UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

TOMÁS GARCIA MOREIRA

Geração Automática de Código VHDL a partir de Modelos UML para Sistemas Embarcados de Tempo-Real

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Carlos Eduardo Pereira
Orientador

Porto Alegre, março de 2012.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Moreira, Tomás Garcia

Geração Automática de Código VHDL a partir de Modelos UML para Sistemas Embarcados de Tempo-Real / Tomás Garcia Moreira – Porto Alegre: Programa de Pós-Graduação em Computação, 2012.

97 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2012. Orientador: Carlos Eduardo Pereira

1. Sistemas tempo-real embarcados. 2. Engenharia dirigida por modelos (Model-Driven Engineering - MDE) 3. Geração de automática de código. 4. UML. 5. VHDL. I. Pereira, Carlos Eduardo. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Luis Lamb
Coordenador do PPGC: Prof. Álvaro Freitas Moreira
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço à minha mãe, Sra. Vanja Moreira, pelo apoio incondicional, pelo amor e pelo suporte de toda uma vida. Particularmente, obrigado pelo incentivo nos diversos momentos da trajetória deste trabalho.

Ao Sr. Antônio Malta Neves, meu pai, que de sua maneira me apoiou em alguns momentos e me possibilitou, junto a minha mãe, a oportunidade de passar uma temporada fora do nosso país, onde realizei projetos profissionais e um sonho.

Agradeço também a minha família, namorada e amigos, pelo apoio e pelas duras críticas que me fizeram enxergar a importância de terminar este ciclo de mestrado na minha vida.

Em especial, um agradecimento ao meu orientador Carlos Eduardo Pereira, por oportunizar a colaboração entre laboratórios que me levou à França para realizar um trabalho, que rendeu muitos frutos, amizades e uma experiência de vida desigual. Também, agradeço por ter sido bastante duro nesta etapa final, uma motivação a mais para finalização deste trabalho.

Aos amigos e companheiros do laboratório de sistemas embarcados, agradeço pela amizade, discussões, dicas, e sobre tudo pela descontração das atividades extraclasse que ocorriam nas quintas-feiras e pelas muitas comemorações que deixam saudades.

Comme j'ai fait partie de ce travail dans le laboratoire CRAN (Centre de Recherche en Automatique de Nancy), je ne peux pas laisser de remercier à tous les collègues pour leur amitié et leur aide. Merci également aux professeurs Jean-François Pétin, Eric Levrat et Benoît Iung pour le soutien à l'élaboration du travail de Master. En particulier, je suis très reconnaissant au professeur Benoît Iung pour l'opportunité. Je vous remercie!

Ao Senhor, agradeço pelas oportunidades, saúde, capacidade para perseguir meus objetivos e desafios, força para seguir em frente, e pelos anjos que estiveram escondidos atrás de pessoas maravilhosas que apareceram na minha vida ao longo de muitas esquinas para ajudar e guiar o meu caminho.

“Merci à la vie pour tout ce qu'elle m'a donné jusqu'á aujourd'hui et par la certitude qu'il y à beaucoup de bonnes choses encore à venir.”

Tomás Moreira

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ACOD	*Aspects Crosscutting Overview Diagram*
AD	Analógico-Digital
AMoDE-RT	*Aspect-oriented Model-Driven Engineering for Real-Time systems*
AO	*Aspect-Oriented*
AOP	*Aspect-Oriented Programming*
API	*Application Programming Interface*
ASIC	*Application Specific Integrated Circuit*
AST	*Abstract Syntax Tree*
BR	Brasil
CASE	*Computer-Aided Software Engineering*
CIM	*The Computation Independent Model*
CISPI	*Conduite de grands systèmes industriels à risque*
ConPar	Controladores Paralelos
COSMOS	*Component based System Modeler and Simulator*
CRAN	*Centre de Recherche em Automatique de Nancy*
DA	Digital-Analógico
DARPA	Departamento de Defesa dos Estados Unidos
DERAF	*Distributed Embedded Real-time Aspects Framework*
DERCS	*Distributed Embedded Real-Time Compact Specification*
DSL	*Domain-Specific Language*
EA	*Enterprise Architect*
EDT	*Estelle Development Toolset*
EMF	*Eclipse Modeling Framework project*
e2v	*Estelle-to-VHDL*
FIFO	*First In First Out*
FPGA	*Field Programmable Gate Array*
FRIDA	*From RequIrements to Design using Aspects*

FR	*Functional Requirements*
FSM	*Finite State Machine*
GCC	*GNU Compiler Collection*
GenERTiCA	*Generation of Embedded Real-Time Code based on Aspects*
HW	*Hardware*
HDL	*Hardware Description Language*
IEEE	*Institute of Electrical and Electronic Engineers*
IP	*Intellectual Property*
JPDD	*Join Point Designation Diagrams*
JVM	*Java Virtual Machine*
MAC	*(Apple) Macintosh*
MARTE	*Modeling and Analysis of Real-Time and Embedded systems*
MBD	*Model-Based Development*
MDA	*Model-Driven Architecture*
MDD	*Model-Driven Development*
MDE	*Model-Driven Engineering*
MOF	*Meta-Object Facility*
NFR	*Non-Functional Requirements*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
OO	*Object-Oriented*
PC	*Personal Computer*
PIM	*Platform Independent Model*
PLC	Controladores de lógica programável
PN	*Petri Nets*
PSM	*Platform Specific Model*
PTII	*framework* Ptolemy II
QVT	*OMG*`s *Queries Views and Transformations document*
ROM	*Read Only Memory*
RTES	*Real-Time Embedded System*
RTL	*Register-Transfer Level*
RT-FRIDA	*Real-Time FRIDA*
RT-UML	*Real-Time UML*
SETRD	*Sistemas Embarcados Tempo-Real Distribuídos*
SDL	*Specification and Description Language*

SETRD	Sistema Embarcado Tempo-Real Distribuído
SiTra	*Simple Transformation*
SMDL	*State Machine Description Language*
SoC	*System on Chip*
SPT	*UML profile for Schedulability, Performance, and Time*
STRE	Sistema Real-Time Embarcado
SW	*Software*
SYNT	*SynVHDL*
TTM	*Time To Market*
UFRGS	Universidade Federal do Rio Grande do Sul
UML	*Unified Modeling Language*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very-High-Speed Integrated Circuit*
VTL	*Velocity Template Language*
XMI	*eXtensible Markup Interface*
XML	*eXtensible Markup Language*
XSLT	*eXtensible Stylesheet Language Transformations*
YACC	*Yet Another Compiler Compiler*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A crescente demanda da indústria exige a produção de dispositivos embarcados em menos tempo e com mais funcionalidades diferentes. Isso implica diretamente no processo de desenvolvimento destes produtos requerendo novas técnicas para absorver a complexidade crescente dos projetos e para acelerar suas etapas de desenvolvimento. A linguagem UML vem sendo utilizada para absorver a complexidade do projeto de sistemas embarcados através de sua representação gráfica que torna o processo mais simples e intuitivo. Para acelerar o desenvolvimento surgiram processos que permitem, diretamente a partir modelos UML, a geração de código para linguagens de descrição de software embarcado (C, C++, Java) e para linguagens tradicionais de descrição de hardware (VHDL, Verilog). Diversos trabalhos e ferramentas comerciais foram desenvolvidos para automatizar o processo de geração de código convencional a partir de modelos UML (software). No entanto, pela complexidade da transformação existem apenas poucos trabalhos e nenhuma ferramenta comercial direcionado à geração de HDL a partir de UML, tornando este processo ainda pouco difundido. Nossa proposta é focada na geração de descrições de hardware na linguagem VHDL a partir de modelos UML de sistemas tempo-real embarcados (STRE), surgindo como alternativa ao processo de desenvolvimento de hardware. Apresenta uma metodologia completa para geração automática de código VHDL, permitindo que o comportamento descrito para o sistema modelado seja testado e validado antes de ser desenvolvido, acelerando o processo de produção de hardware e diminuindo as chances de erros de projeto. É proposto como um processo de engenharia dirigido por modelos (MDE) que cobre desde as fases de análise de requisitos e modelagem UML, até a geração de código fonte na linguagem VHDL, onde o foco é gerar na forma de descrições de hardware, todas aquelas funções lógicas de um sistema embarcado que normalmente são desenvolvidas em software. Para atingir este objetivo, foi desenvolvido neste trabalho um conjunto de regras de mapeamento que estende a funcionalidade da ferramenta GenERTiCA, utilizada como suporte ao processo. Adicionalmente, foram pesquisados e desenvolvidos conceitos que serviram como base para o desenvolvimento de regras utilizadas pela ferramenta suporte para guiar o processo de mapeamento entre as linguagens. Os conceitos e as regras propostas foram validados por meio de um estudo de caso, cujos resultados obtidos estão demonstrados nesta dissertação.

Palavras-Chave: sistemas tempo-real embarcados (STRE), engenharia dirigida por modelos (MDE), geração automática de código, UML, VHDL.

Automatic VHDL code generation from UML Models for Real-Time Embedded Systems

ABSTRACT

The growing market demand requires the production of embedded devices in less time and with more different features. This directly implies on the development process of these products requiring new techniques to absorb the growing complexity of projects and to accelerate their development stages. UML has been used to handle the embedded systems design complexity through its graphical representation that makes the process simpler and more intuitive. To speed up the development cycle, it has emerged some processes that permit code generating directly from UML models to embedded software description languages (C, C++, Java), and traditional hardware description languages (VHDL, Verilog). Several researches and commercial tools have been developed to automate the code generation process from UML models to conventional languages (software). However, due to the transformation complexity there are only few studies and no commercial tool addressed to HDL generation from UML models, making this process almost unknown. Our proposal is focused on generating hardware descriptions as VHDL code from UML models of real-time embedded systems (RTES), emerging as an alternative to the hardware development. It presents a complete methodology to the VHDL code generation, allowing the behavior described to the modeled system to be tested and validated before being implemented, accelerating the hardware production and decreasing the chances of design errors. It is proposed as a model-driven engineering (MDE) process that covers the phases of requirements analysis, UML modeling, models transformations, and the source code generating process to the VHDL language, where the focus is to generate as hardware descriptions all the logic functions of an embedded system which are usually developed as software. To achieve this goal, this work was developed a set of mapping rules which extends the functionality of the tool GenERTiCA, used to support the process. Additionally, it was researched and developed concepts that were the basis for the development of rules used by the tool support to guide the mapping process between languages. The concepts and proposed rules have been validated through a case study, whose results are shown in this dissertation.

Keywords: real-time embedded systems (RTES), model-driven engineering (MDE), automatic code generation, UML, VHDL.

1 INTRODUÇÃO

De acordo com alguns estudos, somente 2% dentre os seis (6) milhões de processadores produzidos anualmente são utilizados em computadores convencionais (aqueles com teclado e monitor). Os restantes 98% encontram-se embarcados em algum tipo de dispositivo ou máquina, incluindo telefones celulares, eletrodomésticos, veículos, robôs, assim como em muitos outros domínios de aplicação (BARR, 2009), (TURLEY, 2002). Enfim, embora os computadores pessoais (PCs) sejam facilmente lembrados quando o assunto é processadores, os sistemas embarcados são muito mais numerosos e são responsáveis por toda a estrutura que utilizamos no dia a dia. Eles também são uma das áreas mais promissoras dentro do campo de tecnologia, visto que a nova categoria de processadores de hardware configurável, como são conhecidos os FPGAs *(Field Programmable Gate Array)*, permite que suas funcionalidades sejam definidas exclusivamente pelos usuários e não pelos fabricantes, podendo ser programados para desempenhar qualquer função.

Adicionalmente, muitos destes sistemas embarcados possuem requisitos de tempo-real, precisando realizar suas tarefas dentro de certas delimitações de tempo (PEREIRA; CARRO, 2007). Para atender esta grande demanda da indústria existe um engajamento global de pesquisadores no sentido de desenvolver metodologias, padrões, arquiteturas e ferramentas para acelerar e auxiliar o desenvolvimento sistemático desta classe de sistemas tempo-real embarcados (LONG *et al.*, 2005), (LE BEUX *et al.*, 2007), (WEHRMEISTER, 2009) e (POHL *et al.*, 2010).

Atualmente, a crescente complexidade dos Sistemas Tempo-Real Embarcados (STRE) impacta fortemente no tempo de projeto destes sistemas. Este aumento de complexidade é causado pela natureza dos diferentes componentes (i. e. componentes de hardware (HW) e software (SW)), pelo crescente número de funcionalidades que vão sendo incorporados dentro de um único STRE e, principalmente, pela característica não funcional de requisitos importantes que estão espalhados nos vários componentes do STRE. Pouca disponibilidade de recursos computacionais (e.g. memória e poder de processamento), restrições no consumo de energia sem degradação de desempenho e necessidade do atendimento das restrições temporais são exemplos de requisitos não funcionais presentes no domínio de sistemas (CARRO; WAGNER, 2003).

Alguns trabalhos indicam que aumentar do nível de abstração é uma solução adequada para o gerenciamento do aumento da complexidade no projeto de STRE (NEBEL *et al.*, 2000), (CHEN *et al.*, 2003), (MARTIN; MÜLLER, 2005) e (WEHRMEISTER, 2005). Contudo, abordagens tradicionais como a Orientação a Objetos (OO), apesar de serem bastante úteis a esta função, deixam a desejar porque não tem suporte adequado ao tratamento de requisitos não funcionais, pois o tratamento destes é misturado com o tratamento dos requisitos funcionais, o que dificulta a

compreensão e causa espalhamento do tratamento dos requisitos não funcionais. Estes problemas acabam por dificultar o reuso de artefatos (e.g. software, modelos, etc.) que foram previamente criados, testados e utilizados. A programação orientada a aspectos (KICZALES *et al.*, 1997) aparece como um conceito interessante para separar interesses que se entrecortam em unidades modulares, ou seja, separar o tratamento dos requisitos funcionais dos não funcionais em unidades distintas. Contudo utilizar esta separação de tratamento apenas na fase de implementação do STRE não é suficiente. Esta separação deve ocorrer também nas fases iniciais do projeto (e.g. fase de modelagem), como proposto na abordagem *Early-Aspects* (RASHID *et al.*, 2002) e (FREITAS, 2007).

Outro meio para diminuir o tempo de projeto é a adoção de uma linguagem comum para a especificação da estrutura e do comportamento dos STRE. Desta forma, várias equipes de desenvolvedores (i.e. equipes que desenvolvem HW e SW) podem facilmente trocar informações e evitar mal-entendidos sobre a especificação (CHEN *et al.*, 2003). Nos últimos anos a Linguagem de Modelagem Unificada (do inglês, *Unified Modeling Language* - UML) (OMG UML, 2010) vem recebendo bastante atenção por parte da comunidade de sistemas embarcados (DOUGLASS, 1999), (NEBEL *et al.*, 2000), (LONG *et al.*, 2005), (GéRARD; SELIC, 2008). Além de servir como uma linguagem comum para modelagem, a UML também aumenta o nível de abstração dos projetos através de sua representação gráfica, reduzindo a complexidade por tornar os projetos mais simples e intuitivos. Esta afirmação pode ser confirmada em (MARTIN; MÜLLER, 2005) onde encontram-se descritas algumas propostas, que permanecem usuais até os dias de hoje, do uso da UML no projeto de *System-on-Chip* (SoC).

A presente proposta objetiva fornecer uma metodologia de projetos baseada em modelos (do inglês, *Model-Driven Engineering* ou MDE) (OMG MDA, 2010) que utiliza a UML e pode utilizar conceitos do paradigma orientado a aspectos. Para atingir tal objetivo, utiliza um protótipo de uma ferramenta para geração de código chamada GenERTiCA (WEHRMEISTER *et al.*, 2008), que foi desenvolvida como parte da tese de doutorado de (WEHRMEISTER, 2009), do grupo de pesquisa em sistemas embarcados tempo-real distribuídos da Universidade Federal do Rio Grande do Sul (UFRGS). Essa ferramenta utiliza um mecanismo de script configurável para gerar automaticamente código fonte para estrutura, comportamento e tratamento de requisitos não funcionais de um Sistema Embarcado Tempo-Real e/ou Distribuído (SETRD). Além de gerar código, GenERTiCA também tem a capacidade de realizar o entrelaçamento (do inglês, *weaving*) dos aspectos que por ventura tenham sido especificados no modelo UML, no código gerado. Dessa forma linguagens que não foram inicialmente criadas para suportar o conceito de aspectos (KICZALES *et al.*, 1997), como C/C++ ou VHDL (*Very high integrated circuits Hardware Description Language*) (ASHENDEN, 1996), podem ser usadas como linguagens alvo do processo de geração de código a partir da UML.

Mais especificamente, a presente dissertação utiliza a ferramenta GenERTiCA para gerar código VHDL a partir de modelos UML de STRE. De forma que, é necessário descrever *scripts* contendo regras de mapeamento dos elementos presentes no modelo UML para convertê-los em elementos da linguagem VHDL. Além disso, é necessário fornecer a implementação (também na forma de *scripts* utilizados pela ferramenta na geração de código) dos aspectos que forem especificados no modelo UML. Como estudo de caso, pretende-se utilizar o modelo de um componente inteligente, desenvolvido em parceria com o CRAN (*Centre de Recherche en Automatique de Nancy* – França) para atuar na manutenção preventiva de componentes eletromecânicos

industriais. Trata-se de um sistema composto por uma válvula automatizada, que serve para controlar o fluxo de fluidos, e de sensores que diagnosticam o estado desta válvula. Através da aplicação da metodologia proposta neste trabalho, intencionamos integrar novas funcionalidades suportadas por um FPGA a esta válvula, transformando-a em um componente inteligente.

1.1 Objetivos e Delimitação do Escopo do Trabalho

Atualmente, a maioria dos projetistas utiliza como primeiro passo de um projeto complexo a modelagem completa de um sistema em alto nível. Descrevem hardware e software na forma de modelos, como um conjunto único e com todas suas conexões internas e relações externas, buscando caracterizar e viabilizar a ideia em desenvolvimento. Da mesma forma, modelam seu comportamento, especificando ações a serem tomadas em resposta a possíveis estímulos internos e externos, ou mesmo, ações que necessitam ser realizadas em determinados intervalos de tempo.

Normalmente, um sistema complexo é composto por requisitos funcionais e requisitos não funcionais, que quando modelados sob uma abordagem tradicional, ficam espalhados entre os diversos diagramas e elementos que compõem o sistema. Este espalhamento de requisitos não funcionais torna o modelo ambíguo e pode representar um problema. Principalmente, se estes modelos forem direcionados à geração automática de código, onde serão lidos por alguma ferramenta responsável por executar de forma automática o processo que traduz as informações contidas nos modelos para o respectivo código fonte em determinada linguagem.

Automatizar a geração de código a partir de modelos UML não é uma tarefa muito simples. O padrão UML dispõe de uma ampla gama de elementos e diagramas para modelagem de sistemas. Isso possibilita modelar a estrutura e o comportamento de sistemas de diversas formas, surgindo assim algumas características que podem influenciar negativamente no momento da geração de código. Adicionalmente, o *gap* semântico existente entre as duas representações do sistema (modelos e código) é muito grande. Para ir do modelo de um sistema até a representação da sua estrutura e comportamento em hardware e software, seriam necessários inúmeros passos num processo de desenvolvimento tradicional, onde o projetista refina o modelo até que o mesmo seja passível de implementação. A geração de código a partir de diagramas UML representa um grande desafio, pois tenta saltar esse *gap* realizando em poucos passos a tradução de um modelo em código.

O objetivo deste trabalho é descrever uma metodologia e desenvolver os meios necessários para tornar possível a geração automática de código fonte VHDL a partir de modelos UML, utilizando a ferramenta GenERTiCA. Também são propostos conceitos que permitam esse mapeamento entre descrições/modelos UML e VHDL. Conceitos que são necessários para orientar a conversão das estruturas do modelo para código e para descrever a forma que isso deve ser feito.

Devido ao grande número de diagramas UML e estilos de modelagem, é preciso orientar metodologicamente a fase inicial de projeto para que ao final tenha-se um modelo claro e com os itens necessários à representação do mesmo sistema na linguagem VHDL. Assim, o intuito da metodologia proposta nesta dissertação é guiar o

projetista a utilizar recursos específicos da UML e de maneira que a ferramenta suporte consiga retirar deste modelo às informações de forma correta.

O principal suporte ferramental deste trabalho é a ferramenta GenERTiCA. Ela mecaniza o processo de geração de código, realizando de forma automática e transparente ao usuário todas as transformações que levam do modelo inicial até o código fonte resultante na linguagem alvo. Seu processo utiliza transformações entre modelos, basendo-se nas práticas de MDE (*Model-Driven Engineering*). Utiliza um mecanismo de *scripts* para guiar seu processo de transformação. O conjunto de *scripts* utilizado para guiar o processo de geração de código para uma linguagem alvo representa as regras de mapeamento desta linguagem.

Atualmente, esta ferramenta dispõe somente de regras de mapeamento para as linguagens Java e C++. Essas linguagens são orientadas a objeto, assim como a linguagem UML. Isso torna o processo de conversão entre linguagens mais simples, visto que todas tem a mesma estrutura. A conversão de UML para uma linguagem estruturada, como o VHDL, é um grande desafio porque o *gap* semântico existente entre as duas linguagens é bastante grande. Ambas não compartilham de estruturas de linguagem semelhantes, exigindo que sua conversão seja realizada baseada em conceitos.

Desta forma, para mapear estruturas da linguagem UML para VHDL utilizando a ferramenta GenERTiCA é necessário desenvolver e formalizar conceitos, que explicitem a representação de uma linguagem em outra, e implementar na forma de *scripts* um conjunto de regras que servem para guiar o processo de transformação entre linguagens. Essas regras para o mapeamento VHDL caracterizam uma extensão à ferramenta GenERTiCA, ampliando sua funcionalidade e permitindo que a partir de um mesmo modelo UML possa ser gerado código para as linguagens Java, C++ e VHDL.

A proposta é validada experimentalmente com o desenvolvimento do estudo de caso da Válvula Inteligente, implementado para uma situação real existente no laboratório CRAN, numa parceria da UFRGS com a Universidade Henry Poincaré da França. O estudo de caso permite a validação da metodologia proposta neste trabalho para geração de código para hardware, assim como, propriamente a validação do código VHDL gerado. Através da demonstração do uso da metodologia proposta no desenvolvimento direto deste estudo de caso e da quantificação em números do código gerado, busca-se defender que o trabalho aqui proposto está bem fundamentado e tem resultados realmente válidos.

1.2 Contribuições

Existem diversos trabalhos publicados dirigidos à geração automática de código a partir de especificações UML. Entretanto, apenas uma faixa restrita deles atenta à geração de código para VHDL. A maioria dos trabalhos encontrados não é direcionada especificamente para sistemas embarcados. Não fazem qualquer menção à separação de requisitos funcionais dos não funcionais na fase de modelagem e, na maioria das vezes, geram código somente para estruturas restritas da linguagem VHDL (i. e. máquinas de estados), utilizando apenas subconjuntos do UML. Além disso, até o presente momento não existe uma ferramenta comercial para dar suporte à geração automática de código

fonte VHDL a partir de modelos UML de forma completa, o que abre espaço para pesquisas na área.

Buscando abordar os itens do parágrafo anterior, vistos como lacunas a serem preenchidas no processo de geração de código de UML para VHDL, este trabalho tem como contribuição a proposta de utilização de uma metodologia que auxilia e torna mais eficaz o processo de geração de código para sistemas embarcados, assim como, automatiza a geração de código VHDL a partir de modelos UML. Propõem-se para geração de código VHDL a partir de modelos UML a utilização de uma metodologia completa, que trata das fases de especificação de sistemas, modelagem, transformação entre modelos e geração de código, com o objetivo de obter ao final do processo descrições de hardware (VHDL) que representem a lógica funcional do sistema modelado (UML). Tal proposta apresenta-se diferenciada, visto que não foram encontradas abordagens semelhantes na análise do estado da arte realizada para este trabalho.

1.3 Organização do Texto

O Capítulo 2 revisa os conceitos primários envolvidos neste trabalho com intuito de ambientar o leitor ao tema que está sendo proposto. No Capítulo 3 é apresentada uma revisão de trabalhos relacionados ao tema "geração automática de código", discutindo-os em relação às técnicas e metodologias abordadas.

Os conceitos e as regras de mapeamento desenvolvidas para conversão de UML em VHDL são apresentados no Capítulo 4, sendo este o principal capítulo do trabalho. Nele estão descritos os passos da metodologia proposta, juntamente com um detalhamento sobre o mapeamento de conceitos realizado entre as linguagens, e também, sobre os *scripts* propostos para guiar a ferramenta suporte na geração do código.

O Capítulo 5 apresenta o estudo de caso desenvolvido para validação experimental deste trabalho. São analisados os resultados obtidos em termos de código, como por exemplo, percentagem de código válido e quantidade de código gerado. Também é realizada a validação do código gerado, verificando se o mesmo representa com fidelidade a estrutura e o comportamento do sistema modelado.

Por fim, o Capítulo 6 apresenta conclusões e motivações para trabalhos futuros.

2 REVISÃO DE CONCEITOS

Este capítulo apresenta uma revisão das áreas de conhecimento abordadas nesta dissertação. São revisados conceitos gerais sobre sistemas embarcados, técnicas para modelagem e implementação de sistemas tempo-real embarcados, noções de orientação a aspectos, conceitos sobre engenharia dirigida por modelos (do inglês, MDE), definições sobre especificação de sistemas tempo-real em modelos UML (*Real-Time* UML), e também, uma apresentação da linguagem de descrição de hardware VHDL.

2.1 Sistemas Embarcados

Existem muitas definições de sistemas embarcados, mas todas estas definições podem ser combinadas em um conceito simples. Um sistema embarcado é um sistema computacional de propósito especial que é utilizado em uma tarefa específica. Este sistema computacional de propósito especial é normalmente menos poderoso que um sistema computacional de propósito geral, embora existam sistemas embarcados que são bastante complexos, desempenhando várias funções diferentes (WOLF, 2001).

Segundo Barr (2009) dos 6,2 bilhões de processadores produzidos em 2002, menos de 2% tornaram-se o cérebro de novos PCs, MACs e *Workstations* Unix. Os outros 6,1 bilhões foram usados em sistemas embarcados. Quase todos os dispositivos eletrônicos modernos desde brinquedos, eletrônica embarcada de aviões a controladores de máquinas industriais, utilizam processadores para ajudar a controlar fábricas, gerenciar sistemas de armamento e habilitar a comunicação entre pessoas e produtos.

Normalmente sistemas embarcados utilizam processadores de baixo consumo de potência com uma quantidade limitada de recursos, como por exemplo, memória. Alguns destes sistemas embarcados utilizam sistemas operacionais muito pequenos, com capacidade limitada de operação, otimizados para controlar apenas suas funcionalidades (CARRO; WAGNER, 2003). Além disso, a escolha dos componentes do sistema sempre deve ser baseada nas características da aplicação para a qual o sistema embarcado está sendo proposto.

Muitos sistemas embarcados são concebidos partindo do pressuposto que devem ser utilizados por um longo período de tempo sem necessitarem manutenção. Na realidade, a intenção de se produzir um sistema embarcado para uma determinada aplicação é construir um sistema e deixá-lo operando independentemente por toda a sua vida útil (WOLF, 2001). Por essa razão a maioria dos sistemas embarcados não possui componentes mecânicos como ventiladores, discos, etc. Estes tipos de componentes sofrem desgaste natural com o tempo e periodicamente necessitam reposição ou manutenção. Ao invés disso são utilizados componentes alternativos que provêm a

mesma funcionalidade como, por exemplo, componentes de memória ROM e *flash* onde são armazenados o sistema operacional e o software da aplicação.

Os sistemas embarcados podem ainda ser divididos segundo sua classe de operação. Neste caso, são ainda mais especializados, com funções direcionadas ao tipo de tarefa que devem executar dentro do domínio de aplicação para o qual foram projetados. O presente trabalho é direcionado ao domínio dos sistemas embarcados de Tempo-Real, onde além de possuírem restrições não funcionais comuns, como por exemplo, poder de processamento, tamanho de memória e consumo de potência, eles ainda possuem restrições temporais, específicas do domínio de aplicação.

Em sistemas de tempo-real o correto processamento dos algoritmos implementados não é suficiente para garantir o correto funcionamento do sistema, ou seja, o resultado do processamento dos algoritmos do sistema deve estar pronto nos tempos definidos pelos requisitos temporais do sistema tempo-real. Nesta classe de sistemas, processar dados em microssegundos não torna um sistema tempo-real, o que importa são os tempos de resposta serem limitados e previsíveis.

A previsibilidade, além de outras, é uma característica fundamental para esse tipo de sistema. O comportamento do sistema ou o tempo de reação aos estímulos precisa ser conhecido, pois quando aliados aos sistemas embarcados (i. e. sistemas tempo-real embarcados), eles são tipicamente encontrados interagindo com o meio físico, onde podem sofrer falhas consideradas críticas se as restrições temporais forem violadas. Um exemplo de um sistema tempo-real embarcado que interage com os seres humanos é um sistema de supervisão do motor de um carro. Uma vez que o processamento de um sinal atrase, pode causar falha do motor e colocar em risco a vida de seres humanos. Outros exemplos são equipamentos médicos como marca-passo cardíaco ou um sistema de controle de tráfego aéreo.

Figura 2.1: Exemplos da utilização de sistemas tempo-real embarcados.

2.2 Desenvolvimento de Sistemas Tempo-Real Embarcados

As práticas correntes para concepção de sistemas tempo-real embarcados têm resolvido de forma aceitável os problemas de desenvolvimento de uma parte das aplicações, principalmente aquelas menos críticas. Porém, devido ao aumento da complexidade dos sistemas tempo-real embarcados e de requisitos mais exigentes quanto ao consumo de energia, à portabilidade, ao desempenho, à confiabilidade e ao tempo de projeto, cresce a demanda por novas metodologias, ferramentas e paradigmas para a concepção desses sistemas.

