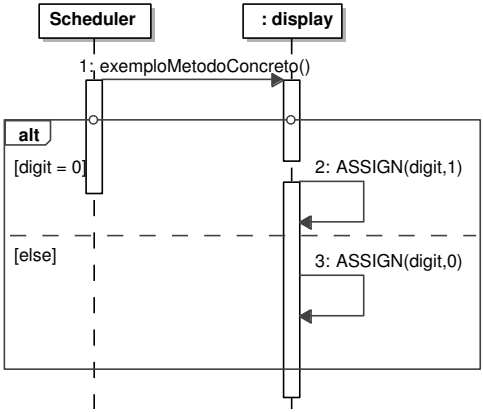


Fonte: Produção do próprio autor.

Agora o *display* pode ser especializado para imprimir letras e números. A Figura 15 apresentada o diagrama de sequência para o método *exemploMetodoConcreto*. Esse método foi adicionado ao modelo para ilustrar a herança de um comportamento.

Não é gerado o código VHDL para superclasses abstratas, sendo todos os seus métodos e atributos incorporados nas subclasses no mapeamento para VHDL. Métodos abstratos das superclasses são implementados na subclasse (i.e. a especificação do comportamento desses métodos é feita nas subclasses). Assim não é necessário nenhuma implementação extra para o mapeamento de métodos abstratos, sendo estes métodos mapeados diretamente nas subclasses que os implementam.

Figura 15 – Diagrama de sequência para o método *exemploMetodoConcreto*.



Fonte: Produção do próprio autor.

Contudo, métodos concretos implementados em superclasses abstratas tem seu comportamento herdado pelas subclasses. Assim, o comportamento dos métodos concretos das superclasses são mapeados como processos nas subclasses (seguindo as mesmas regras apresentadas para métodos ativos, assíncronos e síncronos). A Listagem 4.4 apresenta o código gerado para a classe *number* com a superclasse *display* abstrata. O atributo da classe *display* foi herdado pela classe *number*, sendo incluído como uma porta nesta entidade na linha 8. O comportamento apresentado na Figura 15 foi herdado pela classe *number*. Assim esse comportamento foi incorporado a essa entidade nas linhas 31-38.

Listagem 4.4 – Extrato do código gerado para a classe *number* com a superclasse abstrata.

```
1 entity number is
2  Port (
3    clock : in STD_LOGIC;
4    reset : in STD_LOGIC;
5    segmentsled_0 : OUT BIT;
6    segmentsled_1 : OUT BIT;
7    ...
8    digit : INOUT INTEGER RANGE -252 TO +252);
9 end number;
10
11 architecture Behavioral of number is
12  signal segmentsstatus_0 : states:= SOFF;
13  signal segmentsstatus_1 : states:= SOFF;
```

```
14  ...
15  signal pos : INTEGER RANGE -252 TO +252:= 0;
16  ...
17 begin
18    ...
19  showDigit: process  (clock, reset)
20  begin
21   if (reset = '1') then
22    -- variables initialization
23   elsif (clock'EVENT and clock='1') then
24    if ( digit = 0 ) then
25     segmentsstatus_0 <= SON;
26     segmentsstatus_1 <= SON;
27   ...
28   end if;
29  end process showDigit;
30
31  exemploMetodoConcreto: process (digit )
32  begin
33   if ( digit = 0 ) then
34    digit <= 1;
35   else
36    digit <= 0;
37   end if;
38  end process exemploMetodoConcreto;
39 end Behavioral;
```

Quando ocorre a especialização de uma classe concreta, o comportamento herdado concreto, é executado pela superclasse, exceto no caso de sobrecarga dos métodos. Assim, uma superclasse concreta é mapeada para VHDL como um componente na subclasse. Considerando a classe *display* como concreta, o código da Listagem 4.5 é gerado, contendo os atributos e métodos concretos desta classe.

Listagem 4.5 – Código gerado para a classe *Display*.

```
1  entity display is
2   Port (
3    clock : in STD_LOGIC;
4    reset : in STD_LOGIC;
5    digit : INOUT INTEGER RANGE -252 TO +252);
6  end display;
7
8  architecture Behavioral of display is
9  begin
10  exemploMetodoConcreto: process (digit )
11  begin
12   if ( digit = 0 ) then
13    digit <= 1;
14   else
15    digit <= 0;
```

```
16   end if;
17  end process exemploMetodoConcreto;
18 end Behavioral;
```

A Listagem 4.6 apresenta o código gerado para a classe *number* com a superclasse concreta. A classe *display* é mapeada como um componente nas linhas 17-22 e 26-29. Assim, métodos concretos são executados pelo componente que representa a superclasse concreta.

Listagem 4.6 – Extrato de código gerado para a classe *number* com a superclasse concreta.

```
 1 entity number is
 2  Port (
 3   clock : in STD_LOGIC;
 4   reset : in STD_LOGIC;
 5   segmentsled_0 : OUT BIT;
 6   segmentsled_1 : OUT BIT;
 7   ...
 8   displaydigit : INOUT INTEGER RANGE -252 TO +252);
 9 end number;
10
11 architecture Behavioral of number is
12  signal segmentsstatus_0 : states:= SOFF;
13  signal segmentsstatus_1 : states:= SOFF;
14  ...
15  signal pos : INTEGER RANGE -252 TO +252:= 0;
16  ...
17  component display IS
18   port (
19    clock : in STD_LOGIC;
20    reset : in STD_LOGIC;
21    digit : INOUT INTEGER RANGE -252 TO +252     );
22   end component;
23  ...
24 begin
25  ...
26  :display port map(
27   clock => clock,
28   reset => reset,
29   digit  =>  displaydigit );
30   ...
```

O DERCS não possui suporte à herança múltipla pois não é uma boa prática em projetos orientados a objetos. Assim não foram implementadas regras de mapeamento para a herança múltipla.

A linguagem VHDL possui suporte para sobrecarga de funções e operações. No entanto, não exite suporte para sobrecarga de processos. Nesse caso, quando ocorre a sobrecarga de um método concreto de uma superclasse, não é incluído o método da superclasse nas subclasses. O mapeamento para

VHDL inclui somente o processo referente aos métodos implementados nas subclasses.

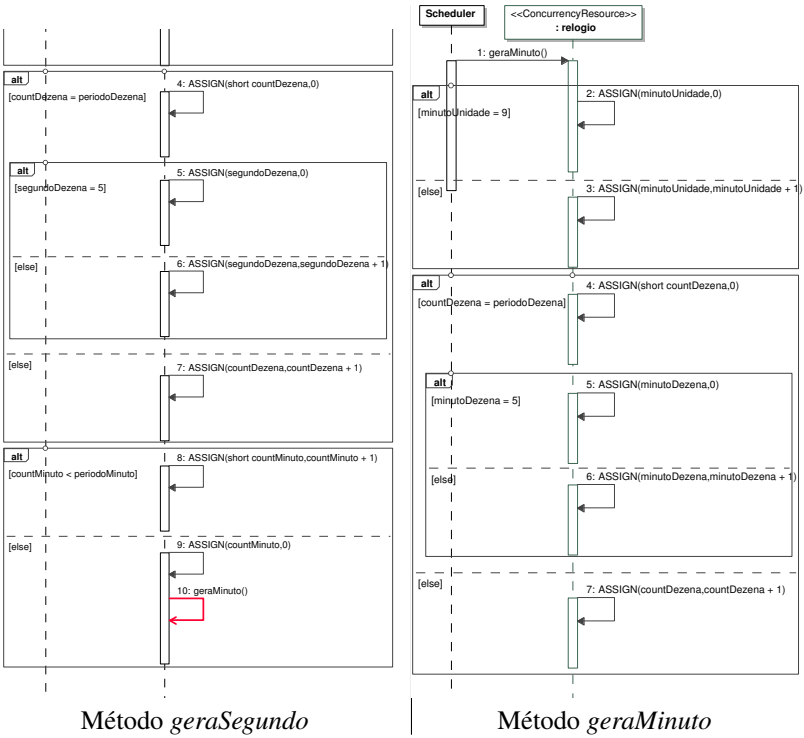
4.3.1.4 Métodos Assíncronos

Um método é convertido em um processo se for marcado como *TimedEvent* ou quando possui pelo menos uma chamada assíncrona em algum diagrama de sequência. Métodos que possuem somente chamadas síncronas não são convertidos em processos, sendo seu código incorporado de forma *inline* ao processo que os chamou.

Métodos assíncronos são os métodos que não são *TimedEvent* e são chamados de forma assíncrona. Os parâmetros do método são definidos na lista de variáveis sensitivas do processo. Caso o método não possua nenhum parâmetro, um sinal de controle de execução do processo, denominado *enable*, é criado. O processo será ativado quando ocorrer alteração do valor de algum dos sinais da lista sensitiva. Essa alteração ocorre no momento da chamada do método. A Figura 16 apresenta o comportamento dos métodos *geraSegundo* e *geraMinuto* da classe *Segmento*. O método *geraSegundo* é marcado como *TimedEvent*, assim será gerado um processo para esse método. No comportamento do método *geraSegundo* apresentado na Figura 16 há uma chamada síncrona para o método *geraMinuto*. Nesse caso o sistema gera um processo para o método *geraMinuto* com o comportamento apresentado na Figura 16, pois o processo *geraSegundo* pode continuar mesmo que o processo *geraMinuto* não tenha finalizado.

A primeira mensagem dos diagramas de sequência, é utilizada pela ferramenta GenERTiCA para definir de qual método pertence o comportamento definido no diagrama. Assim, a chamada do método *geraMinuto* no diagrama *geraMinuto* não é considerado como uma chamada de método. Nesse caso, não há distinção para chamadas síncronas e assíncronas.

Figura 16 – Extrato do diagrama de sequência do método *geraSegundo* e diagrama do método *geraMinuto* da classe *Segmento*.



Fonte: Produção do próprio autor.

A Listagem 4.7 apresenta a descrição dos processos *geraSegundo* e *geraMinuto*. O parâmetro *geraMinutoEnable* foi incluído na lista de variáveis sensitivas do processo *geraMinuto*, conforme apresentado na linha 29. Sempre que for alterado o valor desse sinal, como na linha 25, esse processo será executado.

Listagem 4.7 – Descrição dos processos *geraSegundo* e *geraMinuto*.

```
1 geraSegundo: process( segundoUnidade , segundoDezena )
2  variable countDezena : INTEGER RANGE -252 TO +252 := 0;
3  variable countMinuto : INTEGER RANGE -252 TO +252 := 0;
4 begin
5  if ( segundoUnidade = 9 ) then
6   segundoUnidade <= 0;
```

```
 7  else
 8   segundoUnidade <= segundoUnidade + 1;
 9  end if;
10  if ( countDezena = periodoDezena ) then
11   countDezena := 0;
12   if ( segundoDezena = 5 ) then
13    segundoDezena <= 0;
14   else
15    segundoDezena <= segundoDezena + 1;
16   end if;
17  else
18   countDezena := countDezena + 1;
19  end if;
20  if ( countMinuto < periodoMinuto ) then
21   countMinuto := countMinuto + 1;
22  else
23   countMinuto := 0;
24   -- Ativacao do processo geraMinuto
25   geraMinutoEnable <= '1';
26  end if;
27 end process geraSegundo;
28
29 geraMinuto: process( geraMinutoEnable, reset )
30  variable countDezena : INTEGER RANGE -252 TO +252 := 0;
31 begin
32  if (reset='1') then
33   -- signals initialization
34  elsif (geraMinutoEnable'EVENT and geraMinutoEnable =
        '1') then
35   if ( minutoUnidade = 9 ) then
36    minutoUnidade <= 0;
37   else
38    minutoUnidade <= minutoUnidade + 1;
39   end if;
40   if ( countDezena = periodoDezena ) then
41    countDezena := 0;
42    if ( minutoDezena = 5 ) then
43     minutoDezena <= 0;
44    else
45     minutoDezena <= minutoDezena + 1;
46    end if;
47   else
48    countDezena := countDezena + 1;
49   end if;
50  end if;
51 end process geraMinuto;
```

4.3.1.5 Métodos Síncronos

Quando um método chama outro método de forma assíncrona, não necessita que este finalize sua execução para continuar executando como, por

exemplo, o método *geraSegundo* não precisa aguardar o termino da execução do método *geraMinuto* para continuar executando após chamá-lo. Isso ocorre de fato com todos os processos em VHDL, pois executam de forma concorrente. O que é um problema para os processos que precisam executar de forma síncrona, quando um método precisa aguardar a finalização de outro para continuar executando. Em VHDL os processos não aguardam a finalização de outro processo utilizando elementos sintetizáveis. Uma opção é a utilização de estruturas como *procedures* e *functions* da linguagem VHDL.

As estruturas *procedures* e *functions* da linguagem VHDL funcionam de forma semelhante às linguagens de alto-nível. A finalidade destas estruturas em VHDL é o reaproveitamento e padronização de código, geralmente definidas em bibliotecas.

Algumas ferramentas de síntese incorporam o código das *procedures* e *functions* dentro do processo que as chamam, de modo a permitir que a descrição seja sintetizada (RAMACHANDRAN et al., 1993). Uma limitação na utilização de *procedures* e *functions* é impedir o acesso de forma direta aos atributos da entidade. Assim, a troca de informações é feita por meio de portas de entrada e saída, o que descaracteriza sua utilização como representação para os métodos da classe. Outras limitações apresentadas na Seção 2.4 também inviabilizam a sua utilização no lugar de processos. Por essas razões essas estruturas não foram utilizadas neste trabalho.

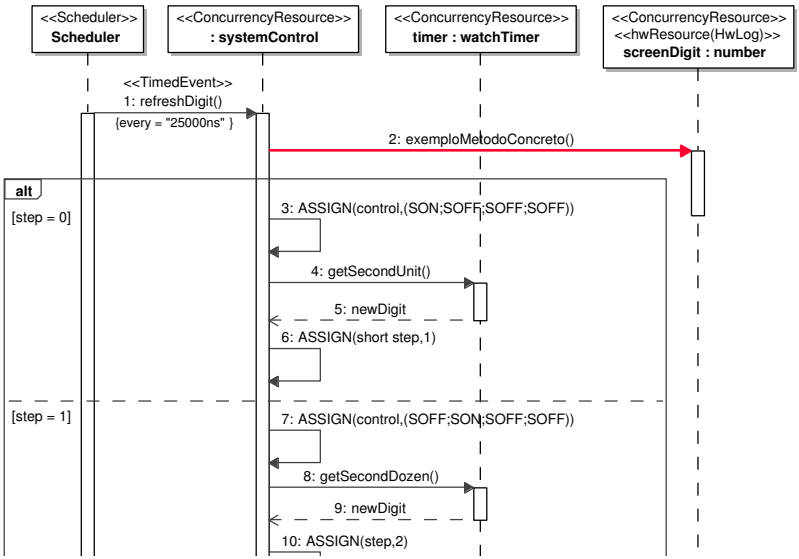
Assim, no código VHDL gerado para chamadas síncronas, o comportamento dos métodos chamados é incorporado ao método que os chama exatamente no ponto em que esses métodos são chamados, i.e. *inline*. A Figura 17 apresenta um extrato do diagrama de sequência do método *refreshDigit* da classe *systemControl*. O método *exemploMetodoConcreto* da classe *display* é chamado de forma síncrona, i.e. ele precisa terminar de executar para que o método *refreshDigit* possa continuar seu processamento.

A Listagem 4.8 apresenta a descrição do processo *refreshDigit* gerado a partir do diagrama de sequência da Figura 17. As linhas 8-14 apresentam a descrição do comportamento do método *exemploMetodoConcreto* incorporado ao método que o chamou. Essa estratégia garante a execução síncrona do comportamento do processo.

Listagem 4.8 – Extrato da descrição do processo *refreshDigit*.

```
1 refreshDigit: process  (clock, reset)
2  variable newDigit : INTEGER RANGE -252 TO +252 := 0;
3  variable step : INTEGER RANGE -252 TO +252 := 0;
4 begin
5  if (reset = '1') then
6   -- variables initialization
7  elsif (clock'EVENT and clock='1') then
8   -- Synchronous process exemploMetodoConcreto from display
```

Figura 17 – Diagrama de sequência para o método *refreshDigit* da classe *systemControl*.



Fonte: Produção do próprio autor.

```
        component
 9    if ( digit = 0 ) then
10     digit <= 1;
11    else
12     digit <= 0;
13    end if;
14    -- End of synchronous process
15    if ( step = 0 ) then
16     control <= (SON;SOFF;SOFF;SOFF);
17     newDigit := timersecondUnit ;
18     step := 1;
19    elsif ( step = 1 ) then
20     control <= (SOFF;SON;SOFF;SOFF);
21     newDigit := timersecondDozen ;
22     ...
```

4.3.1.6 Métodos anotados com o estereótipo TimedEvent

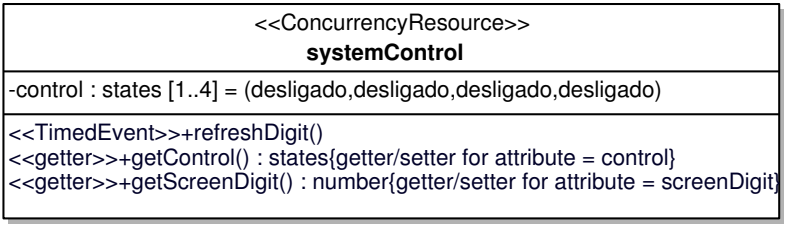
Métodos marcados como *TimedEvent* são considerados métodos ativos, que executam de forma independente a outros processos e são sensíveis

ao clock. Esses métodos são convertidos em processos. Os parâmetros dos métodos são incluídos na lista de variáveis sensitivas do processo.

Moreira (2012) mapeou o processo nomeado como "run" como o processo principal de um objeto ativo, ou seja, o processo responsável pelo controle de execução dos demais métodos do objeto. Esse processo acaba assumindo o comportamento de um escalonador local dos processos da entidade, sendo sensível ao clock (cronômetro) do sistema.

O perfil MARTE da UML possui o estereótipo *TimedEvent* que define eventos desse tipo, ou seja, que são controlados por um clock e executam de forma concorrente. A transformação UML-DERCS foi alterada para incluir a leitura desse estereótipo e determinar se um método é ativo, i.e. quando possui a definição de *TimedEvent*. Na Figura 18 a classe *systemControl* possui o método *refreshDigit* anotado com o estereótipo *TimedEvent*, indicando que este é um método ativo. Na Listagem 4.8 as linhas 5-7 apresentam o resultado do tratamento para o método ativo.

Figura 18 – Classe *systemControl*.



Fonte: Produção do próprio autor.

Assim, as regras de mapeamento para VHDL foram alteradas para verificar se um método é ativo e incluir o tratamento gerado antes somente para os processos chamados "run". Com isso é possível a existência de mais de um processo ativo em uma mesma classe, e eliminando a necessidade do tratamento "*hardcoded*" para o processo "run".

4.3.1.7 Discussão

A descrição de hardware em VHDL segue uma estrutura hierárquica, sendo o nível mais alto da hierarquia (do inglês *top-level*) o componente que se comunicará com o ambiente no qual está inserido. Esse componente tem como responsabilidade repassar as informações para os componentes que estão abaixo de sua hierarquia e exteriorizar as informações geradas por

eles. Essa característica se assemelha à estrutura lógica da orientação a objetos, permitindo que seja possível sua relação com elementos da linguagem VHDL. Essa condição permite criar uma analogia entre os métodos *get/set* e a informação que deve ser repassada para um nível superior da hierarquia.

Outra característica semelhante é a sensibilidade a eventos na orientação a objetos e a sensibilidade aos sinais e suas variações em VHDL. Essa analogia permite a definição de processos síncronos, assíncronos e tratamento de interrupções. Essas semelhanças existem porque o paradigma orientado a objetos e a descrição de hardware em VHDL abordam um mesmo problema: a representação de mundo real por elementos que conseguem atender de forma paralela e em tempo-real aos estímulos externos. Enquanto o paradigma orientado a objetos tenta imitar o mundo real, a linguagem VHDL permite de fato simular esse mundo, uma vez que sua aplicação na plataforma FPGA torna possível a execução de tarefas de forma concorrente como ocorre no mundo real.

Por fim, nas regras apresentadas nesta seção, determinou-se que ao ocorrer uma chamada de método *set*, o mapeamento para VHDL assume essa chamada como a atribuição de um valor no sinal que representa o método. Entretanto isso pode gerar um problema na síntese do código VHDL, especificamente quando uma atribuição de um sinal ocorre em mais de um processo ou em um nível superior. Não é possível sintetizar esse circuito pois não se pode determinar o valor do sinal em um determinado momento. Para solucionar esse problema é possível a implementação de um multiplexador com um seletor. Este seletor indicaria qual processo está conectado ao sinal em dado momento. Entretanto essa solução não foi abordada neste trabalho, sendo indicado como um trabalho futuro.

4.3.2 Regras de Mapeamento para RNF

De acordo com a metodologia AMoDE-RT, os requisitos não funcionais do projeto são especificados como aspectos no modelo UML no diagrama ACOD, seguindo o paradigma orientado a aspectos apresentado na Seção 2.6. Cada aspecto implementa um requisito não funcional. Os aspectos são representados como ***Classes***, anotados com o estereótipo *Aspect*. As adaptações executadas pelo aspecto e os *pointcuts* são representados como ***métodos***.

Adaptações estruturais são marcadas com o estereótipo *StructuralAdaptation* e adaptações comportamentais com *BehavioralAdaptation*. Os detalhes de como a adaptação será implementada são especificados no arquivo de regras de mapeamento, permitindo que um mesmo aspecto seja portável para diferentes plataformas e/ou aplicações.

As adaptações podem ocorrer de duas formas: no nível de modelo,

onde serão introduzidos ou modificados elementos no modelo DERCS, gerado a partir da especificação UML. Estas modificações estarão visíveis para as regras de mapeamento para a transformação do modelo DERCS para a linguagem alvo; ou em nível de código, onde as modificações afetam diretamente o código gerado na linguagem alvo, sem alteração no modelo DERCS. A definição da forma de adaptação é especificada nas regras de mapeamento do aspecto.

Para cada atributo que é adicionado nas classes afetadas pelo aspecto, é incluído um relacionamento *crosscut* (WEHRMEISTER, 2009). Esse relacionamento não define uma associação, em vez disso ele especifica os valores utilizados para inicialização dos atributos criados, i.e. as restrições impostas pelos requisitos não funcionais como, por exemplo, validade dos dados, *deadline*, etc. Dessa forma, o aspecto garante o cumprimento da restrição definida no modelo.

Como mencionando anteriormente, os *pointcuts* são métodos marcados com o estereótipo *pointcut*. Esses métodos possuem três parâmetros: o nome do JPDD que define o *join point*, o nome da adaptação e o tipo de adaptação. Os tipos de adaptações podem ser: BEFORE, AFTER, AROUND, REPLACE, MODIFY_STRUCTURE ou ADD_NEW_FEATURE. Detalhes são apresentados em Wehrmeister (2009).

Os *Join Points* são definidos em diagramas de sequência ou classe, marcados com o estereótipo *JPDD*. Conforme apresentado na Seção 2.6, os *join points* selecionam os elementos que serão afetados pelas adaptações e o local onde essa adaptação ocorrerá. Enquanto que os *pointcuts* ligam as adaptações aos *join poits*. A Figura 19 apresenta a definição do aspecto *PeriodicTiming*. Para ilustrar essa abordagem de especificação, cita-se a adaptação *Period*, que é uma adaptação estrutural, i.e. essa adaptação modifica a estrutura dos elementos afetados. O *pointcut pcAddPeriod* liga a adaptação *Period* ao *join point JPDD_PERIOD* e indica que essa adaptação irá incluir um novo atributo na classe afetada.

Para a utilização dos conceitos de orientação a aspectos em VHDL, foram analisadas implementações de tratamentos para requisitos não funcionais nesta linguagem para a plataforma FPGA. Além disso, foram identificados possíveis locais para adaptações na estrutura da linguagem VHDL. Esse estudo, possibilitou entender e definir como os aspectos podem interferir no código VHDL gerado. Esta seção apresenta os *pointcuts* identificados para a linguagem VHDL e os aspectos implementados que tratam os requisitos não funcionais.

4.3.2.1 Pointcuts

Cada linguagem de programação possui uma estrutura de código diferente. Assim, cada linguagem alvo é construída de forma diferente a partir das regras de mapeamento. As regras de mapeamento possuem um conjunto de estruturas para cada elemento do modelo UML. O código gerado é construído com base na junção dessas estruturas de acordo com a necessidade de cada linguagem. Por isso, cada linguagem tem seu conjunto de regras de mapeamento com estruturas e "junções" diferentes, o que leva à necessidade de definir pontos de adaptações específicos para a linguagem alvo, quando a sua estrutura possui exceções como, por exemplo, a definição par entidade/arquitetura para VHDL. A definição desses pontos de adaptações diferentes para a linguagem VHDL, não implica em especificar *join points* específicos para VHDL. Os *Join Points* filtram os locais onde podem ocorrer adaptações no código final. Os *Pointcuts* indicam a ação que será afetada pela adaptação e ligam os *join points* as adaptações, criando uma definição precisa de onde ocorrerá uma adaptação nos elementos do projeto. Esses pontos são as "junções" que possibilitam a inserção das adaptações. Os *join points* definidos para VHDL podem ser reutilizados para outras plataformas. No entanto, os *pointcuts* são específicos para VHDL, devido as regras de mapeamento criadas para esta linguagem.