O projeto de sistemas tempo-real embarcados é complexo devido ao fato de possuir funcionalidades implementadas em software e/ou hardware, dependendo dos requisitos e das restrições impostas pela aplicação. Segundo Chen *et al.* (2003), a abordagem de projeto de sistemas embarcados usada na indústria é informal, especialmente nas fases iniciais onde os requisitos e funcionalidades do sistema são expressos usualmente em linguagem natural.

Os sistemas tempo-real embarcados possuem um grande espaço de projeto arquitetural onde existem diversas opções que podem ser exploradas, sempre levando em consideração os requisitos da aplicação. Uma solução que recentemente ganhou popularidade é uso de sistemas inteiros integrados em uma única pastilha, também conhecidos como *Systems-on-Chip* (SoC). Os SoCs ganharam popularidade devido ao crescimento da necessidade de mais desempenho em diversos domínios de aplicação como as aplicações de multimídia, dispositivos portáteis, entre outros. Os sistemas computacionais presentes em um SoC podem ser dos mais variados tipos como microcontroladores, microprocessadores, circuitos integrados específicos para a aplicação (*Application Specific Integrated Circuits* ou ASIC) e conversores ADs e DAs (Analógico-Digital – Digital-Analógico).

Entretanto, os sistemas embarcados também possuem componentes implementados em software que necessitam ser desenvolvidos concorrentemente com os componentes em hardware de modo a atender os requisitos do mercado (*time-to-market*). Em muitos casos a parte em software domina o tempo e o custo do projeto de um sistema tempo-real embarcado devido a razões como a sua complexidade e a sua flexibilidade que permite alterar uma funcionalidade nas fases finais do projeto (NEBEL *et al.*, 2000).

Conforme Carro e Wagner (2003), no projeto de sistemas embarcados cada vez mais a inovação de uma aplicação depende do software. Isso se deve ao fato da automação do projeto de hardware ser feita através do reuso de componentes e plataformas de hardware previamente desenvolvidas. O projeto do software embarcado deve essencialmente seguir este mesmo princípio, a reutilização de componentes previamente desenvolvidos, de modo que o projeto de um sistema tempo-real embarcado concentre-se apenas na configuração e integração de todos os componentes de hardware/software.

Segundo Balarin *et al.* (2003), um fluxo de projeto consistente deve capturar projetos em níveis de abstração bem definidos e então progredir em direção a uma implementação eficiente do sistema modelado. O fluxo de projetos utilizado atualmente ainda carece de ferramentas adequadas de suporte, forçando os projetistas no nível de sistema a utilizar conjuntos de ferramentas que não são interligados, o que leva a interações indesejadas e desnecessárias das várias equipes de desenvolvimento a fim de chegar a um consenso dos requisitos do projeto. Muitas vezes as equipes compartilham pouco o entendimento do seu domínio de conhecimento com as outras equipes, o que

provoca incertezas no projeto que podem resultar em erros difíceis de identificar e depurar.

2.2.1 Modelagem de Sistemas Tempo-Real Embarcados

Os modelos de um projeto mostram os elementos que fazem parte de um sistema e/ou interagem com ele. Nestes modelos são mostrados os objetos e classes que compõem o sistema bem como os diferentes tipos de relações entre eles. Seu objetivo é promover a ligação entre os requisitos do sistema e a implementação deste. Uma característica importante dos modelos em um projeto é a sua abstração, onde dependendo do nível de projeto ao qual pertence o modelo, detalhes desnecessários naquele momento não devem ser incluídos de forma a facilitar o entendimento dos elementos modelados. Contudo, devem-se incluir detalhes suficientes de forma que os outros integrantes do projeto possam tomar decisões a respeito da sua implementação (SOMMERVILLE, 2003).

Uma etapa importante na modelagem de um sistema é a decisão de quais os modelos são necessários e qual o adequado nível de detalhamento desses modelos. Essa decisão depende do tipo de sistema que está sendo desenvolvido. Um sistema para computadores de mesa será projetado diferentemente de um sistema tempo-real embarcado e, portanto, diferentes modelos serão utilizados no projeto. Existem poucos sistemas em que todos os tipos de modelos são necessários assim, minimizando o número de modelos produzidos tem-se a diminuição dos custos de projeto.

Segundo Sommerville (2003) existem dois tipos de modelos de projeto que devem ser utilizados em um projeto orientado a objetos:

- **Modelos Estáticos:** são modelos que descrevem a estrutura estática do sistema em termos das classes de objetos destes sistemas bem como os seus relacionamentos. Os relacionamentos importantes a serem expressos nestes modelos são as relações de generalização, de associação e de composição;
- **Modelos Dinâmicos:** são modelos que descrevem a estrutura e o comportamento dinâmico do sistema e mostram as relações entre os objetos do sistema. Dentre as interações que são importantes neste tipo de modelo estão a sequência de requisições de serviços e o modo como o estado do sistema está relacionado com essas interações.

Conforme Axelsson (2000), o propósito das linguagens de modelagem é dar suporte aos projetistas do sistema tempo-real embarcado durante todo o processo de desenvolvimento. Auxiliando desde as especificações iniciais até a entrega do produto final e sua manutenção, permitindo que previsões sobre características finais do sistema possam ser feitas durante as várias etapas de desenvolvimento do projeto. A previsão das características relacionadas com a sua funcionalidade e o desempenho são especialmente relevantes durante o processo de desenvolvimento de sistemas tempo-real embarcados.

2.2.2 Implementação de Sistemas Tempo-Real Embarcados

Após todos os requisitos do sistema tempo-real embarcado terem sido expressos no modelo em alto-nível do sistema, é iniciada a fase de implementação do mesmo. Nesta fase são desenvolvidos os componentes do software e é feita a configuração dos componentes de hardware, em muitos projetos é necessário também o desenvolvimento de ASICs que são responsáveis pela execução de determinadas funções do sistema.

Segundo Balarin *et al.* (2003), no passado as configurações de hardware dominavam a maioria das implementações, todavia na abordagem atual de implementação, a maioria das aplicações é implementada em configurações mistas onde na maioria dos casos o software constitui a principal parte do sistema.

Geralmente o projeto de software para os sistemas tempo-real embarcados era feito utilizando linguagens como *assembly* ou C para tentar otimizar ao máximo o código executável da aplicação. Contudo, programas escritos nesse tipo de linguagem são difíceis de manter e, muitas vezes, não podem ser reutilizados em outros projetos. A utilização desse tipo de linguagem deve-se principalmente ao fato dos sistemas embarcados possuírem restrições em sua capacidade de processamento, memória e consumo de potência, exigindo que o software seja o mais otimizado possível. Em função de avanços obtidos especialmente na área de microeletrônica, as capacidades de hardware evoluíram bastante e o custo de inclusão de mais memória ou um processador com alto desempenho e baixo consumo de energia se tornou mais baixo. Este fato barateou o hardware e transformou o custo do projeto do software do sistema tempo-real embarcado no custo dominante para o desenvolvimento do sistema, aumentando assim a sua importância. Assim sendo, para uma redução no custo de projeto de um sistema tempo-real embarcado, é fundamental se otimizar o processo de desenvolvimento de softwares embarcados.

Por essa razão a utilização da técnica de orientação a objetos na implementação tanto do software como do hardware de sistemas tempo-real embarcados ganhou força (WEHRMEISTER, 2005), pois permite o aumento do nível de abstração durante a codificação e facilita o reuso de componentes desenvolvidos em outros projetos. O aumento do nível de abstração traz consigo um *overhead* no código final gerado pelos compiladores. Contudo este problema não é tão relevante hoje em dia graças ao avanço da tecnologia, pois o poder de processamento e a capacidade de armazenamento aumentaram muito nos dispositivos embarcados.

Uma linguagem orientada a objetos que se tornou praticamente um padrão para o desenvolvimento de software embarcado é a linguagem Java. Principalmente, devido ao fato de ser multi-plataforma. Aplicativos Java executam sobre Máquinas Virtuais Java (do inglês *Java Virtual Machine* - JVM), que são responsáveis por converter os bytecodes (código Java do aplicativo) em código nativo da máquina onde se quer executar. Graças à Máquina Virtual Java, os programas escritos em Java podem funcionar em qualquer plataforma de hardware e software que possua uma versão de JVM, tornando assim essas aplicações independentes da plataforma onde funcionam.

2.3 Orientação a Aspectos

A programação orientada a aspectos (do inglês, *Aspect Oriented Programming* ou AOP) é um paradigma de programação de computadores que permite aos desenvolvedores de software separar e organizar o código de acordo com a sua importância para a aplicação (*separation of concerns*).

O princípio da separação de interesses (*concerns*) foi introduzido por (DIJKSTRA, 1976), com o objetivo de dividir o domínio do problema em partes menores com o intuito de entender melhor cada parte separadamente. Um interesse (*concern*) é alguma parte do problema que se deseja tratar como uma unidade conceitual única. No

desenvolvimento de software, um interesse pode ser visto como um requisito funcional ou não funcional de um sistema. Um software é composto pelos seus requisitos funcionais, a parte do sistema que trata das necessidades do cliente, e não funcionais (aspectos), que tratam o comportamento e as restrições do sistema. De forma geral os vários interesses do sistema devem ser separados em módulos – modularizados – de acordo com as abstrações do desenvolvimento de software providas por linguagens, métodos e ferramentas.

A programação orientada a objetos (do inglês, *Object-Oriented Programming* - OOP) mistura os requisitos funcionais e os requisitos não funcionais do sistema, esse entrelaçamento dificulta a reutilização do código. Uma das soluções para minimizar esse problema é a utilização de padrões de projeto, contudo, o uso dessa técnica também deixa o código que o implementa espalhado e entrelaçado com o código funcional do sistema.

Todo o programa escrito em OOP possui código que é alheio à implementação do comportamento do objeto. Este código é todo aquele utilizado para implementar funcionalidades secundárias e que se encontra espalhado por toda a aplicação (*crosscutting concerns*). A AOP permite que esse código seja encapsulado e modularizado. Adicionalmente, AOP busca separar os requisitos funcionais e não funcionais, ou seja, dividir os interesses (*concerns*), para depois uni-los (*weaving*), formando um sistema completo. A figura 2.2 exemplifica como ficaria um código com requisitos não funcionais sem (OOP) e com a utilização de aspectos (OOP+AOP).

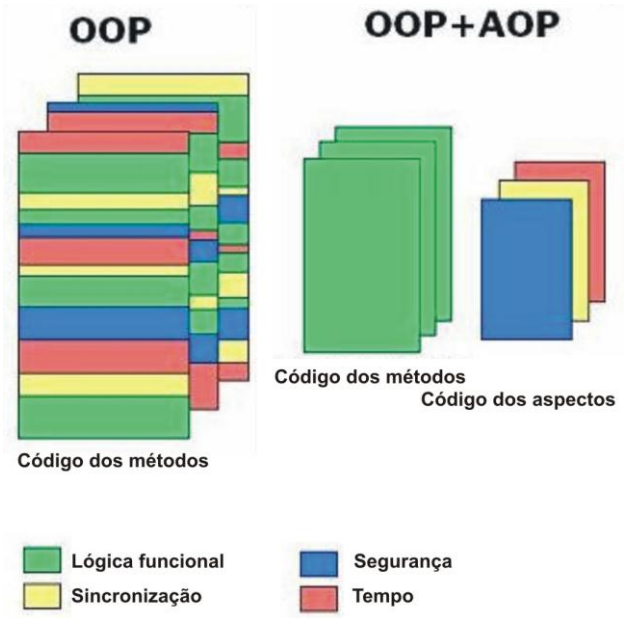

Figura 2.2: Exemplo de OOP e AOP.

Uma questão importante no projeto de sistemas tempo-real embarcados é o tratamento dos requisitos funcionais e dos não funcionais (do inglês, *Functional Requirements* - FR, *Non-Functional Requirements* - NFR). Sendo que a maior dificuldade é lidar com requisitos não funcionais, por possuírem *crosscutting concerns*, o que significa que podem afetar muitas partes distintas de um mesmo sistema. Se não forem devidamente manuseados, esses NFRs tornam-se responsáveis pelo entrelaçamento do código (*tangled code*) e pela perda de coesão no mesmo.

Sistemas de tempo-real têm um importante NFR que está relacionado aos aspectos temporais, tais como *deadline*, *jitter* máximo, pior tempo de execução, e outros. A complexidade relacionada à análise não funcional destes sistemas aumenta quando eles se tornam embarcados, ou ainda, embarcados e distribuídos.

Para lidar com esses NFRs algumas propostas sugerem o uso de aspectos (RASHID *et al.*, 2002), (STANKOVIC *et al.*, 2003) e (FREITAS *et al.*, 2007), pois eles podem ajudar a lidar com *crosscutting* NFRs no projeto de sistemas tempo-real embarcados, modularizando seu tratamento. Além da capacidade de modularização, aspectos mais abstratos são mais fáceis de reusar porque eles definem o tratamento de um interesse (*concern*) em alto nível e também, como esse aspecto pode ser aplicado (ou como afeta) o sistema sem restringir a implementação ou a plataforma.

2.4 Engenharia Dirigida por Modelos (*Model-Driven Engineering*)

Engenharia Dirigida por Modelos (do inglês, *Model Driven Engineering* - MDE) é um paradigma que foi proposto para fazer face à crescente complexidade do campo do desenvolvimento de software. Neste novo paradigma, o ciclo de vida de desenvolvimento de software não é muito diferente do encontrado na literatura tradicional. A diferença fundamental é que os artefatos criados durante o processo de desenvolvimento são modelos "formais", isto é, modelos que podem ser compreendidos e então manipulados por computadores. Um exemplo de MDE é a proposta da OMG para desenvolvimento de softwares, Arquitetura Dirigida por Modelos (do inglês, *Model Driven Architecture* - MDA) (OMG MDA, 2010). Nesta arquitetura, a OMG propõe basicamente a criação de modelos independentes da plataforma (PIM – *Platform Independent Model*) e a transformação de PIMs em modelos específicos a plataforma (PSM – *Platform Specific Model*). Assim, a complexidade de desenvolvimento de software é gerenciada por modelos, favorecendo o desenvolvimento, a manutenção e a evolução de softwares. O framework de modelagem do projeto "Eclipse" e o software *Factories* da Microsoft são outros exemplos de MDE. Todos estes exemplos apresentam pontos em comum, mas diferem com respeito à tecnologia, por exemplo, MDA utiliza MOF (*Meta-Object Facility*), UML e *profiles*, enquanto que o framework EMF (*Eclipse Modeling Framework project*) utiliza *Ecore* e favorece a criação de Linguagens Específicas ao Domínio (DSL – *Domain-Specific Language*). MDE pode abranger: geração de código a partir de um modelo do domínio, linguagens e motores de transformação, gerenciamento de metadados, ferramentas para a especificação de correspondências entre modelos, algoritmos de metamodel *matching* e metodologias.

MDA é uma iniciativa do OMG que promove o uso de modelos independentes de detalhes de implementação no desenvolvimento de software. Através de transformações sucessivas, os modelos originais, que tratam das preocupações ligadas ao domínio do problema, são transformados e adornados com artefatos relativos à implementação, eventualmente gerando um sistema executável completo. O potencial do uso conjunto do paradigma de Aspectos e MDA é promissor. Baseia-se na hipótese de que o maior grau de modularidade provido pela orientação a aspectos, somado ao alto nível de abstração e independência de plataforma proposto por MDA, colabora no aumento de compreensibilidade, flexibilidade, portabilidade e consequentemente manutenibilidade de um sistema. A figura abaixo demonstra o fluxo de projeto determinado pela metodologia MDA. Parte de uma visão do sistema totalmente independente de lógica

computacional (do inglês, *The Computation Independent Model* - CIM). Então, passa pela primeira transformação que significa o desenvolvimento de um modelo independente de plataforma (PIM). Após, esse PIM é transformado em um modelo específico da linguagem alvo (PSM), que serve de base para geração do código fonte na linguagem escolhida.

Figura 2.3: Fluxo utilizado para metodologia MDA.

A própria UML é um modelo de linguagem para modelagem de dados orientada a objetos. Com ela pode-se fazer uma modelagem visual de maneira que os relacionamentos entre os componentes do sistema sejam melhor visualizados, compreendidos e documentados. Podemos dizer também que a UML é uma linguagem para especificar, construir, visualizar e documentar um sistema de software que surgiu com a fusão das metodologias já anteriormente usadas e tem como objetivo:

- Modelar sistemas usando os conceitos da orientação a objetos
- Estabelecer um elo explícito entre os artefatos conceituais e os executáveis
- Tratar questões de representar sistemas complexos de missão crítica
- Criar uma linguagem de modelagem que possa ser usada por homens e por máquinas

Atualmente a UML está na versão 2.4 - Beta 2 (de março de 2011), e desde sua versão 1.5 vem sendo comumente utilizada para construção de modelos, por ser capaz de expressar comportamento através da semântica de ações.

2.5 Real-Time UML

A UML foi criada para ser uma linguagem de especificação de propósito geral para o desenvolvimento de software. Sua ampla aceitação a tornou uma interessante opção para o *codesign* de hardware e software, e também para o desenvolvimento de sistemas tempo-real embarcados. Entretanto, a UML pura tem carência de construções e abstrações para representar conceitos específicos do domínio dos sistemas tempo-real e embarcados. Neste contexto, faz-se necessária a utilização de algum perfil, ou do inglês, *profile*, que estenda a UML com novas estruturas e conceitos, permitindo a modelagem dos requisitos desejados. No caso do presente trabalho, utiliza-se o perfil *Modeling and*

Analysis of Real-Time and Embedded systems (MARTE), para permitir a correta modelagem de sistemas tempo-real embarcados.

2.5.1 Perfil MARTE

Esta especificação de um perfil UML adiciona capacidades à UML para o desenvolvimento dirigido por modelos (*Model-Driven Development* - MDD) para sistemas de tempo real e embarcados. Esta extensão, chamada perfil UML para Modelagem e Análise de sistemas tempo-real e embarcados, do inglês MARTE (ou somente MARTE) (OMG MARTE, 2010), fornece suporte para a especificação, o projeto, e para os estágios de verificação/validação do desenvolvimento de sistemas. Este perfil, ainda em finalização, substitui o perfil existente, UML *profile for Schedulability, Performance, and Time* (SPT), que de acordo com as experiências de Gérard e Selic (2008), revelou falhas e pouca capacidade de modelar com expressividade sistemas de tempo-real e embarcados.

MARTE consiste em definir fundamentos para descrição baseada em modelos de sistemas de tempo-real e embarcados. Estes conceitos principais são então refinados provendo especial fundamentação para os interesses de modelagem e análise de sistemas. As partes relacionadas à modelagem proporcionam suporte ao projeto de sistemas tempo-real e embarcados, desde a especificação até o detalhamento das características do sistema. MARTE proporciona também análise baseada em modelos. Onde a intenção não é definir novas técnicas de análise de sistemas tempo-real embarcados, mas fornecer suporte às técnicas existentes. Assim, ele provê facilidades para descrever modelos com informações requeridas para execução de análises específicas. MARTE está especialmente focado na análise de desempenho e de escalonabilidade. Mas ele também define um *framework* genérico para que análises quantitativas sejam feitas a partir de outros tipos de análises.

Entre outras, as vantagens de usar o perfil MARTE são que ele proporciona uma maneira comum de modelagem tanto para o hardware quanto para o software dos aspectos de sistemas de tempo-real e embarcados, permitindo um maior entendimento entre os desenvolvedores de ambas as partes. Habilita a correspondência entre ferramentas de desenvolvimento usadas para especificação, projeto, verificação, geração de código, etc. Além de favorecer a construção de modelos que podem ser usados para fazer previsões quantitativas a respeito de características de tempo-real e embarcada de sistemas, levando em consideração características de hardware e software.

O perfil foi estruturado para lidar com dois interesses: modelagem de tempo real e de características de sistemas embarcados, e para suportar a análise de propriedades de sistema. São representados através dos pacotes MARTE *Design Model* e MARTE *Analysis Model* na figura 2.4. Estes dois pacotes compartilham conceitos comuns com a descrição de tempo e o uso de recursos concorrentes, que estão contidos no pacote MARTE *Foundations*. O quarto pacote contém perfis anexos a Modelagem e Análise de sistemas de tempo-real e embarcados, e também modelos predefinidos de bibliotecas que podem ser utilizadas pelos desenvolvedores para simbolizar suas aplicações tempo-real e embarcadas.

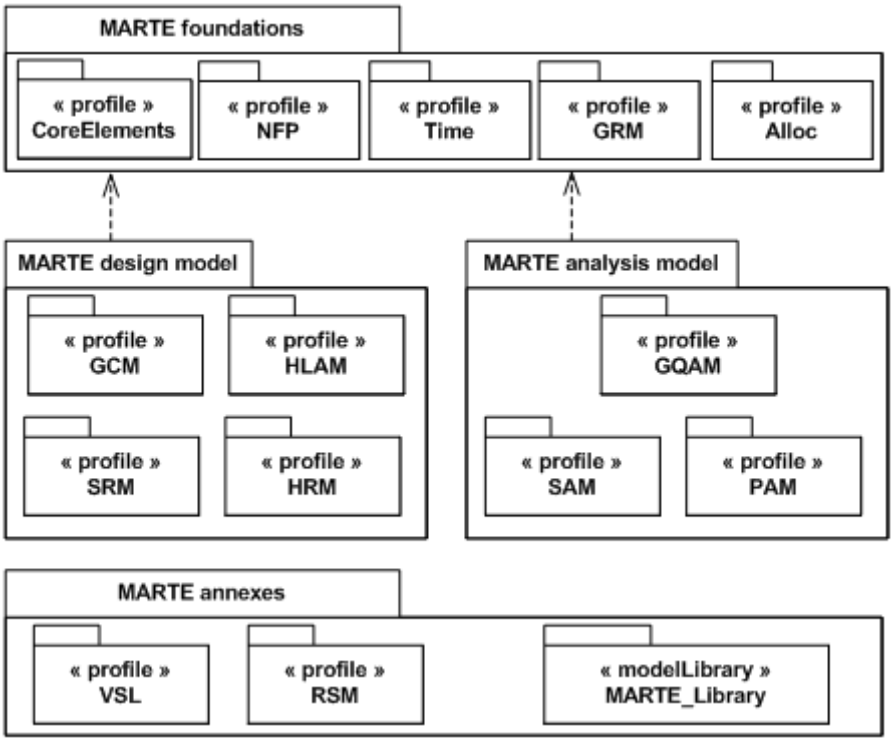

Figura 2.4: Arquitetura do perfil MARTE – Beta 3 (OMG MARTE, 2010).

Onde a descrição dos itens acima é a seguinte: **NFPs** = Non-Functional Properties, **HLAM =** High-Level Application Modeling, **GRM** = Generic Resource Modeling, **GCM** = Generic Component Model, **Alloc** = Allocation modeling, **RTEMoCC** = RTE Model of Computation & Communication, **SRM** = Software Resource Modeling, **HRM** = Hardware Resource Modeling, **GQAM** = Generic Quantitative Analysis Modeling, **SAM** = Schedulability Analysis Modeling, **PAM** = Performance Analysis Modeling, **VSL** = Value Specification Language, **RSM** = Repetitive Structure Modeling.

2.6 Linguagem VHDL

VHDL ou "VHSIC *Hardware Description Language*" (Linguagem de descrição de hardware VHSIC "*Very High Speed Integrated Circuits*") é uma linguagem usada para facilitar o *design* (projeto/concepção) de circuitos digitais em FPGAs e ASICs.

Foi desenvolvida sob o comando do Departamento de Defesa dos Estados Unidos (DARPA), com o intuito de documentar o comportamento de ASICs, substituindo os complexos manuais e diagramas esquemáticos, que até aquele momento eram a única metodologia utilizada no projeto de circuitos (VHDL, 2010).

Serviu inicialmente aos propósitos de documentação do projeto VHSIC. Entretanto, nesta época buscava-se uma linguagem que facilitasse o projeto de um circuito; ou seja, a partir de uma descrição textual, um algoritmo, desenvolver o circuito, sem necessidade de especificar explicitamente as ligações entre componentes. A VHDL presta-se adequadamente a tais propósitos, podendo ser utilizada para as tarefas de documentação, descrição, síntese, simulação, teste, verificação formal e ainda compilação de software, em alguns casos.

Pode ser utilizada em conjunto com ferramentas que suportem o desenvolvimento, verificação e implementação de projetos digitais em vários níveis de abstração. Com placas especiais é possível descarregar o código gerado nessa linguagem em FPGAs (*field programmable gate array* ou matriz de portas programáveis). Assim, o comportamento do circuito pode ser testado diretamente no hardware ou nível de portas lógicas, fornecendo um meio de realmente testar um circuito antes de sua verdadeira prototipação.

Após o sucesso inicial do uso da VHDL, a sua definição foi posta em domínio público, o que levou a ser padronizada pelo IEEE (*Institute of Electrical and Electronic Engineers*) em 1987. O fato de ser padronizada e de domínio público ampliou ainda mais a sua utilização, novas alterações foram propostas, como é natural num processo de aprimoramento e a linguagem sofreu diversas revisões até a mais recente versão de 2008.

Sua sintaxe é baseada na linguagem Pascal e ADA. Descrições de hardware são normalmente feitas a partir de componentes/blocos de hardware que comunicam entre si, formando o sistema como um todo. Na descrição de um componente de hardware o código VHDL se divide em dois blocos principais. O primeiro deles é chamado *Entity*, onde são declaradas as portas do sistema. Essas portas são quem permite a conexão desde componente com o mundo externo, que podem ser outros componentes ou, por exemplo, chaves eletrônicas, teclados, LEDs, etc. O segundo bloco de código de uma descrição VHDL é o *Architecture*, onde é implementado o comportamento que o sistema deve ter. É o corpo do sistema, contendo a descrição do seu funcionamento em detalhes, onde são feitas as atribuições, operações, comparações, associações entre componentes, etc. Maiores informações sobre programação na linguagem VHDL podem ser obtidas em (PERRY, 2002). A figura 2.5 apresenta um exemplo de código VHDL que implementa uma porta lógica da função E (and).

```
1   -- Importa std_logic da IEEE library
2   library IEEE;
3   use IEEE.std_logic_1164.all;
4
5   -- Declara uma entidade
6   entity ANDGATE is
7      port (
8            IN1 : in std_logic;
9            IN2 : in std_logic;
10           OUT1: out std_logic);
11  end ANDGATE;
12
13  -- Declara a arquitetura do hardware
14  architecture RTL of ANDGATE is
15  Begin
16
17     OUT1 <= IN1 and IN2;
18
19  end RTL;
```

Figura 2.5: Código de uma porta lógica E (and) retirado da Wikipédia.

3 ANÁLISE DO ESTADO DA ARTE

Este capítulo discute trabalhos relacionados à geração automática de código fonte a partir de especificações de sistemas embarcados. Apresenta trabalhos e ferramentas comerciais que discutem a transformação entre linguagens, buscando identificar esforços já realizados na área para oferecer um maior entendimento da proposta desta dissertação que trata da conversão de UML para VHDL.

Mais especificamente, é apresentada uma análise geral envolvendo transformações que abordem as linguagens alvo deste trabalho (i.e. UML e VHDL) como meio de esclarecimento sobre o tema. O capítulo foi dividido em blocos que analisam transformações de especificações em alto nível para VHDL, de UML para qualquer linguagem e, como foco principal, de UML para VHDL. Cada bloco apresenta o conteúdo em ordem cronológica a fim de demonstrar a evolução de técnicas e processos.

3.1 Geração de código VHDL a partir de diferentes linguagens

A alta demanda do mercado consumidor de tecnologia vem exigindo que novos produtos sejam lançados cada vez mais rapidamente. Empresas que desejam manter ou conquistar novas fatias do mercado necessitam desenvolver produtos em menos tempo para agradar consumidores e se manterem competitivas perante suas concorrentes. Esse alto consumo tecnológico influenciou diretamente o desenvolvimento de hardware, que precisou ser acelerado em resposta à nova necessidade de produção. Plataformas de hardware começaram a ser desenvolvidas a partir das mesmas técnicas utilizadas para engenharia de software, em busca de um processo de desenvolvimento mais rápido e intuitivo. Assim, ferramentas que conduziam o processo de desenvolvimento de software passaram também a conduzir processos de hardware, empregando técnicas de abstração de dados que escondem detalhes da complexidade dos sistemas até a fase de implementação.

Neste modelo de desenvolvimento o projeto é concebido como um conjunto de componentes, sem definição alguma do que é hardware ou software. A distinção entre componentes de hardware ou de software ocorre quando o sistema é validado para implementação. Ponto a partir do qual existe a real necessidade de uma implementação rápida do hardware. Pois, é desejável que o desenvolvimento do hardware acompanhe o desenvolvimento do software para que a compatibilidade entre ambos seja realizada ao longo do projeto e não somente no final, dando assim maior agilidade ao projeto.

Assim, buscando acelerar o desenvolvimento de hardware, surgiram ferramentas que permitem automatizar o processo de descrição de hardware através de linguagens HDL.

Essas ferramentas realizam todas as etapas de transformação dos componentes de hardware, especificados em modelos de sistema com alto nível de abstração de dados, em código fonte da linguagem HDL escolhida (i.e. VHDL ou Verilog). Desta forma, o hardware pode ser validado rapidamente através de simulações comportamentais antes da sua prototipação. Fato que acelera o processo de produção e diminui o retrabalho, reduzindo consideravelmente o custo do projeto como um todo.

Os trabalhos desta seção foram classificados em duas categorias principais, de acordo com o tipo da especificação inicial. Especificação inicial é aquela que modela a estrutura e comportamento do sistema, podendo ser desenvolvida *a partir de linguagens de programação* ou *em nível de sistema*. Além destas duas categorias, existe uma terceira para representar as *ferramentas comerciais* que foram encontradas.

3.1.1 VHDL a partir de linguagens de programação

O trabalho descrito em (PARKINSON *et al.*, 1994) traz uma ferramenta capaz de converter código padrão na linguagem C para código VHDL. Algoritmos escritos em C são convertidos em código VHDL comportamental. A metodologia utilizada é baseada em uma ferramenta que faz uso de um algoritmo específico para identificar segmentos de código críticos para aplicação dentro do software. Estes segmentos são usados como base para um particionamento entre o que será considerado hardware ou software dentro da aplicação. O projeto do sistema é baseado no compilador GCC (*GNU Compiler Collection*) e centrado em torno das descrições YACC (*Yet Another Compiler Compiler*) da linguagem C. Assim, o código C passa por um processo de análise gramatical baseado no YACC e neste momento as construções da linguagem C são mapeadas para VHDL. O código VHDL é produzido como parte do pacote de síntese SYNT, que resulta num código chamado *SynVHDL*. O código produzido pelo pacote SYNT é um subconjunto, não cobrindo todas as estruturas da linguagem VHDL. Por conseguinte, os próprios autores do trabalho confirmam que essa característica limita a geração de código, mas afirmam que o subconjunto de estruturas da linguagem VHDL utilizado é suficiente para dar cobertura às estruturas da linguagem C mais importantes.

Em (FISCHER *et al.*, 1996) é proposto um ambiente baseado em ferramentas, que suporta o processo de projeto e implementação de protocolos de comunicação. A especificação do sistema é feita na linguagem *Estelle*. As partes de software são traduzidas para linguagem C enquanto as partes de hardware são mapeadas para linguagem VHDL. *Estelle* é uma linguagem formal baseada na teoria das máquinas de estado finito estendidas. É uma linguagem especialmente adaptada para descrição de protocolos de comunicação e sistemas distribuídos. Sua sintaxe é um super-conjunto da linguagem *Pascal*. A parte *Pascal* da linguagem é utilizada para descrever o processamento sequencial de dados. Sua semântica operacional torna mais fácil a derivação de simulações e implementações eficientes do que em outras linguagens formais. O processo da especificação até a implementação segue o seguinte princípio: primeiramente o projetista do protocolo de comunicação constrói uma especificação *Estelle* para refinar seus conceitos, validar suas ideias e analisar as propriedades do protocolo sendo projetado. Essa especificação é muito útil para testes. O responsável por implementar o protocolo extrai dessa especificação descrições sobre os elementos distribuídos do sistema e a partir delas, divide o sistema em partes para serem implementadas em hardware e em software. Subsequentemente, serão geradas descrições de unidades de computação e comunicação para as partes de hardware e de software. Finalmente, são adicionados os detalhes de implementação e concretizada a

implementação através de compiladores C e de ferramentas de síntese em alto nível para o código VHDL. O ambiente de desenvolvimento EDT (*Estelle Development Toolset*) dá suporte a especificação e simulação em linguagem *Estelle*. O software é gerado diretamente a partir deste mesmo ambiente. Já o hardware, utiliza um tradutor *Estelle-to-VHDL* chamado *e2v*, desenvolvido por estes autores em trabalhos anteriores.

O trabalho desenvolvido por (DAVEAU *et al.*, 1997) consiste em uma metodologia para gerar código VHDL a partir de uma linguagem de especificação e descrição chamada SDL (do inglês, *Specification and Description Language*). Ela é capaz de modelar e simular sistemas de tempo-real, distribuídos e de telecomunicações. Uma descrição em SDL é baseada em processos concorrentes que se comunicam entre si através de sinais. Essa linguagem tem dois formatos, textual e gráfico. Este trabalho é baseado no ambiente COSMOS (*Component based System Modeler and Simulator*) que permite a geração de código C e VHDL a partir de especificações SDL. O processo de coprojeto a partir deste ambiente começa com uma especificação funcional que é traduzida num modelo intermediário. Seus próximos passos são: particionamento, síntese de comunicação e geração da arquitetura. O particionamento decompõe a especificação inicial em processos abstratos (i. e. partições) que podem ser transcritos na arquitetura alvo como partes de hardware (i. e. ASIC, FPGA), como partes de software (i. e. processador + código) e como módulos de comunicação (i. e. FIFO, memórias, interrupções). O passo de síntese de comunicação objetiva fixar protocolos e interfaces que serão usadas pelo subsistema de comunicação. Para tanto utiliza modelos de comunicação existentes na linguagem de especificação. O passo de geração da arquitetura produz descrições C ou VHDL para cada processo abstrato gerado no passo de particionamento. Entre cada passo, existem inúmeros outros passos de refinamento. Segundo os autores, esta metodologia é utilizada para aproximar o *gap* existente entre os conceitos das duas linguagens, tornando possível o correto mapeamento entre elas. Além disso, essa metodologia cobre um grande subconjunto da linguagem SDL, sendo capaz de gerar código VHDL para simulação e síntese de alto nível a partir de ferramentas comerciais.

Analisando a geração de código VHDL a partir de linguagens de programação, um trabalho interessante é apresentado em (FLEISCHMANN *et al.*, 1998). Neste trabalho os autores tinham uma visão futurística da área de sistemas embarcados, apostando que no futuro os sistemas necessitariam apresentar características adaptativas e reconfiguráveis. Assim, desenvolveram um novo conceito para exploração do espaço de projetos assistidos por ferramentas e para a rápida prototipação de sistemas de hardware e software. Propõem que a especificação dos sistemas seja feita na linguagem Java, por esta ser *multi-thread* e assim suportar a descrição de sistemas como um conjunto de comportamentos concorrentes. A questão chave é a decisão de quais partes serão implementadas em software ou em hardware. Decisão que fica a cargo do projetista. Com isto decidido, *bytecodes* são gerados para a parte de software, que será executada sobre uma *Java Virtual Machine* (JVM) e código VHDL é gerado para parte de hardware, que será sintetizado e mapeado para a plataforma alvo de um FPGA reconfigurável. É gerado código VHDL sintetizável para os métodos Java que foram previamente selecionados como parte do hardware, os quais devem ser baseados em uma biblioteca específica do *framework* que permite abstrações do hardware na linguagem Java. Então, o projeto é dividido em um caminho de dados e outro de controle. Expressões e comandos são convertidos em construções VHDL correspondentes para a descrição do caminho de dados. Construções como *loop*s e

construções condicionais, são transformadas em representações na forma de máquinas de estado (i. e. FSMs). Este trabalho não trata diretamente de questões como análise de requisitos e não tem uma metodologia dirigida por modelos para suportar a modelagem e geração de código, mas tem uma técnica interessante para permitir a exploração do espaço de projetos.

Um algoritmo para traduzir especificações de sistema feitas na linguagem SystemC para VHDL é proposto em (CÔTÉ; ZILIC, 2002). Este trabalho explica de uma maneira bastante expressiva os ganhos obtidos a partir do coprojeto de hardware e software. O objetivo principal foi desenvolver um algoritmo de tradução entre SystemC e VHDL, sem resultar em muitas restrições ao desenvolvedor no momento da especificação do sistema. O trabalho foi facilitado pelo fato de grande número das construções SystemC serem mapeadas diretamente para VHDL. É possível mapear diretamente quase todas as construções da linguagem SystemC, com exceção das construções *loops* e *waits*. Para estas construções o algoritmo cria estruturas VHDL equivalentes e sintetizáveis, capazes de realizar a mesma função daquelas em SystemC. Assim, o algoritmo de tradução desenvolvido tem uma fase inicial de análise gramatical (i. e. *parsing*) do código dado como especificação do sistema. Neste primeiro passo o processo de tradução consiste em ler as descrições SystemC e gravá-las na memória, na forma de uma lista hierárquica. Isto faz-se necessário, pois decisões de tradução podem depender de outras estruturas encontradas ao longo do código, e traduzir linha após linha pode tornar impossível a geração do código alvo. Ainda nesta primeira fase, quando estruturas de *loop* e *wait* são identificadas, *flags* são ativados para sinalizar que tais construções existem no código e que devem ser tratadas. Então, a segunda fase só é ativada quando tais *flags* estão ativos e consiste em transformar as estruturas acima citadas em máquinas de estado finito, representando o mesmo comportamento das estruturas anteriores. O terceiro e último passo representa o efetivo processo de tradução entre as estruturas. Neste ponto estruturas complexas da linguagem SistemC são subdivididas em partes menores para tornar mais fácil a tradução para VHDL, Operações sobre múltiplas variáveis também são simplificadas e dependências verificadas para assegurar que sinais intermediários mantenham valores corretos. Os autores mostram por estudos de caso que conseguiram atingir o objetivo de gerar um código VHDL totalmente sintetizável e ainda citam alguns itens que ainda devem ser adicionados ao algoritmo tradutor para dar maior suporte as estruturas da linguagem de especificação, como por exemplo, o uso de ponteiros, ou funções e variáveis globais.

(GUO *et al.*, 2008) apresenta parte do framework ROCCC, uma ferramenta que realiza transformações de C/C++ e Fortran para VHDL focado na otimização do código gerado. Na fase de entrada o compilador realiza análises de alto nível sobre o caminho de dados, buscando identificar lógicas que envolvam laços para realizar transformações que maximizem o paralelismo do código gerado e diminuam a área utilizada. Na fase final o compilador explora o paralelismo de baixo nível, tentando implementar pipelines no caminho de dados e reduzir o tamanho de bits dos sinais internos. Como ponto chave do trabalho é apresentado um componente inteligente que reutiliza dados de entrada para iterações de loops adjacentes, chamado *Smart* Buffer. Este componente é responsável por aumentar significativamente a desempenho do circuito gerado, através da simplificação do loop de controle. O trabalho não fornece maiores detalhes sobre o código gerado, mas detalha através de exemplos e comparações dos resultados os processos utilizados.

3.1.2 VHDL a partir de especificações em nível de sistema

Em (NARAYAN *et al.*, 1991) é apresentado um tradutor VHDL a partir da linguagem de especificação de sistemas *SpecCharts*. Essa linguagem foi desenvolvida anteriormente pelos próprios autores do trabalho. Ela é direcionada para especificações em nível de sistema e tem suas principais construções baseadas em diagramas de estado com hierarquia e concorrência. Nela, um sistema é especificado através de seu comportamento na forma de estados. Dentro de cada estado sua funcionalidade pode ser descrita por comandos VHDL sequenciais, estados concorrentes ou subestados sequenciais. Ela tem abstrações para diversos tipos de construções necessárias à modelagem de sistemas, como por exemplo, estados hierárquicos e estados sequenciais, transição de eventos assíncronos, estado de conclusão, variáveis globais, sinais globais, etc. A linguagem *SpecCharts* tem ainda duas poderosas abstrações: transferência de dados baseada em protocolo e arbitrariedade de sinais. A primeira permite a transferência de dados ser especificada usando protocolos em vez de ser detalhada em baixo nível. A segunda proporciona uma concisa representação do acesso arbitrário a recursos compartilhados. A ferramenta tradutora foi desenvolvida na linguagem C para ser executada dentro de ambiente UNIX. Ela converte um arquivo texto oriundo da especificação do sistema feito na linguagem *SpecCharts* para um arquivo VHDL. Esse arquivo possui uma única entidade e processos relativos ao comportamento do sistema. O mapeamento entre a especificação *SpecChart* e o código VHDL é praticamente direto, visto que o código que descreve o funcionamento dos estados é escrito em VHDL. Assim, é necessária apenas a adição de abstrações referentes ao posicionamento destes blocos de código dentro da entidade VHDL que representa o sistema. Para simular o código gerado, é criado de forma automática, outro arquivo VHDL onde é instanciada como um componente aquela entidade criada para o sistema. Este arquivo então é simulado para avaliar e validar o comportamento do sistema.

A especificação de unidades de controle de sistemas digitais é facilitada se for feita de forma gráfica, pois os caminhos que levam ao resultado desejado se tornam mais instintivos e claros. Especificações gráficas desse tipo de sistema normalmente são baseadas em máquinas de estado finito (FSM). Entretanto, o trabalho de Fernandes *et al.* (1997) considera que esse tipo de especificação não é o mais adequado para sistemas que apresentam atividades paralelas. Para Fernandes, Redes de Petri (do inglês, *Petri Nets* - PN) proporcionam um meio adequado para modelar e animar sistemas paralelos baseados em uma abordagem de caminhos de controle e de dados, de uma forma hierarquicamente estruturada. Um conjunto de ferramentas na forma de um *framework* em software foi desenvolvido para dar suporte a este trabalho. Este *framework* recebe como entrada especificações PN de controladores e realiza a validação dessas propriedades especificadas para este controlador, permitindo assim, a animação do modelo PN do mesmo e a geração de código VHDL em nível de registradores (*Register-Transfer Level* - RTL) para o comportamento deste sistema. Neste trabalho, a especificação é feita somente para o comportamento do sistema, de modo que requisitos de sistema são desconsiderados. A metodologia utilizada sugere duas transformações diretas da especificação até a geração do código. Primeiramente a especificação de entrada feita em PN é mapeada para notações da linguagem ConPar, que representa tais especificações como equações booleanas. ConPar é uma linguagem desenvolvida no âmbito do referido trabalho para especificação de Controladores Paralelos. Num segundo momento, essa representação do sistema na linguagem ConPar é traduzida para código VHDL. Os autores ainda salientam que não há necessidade da validação formal do

código VHDL gerado, pois a especificação PN do sistema é validada pelo framework utilizado, e porque existe uma correspondência direta entre a descrição do sistema feita na linguagem ConPar e o código VHDL gerado.

Um *framework* para geração de código VHDL sintetizável e para *testbenchs* é apresentado em (FILIBA *et al.*, 2006). Este trabalho propõe um ambiente melhorado para projeto e modelagem visual do processamento de sinais tempo-real baseados em FPGA. Este ambiente é baseado no *framework* Ptolemy II (PTII), desenvolvido para o estudo da modelagem, simulação e projeto concorrente de sistemas tempo-real (DAVIS II *et al.*, 1999). O *framework* PTII atualmente gera código para linguagem C e utiliza um mecanismo baseado em ajudantes (i.e. *helpers*), onde um conjunto destes componentes é utilizado para gerar código para os atores presentes no modelo PTII. Cada *helper* tem correspondência direta com um ator. Contudo, existe uma separação lógica entre os domínios de relevância, onde *helpers* são componentes para geração de código, enquanto atores são componentes somente de simulação. Componentes para geração de código possuem *templates* de código e macros escritos na linguagem alvo. A linguagem de macros permite que os blocos de código sejam parametrizados segundo as configurações específicas do componente que está sendo modelado. A metodologia do processo gerador de código é baseada na interação do núcleo responsável pela geração de código (i. e. *kernel*) com o conjunto de *helpers*. O núcleo une todos os blocos de código produzidos pelos *helpers*, gerando assim o código completo para o comportamento do sistema modelado. É gerado código VHDL em nível de registradores (RTL), que é independente da plataforma de implementação, podendo ser utilizado tanto em FPGAs quanto em ASICs. PTII ainda fornece um ator para teste que verifica se a saída da simulação é consistente com a saída do bloco de hardware gerado. Assim proporcionam um meio para validação do código gerado.

O trabalho desenvolvido em (TRANCHERO; REYNERI, 2007) introduz uma metodologia para trabalhar com sistemas assíncronos no desenvolvimento de sistemas embarcados. Foi desenvolvida no âmbito deste trabalho uma ferramenta chamada CodeSimulink, que recebe como entrada especificações de sistemas feitas em Simulink® e a partir delas é capaz de gerar código para linguagem VHDL (usando um compilador interno) e para linguagem C (através do *MathWorks' Real-Time Workshop compiler*, ferramenta comercial fornecida pela MathWorks). A especificação de sistemas a partir do Simulink® gera modelos baseados em fluxo de dados (i. e. *data-flow*). A forma de modelar é totalmente gráfica, através de blocos de manipulação de dados e suas conexões. A ferramenta CodeSimulink é uma extensão do Simulink® capaz de trabalhar com circuitos assíncronos e gerar código VHDL para este tipo de sistema. CodeSimulink recebe como entrada uma especificação do sistema feita em Simulink® e utiliza bibliotecas próprias para acrescentar parâmetros assíncronos ao sistema e gerar código VHDL. Existe uma biblioteca que contém *templates* de implementações VHDL padrão para cada um dos blocos disponíveis no programa Simulink® para especificação de sistemas. Essas *templates* são modificadas de acordo com as características dos blocos que estão especificados para certo sistema e resultam em diversos blocos de código VHDL. Esses blocos de código VHDL são então unidos através de interfaces, pré-existentes e customizáveis, que também existem no conjunto de bibliotecas utilizado pelo CodeSimulink. Desta forma, ao final do processo, tem-se o código VHDL completo para representar o sistema em questão. Como validação do código os autores ainda desenvolveram um gerador automático de arquivos de *test-bench* (i. e. arquivos para simulação comportamental em alto-nível), que converte

simulações Simulink® de alto nível em arquivos *test-benches* para simulação de VHDL em baixo nível. Com isso, permitem que avaliações do comportamento do sistema sejam realizadas antes da efetiva geração do código VHDL final.

Outro trabalho que é interessante ser citado, foi desenvolvido em Paderborn na Alemanha por Pohl *et al.* (2010). Trata-se de uma biblioteca Java para leitura, manipulação e escrita de código VHDL chamada *VHDL Maniputation and Generation Interface* (vMAGIC). Essa biblioteca está disponível na internet e permite criar geradores automáticos de código VHDL através de sua API. Desenvolvedores podem utilizar as funções disponíveis nesta biblioteca para *parser, modification and generation of code* e *VHDL writer*, para criarem seus próprios geradores capazes de interpretar códigos existentes em outras linguagens e transformá-los numa representação interna chamada *Abstract Syntax Tree* (AST), para então, desta representação interna, gerar código VHDL. Os próprios autores dizem que a princípio a criação do modelo de algoritmo que será utilizado como *parser*, pode tomar mais tempo do desenvolvedor. Porém, num segundo momento esse modelo, se torna uma espécie de *template*, e poderá ser reutilizado diversas vezes para o mesmo propósito. Assim, resultando em maior agilidade para produção de código fonte VHDL.

3.1.3 VHDL a partir de ferramentas comerciais

Como o processo de geração automática de código para hardware ainda é um assunto relativamente novo, não existe no mercado uma grande quantidade de produtos comerciais destinados a este fim. Foram encontradas somente duas ferramentas totalmente comerciais destinadas especificamente à geração de código fonte HDL.

A primeira delas é a *StateCAD*[1] da Xilinx, uma ferramenta gráfica destinada ao projeto de sistemas digitais. Esta ferramenta permite expressar as ideias de uma forma natural. Utiliza um diagrama de bolhas (i. e. *bubble diagrams*), que é uma notação gráfica própria para representar diagramas de estado, sua forma de especificação de sistemas. Gera automaticamente, a partir de seus diagramas de estado, código HDL simulável e sintetizável (i. e. VHDL ou Verilog). A Xilinx é uma das pioneiras no desenvolvimento de ferramentas para simulação e síntese de código HDL em FPGAs. Assim, *StateCAD* está integrada no pacote *ISE Design Suite*, principal ferramenta de desenvolvimento da Xilinx.

A outra ferramenta encontrada é a Simulink HDL Coder[2] da MathWorks™. Essa ferramenta é capaz de gerar código VHDL, ou Verilog, sintetizável a partir de modelos simulink, diagramas de estados e código Matlab embarcado. Simulink HDL Coder também gera *scripts* para simulação e síntese permitindo assim que simulações e síntese sejam feitas rapidamente a partir do modelo desenvolvido.

3.2 Geração de código a partir de especificações UML

A grande quantidade de funcionalidades que um único sistema atual deve suportar aumenta consideravelmente a complexidade do seu projeto, requisitando ferramentas e técnicas para ajudar o desenvolvedor no gerenciamento dessa complexidade. Uma

[1] Xilinx Inc. www.xilinx.com

[2] The MathWorks Inc. www.mathworks.com

técnica comumente utilizada é elevar o nível de abstração da linguagem na qual a especificação do sistema é desenvolvida (SANGIOVANNI-VINCENTELLI, 2003). Um método para elevar o nível de abstração no coprojeto de sistemas embarcados é a utilização de uma linguagem de projeto gráfica, padronizada e de alto-nível, que inclua noções de orientação a objetos e permita mapear seus modelos para linguagens tradicionais de descrição de hardware (VHDL, Verilog) e/ou de descrição de software embarcado (C, Java).

A UML vinha sendo largamente utilizada para o desenvolvimento de sistemas de software (LONG *et al.*, 2005). Entretanto, mostrou-se ampla o suficiente para servir também ao projeto de hardware. Por isso, vem sendo utilizada como linguagem padrão para o desenvolvimento de sistemas complexos, permitindo o coprojeto de hardware e software em alto nível. Adicionalmente, sua utilização vem sendo investigada em metodologias que envolvem o desenvolvimento dirigido por modelos (do inglês, *Model-Driven Development* - MDD) no projeto de sistemas embarcados. Por ser uma linguagem gráfica tem uma vantagem clara sobre outras formas de especificação. Torna o projeto mais intuitivo pela representação gráfica de estruturas e componentes que compõem um sistema, diminuindo a complexidade do processo de desenvolvimento.

Em um processo de desenvolvimento baseado em UML, diferentes tipos de construções da linguagem podem ser utilizados para representar e analisar os artefatos criados na fase de especificação do sistema: diagrama de classe para análise estática (visão estática), máquinas de estado para especificação comportamental dinâmica e validação (visão comportamental), diagrama de sequência e colaboração para representar a interação entre objetos (interação), OCL (do inglês, *Object Constraint Language*) para especificação formal de funcionalidades e restrições de objetos (visão funcional), etc. Essas múltiplas possibilidades de visões oferecidas pela UML à modelagem de um sistema têm inúmeras vantagens, mas também desvantagens. Cada uma dessas visões foca em um diferente aspecto do sistema, permitindo que análises e entendimentos de várias características do projeto sejam feitas separadamente a partir de cada uma delas. Adicionalmente, baseado nestas visões o projetista pode decompor o sistema em porções de tamanhos mais facilmente gerenciáveis. Essa característica é importante para ferramentas de desenvolvimento e ferramentas para geração de código. No entanto, um modelo baseado em múltiplas visões normalmente tem problemas de consistência, causado por diferenças na especificação de um mesmo item. Diferentes modelos podem descrever um mesmo aspecto do sistema de forma diferente. Isso implica em problemas de consistência da informação e assim, complica o desenvolvimento de ferramentas para geração de código.

Assim, esta seção aborda a linguagem UML, observando como ela vem sendo utilizada para especificação de modelos e identificando as técnicas que guiam o processo que vai da modelagem à geração de código. Como o uso desta linguagem foi bastante difundido, ela é utilizada como base para processos de geração de código direcionados a diversas linguagens. Desta forma, diferentes trabalhos foram escolhidos para serem apresentados nesta seção, buscando uma análise ampla sobre o estado da arte deste tema.

Em (HECHT *et al.*, 2006) é apresentado um trabalho que propõe a geração de código na linguagem *AspectJ* a partir de modelos UML decorados com o perfil *Theme/UML*. *AspectJ* é uma linguagem orientada a aspectos bastante utilizada na atualidade. Ela permite uma separação explicita de interesses (*concerns*) que afetam múltiplas partes do sistema de software, modelando e implementando cada um destes

aspectos independentemente dos outros e dos principais interesses envolvidos na modelagem do sistema, proporcionando assim, melhoria na qualidade de muitos fatores do software, tais como extensibilidade, manutenibilidade e reusabilidade. O perfil *Theme* adiciona à UML uma construção para lidar com *composition pattern* (em português, padrão de composição), uma definição de como integrar artefatos de modelagem (classes e métodos) de dois diferentes pacotes. Essa estrutura trabalha de uma maneira similar aos modelos customizáveis do UML (i. e. *templates*), permitindo que elementos de um modelo não sejam totalmente definidos, possuindo parâmetros que possibilitam sua posterior modificação. No caso do *Theme/UML*, essas *templates* são utilizadas para acrescentar código extra aos objetos modelados que forem influenciados por aspectos. Para gerar código os autores extraem do modelo um arquivo XMI (*XML Metadata Interchange*) (OMG XMI, 2010) contendo dados do modelo UML num formato de arquivo XML (*eXtensible Markup Language*) (W3C, 2011). Então, alteram esse arquivo XMI adicionando algumas diretrizes correspondentes a orientação a aspectos. Em seguida o utilizam como arquivo de entrada num mecanismo gerador de código desenvolvido através da linguagem XSLT (*Extensible Stylesheet Language Transformations*). Esse gerador de código é baseado em um algoritmo que lê o arquivo de entrada e mapeia os dados correspondentes de forma direta para código *AspectJ*. Os autores ainda salientam que poderiam ter utilizado uma conversão de modelos intermediária antes de gerarem o código, caracterizando uma metodologia dirigida por modelos. Porém, acrescentam que esse passo extra poderia provocar problemas na manutenção do aplicativo, uma vez que mudanças na linguagem *AspectJ* exigiriam modificações também neste modelo intermediário, ao invés de mudanças somente nas regras de mapeamento, como pretendido por eles.

(NASCIMENTO *et al.*, 2006) desenvolveu um repositório de metadados UML/MOF (OMG MOF, 2011), que permite o armazenamento e a manipulação de informações de modelos de aplicação e plataforma, usados no projeto e desenvolvimento de sistemas embarcados. O metamodelo do repositório permite a representação de diferentes tipos de linguagens de especificação, tais como UML e Simulink, e ainda, diferentes tipos de infraestrutura para implementação de linguagens alvo como, por exemplo, Java e C. Adicionalmente, este repositório ainda pode armazenar informações sobre regras de mapeamento para transformar modelos de aplicação em modelos de plataforma. Ferramentas de geração de código que automatizam e facilitam o processo de desenvolvimento de sistemas, podem utilizar os modelos de implementação fornecidos pelo repositório para fins de mapeamento. A metodologia deste trabalho é totalmente baseada em arquiteturas dirigidas por modelos (MDA). Criada pela OMG essa metodologia define o processo de desenvolvimento de software baseado em transformações entre modelos. Este trabalho permite que sistemas sejam especificados em diferentes linguagens, propondo que estas especificações sejam transformadas, através de específicos metamodelos de transformação, em um metamodelo padrão para representação de aplicações. Da mesma forma, as linguagens alvo para geração de código, também seriam modeladas por meio de transformações entre modelos dedicados, para um modelo padrão para representação de plataformas. Estes modelos de plataforma padrão, junto com modelos de regras de mapeamento ficariam armazenados no repositório para serem utilizados quando necessário. Os modelos que representam as regras de mapeamento contêm *templates* de código fonte das linguagens alvo, que no momento da geração de código são preenchidos por dados oriundos daquele metamodelo padrão de representação da aplicação. Assim, essas *templates* são configuradas e preenchidas com dados do sistema que foi especificado, gerando assim

outro modelo padrão de implementação, que servirá diretamente para geração de código.

(RICCOBENE *et al.*, 2006) apresenta um ambiente para codesenvolvimento de hardware e software de sistemas embarcados baseado em UML e SystemC. Este propõe uma metodologia para o projeto de SoCs dirigido por modelos e o protótipo de uma ferramenta para dar suporte à geração de código para C/C++/SystemC a partir de tais modelos. Além disso, a ferramenta proposta ainda permite um processo de engenharia reversa de código para modelos. Como suporte a metodologia é utilizada a ferramenta comercial *Enterprise Architect* (EA) da Sparx Systems[3]. Um *plugin* foi criado para essa ferramenta que atua como motor de todo o processo de geração de código. O processo de engenharia reversa se dá a partir de outro componente, composto por um *parser*, uma estrutura de dados e um XMI *writer*. Este componente importa códigos C/C++ e SystemC, transformando-os em construções da linguagem UML representando o código segundo sua metodologia. São apresentados também três estudos de caso onde as funcionalidades de geração de código e engenharia reversa foram testadas. Além do ambiente e metodologia, este trabalho identifica três categorias possíveis para geração de código, sendo elas: geração somente do esqueleto, geração parcial e geração total. Na primeira, somente a parte estrutural é gerada a partir das informações do modelo, similarmente ao próprio esqueleto de código do sistema. A geração parcial envolve geração de código para os comportamentos, porém ainda não de modo completo. Os autores afirmam que seu trabalho está classificado como geração completa. Explicam também, que este grupo (geração total) se diferencia de geração parcial, pois utiliza em nível de PIM uma nova linguagem de ação, que á baseada numa semântica de ação e possui um metamodelo que facilita a geração de código para comportamento dinâmico. Porém, adotam linguagem de ação para C/C++/SystemC somente no nível de PSM, afirmando que desta forma, mesmo se adotar qualquer linguagem de ação em nível de PIM, conseguem obter geração total de código. Finalizam o trabalho sem informar detalhes sobre o código resultante, nem tão pouco, como ele é testado e validado.

Em (ANDERSSON; HÖST, 2008) é feita uma comparação entre as linguagens UML e SystemC e são propostas regras de mapeamento para a automática conversão entre os modelos. Neste trabalho foi desenvolvido o protótipo de uma ferramenta para gerenciar a transformação e a geração automática de código SystemC a partir de modelos UML. Essa ferramenta protótipo é um *plug-in* para ferramenta comercial *Telelogic-TAU UML2 Modelling Tool*. Contém uma metodologia que propõe três passos para geração de código. Primeiramente, o modelo UML desenvolvido para o sistema é refinado para um modelo UML decorado com estruturas do perfil *UML profile for SystemC*. Como sequência desde primeiro passo, o modelo UML decorado com o perfil SystemC é refinado novamente para um modelo UML que inclui construções com representação direta na linguagem SystemC. Estas duas primeiras etapas são manuais. O terceiro passo corresponde à transformação direta entre o modelo UML resultante do segundo passo e o código SystemC. Este passo é implementado usando como suporte o gerador de código C++ da Telelogic. Após, um *agent* desenvolvido neste trabalho é aplicado ao código gerado, para transformá-lo de acordo com as macros SystemC. Assim, tal código é transformado de simples declarações de classes C++ para módulos SystemC. O ponto chave deste trabalho está na representação em SystemC das estruturas de comunicação UML. Como essa estrutura é representada de forma diferente

[3] Sparx Systems Pty Ltd. www.sparxsystems.com

nas duas linguagens, foi preciso desenvolver modelos em SystemC (i. e. *templates*) para representar as estruturas de comunicação dos modelos UML.