Identificaram-se como *pointcuts* para a linguagem VHDL os pontos listados na Tabela 4. A coluna "Elemento UML" é a representação do elemento selecionado pelo *join point* em um nível abstrato. A coluna "Elemento VHDL" é o elemento afetado na estrutura VHDL. A coluna "Tipo" indica se a adaptação será estrutural ou comportamental. A coluna "Adaptação" indica o que o aspecto efetuará no elemento selecionado. A coluna "Nível" indica se a adaptação ocorrerá em um nível de modelo, i.e. uma modificação no modelo DERCS, que refletirá na geração do código, ou uma adaptação no código final gerado.

Tabela 4 – Pontos de adaptação para VHDL.

Elemento UML		**Elemento VHDL**	**Tipo**	**Adaptação**	**Nível**
Classes	Atributo	Entidade	Estrutural	Inclusão de Portas	Modelo
	Implementação	Entidade	Estrutural	Inclusão de Componentes	Código
	Atributo	Arquitetura	Estrutural	Inclusão de sinais	Modelo
	Implementação	Arquitetura	Estrutural	Mapeamentos de Componentes	Código
	Implementação	Arquitetura	Estrutural	Inclusão de processos	Código
Métodos	Comportamento dos Métodos	Processos	Comportamental	Adaptações (BEFORE e AFTER)	Código
	Comportamento dos Métodos	Processos	Comportamental	Inclusão de variáveis locais	Modelo
	Métodos	Processos	Comportamental	Inclusão de sinais na lista sensitiva	Modelo
Comportamento	Ações	Atribuição de sinais	Comportamental	Adaptações (BEFORE e AFTER)	Código

Fonte: Produção do próprio autor.

Os elementos da Tabela 4 definem locais em que uma adaptação pode ocorrer na linguagem VHDL. Por exemplo, deseja-se adicionar uma nova porta em uma entidade. Para isso, é necessário inserir um novo atributo na classe afetada. A Tabela 4 indica que essa adaptação é possível a nível de modelo, pois existe um ponto nas regras de mapeamento que permitem essa inserção que é no processamento dos atributos de uma classe (entidade). Já a inclusão de um novo processo (método) em uma entidade somente é possível em nível de código, e o ponto onde essa modificação ocorre é no processamento das regras de implementação da classe (entidade). Exemplos de *pointcuts* que utilizam essas regras são apresentados na Seção 4.3.2.2.

4.3.2.2 Aspectos DERAF

Com o estudo da revisão da literatura realizado neste trabalho, foi possível identificar um conjunto de requisitos não funcionais que tem grande impacto em sistemas embarcados desenvolvidos em FPGAs (ver Seção 4.1). No contexto do presente trabalho, foram implementados requisitos não funcionais relacionados ao domínio de aplicação dos estudos de caso realizados, tendo como base o conjunto de requisitos apresentados na Seção 4.1. Tal afirmação não implica que os requisitos não funcionais implementados não possam ser utilizados em outros projetos, mas indica que o conjunto de requisitos apresentado é limitado ao escopo dos estudos de caso elaborados. Os requisitos não funcionais identificados foram modelados como aspectos em um nível abstrato, sendo possível sua utilização em outras plataformas além da FPGA.

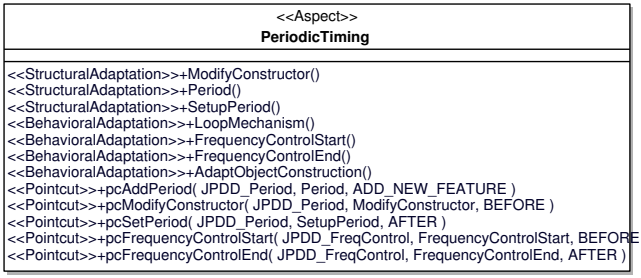
Para a especificação da transformação dos aspectos para VHDL foi utilizado o DERAF (WEHRMEISTER, 2009). Para alguns requisitos foi necessário estender os aspectos definidos no DERAF incluindo novas adaptações. Esses aspectos são o *PeriodicTiming* e o *DataFreshness*. Além disso, novos *join points* foram definidos neste trabalho para estes aspectos, adicionando o tratamento para os estereótipos *ConcurrencyResource*, *hwResource* e *ResourceUsage* do perfil MARTE. Além disso, adicionou-se o pacote *FaultHandling* no DERAF de modo a fornecer algum suporte para o tratamento de falhas. Desta forma, este trabalho propõe o aspecto *COPMonitoring*.

4.3.2.2.1 PeriodicTiming

Este aspecto fornece mecanismos para disparar periodicamente a execução do comportamento de um objeto ativo (WEHRMEISTER, 2009). A Figura 19 apresenta a definição do aspecto *PeriodicTiming*. Para controlar a frequência de execução do processo, o aspecto *PeriodicTiming* adiciona

atributos e componentes as entidades afetadas, pelas adaptações *Period*, *ModifyConstructor*, *SetupPeriod* e *FrequencyControl*.

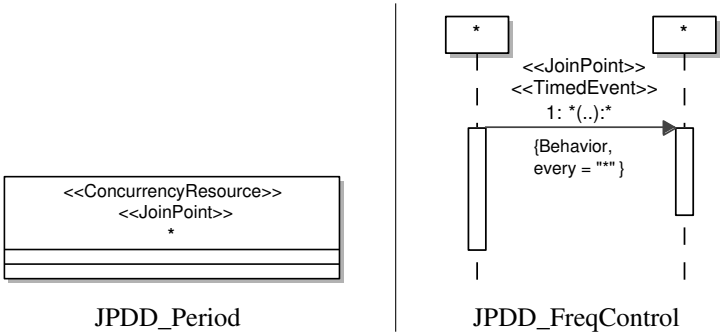
Figura 19 – Definição do Aspecto *PeriodicTiming*.



Fonte: Produção do próprio autor.

A Figura 20 apresenta os diagramas JPDD que definem os *join points* para o aspecto *PeriodicTiming* definidos neste trabalho. Na linguagem VHDL, objetos ativos são todas as entidades que possuem um comportamento que executa de forma concorrente e assíncrona, ou seja, ele é independente de outros processos. Assim, classes marcadas como *ConcurrencyResource* são entendidas como objetos ativos por este aspecto. Essas classes terão um método ativo que é marcado como *TimedEvent*, i.e. que possuem características temporais, que por sua vez, são controladas pelo clock do sistema. Em outras palavras, todos as operações marcados como *TimedEvent* são processos que executam de forma independente e concorrente em VHDL.

Figura 20 – Diagramas JPDD para o aspecto *PeriodicTiming*.



Fonte: Produção do próprio autor.

Para controlar a frequência de execução do processo *TimedEvent*, utiliza-se um componente divisor de clock[5], pois a frequência que se deseja executar o processo pode ser diferente da frequência do clock do sistema. O divisor de clock é inserido como um componente da plataforma FPGA pelas regras de mapeamento das configurações da plataforma, quando o aspecto *PeriodicTiming* é utilizado no modelo. Segundo Wehrmeister (2009), a plataforma alvo deve fornecer recursos para a implementação dos aspectos relacionados aos requisitos não funcionais. Esses recursos podem ser criados ou alterados pelas regras de mapeamento de configuração da plataforma. O fato de o divisor de clock ser um componente independente torna o aspecto portável para outras plataformas, onde cada plataforma terá seu componente responsável por dividir o clock. A Listagem 4.9 apresenta o *script* que gera o divisor de clock para a plataforma FPGA.

Listagem 4.9 – Script de geração do componente divisor de clock.

```
1 <File Name="ClockDivider.vhd" OutputDirectory="" Aspects="
      PeriodicTiming">
2  <Fragment>
3    -- Basic libraries
4    library IEEE;
5    use IEEE.STD_LOGIC_1164.ALL;
6    use IEEE.STD_LOGIC_ARITH.ALL;
7    use IEEE.STD_LOGIC_UNSIGNED.ALL;
8
9    entity clockDiv is
```

[5] Componente utilizado para controlar a frequência de ativação de um processo quando tal frequência é inferior a do hardware.

```
10      Port (
11        clock : in STD_LOGIC;
12        threshold: in integer range -2147483647 TO
              +2147483647;
13        clockdiv : out BIT);
14      end clockDiv;
15
16    architecture Behavioral of clockDiv is
17    begin
18      div:process (clock, threshold)
19        variable count: integer range -2147483647 TO
              +2147483647 := 0;
20      begin
21        if (clockEVENT and clock='1') then
22          if (count &lt; threshold) then
23            count := count + 1;
24            clockdiv &lt;= '0';
25          else
26            count := 0;
27            clockdiv &lt;= '1';
28          end if;
29        end if;
30      end process div;
31    end Behavioral;
32  </Fragment>
33 </File>
```

O aspecto *PeriodicTiming* instancia esse componente na entidade afetada e efetua o mapeamento de suas portas, considerando o processo *TimedEvent* e sua restrição de período de execução.

A Listagem 4.10 apresenta as regras de mapeamento definidas para as adaptações do aspecto *PeriodicTiming*. As adaptações do aspecto *PeriodicTiming* executam as seguintes atividades:

- **Period**: Cria uma variável que define a periodicidade de execução do processo nas linhas 1-10. O valor dessa variável é definido no relacionamento *crosscut* na *tag Period*;
- **ModifyConstructor**: Instancia o componente divisor de clock nas linhas 11-17;
- **SetupPeriod**: Insere um trecho de código que efetua o mapeamento do componente divisor de clock nas linhas 18-28;
- **FrequencyControlStart**: Introduz um trecho de código para controlar a periodicidade do processo pelo uso do divisor de clock. O processo *TimedEvent* passa a ter em sua lista de variáveis sensitivas o novo clock dividido e executa a cada evento desse sinal.

Listagem 4.10 – Regras de mapeamento das adaptações do aspecto *PeriodicTiming*.

```
1  <Structural Name="Period" Order="0" ModelLevel="yes">
2    #foreach ($Message in $Class.getMethods())
3      #if ($Message.getActiveMode())
4        #set ($attr = $Class.addAttribute("${Message.Name}
             Threshold",$DERCSFactory.newLong(true),$
             DERCSFactory.getPrivate(),false,$DERCSHelper.
             strTimeToInteger( $Crosscutting.getValueOf("
             Period"), "ns" ),true))
5        #set ($attr = $Class.addAttribute("${Message.Name}
             Clockdiv",$DERCSFactory.newChar(),$DERCSFactory.
             getPrivate(),false,"",false))
6        $Message.addParameter("${Message.Name}Clockdiv", $
             DERCSFactory.newChar(), $DERCSFactory.
             getParameterIn())
7        $Message.addParameter("reset", $DERCSFactory.newChar
             (), $DERCSFactory.getParameterIn())
8      #end
9    #end
10 </Structural>
11 <Structural Name="ModifyConstructor" Order="0" ModelLevel="
       no">
12   \n component clockDiv IS
13   \n port ( clock : in STD_LOGIC;
14   \n  threshold: in integer range -2147483647 TO
         +2147483647;
15   \n  clockdiv : out BIT);
16   \n end component;
17 </Structural>
18 <Structural Name="SetupPeriod" Order="2" ModelLevel="no">
19   #foreach ($Message in $Class.getMethods())
20     #if ($Message.getActiveMode())
21       \n ${Message.Name}Divider: clockDiv port map(
22       \n  clock =&gt; clock,
23       \n  threshold =&gt; ${Message.Name}Threshold,
24       \n  clockdiv =&gt; ${Message.Name}Clockdiv
25       \n );
26     #end
27   #end
28 </Structural>
29 <Behavioral Name="FrequencyControlStart" Order="1"
       ModelLevel="no">
30   \n if (reset = '1') then
31   \n  -- variables initialization
32   \n elsif (${Message.Name}ClockdivEVENT and ${Message.Name
         }Clockdiv='1') then
33 </Behavioral>
34 <Behavioral Name="FrequencyControlEnd" Order="2" ModelLevel
       ="no">
35   \n end if;
36 </Behavioral>
```

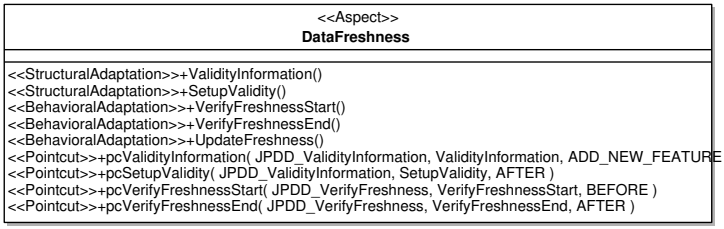
Na linguagem VHDL não precisa-se criar um *loop* para controlar

a periodicidade de execução, pois o hardware executa cada processo a cada ciclo de clock. Assim as adaptações *LoopMechanism* e *AdaptObjectConstruction* não são utilizadas.

4.3.2.2.2 DataFreshness

Este aspecto trata da validade temporal dos dados do sistema. O tempo de validade dos dados (*freshness*) é um fator importante para sistemas de tempo real (FARINES; FRAGA; OLIVEIRA, 2000). A Figura 21 apresenta a definição do aspecto *DataFreshness*.

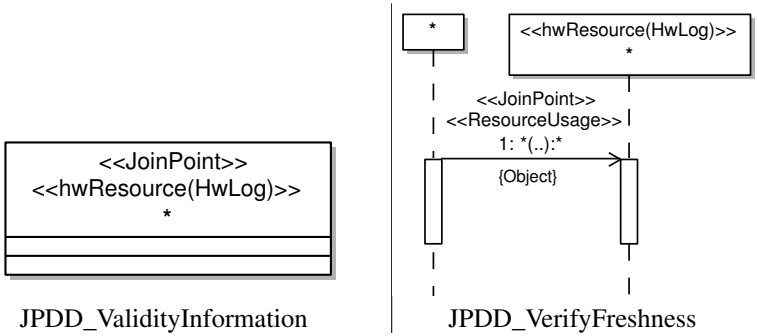
Figura 21 – Definição do aspecto *DataFreshness*.



Fonte: Produção do próprio autor.

Dados de informações provenientes de outros componentes podem alterar constantemente, principalmente aqueles fornecidos por sensores. Componentes que fornecem informações para o sistema e que os dados fornecidos possuem um prazo de validade podem ser definidos como *HWResource*. Assim, classes marcadas com este estereótipo são afetadas pelo aspecto *DataFreshness* pois o prazo de validade de seus atributos precisa ser controlado. A Figura 22 apresenta os *join points* definidos neste trabalho para o aspecto *DataFreshness*. O *join point JPDD_ValidityInformation* seleciona as classes que são marcadas com o estereótipo *HWResource*. O *join point JPDD_VerifyFreshness* seleciona os métodos que lêem informações das classes *HWResource*, que são marcados como *ResourceUsage*.

Figura 22 – Diagramas JPDD para o aspecto *DataFreshness*.



Fonte: Produção do próprio autor.

A Listagem 4.11 apresenta as regras de mapeamento das adaptações do aspecto *DataFreshness*. As adaptações do aspecto *DataFreshness* executam as seguintes atividades:

- **ValidityInformation**: Insere variáveis nas classes selecionadas para controlar o prazo de validade de cada atributo da classe. Na linha 2 é criado a variável *validityDeadline* para indicar o prazo de validade dos atributos da classe selecionada, sendo único para todos os atributos desta classe. O valor da variável *validityDeadline* é definido no relacionamento *crosscut* com a classe selecionada, na *tag Validity*. Nas linhas 3-8 são criadas as variáveis de controle para cada atributo afetado da classe selecionada. É criado a variável *validity+Nome_atributo_afetado* na linha 6, para indicar se o valor dos atributos da classe selecionada é válido. Essa variável deve ser apresentada como uma porta na entidade afetada em VHDL, permitindo que entidades que usam esse componente consultem a validade de seus dados. Assim, um método *get* é criado para essa variável na linha 7;

- **SetupValidity**: Insere um processo na classe selecionada para controlar o tempo de validade de seus atributos nas linhas 11-33. Esse processo verifica a cada ciclo de clock se houve uma alteração no valor dos atributos da classe selecionada. Se houve alteração, o processo reinicia os contadores e a variável de validade dos atributos nas linhas 19-21. Caso contrário, continua contando até alcançar o limite *validityDeadline* para então marcar o atributo como não válido na linha 27;

- **VerifyFreshness**: Classes que acessam as informações das classes do tipo *HWResource* precisam verificar se o dado lido é ainda válido. A

adaptação *VerifyFreshness* insere uma validação no método que lê o valor dos atributos de classes *HWResource*, i.e. os métodos marcados como *ResourceUsage*. Na linha 37 é verificado se a variável de controle da validade do atributo lido é válido para então ler o seu valor.

Listagem 4.11 – Regras de mapeamento das adaptações do aspecto *DataFreshness*.

```
1  <Structural Name="ValidityInformation" Order="0" ModelLevel
       ="yes">
2    #set ($attr = $Class.addAttribute("validityDeadLine",$
         DERCSFactory.newLong(true),$DERCSFactory.getPrivate()
         ,false,$DERCSHelper.strTimeToInteger( $Crosscutting.
         getValueOf("Validity"), "ns" ),true))
3   ...
4   #set ($attrAOPName = "validity${attrAOPpar.Name}")
5   #set ($attrAOPpar = $Class.addAttribute("old${attrAOPpar.
        Name}",$attrAOPpar.getDataType(),$attrAOPpar.
        getVisibility(),false,$attrAOPpar.getDefaultValue(),$
        attrAOPpar.getReadOnly()));
6   #set ($attrval = $Class.addAttribute($attrAOPName,$
        DERCSFactory.newChar(),$DERCSFactory.getPrivate(),
        false,"'0'",false));
7   #set ($mthAop = $DERCSHelper.addGetMethod($Class,$attrval)
        );
8   ...
9  </Structural>
10 <Structural Name="SetupValidity" Order="0" ModelLevel="no">
11   \n setupValidity:process (clock)
12   ...
13   \n
14   \n begin
15   \n   if (clockEVENT and clock='1') then
16        ...
17   #if ($chk and !$attrAOP.Name.startsWith("validity"))
18     \n if (old${attrAOP.Name} /= $attrAOP.Name) then
19     \n   validity${attrAOP.Name} &lt;= '0';
20     \n   countValidity${attrAOP.Name} := 0;
21     \n   old${attrAOP.Name} &lt;=  $attrAOP.Name;
22     \n else
23     \n   if (countValidity${attrAOP.Name} &lt;
           validityDeadLine) then
24     \n     countValidity${attrAOP.Name} := countValidity${
           attrAOP.Name} + 1;
25     \n   else
26     \n     countValidity${attrAOP.Name} := 0;
27     \n     validity${attrAOP.Name} &lt;= '1';
28     \n   end if;
29     \n end if;
30   ...
31   #end
```

```
32   \n end if;
33   \n end process setupValidity;
34 </Structural>
35 <Behavioral Name="VerifyFreshnessStart" Order="0"
       ModelLevel="no">
36   \n
37   \nif (${Action.getAction().getToObject().getName()}
         validity${Action.getAction().RelatedMethod.
         getAssociatedAttribute().Name} = '0') then
38 </Behavioral>
39 <Behavioral Name="VerifyFreshnessEnd" Order="0" ModelLevel=
      "no">
40   \nend if;
41   \n
42 </Behavioral>
```

4.3.2.2.3 COPMonitoring

Conforme observado na análise da literatura, o tratamento e tolerância a falhas para projetos baseados em FPGA é muito importante, assim como em outras plataformas de sistemas embarcados. Esse trabalho propõe a adição do pacote *FaultHandling* para os aspectos que tratam de falhas de hardware em sistemas embarcados.

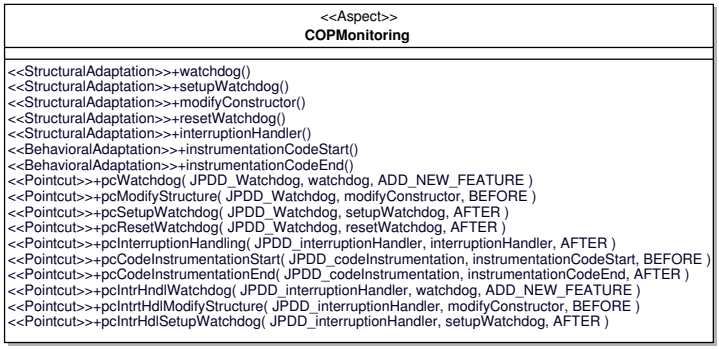
O aspecto *Computer Operating Properly Monitoring* (COPMonitoring) foi adicionado ao DERAF para incorporar o tratamento de falhas no pacote *FaultHandling*. O objetivo desse aspecto é instrumentar o código para o tratamento de interrupções por alguma falha. A Figura 23 apresenta a especificação desse aspecto.

O aspecto *COPMonitoring* afeta todos os objetos que são marcados como *hwResource*[6] do perfil MARTE. As entidades marcadas com esse estereótipo são recursos de hardware como, por exemplo, componentes da FPGA, nos quais podem ocorrer falhas. Classes marcadas como *hwResource* devem ser instrumentadas para o tratamento do watchdog, que indicará se ocorreu alguma interrupção nesse recurso. A Figura 24 apresenta o *join point JPDD_Watchdog*, que seleciona as classes marcadas como *hwResource*.

Objetos que utilizam recursos marcados como *hwResource* devem ser instrumentados para verificar se houve alguma interrupção neste recursos e, caso ocorre alguma interrupção, executar uma rotina de tratamento. Esses objetos são avisados pelo *watchdog* quando ocorrer uma interrupção. A Figura 25 apresenta os *join points* que selecionam comportamentos de objetos que utilizam métodos de recursos *hwResource*. O *join point*

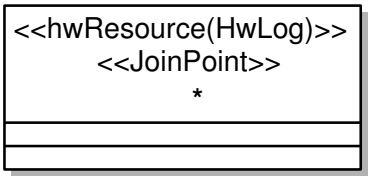
[6] O estereótipo utilizado é do pacote *HwLogical*.

Figura 23 – Definição do aspecto *COPMonitoring*.



Fonte: Produção do próprio autor.

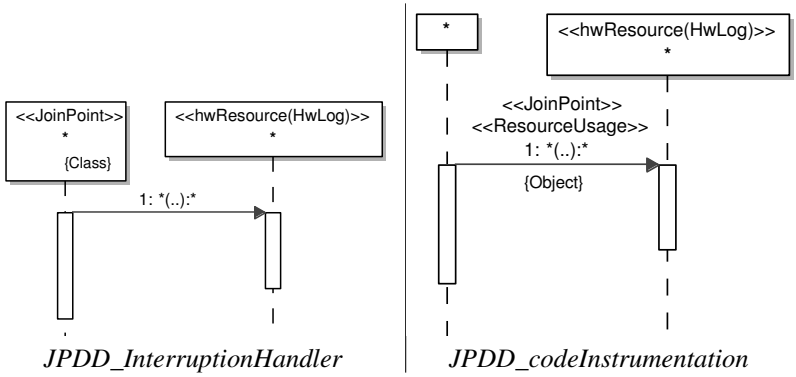
Figura 24 – Diagrama JPDD *JPDD_Watchdog* do aspecto *COPMonitoring*.



Fonte: Produção do próprio autor.

JPDD_InterruptionHandler seleciona classes que acessam recursos marcados como *hwResource* para a inserção de elementos que permitam verificar quando houve uma interrupção no recurso acessado. O *join point JPDD_codeInstrumentation* seleciona os métodos que acessam recursos marcados como *hwResource* que devem ser instrumentados para o caso de uma interrupção.

Figura 25 – Diagramas JPDD para o aspecto *COPMonitoring*.



Fonte: Produção do próprio autor.

A proposta de implementação do aspecto *COPMonitoring* em VHDL utiliza o componente *Watchdog*. O componente *watchdog* é gerado pelas regras de configuração da plataforma para VHDL, assim como ocorre para o divisor de clock. Ele fornece um sinal que deve ser reinicializado pela aplicação dentro de um período determinado. Caso não seja reiniciado, o *watchdog* ativa o sinal de *resetWatchdog*, indicando que algum problema ocorreu com o recurso, e, consequentemente, uma interrupção é disparada. Os objetos *hwResource* devem reinicializar o *watchdog* para indicar que estão operando normalmente. Caso o *watchdog* não seja reiniciado, ele dispara uma interrupção e avisa os objetos que dependem do recurso interrompido, que executam rotinas de tratamento a interrupção. A Listagem 4.12 apresenta o *script* de geração do componente *Watchdog*.

Listagem 4.12 – Script de geração do componente WatchDog.

```
1  <File Name="WatchDog.vhd" OutputDirectory="" Aspects="
       COPMonitoring">
2   <Fragment>
3    -- Basic libraries
4    library IEEE;
5    use IEEE.STD_LOGIC_1164.ALL;
6    use IEEE.STD_LOGIC_ARITH.ALL;
7    use IEEE.STD_LOGIC_UNSIGNED.ALL;
8
9    entity watchdog is
10    Port (
11      clock : in STD_LOGIC;
12      watchdogSignal: in BIT;
```

```
13      watchdogTimer: in INTEGER RANGE -2147483647 TO
            +2147483647;
14      watchdogReset: out BIT
15    );
16   end watchdog;
17
18   architecture Behavioral of watchdog is
19   begin
20     main:process (clock)
21       variable count: INTEGER RANGE -2147483647 TO
             +2147483647 := 0;
22     begin
23       if (clockEVENT and clock='1') then
24         if (count &lt;= watchdogTimer) then
25           count := count + 1;
26         else
27           count := 0;
28           if (watchdogSignal = '1') then
29             watchdogReset &lt;= '1';
30           else
31             watchdogReset &lt;= '0';
32           end if;
33         end if;
34       end if;
35     end process main;
36   end Behavioral;
37  </Fragment>
38 </File>
```

A Listagem 4.13 apresenta as regras de mapeamento das adaptações do aspecto *COPMonitoring*. As adaptações do aspecto *COPMonitoring* executam as seguintes atividades:

- **Watchdog**: Cria variáveis de controle para comunicação com o componente *watchdog*. Na linha 2 da Listagem 4.13 é inserido a variável *watchdogSignal* que deve ser reiniciada para não ocorrer uma interrupção. A variável *watchdogReset*, criada na linha 3, indica que o *wachdog* não foi reiniciado. Na linha 4 é criado a variável *watchdogTimer* que indica o tempo limite para reiniciar o *watchdog*. O tempo de reinicialização do *watchdog* é informado no relacionamento *crosscut* na *tag Deadline*;
- **ResetWatchdog**: Insere um processo para reinicializar o sinal *watchdog*, conforme mostrado nas linhas 28-39;
- **ModifyConstructor**: Instancia o componente *watchdog*;
- **SetupWatchdog**: Insere um trecho de código para mapear as portas do componente *watchdog* para os sinais locais;
- **InterruptionHandler**: Cria o processo *handlingInterruption* que efe-

tuará o tratamento da interrupção quando o sinal do *watchdog* não for reiniciado; Esse processo é sensível ao sinal *resetWatchdog*, assim, quando este sinal for ativado, o processo de tratamento da interrupção irá executar. O processo *handlingInterruption* reinicializa o objeto no qual ocorre a interrupção pelo sinal *reset*;

- **InstrumentationCode**: Instrumentam os processos para prepara-los para uma interrupção causada pelo *watchdog*.

Listagem 4.13 – Regras de mapeamento das adaptações do aspecto *COPMonitoring*.

```
1 <Structural Name="watchdog" Order="0" ModelLevel="yes">
2  #set ($attr = $Class.addAttribute("watchdogSignal",$
       DERCSFactory.newChar(),$DERCSFactory.getPrivate(),
       false,"",false))
3  #set ($attr = $Class.addAttribute("watchdogReset",$
       DERCSFactory.newChar(),$DERCSFactory.getPrivate(),
       false,"",false))
4  #set ($attr = $Class.addAttribute("watchdogTimer",$
       DERCSFactory.newLong(true),$DERCSFactory.getPrivate(),
       false,$DERCSHelper.strTimeToInteger( $Crosscutting.
       getValueOf("Deadline"), "ns" ),true))
5  #foreach ($Message in $Class.getMethods())
6    #if (!$Message.isGetSetMethod() and $Message.Name != $
       Class.Name)
7      $Message.addParameter("watchdogReset", $DERCSFactory.
         newChar(), $DERCSFactory.getParameterIn())
8    #end
9  #end
10 </Structural>
11 <Structural Name="modifyConstructor" Order="1" ModelLevel="
     no">
12  \n     component watchdog IS
13  \n  port (     clock : in STD_LOGIC;
14  \n    watchdogSignal: in BIT;
15  \n    watchdogTimer: in INTEGER RANGE -2147483647 TO
       +2147483647;
16  \n    watchdogReset: out BIT);
17  \n end component;
18 </Structural>
19 <Structural Name="setupWatchdog" Order="3" ModelLevel="no">
20 \n ${Class.Name}Watchdog: watchdog port map(
21 \n   clock =&gt; clock,
22 \n   watchdogSignal =&gt; watchdogSignal,
23 \n   watchdogTimer =&gt; watchdogTimer,
24 \n   watchdogReset =&gt; watchdogReset
25 \n );
26 </Structural>
27 <Structural Name="resetWatchdog" Order="4" ModelLevel="no">
28  \n resetWatchdog:process (clock)
```

```
29  \n  variable count: integer := 0;
30  \n begin
31  \n  if (clockEVENT and clock='1') then
32  \n   if (count &lt; watchdogTimer) then
33  \n    count := count + 1;
34  \n   else
35  \n    count := 0;
36  \n    watchdogSignal &lt;= '0';
37  \n   end if;
38  \n  end if;
39  \n end process resetWatchdog;
40 </Structural>
41 <Structural Name="interruptionHandler" Order="5" ModelLevel
       ="no">
42  \n handlingInterruption:process (watchdogReset)
43  \n begin
44  \n  if (watchdogResetEVENT and watchdogReset='1') then
45  \n   -- reset the state of signals controled by
        architecture
46  \n   reset &lt;= '1';
47  \n  end if;
48  \n end process handlingInterruption;
49 </Structural>
50 <Behavioral Name="instrumentationCodeStart" Order="0"
       ModelLevel="no">
51  \nif (watchdogReset = '0') then
52 </Behavioral>
53 <Behavioral Name="instrumentationCodeEnd" Order="6"
       ModelLevel="no">
54  \nend if;
55 </Behavioral>
```

4.3.2.3 Discussão

Com a implementação dos aspectos *PeriodicTiming* e *DataFreshness* para a linguagem VHDL, foram definidos novos *join points* utilizando estereótipos do perfil MARTE. Componentes para suporte as necessidades especificas da plataforma como o divisor de clock e o *watchdog*, foram utilizados na implementação destes aspectos. Assim, essas modificações permitem que os aspectos sejam portáveis para outras plataformas, habilitando o seu uso tanto na linguagem VHDL, como também em outras linguagens de alto nível como C++ e Java.

A adição do pacote *FaultHandling* com a implementação do aspecto *COPMonitoring* adiciona ao DERAF suporte a uma importante área de requisitos não funcionais para sistemas embarcados de tempo-real. As regras de mapeamento criadas para as adaptações dos aspectos implementados neste trabalho, permitiram adaptar o código para o tratamento dos requisitos não

funcionais identificados, sem a necessidade de manutenções manuais.

Entretanto, alguns problemas foram identificados na implementação dos aspectos. Foi identificado o conflito de aspectos, que ocorre quando um aspecto adiciona uma funcionalidade a uma entidade afetada, que é mal interpretada por outro aspecto, que por sua vez, modifica indevidamente esta nova funcionalidade. Outro problema identificado foi em relação a *join points* semelhantes entre diferentes aspectos, que levam a aplicação de um aspecto indevidamente em determinado elemento. Porém, ambos os problemas identificados foram tratados por validações extras nas regras de mapeamento. Ressalta-se que os mesmos problemas podem ocorrer em outras linguagens de desenvolvimento, não estando limitados à linguagem VHDL.

A definição dos *pointcuts* apresentados na Tabela 4 cobre boa parte das estruturas da linguagem VHDL necessárias para a descrição de um circuito complexo. O Capítulo 5 apresenta estudos de caso que validam estas definições. Outros *pointcuts* não abordados neste trabalho, podem ser inclusos, permitindo aperfeiçoar a metodologia. Tais *pointcuts* são apresentados na Tabela 5.

Tabela 5 – Pontos de adaptação para VHDL não abordados.

Elemento UML	**Elemento VHDL**	**Tipo**	**Ação**	**Nível**
Enumerados	Definição de tipos de dados	Estrutural	Inclusão de tipos	Modelo
Ações - Expressões	Expressões	Comportamental	Adaptações (BEFORE e AFTER)	Código
Pacotes	Bibliotecas	Estrutural	Inclusão de bibliotecas	Modelo

Fonte: Produção do próprio autor.

5 VALIDAÇÃO EXPERIMENTAL

5.1 INTRODUÇÃO

Este Capítulo apresenta os estudos de caso realizados para validação da abordagem proposta. Essa validação inclui o uso das regras de mapeamento construídas para a geração do código VHDL, assim como dos aspectos implementados. Foram desenvolvidos três estudos de caso: controle de robô seguidor de linha, controle automático de válvula e um relógio de minutos e segundos.

A especificação dos estudos de caso seguiu a metodologia AMoDE-RT. A abordagem AMoDE-RT prevê o levantamento e especificação dos requisitos funcionais pela ferramenta RT-FRIDA (FREITAS, 2007). Porém, esta ferramenta não foi utilizada neste trabalho. As demais etapas da abordagem foram seguidas completamente.

Em cada estudo de caso, um modelo UML/MARTE foi criado seguindo as regras de modelagem da metodologia AMoDE-RT. Em seguida, o modelo criado foi usado para a geração automática do código VHDL. A modelagem dos estudos de caso foi feita na ferramenta MagicDraw e o código VHDL foi gerado com a ferramenta GenERTiCA. Para comparação e análise foram geradas duas versões de código para cada estudo de caso, sendo uma sem a definição dos aspectos que tratam os requisitos não funcionais e outra com os aspectos. Entretanto, ambas as versões do código possuíam os mesmo requisitos funcionais e não funcionais implementados, sendo que na versão sem o uso dos aspectos, os requisitos não funcionais foram implementados manualmente. O código gerado foi sintetizado e validado por protótipos e/ou simulação.

A análise dos resultados dos estudos de caso foi realizada com base nas métricas levantadas a partir da revisão da literatura e descritas na Seção 4.2. Para analisar a abordagem proposta, foram utilizadas as métricas relacionadas a área ocupada, reusabilidade, desempenho e escalabilidade.

Para a realização dos estudos de caso foi utilizada a placa de prototipação Nexys 3 com a FPGA Spartan-6, modelo XC6LX16-CS324. As especificações dos recursos básicos desta FPGA são apresentadas na Tabela 6. Cada Bloco de Lógica Configurável (BLC) possui dois elementos denominados *Slice*, que constitui uma unidade básica da FPGA, que são constituídas por 4 LUTs e 8 FFs cada.

Foi utilizada a ferramenta ISE WebPack da Xilinx para a síntese do código VHDL gerado. Esta versão é gratuita e permite a realização do projeto e síntese de códigos em VHDL e Verilog, além de possuir ferramentas de simulação. Os dados extraídos sobre o consumo de recursos da FPGA foram

Tabela 6 – Especificação Spartan6 XC6LX16.

Recurso	Quantidade
Blocos de Lógica Configuráveis (BLC)	1.139
Slices	2.278
Look-Up Table (LUT)	9.112
Flip-Flops (FF)	18.224
Blocos de Entrada/Saída (IOB)	232

Fonte: Produção do próprio autor.

retirados de relatórios fornecidos pela ferramenta ISE WebPack.

Para a análise da área foi medida a quantidade de *Slices*, LUTs, FFs e IOBs. Considerando que, cada *slice* contém 4 LUTs e 8 FFs, durante a fase de localização e roteamento na síntese, alguns ou todos componentes da *slice* serão ocupados, dependendo da lógica implementada para o circuito (XILINX, 2014). Por exemplo, uma lógica pode exigir a utilização de FFs mais não de LUTs. Como nesse caso as LUTs não são utilizadas, a ferramenta de síntese indica que foi utilizada uma *slice* e também o número de FFs. Pode-se adicionar mais lógica nesse circuito sem aumentar o número de *slices* ocupadas, caso seja possível a utilização das LUTs não alocadas na *slice* que foi usada parcialmente. Quanto mais recursos de uma *slice* são utilizados, mais otimizado é a localização e roteamento, o que depende da organização da lógica implementada e dos algoritmos implementados na ferramenta de síntese.

Para análise do desempenho foram utilizadas as métricas maior caminho crítico e máxima frequência. O maior caminho crítico indica o WCET para o circuito. Para a análise de escalabilidade foram utilizadas as equações apresentadas na Seção 4.2. Para a análise de reusabilidade dos aspectos implementados, foram utilizadas as métricas número de linhas de código, número de linhas de adaptação, taxa de entrelaçamento e o número de linhas de código entrelaçado. Os valores das métricas são apresentadas com até duas casas decimais.

O restante deste Capítulo apresenta cada um dos estudos de caso desenvolvidos, bem como discute e analisa os resultados alcançados.

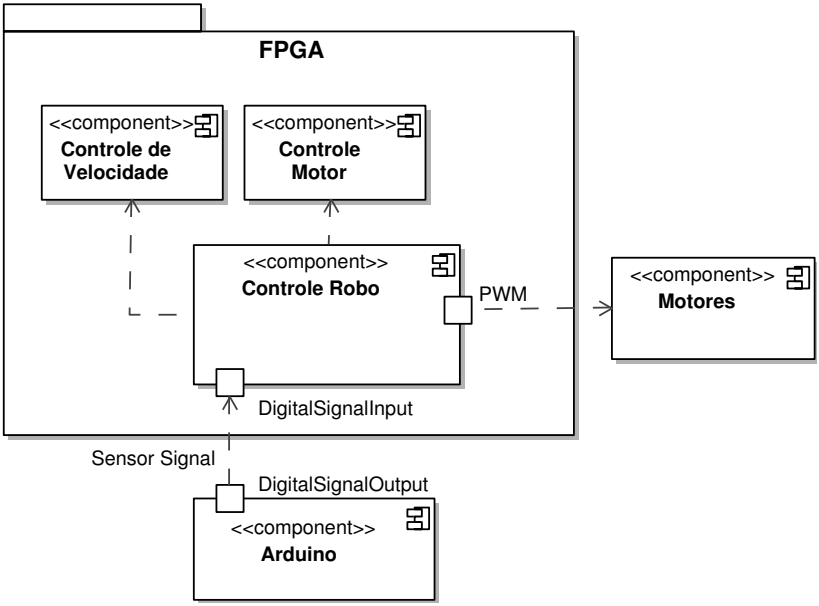
5.2 CONTROLE DE ROBÔ AUTÔNOMO

Este estudo de caso consiste em um robô autônomo que segue uma linha. O controle do robô foi implementado como um ASIP na FPGA. O robô consiste em: (i) dois sensores infravermelho; um para o lado esquerdo e outro

para o lado direito; (ii) duas rodas; (iii) dois servo-motores.

Como a FPGA utilizada não possui entradas analógicas para a leitura dos sensores, uma placa Arduino Uno foi usada para esta tarefa. A placa Arduino codifica os valores analógicos dos sensores e envia um sinal digital para a FPGA. O controle de movimento e de direção é implementado na FPGA, que, ao processar os dados, envia um sinal para os motores, indicando direção e velocidade. Um sinal PWM (*Pulse Width Modulation*) controla a velocidade dos motores. O gerador de sinal PWM foi implementado na FPGA. A Figura 26 apresenta o diagrama de componentes do projeto Controle de Robô para ilustração dos componentes do projeto e comunicação entre eles.

Figura 26 – Diagrama de componentes do projeto Controle de Robô.

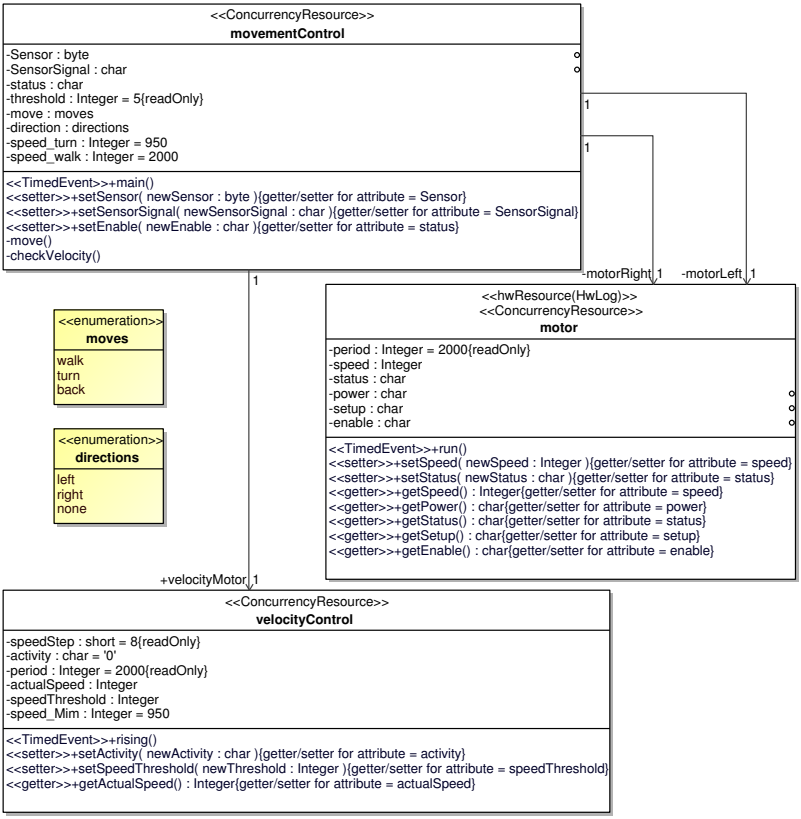


Fonte: Produção do próprio autor.

Na Figura 27 é apresentado o diagrama de classes especificado para atender aos requisitos funcionais do projeto. O controle do robô é feito pela classe *movementControl*, que é responsável por determinar a direção e velocidade dos motores. Esse comportamento é apresentado na Figura 28. O movimento do robô é controlado pelo método *move()* e a velocidade pelo método *checkVelocity()*. A velocidade é determinada por meio da classe *velocityControl*, que é responsável por controlar a velocidade dos motores e implementar

uma rampa de velocidade. A classe *motor* é responsável por gerar o sinal PWM.

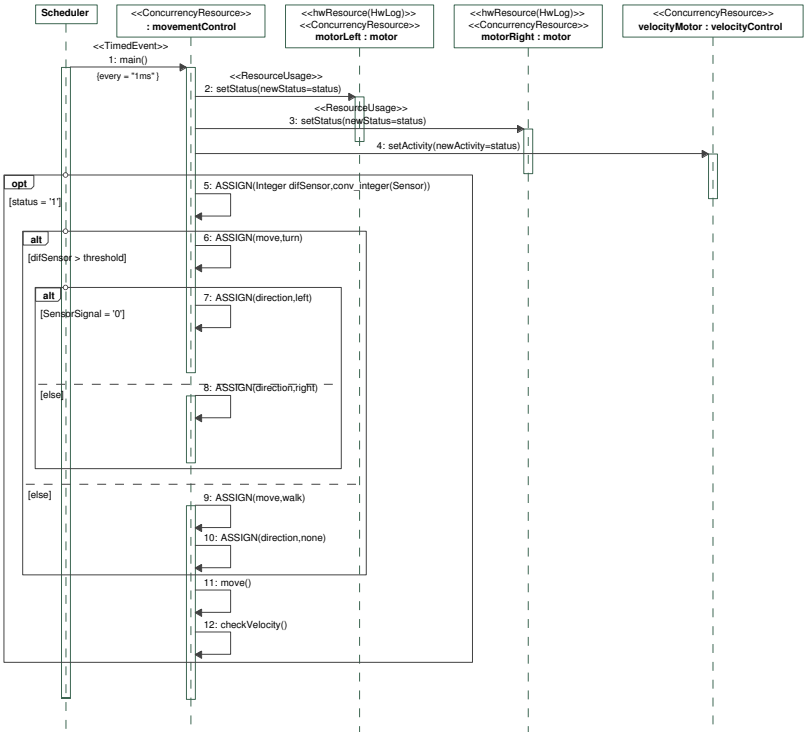
Figura 27 – Diagrama de classes do Controle de Robô Autônomo.



Fonte: Produção do próprio autor.

A Listagem 5.1 apresenta o extrato do código VHDL gerado para o comportamento do método *main* da classe *movementControl*. As linhas 21-36 e 37-41 apresentam os comportamentos dos métodos *move()* e *checkVelocity()*, pois são processos executados de forma síncrona. Estes comportamentos foram especificados nos diagramas de sequência apresentados nas Figuras 42 e 43 do Apêndice B. As demais linhas de código foram geradas a partir do diagrama de sequência apresentado na Figura 28.

Figura 28 – Diagrama de sequência do método *main* da classe *movementControl*.



Fonte: Produção do próprio autor.

Listagem 5.1 – Extrato da descrição VHDL para a entidade *movementControl*.

```
1  ...
2  main: process (SensorSignal , Sensor , status )
3   variable difSensor : INTEGER RANGE -32767 TO +32767 := 0;
4  begin
5   motorLeftstatus <= status;
6   motorRightstatus <= status;
7   velocityMotoractivity <= status;
8   if ( status = '1' ) then
9    difSensor := conv_integer(Sensor);
10   if ( difSensor > threshold ) then
11    move <= turn;
12    if ( SensorSignal = '0' ) then
```

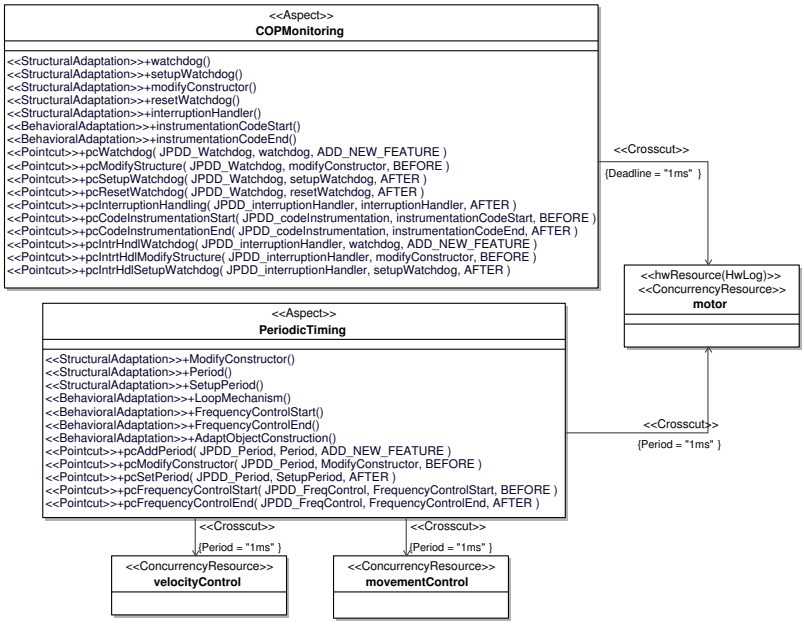
```
13      direction <= left;
14     else
15      direction <= right;
16     end if;
17    else
18     move <= walk;
19     direction <= none;
20    end if;
21    -- Synchronous process move from movementControl
          component
22    if ( move = walk ) then
23     motorLeftstatus <= '1';
24     motorRightstatus <= '1';
25    else
26     if ( direction = left ) then
27      motorLeftstatus <= '0';
28      motorRightstatus <= '1';
29     elsif ( direction = right ) then
30      motorLeftstatus <= '1';
31      motorRightstatus <= '0';
32     end if;
33    end if;
34    velocityMotorspeedThreshold <= speed_walk;
35    velocityMotorspeedThreshold <= speed_turn;
36    -- End of synchronous process
37    -- Synchronous process checkVelocity from movementControl
          component
38    speedMotors := velocityMotoractualSpeed ;
39    motorLeftspeed <= speedMotors;
40    motorRightspeed <= speedMotors;
41    -- End of synchronous process
42   end if;
43 end process main;
44 ...
```

5.2.1 Aplicação dos Aspectos

Dois requisitos não funcionais foram implementados no projeto do controle de robô: (i) requisito de periodicidade para os processos periódicos; (ii) requisito de controle de falha para os recursos de hardware controlados pela FPGA. O requisito de tempo foi tratado pelo aspecto *PeriodicTiming* (ver detalhes na Seção 4.3.2.2.1), e o requisito de controle de falha foi tratado pelo aspecto *COPMonitoring* (ver detalhes na Seção 4.3.2.2.3). A Figura 29 apresenta o diagrama ACOD especificado para o controle do robô.

A Listagem 5.2 apresenta o código gerado para o método *run* da classe *Motor* com os aspectos aplicados. O aspecto *PeriodicTiming* efetua as seguintes adaptações:(i) as linhas 7-8 são inseridas pela adaptação *Period*;(ii) 17-21 são inseridas pela adaptação *ModifyConstructor*; (iii) as linhas 24-27

Figura 29 – Diagrama ACOD do projeto Robô.



Fonte: Produção do Próprio autor.

são inseridas pela adaptação *SetupPeriod*; e (iv) as linhas 51-53 e 57 são inseridas pela adaptação *FrequencyControl*. O aspecto *COPMonitoring* efetua as seguintes adaptações: (i) as linhas 3-5 são inseridas pela adaptação *watchdog*; (ii) as linhas 10-15 são inseridas pela adaptação *modifyConstructor*; (iii) as linhas 29-33 são inseridas pela adaptação *setupWatchdog*; e (iv) as linhas 35-46 são inseridas pela adaptação *resetWatchdog*.

Listagem 5.2 – Extrato da descrição VHDL do comportamento *run* da entidade *Motor*.

```
1  ...
2  -- COPMonitoring
3  signal watchdogSignal : BIT:= '0';
4  signal watchdogReset : BIT:= '0';
5  constant watchdogTimer : INTEGER RANGE -2147483647 TO
       +2147483647:= 1000000;
6  -- PeriodicTiming
7  constant runThreshold : INTEGER RANGE -2147483647 TO
       +2147483647:= 1000000;
```

```
8  signal runClockdiv : BIT:= '0';
9  -- COPMonitoring
10 component watchdog IS
11  port( clock : in STD_LOGIC;
12        watchdogSignal: in BIT;
13        watchdogTimer: in INTEGER RANGE -2147483647 TO
              +2147483647;
14        watchdogReset: out BIT);
15 end component;
16 -- PeriodicTiming
17 component clockDiv IS
18  port( clock : in STD_LOGIC;
19        threshold: in integer range -2147483647 TO
              +2147483647;
20        clockdiv : out BIT);
21  end component;
22 begin
23  -- PeriodicTiming
24  runDivider: clockDiv port map(
25    clock => clock,
26    threshold => runThreshold,
27    clockdiv => runClockdiv );
28  -- COPMonitoring
29  motorWatchdog: watchdog port map(
30    clock => clock,
31    watchdogSignal => watchdogSignal,
32    watchdogTimer => watchdogTimer,
33    watchdogReset => watchdogReset );
34  -- COPMonitoring
35  resetWatchdog:process (clock)
36   variable count: integer := 0;
37  begin
38   if (clock'EVENT and clock='1') then
39    if (count < watchdogTimer) then
40     count := count + 1;
41    else
42     count := 0;
43     watchdogSignal <= '0';
44    end if;
45   end if;
46  end process resetWatchdog;
47
48  run: process  (watchdogReset, runClockdiv, reset, speed ,
         status )
49   ...
50   -- PeriodicTiming
51   if (reset = '1') then
52    -- variables initialization
53   elsif (runClockdiv'EVENT and runClockdiv='1') then
54   -- End PeriodicTiming
55    ...
56   -- PeriodicTiming
```

```
57   end if;
58  ...
```

A Listagem 5.3 apresenta o extrato da descrição da entidade *movementControl*, que foi afetada pelo aspecto *COPMonitoring*. Um tratador de interrupção é adicionado para cada motor pelo aspecto *COPMonitoring* (linhas 2-16), para reinicializar os sinais que controlam os motores. Sempre que um sinal do motor é lido/escrito, verifica-se ocorreu ou não uma interrupção como, por exemplo, nas linha 20 e 23.