O trabalho de (VIDAL *et al.*, 2009) apresenta a metodologia MoPCoM que utiliza uma proposta baseada em UML/MDA para projetar sistemas tempo-real embarcados de alta qualidade. Eles definem um conjunto de regras para construção de modelos UML, a partir dos quais suas ferramentas permitem a geração de código para plataforma de SW em linguagem C embarcado e código para plataforma de HW em VHDL. A metodologia utilizada define três níveis de abstração: abstrato, execução e detalhado. O nível detalhado é apresentado neste trabalho, pois a intenção dos autores é demonstrar o uso das regras de projeto desenvolvidas para a geração automática de código VHDL. Seu processo é inicialmente dividido em modelo de aplicação e de plataforma. O modelo de aplicação utiliza diagramas de classe, de estrutura composta, de objetos e de máquinas de estado para especificar a estrutura estática e dinâmica do sistema modelado. O modelo de plataforma é desenvolvido sobre o diagrama de componentes, sendo responsável por definir itens específicos da plataforma que será implementada, através de elementos do sub-*profile* MARTE HRM (*Hardware Resource Modeling*). Após, um modelo de alocação é desenvolvido a partir dos modelos anteriores, tendo como principal função ligar as funcionalidades descritas no modelo de aplicação nos respectivos componentes do modelo de plataforma. A metodologia MoPCoM utiliza como base para desenvolvimento dos modelos a ferramenta *Rhapsody*[4] e a ferramenta de transformação *MDworkbench*[5], largamente utilizada na indústria. Maiores detalhes sobre o código gerado ou sobre regras de mapeamento e transformações entre modelos não foram descritos.

Por último nesta seção, temos um trabalho bastante recente de (CHARFI *et al.*, 2010) que traz novos conceitos relacionados ao tratamento dos modelos UML e geração de código focado em otimização. Apresenta um desenvolvimento baseado em modelos (*Model-Based Development* - MBD) como forma de adicionar um novo nível de abstração chamado modelo. Este nível de abstração adicional fornece à modelagem ferramentas dedicadas que permitem automatizar o tratamento do que está sendo modelado. Como por exemplo, síntese de aplicações do sistema através da geração automática de código fonte. Mais especificamente os autores discutem questões de otimização sobre MBD para Sistemas tempo-real e embarcados (*Real*-time and Embedded Systems - RTES). Propõem técnicas para modelagem com maior nível de abstração, realizam a geração de código para C++ a partir de diagramas de estado e, compilam e otimizam o código gerado usando o bem conhecido compilador *open source – Gnu Compiller Collection* (GCC), adicionado a novas abordagens e padrões para otimização de códigos. Agora focando em geração automática de código, este trabalho implementa máquinas de estado utilizando três padrões diferentes: *State Pattern, State Table Transition e Nested Switch Case Statemens*, para testar suas técnicas de otimização. Aplicam técnicas de otimização relativas a cada padrão sobre um código gerado para o mesmo problema (i.e. mesmo código). Assim, apresentam como resultados experimentais uma tabela com as taxas de otimização obtidas para cada caso, comprovando os ganhos que podem ser obtidos a partir da utilização de tais técnicas e distinta abordagem de modelagem. Não relatam maiores detalhes referentes ao processo de geração de código utilizado. Contudo, abordam o assunto como parte de

[4] Rhapsody UML Modeller - http://www.telelogic.com/products/rhapsody,from Telelogic, an IBM company

[5] Sodius / MDWorkbench - http://sodius.com/mdworkbench

um processo maior que visa melhorar o código obtido como resultado de um fluxo que utiliza as melhores técnicas de modelagem e geração de código.

3.3 Geração de código VHDL a partir de especificações UML

Após estudos demonstrarem que a UML também poderia ser aplicável ao coprojeto de hardware e software, surgiram inúmeros trabalhos científicos que discutem melhores formatos e metodologias para cobrir o *gap* semântico existente entre a especificação de um sistema em alto-nível e a posterior codificação de seus componentes de hardware.

A geração de código para software a partir de especificações UML torna-se mais simples pela existência de farta documentação sobre o assunto, visto que a utilização desta linguagem na área de engenharia de software não é novidade. A literatura conta com diversos trabalhos, metodologias e ferramentas abordando conversões de UML para software. Contudo, a conversão de especificações de hardware de alto para baixo nível, ainda é um grande desafio a ser superado. Propostas diversas discutem formas de fazer essa conversão entre uma linguagem gráfica e uma descrição em nível de portas lógicas. O objetivo é encontrar um meio para superar da melhor forma possível o salto semântico existente para tradução destes componentes de hardware. Entretanto, apesar dos esforços, a comunidade acadêmica ainda não pode chegar num consenso sobre padrões para geração de hardware a partir de especificações em alto nível.

Atualmente, existem esforços de pesquisa voltados principalmente para geração de código às linguagens SystemC e VHDL. Para SystemC já foram desenvolvidos diversos *profiles* que permitem decorar estruturas da linguagem UML com marcações (estereótipos) para guiar o processo de geração de código naquela linguagem. Contudo, para linguagem VHDL ainda não existem *profiles* que funcionem do mesmo modo. Além disso, a linguagem VHDL é estruturada, diferindo totalmente do SystemC, que é orientado a objetos da mesma forma que o UML, tornando assim, o salto semântico a ser dado neste caso, maior ainda.

Desta forma, descreveremos a seguir alguns trabalhos que abordam a geração de código na forma de descrições de hardware, tentando exemplificar e mostrar o estado da arte atual deste tema. Fazendo comparações com nossa abordagem e identificando tópicos interessantes destes trabalhos que devam ser ressaltados.

Em (McUMBER; CHENG, 1999) temos um trabalho bastante antigo, mas que traz conceitos importantíssimos referentes à geração de código VHDL a partir de UML. Nele foi desenvolvido um *framework* capaz de derivar especificações de hardware na linguagem VHDL a partir de classes e diagramas de estado da linguagem UML. Este *framework* captura a estrutura e o comportamento do sistema embarcado modelado buscando manter o comprometimento entre as linguagens em questão. Os autores consideram que para modelagem e projeto de sistemas embarcados em alto-nível é suficiente apenas um subconjunto da linguagem UML. Assim, utilizaram somente diagramas de classes e diagramas de estado como subconjunto das notações UML a serem mapeadas para VHDL. Regras formais que habilitam técnicas para automatizar o processo de geração de especificações de hardware foram criadas baseadas somente em tais notações. Essas técnicas proporcionam um mapeamento homomórfico (*1-to-1*) entre o metamodelo da linguagem informal (*source*) e formal (*target*). Essas relações homomórficas permitem consistência às regras de formalização, sendo que os autores

creem que a transformação de um modelo semiformal, tal como os modelos dinâmicos do UML, para um modelo formal, da linguagem alvo, é questão somente de ser fornecida uma semântica precisa e de um processo de mapeamento rigoroso entre as linguagens. Os autores deixam claro que sua preocupação não está focada em otimização, mas sim na obtenção de um código em condições de ser simulado comportamentalmente. Apresentam a teoria completa sobre o conjunto de regras de mapeamento desenvolvido e, além disso, trazem um estudo de caso demonstrando a validação do comportamento simulado. Apesar de limitado a geração de código VHDL para somente um subconjunto da linguagem UML, o código gerado é bastante completo segundo exemplos. Nenhuma comprovação ou maiores detalhes sobre a porcentagem de código gerado é apresentada.

(BJÖRKLUND; LILIUS, 2002) detalhou em passos uma metodologia para a tradução de UML para VHDL. Primeiramente, o modelo UML com descrições comportamentais, descritas na forma de diagramas de estado e diagrama de colaboração, é traduzido em código textual na linguagem SMDL (*State Machine Description Language*). Essa é uma linguagem de alto-nível para múltiplos modelos de computação que, posteriormente, será traduzida (simplificada) para uma máquina de estados finita em baixo-nível, podendo então, ser compilada para código na linguagem alvo. A tradução do modelo UML para SMDL é feita pelo aUML toolkit (PORRES, 2002), que cria o código SMDL equivalente do modelo a partir de informações extraídas do arquivo XMI gerado pelo editor UML. No passo seguinte, o código SMDL é convertido para uma máquina de estados finita. Ambas têm semânticas similares, o que permite uma conversão de forma praticamente direta. Após, este autômato passa por um processo de otimização, onde seus estados triviais são removidos e, posteriormente, políticas de escalonamento são aplicadas. O resultado deste processo é uma máquina de estados finita bem otimizada que representa o modelo original. Cada ramo desta FSM (*Finite State Machine*) é traduzido em processos que implementam máquinas de estado. Diferentes ramos significam processos em paralelo. Cada diagrama de estados UML é equipado com uma fila do tipo FIFO (*First In First Out*) para representar mecanismos de comunicação. Contudo, o presente trabalho ainda não cobre esse tipo de situação, pois a comunicação entre máquinas de estado dentro de diagramas de colaboração, ainda não é bem definido no UML padrão. Apesar dos passos serem exemplificados graficamente, este trabalho carece de maiores detalhes em relação à modelagem dos sistemas e ao nível de código VHDL gerado.

Uma metodologia bem prática para o mapeamento de UML para VHDL é apresentada em (RIEDER *et al.*, 2006). Este trabalho explica de forma simples os conceitos que norteiam um mapeamento direto entre estas linguagens. Primeiramente, o sistema é totalmente modelado na linguagem UML e dividido manualmente em componentes de hardware e software. Os autores afirmam que essa divisão manual serve para dar maior flexibilidade ao projetista na escolha dos componentes. Não há maiores detalhes sobre essa distinção dos componentes em relação à sua funcionalidade. Após, o modelo é exportado em um arquivo XMI que é passado por um *parser*, já com as partes de hardware e software separadas. Este *parser* é uma ferramenta que proporciona a automatização da geração de código. Ele contém regras de mapeamento direto entre as duas linguagens. Gera código na linguagem C/C++ para a parte do sistema relativa ao software e código VHDL para parte de hardware. O trabalho não detalha a parte relativa ao software, pois é considerada bem experimentada na área acadêmica possuindo inclusive ferramentas comerciais para gerenciamento deste

processo. A parte de hardware é detalhada de forma prática. Começando por restrições de modelagem, que delimitam a utilização de apenas alguns diagramas e elementos da UML, por serem considerados suficientes à modelagem estrutural e dinâmica de sistemas simples. Regras devem ser seguidas durante a modelagem em relação à construção dos diagramas. Essas regras garantem a construção de um modelo não ambíguo e limitado às estruturas que são reconhecidas pelo gerador de código. O diagrama de objetos é utilizado para modelar a estrutura física do sistema, com classes e suas comunicações. A partir dele são geradas entidades e componentes VHDL representando respectivamente, classes e instâncias de classes do UML. Portas de *reset* e *clock*, representativas de estruturas de hardware, são adicionadas à estrutura das entidades do código VHDL. Além disso, neste momento também é mapeada a comunicação entre as classes através de portas adicionadas ao componente que instancia as classes como itens físicos do hardware. Diagramas de estado são utilizados para modelar o comportamento de cada componente do sistema. São traduzidos para estruturas do tipo *case* em VHDL, onde cada item do *case* representa um estado do diagrama de estados. Transições podem ocorrer a cada pulso de *clock* e só dependem do que foi gerado como próximo estado. Eventos e a troca de mensagens são feitos através de bordas de subida ou descida nas portas que simbolizam relações e comunicação entre objetos. Ainda não há suporte para estruturas mais complexas da orientação a objetos, como por exemplo, polimorfismo, herança, entre outras, que poderiam ser modeladas em UML. Através dos exemplos disponíveis junto ao texto é possível perceber que existem trechos de código que não são gerados. Por exemplo, as portas de comunicação, são inseridas na declaração dos componentes e não na declaração da entidade. Contudo, este trabalho é muito bom e tem diversas semelhanças com a proposta dessa dissertação, como por exemplo, os processos que transformam UML em código, a utilização de regras de mapeamento, a imposição de restrições à modelagem e as estruturas mapeadas entre as linguagens.

Outro trabalho oriundo do projeto ModEasy é apresentado em (AKEHURST *et al.*, 2007). Trata de uma discussão e avaliação do uso de técnicas de MDD (*Model-Driven Development*) como meio de compilar diagramas de estado UML em código VHDL sintetizável. Aborda uma nova metodologia para o processo de geração de hardware que acontece no ModEasy, utilizando transformações entre metamodelos e suas instâncias (modelo) para chegar ao código. No primeiro passo desta metodologia, o modelo comportamental de alguma parte do sistema, desenvolvido a partir de diagramas de estado, é transformado em um metamodelo de máquinas de estado. Na etapa seguinte, aquele metamodelo de máquinas de estado é transformado em um metamodelo VHDL. Os conceitos do UML são transformados em conceitos do VHDL com a ajuda de técnicas e ferramentas de suporte que utilizam regras de transformação baseadas no padrão OMG`s *Queries Views and Transformations document* (QVT) (OMG QVT, 2010). Por fim, o metamodelo VHDL passa por uma transformação modelo-texto que o converte em código, de forma automatizada, a partir da linguagem de *templates* OCL (OCL *based templates language*) (OMG OCL, 2010). A linguagem OCL navega pelas estruturas do metamodelo obtendo elementos com os quais irá preencher suas *templates*, já bem escritas em VHDL, para gerar o código esperado. Este trabalho foca principalmente no aspecto da aplicação desta nova técnica, fazendo uma avaliação dos resultados obtidos a partir dos estudos de caso apresentados. Contudo, nenhum outro ponto referente à geração de código para VHDL é abordado, o que deixa a desejar.

(WOOD *et al.*, 2008) apresenta outro trabalho derivado do projeto ModEasy. Parece ser uma continuação daquele desenvolvido por (AKEHURST *et al.*, 2007), mas muito melhor elaborado e trazendo maior detalhamento sobre o fluxo de projeto, possibilidades de modelagem, ferramentas e técnicas de transformação de modelos. Apesar de ser praticamente igual àquele de Akehurst, neste trabalho as técnicas de MDD são melhores definidas e o processo de compilação trata os metamodelos de transformação por PIM e PSM. Assim, os diagramas de estado são convertidos em um modelo independente de plataforma (*Platform Independent Model* - PIM), que é baseado no metamodelo de máquinas de estado UML. Depois, este PIM é submetido a um framework chamado *Simple Transformation* (SiTra), que realiza sua transformação para um modelo VHDL do tipo PSM (*Platform Specific Model*), originado do metamodelo VHDL desenvolvido pelos autores para este trabalho. O *framework* SiTra é responsável por automatizar o processo de transformação entre modelos, aplicando regras de mapeamento previamente estipuladas na linguagem OCL, que percorrem o modelo PIM em busca de elementos para completar suas *templates* de código VHDL, para então, utilizá-las para construir os arquivos VHDL correspondentes ao sistema. O trabalho traz também, descrições de técnicas para o projeto de FSM em VHDL, transformações informais que norteiam o mapeamento de conceitos entre as linguagens, explicação detalhada da implementação das regras que realizam o mapeamento formal entre o PIM e o PSM, e também, um detalhamento de todo o fluxo de projeto com a utilização da ferramenta ModEasy. Um estudo de caso bem completo também é apresentado, juntamente com uma conclusão que tem o intuito de divulgar (e orientar) novas tendências à geração automática de código VHDL a partir de especificações UML. Por fim, mesmo possuindo restrições à modelagem do comportamento do sistema, sem tratar da análise de requisitos e sem artifícios para validação do código gerado, este trabalho obteve excelentes resultados em relação à qualidade e à quantidade de código VHDL gerado.

Um trabalho totalmente baseado em MDE é apresentado em (LE BEUX *et al.*, 2008). Ele faz parte do projeto ModEasy[6] que pesquisa o desenvolvimento de técnicas e ferramentas de software para auxiliar no desenvolvimento de sistemas (embarcados) eletrônicos baseados em microprocessadores, usando avançados sistemas para desenvolvimento e verificação. Tem como objetivo principal desenvolver ferramentas que automaticamente gerem código VHDL a partir de modelos UML com altos níveis de abstração. O UML foi escolhido como linguagem de especificação por ser altamente modular e expansível, permitindo especificações de comportamento em modelagem de alto nível. Usando MDE e técnicas de compilação para transformação entre modelos, são aplicadas sucessivas transformações de um modelo UML até a geração do código VHDL. O fluxo de projeto apresentado é suportado por ferramentas que permitem especificar sistemas utilizando ferramentas UML, reusar componentes funcionais e físicos e, assegurar um refinamento entre níveis de abstração através de regras de transformação, que por sua vez também asseguram a abertura deste modelo a outras ferramentas através da utilização do *profile* padrão do UML para modelagem e análise de sistemas tempo-real e embarcados (do inglês, *Modeling and Analysis of Real-Time and Embedded Systems* - MARTE). Seu fluxo parte de uma transformação do modelo UML para um modelo de aplicação, que contém simplificações do UML e maiores detalhes da aplicação. Seguindo, este modelo de aplicação é então transformado em um modelo em nível de registradores e portas lógicas (RTL). Este modelo representa o

[6] ModEasy Project. www.lifl.fr/modeasy/

hardware a ser gerado, sendo independente de qualquer linguagem de descrição de hardware. Na sequência, este modelo RTL sofre uma transformação textual que o traduz para a linguagem de descrição de hardware alvo, que neste caso é o VHDL. A definição de transformação de modelos é bastante clara neste trabalho. No entanto, alguns procedimentos e passos de modelagem e as ferramentas utilizadas não foram bem detalhados.

Mischkalla *et al.*(2010) apresenta um trabalho que contém uma metodologia completa para cobrir o *gap* entre modelagem baseada em UML e simulação baseada em SystemC. A modelagem do sistema é feita através da utilização de um conjunto de *profiles* específicos, que servem para estender a capacidade da ferramenta comercial ARTiSAN Studio, utilizada como suporte. Além do SysML que já faz parte desta ferramenta, foram adicionados outros três *profiles* que permitem a especificação de modelos direcionados as linguagens SystemC e C, e ainda, direcionados à síntese de hardware. Estes *profiles* dispõem de estereótipos que permitem decorar o modelo com marcações que auxiliam a ferramenta de geração de código, da própria ARTiSAN, a identificar quais itens do modelo são código referentes ao software, devendo ser gerados como linguagem C, e quais são hardware, devendo ser gerados em SystemC. A parte de software é completamente gerada neste passo. O hardware é especificado em SystemC, cuja especificação, contém também alguns macros pertencentes ao *Agility SC Compiler*. A geração de código é baseada em transformações *1-to-1*, definidas deste modo para que a captura do modelo fosse mais eficiente e mais clara evitando ambiguidades nas transformações entre especificações. Após a fase de modelagem e geração de código, ainda são necessários mais duas etapas. A primeira delas é síntese de hardware. O código SystemC gerado pela ARTiSAN é passado pelo compilador *Agility* que converte tal especificação para código VHDL, que é sintetizado pelo conjunto de ferramentas ISE/EDK *Design Suite* da Xilinx. Então, o software em linguagem C é executado pelo hardware sintetizado. Este trabalho tem particularidades bem interessantes, como por exemplo, seu método de simulação, que é capaz de simular o hardware e software em conjunto, e ainda, o modo como deve ser modelada a especificação do sistema e suas peculiaridades. Dois estudos de caso são apresentados para avaliação da eficiência e comprovação desta metodologia. Porém, o código gerado não foi exemplificado e nem foram dadas maiores informações sobre ele.

3.4 Trabalho de maior relevância

Para esta seção de trabalhos relacionados ao tema, o trabalho de (WEHRMEISTER *et al.*, 2008) com certeza é o mais importante. Nele foi desenvolvida a ferramenta e a metodologia que norteiam esta dissertação. A metodologia proposta é baseada em transformação de modelos (MDE) e utiliza conceitos de orientação a aspectos para tratar os requisitos não funcionais de um sistema, desde a fase inicial do projeto, até a geração do código fonte na linguagem Java. A análise que proporciona a identificação dos requisitos de um projeto é realizada na fase inicial da metodologia. Ponto onde os requisitos identificados são divididos entre funcionais e não funcionais conforme o impacto que causam sobre o sistema. Os requisitos funcionais são somente identificados durante a fase de modelagem para uso posterior. Já os requisitos não funcionais passam por um processo que serve para detectar quais aspectos[7] representam aquele único

[7] Aspectos são entidades que contém comportamentos que afetam múltiplas classes do sistema e foram encapsulados como módulos de código secundário, separados do restante da aplicação.

requisito. Esse desmembramento de um requisito não funcional em um ou mais aspectos é possível através da utilização de um framework de aspectos chamado em inglês de *Distributed Embedded Real-time Aspects Framework* (DERAF), que contém especificações de aspectos e de como eles podem afetar sistemas (FREITAS, *et al.*, 2007). Assim, um requisito é representado por aspectos que definem características que serão impostas ao código final do sistema para representar de fato aquele requisito não funcional. Passando à fase de modelagem, os requisitos funcionais são modelados junto aos diagramas principais, que especificam características estáticas e dinâmicas do sistema. Já os aspectos que representam os requisitos não funcionais do sistema são modelados a partir de um diagrama diferenciado chamado em inglês de *Aspects Crosscutting Overview Diagram* (ACOD), que deverá conter todo o conjunto de aspectos que afeta o sistema. Na sequência, o projetista deverá especificar quais elementos do modelo UML são afetados por aqueles aspectos. Para isso, são criados diagramas especiais que especificam uma forma de selecionar elementos do modelo, os chamados *Join Point Designation Diagrams* (JPDD). A relação ACOD x JPDDs identifica quais serão as estruturas que sofrerão modificações devido à atuação dos aspectos, e de que forma estas estruturas serão afetadas. Com o modelo pronto, começa o processo de transformação de modelos. Neste ponto da metodologia, ocorre a transformação daquele modelo inicial (PIM) num outro modelo intermediário, que se chama DERCS (*Distributed Embedded Real-Time Compact Specification*). Essa transformação intermediária cria um modelo compacto representativo de sistemas embarcados distribuídos que é totalmente livre de ambiguidades. Tal modelo servirá como entrada à ferramenta GenERTiCA, que automatiza o processo de geração de código, realizando outra transformação entre modelos que resulta num modelo de plataforma específica (PSM), que é convertido em código textual na linguagem Java.

Maiores explicações sobre a metodologia e o processo de geração de código serão vistos na próxima seção que trata do cerne deste trabalho de mestrado, apresentando em detalhes o porquê da escolha do trabalho de Wehrmeister como base para dissertação, seus benefícios, pontos a melhorar, e obviamente, a extensão desenvolvida para ser utilizada junto a GenERTiCA, que permite a geração de código VHDL.

3.5 Conclusões após análise do estado da arte

Especificar um sistema com alto nível abstração significa descrever seus componentes e comportamento de uma forma mais geral, sem dar maiores detalhes sobre a implementação. Linguagens de programação com mecanismos de abstração de dados e linguagens gráficas servem à especificação de sistemas em alto nível. Modelar a estrutura física e comportamental de um sistema a partir de uma linguagem de programação é bastante trabalhoso, exigindo amplo conhecimento sobre as estruturas fornecidas pela linguagem para utilizá-las de maneira adequada tornando a especificação otimizada, além de ser um método nada intuitivo. A utilização de linguagens gráficas (e.g. UML) torna o processo de especificação mais intuitivo, dando ao projetista uma visão geral do sistema e das relações que o compõem, permitindo assim, que a preocupação fique apenas com o aspecto funcional do sistema.

O resultante de um processo de geração de código automatizado pode ser descrições de hardware ou código fonte em alguma linguagem alvo (software). Código fonte representa a parte lógica de um sistema, descrevendo suas funcionalidades e serviços. Já uma descrição de hardware significa a representação dos seus componentes, suas

interações e comportamento de uma forma textual, como linguagem de programação com sintaxe e semântica bem definidas (e.g. VHDL, Verilog e SystemC). Contudo, diferenciam-se das demais linguagens pela possibilidade de serem diretamente simuladas e sintetizadas em FPGAs (i.e. transformadas em uma representação em nível de registradores e gravada num FPGA formando efetivamente aquele circuito que representava).

Assim, analisando a geração de código de um modo geral para VHDL, é interessante salientar que devido ao *gap* semântico existente entre uma especificação de hardware com alto nível abstração e sua descrição, ainda não existem trabalhos, nem ferramentas comerciais capazes de suportar a geração de código VHDL completa para qualquer tipo de sistema. Além disso, não existe um padrão em termos de metodologia a ser utilizada ou descrição de hardware gerada. Considerando conversões de UML para VHDL, ainda existem poucos trabalhos direcionados especificamente a este tema. Porém, alguns deles têm bons resultados, ainda que, limitados a subconjuntos da linguagem UML. Isso demonstra que essa área ainda necessita de esforços em pesquisa e que há espaço para boas propostas e ferramentas que venham a servir com exatidão a este propósito.

Quanto à geração de código a partir de UML, foram encontrados inúmeros trabalhos que utilizam diferentes metodologias e geram código para diferentes linguagens. A análise identificou diferenças entre os processos de engenharia utilizados e nenhuma padronização. Os trabalhos estudados variam bastante em relação à metodologia de projeto utilizada. Alguns deles utilizam como base para a etapa de modelagem perfis (i. e. *profiles*) disponíveis para UML 2.x na OMG enquanto outros utilizam perfis próprios, desenvolvidos especificamente para auxiliar na caracterização e captura de informações do modelo. Muitos ainda transformam primeiramente o modelo UML em um modelo próprio para depois realizar o mapeamento para linguagem alvo. Outros usam processos bem elaborados como técnicas de MDA. Contudo, nenhum destes trabalhos detalha a fase de especificação de requisitos ou trata a separação de interesses relacionada aos requisitos funcionais e não funcionais do projeto de um sistema. Apesar disso, a diversidade de trabalhos examinados nesta linguagem demonstra a alta escalabilidade do UML para especificação de sistemas e o quanto essa linguagem ainda pode ser explorada.

Focando nas transformações de UML para VHDL, os trabalhos analisados compartilham bons conceitos e deixam lacunas que podem ser exploradas. O trabalho de (McUMBER; CHENG, 1999) é uma fonte de teoria e conceitos sobre o mapeamento entre estas linguagens. (BJÖRKLUND; LILIUS, 2002) e (RIEDER *et al.*, 2006) apresentam processos mais elaborados, onde o modelo inicial é transformado em modelo intermediário antes de ser convertido em código VHDL. A proposta desta dissertação assemelha-se bastante ao trabalho de Rieder por utilizar regras de mapeamento para guiar a ferramenta de geração de código e por definir restrições à modelagem. Os trabalhos de (AKEHURST *et al.*, 2007), (WOOD *et al.*, 2008) e (LE BEUX *et al.*, 2008) abordam transformações entre modelos baseados em técnicas de MDD e MDE. Essa dissertação também está baseada em MDE e compartilha diversos conceitos com o trabalho de Wood, cuja metodologia trabalha com PIM, PSM e utiliza uma linguagem de templates (OCL) como ferramenta auxiliar na conversão modelo-texto que gera o código VHDL final. (VIDAL *et al.*, 2009) apresenta uma abordagem que divide o processo de modelagem em três fases, cuja característica de união dos modelos através do uso do diagrama de componentes do UML aparenta facilitar o desenvolvimento das estruturas de comunicação que devem ser geradas juntos aos

componentes de hardware. (MISCHKALLA *et al.*, 2010) utiliza-se de uma ferramenta comercial (ARTiSAN) para modelagem e como suporte à geração de código. Difere da metodologia desta dissertação pelo fato de utilizar profiles próprios para adequar os modelos à ferramenta de geração de código. Esta dissertação, além de utilizar a ferramenta Magic Draw como suporte, utiliza também o bem conhecido profile MARTE para caracterizar o modelo com marcações de sistemas de tempo-real. Adicionalmente, Mischkalla não baseia seu processo em transformação de modelos, porém, gera através de transformações diretas código em linguagem C para parte de software e System C para parte de hardware, cujo processo ainda permite outro mapeamento direto, que gera VHDL a partir de System C.

Concluindo, todos os trabalhos apresentados nesta seção foram utilizados como referência à proposta desta dissertação, tanto no embasamento teórico, quanto na percepção de melhorias que ainda podem ser desenvolvidas nos processos existentes. Desta forma, a proposta aqui apresentada defende a utilização de uma metodologia completa para o projeto de sistemas tempo-real embarcados (STRE), AMoDE-RT, desenvolvida por (WEHRMEISTER, 2009), e apresenta uma extensão à ferramenta suporte desta metodologia, GenERTiCA, que permite a geração de código VHDL a partir de modelos UML (MOREIRA *et. al.*, 2010a, 2010b). Este conjunto difere dos demais, pelo fato de utilizar um ciclo completo de projeto, provendo técnicas e meios que auxiliam o projetista desde a fase de análise de requisitos até a geração do código. Adicionalmente, fornece uma metodologia para o tratamento dos requisitos funcionais e não funcionais, que quando misturados nas fases iniciais do projeto, principalmente na fase de modelagem, tornam a implementação do sistema mais complexa. E ainda, utiliza uma proposta genérica para geração de código, dependendo somente do conjunto de regras de mapeamento utilizado sobre a ferramenta motor de transformação, para gerar código para software, nas linguagens C++ e Java, e para hardware, na linguagem VHDL, a partir do mesmo modelo.

4 CONCEITOS E REGRAS DE MAPEAMENTO ENTRE AS LINGUAGENS

Este trabalho propõe a utilização de uma metodologia completa para o projeto de sistemas tempo-real embarcados (STRE) e apresenta uma extensão, na forma de um conjunto de regras de mapeamento, que permite a ferramenta motor de transformação entre as linguagens gerar código VHDL a partir de modelos UML. A metodologia utilizada propõe um processo que guia o projetista deste a fase de análise de requisitos até a geração de código, fornecendo técnicas e meios que facilitam e tornam o ciclo de desenvolvimento mais ágil. Desta forma, esta seção apresenta a metodologia *Aspect-oriented Model-Driven Engineering for Real-Time systems* (AMoDE-RT), detalhando suas etapas. Além disso, discute o mapeamento de conceitos entre as linguagens UML e VHDL, salientando suas diferenças e apresentando esforços e dificuldades necessários à transformação de uma linguagem na outra. Com foco principal, também é apresentado nesta seção, o conjunto de regras de mapeamento, que estende a funcionalidade da ferramenta GenERTiCA (*Generation of Embedded Real-Time Code based on Aspects*), permitindo o mapeamento de modelos UML em código fonte da linguagem de descrição de hardware VHDL. A figura 4.1 apresenta uma visão geral do fluxo da metodologia AMoDE-RT, que se caracteriza por um conjunto de etapas sequenciais na forma de um processo.

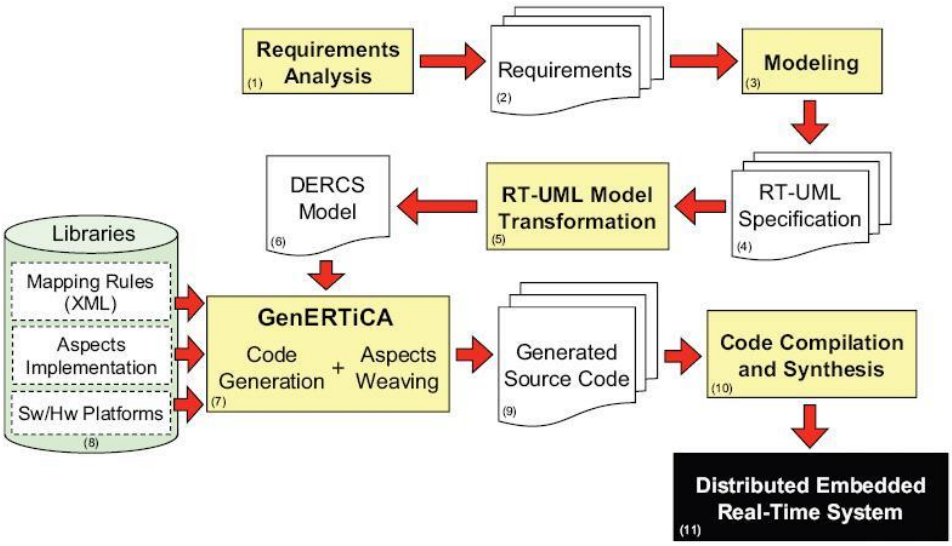

Figura 4.1: Visão geral do fluxo realizado pela metodologia AMoDE-RT (WEHRMEISTER, 2009).

4.1 Metodologia Utilizada – AMoDE-RT

A figura 4.1 apresenta o fluxo de projeto proposto pela metodologia AMoDE-RT (WEHRMEISTER et al., 2008, WEHRMEISTER, 2009), que utiliza técnicas de MDE combinadas a conceitos de orientação a aspectos para o projeto de STRE. AMoDE-RT é suportada pela ferramenta de geração de código GenERTiCA, também desenvolvida por Wehrmeister. Essa ferramenta utiliza scripts contendo regras de mapeamento para produzir arquivos de código fonte em uma plataforma alvo a partir de modelos UML decorados com especificações do *profile* MARTE. Adicionalmente, GenERTiCA é capaz de gerar código para distintas linguagens (Java, C++, etc.), desde que exista um conjunto de regras de mapeamento que guie seu processo de geração de código àquela plataforma. O processo para gerar código é considerado genérico, pois gera código às diferentes plataformas a partir do mesmo modelo UML. A geração de código para uma plataforma alvo depende apenas do conjunto de regras de mapeamento utilizado (i.e. se utilizado o conjunto de regras RA, o código gerado será A, se RB, o código gerado é B, ambos a partir do mesmo processo). Os *scripts* que realizam o mapeamento ficam armazenados sobre um documento que possui uma estrutura padrão (*template*), sendo exatamente o mesmo independente da linguagem para qual foi concebido. O que muda são os scripts. Por isso, o resultado da geração de código depende somente do arquivo de regras utilizado pela ferramenta que automatiza o processo. Esta dissertação propõe uma extensão à ferramenta GenERTiCA, em termos de um novo conjunto de regras de mapeamento, para mapear elementos do modelo UML em construções da linguagem VHDL.

Na sequência, são detalhadas as etapas do fluxo de projeto que define a metodologia AMoDE-RT. Essas etapas devem ser executadas de forma sequencial, da maneira como estão sendo aqui apresentadas. Cada uma delas contém particularidades que devem ser seguidas pelo projetista a fim de permitir uma transição suave para etapa posterior. Dentro destas particularidades existem regras que definem informações específicas que devem ser inseridas no modelo para representar cada situação, tornando-o desta maneira apto e com dados suficientes para correta passagem à próxima etapa.

4.1.1 Análise e identificação de requisitos

A primeira etapa da metodologia AMoDE-RT é a identificação e especificação dos requisitos do sistema que será modelado. Essa tarefa é realizada através da ferramenta para análise de requisitos RT-FRIDA (*Real-Time* FRIDA) (FREITAS, 2007), que é uma extensão da metodologia para análise de requisitos FRIDA (*From RequIrements to Design using Aspects*) (BERTAGNOLLI, 2004). A extensão *Real-Time* desta metodologia facilita o processo de análise e identificação de requisitos no domínio dos SETRD. Seguindo esta metodologia o projetista identifica mais facilmente, através de uma análise dos elementos externos que interagem com o sistema modelado, quais os requisitos que o projeto deve conter e quais as restrições devem ser obedecidas. Além de ser uma ferramenta chave na etapa de análise de requisitos, a ferramenta RT-FRIDA também compartilha o passo de modelagem da metodologia AMoDE-RT, auxiliando no enriquecimento do modelo com alguns detalhes importantes ao processo. São apresentados na figura 4.2 os passos realizados pela ferramenta RT-FRIDA para analisar e identificar requisitos funcionais e não funcionais. Porém, somente uma visão superficial do processo será exposta aqui, para maiores informações a respeito, a referência (FREITAS, 2007) está disponível.

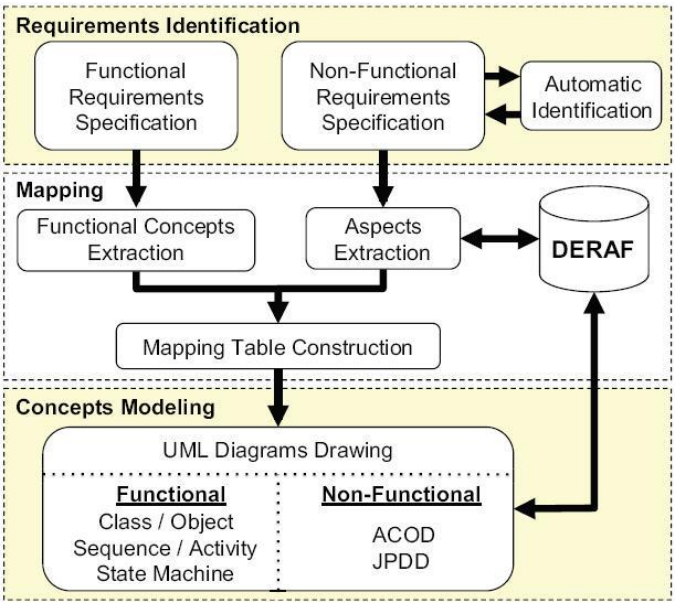

Figura 4.2: Processo de análise de requisitos da ferramenta RT-FRIDA (WEHRMEISTER, 2009).

A identificação de requisitos é o primeiro passo da metodologia, sendo dividido em duas atividades: especificação dos *requisitos funcionais* e *requisitos não funcionais*. Primeiramente, é desenvolvido um diagrama de caso de uso para o sistema. Este diagrama apresenta todas as funcionalidades esperadas de um STRE, e também, todos os elementos externos que interagem com o sistema através daquelas funcionalidades. Na sequência, para cada caso de uso específico identificado no diagrama, é preenchida uma planilha que contém detalhes do requisito, como por exemplo, nome, identificador, objetivo, condições, prioridade, entre outros campos. A metodologia utiliza esse esquema de planilhas para analisar e identificar os requisitos, contando ainda com planilhas de resolução de conflitos e com planilhas específicas para requisitos não funcionais. Ao final deste primeiro passo, temos uma noção clara dos requisitos existentes e onde eles afetam o sistema.

O segundo passo do processo RT-FRIDA é o mapeamento de requisitos para elementos de projeto (elementos candidatos, não definitivos). Isso é feito também através de planilhas e resulta em uma tabela de mapeamento que faz link entre requisitos e elementos de projeto, permitindo rastrear requisitos desde a análise até o projeto do sistema. Essa tabela de mapeamento ainda indica quais aspectos são usados para lidar com elementos não funcionais. Os aspectos são fornecidos por um *framework* de aspectos predefinidos, chamado *Distributed Embedded Real-Time Aspects Framework* (DERAF). Este framework contém um conjunto de predefinições que determina como estes aspectos afetam o modelo e o código gerado.

A ferramenta RT-FRIDA tem como terceiro passo, o compartilhamento da etapa de modelagem da metodologia AMoDE-RT, auxiliando no enriquecimento do modelo com alguns detalhes e essências importantes ao processo. O projetista que está implementando um sistema complexo, e optou por executar o processo de análise de requisitos baseado RT-FRIDA (essa etapa é importante, mas não mandatória para o projeto), tem como resultado uma tabela de mapeamento, que facilita bastante a fase de modelagem, pois indica de forma direta itens que devem ser modelados em UML para representar os requisitos funcionais e não funcionais existentes.

4.1.2 Modelagem

Na fase de modelagem, diagramas UML decorados com estereótipos do *profile* MARTE são usados para especificar/modelar a estrutura e o comportamento de sistemas tempo-real embarcados (STRE). Nesta fase, modelos UML são criados e sucessivamente refinados até atingirem um nível de detalhamento desejado, com informações suficientes, que permitam a realização do sistema. No modelo UML inicial, elementos descrevem os conceitos mais próximos ao domínio de aplicação alvo, e.g. sensores, dispositivos de direção, turbinas, informações de velocidade e trajetória, braços robóticos, etc. Estes elementos representam conceitos de domínio problema, escondendo detalhes sobre suas implementações. Níveis mais altos de abstração favorecem o entendimento e permitem ao projetista focar na base das aplicações ao invés de preocupar-se com questões de implementação. Portanto, eles representam o gerenciamento de requisitos funcionais. Elementos de aplicação podem ser reusados de projetos anteriores. Isso permite que sejam criados repositórios de elementos de domínio de aplicação específicos. Tais elementos podem ser compostos de muitos elementos UML ou diagramas diferentes. Por exemplo, um braço robô pode ser composto por três juntas e uma garra. Para reusar estes elementos de domínio, ao menos cinco classes seriam reusadas (três para juntas, uma para garra e a classe de composição que representaria o braço do robô). Adicionalmente, diagramas comportamentais descrevendo o funcionamento do braço robótico também poderiam ser reusados.

A especificação de gerenciamento e manipulação de elementos não funcionais é feita com o auxílio de aspectos providos pelo framework DERAF. Estes aspectos são utilizados em dois momentos: *(i)* na fase de modelagem, e *(ii)* na fase de implementação, mais especificamente, durante a geração de código na etapa de *aspects weaving*. Durante a fase de modelagem, os aspectos são escolhidos com base em sua semântica de alto nível para tratar requisitos não funcionais que afetam mais de um interesse ao mesmo tempo, atingindo de forma transversal elementos do projeto (i.e. *crosscutting* de requisitos não funcionais). Por exemplo, existe um aspecto chamado *ConcurrentAccessControl*, que trata de questões relacionadas ao controle de acesso concorrente a recursos compartilhados. Assim, se o sistema sendo projetado possui esse requisito não funcional, o aspecto *ConcurrentAccessControl* deve ser selecionado e especificado no diagrama ACOD (*Aspects Crosscutting Overview Diagram*), para tratar disso. Além disso, baseado em informações da tabela de mapeamento criada anteriormente, na etapa de análise de requisitos, os projetistas devem especificar quais elementos do modelo UML são afetados por tal aspecto. Para isso, os projetistas criam *Join Point Designation Diagrams* (JPDD), que representam uma classe especial de diagramas que servem para especificar a seleção de elementos do modelo (i.e. selecionam elementos do modelo). JPDDs podem ser armazenados em repositório para uso posterior e, além disso, podem ser construídos a partir de ferramentas de modelagem UML comuns com suporte para *profiles*. Maiores detalhes sobre modelagem de requisitos funcionais e requisitos não funcionais são encontrados em (WEHRMEISTER, 2009).

4.1.3 Transformação UML-to-DERCS

Embora aumentar o nível de abstração do modelo seja bom para o gerenciamento do projeto de sistemas complexos, quanto mais alto o nível de abstração de dados do modelo, maiores são as chances de ambiguidades ou interpretações erradas sobre a mesma especificação. Usualmente, especificações de alto nível não podem ser

executadas por dispositivos computacionais (e.g microprocessadores, circuitos integrados ou controladores de lógica programável - PLC), devido à sua semântica incompleta e/ou a falta de detalhes. Para superar tais questões, quaisquer ambiguidades existentes devem ser removidas das especificações, assim como, elementos computacionais (e.g. filas tipo FIFO, *schedulers*, mecanismos de sincronização, e outros) devem ser incluídos dentro destas especificações de alto-nível. Em outras palavras, deve ocorrer uma transformação daquele modelo inicial em outro mais conciso.

Esta terceira etapa da metodologia AMoDE-RT executa a transformação do modelo UML decorado com estereótipos do *profile* MARTE, um PIM (*Platform-Independent Model*), em uma instância de um modelo intermediário livre de ambiguidades chamado *Distributed Embedded Real-time Compact Specification* (DERCS), que é um PSM (*Platform Specific Model*) configurável para propósitos de geração de código e execução de modelos. A transformação de UML para DERCS sintetiza em poucos e concisos elementos, todas as informações sobre a estrutura, o comportamento e a manipulação de requisitos não funcionais de um sistema, que poderiam estar espalhadas ao longo de diferentes diagramas UML. Assim, um modelo DERCS, intermediário, de plataforma específica e livre de ambiguidades, é obtido para ser utilizado em transformações modelo-texto, cuja etapa remete a efetiva geração de código a diferentes plataformas-alvo.

4.1.4 Geração de código

Após sucessivos passos de refinamento, o modelo inicial é transformado em modelo DERCS, que é base para geração de código fonte para alguma plataforma-alvo. Como foi mencionado anteriormente, um dos objetivos deste trabalho é proporcionar uma transição "suave" de modelos em alto-nível de abstração para descrições de hardware de um sistema tempo-real embarcado. O objetivo de proporcionar uma transição suave é sinônimo de fornecer ao projetista e/ou usuário, meios para ir do alto para o baixo nível sem esforço, preferencialmente de forma automática. Assim, a ferramenta GenERTiCA, desenvolvida por (WEHRMEISTER *et al.*, 2008), é utilizada para automatizar este processo. Esta ferramenta atua como mecanismo motor das transformações entre modelos e geração de código, tornando o processo e todos os passos que levam do modelo UML até o código fonte, totalmente transparentes ao projetista/usuário. Adicionalmente, quando o sistema modelado contém requisitos não funcionais, GenERTiCA realiza também o entrelaçamento dos aspectos que foram gerados para aqueles requisitos (do inglês, *aspects weaving*), que significa realizar a junção dos códigos daqueles requisitos funcionais e não funcionais gerados separadamente pelo processo, formando um sistema.

Para gerar o código fonte na linguagem requerida, GenERTiCA processa um conjunto de *scripts* que formam regras de mapeamento e servem para guiar o processo de transformação modelo-texto. O processo itera sobre os elementos do modelo DERCS, tomando um a um e percorrendo as regras de mapeamento, procurando pelo *script* que melhor representa aquele elemento do modelo. Essa rotina é executada de forma automática até mapear todos os elementos do modelo DERCS. É importante salientar que o modelo DERCS é livre de ambiguidades e que existe uma regra de mapeamento para cada elemento do modelo. Assim, nenhum elemento do modelo DERCS encontrará mais de uma regra de mapeamento, e nem deve existir elementos do modelo sem regras.

Regras de mapeamento são especificadas como pequenos *scripts* que criam fragmentos de código (representando construções da linguagem alvo) para elementos do modelo DERCS. Arquivos de código fonte são construídos a partir de combinações destes fragmentos de código. *Scripts* são armazenados e organizados em arquivos de regras de mapeamento, que por sua vez são especificados e estruturados usando o formato *eXtensible Markup Language* (XML) (W3C, 2006). Assim, é possível criar um repositório para reutilização de *scripts* e regras de mapeamento, desenvolvidos em projetos anteriores. Para o elemento DERCS sendo avaliado, o processo de geração de código itera sobre as regras de mapeamento, procurando pelo *script* que define o mapeamento que o transformará em uma construção configurável na plataforma alvo.

Adicionalmente, se o elemento em avaliação estiver afetado por um aspecto do framework DERAF, o processo de entrelaçamento de aspectos (*aspect weaving*) é executado logo após o fragmento de código ser gerado. A ferramenta GenERTiCA usa implementações de aspectos para modificar os fragmentos de código. Em outras palavras, o fragmento de código gerado sofre modificações que refletem as adaptações impostas por aquela característica do aspecto que lhe afeta. Também existe a possibilidade de serem realizadas adaptações diretamente nos elementos do modelo DERCS antes da geração de código. Deste modo, GenERTiCA dá suporte ao entrelaçamento de aspectos no código e no modelo. É importante salientar que a implementação das modificações/adaptações promovidas por aspectos, também é feita por *scripts*, igualmente como as regras de mapeamento. Consequentemente, também é possível criar repositórios de diferentes implementações para a mesma adaptação de um aspecto, dependendo somente da plataforma alvo. Além de que, aspectos DERAF são utilizados também para adequar plataformas, de modo a configurar a plataforma alvo selecionada pela adição de serviços requisitados pela aplicação.

4.1.5 Compilação do código e síntese

O último processo da metodologia AMoDE-RT utiliza ferramentas de terceiros para compilar e sintetizar o código gerado para aplicação. Adicionalmente, os arquivos de configuração gerados para plataforma são usados para configurar a plataforma final que será implantada. Depois disso, a realização do sistema tempo-real embarcado que está sendo projetado, está pronta para ser executada e testada.

4.2 Mapeamento dos Conceitos entre as linguagens

A linguagem UML é bastante abstrata e foi especificada para o projeto de software. Além disso, por ser orientada a objetos contém um grande número estruturas, como por exemplo, vários tipos de dados, opções de visibilidade e tipo de parâmetros. Já o VHDL é uma linguagem estruturada, específica para descrição de hardware, possuindo uma tipagem de dados específica ao domínio de aplicação dos circuitos digitais. Essa brusca diferença entre os domínios de aplicação e o enorme *gap* existente entre o conceito das duas linguagens, torna o mapeamento entre elas uma tarefa nada trivial.

Para tornar o mapeamento entre essas linguagens possível, primeiramente, seus conceitos devem ser mapeados para criar uma relação entre as linguagens. Mapear conceitos significa criar convenções que formalizem transformações entre estruturas, definindo qual estrutura da linguagem A deve ser mapeada numa estrutura da linguagem B. Porém, para transformar uma estrutura em outra, é necessário que elas

tenham alguma similaridade, mesmo que em domínios de aplicação diferentes. Assim, as regras de mapeamento deste trabalho foram baseadas em transformações que levam em consideração o conceito que cada estrutura representa na sua linguagem, mapeando apenas estruturas que representam conceitos semelhantes entre as linguagens.

A Tabela 4.1 apresenta os conceitos que foram desenvolvidos para representar o mapeamento entre as linguagens. A primeira coluna exibe os elementos da linguagem UML que podem ser mapeados pelas regras desenvolvidas neste trabalho. A segunda coluna descreve as estruturas da linguagem VHDL, em quais os elementos da primeira coluna são convertidos.

Tabela 4.1: Mapeamento de conceitos entre UML e VHDL.

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
Herança	Palavras chave “new” e “tagged” do VHDL
Polimorfismo Estático	Estrutura de Configuração
Instanciação de Objetos	Estrutura de Componentes

Pelo fato de uma especificação UML conter inúmeras estruturas que representam o mesmo elemento no momento da geração do código, e em função da grande diferença estrutural entre as linguagens, apenas um subconjunto de elementos da linguagem UML foi mapeado para elementos da linguagem VHDL. Consequentemente, para evitar que o modelo seja construído com estruturas ambíguas e forçar para que sejam utilizadas somente estruturas da linguagem UML que estão mapeadas para a linguagem VHDL, é necessário orientar o ato de especificação/modelagem. Ambiguidades e a utilização de estruturas para as quais não existem regras de mapeamento resultariam, respectivamente, em erro no processo de geração de código através da ferramenta GenERTiCA e na geração de códigos incompletos.

Para orientar o projetista na correta construção de um modelo que possa ser totalmente transformado em código através da ferramenta GenERTiCA, existem regras de desenvolvimento (e.g. não utilizar classes abstratas, entre outras) que conduzem o projetista a utilizar somente construções da linguagem UML adequadas à transformação entre modelos e a geração de um código fonte completo pela ferramenta. Essa metodologia de desenvolvimento de modelos é detalhada em (WEHRMEISTER, 2009). Contudo, além da necessidade de desenvolver o modelo sobre a regulamentação de determinadas regras, também é mandatório utilizar no projeto somente aquelas estruturas para as quais existe o mapeamento entre as linguagens. Caso contrário, a geração de código será incompleta. No caso deste trabalho, o projetista poderia utilizar somente o subconjunto de estruturas da Tabela 4.1.

Alguns autores consideram que para modelagem e projeto de sistemas embarcados em alto-nível é suficiente apenas um subconjunto da linguagem UML. Por exemplo, (McUMBER; CHENG, 1999), autor de bem sucedidas pesquisas na área, utiliza somente diagramas de classes e diagramas de estado como subconjunto das notações

UML a serem mapeadas para VHDL. A metodologia AMoDE-RT compartilha da mesma lógica, porém trabalha com outros diagramas: Diagrama de Classes e Diagrama de Sequência. Considera-se que estes diagramas são suficientes para uma boa modelagem e para manter compatibilidade entre as estruturas mapeadas. Pois, o diagrama de classes possibilita especificar a estrutura do sistema, evidenciando os objetos que o compõe e sua rede de comunicação com os demais. Enquanto, o diagrama de sequência permite a descrição da dinâmica do sistema como um todo, através do detalhamento do conjunto de ações realizado por cada um de seus objetos.

Os conceitos que norteiam as transformações de UML para VHDL deste trabalho foram desenvolvidos a partir de pesquisas sobre o estado da arte da geração de código de UML para VHDL. Sobretudo, os conceitos propostos nessa dissertação foram baseados nos trabalhos de (McUMBER; CHENG, 1999) e (ECKER, 1996). De McUmber foram aproveitados os conceitos para a transformação de estruturas mais básicas, onde o mapeamento pode ser direto para certos elementos. Citando como exemplo, os conceitos elementares desenvolvidos para o mapeamento de estruturas como classes, métodos e atributos, cujas transformações são de forma direta para estruturas do VHDL. A partir do trabalho de Ecker foram visados conceitos para o mapeamento de estruturas mais complexas, como por exemplo, as estruturas de polimorfismo e herança, que são muito utilizadas na especificação de modelos UML. Entretanto, o mapeamento destas estruturas não foi desenvolvido. Pois, sua implementação exigiria o desenvolvimento de regras de mapeamento bastante extensas e complexas, para construir um código VHDL genérico o suficiente à representação de tais estruturas sem afetar o resto do código. Assim, optou-se apenas pelo desenvolvimento de estruturas mais simples que pudessem validar o trabalho.

4.2.1 Diagrama de classes

Representando o principal diagrama estrutural de um modelo UML, o diagrama de classes é uma representação da estrutura e das relações entre as classes que servem de modelo para objetos que compõem o sistema. Utilizar este diagrama é bastante importante para representação de um sistema, pois define o conjunto de estruturas e relações que o sistema possui, servindo como base para a construção dos diagramas de interação e comportamento, mais especificamente o diagrama de sequência que é considerado neste trabalho.

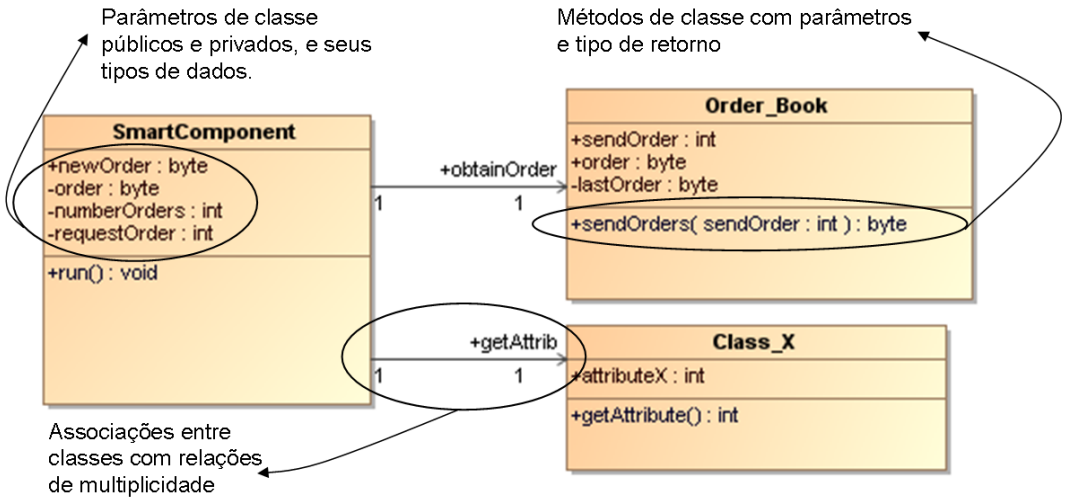

Figura 4.3: Detalhamento das estruturas UML que podem ser mapeadas para VHDL.

A figura 4.3 exemplifica as estruturas dos diagramas de classes que podem ser mapeadas para VHDL. Na sequência, é apresentado um detalhamento sobre o mapeamento de cada uma dessas estruturas.

A estrutura **classe** configura no UML um elemento abstrato que representa um conjunto de objetos. A classe contém a especificação do objeto, com suas características: atributos e métodos (ações / comportamentos). É mapeada para o par entidade-arquitetura na linguagem VHDL. A *entidade* representa o encapsulamento daquele elemento classe, com portas que representam seus atributos, e a *arquitetura*, contém as ações e comportamentos que essa classe pode realizar, representado pelos seus métodos.

Dentre as características possíveis para classes, foram desenvolvidas regras de mapeamento para atributos, públicos e privados, e para métodos. Os **atributos públicos** são mapeados para portas de entidade e ganham duplo sentido de direção (i.e. são ao mesmo tempo portas de entrada e saída - bidirecionais). Isso ocorre devido à impossibilidade de identificar a direção do atributo a partir do diagrama de classes. Imagina-se ser possível a identificação dessa característica de direção do atributo, se o mesmo estivesse modelado como uma porta no diagrama de composição, ou ainda, com o auxílio de estereótipos (VIDAL *et al.*, 2009). Porém, para este trabalho, a melhor solução encontrada foi mapear como uma porta bidirecional do VHDL (i.e. *inout*).

Atributos privados são mapeados como sinais internos (*signal*), definidos dentro da arquitetura do módulo. O mapeamento destes atributos para o VHDL é facilitado pelo fato destes sinais internos não serem exteriorizados, desta forma, não necessitando de direção. Assim, seu mapeamento só necessita levar em consideração o tipo de dado.

Os **métodos** são convertidos para processos do VHDL. Seus argumentos são representados como portas de entidade para permitir comunicação com o mundo exterior e desta forma permitir a sensibilização da função/processo, através de eventos externos (mudanças nos seus argumentos). Existe a necessidade de sua visibilidade ser sempre pública. Além disso, seu tipo de retorno de dados torna-se meramente ilustrativo no diagrama UML, pois este retorno necessita ser definido pela lógica de comportamento do método. Isto é, para ser mapeado corretamente para VHDL o comportamento/ação do método deve retornar o resultado em um dos atributos definidos na classe. Não foi possível mapear nessa versão de regras de mapeamento a diretiva *return*. Assim, a ação do método deve envolver variáveis da própria classe, e seu tipo de dado de retorno pode até ser diferente de *void*, mas dessa forma sua lógica deve garantir que a variável que será afetada ou receberá o resultado, será do mesmo tipo de dado definido para o método. O comportamento do método é desenvolvido na forma de uma sequência de ações, que será detalhada no item “Diagrama de Sequência”.

A estrutura de **associação entre classes** configura os relacionamentos que aquela classe pode ter com outras classes do sistema, tendo como características o nome daquele relacionamento, sua multiplicidade e o sentido de navegação (i.e. de onde vêm as informações da classe e para onde vai). No UML uma relação de associação pode ser de quatro tipos: simples, ou também conhecida como “*uses*”, agregação, composição e herança. Neste trabalho apenas a estrutura de associação simples foi desenvolvida. Pois, excetuando a relação de herança, a diferença entre essas estruturas dá-se apenas na abstração da orientação a objetos. Conceitualmente, no VHDL, elas representam a comunicação entre duas entidades. Motivo pelo qual apenas um mapeamento foi

desenvolvido. Como características, essa associação tem multiplicidade limitada em um para um (1-1) e sentido de navegação único, definindo uma relação bilateral onde a transferência de informações é realizada somente em um sentido, apenas de uma das classes para outra.