Listagem 5.3 – Extrato da descrição VHDL para a entidade *movementControl* após a implementação dos aspectos.

```
1  ...
2  handlingInterruptionmotorLeft:process (
       watchdogResetmotorLeft)
3  begin
4   if (watchdogResetmotorLeft'EVENT and
        watchdogResetmotorLeft='1') then
5    -- reset the state of signals controled by architecture
6    resetmotorLeft <= '1';
7   end if;
8  end process handlingInterruptionmotorLeft;
9
10 handlingInterruptionmotorRight:process (
       watchdogResetmotorRight)
11 begin
12  if (watchdogResetmotorRight'EVENT and
        watchdogResetmotorRight='1') then
13   -- reset the state of signals controled by architecture
14   resetmotorRight <= '1';
15  end if;
16 end process handlingInterruptionmotorRight;
17
18 main: process  (mainClockdiv, reset, watchdogResetmotorLeft
       , watchdogResetmotorRight, SensorSignal , status ,
       Sensor )
19  ...
20   if (watchdogResetmotorLeft = '0') then
21    motorLeftstatus <= status;
22   end if;
23   if (watchdogResetmotorRight = '0') then
24    motorRightstatus <= status;
25   end if;
26   velocityMotoractivity <= status;
27   if ( status = '1' ) then
28    ...
```

5.2.2 Análise dos Resultados

Esta seção apresenta a análise dos resultados das métricas coletadas para o projeto Controle de Robô. Foram feitas duas análises. Primeiramente, comparou-se a versão do projeto com o código VHDL escrito manualmente em relação a versão com o código gerado pela ferramenta GenERTiCA. Essa versão não inclui a implementação dos aspectos para o tratamento dos requisitos não funcionais. Esta análise permitiu verificar as vantagens e desvantagens na utilização da ferramenta para a geração do código VHDL.

A segunda análise foi feita com as duas versões do código gerado pela ferramenta GenERTiCA, sendo uma versão sem a aplicação dos aspectos e a outra versão com o uso dos aspectos. Entretanto, ambas as versões possuíam o mesmo conjunto de requisitos funcionais e não funcionais implementados, sendo que na versão sem o uso dos aspectos, a implementação dos requisitos não funcionais foi feita manualmente no código gerado.

É importante destacar que, antes da modelagem e implementação do controle de robô na abordagem AMoDE-RT, o projeto foi primeiro implementado manualmente em VHDL. Nesta implementação a rampa de velocidade foi implementada em conjunto com a entidade motor. Ambas as versões (código escrito manualmente e código gerado) possuem os mesmos requisitos funcionais e não funcionais implementados.

Na primeira análise, a versão gerada automaticamente não inclui a implementação dos aspectos para o tratamento dos requisitos não funcionais. O requisito não funcional comum para ambas as versões do código VHDL (manual e gerado) é o de periodicidade, adicionado manualmente na versão gerada pela ferramenta. A Tabela 7 apresenta um comparativo da utilização de recursos da FPGA entre a versão implementada manualmente e a versão gerada automaticamente pela especificação em UML.

Tabela 7 – Análise do tamanho do projeto Controle de Robô Autônomo.

		Código Manual	**Código Gerado**	
Métricas	**Disp**	**Utilizado**	**Utilizado**	**Variação**
Slices	2.278	29	40	38%
FFs	18.224	35	73	109%
LUTs	9.112	79	140	77%
IOB	232	16	20	25%
Nr. de Linhas		180	204	13%

Fonte: Produção do próprio autor.

Para esta análise não foram levantadas as métricas de reusabilidade pois não há a implementação dos aspectos. A descrição codificada manualmente possui 180 linhas, enquanto que o código gerado possui 204 linhas, não contabilizando linhas em branco ou comentários. Essa diferença de número de linhas se deve ao fato de que, na versão do código manual, o tratamento dos requisitos não funcionais foi implementado dentro dos componentes que apresentavam tais requisitos; ou seja, não foram utilizados os componentes para a implementação dos requisitos não funcionais que foram desenvolvidos na versão gerada automaticamente. Assim, diminuindo a quantidade de linhas na versão do código manual. Embora em VHDL o número de linhas de código não representa exatamente o tamanho do circuito, ele representa o esforço necessário para produzir o mesmo circuito. Assim, a versão do código gerado com maior número de linhas, foi a versão com o menor esforço de desenvolvimento. Verificando a Tabela 7, pode-se observar que a versão do código gerado consumiu mais recursos do que a versão manual, que, na prática, possuíam o mesmo tamanho de descrição, i.e. se trata do mesmo tamanho de projeto. A quantidade maior de recursos está no maior número de *slices*, LUTs, FFs e IOBs utilizadas pela versão de código gerado, com aumento médio de 52%[1]. Esse aumento pode estar relacionado aos componentes utilizados para a implementação dos requisitos não funcionais e a separação de responsabilidades entre as classes, que não ocorre na versão manual. Por exemplo, na versão do código gerado, para cada sinal de um componente adicionado, é necessário criar um sinal na classe que o instancia para manipular os seus valores. Esses detalhes de mapeamento acabam impactando no consumo de área. Além disso, otimizações no código podem ser realizadas na versão automatizada visando diminuir a utilização dos recursos da FPGA. No entanto, dada a restrição de tempo para a realização deste trabalho, as otimizações mencionadas não foram realizadas neste trabalho.

A Tabela 8 apresenta os resultados da análise de desempenho da versão manual em relação à versão de código gerado. Pode-se observar que a versão de código gerado possui um tempo de atraso menor no caminho crítico, o que resulta em um desempenho superior, podendo chegar a 275Mhz de frequência. Isso é possível devido a separação de funcionalidades entre os componentes, que gera circuitos mais simples e, consequentemente, diminui o atraso. Contudo, é importante salientar que engenheiros mais experientes em descrever circuitos com VHDL podem ser capazes de produzir uma implementação manual mais otimizada. Por outro lado, as regras de mapeamento também podem ser aperfeiçoadas para a geração de um código mais otimizado. A Tabela 8 também apresenta o índice de escalabilidade da versão

[1] O aumento médio é a soma do percentual de aumento de todas as métricas avaliadas, dividido pela quantidade de métricas.

do código gerado em relação a versão do código manual. O índice de escalabilidade é calculado de acordo com as Equações 4.2 e 4.3 apresentadas na Seção 4.2. Esse índice tem como base a taxa de utilização dos recursos da FPGA, apresentados na Tabela 7 e a frequência máxima do circuito. O índice de $1,49$ indica que a nova versão é escalável em relação a velocidade e consumo de área, pois o aumento da velocidade foi proporcional e maior que o aumento da área utilizada.

Tabela 8 – Análise do desempenho do projeto Controle de Robô Autônomo.

Métricas	**Código Manual**	**Código Gerado**	**Variação**
Maior Caminho Crítico	7,229ns	3,634ns	-50%
Máxima Frequência	138,330MHz	275,202MHz	99%
Escalabilidade			1,49

Fonte: Produção do próprio autor.

A segunda análise foi a comparação da versão do código VHDL sem o uso dos aspectos em relação a versão que utiliza os aspectos no modelo UML, sendo ambas as versões do código geradas pela ferramenta GenERTiCA. Após a especificação dos aspectos que tratam os requisitos não funcionais do projeto, o código VHDL foi gerado e sintetizado. A Tabela 9 apresenta o resultado das métricas do código gerado contendo os aspectos, comparando com os valores para o mesmo projeto sem os aspectos. Vale ressaltar que, ambas as versões possuem os mesmos requisitos funcionais e não funcionais, sendo que na versão sem os aspectos, os requisitos não funcionais foram implementados manualmente. Além disso, os valores apresentados na Tabela 9 são diferentes dos valores da Tabela 7 na versão do código gerado automaticamente mas sem o uso dos aspectos, porque nessa nova versão há mais requisitos não funcionais implementados, que não existiam na versão manual do código. As linhas de código referente aos componentes da plataforma adicionados pelos aspectos, foram considerados na análise. Essa mesma regra é válida para os demais estudos de caso.

Observa-se um aumento alto no número dos recursos utilizados da FPGA, o que, de fato, é esperado, uma vez que os aspectos adicionam funcionalidades e componentes ao projeto, embora o número de linhas não tenha o mesmo percentual de aumento. O que indica que para as FPGAs o número de linhas não é uma boa métrica para analisar o tamanho do projeto. Entretanto, em relação ao desempenho do projeto (ver Tabela 10) o impacto não foi significativo quando comparado ao aumento da área da FPGA. O índice de escalabilidade $(0,57)$ indica que o desempenho do projeto não obteve o

Tabela 9 – Análise do tamanho do projeto Controle de Robô Autônomo. Comparativo da utilização dos aspectos.

		Código Sem Aspectos	**Código Com Aspectos**	
Métricas	**Disp**	**Utilizado**	**Utilizado**	**Variação**
Slices	2.278	39	126	223%
FFs	18.224	73	214	193%
LUTs	9.112	139	439	216%
IOB	232	20	20	0%
Nr. de Linhas		276	373	35%

Fonte: Produção do próprio autor.

mesmo crescimento em relação ao área utilizada, podendo ocorrer um gargalo no desempenho em projetos maiores. No entanto, análises adicionais de causa/efeito são necessárias para melhorar a compreensão das relações entre o código adicional introduzido pelos aspectos e os algoritmos (e.g. *place/routing*) implementados nas ferramentas de síntese. Esses são indicativos de que o uso dos aspectos para o tratamento dos requisitos não funcionais, embora impacte na área da FPGA, não ocasiona o mesmo impacto sobre o desempenho do projeto.

Tabela 10 – Análise do desempenho do projeto Controle de Robô Autônomo. Comparativo da utilização dos aspectos.

Métricas	**Código Sem Aspectos**	**Código Com Aspectos**	**Variação**
Maior Caminho Crítico	3,634ns	3,816ns	5%
Máxima Frequência	275,202MHz	262,089MHz	-5%
Escalabilidade			0,57

Fonte: Produção do próprio autor.

A Tabela 11 apresenta a análise de reusabilidade e impacto dos aspectos *PeriodicTiming* e *COPMonitoring* no projeto Controle de Robô. As colunas LOC (LOC,LOWC,LOAC) indicam, respectivamente, o número de linhas do código original (sem a inserção dos aspectos), número de linhas de código com os aspectos e o número de linhas da especificação dos aspectos. A coluna CDLOC indica o número de vezes em que houve alteração de contexto entre o código original e o código do aspecto inserido, que auxilia na

medição da taxa de entrelaçamento do aspecto. A coluna AB ("inchaço") é um índice que índica quanto o código do aspecto aumentou após a sua aplicação e entrelaçamento no código final. A taxa de entrelaçamento e o índice de inchaço mostram o nível de reutilização do aspecto dentro do projeto. Segundo Cardoso et al. (2012), quanto maior os valores maior a reutilização e impacto do aspecto sobre o código final.

Comparando os resultados com o valores apresentados em (CARDOSO et al., 2012), a taxa de entrelaçamento média de $15,5\%$ é semelhante a apresentada naquele trabalho de $15,63\%$. O impacto do código do aspecto sobre o código final no projeto Controle de Robô também é considerada alta, quando o índice de inchaço é superior a 1.

Tabela 11 – Análise de reusabilidade e impacto dos aspectos do projeto Controle de Robô Autônomo.

Aspecto	LOC	LOWC	LOAC	CDLOC	TR	AB
Periodic Timing	198	241	31	38	16%	1,38
COPMonitoring	198	275	64	40	15%	1,20

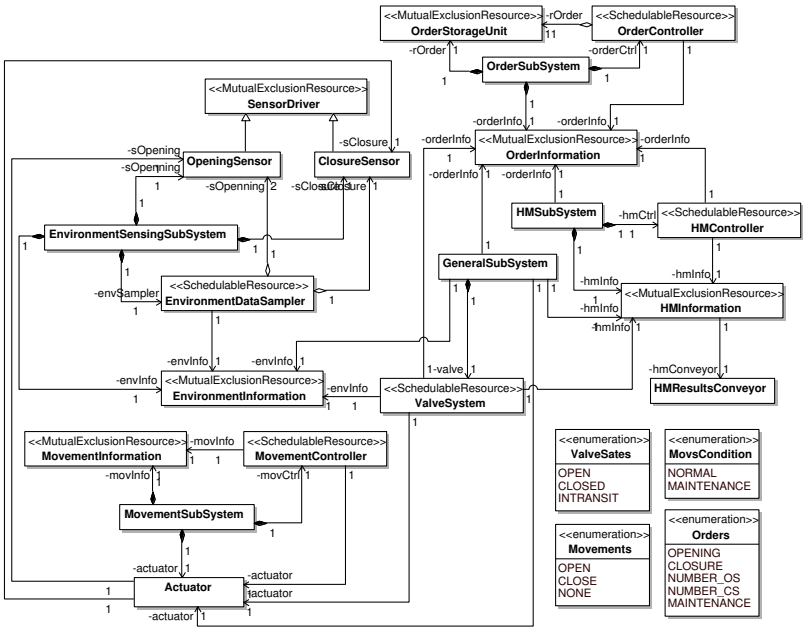
Fonte: Produção do próprio autor.

5.3 CONTROLE AUTOMÁTICO DE VÁLVULA

Este estudo de caso foi apresentado originalmente em (MOREIRA, 2012) para validação da metodologia e regras de mapeamento propostas naquele trabalho. Com o objetivo de comparar os resultados do presente trabalho com os apresentados em (MOREIRA, 2012), utilizou-se o sistema de controle de válvula. Este sistema é constituído por um controle automático de válvula que regula o fluxo de fluidos e sensores que captam informações da válvula. O sistema é dividido em quatro subsistemas: subsistema de sensoriamento que é responsável por coletar dados da válvula; subsistema do atuador, que controla os movimentos da válvula; subsistema de pedidos que controla os comandos que o controle de válvula deve executar; e subsistema de gerenciamento da saúde da válvula, que afere a saúde física da válvula e avalia a necessidade de manutenção do dispositivo (MOREIRA, 2012). A Figura 30 apresenta o diagrama de classes do controle automático de válvula proposto por Moreira (2012).

Segundo Moreira (2012), não foi possível gerar o código completo a partir do diagrama de classes apresentado na Figura 30 em seu trabalho. As regras de mapeamento propostas por aquele autor não contemplavam relaci-

Figura 30 – Diagrama de classes do Controle de Válvula proposto por Moreira (2012).



Fonte: (MOREIRA, 2012).

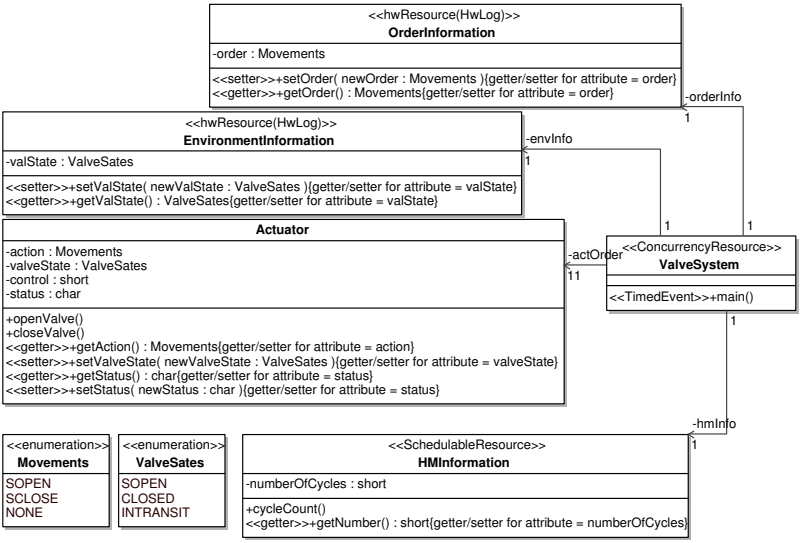
onamentos de herança, agregação e composição, ou elementos como classes abstratas e tipos enumerados.

Com as regras de mapeamento propostas no presente trabalho, foi possível gerar o código para toda a estrutura do diagrama de classes da Figura 30. Todo o código gerado para a estrutura foi compilado/sintetizado sem erros. Entretanto não foi possível a síntese do código gerado por falta da definição do comportamento de cada método. Foram gerados 20 arquivos VHDL com o total de 1.034 linhas de código.

Em seguida, criou-se uma nova versão do sistema de controle de válvula. Todos os requisitos funcionais e não funcionais foram incluídos, porém, a nova versão é mais enxuta em termos de número de elementos. Também foi incluída a especificação do comportamento. A Figura 31 apresenta o diagrama de classes da nova versão do sistema para o controle de válvula, adaptado de (MOREIRA, 2012). Esse modelo é semelhante ao apresentado por Moreira (2012) com a diferença que no modelo da Figura 31 foram utiliza-

dos tipos enumerados e alguns atributos não utilizados foram eliminados do diagrama.

Figura 31 – Diagrama de classes resumido do Controle de Válvula.

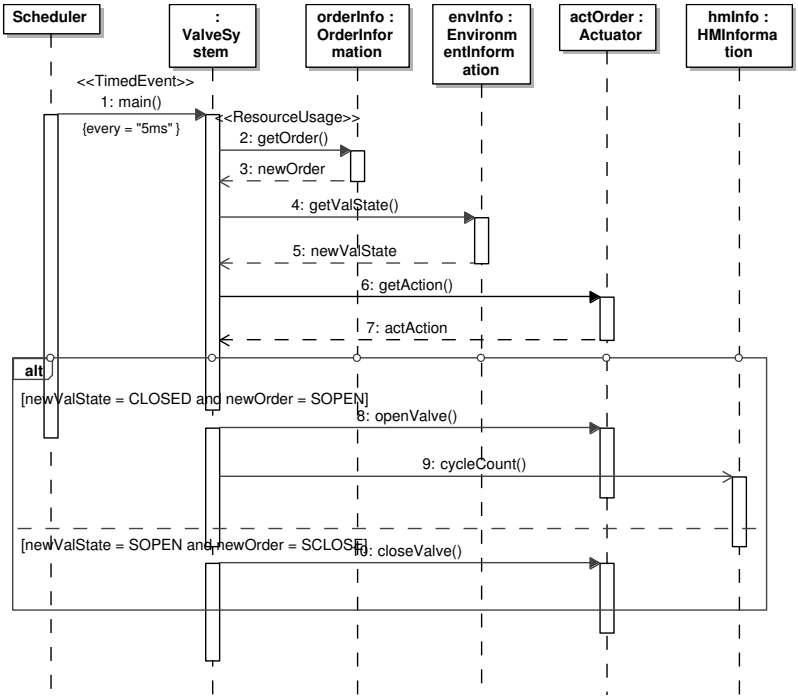


Fonte: Produção do próprio autor. Adaptado de (MOREIRA, 2012).

A Figura 32 apresenta o diagrama de sequência do método *main* da classe *ValveSystem*. Originalmente a classe *HMInformation* verificava quando a válvula era aberta, durante o processamento do método *main* (MOREIRA, 2012). O comportamento desse método foi alterado. Essa função foi passada para o método *cycleCount* da classe *HMInformation*, que é chamado pela classe *ValveSystem*.

As regras de mapeamento propostas em (MOREIRA, 2012) não permitiram a geração do código VHDL para o diagrama de sequência da Figura 32, pois não era prevista a troca de mensagens nas regras de mapeamento. Com as novas regras de mapeamento propostas no presente trabalho foi possível a geração de todo o código. Foram gerados 6 arquivos com total de 147 linhas. Para que fosse possível a síntese dos arquivos VHDL gerados, foi incluído o código apresentado na Listagem 5.4 no comportamento das classes *EnvironmentInformation*, *OrderInformation* e *Actuator*, pois sem um comportamento a ferramenta de síntese não processa as entidades geradas por essas classes. Com isso, a síntese do projeto ocorreu sem erros.

Figura 32 – Diagrama de sequência do método *main* da classe *ValveSystem*.



Fonte: Produção do próprio autor. Adaptado de (MOREIRA, 2012).

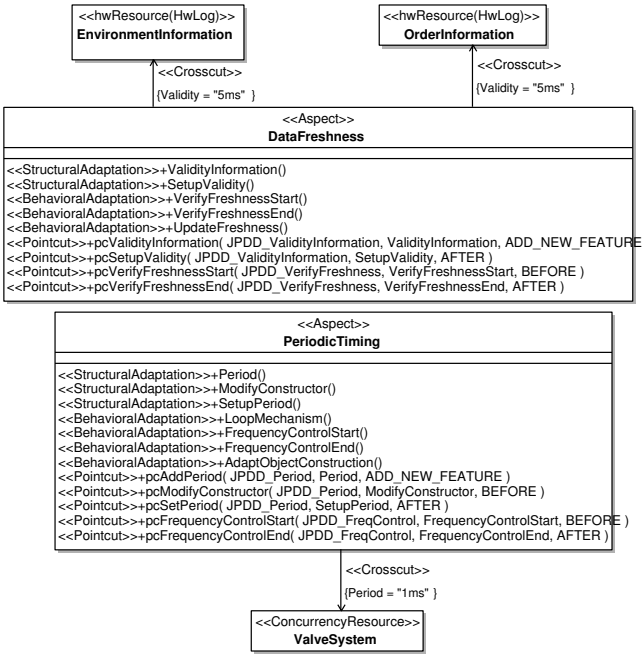
Listagem 5.4 – Descrição de um processo genérico.

```
1  main: process( clock, reset )
2    variable count: integer := 0;
3  begin
4    if (reset='1') then
5
6    elsif (clock'EVENT and clock = '1') then
7      if (count < 100000) then
8        count := count + 1;
9        --<atributo classe> <= valor;
10     else
11       count := 0;
12       --<atributo classe> <= valor;
13     end if;
14   end if;
15 end process main;
```

5.3.1 Aplicação dos Aspectos

Foram identificados dois requisitos não funcionais no projeto do controle de válvula. O requisito de periodicidade para o método *main* da classe *ValveSystem* e o requisito de validade dos dados das classes *EnvironmentInformation* e *OrderInformation*. O requisito de periodicidade define a restrição de tempo entre cada execução do método *main*. Este requisito é implementado pelo aspecto *PeriodicTiming*. O requisito de validade dos dados define quanto tempo o valor dos atributos das classes *EnvironmentInformation* e *OrderInformation* é valido. Após o tempo determinado, o valor dos atributos não é mais válido e o sistema deve aguardar por uma nova geração desses dados. Este requisito é implementado pelo aspecto *DataFreshness*. A Figura 33 apresenta o diagrama ACOD do projeto Controle de Válvula.

Figura 33 – Diagrama ACOD do projeto Válvula resumido.



Fonte: Produção do próprio autor.

A Listagem 5.5 apresenta o código gerado para o método *main* da classe *ValveSystem* com os requisitos não funcionais implementados. Foram

incluídas linhas de comentário para identificar o inicio e fim de uma adaptação inserida como, por exemplo, nas linhas 5 e 9. As linhas 4-6 são geradas pelo aspecto *PeriodicTiming*. As linhas 8,11,13 e 16 são geradas pelo aspecto *DataFreshness*. Este código é um extrato da descrição VHDL gerada para a classe *ValveSystem*, ocorrendo mais adaptações em outros trechos do código, conforme necessário.

Listagem 5.5 – Extrato da descrição VHDL do processo *main* da entidade *ValveSystem*

```
1  main: process  (mainClockdiv, reset)
2   ...
3    -- Periodic Timing
4    if (reset = '1') then
5      -- variables initialization
6    elsif (mainClockdiv'EVENT and mainClockdiv='1') then
7      --DataFreshness
8      if (orderInfovalidityorder = '0') then
9        newOrder := orderInfoorder ;
10     --DataFreshness
11     end if;
12     --DataFreshness
13     if (envInfovalidityvalState = '0') then
14       newValState := envInfovalState ;
15     --DataFreshness
16     end if;
17  ...
```

A Listagem 5.6 apresenta a descrição VHDL gerada para a classe *OrderInformation*. O processo *setupValidity* foi gerado pelo aspecto *DataFreshness* para controle do tempo de validade dos dados desta entidade. O mesmo comportamento foi adicionado para a classe *EnvironmentInformation*.