O desenvolvimento de regras de mapeamento para o elemento associação entre classes do UML está diretamente relacionado com outros dois elementos do UML que estão na Tabela 4.1*: **troca de mensagens* e *instanciação de objetos***. A lógica de uma relação de associação é baseada nestes dois últimos elementos. Exemplificando, na associação classe A → classe B, são criadas portas de entidade para comunicação entre essas classes. Cada classe terá duas portas adicionais, uma para receber e outra que enviar/solicitar dados. Adicionalmente, na classe que comanda a comunicação (classe A) é instanciado um componente da outra classe (classe B). Assim, suas portas de entidade são relacionadas internamente para efetivar a comunicação. Dentro da classe B deve ser criado um método que recebe informações da classe A através de uma porta, processa e gera dados que devem ser enviados de volta à mesma classe através da outra porta específica para o retorno. A limitação de multiplicidade existe, pois é considerado que não existirem múltiplos objetos de classes dentro do sistema VHDL. Também, existe a limitação do sentido de navegação ter sido determinado como unidirecional, porém apenas com a replicação da mesma estrutura de comunicação em ambas as classes relacionadas, a navegação poderia ser tornada bidirecional.

4.2.2 Diagrama de sequência

O diagrama de sequência é uma ferramenta UML usada para representar interações entre objetos, realizadas através de operações ou métodos (procedimentos ou funções). Neste trabalho, diagramas de sequência são utilizados para representar a parte dinâmica do sistema, ilustrando através de comportamentos sua lógica de atividade. A figura 4.4 exemplifica a implementação do método *run()* da classe *SmartComponent* na forma de um diagrama de sequência e o código resultante na linguagem VHDL (i.e. *process*).

Métodos UML são mapeados em **processos VHDL** pela semelhança dos seus conceitos. Apesar de a linguagem VHDL tratar processos de um mesmo módulo como estruturas concorrentes, a lógica interna de um processo é sequencial, assim como as ações executadas dentro de um diagrama de sequência.

O diagrama de sequência da figura 4.4 mostra que o método *run()* da classe *SmartComponent* é chamado por alguma estrutura (e.g. *timer*, *schedule*) e desencadeia uma sequência de ações que constituem a lógica daquela funcionalidade do sistema, sendo completamente implementada em hardware. Cada vez que a sequência de ações do método *run()* é executada, a variável *order* é processada e conforme seu estado um conjunto de ações é tomado como ação resultante. No diagrama da figura podemos verificar como devem ser desenvolvidos em UML uma estrutura de teste (e.g. if, opt), as atribuições e uma chamada para método de outra classe.

Quando mapeada para um processo VHDL, aquela sequência de ações do diagrama UML se transforma no código que está dentro do quadro pontilhado da figura acima. Em VHDL o método *run*() se caracteriza como um processo sensibilizado por entradas de *clock* e *reset* externas. Quando sensibilizado pelo reset realiza a iniciação das variáveis do sistema. Caso contrário, a cada borda de subida do *clock*, o sinal *order* é avaliado e ações são tomadas de acordo com o seu estado.

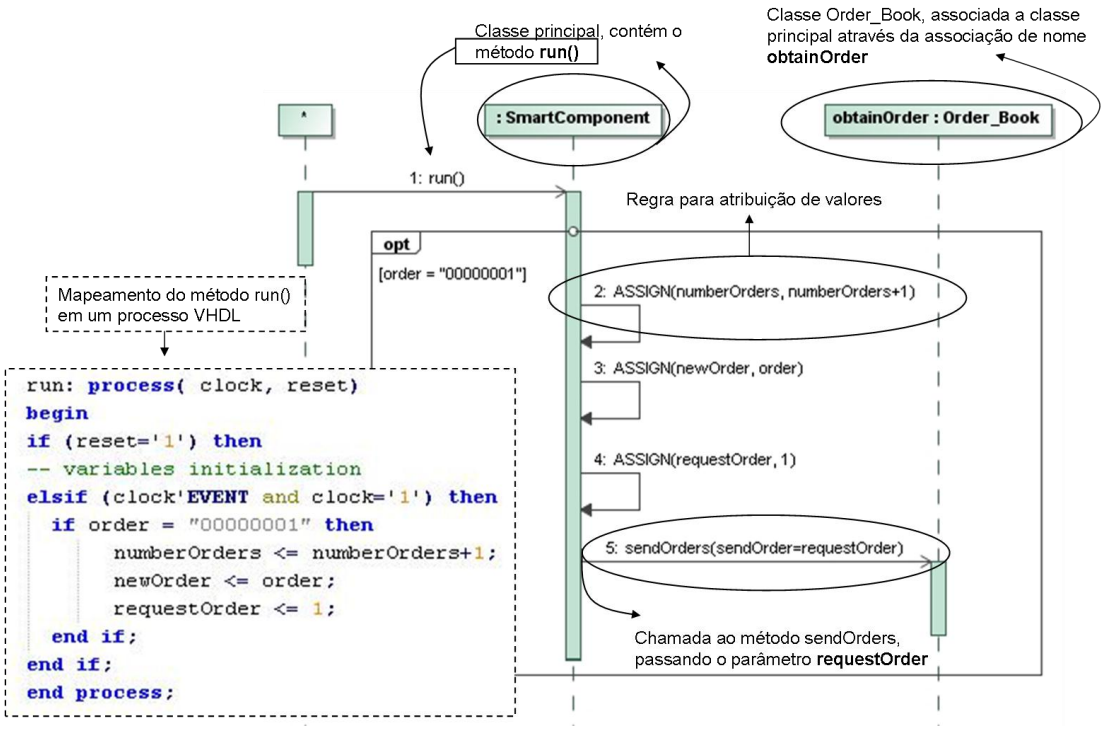

Figura 4.4: Detalhamento do método *run()* de UML para VHDL.

4.2.3 Estruturas importantes não mapeadas

Os mecanismos de herança e de polimorfismo seguem a abordagem desenvolvida em (ECKER, 1996). Por causa da complexidade de implementação, o mapeamento para estas estruturas não foi desenvolvido neste primeiro conjunto de regras de mapeamento. O arquivo que comporta as regras de mapeamento não oferece suporte para o tipo de lógica necessário ao mapeamento. A ferramenta GenERTiCA, motor do processo e das transformações, precisaria ser estendida em diversas funcionalidades somente para o mapeamento destas duas estruturas. Então, optou-se por priorizar a validação das estruturas mais básicas, objetivando para o futuro, a continuidade do trabalho com o desenvolvimento de regras mais elaboradas. Apesar disso, a metodologia para sua implementação pode ser explicada:

Podemos representar a **estrutura de herança** em VHDL utilizando duas palavras chave desta linguagem, chamadas "*new*" e "*tagged*". A marcação *tagged* associada à declaração de uma estrutura indica que a mesma pode ser utilizada, mas que não está totalmente finalizada. A marcação *new* permite declarar novamente uma estrutura (i.e. declarar duas estruturas com o mesmo nome), adicionando elementos diferentes. Assim, tudo que for declarado na segunda estrutura será incorporado na primeira, já que a mesma estava marcada como incompleta. Desta maneira, entende-se que seria possível mapear o comportamento do mecanismo de herança para o VHDL.

Já o mecanismo de **polimorfismo** estático, poderia ser representado através da estrutura *configuration* do VHDL. Essa marcação indica no VHDL que uma mesma estrutura pode ser configurada para executar de maneiras diferentes, conforme a situação de seus sinais de entrada ou sua declaração dentro do código. Esse comportamento é similar ao mecanismo de polimorfismo em linguagens de alto-nível, podendo representar a mesmo como forma de mapeamento ao VHDL.

4.2.4 Considerações

Como uma particularidade, nossa abordagem é orientada para o projeto de sistemas baseados em reuso de componentes, através da integração e adaptação de elementos de Propriedade Intelectual (do inglês, *Intelectual Property* - IP). Componentes do tipo IP são na verdade classes mapeadas para módulos da linguagem VHDL, que definidos de forma genérica e configurável podem ser utilizados em outros projetos que necessitem de estruturas iguais ou semelhantes. Para reusar um módulo VHDL em outro projeto, somente é preciso declarar o componente representativo deste módulo dentro do projeto desejado, semelhante à maneira de instanciar um componente de uma classe no cenário de orientação a objetos. Esses componentes poderiam ser armazenados em repositórios de componentes, formando um framework de componentes reusáveis. Isso facilitaria bastante o desenvolvimento de novos projetos, diminuindo o *Time To Market* (TTM) dos produtos e, consequentemente, o tempo de desenvolvimento e o custo de projetos.

4.3 Mapeamento UML para VHDL

Para gerar código a partir de modelos UML, a ferramenta GenERTiCA adota uma metodologia baseada em *scripts*, onde pequenos *scripts* definem como mapear elementos do modelo em construções da plataforma alvo, gerando fragmentos de código que são reunidos em arquivos formando código fonte. Neste modelo de transformação, cada *script* está direcionado à transformação de um elemento de um modelo único (ou alguns deles) em fragmento de código fonte. Por esta razão, existem diversos *scripts* unitários que precisam ser armazenados em conjunto e dentro de uma estrutura adequada para que possam ser identificados pela ferramenta motor de transformação. Neste trabalho, são armazenados em um arquivo com formato específico que facilita o agrupamento e a identificação de cada *script*. Assim, esse conjunto é denominado arquivo de regras de mapeamento e será totalmente explicado neste capítulo, juntamente com cada grupo de scripts.

4.3.1 Armazenamento das regras de mapeamento

Regras de mapeamento são descritas na forma de um arquivo XML (W3C, 2006), cujo formato é portátil e permite a especificação de conteúdo auto documentado (i.e. o próprio formato descreve a sua estrutura e os nomes dos campos, assim como valores válidos), organizado em uma estrutura na forma de árvore. Segundo Wehrmeister, estas características e o fato do XML ser realmente um formato padrão influenciaram sua escolha como linguagem usada para descrição das regras de mapeamento da ferramenta GenERTiCA (WEHRMEISTER, 2009). Além disso, a organização do XML em árvore facilita a armazenagem dos scripts em termos de repositórios de regras de mapeamento por plataformas, permitindo que os scripts sejam reusados em projetos posteriores que utilizem a mesma plataforma. Por isso, os esforços para o desenvolvimento de um sistema a partir de modelos UML tornam-se menores.

A estrutura em árvore das regras de mapeamento é dividida em ramos que se subdividem até tornarem-se “folhas”, estrutura final, também chamada de *nodo* ou *nó* folha. Os scripts executados para gerar código a partir dos elementos DERCS (representando um elemento UML correspondente) ficam armazenados nas folhas. Como mencionado anteriormente, cada script é direcionado a geração de um fragmento de código relacionado a um único elemento DERCS. O script correto é selecionado

baseado em qual elemento está sendo acessado pelo algoritmo de geração de código. Isto é, o nodo folha que contém o script, deve corresponder exatamente ao elemento sendo avaliado. Estes scripts têm acesso integral às informações contidas no modelo DERCS, para que se tornem habilitados a gerar código mais completo possível. Por conseguinte, uma boa geração de código a partir de scripts significa que o código foi completamente gerado, ou uma grande porção dele, sem nenhum ou pouco esforço do projetista para escrever o código de forma manual. Um dos objetivos da ferramenta GenERTiCA é produzir um código tão completo quanto possível, diminuindo (ou mesmo eliminando) a necessidade da produção de código manual.

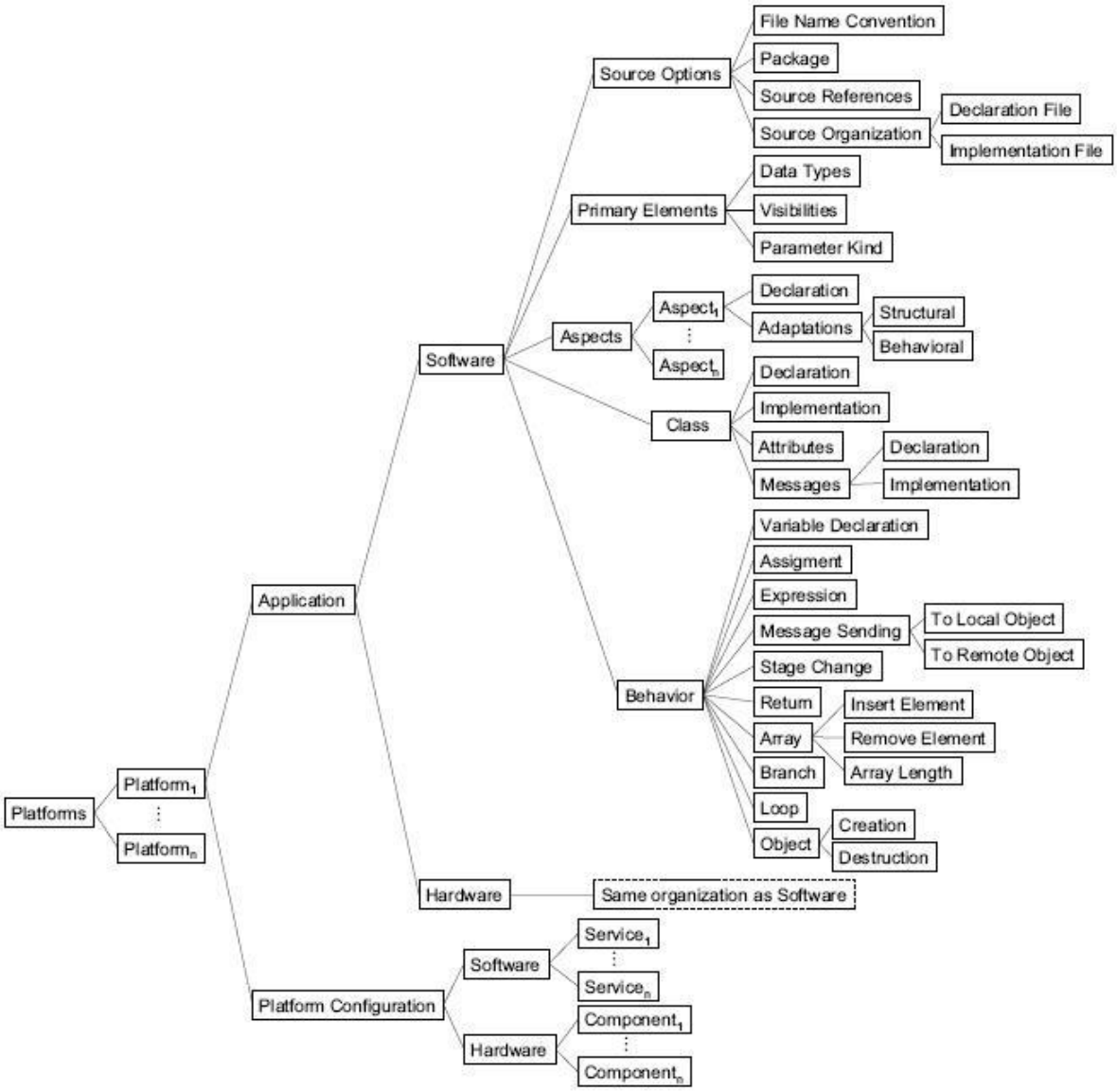

Figura 4.5: Organização das regras de mapeamento.

Este trabalho não define uma linguagem de scripts própria ou um mecanismo de execução de scripts. É utilizado o bem conhecido framework de código aberto chamado *Velocity* (APACHE, 2011), que define a linguagem *Velocity Template Language* (VTL), cuja utilização provê suporte a todas as funcionalidades requisitadas pela ferramenta

GenERTiCA, auxiliando no processo que automatiza as transformações de código. VTL é uma linguagem de scripts estilo Java (*Java-like*). Retorna uma *string* como resultado da execução de cada *script*. Assim, o fragmento de código gerado é obtido por meio do acesso destes scripts as informações do modelo através da interface DERCS API.

Considerando a organização das regras de mapeamento, podemos ver na figura 4.5 que a raiz de arquivos XML é dividida em um conjunto de diferentes plataformas alvo, cada uma com uma árvore-filha que representa o conjunto de regras de mapeamento para aquela plataforma específica. Existem duas categorias de código fonte possível de ser gerado para cada plataforma: *(i)* código para aplicação; e *(ii)* código para plataforma. Ambos se subdividem em código para software e para hardware. No ramo que define código para aplicação, as subárvores de software e hardware têm a mesma estrutura. O que significa que as duas subárvores contêm os mesmos tipos de scripts para gerar código a partir do modelo DERCS. No ramo da plataforma de configuração a diferença entre suas subárvores é que uma produz serviços enquanto a outra produz componentes de hardware.

Em projetos embarcados que utilizam FPGA, normalmente o código VHDL é utilizado para gerar a plataforma de hardware que fornece suporte ao sistema, onde os módulos contidos dentro do FPGA são controlados por processadores externos ou internos, em quais é embarcado o código da aplicação na forma de software. A proposta dessa dissertação difere do usual e sugere que toda aquela lógica de software, seja desenvolvida agora como hardware. Deste modo, tudo que estaria implementado em UML para ser executado dentro de um processador, seria gerado como blocos de hardware, tendo comportamento análogo à lógica de software. Além disso, a substituição de software por hardware traz consigo, um aumento considerável no desempenho do sistema, reduzindo o tempo de projeto e, consequentemente, o seu custo. Deste modo, pelo fato de implementar hardware utilizando os mesmos meios de software, as regras de mapeamento deste trabalho, destinadas a gerar código VHDL, utilizam como base para o armazenamento de seus *scripts* a ramificação orientada para software no ramo *(i)* código para aplicação.

4.3.2 Regras de Mapeamento

Na árvore de regras de mapeamento, o ramo de aplicação de software/hardware é subdividido em: *(i)* configuração do código fonte resultante; *(ii)* script para os elementos primários; *(iii)* scripts para elementos relacionados às classes; *(iv)* scripts relacionados aos elementos de comportamento; e *(v)* scripts para especificação da implementação dos aspectos do framework DERAF.

4.3.2.1 Configuração do código fonte resultante

As linhas de 001-026 da figura 4.6 remetem ao trecho de código do arquivo XML que contém as regras de mapeamento correspondentes ao item *(i)*, em qual existem opções para configuração do código fonte resultante. GenERTiCA utiliza essas informações para parametrizar o resultado do processo de geração de código. O nodo <*SourceOptions*> gerencia as questões relacionadas à criação dos arquivos de código fonte, definindo convenções para nome e organização interna. Dentro deste nodo existem atributos, que diretamente configuram alguns itens, e outros nodos, portadores de *scripts*, que capturam informações dinâmicas diretamente do modelo DERCS.

```
<!-- Source code generation options -->
001 <SourceOptions
002 isAspectLanguage="no"
003 ClassesPerFile="1"
004 hasClassesDeclaration="no"
005 Identation="5"
006 BlockStart="begin"
007 BlockEnd="end">
008 <FileNameConvention>
009   $Class.Name
010 </FileNameConvention>
011 <Package></Package>
012 <SourceReference>
013 </SourceReference>
014 <SourceOrganization>
015   <DeclarationFile
016   FileExtension="">
017   </DeclarationFile>
018   <ImplementationFile
019   FileExtension=".vhd">
020   $SourceCode.PackagesDeclaration
021   \n$SourceCode.ReferencesDeclaration
022   \n$SourceCode.ClassesDeclaration
023   \n$SourceCode.ClassesImplementation
024   </ImplementationFile>
025 </SourceOrganization>
026 </SourceOptions>
```

```
<!-- Mapping rules for PRIMARY ELEMENTS -->
027 <PrimaryElementsMapping>
028 <DataTypes>
029   <Array></Array>
030   <Boolean>BOOLEAN</Boolean>
031   <Byte>
032     STD_LOGIC_VECTOR(7 downto 0)
033   </Byte>
034   <Char>CHARACTER</Char>
035   <Class></Class>
036   <DateTime></DateTime>
037   <EnumerationDefinition>
038   </EnumerationDefinition>
039   <Enumeration></Enumeration>
040   <Integer>INTEGER</Integer>
041   <Long></Long>
042   <Short></Short>
043   <String></String>
044   <Void></Void>
045   <Double></Double>
046   <Float>REAL</Float>
047 </DataTypes>
048 <Visibilities>
049   <Private></Private>
050   <Protected></Protected>
051   <Public></Public>
052 </Visibilities>
053 <ParameterKinds>
054   <In></In>
055   <Out></Out>
056   <InOut></InOut>
057 </ParameterKinds>
058 </PrimaryElementsMapping>
```

Figura 4.6: Regras de mapeamento para (i) Configuração do código resultante e (ii) Elementos primários.

Ainda dentro do nodo <*SourceOptions*> é importante salientar dois atributos: *isAspectLanguage* e *hasClassesDeclaration*. O primeiro indica que a linguagem alvo é uma linguagem orientada a aspectos, e habilita a ferramenta GenERTiCA para realizar o entrelaçamento de aspectos, caso existam regras para este fim. O outro atributo, *hasClassesDeclaration,* indica se a linguagem alvo requisita ou não uma declaração de classe antes da implementação da mesma, tal como na linguagem C/C+. Os outros atributos são para configuração de identação, marcas de início e fim de blocos de código, e quantidade de classes por arquivo.

Também são importantes os nodos <*FileNameConvention*>, que é responsável pela definição do nome dos arquivos, e <*SourceOrganization*>, que define a organização interna dos arquivos que são gerados. A ferramenta GenERTiCA assume que a linguagem alvo pode ter tanto arquivos de declaração quanto arquivos de implementação, tal como em C/C++, onde arquivos de cabeçalho e implementação são definidos separadamente. Assim, a partir do nodo <*SourceOrganization*> o projetista pode configurar a forma como a ferramenta irá criar e parametrizar esses arquivos. Se existe algum tipo de dependência entre arquivos, o nodo <*SourceReference*> contém as construções da linguagem alvo para fazer a referências entre eles. No nodo <*ImplementationFile*> é configurada a extensão do arquivo que será gerado e a ordem de suas estruturas internas. A extensão é configurada de forma direta, através do atributo *FileExtension*. Já a ordem das estruturas internas é dada a partir da ordem cronológica dos scripts que são chamados neste nodo. Por exemplo, na linha 022 está o *script* que insere no arquivo um bloco de código referente à declaração da classe, e na linha 023 está o *script* responsável por inserir o código da implementação daquela mesma classe.

4.3.2.2 Elementos primários

As linhas de 027-058 da figura 4.6 exibem as regras de mapeamento referentes ao item *(ii)*, definidas para os elementos primários. Este item é divido em: *DataType*, que se refere aos tipos de dado existentes no código, *Visibilities*, que trata das visibilidades que podem ser dadas as variáveis, e *ParameterKinds*, que define o tipo do parâmetro, podendo este ser de entrada, saída ou ambos ao mesmo tempo.

Utilizando somente o diagrama de classes no UML não foi possível encontrar correspondência para desenvolver regras ao mapeamento dos tipos de parâmetros (i.e. *ParameterKinds)*, por isso estes elementos são sempre gerados como entrada e saída (i.e. *inout*). Configurações de visibilidade (i.e. *Visibilities*) também não foram consideradas, pois no VHDL estes atributos não têm marca específica para visibilidade, existindo apenas a distinção entre atributos privados e públicos que determina como são declarados no VHDL, respectivamente, como sinais internos ou portas de entidade.

Ao mapeamento dos tipos de dado (i.e. *DataType*) foram criadas conversões diretas. Isto é, não dependem de *script* para gerar o código fonte resultante. No momento da conversão, GenERTiCA encontrará como regra de mapeamento somente um valor a ser substituído de forma direta pelo elemento do modelo DERCS sob análise. A figura 4.7 ilustra o mapeamento de dois atributos com o tipo de dado "Byte". Primeiramente, o modelo UML é transformado no modelo DERCS. Então, a ferramenta GenERTiCA toma um a um os elementos do modelo DERCS, buscando no arquivo de regras de mapeamento o elemento correspondente para efetuar a conversão. O exemplo da figura caracteriza o mapeamento do elemento byte, onde o motor de transformação realizaria o mapeamento das variáveis, uma privada e outra pública, para uma porta e um sinal do VHDL, e na sequência realizaria o mapeamento do tipo de dado de ambas, de <Byte> para um vetor de bits do tipo *standard logic*: STD_LOGIC_VECTOR(7 downto 0).

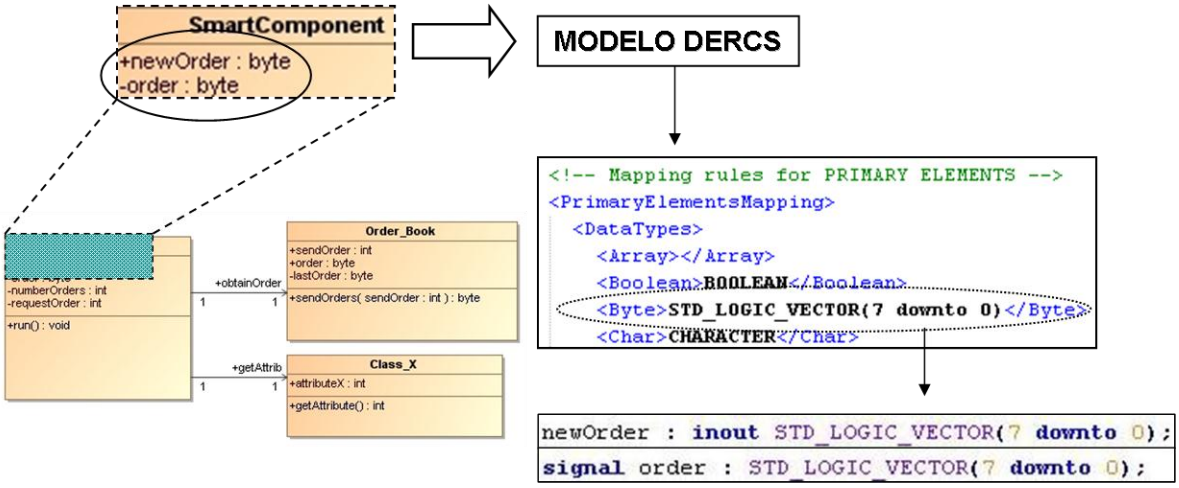

Figura 4.7: Regra de mapeamento para tipo de dado "Byte".

Adicionalmente, existem regras para outros tipos de dados. Para elementos do tipo boleano (i.e. *boolean*) existe a estrutura BOOLEAN do VHDL. Para elementos do tipo caractere (i.e. *char*), também existe em VHDL uma estrutura que representa o mesmo tipo de dado: CHARACTER. Aos elementos do tipo inteiro (i.e. *integer*) existe a

estrutura INTEGER no VHDL. E, os elementos do tipo ponto flutuante (i.e. *float*) são convertidos para o tipo REAL do VHDL que tem o mesmo significado.

Para outros elementos, como por exemplo, os tipos de dado *Array* e *String*, também existem estruturas que poderiam representar estes tipos de dado em VHDL. Contudo, o mapeamento para essas estruturas não seria direto e envolveria o desenvolvimento de regras de mapeamento complexas, que não foram implementadas neste primeiro conjunto de regras de mapeamento, ficando como melhorias para um próximo trabalho.

4.3.2.3 Elementos relacionados às classes

As próximas figuras (4.8, 4.9 e 4.10) exibem trechos do arquivo de regras de mapeamento que tratam da transformação de elementos e estrutura das classes. A figura 4.8 traz a regra de mapeamento que foi desenvolvida para representar em VHDL a declaração de uma classe do UML. A regra é o código que está entre as linhas 060-099. Trata-se de uma estrutura padrão (*template*) na linguagem VHDL, com uma série de *scripts* que configuram a estrutura em tempo de execução. Isto é, quando esse ramo das regras de mapeamento é executado pela ferramenta GenERTiCA, os *scripts* buscam do modelo DERCS informações específicas da classe em transformação. Assim, segundo a sua lógica, eles constroem uma nova estrutura, que resulta na declaração completa de uma entidade VHDL, estrutura representativa do mapeamento das classes.

```
<!-- Mapping rules for CLASSES -->
059 <Classes>
060 <Declaration>
061 -- Basic libraries
062 \nlibrary IEEE;
063 \nuse IEEE.STD_LOGIC_1164.ALL;
064 \nuse IEEE.NUMERIC_STD.ALL;
065 \n
066 \nentity $Class.Name is
067 \nPort (
068 \nclock : in STD_LOGIC;
069 \nreset : in STD_LOGIC;
070 #set( $Mth = "_mth")
071 #set( $Ret = "_ret")
072 #if ($Class.getAttributesCount() > 0)
073   #foreach ($Attribute in $Class.getAttributes())
074     #if ($DERCSHelper.isPublic($Attribute))
075       #if (!$Attribute.getDataType().getRepresent())
076         \n $Attribute.Name :
077            inout $CodeGenerator.getDataTypeStr($Attribute.getDataType());
078       #else
079         #foreach($Method in $DERCSHelper.getListOfMethodsAinB(
080                  $Class, $Attribute.getDataType().getRepresent()))
081           #if ($Method.getParametersCount() > 0)
082             #foreach($Parameter in $Method.getParameters())
083              \n $Method.Name$Mth :
084                 out $CodeGenerator.getDataTypeStr($Parameter.getDataType());
085             #end
086           #else
087             \n $Method.Name$Mth : out STD_LOGIC;
088           #end
089           \n $Method.Name$Ret :
090              in $CodeGenerator.getDataTypeStr($Method.getReturnType());
091         #end
092       #end
093     #end
094   #end
095 #end
096 \n
097 );
098 \nend $Class.Name;
099 </Declaration>
```

Figura 4.8: Regras de mapeamento para declaração de classes.

Cada classe existente no modelo UML é declarada como uma entidade. É realizada uma busca dentre os atributos da classe, para identificar quais são atributos públicos e declará-los como portas de entrada e saída (*inout*) da entidade. Seguindo a mesma lógica, faz-se uma busca entre as associações daquela classe, buscando pela chamada de métodos. Quando a classe realiza a chamada de um método através da associação, duas outras portas são criadas na entidade: uma como entrada, para receber o retorno daquele método; e outra como saída para fornecer a chamada do método. Ambas com seus nomes precedidos de "_mth" e "_ret", sinalizando método e retorno, respectivamente.

```
100 <Implementation>
101 \narchitecture Behavioral of $Class.Name is
102 \n$CodeGenerator.getAttributesDeclaration(1)
103 \n
104 #if ($Class.getAttributesCount() > 0)
105   #foreach ($Attribute in $Class.getAttributes())
106     #if ($Attribute.getDataType().getRepresent())
107       #set($NewClass = $Attribute.getDataType().getRepresent())
108       \ncomponent $NewClass.Name
109       \n port (
110       \n   clock : in STD_LOGIC;
111       \n   reset : in STD_LOGIC;
112       #foreach($Attribute in $NewClass.getAttributes())
113         #if ($DERCSHelper.isPublic($Attribute))
114           \n $Attribute.Name :
115           inout $CodeGenerator.getDataTypeStr($Attribute.getDataType());
116         #end
117       #end
118       \n );
119     #end
120   #end
121 #end
122 \n
123 \n$Options.BlockStart
124 \n
125 #if ($Class.getAttributesCount() > 0)
126   #foreach ($Attribute in $Class.getAttributes())
127     #if ($Attribute.getDataType().getRepresent())
128       #set($NewClass = $Attribute.getDataType().getRepresent())
129       #set( $Mth = "_mth")
130       #set( $Ret = "_ret")
131       \n port map(
132       \n   clock =&lt; clock,
133       \n   reset =&lt; reset,
134       #foreach($Attribute in $NewClass.getAttributes())
135         #if ($DERCSHelper.isPublic($Attribute))
136           \n $Attribute.Name =&lt; x,
137         #end
138       #end
139       \n );
140     #end
141   #end
142 #end
143 \n
144 \n$CodeGenerator.getMessagesImplementation(1)
145 \n
146 \nend Behavioral;
147 </Implementation>

148 <Attributes>
149 #if ($DERCSHelper.isPrivate($Attribute))
150   signal $Attribute.Name : $DataTypeStr;
151 #end
152 </Attributes>
```

Figura 4.9: Regras de mapeamento para implementação das classes.

A regra de mapeia classes UML em entidades VHDL é apresentada pela figura 4.9. A linha 101 contém a declaração **architecture Behavioral of $Class.Name is**, que representa a implementação de uma arquitetura para representar o comportamento daquela entidade. Esta declaração abre um bloco de código para o desenvolvimento de sua arquitetura, sendo finalizado através da marcação **end Behavioral** da linha 146.

A lógica desta regra declara *signals* para cada atributo privado da classe e instancia objetos para cada classe associada através da estrutura *component*, criando também seus mapeamentos de portas. Adicionalmente, implementa *processes* com sequências de atividades para cada método da classe que tenha um diagrama de sequência.

```
153 <Messages>
154   <Declaration></Declaration>
155   <Implementation>
156   #if ($Message.Name != $Class.Name)
157     #if (!$Message.isGetSetMethod())
158       \n${Message.Name}: process(
159       #if ($Message.Name == "run")
160         #if ($Message.ParametersCount > 0)
161           clock, reset,
162           #foreach( $param in $Message.Parameters )
163             #if ($velocityCount > 1), #end
164             $param.Name
165           #end
166         #else
167           clock, reset
168         #end
169       #else
170         #foreach( $param in $Message.Parameters )
171           #if ($velocityCount > 1), #end
172           $param.Name
173         #end
174       #end
175       )
176       \n$Options.BlockStart
177       #if ($Message.Name == "run")
178         \n
179         \nif (reset='1') then
180         \n-- variables initialization
181         #if ($Class.getAttributesCount() > 0)
182           #foreach( $Attribute in $Class.getAttributes() )
183             #if ($DERCSHelper.isPrivate($Attribute))
184               #if( $Attribute.getDataType().toString() == "Integer" )
185                 \n $Attribute.Name &lt;= 0;
186               #elseif( $Attribute.getDataType().toString() == "Boolean" )
187                 \n $Attribute.Name &lt;= FALSE;
188               #elseif( $Attribute.getDataType().toString() == "Byte" )
189                 \n $Attribute.Name &lt;= "00000000";
190               #elseif( $Attribute.getDataType().toString() == "Float" )
191                 \n $Attribute.Name &lt;= 0;
192               #end
193             #end
194           #end
195         #end
196         \nelsif (clock'EVENT and clock='1') then
197       #end
198       \n$CodeGenerator.getVariablesDeclaration(1)
199       \n$CodeGenerator.getActionsCode(1, 1)
200       #if ($Message.Name == "run")
201         \nend if;
202       #end
203       \n$Options.BlockEnd process;
204     #end
205   #end
206   </Implementation>
207 </Messages>
208 </Classes>
```

Figura 4.10: Regras de mapeamento para implementação de métodos.

Ainda na figura 4.9, entre as linhas 148-152, encontra-se a regra que determina o mapeamento dos atributos privados. É uma regra simples, que apenas verifica quais atributos da classe são privados e os declara como *signals* através da *template* da linha 150, **signal $Attribute.Name : $DataTypeStr**, cujas variáveis (i.e. $variável) são preenchidas pela ferramenta GenERTiCA através dos dados do modelo DERCS. Essa regra é executada pelo script **$CodeGenerator.getAttributesDeclaration(1)**, linha 102, cujo processamento resulta na declaração de atributos da classe, identando o resultado com um (1) espaço padrão.

A figura 4.10 apresenta a regra que converte métodos de classe UML em processos VHDL. Essa regra define o tipo de estrutura *message*, que representa os métodos. É chamada como parte da regra que implementa a arquitetura da entidade (figura 4.9), na linha 144, **$CodeGenerator.getMessagesImplementation(1)**. Quando processada resulta um bloco de código que representa o mapeamento das ações desenvolvidas para aquele método através do diagrama de sequência UML.

Esta regra cria um *process* VHDL para cada método desenvolvido no modelo UML. O *process* criado contém os parâmetros e argumentos daquele método. Se o método for do tipo *run()* serão incluídos sinais de *clock* e *reset*, pois métodos deste tipo necessitam destes sinais para, respectivamente, reiniciar o sistema e realizar seu processamento.

O código das ações dos métodos é gerado a partir de regras que mapeiam comportamentos. O *script* da linha 199, **$CodeGenerator.getActionsCode(1, 1)**, processa todos os itens referentes à comportamento, devolvendo como resultado um bloco de código que representa o conjunto de ações do diagrama de sequência UML desenvolvido para aquele método. Adicionalmente, a concorrência permitida pelo hardware é um detalhe importante que deve ser considerado pelo projetista no desenvolvimento de um modelo destinado à geração de código VHDL. Pois, cada método UML será gerado como processos (VHDL) que executam em paralelo uns com os outros. Isso exige atenção no uso de variáveis compartilhadas, a fim de evitar erros e incoerências de comportamento. Também, limita as possibilidades do projetista, ao mesmo tempo em que oportuniza a situação de paralelismo tão desejada em software.

4.3.2.4 Elementos de comportamento

A figura 4.11 contém as regras de mapeamento referentes aos elementos de comportamento. Esse conjunto de regras é responsável pela transformação dos elementos de ação do diagrama de sequência UML em estruturas da linguagem VHDL. Neste grupo, foram desenvolvidas regras para declaração e atribuição de variáveis, teste de condições (*if*) e estruturas para loop (*for, while*). A estrutura do arquivo de regras de mapeamento ainda comportaria regras e *scripts* para mapear elementos relacionados a objetos, expressões, envio de mensagem entre objetos locais e remotos, e uso completo de elementos de *array*. Porém, estes elementos representam estruturas complexas, que não foram estudadas e nem tiveram seus conceitos mapeados para VHDL no âmbito deste trabalho.

O bloco *Behavior* da figura 4.11 engloba as regras de mapeamento para todas as estruturas de comportamento. Neste bloco, a primeira regra representa a declaração de variáveis. Trata-se da regra contida entre as linhas 211-213, cujo *script* **$CodeGenerator.getVariablesDeclaration(1),** localizado na linha 199 da figura 4.10, ao ser processado apenas retorna o tipo e o nome do elemento sob análise naquele instante.

```
<!-- Mapping rules for BEHAVIOR, i.e. sequence of actions -->
<Behavior>
<VariableDeclaration>
 $DataTypeStr $Variable.Name;
</VariableDeclaration>

<Branch>
if $Branch.EnterCondition then
#set( $ident = $IdentationLevel + 0)
\n$CodeGenerator.getVariablesDeclaration($ident)
\n$CodeGenerator.getActionsCode($ident)
\nend if;
#if ( $Branch.hasAlternativeBehavior() )
  \n elsif $Branch.AlternativeCondition then
  \n$CodeGenerator.getVariablesDeclaration($Branch.AlternativeBehavior,$ident)
  \n$CodeGenerator.getActionsCode($Branch.AlternativeBehavior,$ident)
  \nend if;
#end
</Branch>

<Loop>
#if ($Loop.NumberOfRepetitions > 0)
  for(int $IndexVariableName = 0; $IndexVariableName &lt;
          $Loop.NumberOfRepetitions; $IndexVariableName++)
#elseif ($Loop.ExitCondition)
  #if ($Loop.EnterCondition)
    ${Loop.EnterCondition};
  #end
  \n while ($Loop.ExitCondition)
#end
$Options.BlockStart
\n$CodeGenerator.getVariablesDeclaration(1)
\n$CodeGenerator.getActionsCode(1)
\n$Options.BlockEnd
</Loop>

<Assignment>
#if ($Action.isVariableAssignment())
  $Action.Variable.Name
#else
  #if ($Action.Object)
    ${Action.Object.Name}.${Action.Attribute.Name}
  #else
    ${Action.Attribute.Name}
  #end
#end
&lt;=
#if ($Action.isAssignmentOfValue())
  $Action.Value;
#else
  $CodeGenerator.getActionCode($Action.Action)
#end
</Assignment>

<Object>
  <Creation></Creation>
  <Destruction></Destruction>
</Object>

<Expression></Expression>
<Return></Return>
<StateChange></StateChange>

<SendMessage>
  <ToLocal>
    <Software></Software>
    <Hardware></Hardware>
  </ToLocal>
  <ToRemote>
    <Software></Software>
    <Hardware></Hardware>
  </ToRemote>
</SendMessage>
```

```
275 <InsertArrayElement></InsertArrayElement>
276 <RemoveArrayElement></RemoveArrayElement>
277 <GetArrayElement></GetArrayElement>
278 <SetArrayElement></SetArrayElement>
279 <ArrayLength></ArrayLength>
280 </Behavior>
```

Figura 4.11: Regras de mapeamento para elementos de comportamento.

A regra que mapeia as estruturas de teste condicional está entre as linhas 213-225. Quando convertidas, as estruturas **opt** (*option*) e **alt** (*alternatives*) do diagrama de sequência UML, são mapeadas na estrutura de testes condicionais **if**, que também existe na linguagem VHDL. A utilização destas estruturas pode ser analisada na figura 4.4, onde está demonstrado um exemplo do mapeamento de um diagrama de sequência para um processo. A regra que define este mapeamento entre as estruturas é um *template* de código VHDL que contém *scripts* capazes de retornar as variáveis que foram declaradas dentro do laço condicional e o bloco de código da ação completa. Assim, esse *template* é preenchido com os dados de declaração de variáveis, que retornam através do processamento do *script* da linha 216, e com o código de ação do teste condicional, que retorna através do *script* da linha 217, formando o código VHDL gerado ao elemento.

Entre as linhas 226-240 está a regra que mapeia as estruturas de iteração controlada (*loop*). A regra existente analisa a condição de entrada do elemento a ser mapeado, podendo gerar duas estruturas diferentes. Se o elemento tem uma pré-condição de entrada, definindo o número de iterações que será executado (linha 227), esse elemento é mapeado para estrutura **for**. Caso exista somente uma condição de saída (linha 230), esse elemento é mapeado para estrutura **while**. O restante da regra é igual para ambas, onde existem scripts para abrir e fechar o laço de iteração e para retornar as variáveis declaradas e o código da ação.

As ações de atribuição (*assignment*) são definidas pela última regra válida do bloco *Behavior*. O restante do arquivo XML contém entradas para outras regras que não foram desenvolvidas. A regra de atribuição é um *template*, cujo *script* principal busca elementos do tipo atribuição dentro do código de ação sob análise. Encontrando elementos de atribuição, outros scripts são executados e retornam o nome da variável e o valor que lhe será atribuído. Desta forma, preenche a *template* com os dados retornados e monta o código VHDL correspondente. Contudo, esse mapeamento somente acontece da forma correta caso a modelagem do diagrama de sequência siga uma regra para definição deste tipo de ação. Conforme é possível identificar a partir da figura 4.4, qualquer ação de atribuição deve ser modelada através da estrutura ASSIGN, no formado que é demonstrado na figura mencionada. Modelada de outro modo esse elemento não seria reconhecido pela ferramenta de transformação, que não iria criar o elemento referente a esta ação no modelo DERCS. O resultado disso seria que quando essa regra de mapeamento fosse processada, nenhum dado (código) seria retornado como resultado, pois não existiria no modelo intermediário.

A Tabela 4.2 apresenta uma lista de elementos que dependem de regras de modelagem para serem mapeados corretamente. Essas regras são essenciais, imprimindo ao modelo o nível de padronização necessário à correta interpretação das estruturas e elementos modelados em UML pela ferramenta que suporta a geração de código (GenERTiCA). Maiores informações sobre os processos executados pela ferramenta suporte e sobre regras de modelagem podem ser encontrados na referência (WEHRMEISTER, 2009).

Tabela 4.2 : Regras de modelagem para um correto mapeamento.

Elemento	**Norma de modelagem UML**
Classe	Simples, sem classe abstrata, herança ou polimorfismo
Atributos	Público e Privado, sem modo Protegido
Métodos	Argumentos e lógica sem *return*
Associação entre classes	Unidirecional, 1-to-1 e com nome
Diagrama de sequência	Usar somente estruturas permitidas
Loop	Loop
Atribuição	ASSIGN([data type] target, value)
Estruturas condicionais	Alternatives (alt), Option (opt) e Parallel (par)

4.3.2.5 Especificação da implementação dos Aspectos

Ainda se tratando do arquivo de regras de mapeamento, além da sequência das regras descritas acima, este arquivo comporta também *scripts* para o mapeamento de requisitos não-funcionais relacionadas aos aspectos. Durante a fase de modelagem aqueles requisitos não-funcionais identificados para o sistema, são modelados na forma de aspectos, que ao final do processo, incidem sobre o resultado da geração de código fonte, com a inserção de novos elementos e alterações de código, para adicionar as características daquele requisito não-funcional identificado.

Aspectos são identificados na fase de análise de requisitos e reproduzidos na fase de modelagem através dos diagramas ACOD e JPDD, e também, a partir de estereótipos (<*Stereotypes*>) usados para identificar quais os elementos do modelo que portam requisitos não-funcionais. Assim como os demais elementos do modelo, aqueles elementos marcados como portadores de requisitos não-funcionais também são transformados em elementos do modelo DERCS e, posteriormente, gerados como código VHDL. Ao final do processo de geração de código convencional, a ferramenta GenERTiCA executa a etapa de *Aspect Weaving*, conhecida como entrelaçamento de aspectos. Onde, as regras de mapeamento para os aspectos são executadas e adicionam ao código, já existente, elementos e estruturas que caracterizam o aspecto em foco no processo de GenERTiCA naquele instante. A ferramenta processa essa mesma rotina para todos os aspectos que tenham sido representados no modelo, realizando o entrelaçamento de aspectos, que significa a sobreposição de características (de aspectos), naqueles elementos que são afetados por mais de um aspecto.

Apesar de suportadas pela metodologia e pela ferramenta GenERTiCA, nenhuma das características relacionadas aos Aspectos foi desenvolvida neste conjunto de regras de mapeamento.

Desenvolver em VHDL características não-funcionais não é uma tarefa simples. Por exemplo, adicionar restrições temporais ou de área física em um código VHDL significa não apenas adicionar novos trechos de código, ou sobrescrever trechos antigos, mas também implementar regras de mapeamento para gerar código VHDL genérico, capaz de executar comportamentos distintos para cada tipo de situação diferente, previamente conhecida. Além disso, o mecanismo para inserção e entrelaçamento destas características no código VHDL também seria complexo, pois o processo de declaração de variáveis e instanciação de objetos é descontinuado e diferente da orientação a objetos, onde as declarações são realizadas de forma pontual e simplificadas.

Adicionalmente a estes pontos, ainda existe a questão de otimização que está estritamente relacionada à forma como o código VHDL é implementado. Por exemplo, um aspecto representando características de área, poderia implicar ao código restrições na quantidade de área de FPGA (células lógicas) utilizadas na implementação de certas estruturas. Ou ainda, um aspecto representando características de desempenho poderia implicar na implementação de estruturas focadas em performance. Entretanto, a melhor forma de desenvolver essas características em VHDL depende exclusivamente do sistema que está sendo implementado. Dependendo do sistema essas estruturas podem ser desenvolvidas de uma forma ou de outra. Como a ferramenta de geração de código precisa ser genérica para atender a qualquer tipo de sistema, ela contém modelos de soluções-padrão. Assim, essas soluções poderiam resultar excelentes para alguns problemas e muito ruins para outros, sendo até mesmo proibitivas em certos casos. Desta forma, esse conjunto de fatores inviabiliza o desenvolvimento de regras para aspectos, ficando o desenvolvimento destas características como possíveis melhorias para trabalhos futuros.

5 VALIDAÇÃO EXPERIMENTAL

Este capítulo apresenta um estudo de caso desenvolvido com a aplicação da proposta de metodologia e regras de mapeamento deste trabalho. Trata-se de um exemplo completo do processo de geração automática de código VHDL a partir do modelo UML de um sistema embarcado desenvolvido para fins de manutenção preventiva. O sistema sob avaliação é composto por uma válvula automática, cuja funcionalidade é regular o fluxo de fluídos, e um conjunto sensores, que servem para obter informações sobre os estados da válvula. Este sistema é parte de uma plataforma experimental chamada CISPI (*Conduite de grands systèmes industriels à risque*)[8], que em tradução livre ao português significaria "Gerenciamento de sistemas industriais de larga escala sob risco". Esta plataforma está localizada no laboratório CRAN (*Centre de Recherche en Automatique de Nancy*)[9], onde foi desenvolvido este estudo de caso com o intuito de viabilizar a utilização desta proposta em sistemas reais e de grande porte.

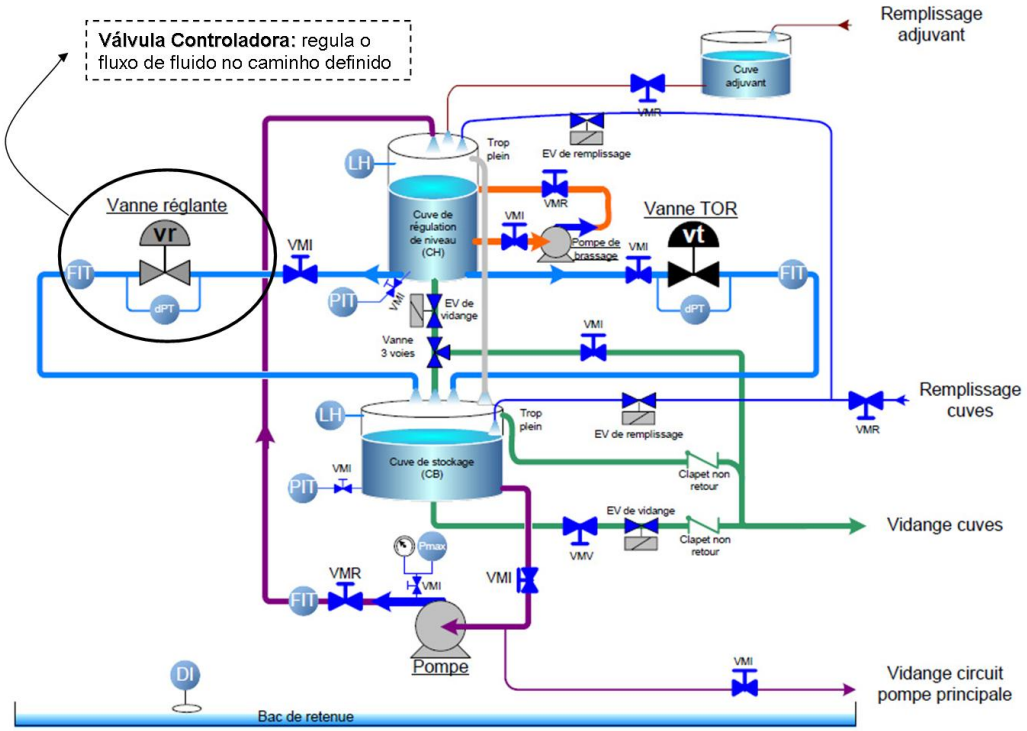

Figura 5.1: Plataforma CISPI – Plano de circulação de fluidos (CRAN, 2011).

[8] http://safetech.cran.uhp-nancy.fr/

[9] http://www.cran.uhp-nancy.fr/

Através da aplicação da metodologia, a intenção foi integrar novas funcionalidades suportadas por um FPGA nessa válvula, desenvolvendo desta forma um *componente inteligente* (IUNG *et al.*, 2001; PÉTIN *et al.*, 1998). Componentes ou Dispositivos inteligentes são sistemas, normalmente embarcados, que fazem uso de capacidades de processamento e inteligência para executar atividades específicas, como no exemplo desta aplicação, **manutenção inteligente**. Levando em consideração o fato de que máquinas usualmente não falham "de repente", mas sim através de um processo mensurável de degradação antes de falhar, a ideia básica de sistemas de manutenção inteligente ou sistemas de manutenção preventiva é usar informações oriundas de sensores e sistemas computadorizados embarcados em dispositivos, para aplicar algoritmos de estimação de saúde (*health estimation*) e prevenção de falhas (*failure predictition*), para estimar ou prever o momento correto para a manutenção ou troca deste componente, ou partes do seu sistema, evitando assim que a ocorrência da falha traga consigo maiores estragos no sistema como um todo.

Componentes inteligentes representam sistemas mecatrônicos que consistem de partes mecânicas, eletrônicas e elementos computacionais. Sua parte eletrônica aliada à computacional configura o sistema de controle do dispositivo. Este sistema de controle é composto por funções lógicas que executam sob uma plataforma de hardware específica. Estas funcionalidades lógicas representam o comportamento deste componente, e usualmente são desenvolvidas como software. O objetivo do trabalho desenvolvido em cima desta válvula foi implementar em hardware todas aquelas funcionalidades lógicas de controle do sistema. Assim, este sistema foi modelado em UML e transformado em código VHDL através da metodologia proposta como foco desta dissertação. O código gerado representa a descrição de hardware das funções lógicas da parte de controle do sistema, cuja execução aconteceria dentro de um FPGA (plataforma de hardware).

5.1 Etapa 1 – Diagrama de caso de uso

Seguindo a metodologia definida na seção 4.1, a primeira etapa para o desenvolvimento do estudo de caso deste trabalho é relativa à análise e identificação de requisitos. Neste ponto é realizado o desenvolvimento de um diagrama de caso de uso, através do qual, é possível de identificar os atores que interagem com a válvula automática, e os serviços que ela deve oferecer. Para este exemplo, foram considerados apenas dois (2) atores interagindo com a válvula. Essa interação dá-se na forma de requisição de serviços. Um dos atores é o Usuário, que requisita o serviço básico de abrir e fechar a válvula. O outro ator é o Operador, cuja função é realizar manutenção na válvula, podendo requisitar, além dos serviços do Usuário, outro serviço que retorna o número de ciclos de abertura/fechamento da válvula. Informação que é utilizada para avaliar a saúde física do componente.

A figura 5.2 exibe o diagrama de caso de uso desenvolvido nesta etapa do estudo de caso. Nele estão demonstradas as características do sistema, onde a interação dos atores externos gera as demandas principais (demanda), ocasionando ações de controle do processo de abrir e fechar a válvula (serviço 1), e também, ações de registro e processamento dos dados e do algoritmo utilizado para estimação da saúde física do componente (serviço 2). Devido à simplicidade de interpretação desta etapa, as planilhas para análise e identificação de requisitos funcionais e não funcionais propostas

como parte deste primeiro passo da metodologia não foram desenvolvidas, permitindo que o processo passe deste ponto diretamente para fase seguinte, de modelagem.

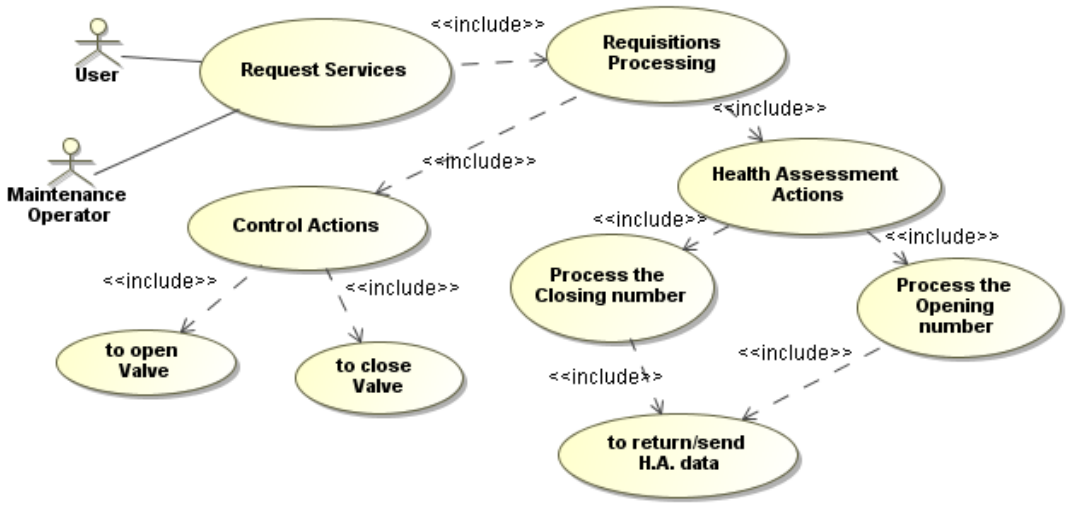

Figura 5.2: Diagrama de caso de uso desenvolvido para o sistema da válvula.

5.2 Etapa 2 – Desenvolvimento do modelo

A segunda etapa da metodologia AMoDE-RT consiste na modelagem UML do sistema proposto, onde são desenvolvidos os diagramas utilizados como base à transformação entre as linguagens: diagrama de classe e diagrama de sequência.

5.2.1 Diagrama de classes principal

O diagrama de classe de um modelo UML representa a estrutura física do sistema, sendo construído a partir de toda a informação adquirida do diagrama de caso de uso. As requisições de demanda e os serviços identificados no caso de uso são modelados como entidades secundárias do modelo. Essas entidades secundárias são classes que trabalham em conjunto para fornecer os serviços oferecidos pelo sistema. Esse grupo de entidades secundárias compõe uma entidade principal, aquela responsável por controlar todo o sistema. Neste caso, a função da entidade principal é requisitar informações da entidade secundária responsável pelas demandas dos atores (usuário, operador de manutenção) e processar tais demandas, gerando ações para as entidades secundárias responsáveis por controlar a válvula e por avaliar a saúde física do dispositivo.

Por exemplo, uma demanda do usuário requisitaria o serviço de abrir ou fechar a válvula, que resultaria, como ação interna, o serviço de registrar o número de vezes que esse componente foi aberto e fechado, para então processar o algoritmo de avaliação da saúde de dispositivo. Uma demanda do operador de manutenção, além de requisitar os mesmos serviços do usuário, também poderia requisitar o serviço de estatística da avaliação de saúde da válvula ou somente seus dados, e ainda, o serviço de colocar o dispositivo em modo de manutenção.

A figura 5.3 exibe o diagrama de classes integral inicialmente desenvolvido para este exemplo de aplicação. Para especificar um sistema embarcado em UML, mesmo um sistema relativamente simples, é necessário que cada subsistema (serviços) seja bastante organizado em termos de hierarquia de classes e estruturas UML. O projeto da

figura abaixo representa um modelo da estrutura física deste sistema com boa utilização das estruturas de uma linguagem de alto-nível. Como não existem requisitos não funcionais nesta especificação de sistema, os diagramas ACOD e JPDD não foram desenvolvidos.

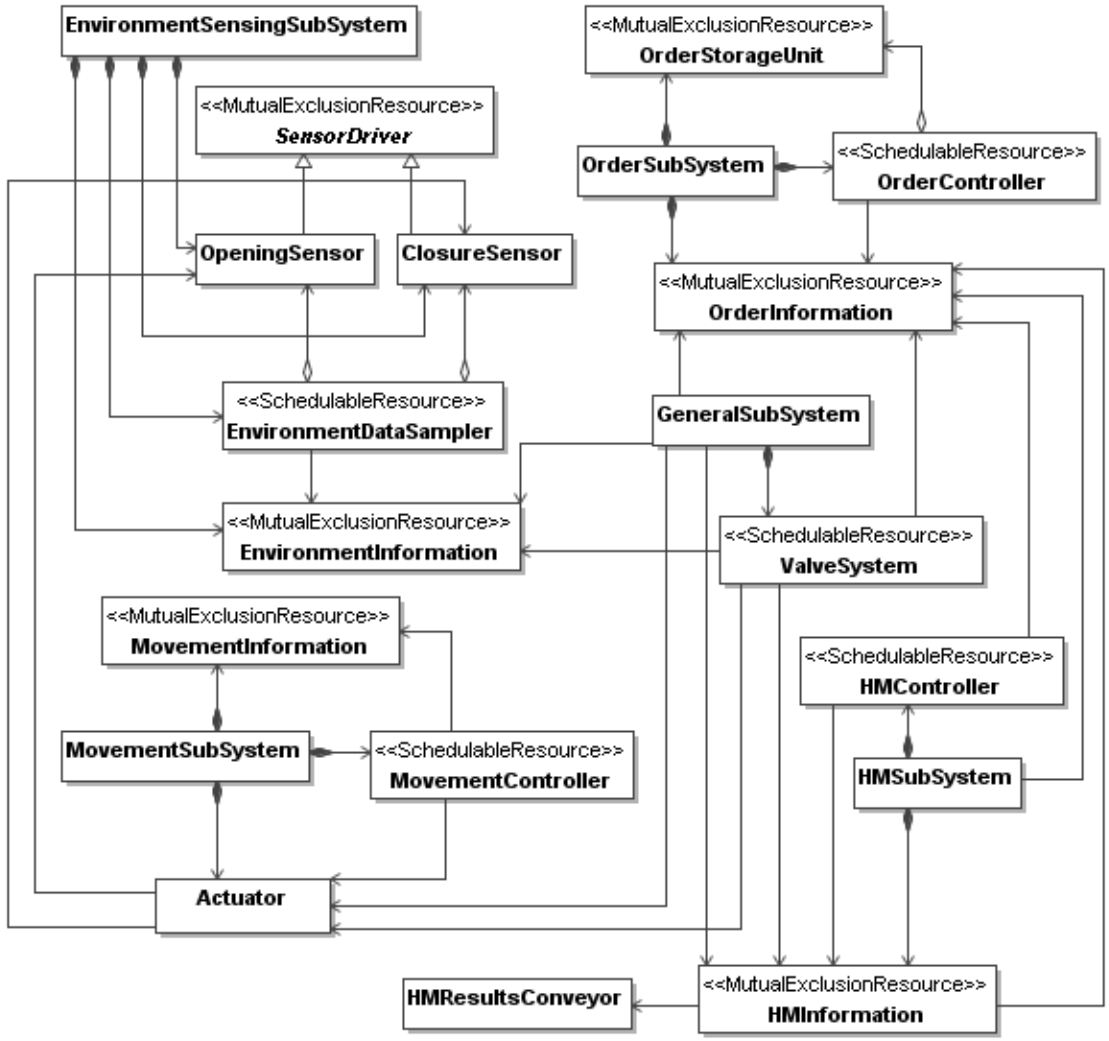

Figura 5.3: Diagrama de classes integral desenvolvido para essa aplicação.