Listagem 5.6 – Descrição da entidade *OrderInformation*

```
1  entity OrderInformation is
2    Port ( clock : in STD_LOGIC;
3           reset : in STD_LOGIC;
4           order : INOUT Movements;
5           -- DataFreshness
6           validityorder : OUT BIT);
7           --
8    end OrderInformation;
9  architecture Behavioral of OrderInformation is
10   -- Datafreshness
11   constant validityDeadLine : INTEGER RANGE -2147483647
12                               TO +2147483647:= 5000000;
13   signal oldorder : Movements;
14   --
15 begin
16   -- Datafreshness
```

```
17   setupValidity:process (clock)
18     variable countValidityorder: integer := 0;
19   begin
20     if (clock'EVENT and clock='1') then
21       if (oldorder /= order) then
22         validityorder <= '0';
23         countValidityorder := 0;
24         oldorder <=  order;
25       else
26         if (countValidityorder < validityDeadLine) then
27           countValidityorder := countValidityorder + 1;
28         else
29           countValidityorder := 0;
30           validityorder <= '1';
31         end if;
32       end if;
33     end if;
34   end process setupValidity;
35   --
36 end Behavioral;
```

5.3.2 Análise dos Resultados

A Tabela 12 apresenta o comparativo da utilização dos recursos da FPGA entre a versão com os requisitos não funcionais implementados manualmente e a versão com o uso dos aspectos para a implementação dos requisitos não funcionais. Ressalta-se que na versão com o uso dos aspectos, o código apresentado na Listagem 5.4 foi adicionado nas entidades *Actuator*, *OrderInformation* e *EnvironmentInformation* para manter as mesmas características da versão do projeto com os requisitos não funcionais implementados manualmente. Observa-se que a versão com os requisitos não funcionais implementados manualmente consumiu menos recursos (número de *slices*) da FPGA do que a versão com o uso dos aspectos.

A Tabela 13 apresenta o comparativo de desempenho do projeto Controle de Válvula entre a versão sem os aspectos com a versão com os aspecto. A versão sem os aspectos apresentou uma frequência maior. A implementação dos aspectos nesse projeto comprometeu em 8% o desempenho do sistema. A versão com os aspectos apresentou um índice $0,64$ de escalabilidade, que indica que essa versão não é escalável. Índices de escalabilidade menores que 1 indicam que o projeto não é escalável, pois conforme cresce o projeto, esse valor tende a chegar a zero, comprometendo negativamente o desempenho.

A Tabela 14 apresenta a análise de reusabilidade e impacto dos aspecto sobre o projeto Controle de Válvula. Observa-se que o aspecto *PeriodicTiming* teve pouco impacto sobre o projeto. De fato, apenas uma classe

Tabela 12 – Análise do tamanho do projeto Controle de Válvula.

		Código Sem Aspectos	Código Com Aspectos	
Métricas	**Disp**	**Utilizado**	**Utilizado**	**Variação**
Slices	2.278	12	39	142%
FFs	18.224	18	67	272%
LUTs	9.112	42	143	240%
IOB	232	17	19	12%
Nr. de Linhas		206	273	32,5%

Fonte: Produção do próprio autor.

Tabela 13 – Análise do desempenho do projeto Controle de Válvula.

Métricas	**Código Sem Aspectos**	**Código Com Aspectos**	**Variação**
Maior Caminho Crítico	3,301ns	3,572ns	8%
Máxima Frequência	302,897MHz	279,963MHz	-8%
Escalabilidade			0,64

Fonte: Produção do próprio autor.

foi afetada pelo aspecto. Já o aspecto *DataFreshness* teve alto impacto sobre o projeto, mantendo a média de entrelaçamento do código com os outros estudos de caso analisados.

Tabela 14 – Análise de reusabilidade e impacto dos aspectos do projeto Controle de Válvula.

Aspecto	**LOC**	**LOWC**	**LOAC**	**CDLOC**	**TR**	**AB**
Periodic Timing	181	198	31	12	6%	0,54
DataFreshness	181	233	69	28	12%	1,85

Fonte: Produção do próprio autor.

5.4 RELÓGIO

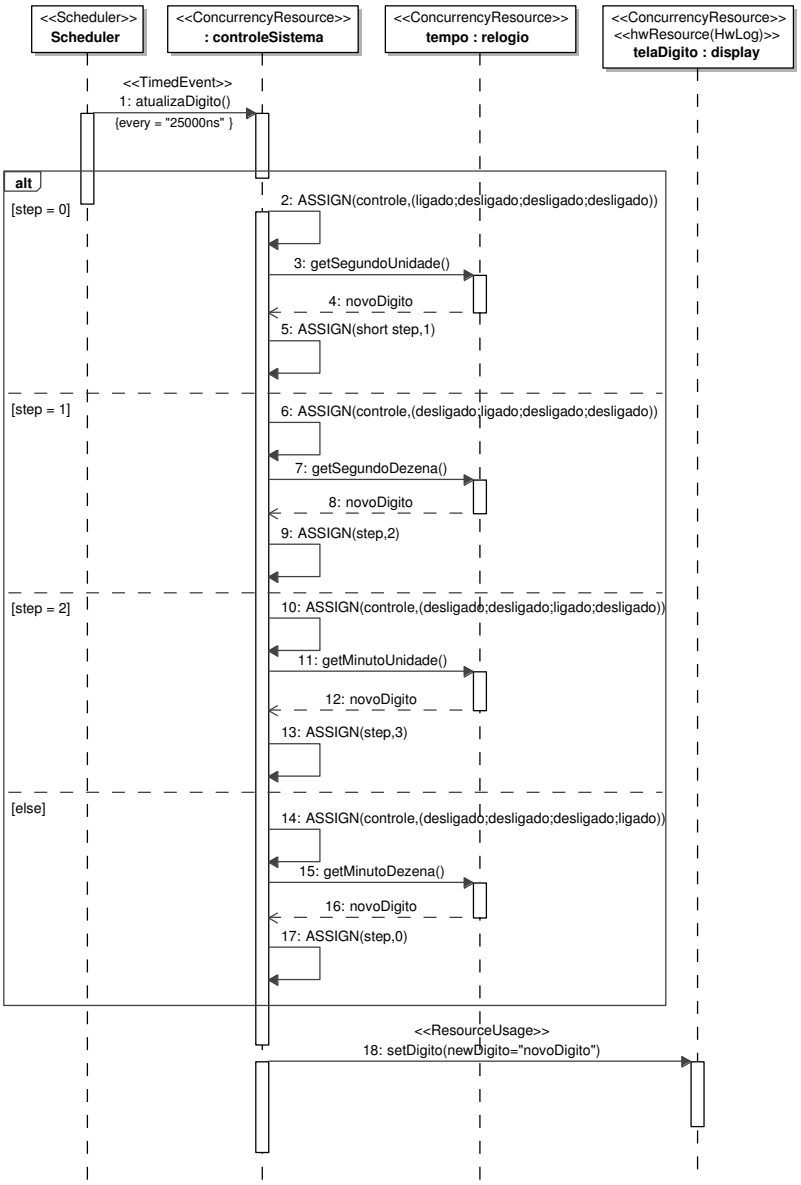
Este estudo de caso consiste em um relógio que apresenta os minutos e segundos em um display de sete segmentos de LED. O display utilizado

é disponibilizado na placa de prototipagem Nexys 3. Cada dígito possui uma porta individual para ativá-lo (ligar/desligar) e é formado por sete (7) segmentos de LED. Assim é possível ligar e desligar cada dígito individualmente. Da mesma forma, cada segmento é acionado por uma porta. Entretanto, uma porta aciona o mesmo LED para os quatro dígitos ao mesmo tempo. Para mostrar dígitos diferentes, o projeto do relógio atribui o número corrente do minuto ou segundo ao display, liga o dígito que deve ser apresentado e desliga os demais. Contudo, usando uma alta taxa de frequência de atualização dos números no display, torna-se imperceptível aos olhos humanos a troca dos dígitos. A Figura 10, apresentada na Seção 4.3.1, mostra o diagrama de classes do projeto Relógio.

A classe *Segmento* representa um segmento de LED e a classe *Display* representa um dígito de sete segmentos. A classe *Relógio* é responsável por contar os minutos e segundos, gerando um valor individual para cada dígito. A classe *controleSistema* controla o sistema lendo os valores gerados pela classe *Relogio* e enviando para a classe *Display* apresentá-lo. É importante destacar que a classe *Display* representa apenas um dígito, pois, devido as restrições impostas pelo projeto da placa Nexys 3, é possível controlar apenas um único dígito por vez. Assim, a classe *controlaSistema* é responsável por ligar e desligar individualmente cada dígito. A Figura 34 apresenta o diagrama de sequência do método *atualizaDigito* da classe *controlaSistema*.

O comportamento do método *atualizaDigito* representa um comportamento sequencial, que, a cada período, aciona um dígito diferente e atualiza o valor do display. A Listagem 5.7 apresenta o código gerado para o comportamento do método *atualizaDigito*. O código gerado é 100% sintetizável e funcional, com a exceção da necessidade de substituir o caractere “;” por “,” nas linhas 10,14,18 e 22. Esse problema é devido a uma limitação da ferramenta MagicDraw para a definição de expressões de seleções (*if,else*) e mensagens de ASSIGN em diagramas de sequência. Não é possível a definição de expressões mais complexas, pois o campo para entrada de informação é do tipo texto. Com isso, a ferramenta GenERTiCA não permite o uso de “,” em expressões de mensagens de ASSIGN. Esse caractere é interpretado como separador de atributos do ASSIGN. Por isso, um “;” foi utilizado no lugar do “,” no modelo, o que resulta em um erro na sintaxe do código VHDL.

Figura 34 – Diagrama de sequência do método *atualizaDigito* da classe *controleSistema*.



Fonte: Produção do próprio autor.

Listagem 5.7 – Decrição VHDL do comportamento *atualizaDigito* da entidade *controleSistema*.

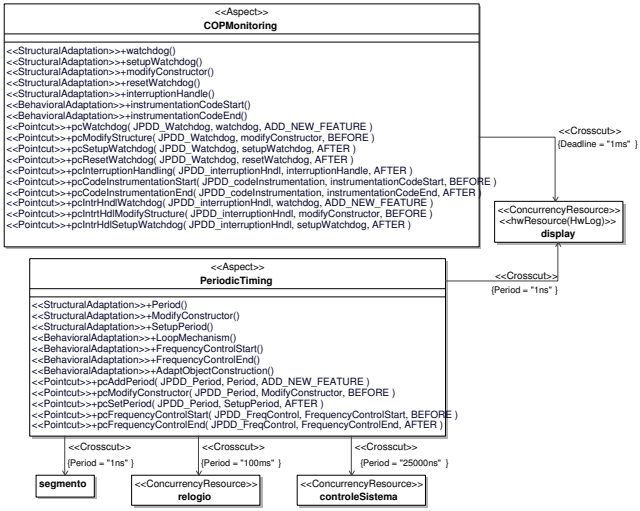
```
1 atualizaDigito: process (clock, reset)
2  variable novoDigito : INTEGER RANGE -252 TO +252 := 0;
3  variable step : INTEGER RANGE -252 TO +252 := 0;
4 begin
5  if (count < 25000 then
6   count := count + 1;
7  else
8   count := 0;
9   if ( step = 0 ) then
10   controle <= (ligado;desligado;desligado;desligado);
11   novoDigito := temposegundoUnidade ;
12   step := 1;
13  elsif ( step = 1 ) then
14   controle <= (desligado;ligado;desligado;desligado);
15   novoDigito := temposegundoDezena ;
16   step := 2;
17  elsif ( step = 2 ) then
18   controle <= (desligado;desligado;ligado;desligado);
19   novoDigito := tempominutoUnidade ;
20   step := 3;
21  else
22   controle <= (desligado;desligado;desligado;ligado);
23   novoDigito := tempominutoDezena ;
24   step := 0;
25  end if;
26  telaDigitodigito <= novoDigito;
27 end if;
28 end process atualizaDigito;
```

5.4.1 Aplicação dos Aspectos

Neste projeto foram aplicados os aspectos *PeriodicTiming*, para controlar a periodicidade das tarefas *TimedEvent*, e o aspecto *COPMonitoring*, para controlar falhas no dispositivo display. A Figura 35 apresenta o digrama ACOD do projeto Relógio.

A Listagem 5.8 apresenta o código gerado para o comportamento do método *atualizaDigito* após a aplicação dos aspectos. Agora o código está instrumentado nas linhas 5-7 e 15, para o tratamento da periodicidade que o display deve ser atualizado.

Figura 35 – Diagrama ACOD do projeto Relógio.



Fonte: Produção do próprio autor.

Listagem 5.8 – Extrato da descrição VHDL do comportamento *atualizaDigito* da entidade *controleSistema* após a aplicação dos Aspectos.

```
1  atualizaDigito: process  (atualizaDigitoClockdiv, reset)
           variable novoDigito : INTEGER RANGE -252 TO +252 :=
       0;
2   variable step : INTEGER RANGE -252 TO +252 := 0;
3  begin
4   -- PeriodicTiming
5   if (reset = '1') then
6    -- variables initialization
7   elsif (atualizaDigitoClockdiv'EVENT and
       atualizaDigitoClockdiv='1') then
8   --
9    if ( step = 0 ) then
10    controle <= (ligado;desligado;desligado;desligado);
11    novoDigito := temposegundoUnidade ;
12    step := 1;
13   elsif ( step = 1 ) then
14     ...
15   end if
16  ...
```

5.4.2 Análise dos Resultados

A utilização da composição da classe *Segmento* na classe *Display* representou melhor o comportamento real dos recursos de hardware disponíveis para o projeto, pois o dígito do display é, de fato, uma entidade composta de segmentos. Sua conversão para o código VHDL foi satisfatória, possibilitando o uso de outros componentes, que garante a reutilização de código, no caso da FPGA, de componentes de hardware.

A Tabela 15 apresenta o comparativo do tamanho do projeto Relógio da versão do código VHDL sem o uso dos aspectos em relação a versão com o uso dos aspectos, sendo ambas as versões do código geradas pela ferramenta GenERTiCA. Vale ressaltar que, ambas as versões possuem os mesmos requisitos funcionais e não funcionais, sendo que na versão sem os aspectos, os requisitos não funcionais foram implementados manualmente.

Tabela 15 – Análise do tamanho do projeto Relógio.

		Código Sem Aspectos	**Código Com Aspectos**	
Métricas	**Disp**	**Utilizado**	**Utilizado**	**Variação**
Slices	2.278	75	103	37%
FFs	18.224	145	173	19%
LUTs	9.112	251	369	47%
IOB	232	49	49	0%
Nr. de Linhas		412	494	20%

Fonte: Produção do próprio autor.

Observa-se aumento na utilização dos recursos da FPGA, como o número de *slices*, de 37%. Contudo, não houve aumente expressivo no número de linhas de código VHDL. Tal situação ocorre devido a instanciação e mapeamento de componentes que pode ocasionar aumento na utilização de recursos. Por outro lado, a não utilização de componentes pode levar a implementação *inline* dos tratamentos executados pelo componente, que elevam o número de linhas do código afetado.

A Tabela 16 apresenta o comparativo das métricas de desempenho entre a versão do código sem os aspectos, com a versão com os aspectos para o projeto *Relogio*. A versão com os aspectos implementados apresentou desempenho melhor, dado o menor tempo de caminho crítico e aumento da frequência em 27%. O índice de escalabilidade de $1,15$ indica que a versão com a implementação dos aspectos é escalável em relação ao desempenho e consumo de área, pois houve aumento do desempenho superior ao cresci-

mento da área utilizada.

Tabela 16 – Análise do desempenho do projeto Relógio.

Métricas	**Código Sem Aspectos**	**Código Com Aspectos**	**Variação**
Maior Caminho Crítico	4,938ns	3,897ns	-21%
Máxima Frequência	202,509MHz	256,624MHz	27%
Escalabilidade			1,15

Fonte: Produção do próprio autor.

A Tabela 17 apresenta a análise de reusabilidade e do impacto dos aspectos aplicados ao projeto *Relógio*. Apesar da baixa taxa de entrelaçamento, o índice de inchaço do aspecto (coluna AB) permaneceu alto, pois este tem alto impacto sobre o código final. Isso ocorre porque diminuiu o espalhamento do código de tratamento dos requisitos não funcionais no código final. Contudo, apesar disso, a quantidade de linhas manteve-se alta, que aumenta significativamente o tamanho do código final.

Tabela 17 – Análise de reusabilidade e impacto dos aspectos do projeto Relógio.

Aspecto	**LOC**	**LOWC**	**LOAC**	**CDLOC**	**TR**	**AB**
Periodic Timing	346	385	31	36	9%	1,25
COPMonitoring	346	396	64	20	5%	2

Fonte: Produção do próprio autor.

5.5 DISCUSSÃO

Este Capítulo apresentou os estudos de caso realizados para a avaliação da abordagem proposta, das regras de mapeamento, dos aspectos que tratam os requisitos não funcionais e das métricas propostas para avaliar projetos implementados em FPGA. Foram feitas três análises: tamanho do projeto (área), desempenho e reusabilidade e impacto dos aspectos sobre o projeto.

A análise de tamanho do projeto, ou área, considerando a utilização de *slices*, mostra que houve um aumento médio de 134% na utilização dos recursos da FPGA, com a aplicação dos aspectos para o tratamento dos requisitos não funcionais. Em dois estudos de caso (controle de robô e controle de válvula) o impacto no desempenho do sistema utilizando os aspectos

foi de, respectivamente, 5% e 8% inferior a versão sem o uso dos aspectos para o tratamento dos requisitos não funcionais. Já no estudo de caso Relógio o impacto no desempenho do uso dos aspectos foi positivo, tendo 27% de aumento no desempenho do que a versão sem o uso dos aspectos. Esses resultados indicam um impacto médio de 4% mais rápido na versão com o uso dos aspectos para o tratamento dos requisitos não funcionais. Destaca-se que, a versão do código VHDL sem o uso dos aspectos também trata os mesmos requisitos não funcionais, mas a implementação foi manual. Assim, o aumento do desempenho é um indicador de que o código bem estruturado, principalmente para VHDL, pode melhorar o desempenho do projeto. Isso porque a linguagem VHDL descreve o comportamento e arquitetura do circuito, que quanto mais otimizado, melhor será o resultado do algoritmo de roteamento e localização no circuito sintetizado, melhorando o desempenho do sistema.

Além disso, observa-se que o aumento no número de linhas causado pela inserção das adaptações dos aspectos (média de 14%) não está diretamente relacionado com o aumento da área ou comprometimento do desempenho. Quanto mais otimizada a descrição do código VHDL, melhor será a utilização dos recursos de cada *slice* (FFs e LUTs), o que significa, melhor uso dos recursos da FPGA. O aumento da área utilizada foi justificado pela implementação mais otimizada dos requisitos não funcionais com o uso dos aspectos, melhorando o desempenho do sistema. Isso quer dizer que a área foi utilizada de forma mais eficiente. Assim, conclui-se que o uso dos aspectos para o tratamento dos requisitos não funcionais contribui com a estruturação do código, melhorando o desempenho e área utilizada em projetos implementados em FPGA.

A Tabela 18 apresenta um resumo da análise de reusabilidade e impacto dos aspectos sobre o código VHDL final. A taxa de entrelaçamento médio de 10% é considerada satisfatória, quando comparada com outros trabalhos semelhantes como (CARDOSO et al., 2012). O índice de impacto do aspecto sobre o código final (coluna "AB"), indica um alto impacto, que sugere alto índice de reaproveitamento dos aspectos. Esses indicadores mostram que o uso de aspecto para o tratamento de requisitos não funcionais em projetos implementados em FPGA auxilia no desenvolvimento de projetos pois promove a reusabilidade do aspecto, que terá grande impacto sobre o código final. Assim, um aspecto desenvolvido e testado, poderá ser facilmente implementado em outros projetos, podendo ter impacto ainda maior se ele afetar muitos elementos em um mesmo projeto.

A Tabela 19 apresenta um resumo do número de elementos especificados para a implementação dos aspecto em nível de modelo. Esse resumo inclui os valores de todos os estudos de caso. Esses dados indicam o esforço necessário para a especificação dos aspectos em nível de modelo em relação

Tabela 18 – Análise de reusabilidade e impacto dos aspectos.

Aspecto	Projeto	LOC	LOCW	LOCA	CDLOC	TR	AB
Periodic Timing	Robô	198	241	31	38	16%	1,38
COPMonitoring	Robô	198	275	64	40	15%	1,20
Periodic Timing	Válvula	181	198	31	12	6%	0,54
DataFreshness	Válvula	181	233	69	28	12%	1,85
Periodic Timing	Relógio	346	385	31	36	9%	1,25
COPMonitoring	Relógio	346	396	64	20	5%	2
Média		242	288	48	29	10%	1,37

Fonte: Produção do próprio autor.

aos projetos em que foram utilizados. A média de 42% de classes afetadas pelos aspectos modelados indica grande impacto dos aspectos sobre os projetos. Sem a necessidade do esforço de desenvolvimento de linhas de código para a implementação de cada aspecto ou de regras de mapeamento para sua geração (dado que, definidas as regras, elas não mudam) o esforço de implementação dos aspectos se reduz a definição dos *pointcuts* no ACOD.

Tabela 19 – Análise do esforço de modelagem.

Aspecto	Nr. JPDD	CLS A	CLS B	CLS	NA	PC
Periodic Timing	2	8	12	67%	7	5
COPMonitoring	3	4	12	33%	7	10
DataFreshness	2	3	12	25%	5	4

Fonte: Produção do próprio autor.

Nota: CLS A = Classes Afetadas. CLS B = Total de Classes. %CLS = % de Classes Afetadas. NA = Número de Adaptações. PC = Pointcuts.

Comparando o esforço de especificação dos aspectos sobre o projeto e o seu impacto sobre o código final, percebe-se a facilidade e agilidade que a definição dos aspectos em um nível mais abstrato traz para o projeto. Assim, a abordagem utilizada traz como benefícios: (i) a diminuição do tempo de projeto; (ii) aumento da reusabilidade e manutenabilidade do código com o uso dos aspectos; (iii) diminui os erros de desenvolvimento de implementação dos tratamentos dos requisitos não funcionais e (iv) o aperfeiçoamento do código VHDL, descrevendo um circuito eficiente em termos de consumo de área e desempenho.

6 CONCLUSÕES E TRABALHOS FUTUROS

6.1 CONCLUSÕES

A presente dissertação mostrou que é possível especificar sistemas embarcados implementados em FPGA usando uma linguagem de alto nível como a UML, que pode ser transformada automaticamente em elementos do código VHDL, gerando descrições de hardware sintetizáveis e funcionais. Modelos orientados a objetos foram completamente transformados em código VHDL a partir de regras de mapeamento. No entanto, é importante destacar que, como a especificação é independente de plataforma, o mesmo modelo pode gerar código para outras plataformas que utilizam outras linguagens como Java e C++. Assim, este trabalho mostra a compatibilidade entre modelos UML e entidades da linguagem alvo VHDL, além da possibilidade de especificação de sistemas embarcados implementados em FPGA a partir desses modelos em alto nível. Este trabalho apresentou o mapeamento para VHDL de elementos e relacionamentos complexos da orientação a objetos como herança, composição, dados multi-valorados, tipos enumerados, etc. Com isso, demonstrou-se a compatibilidade da especificação em alto nível em UML de sistemas embarcados implementados em FPGA por meio de uma abordagem orientada a objetos.

Os requisitos não funcionais foram especificados em UML/MARTE, utilizando o paradigma orientado a aspectos e permitindo a geração do código VHDL com o tratamento para os requisitos não funcionais. Três aspectos foram implementados: *PeriodicTiming*, *DataFreshness* e *COPMonitoring*. O DERAF foi estendido com a adição do pacote *FaultHandling* para o tratamento de falhas, no qual foi adicionado o aspecto *COPMonitoring* definido neste trabalho. Novos *join points* foram definidos para os aspectos *PeriodicTiming* e *DataFreshness* utilizando estereótipos do perfil MARTE, tornando esses aspectos mais portáveis para outras aplicações e plataformas. Além disso, um conjunto de *pointcuts* foi identificado para a linguagem VHDL que permite a inserção de adaptações em grande parte da estrutura dessa linguagem. Regras de mapeamento foram definidas para a implementação dos aspectos em VHDL, que permitiram a geração completa dos tratamentos para os requisitos não funcionais nos elementos afetados do modelo, sem a necessidade de manutenções manuais. Esses resultados demonstram a compatibilidade da linguagem VHDL com a abordagem orientada a aspectos. Contudo, algumas questões foram identificadas como o conflito entre aspectos e problemas ocasionados devido a definição de *join points* semelhantes entre diferentes aspectos. Estas questões foram tratadas por meio de modificações nas regras de mapeamento das adaptações.

Os resultados apresentados sugerem que o uso de aspectos para o tratamento dos requisitos não funcionais permite diminuir o tempo de desenvolvimento, diminuindo a complexidade do tratamento dos requisitos não funcionais, além de permitir sua definição de forma visual. A média de 42% de classes afetadas pelos aspectos modelados sugere alto impacto dos aspectos sobre o projeto. Além disso, é necessário levar em consideração o esforço de desenvolvimento dos JPDDs e *pointcuts* que definem a aplicação dos aspectos sobre o modelo, em relação ao esforço de desenvolvimento necessário para implementar os mesmos tratamentos para os requisitos não funcionais manualmente.

Requisitos não funcionais geralmente não estão associados com uma determinada funcionalidade do projeto. Além disso, tais requisitos estão presentes em todo o projeto de forma transversal. Paradigmas de desenvolvimento tradicionais, como a orientação a objetos, não permitem tratar esse comportamento de forma eficiente, dessa forma, a orientação a aspecto vem preencher essa lacuna.

Conforme mencionado na revisão da literatura, o uso de aspectos para o tratamento de requisitos não funcionais traz benefícios para o desenvolvimento de software como a reutilização de um código já testado e melhora na manutenção do tratamento de um requisito. Essa visão pode ser estendida para a descrição de hardware.

O presente trabalho mostrou que o uso de aspectos para o tratamento de requisitos não funcionais na descrição de hardware em VHDL é possível e traz benefícios para o projeto. Além do tratamento relacionado aos requisitos de projeto, traz benefícios ao desempenho e consumo de área da FPGA. A aplicação dos aspectos ocasionou aumento médio de 134% na quantidade de *slices* utilizadas da FPGA, que de fato representa um aumento na área da FPGA. Porém o aumento de FFs e LUTs utilizadas foi maior (161% e 167%), representando uma melhor utilização dos recursos da FPGA. Levando isso em consideração e a melhora no desempenho do circuito, pode-se considerar que há uma otimização no uso dos recursos da FPGA. O uso de aspectos permite uma melhor estruturação do circuito, que leva a um menor caminho crítico (média de 8% menor), possibilitando uma maior frequência de operação para o circuito implementado (média de $4,5\%$ de aumento no desempenho). Contudo, ressalta-se que, resultados semelhantes poderiam ser obtidos com a padronização do código VHDL e a partir da experiência dos engenheiros. Entretanto, tal abordagem deixaria de lado as vantagens de reutilização do código que trata os requisitos não funcionais, assim como a facilidade de manutenção que o uso dos aspectos permite. Além disso, podem ser feitas otimizações nas regras de mapeamento permitindo a geração de um código mas eficiente.

A abordagem proposta traz indícios da melhora no desempenho do projeto, quando comparado com a implementação dos requisitos não funcionais manualmente. Outra vantagem é a redução do tempo de desenvolvimento por possibilitar a reutilização do tratamento dos requisitos não funcionais, assim como permite melhorar a manutenção e diminuir erros de implementação devido a automatização do processo de codificação e a reutilização de uma especificação já testada e validada. Adicionalmente, a especificação do tratamento dos requisitos em um alto nível reduz a complexidade de desenvolvimento e permite utilizar elementos e parâmetros já definidos no modelo do projeto, agilizando o processo de desenvolvimento. Entretanto, conforme apresentado nas análises dos estudos de caso, a abordagem proposta trouxe como impacto para o projeto o aumento na área da FPGA.

Por fim, para medir o impacto do tratamento dos requisitos não funcionais e a abordagem proposta, foram utilizadas algumas métricas identificadas na revisão da literatura e outras identificadas neste trabalho. O uso dessas métricas na validação dos estudos de caso apresentados, permitiu mensurar de forma quantitativa o impacto da utilização dos aspectos, assim como toda a abordagem proposta. Algumas dessas métricas também foram utilizadas em trabalhos relacionados, o que permitiu comparar os resultados obtidos neste trabalho com outros trabalhos. Assim, este trabalho também traz como resultado o conjunto de métricas proposto para a avaliação dos requisitos não funcionais em sistemas embarcados implementados em FPGA.

O mercado de sistemas embarcados está cada vez mais competitivo, tornando imprescindível melhorar os processos de desenvolvimento para diminuir custos e tempo de fabricação. Este trabalho apresentou uma abordagem que pode ajudar nessa tarefa. Não se trata de uma solução final, mas um passo em direção ao desenvolvimento de metodologias e ferramentas para melhorar o projeto de sistemas embarcados, que foi aplicado no contexto da plataforma FPGA. A aplicação do paradigma orientado a aspectos no desenvolvimento de hardware com estudos de casos de sistemas reais, abre espaço para mais pesquisas que podem melhorar e avançar o estado-da-arte da engenharia de sistemas.

6.2 TRABALHOS FUTUROS

São apresentadas as seguintes propostas como sugestões de trabalhos futuros:

1. Em relação as ferramentas de desenvolvimento:

    - Os estereótipos atribuídos aos elementos do modelo UML não são armazenados pelo DERCS. Estas informações são úteis para a ge-

ração do código para a linguagem alvo, se forem acessíveis pelas regras de mapeamento. A proposta é estender o DERCS para armazenar os estereótipos e seus atributos, aplicados aos elementos do modelo UML;

- Mensagens ASSIGN, assim como expressões de elementos de seleção (*if,else*), são tratadas como elementos textuais pelo DERCS, não possibilitando definições mais complexas como, por exemplo, a conexão com outro elemento em uma expressão. Essa limitação prejudica a geração do código, que poderia ser melhor adaptado para a linguagem alvo ao eliminar o uso de elementos da linguagem alvo no modelo para a definição das mensagens ASSIGN e expressões. Assim, propõe-se o desenvolvimento e/ou utilização de uma linguagem de especificação de expressões para serem usadas em mensagens ASSIGN e expressões em elementos de seleção. Isso permitirá uma definição independente da linguagem alvo para estes elementos, em um alto nível no modelo;
- Atualmente a ferramenta GenERTiCA não possui suporte para a transformação do diagrama de estados. Propõe-se desenvolver o tratamento para diagrama de estados, pois será muito útil no desenvolvimento de sistemas embarcados, principalmente os implementados em FPGA;
- Desenvolver suporte para a leitura de arquivos XMI para importação dos elementos UML, tornando a ferramenta GenERTiCA independente da ferramenta Magic Draw.

2. Em relação as regras de mapeamento e implementação dos aspectos em VHDL:

   - Desenvolvimento de regras de mapeamento de um seletor para o tratamento de atribuição de valores a sinais em mais de um processo;
   - Estender o DERAF de modo a incluir o tratamento para os demais requisitos não funcionais identificados para o projeto de sistemas embarcados em Wehrmeister (2009);
   - Implementar os *pointcuts* não abordados neste trabalho, indicados na Tabela 5 apresentado na Seção 4.3.2.1.

REFERÊNCIAS

ANNE, M. et al. Think: View-Based Support of Non-functional Properties in Embedded Systems. In: *Embedded Software and Systems, 2009. ICESS '09. International Conference on*. [S.l.: s.n.], 2009. p. 147 –156.

BAINBRIDGE-SMITH, A.; PARK, S.-H. ADH: an aspect described hardware programming language. In: *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*. [S.l.: s.n.], 2005. p. 283 – 284.

BELTRAN, M.; GUZMáN, A.; SEVILLANO, F. High level performance metrics for fpga-based multiprocessor systems. *Performance Evaluation*, v. 67, n. 6, p. 417 – 431, 2010. ISSN 0166-5316.

BERTAGNOLLI, S. d. C. *FRIDA: um método para elicitação e modelagem de RNFs*. Dissertação (These) — Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, Brasil, 2004. Disponível em: http://hdl.handle.net/10183/3618, Acessado em: 20/05/2013.

CANCILA, D. et al. Toward Correctness in the Specification and Handling of Non-Functional Attributes of High-Integrity Real-Time Embedded Systems. *Industrial Informatics, IEEE Transactions on*, v. 6, n. 2, p. 181 –194, may 2010. ISSN 1551-3203.

CARDOSO, J. a. M. et al. LARA: an aspect-oriented programming language for embedded systems. In: *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. New York, NY, USA: ACM, 2012. (AOSD '12), p. 179–190. ISBN 978-1-4503-1092-5. Disponível em: <http://doi.acm.org/10.1145/2162049.2162071>.

CARRO, L.; WAGNER, F. Sistemas Computacionais Embarcados. In: *Jornadas de Atualização em Informática. In: XXII JAI 2003. (Org.)*. [S.l.: s.n.], 2003. v. 1, p. 45 –94.

CHARAABI, L.; MONMASSON, E.; SLAMA-BELKHODJA, I. Presentation of an efficient design methodology for FPGA implementation of control systems. Application to the design of an antiwindup PI controller. In: *IECON 02 [Industrial Electronics Society, IEEE 2002 28th Annual Conference of the]*. [S.l.: s.n.], 2002. v. 3, p. 1942 – 1947 vol.3.

CORREA, U. B. et al. Towards estimating physical properties of embedded systems using software quality metrics. In: *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*.

Washington, DC, USA: IEEE Computer Society, 2010. (CIT '10), p. 2381–2386. ISBN 978-0-7695-4108-2.

COUTINHO, J. G. et al. Experiments with the LARA aspect-oriented approach. In: *Proceedings of the 11th annual international conference on Aspect-oriented Software Development Companion*. New York, NY, USA: ACM, 2012. (AOSD Companion '12), p. 27–30. ISBN 978-1-4503-1222-6. Disponível em: <http://doi.acm.org/10.1145/2162110.2162129>.

DEITEL, P.; DEITEL, H. *Java: como programar*. 8. ed. São Paulo, SP, Brasil: Pearson Education do Brasil, 2010. ISBN 978-85-7605-563-1.

DENG, G.; SCHMIDT, D. C.; GOKHALE, A. Addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems via aspect-oriented & model-driven software development. In: *Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006. (ICSE '06), p. 811–814. ISBN 1-59593-375-1. Disponível em: <http://doi.acm.org/10.1145/1134285.1134421>.

DRIVER, C. et al. Managing Embedded Systems Complexity with Aspect-Oriented Model-Driven Engineering. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 10, n. 2, p. 21:1–21:26, jan. 2010. ISSN 1539-9087. Disponível em: <http://doi.acm.org/10.1145/1880050.1880057>.

EBEID, E.; QUAGLIA, D.; FUMMI, F. Generation of vhdl code from uml/marte sequence diagrams for verification and synthesis. In: *Digital System Design (DSD), 2012 15th Euromicro Conference on*. [S.l.: s.n.], 2012. p. 708–714.

ELHAJI, M. et al. System level modeling methodology of noc design from uml-marte to vhdl. *Design Automation for Embedded Systems*, Springer US, v. 16, n. 4, p. 161–187, 2012. ISSN 0929-5585. Disponível em: <http://dx.doi.org/10.1007/s10617-012-9101-2>.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-oriented programming: Introduction. *Commun. ACM*, ACM, New York, NY, USA, v. 44, n. 10, p. 29–32, out. 2001. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/383845.383853>.

ENGEL, M.; SPINCZYK, O. Aspects in hardware: what do they look like? In: *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*. New York, NY, USA: ACM, 2008. (ACP4IS '08), p. 5:1–5:6. ISBN 978-1-60558-142-2. Disponível em: <http://doi.acm.org/10.1145/1404891.1404896>.

FARINES, J.-M.; FRAGA, J. d. S.; OLIVEIRA, R. S. d. *Sistemas de Tempo Real*. Florianópolis, SC, BR: [s.n.], 2000. Disponível em: http://www.academia.edu/3178837/Sistemas_de_Tempo_Real. Acessado em: 06/06/2013.

FIGUEIREDO, E. et al. On the maintainability of aspect-oriented software: A concern-oriented measurement framework. In: *12th European Conference on Software Maintenance and Reengineering*. [S.l.: s.n.], 2008. p. 183–192. ISSN 1534-5351.

FREITAS, E. P. d. *Metodologia Orientada a Aspectos para a Especificação de Sistemas Tempo-Real Embarcados Distribuídos*. Dissertação (Master of Computing Science) — Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, Brasil, 2007. Disponível em: http://hdl.handle.net/10183/10268, Acessado em: 05/02/2013.

GAJSKI, D.; VAHID, F. Specification and design of embedded hardware-software systems. *Design Test of Computers, IEEE*, v. 12, n. 1, p. 53 – 67, spring 1995. ISSN 0740-7475.

GHODRAT, M.; LAHIRI, K.; RAGHUNATHAN, A. Accelerating System-on-Chip Power Analysis Using Hybrid Power Estimation. In: *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*. [S.l.: s.n.], 2007. p. 883 –886. ISSN 0738-100X.

GHOLAMIPOUR, A. et al. Area, reconfiguration delay and reliability trade-offs in designing reliable multi-mode FIR filters. In: *Design and Test Workshop (IDT), 2011 IEEE 6th International*. [S.l.: s.n.], 2011. p. 82 –87. ISSN 2162-0601.

GOKHALE, A.; BALASUBRAMANIAN, K.; LU, T. CoSMIC: addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems. In: *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2004. (OOPSLA '04), p. 218–219. ISBN 1-58113-833-4. Disponível em: <http://doi.acm.org/10.1145/1028664.1028758>.

GRANDPIERRE, T.; LAVARENNE, C.; SOREL, Y. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In: *Proceedings of the seventh international workshop on Hardware/software codesign*. New York, NY, USA: ACM, 1999. (CODES '99), p. 74–78. ISBN 1-58113-132-1. Disponível em: <http://doi.acm.org/10.1145/301177.301489>.

HILL, M. D. What is scalability? *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 18, n. 4, p. 18–21, dez. 1990. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/121973.121975>.

HUFFMIRE, T. et al. Managing Security in FPGA-Based Embedded Systems. *Design Test of Computers, IEEE*, v. 25, n. 6, p. 590 –598, nov.-dec. 2008. ISSN 0740-7475.

IEEE. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, p. c1–626, 2009.

KICZALES, G. et al. Aspect-oriented programming. In: AKşIT, M.; MATSUOKA, S. (Ed.). *ECOOP'97 — Object-Oriented Programming*. Springer Berlin Heidelberg, 1997, (Lecture Notes in Computer Science, v. 1241). p. 220–242. ISBN 978-3-540-63089-0. Disponível em: <http://dx.doi.org/10.1007/BFb0053381>.

KITCHENHAM, B. et al. Systematic literature reviews in software engineering – A systematic literature review. *Information and Software Technology*, v. 51, n. 1, p. 7 – 15, 2009. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584908001390>.

LINEHAN, E.; CLARKE, S. An aspect-oriented, model-driven approach to functional hardware verification. *Journal of Systems Architecture*, v. 58, n. 5, p. 195 – 208, 2012. ISSN 1383-7621. <ce:title>Model Based Engineering for Embedded Systems Design</ce:title>. Disponível em: <http://www.sciencedirect.com/science/article/pii/S138376211100018X>.

LIU, Q. et al. Data-reuse exploration under an on-chip memory constraint for low-power FPGA-based systems. *IET Computers & Digital Techniques*, IET, v. 3, n. 3, p. 235–246, 2009. Disponível em: <http://link.aip.org/link/?CDT/3/235/1>.

LOPES, C. V. *D: A Language Framework for Distributed Program- ming*. Tese (Doutorado) — College of Computer Science, Northeastern University, 1997.

MALINOWSKI, A.; YU, H. Comparison of Embedded System Design for Industrial Applications. *Industrial Informatics, IEEE Transactions on*, v. 7, n. 2, p. 244 – 254, 2011. ISSN 1551-3203.

MAXFIELD, C. *The Design Warrior's Guide to FPGAs*. Orlando, FL, USA: Academic Press, Inc., 2004. ISBN 0750676043.

MEI, B. et al. Design and optimization of dynamically reconfigurable embedded systems. In: PLAKS, T.; ATHANAS, P. (Ed.). 115 AVALON DR, ATHENS, GA 30606 USA: C S R E A PRESS, 2001. p. 78–84. ISBN 1-892512-76-9. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2001), LAS VEGAS, NV, JUN 25-28, 2001.

MEIER, M.; HANENBERG, S.; SPINCZYK, O. AspectVHDL Stage 1: The prototype of an aspect-oriented hardware description language. In: *Proceedings of the 2012 workshop on Modularity in Systems Software*. New York, NY, USA: ACM, 2012. (MISS '12), p. 3–8. ISBN 978-1-4503-1217-2. Disponível em: <http://doi.acm.org/10.1145/2162024.2162028>.

MEYER-BAESE, U. et al. Energy optimization of Application-Specific Instruction-Set Processors by using hardware accelerators in semicustom ICs technology. *Microprocessors and Microsystems*, v. 36, n. 2, p. 127 – 137, 2012. ISSN 0141-9331. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0141933111000780>.

MONMASSON, E.; CIRSTEA, M. FPGA Design Methodology for Industrial Control Systems - A Review. *Industrial Electronics, IEEE Transactions on*, v. 54, n. 4, p. 1824 –1842, aug. 2007. ISSN 0278-0046.

MONMASSON, E. et al. FPGAs in Industrial Control Applications. *Industrial Informatics, IEEE Transactions on*, v. 7, n. 2, p. 224 –243, may 2011. ISSN 1551-3203.

MOREIRA, T. G. *Geração Automática de Código VHDL a partir de Modelos UML para Sistemas Embarcados de Tempo-Real*. Dissertação (Master of Computing Science) — Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, Brasil, 2012. Disponível em: http://hdl.handle.net/10183/55444, Acessado em: 25/11/2012.

MUCK, T. et al. A Case Study of AOP and OOP Applied to Digital Hardware Design. In: *Computing System Engineering (SBESC), 2011 Brazilian Symposium on*. [S.l.: s.n.], 2011. p. 66 –71.

PAPADIMITRIOU, K.; DOLLAS, A.; HAUCK, S. Performance of partial reconfiguration in FPGA systems: A survey and a cost model. *ACM Trans. Reconfigurable Technol. Syst.*, ACM, New York, NY, USA, v. 4, n. 4, p. 36:1–36:24, dez. 2011. ISSN 1936-7406. Disponível em: <http://doi.acm.org/10.1145/2068716.2068722>.

PARK, S.-H. *ADH, Aspect Described Hardware-Description-Language*. Dissertação (Master of Engineering) — University of Canterbury. Electrical and Computer Engineering, Private Bag 4800, Christchurch, New Zealand, 2006. Disponível em: http://hdl.handle.net/10092/1113, Acessado em: 25/11/2012.

PEDRONI, V. A. *Circuit Design with VHDL*. Cambridge, MA, USA: MIT Press, 2004. ISBN 0262162245.

PETROV, Z. et al. Programming safety requirements in the REFLECT design flow. In: *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*. [S.l.: s.n.], 2011. p. 841 – 847.

QIU, W.; ZHANG, L.-C. Application of Model Driven Architecture to Development Real-Time System Based on Aspect-Oriented. In: LIU, B.; CHAI, C. (Ed.). *Information Computing and Applications*. Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 7030). p. 569–576. ISBN 978-3-642-25254-9. Disponível em: <http://dx.doi.org/10.1007/978-3-642-25255-6_72>.

QUADRI, I.; MEFTALI, S.; DEKEYSER, J.-L. MARTE based modeling approach for Partial Dynamic Reconfigurable FPGAs. In: *Embedded Systems for Real-Time Multimedia, 2008. ESTImedia 2008. IEEE/ACM/IFIP Workshop on*. [S.l.: s.n.], 2008. p. 47 –52. ISBN 978-1-4244-2612-6.

QUADRI, I.; MEFTALI, S.; DEKEYSER, J.-L. Designing dynamically reconfigurable SoCs: From UML MARTE models to automatic code generation. In: *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*. [S.l.: s.n.], 2010. p. 68 –75.

QUADRI, I. R. et al. Expressing embedded systems configurations at high abstraction levels with UML MARTE profile: Advantages, limitations and alternatives. *Journal of Systems Architecture*, v. 58, n. 5, p. 178 – 194, 2012. ISSN 1383-7621. Model Based Engineering for Embedded Systems Design. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1383762112000021>.

QUADRI, I. R. et al. MARTE based design flow for Partially Reconfigurable Systems-on-Chips. In: *17th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC 09)*. Florianópolis, Brasil: [s.n.], 2009. Disponível em: <http://hal.inria.fr/inria-00486846>.

RAMACHANDRAN, L. et al. Synthesis of functions and procedures in behavioral vhdl. In: *Design Automation Conference, 1993, with*

EURO-VHDL '93. Proceedings EURO-DAC '93., European. [S.l.: s.n.], 1993. p. 560–565.

RASHID, A. et al. Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe. *Computer*, v. 43, n. 2, p. 19–26, 2010. ISSN 0018-9162.

REDIN, R. et al. On the Use of Software Quality Metrics to Improve Physical Properties of Embedded Systems. In: KLEINJOHANN, B.; WOLF, W.; KLEINJOHANN, L. (Ed.). *Distributed Embedded Systems: Design, Middleware and Resources*. [S.l.]: Springer Boston, 2008, (IFIP International Federation for Information Processing, v. 271). p. 101–110. ISBN 978-0-387-09660-5.

ROTH, C. H. *Digital System Design Using VHDL*. [S.l.]: PWS Publishing Company, 1998.

SALEWSKI, F.; TAYLOR, A. Systematic considerations for the application of FPGAs in industrial applications. In: *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*. [S.l.: s.n.], 2008. p. 2009 –2015.

SELIC, B. The pragmatics of model-driven development. *Software, IEEE*, v. 20, n. 5, p. 19–25, 2003. ISSN 0740-7459.

SHIMIZU, S. et al. On-demand design service innovations. *IBM Journal of Research and Development*, IBM, v. 48, n. 5.6, p. 751 –765, sep. 2004. ISSN 0018-8646.

SOMMERVILLE, I. *Engenharia Software*. São Paulo, SP, BR: Pearson, 2007. ISBN 9788588639287.

SOREL, Y. Massively parallel computing systems with real time constraints:"Algorithm Architecture Adequation"methodology. In: *Massively Parallel Computing Systems, 1994., Proceedings of the First International Conference on*. [S.l.: s.n.], 1994. p. 44 –53.

STEIN, D.; HANENBERG, S.; UNLAND, R. Expressing different conceptual models of join point selections in aspect-oriented design. In: *Proceedings of the 5th International Conference on Aspect-oriented Software Development*. New York, NY, USA: ACM, 2006. (AOSD '06), p. 15–26. ISBN 1-59593-300-X. Disponível em: <http://doi.acm.org/10.1145/1119655.1119661>.

TAHOORI, M. et al. Obtaining FPGA soft error rate in high performance information systems. *Microelectronics Reliability*, v. 49, n. 5, p. 551 – 557, 2009. ISSN 0026-2714. 2008 Reliability

of Compound Semiconductors (ROCS) Workshop. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0026271409000778>.

TANENBAUM, A. S.; RENESSE, R. V. Distributed operating systems. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 17, n. 4, p. 419–470, dez. 1985. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/6041.6074>.

TSAI, J. J. P. et al. *Distributed real-time systems: monitoring, visualization, debugging, and analysis*. New York, NY, USA: John Wiley & Sons, Inc., 1996. ISBN 0-471-16007-5.

WANDERLEY, E. et al. Security fpga analysis. In: BADRIGNANS, B. et al. (Ed.). *Security Trends for FPGAS*. Springer Netherlands, 2011. p. 7–46. ISBN 978-94-007-1337-6. Disponível em: <http://dx.doi.org/10.1007/978-94-007-1338-3_2>.

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

WEHRMEISTER, M. A. *An aspect-oriented model-driven engineering approach for distributed embedded real-time systems*. Tese (Doctor of Computing Science) — Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, Brasil, 2009. Disponível em: http://hdl.handle.net/10183/17792, Acessado em: 02/03/2012.

WEHRMEISTER, M. A.; PEREIRA, C. E.; RAMMIG, F. Aspect-oriented model-driven engineering for embedded systems applied to automation systems. *IEEE Transactions on Industrial Informatics.*, v. 9, n. 4, p. 2373–2386, Nov 2013. ISSN 1551-3203. Doi:10.1109/TII.2013.2240308.

WOLF, W. *FPGA-Based System Design*. 4th. ed. New Jersey, USA: Pearson Education, 2004. ISBN 0-13-142461-0.

WOLF, W. *Computers as Components: Principles os Embedded Computing Systems Design*. Burlington, MA, USA: Morgan Kaufmann, 2008. ISBN 9780123743978.

XILINX. *Xilinx User Community Forums*. 2014. Disponível em: <http://forums.xilinx.com>. Acesso em: 15 de Jan de 2014.

Apêndices

APÊNDICE A – METODOLOGIA DA PESQUISA BIBLIOGRÁFICA

Para o levantamento das tendências e avanços no tratamento de requisitos não funcionais em sistemas embarcados implementados em FPGAs, assim como o entendimento dos principais desafios encontrados nos trabalhos científicos, foi realizada uma pesquisa bibliográfica seguindo a metodologia descrita nesta seção. A metodologia é dividida em duas etapas: elaboração da frase de busca e realização da pesquisa por mecanismos de busca; e catalogação e priorização de leitura dos artigos selecionados.

A.1 FRASE DE BUSCA E PESQUISA

A definição da frase de busca não é uma tarefa simples, mesmo contando com ferramentas de busca especialistas em pesquisa científica. A quantidade de resultados obtidos em uma pesquisa torna muitas vezes inviável uma análise aprofundada dos trabalhos fortemente relacionados com o tema da busca. Dessa forma, a frase de busca foi elaborada seguindo alguns passos apresentados por Kitchenham et al. (2009) que estabeleceram um protocolo para a definição dos termos e critérios de busca.

O primeiro passo foi definir as questões que se desejam responder com a pesquisa. São elas:

1. Como os requisitos não funcionais estão sendo tratados em sistemas embarcados implementados em FPGAs?
    a) Quais são as metodologias e abordagens utilizadas?
    b) Quais as ferramentas utilizadas?
    c) Quais os principais desafios?

Considerando as questões propostas, inicialmente foram assumidas algumas restrições afim de delimitar o escopo da pesquisa. A primeira restrição foi em relação ao período pesquisado. O objetivo deste trabalho é obter uma visão da última década de pesquisas científicas em relação às FPGAs. Contudo, para também resgatar alguns conceitos e primeiros esforços direcionados ao tratamento de requisitos não funcionais em projetos com FPGAs, buscou-se publicações a partir de 1990, quando, de fato, as FPGAs começaram a ser utilizadas em aplicativos comerciais (MAXFIELD, 2004). A segunda limitação foi em relação ao tipo de publicação. Como o intuito é obter uma visão do estado-da-arte, optou-se por considerar apenas artigos publicados em periódicos e anais de congressos, pois eles dão mais ênfase às tendências e novas tecnologias.

Baseado nas questões 1.(a),(b) e (c) assumiu-se como critérios de inclusão os termos: *"Embedded System"*, *"Cyber Physical"*, *"Software Engineering"*, *"Electronic Design Automation"*, *"EDA"*, *"Design"*, *"Modeling"*, *"Model Driven"*, utilizando o operador *ou* entre os termos. Estes termos devem aparecer no resumo, palavras chaves ou texto dos artigos.

A questão 1 definiu os termos chave da busca, sendo compostos de termos que selecionam a tecnologia investigada, que, no caso, trata-se das FPGAs e do tratamento dos requisitos não funcionais. Estes termos foram selecionados do título das publicações. Então, foram utilizados dois conjuntos de frases em intersecção, formadas pelos termos *"Field-programmable gate array"* ou *FPGA* para a seleção da tecnologia, e os termos *"Non-functional"* ou *Safety* ou *Robustness* ou *Stress* ou *Security* ou *Efficiency* ou **ability*[1] ou *Performance* ou *"Crosscutting Concern"* para a seleção do problema abordado. O termo **ability* foi utilizado com base em (KITCHENHAM et al., 2009), para selecionar trabalhos que abordam a qualidade dos projetos de sistemas embarcados.

Para a realização da pesquisa foram utilizados os mecanismos de busca ACM Digital Library, IEEEXplorer, ISI Web of Knowledge e Scopus. Estes mecanismos de busca foram selecionados pois realizam a pesquisa em bases de dados específicas de trabalhos científicos na área de tecnologia, além de possuírem ferramentas que facilitam a aplicação de filtros. A ferramenta Periódicos da CAPES não foi utilizada por não possuir uma interface com os filtros necessários para a realização busca.

A.2 CATALOGAÇÃO E PRIORIZAÇÃO DE LEITURA

Durante a execução da pesquisa foram catalogados 131 artigos. Nesta seleção inicial realizou-se uma análise do título e do resumo dos artigos para identificar o objetivo principal de cada um. O intuito foi selecionar os artigos que tem as FPGAs como tema central e não como um meio. Como resultado da análise foram selecionados 71 artigos dos 131.

Por tratar-se de uma quantidade considerável de artigos para leitura e considerando o tempo disponível para a realização deste trabalho, foi necessário restringir o levantamento bibliográfico a 30 artigos, selecionados destes 71. Para isso, estipularam-se critérios de seleção e priorização baseados nos conceitos abordados nos artigos e temas centrais. Na Tabela 20 são apresentados os critérios de seleção que foram considerados quanto aos conceitos e aos requisitos não funcionais abordados nos artigos. Cada critério recebeu um peso (coluna "Peso"), atribuído com base no relacionamento do critério com

[1] "*" é um curinga na pesquisa para que o mecanismo de busca não filtre o prefixo do termo *ability*.

o foco do levantamento. Os pesos são normalizados, assim o a soma dos pesos de cada grupo de critérios (conceitos e requisitos não funcionais) corresponde a 1, ou seja, estes pesos refletem o percentual de importância do critério para o trabalho. Para a definição dos pesos também levou-se em consideração a frequência do termo nos artigos. Assim, termos muito comuns receberam um peso menor por não terem muita influência para a diferenciação dos artigos como, por exemplo, o termo *"Desempenho"*, presente na maior parte dos artigos, não serve como um bom critério de seleção por não representar diferença entre os artigos. Após, foi atribuída a soma dos pesos dos critérios ao artigo que os possuía, definindo a sua prioridade. Foram então ordenados e selecionados os 30 artigos com maior pontuação.

Tabela 20 – Critérios de seleção e priorização dos artigos

	Critério	**Peso**
Conceitos	FGPA	0.35
	Desenvolvimento Guiado por Modelos	0.20
	Requisitos não funcionais (requisito não funcional)	0.15
	Abordagem Baseada em Componentes	0.10
	Modelagem	0.10
	ASIC	0.05
	Ferramenta de Desenvolvimento	0.05
Requisitos Não Funcionais	Energia	0.20
	Segurança	0.20
	Custo	0.10
	Paralelismo	0.10
	Processamento Distribuído	0.10
	Tempo de Projeto	0.10
	Desempenho	0.05
	Memória	0.05
	Qualidade de Projeto	0.05
	Reusabilidade	0.05

Dada esta parametrização, os artigos catalogados foram priorizados de acordo com sua pontuação obtida pela soma dos pesos dos critérios que cada artigo possuía. Na Tabela 21 são apresentados os 30 artigos selecionados para o estudo do problema abordado, com sua classificação (C) e pontuação (P). A leitura e análise dos artigos foram realizadas em ordem ascendente de ano de sua publicação.

Tabela 21 – Artigos selecionados para a pesquisa bibliográfica

C	Título - Autores (ano)	P
1	Quadri et al. (2012)	1.55
2	Cardoso et al. (2012)	1.55
3	Coutinho et al. (2012)	1.55
4	Shimizu et al. (2004)	1.50
5	Monmasson et al. (2011)	1.45
6	Salewski e Taylor (2008)	1.40
7	Deng et al. (2006)	1.35
8	Quadri et al. (2008)	1.35
9	Monmasson et al. (2007)	1.30
10	Huffmire et al. (2008)	1.30
11	Wang et al. (2008)	1.20
12	Tahoori et al. (2009)	1.20
13	Gajski et al. (1995)	1.15
14	Petrov et al. (2011)	1.15
15	Meyer-Baese et al. (2011)	1.15
16	Malinowski e Yu (2011)	1.15
17	Mei et al. (2001)	1.10
18	Ghodrat et al. (2007)	1.10
19	Kharchenko et al. (2009)	1.10
20	Driver et al. (2010)	1.10
21	Liu et al. (2008)	1.05
22	Papadimitriou et al. (2011)	1.00
23	Cancila et al. (2010)	0.95
24	Wanderley et al. (2011)	0.95
25	Gholamipour et al. (2011)	0.90
26	Anne et al. (2009)	0.65
27	Qiu e Zhang (2011)	0.65
28	Meier et al. (2012)	0.65
29	Gokhale et al. (2004)	0.60
30	Engel e Spinczyk (2008)	0.55

A.3 DISCUSSÃO

Devido a amplitude e abrangência dos mecanismos de busca atuais, há uma grande quantidade de trabalhos científicos disponíveis. Porém, o que permite enriquecer rapidamente a pesquisa e o conhecimento, esbarra na dificuldade da seleção e mineração dos trabalhos relacionados com a pesquisa desejada e com alto impacto sobre esta. Por isso, é importante a utilização de

um processo sistemático para lidar de forma adequada e eficiente com todo o universo de trabalhos que podem contribuir de forma significativa com a pesquisa pretendida. Nesta seção, foi apresentada a metodologia seguida para o levantamento e seleção dos artigos analisados na pesquisa bibliográfica realizada.

Com os termos utilizados na pesquisa e mecanismos de busca, obteve-se uma grande quantidade de trabalhos. Entretanto, durante a leitura e análise destes trabalhos, identificaram-se outros trabalhos interessantes para a pesquisa, referenciados pelos artigos selecionados. Isso levou a adição de novos trabalhos. Todavia foi excluído um dos trabalhos da pesquisa que, após análise, foi identificado como um capítulo de livro e não de um artigo, conforme definido no escopo inicial da pesquisa. Na Tabela 22 são apresentados os artigos excluídos (E) e incluídos (I). Os artigos incluídos auxiliaram no embasamento e entendimento das propostas apresentadas nos artigos selecionados na metodologia, como (SOREL, 1994; GRANDPIERRE; LAVARENNE; SOREL, 1999; CHARAABI; MONMASSON; SLAMA-BELKHODJA, 2002). Outros acrescentaram propostas aos trabalhos relacionados e análise do estado-da-arte do tema, como (MUCK et al., 2011; QUADRI et al., 2009; QUADRI; MEFTALI; DEKEYSER, 2010; BELTRAN; GUZMáN; SEVILLANO, 2010).

Tabela 22 – Artigos incluídos e excluídos na pesquisa

Título - Autores (Ano)	**I/E**
Wanderley et al. (2011)	E
Muck et al. (2011)	I
Bainbridge-Smith e Park (2005)	I
Quadri et al. (2009)	I
Quadri et al. (2010)	I
Sorel (1994)	I
Grandpierre et al. (1999)	I
Charaabi et al. (2002)	I
Beltrán et al. (2010)	I

Após a leitura e análise dos artigos, foram atualizados os critérios utilizados na metodologia apresentada, dado o maior entendimento sobre os trabalhos adquirido com a sua leitura. Com isso, observou-se uma variação em média de -27.10% na pontuação final dos artigos, considerando um desvio padrão de 24.18%. Isso indica que em geral os artigos foram supervalorizados em relação ao tema pesquisado e, que, após a leitura dos artigos, verificou-se que em geral abordavam menos tópicos do que o esperado. Na

Tabela 23 é apresentada a re-priorização dos artigos após a leitura e análise dos trabalhos, com a variação (Var) da pontuação inicial (Ini) para a final (Fin). Nesta tabela, os artigos foram ordenados pela pontuação final, considerando apenas os artigos selecionados no processo inicial.

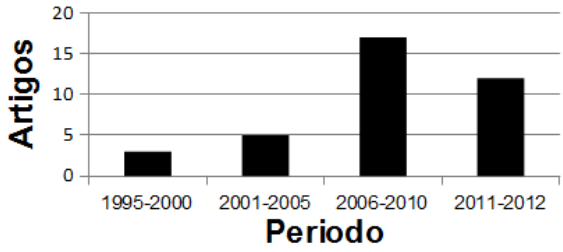
Tabela 23 – Comparação da priorização dos artigos inicial e após a análise

C	Título - Autores (Ano)	Ini	Fin	Var
1	Shimizu et al. (2004)	1.50	1.50	0.00%
2	Gajski et al. (1995)	1.15	1.30	13.04%
3	Mei et al. (2001)	1.10	1.15	4.55%
4	Petrov et al. (2011)	1.15	1.05	-8.70%
5	Wang et al. (2008)	1.20	1.00	-16.67%
6	Salewski e Taylor (2008)	1.40	0.95	-32.14%
7	Kharchenko et al. (2009)	1.10	0.95	-13.64%
8	Deng et al. (2006)	1.35	0.90	-33.33%
9	Monmasson et al. (2007)	1.30	0.90	-30.77%
10	Quadri et al. (2012)	1.55	0.90	-41.94%
11	Gokhale et al. (2004)	0.60	0.80	33.33%
12	Quadri et al. (2008)	1.35	0.80	-40.74%
13	Driver et al. (2010)	1.10	0.80	-27.27%
14	Liu et al. (2008)	1.05	0.75	-28.57%
15	Tahoori et al. (2009)	1.20	0.75	-37.50%
16	Anne et al. (2009)	0.65	0.75	15.38%
17	Monmasson et al. (2011)	1.45	0.75	-48.28%
18	Huffmire et al. (2008)	1.30	0.70	-46.15%
19	Ghodrat et al. (2007)	1.10	0.60	-45.45%
20	Gholamipour et al. (2011)	0.90	0.60	-33.33%
21	Cardoso et al. (2012)	1.55	0.60	-61.29%
22	Cancila et al. (2010)	0.95	0.60	-36.84%
23	Coutinho et al. (2012)	1.55	0.55	-64.52%
24	Meyer-Baese et al. (2011)	1.15	0.55	-52.17%
25	Meier et al. (2012)	0.65	0.55	-15.38%
26	Qiu e Zhang (2011)	0.65	0.55	-15.38%
27	Papadimitriou et al. (2011)	1.00	0.50	-50.00%
28	Engel e Spinczyk (2008)	0.55	0.45	-18.18%
29	Malinowski e Yu (2011)	1.15	0.40	-65.22%

Foram extraídos alguns dados dos artigos selecionados para a geração de alguns indicadores sobre o estado-da-arte do tema, o que também ajudou na análise da literatura (Figuras 36, 37 e 38). Na Figura 36 é apre-

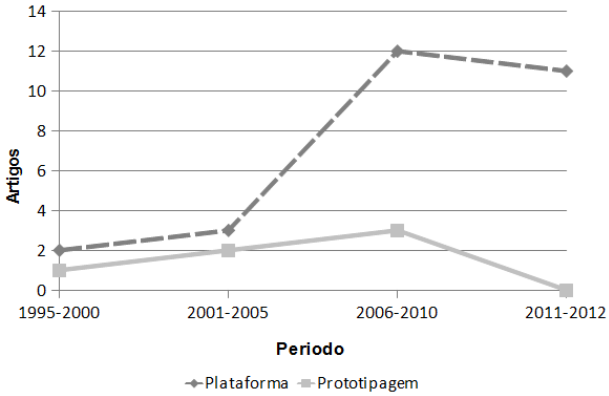
sentado o número de artigos avaliados por período e na Figura 37 a forma de utilização das FPGAs. Pode-se observar que as FPGAs passaram a ser vistas como plataforma final de desenvolvimento nos estudos mais recentes.

Figura 36 – Análise do número de artigos.



Fonte: Produção do próprio autor.

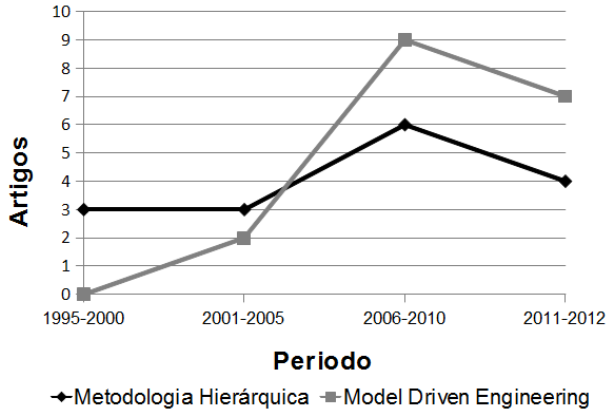
Figura 37 – Análise da utilização das FPGAs como plataforma.



Fonte: Produção do próprio autor.

A Figura 38 apresenta as abordagens de desenvolvimento mais utilizadas. Observou-se que inicialmente as abordagens propostas seguiam um modelo de desenvolvimento em cascata, desenvolvendo o software primeiro depois o hardware. Mas a partir de 2005 a atenção dos pesquisadores voltou-se para técnicas como a MDE e que estudos mais recentes estão focando no paradigma orientado a aspectos.

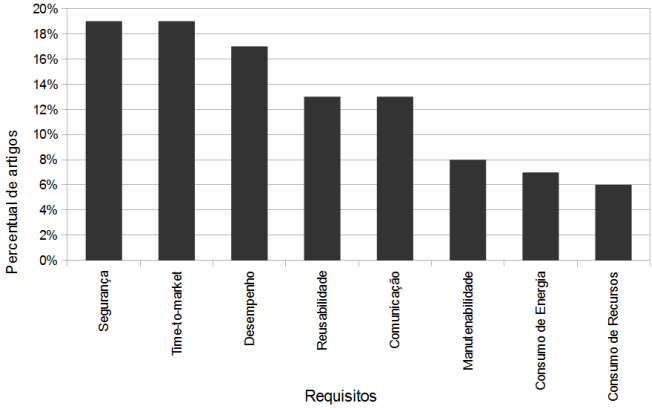
Figura 38 – Análise das abordagens utilizadas em relação ao período de publicação dos artigos.



Fonte: Produção do próprio autor.

A Figura 39 apresenta um resumo dos requisitos não funcionais abordados em projetos implementados em FPGA nos artigos analisados. Observa-se que *Desempenho*, embora continue sendo um assunto constante, deixou de ser o foco nos últimos anos, o que pode ser resultado do avanço tecnológico das FPGAs. Os requisitos de *Confiabilidade* e *Produtividade* vem ganhando atenção nos últimos anos.

Figura 39 – Análise dos requisitos não funcionais abordados em relação ao período de publicação dos artigos.

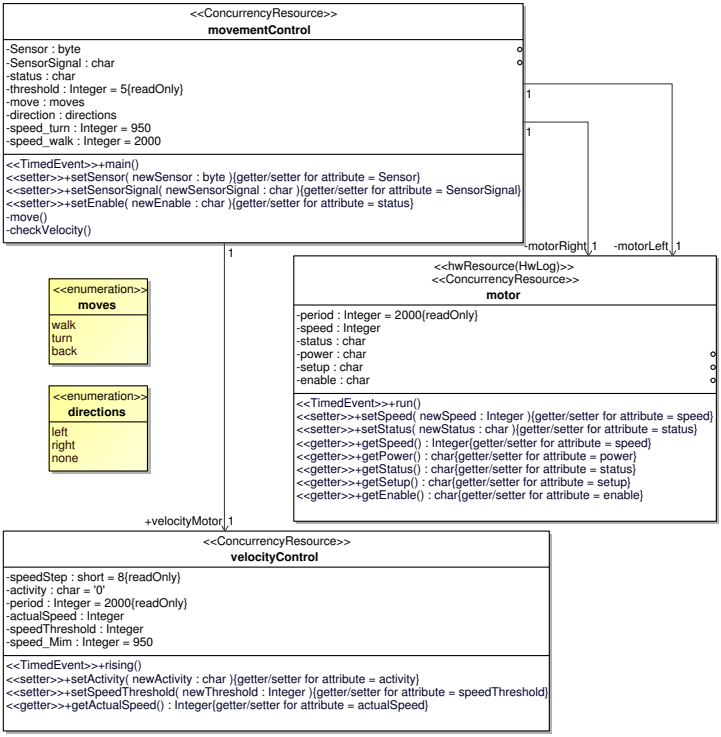


Fonte: Produção do próprio autor.

APÊNDICE B – DIAGRAMAS DOS ESTUDOS DE CASO

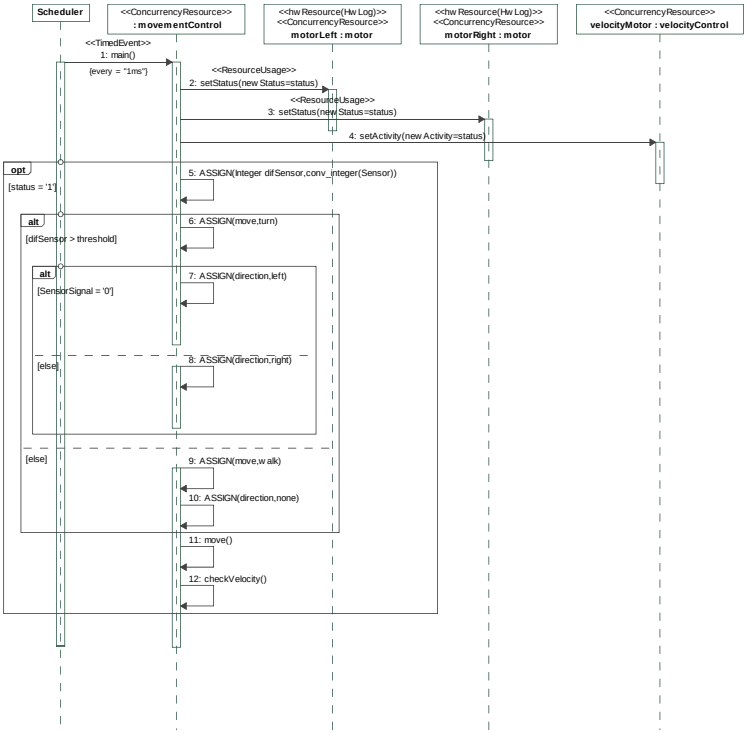
B.1 CONTROLE DE ROBÔ AUTÔNOMO

Figura 40 – Diagrama de classes do Controle de Robô Autônomo.



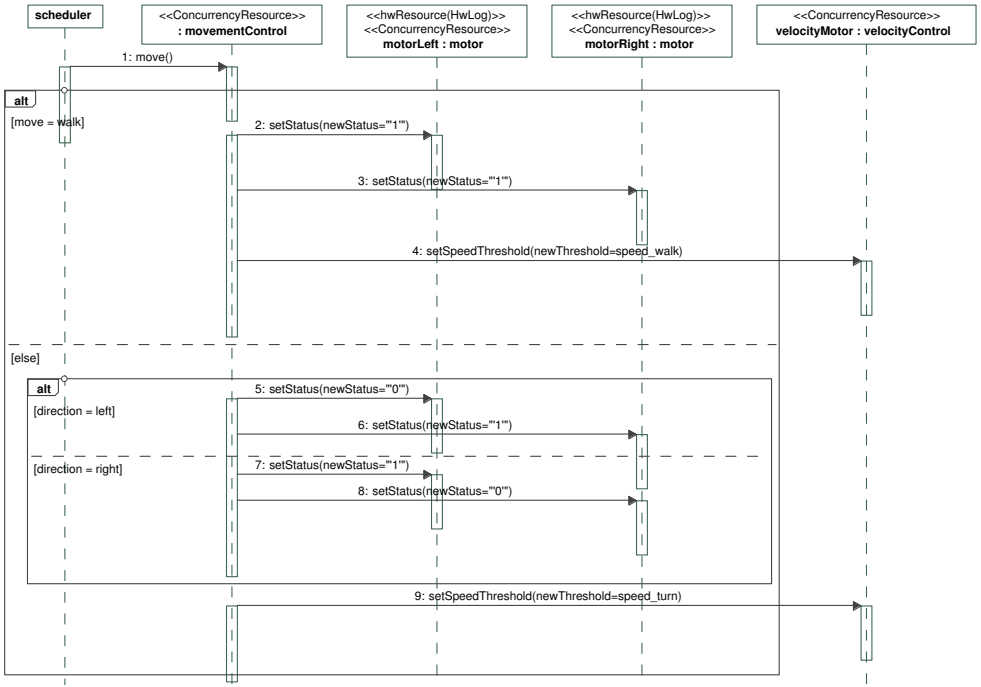
Fonte: Produção do próprio autor.

Figura 41 – Diagrama de sequência do método *main* da classe *movementControl*.



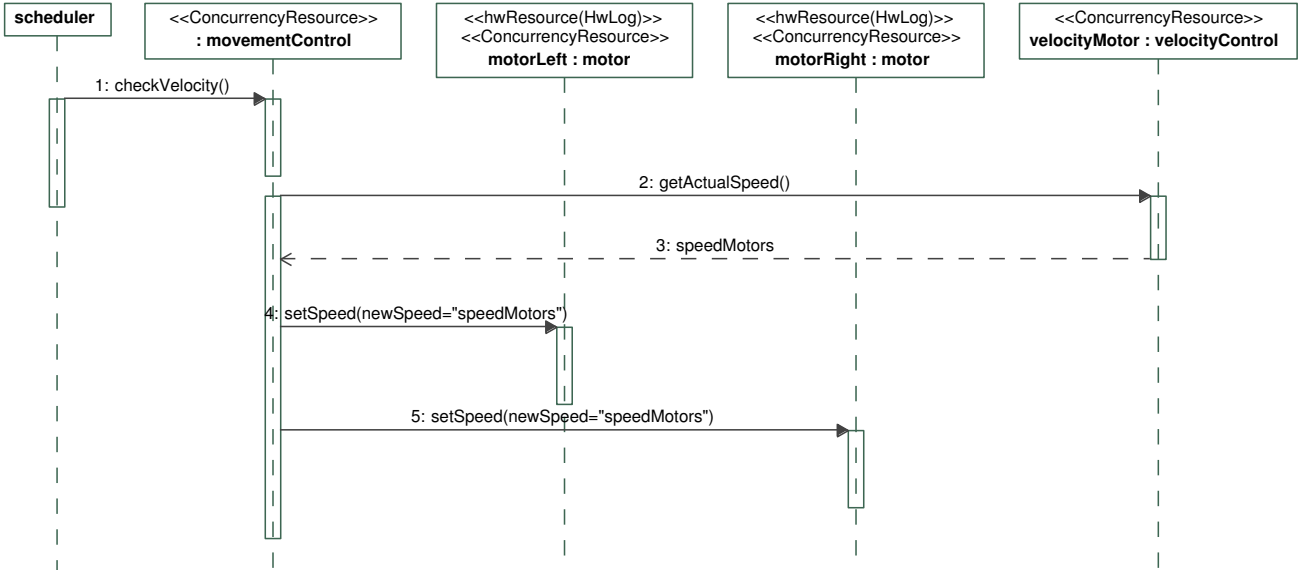
Fonte: Produção do próprio autor.

Figura 42 – Diagrama de sequência do método *move* da classe *movementControl*.



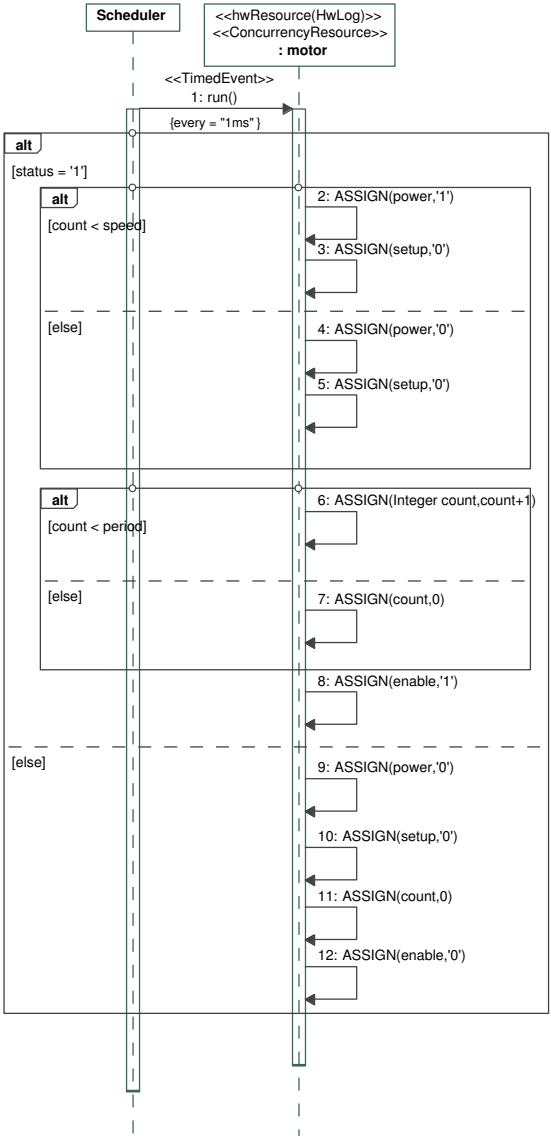
Fonte: Produção do próprio autor.

Figura 43 – Diagrama de sequência do método *checkVelocity* da classe *movementControl*.



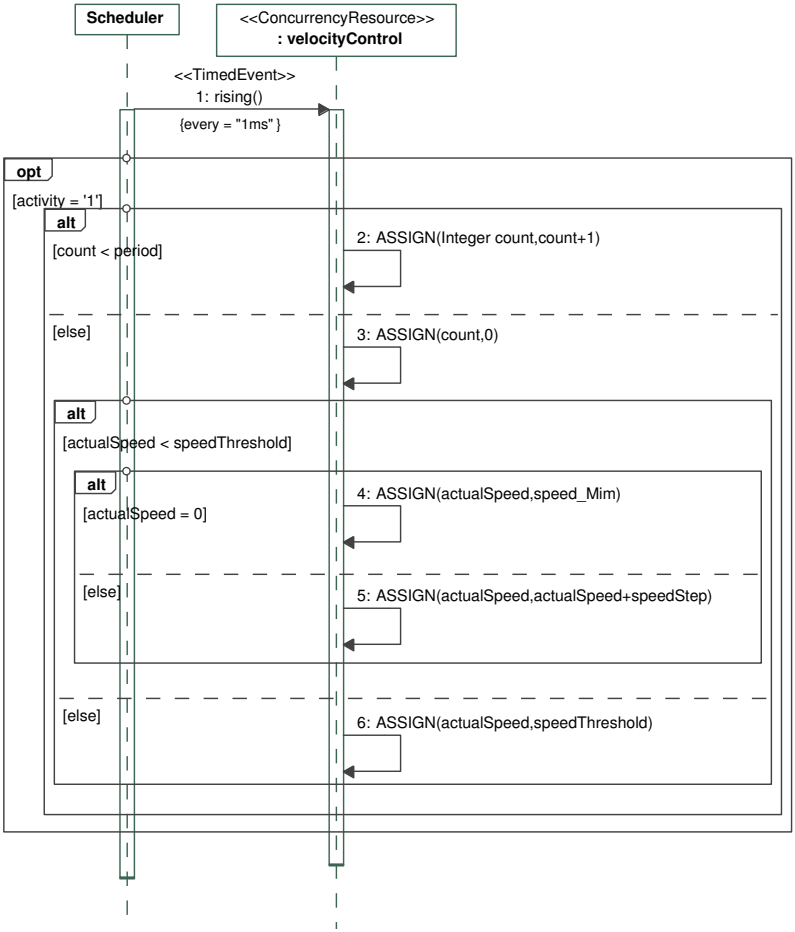
Fonte: Produção do próprio autor.

Figura 44 – Diagrama de sequência do método *run* da classe *motor*.



Fonte: Produção do próprio autor.

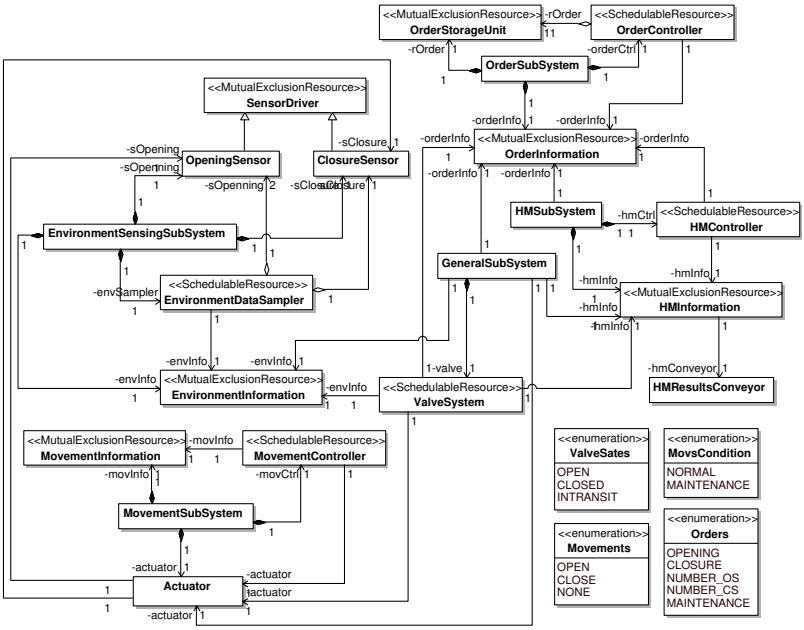
Figura 45 – Diagrama de sequência do método *rising* da classe *velocityControl*.



Fonte: Produção do próprio autor.

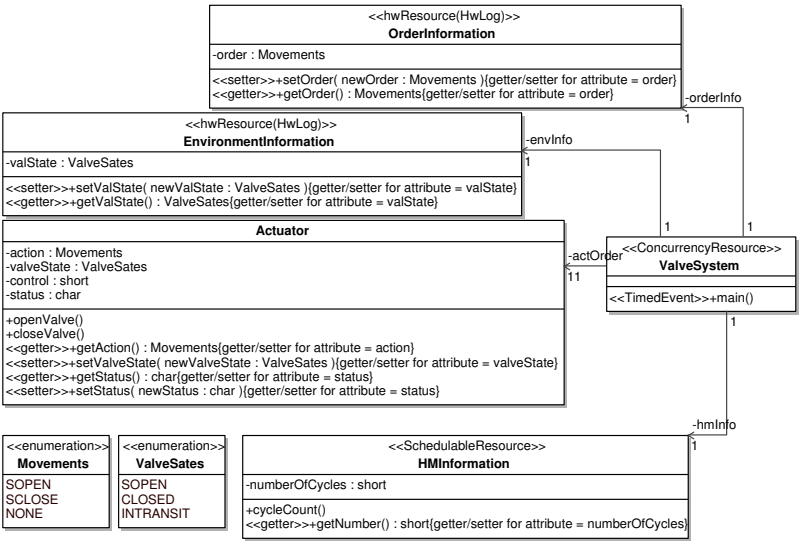
B.2 CONTROLE AUTOMÁTICO DE VÁLVULA

Figura 46 – Diagrama de classes do Controle de Válvula proposto por Moreira (2012).



Fonte: (MOREIRA, 2012).

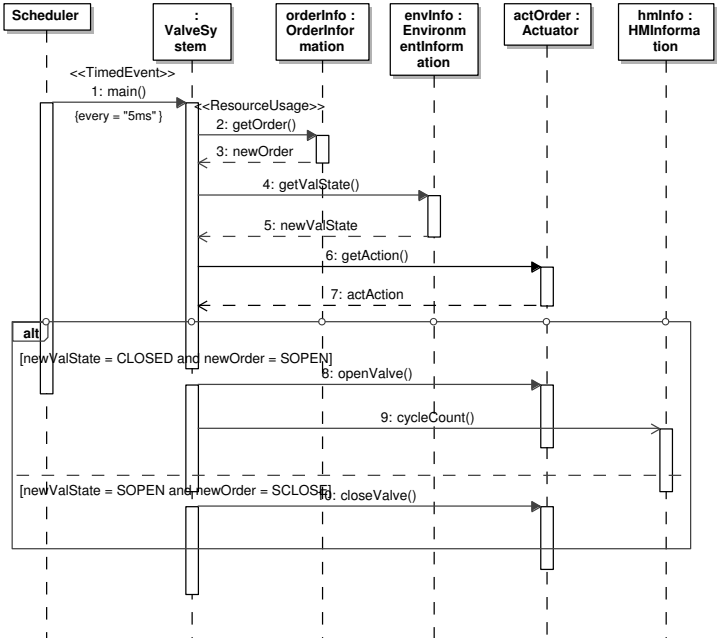
Figura 47 – Diagrama de classes resumido do Controle de Válvula.



Fonte: Produção do próprio autor. Adaptado de (MOREIRA, 2012).

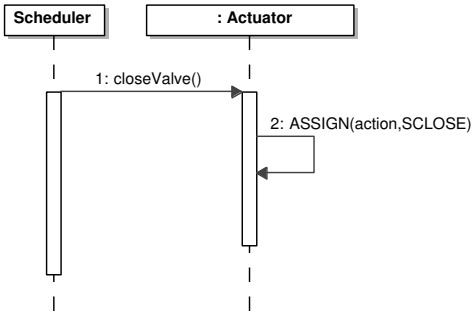
B.3 RELÓGIO

Figura 48 – Diagrama de sequência do método *main* da classe *ValveSystem*.



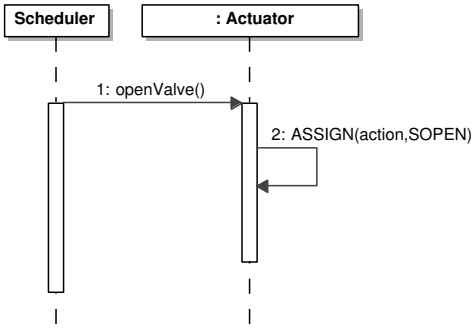
Fonte: Produção do próprio autor. Adaptado de (MOREIRA, 2012).

Figura 49 – Diagrama de sequência do método *closeValve* da classe *Actuator*.



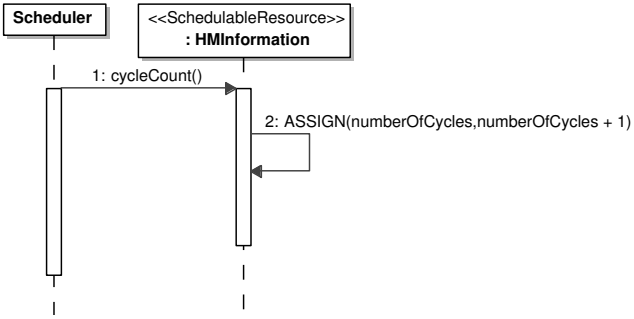
Fonte: Produção do próprio autor.

Figura 50 – Diagrama de sequência do método *openValve* da classe *Actuator*.



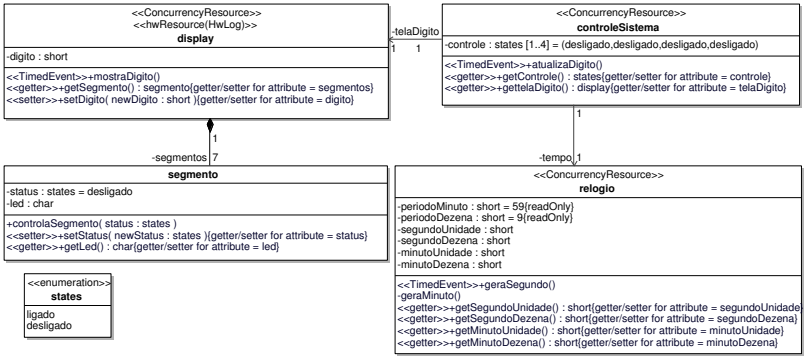
Fonte: Produção do próprio autor.

Figura 51 – Diagrama de sequência do método *cycleCount* da classe *HMInformation*.



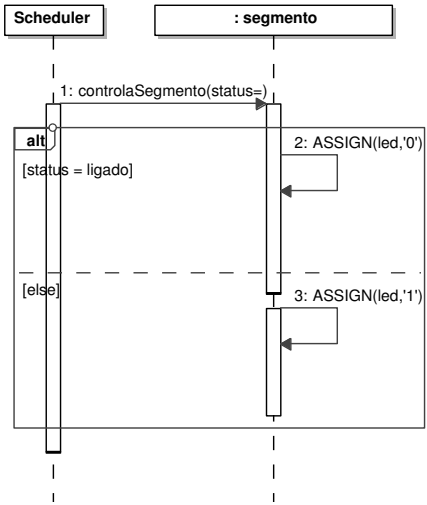
Fonte: Produção do próprio autor.

Figura 52 – Diagrama de classes do projeto Display.



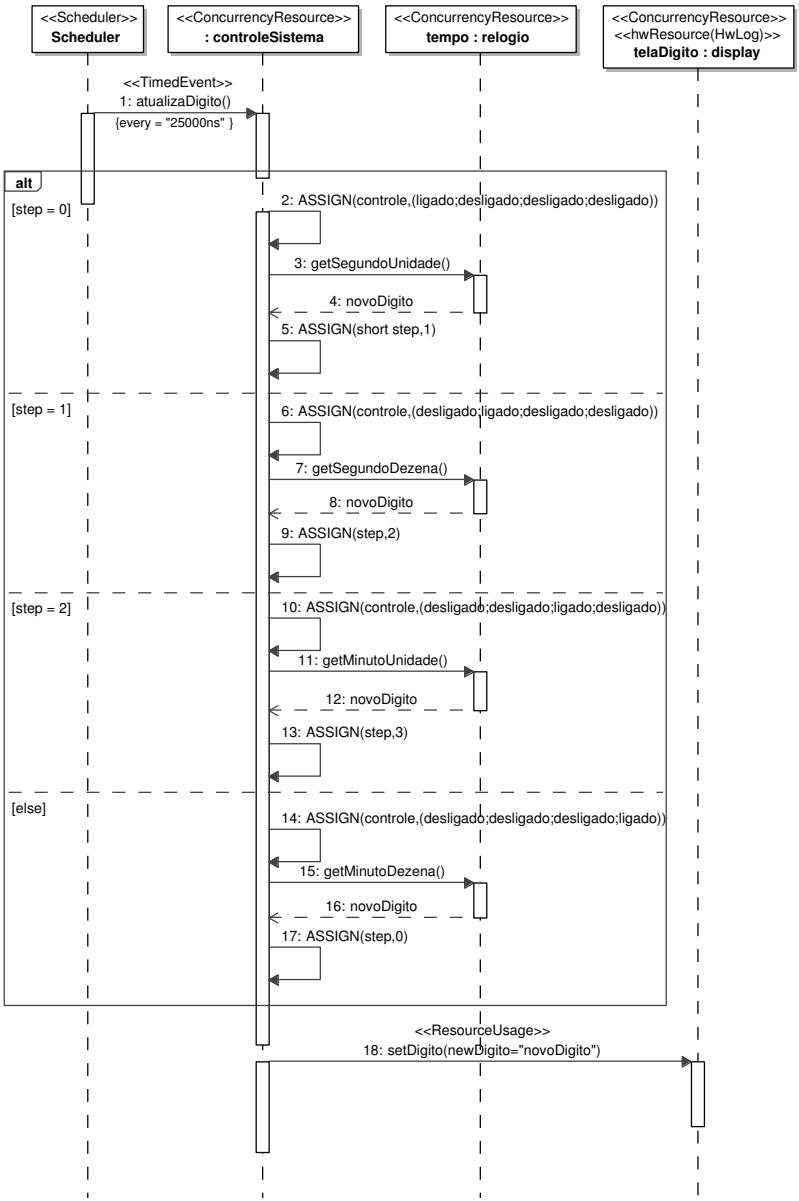
Fonte: Produção do próprio autor.

Figura 53 – Diagrama de sequência do método *controlaSegmento* da classe *Segmento*.



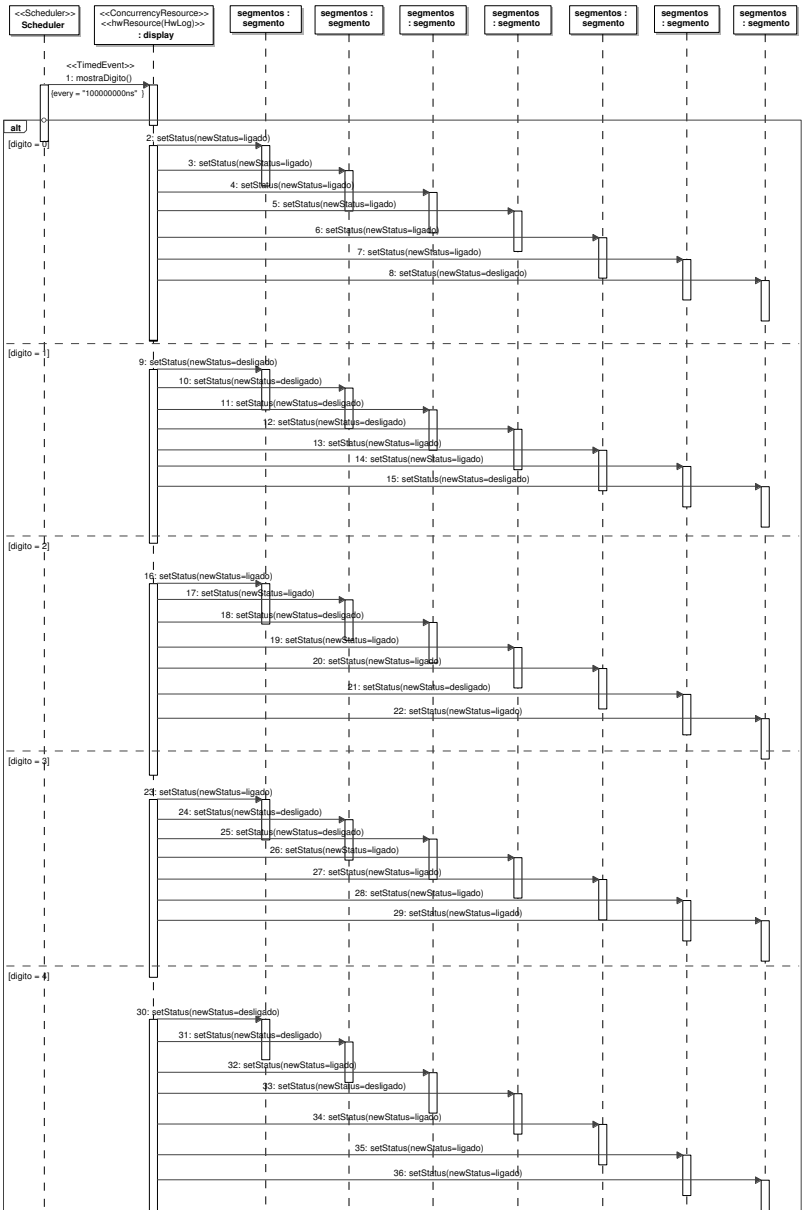
Fonte: Produção do próprio autor.

Figura 54 – Diagrama de sequência do método *atualizaDigito* da classe *controleSistema*.



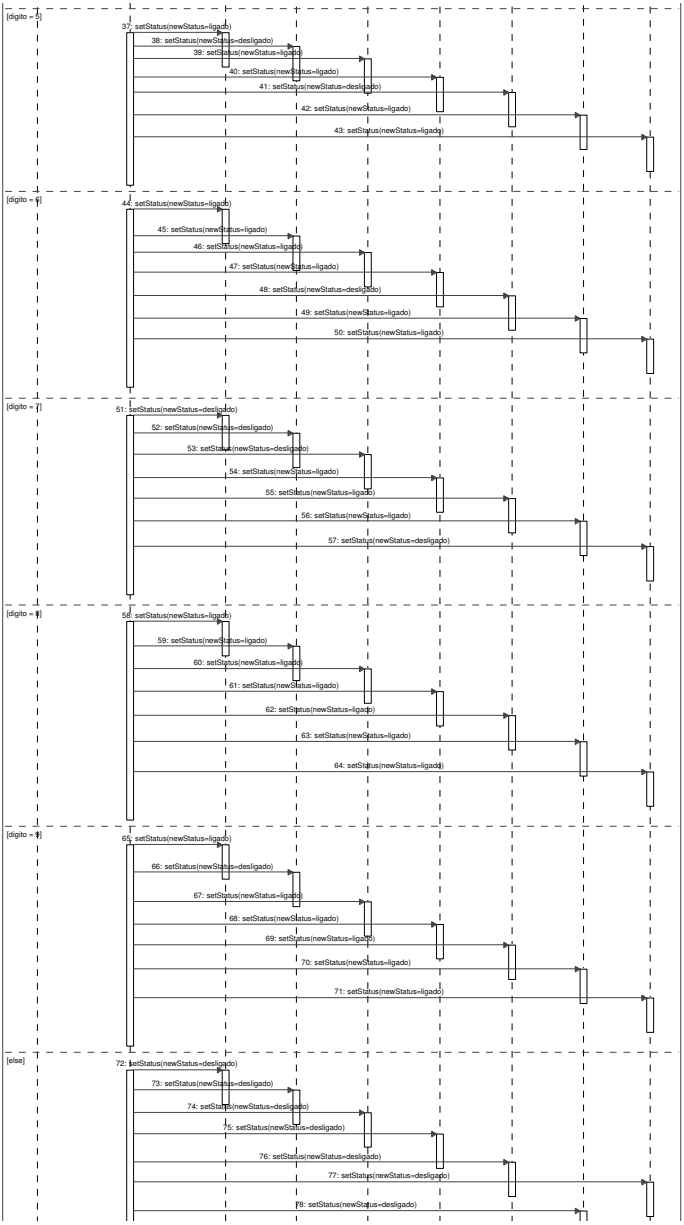
Fonte: Produção do próprio autor.

Figura 55 – Diagrama de sequência do método *mostraDigito* da classe *Display*.



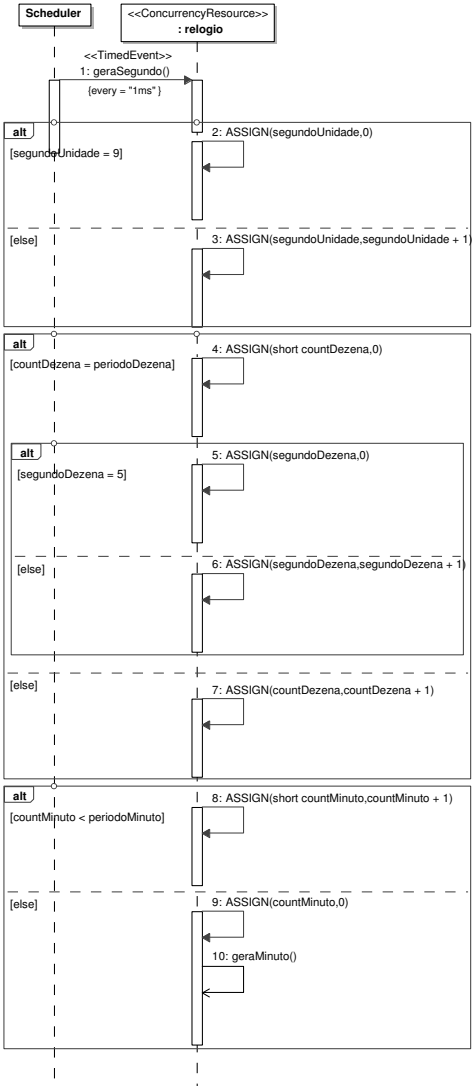
Fonte: Produção do próprio autor.

Figura 56 – Continuação do diagrama de sequência do método *mostraDigito* da classe *Display*.



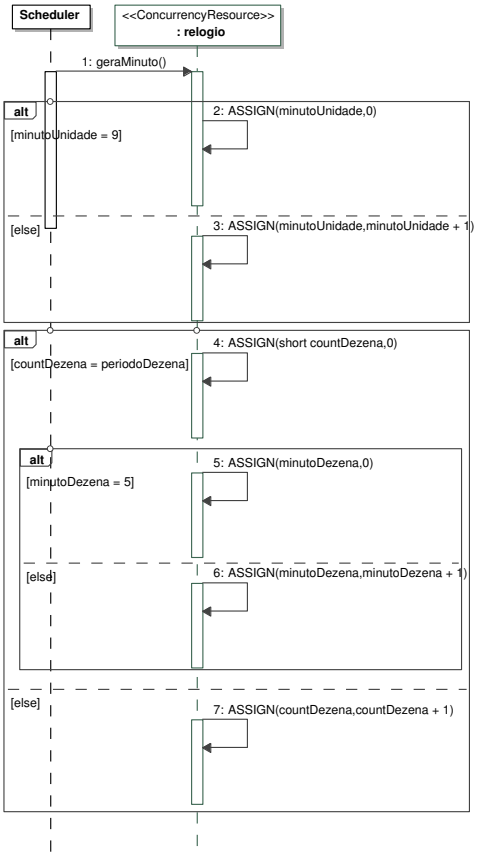
Fonte: Produção do próprio autor.

Figura 57 – Diagrama de sequência do método *geraSegundo* da classe *Relogio*.



Fonte: Produção do próprio autor.

Figura 58 – Diagrama de sequência do método *geraMinuto* da classe *Relogio*.



Fonte: Produção do próprio autor.

Esse trabalho apresenta uma abordagem para o desenvolvimento de sistemas embarcados implementados em FPGA que agrega técnicas de Engenharia Guiada por Modelos com técnicas e conceitos do paradigma de Desenvolvimento de Software Orientado a Aspectos.

Tal abordagem melhora o tratamento e gerenciamento de requisitos não funcionais já nas fases iniciais do projeto, utilizando níveis mais altos de abstração.

A implementação do sistema na plataforma FPGA/VHDL é gerada automaticamente a partir de modelos UML/MARTE que incluem a definição de aspectos que tratam os requisitos não-funcionais do sistema.

Os resultados encontrados mostram que é possível obter melhora no desempenho do sistema através da aplicação da abordagem proposta em descrições de componentes em hardware usando VHDL.

Orientador: Marco Aurélio Wehrmeister

Coorientador: Cristiano Damiani Vasconcellos

Joinville, 2014