O diagrama de classes integral do sistema, exibido na figura acima, é composto por quatro (4) subsistemas. Cada subsistema é um conjunto de entidades secundárias que trabalham interligadas para formarem um dos serviços oferecidos pelo sistema. O sistema principal pode ser dividido em: Subsistema de sensoriamento, Subsistema do atuador, Subsistema de pedidos e Subsistema de gerenciamento da saúde do componente.

5.2.1.1 Subsistema de sensoriamento

É composto por um conjunto de subclasses correlacionadas que trabalham de forma ordenada para caracterizar um serviço do sistema principal. A figura 5.4 expande parte do diagrama exibido na figura 5.3, enfatizando a estrutura de classes e as interligações que constituem o subsistema de sensoriamento.

Analisando a figura abaixo é possível identificar dois sensores (*OpeningSensor* e *ClosureSensor*), um para abertura e outro para fechamento da válvula, derivados da classe abstrata *SensorDriver*, de qual herdam todas suas funcionalidades. Estes dois sensores agregam outra classe chamada *EnvironmentDataSampler*, cuja função é avaliar os sensores periodicamente, registrando seus valores para uso posterior pelo sistema principal. O sistema principal acessa essas informações através da classe *EnvironmentInformation*, que fornece funções públicas a serem utilizadas para resgatar os valores registrados pela classe *EnvironmentDataSampler*. Todas essas classes secundárias compõem o subsistema *EnvironmentSensingSubSystem*, que seria uma classe responsável por unir e formalizar o serviço de sensoriamento.

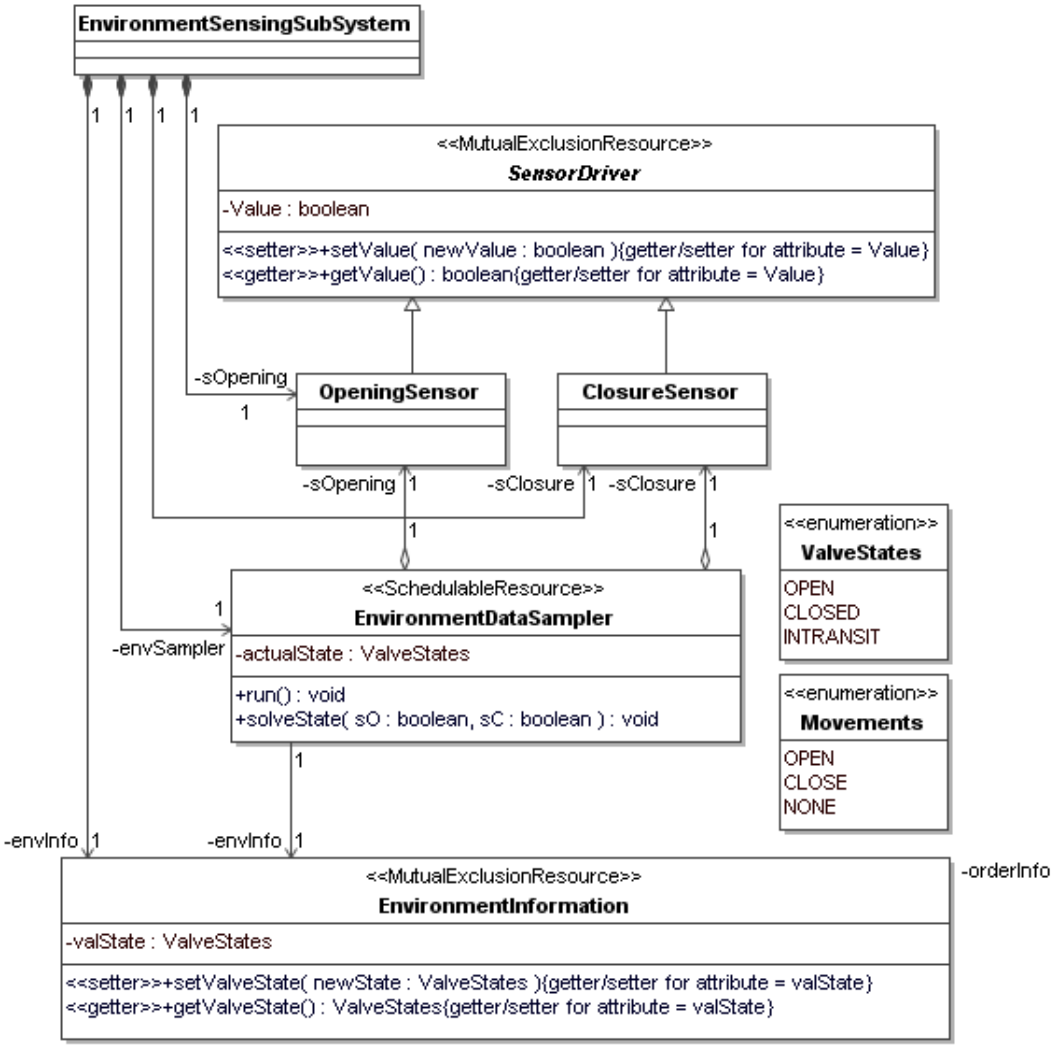

Figura 5.4: Conjunto de classes que formam o Subsistema de sensoriamento.

5.2.1.2 Subsistema do atuador

É composto pelo conjunto de subclasses que controlam os movimentos da válvula. Disponibiliza funções que permitem controlar e agendar tarefas no atuador. A figura 5.5 exibe o bloco de classes do subsistema do atuador.

Quando o sistema principal recebe e processa uma nova solicitação de pedido (ordem), ele cadastra o pedido recebido diretamente na classe *Actuator*, que executa esse movimento controlado pela classe *MovementController*, cujas ações são registradas pela classe *MovementInformation* para uso posterior do sistema principal. Todas essas classes compõem a classe *MovementSubSystem*, que formaliza a união destas subclasses formando o subsistema responsável pelos movimentos do atuador.

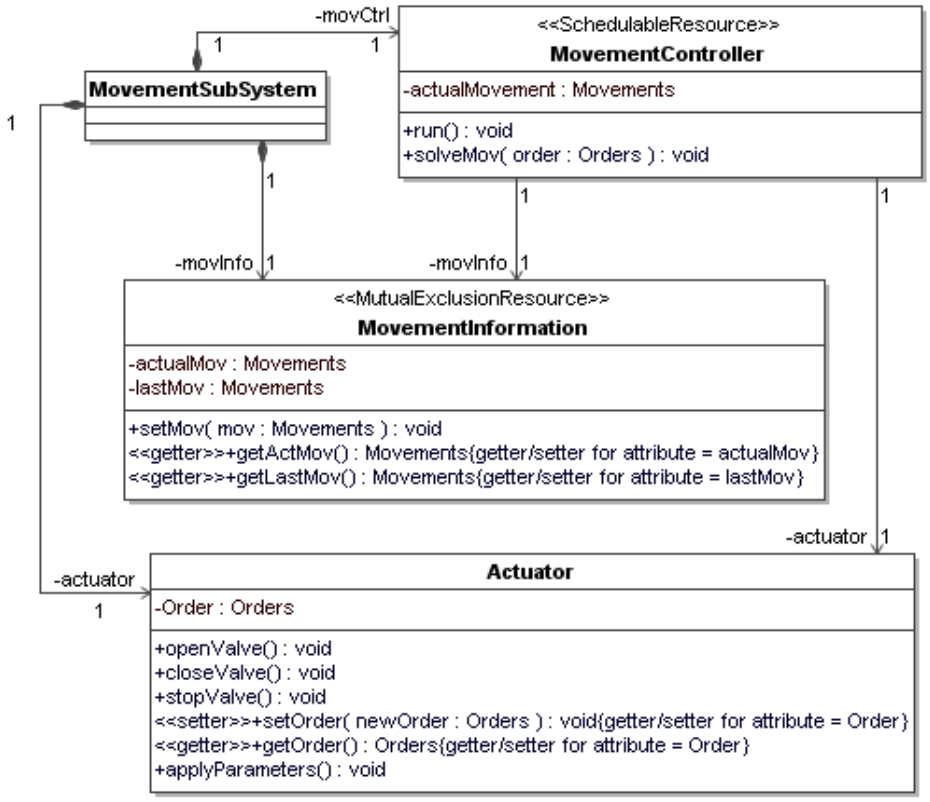

Figura 5.5: Conjunto de classes do Subsistema do atuador.

5.2.1.3 Subsistema de pedidos

É composto pelas subclasses que coordenam a recepção e o processamento de pedidos. Dispositivos externos são responsáveis por adicionar novos pedidos, enquanto que, este subsistema armazena e disponibiliza funções para que esses pedidos possam ser consumidos pelo sistema principal. A figura 5.6 exibe a estrutura deste subsistema.

A classe *OrderStorageUnit* armazena e disponibiliza funções para que outras estruturas possam adicionar e obter os pedidos armazenados. A classe *OrderController* é a responsável por consumir esses pedidos. Periodicamente o componente desta classe executa o procedimento de obter e processar um novo pedido. Ao processar cada pedido, este componente ainda registra internamente o número de vezes que cada tipo de serviço foi solicitado (neste caso, os serviços de abrir e fechar a válvula). Informações sobre pedidos e as estatísticas do serviço ficam registrados na classe *OrderInformation*, que tem como função justamente armazenar estes dados e disponibilizar funcionalidades para que o sistema principal consiga fazer uso destas informações. Neste subsistema, como nos demais, as classes secundárias compõem uma classe especial que formaliza o subconjunto: *OrderSubSystem*.

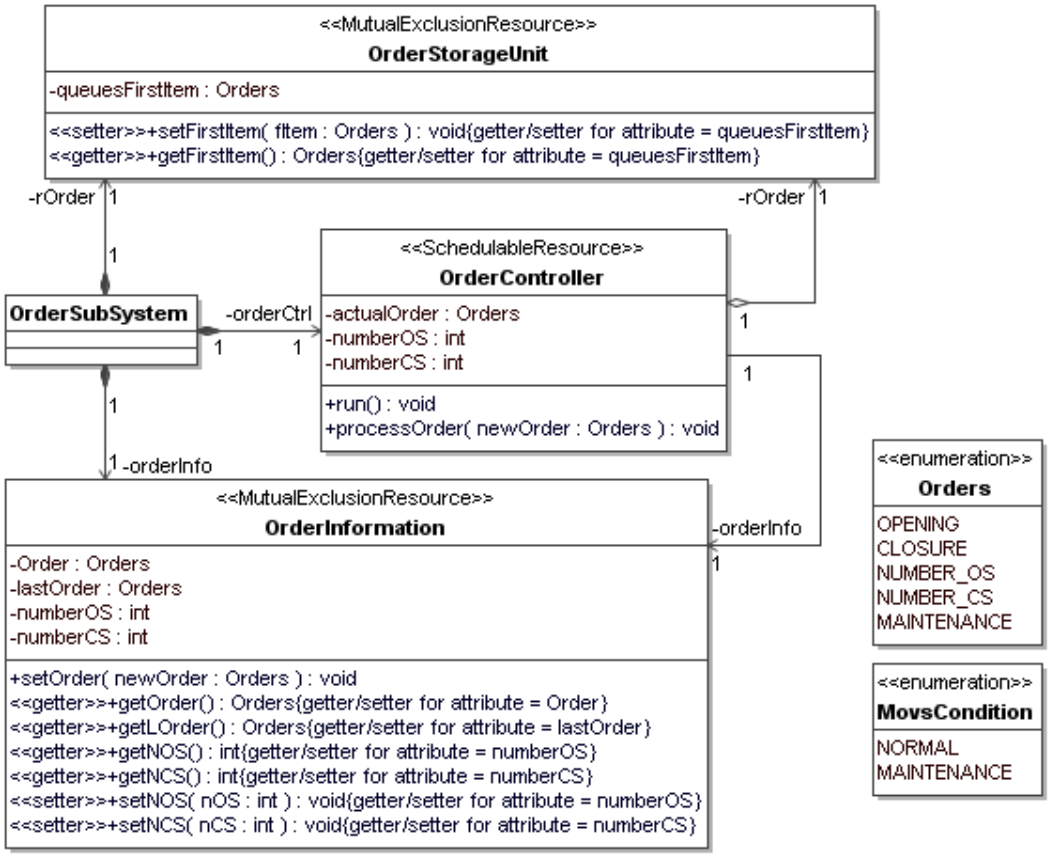

Figura 5.6: Conjunto de classes do Subsistema de pedidos.

5.2.1.4 Subsistema de gerenciamento da saúde do componente

É composto pelas classes responsáveis por aferir a saúde física do dispositivo. Este subsistema contém variáveis que auxiliam na avaliação do componente, através do processamento dos algoritmos de estimação de saúde, além de funções que permitem obter dados de manutenção. A figura 5.7 exibe a estrutura de classes deste subsistema, desenvolvido para realizar manutenção preventiva no sistema da válvula inteligente.

A classe *HMController* é responsável por avaliar periodicamente a saúde do dispositivo, desencadeando o procedimento que adquire e processa novas informações de utilização do componente. A classe *HMInformation* contém todas as informações referentes à manutenção e ao estado do dispositivo. Além disso, disponibiliza funcionalidades para que estas informações sejam adquiridas, processadas e avaliadas a partir de algoritmos capazes de aferir a saúde física do componente através dos seus dados de utilização. A classe *HMResultsConveyor* suporta a classe *HMInformation* com funcionalidades essenciais para validação do componente. E a classe *HMSubSystem* é uma composição das demais, formando o subsistema de gerenciamento da saúde do componente.

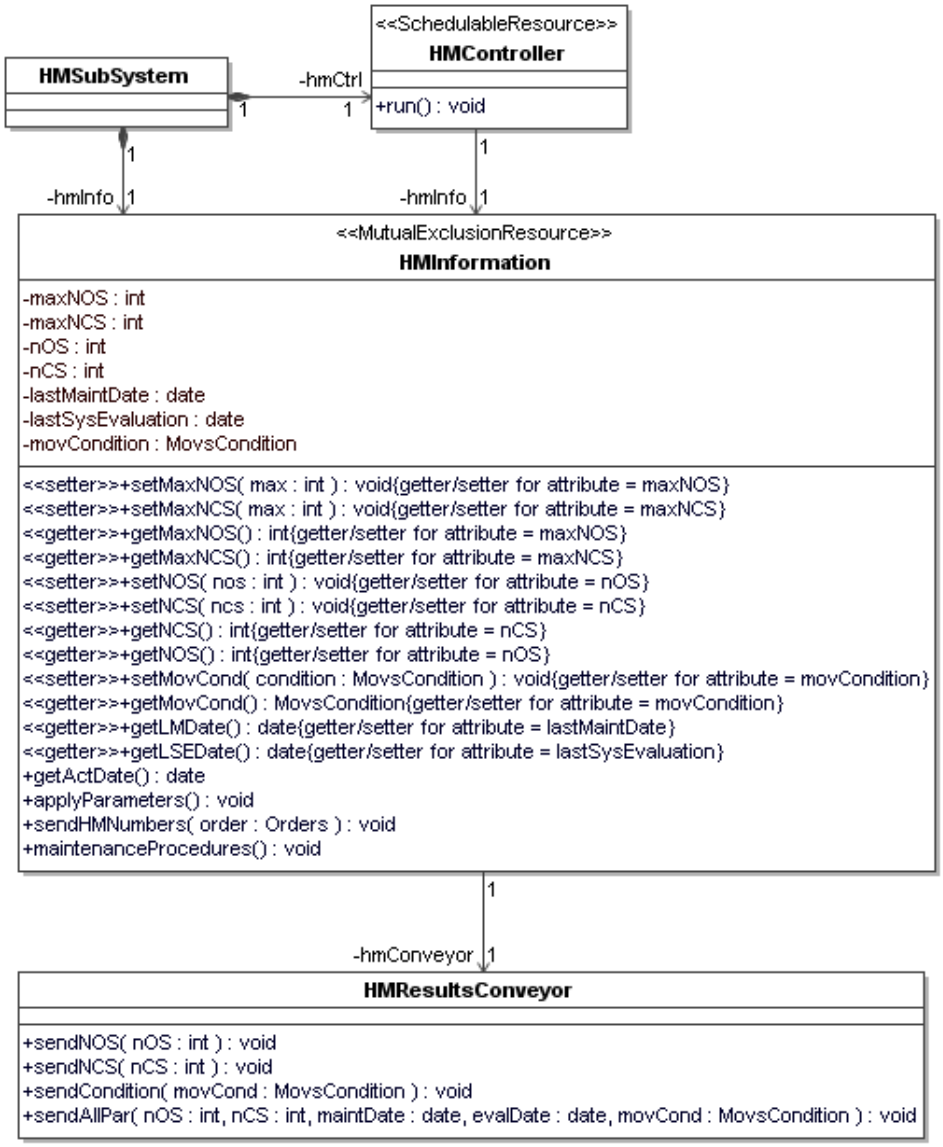

Figura 5.7: Conjunto de classes do Subsistema de gerenciamento da saúde do componente.

5.2.2 Diagramas de classes alternativos

Em função de não existirem regras de mapeamento para algumas estruturas importantes (classes abstratas, herança, polimorfismo, relações de composição e agregação, enumeração, entre outras), a estrutura do modelo precisou ser alterada, para ampliar a compatibilidade com as regras de mapeamento existentes, adequando-o à correta execução dos passos 3 e 4 da metodologia. Deste modo, a título de teste e de estudo de caso, o sistema da válvula inteligente foi remodelado de outras duas formas mais simplificadas, contendo somente elementos capazes de serem mapeados.

Primeiramente, o sistema da válvula inteligente foi remodelado em um conjunto de classes simples com interações representadas na forma de associações, conforme a figura 5.8. Este modelo, diferentemente do modelo integral da figura 5.3, foi projetado atendendo todas as especificações (regras) de modelagem definidas pela proposta deste

trabalho. Tais especificações de modelagem definem como deve ser desenvolvido o modelo, para que seja melhor interpretado pela ferramenta GenERTiCA e melhor transformado em código através das regras de mapeamento.

O modelo da figura 5.8 manteve a mesma lógica estrutural do modelo original. Contudo, estruturas complexas, como classes abstratas, herança, composição e agregação, foram retiradas do modelo e substituídas por estruturas simples para que a ferramenta de transformação pudesse interpretar as estruturas deste modelo estático com totalidade. Adicionalmente, atributos e métodos de classes também foram reformulados para atenderem as regras de modelagem da Tabela 4.2.

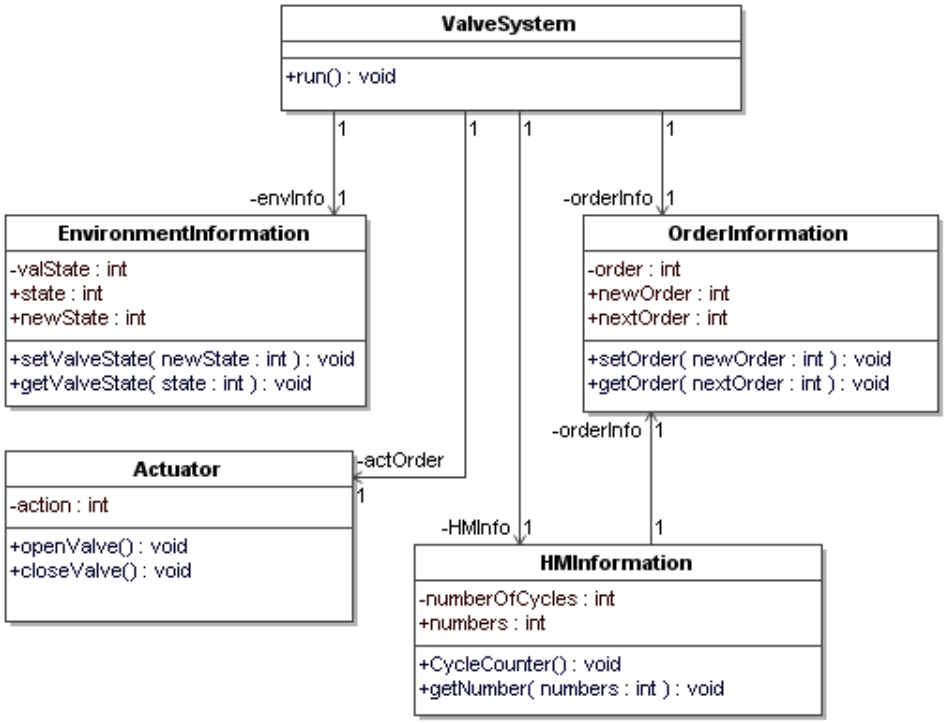

Figura 5.8: Sistema remodelado para um conjunto de classes simples com associações.

Cada uma destas classes secundárias fornece um serviço que é acessado/controlado pela classe principal. Quando instanciado, o objeto da classe *ValveSystem* executa periodicamente seu método *run()*, através de um recurso externo de *scheduler*, realizando o processamento completo do sistema. O procedimento deste método realiza uma busca por novos pedidos no componente *OrderInformation*, contabiliza uma nova ação em *HMInformation*, verifica o estado da válvula em *EnvironmentInformation* e executa a nova ação através do componente *Actuator*. Todas as interações entre a classe principal e as demais são realizadas através da relação de associação direta existente entre elas.

Para obter resultados ainda melhores através do processo de geração de código, o sistema da válvula inteligente também foi remodelado como uma classe unitária chamada *SmartComponent* (Componente Inteligente), cujo modelo é um compacto de todas as funcionalidades do sistema, com o mesmo comportamento esperado para cada serviço. Este sistema é demonstrado através da classe da figura 5.9.

Apesar da simplicidade, essa representação do sistema engloba praticamente todas as estruturas da linguagem UML que podem ser modeladas através das regras de mapeamento desenvolvidas para este trabalho. Apenas a estrutura representativa da relação de associação não está inclusa neste "modelo". Além disso, quando implementado desta forma, a lógica funcional também fica compactada, tornando a implementação dinâmica do sistema um conjunto de ações grande e mais complexo. Assim, gerar código para essa representação valida, além da estrutura estática, também a estrutura dinâmica do sistema. Considerando que um dos objetivos desta proposta de dissertação é gerar em hardware lógicas de sistemas embarcados que antigamente eram desenvolvidas somente em software, validar esse tipo de transformação torna-se bastante interessante.

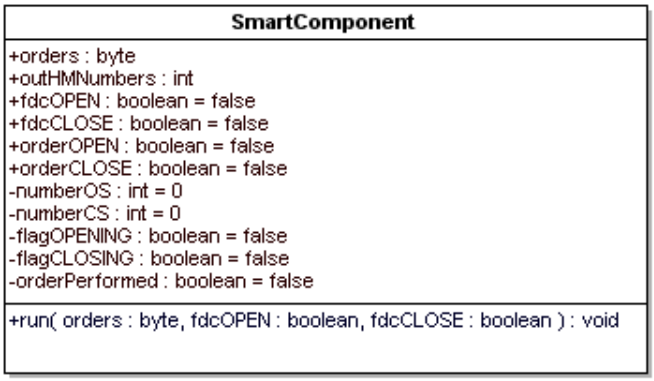

Figura 5.9: Representação compacta do sistema da válvula inteligente.

O método *run()* deste modelo implementa de forma análoga o comportamento integral esperado para o sistema da figura 5.3. Realiza a partir de um procedimento único rotinas para processar novos pedidos, contabilizar novas ações e verificar o estado do sistema, resultando em ações que são executadas pelo atuador da válvula. A representação do conjunto de ações deste método é demonstrada parcialmente na figura 5.11 do item 5.2.3, destinado aos diagramas de sequência do estudo de caso.

5.2.3 Diagramas de sequência

Um diagrama de sequência representa a troca de mensagens entre classes que compõem o sistema, com o objetivo de resultar em ações. Para cada serviço requisitado pelos atores externos (i.e. usuário, operador de manutenção) são necessários um (1) ou mais diagramas de sequência. Cada serviço fornecido pelo sistema é composto por um conjunto de ações que são modeladas dentro de diagramas de sequência. Cada ação individual ainda pode conter também o seu próprio conjunto de ações, que obviamente precisa ser modelado como outro diagrama de sequência, e assim por diante, até que o conjunto de ações tenha somente ações simples (e.g. atribuição).

A figura 5.10 exibe o diagrama de sequência principal do modelo que representa o sistema da válvula inteligente da figura 5.8. Este é o diagrama de sequência do método *run()*, cuja funcionalidade é executar periodicamente processando um ciclo de ações que determina o funcionamento de todo o sistema. Verifica-se a partir da figura que este método é processado a cada 5ms através de um timer externo, cuja *tag* de identificação, *TimedEvent*, remete a um evento de timer periódico do *profile* de tempo real utilizado

(OMG MARTE, 2010). Quando em execução este método desencadeia o fluxo de ações que caracteriza o funcionamento do sistema.

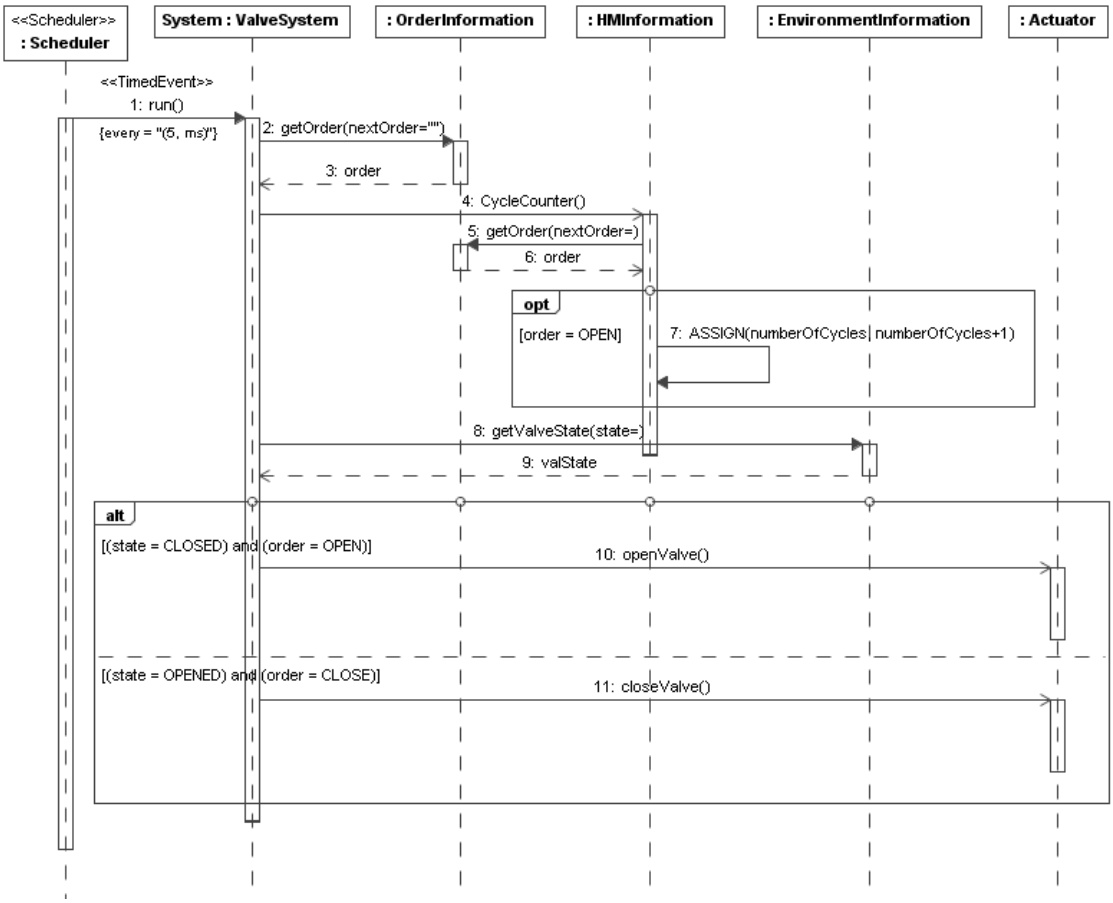

Figura 5.10: Diagrama de sequência do método *run()* do modelo da figura 5.8.

O diagrama de sequência parcial do modelo compacto (figura 5.9) é demonstrado através da figura 5.11. Sua lógica de funcionamento é similar àquela apresentada no diagrama da figura 5.10. Contudo, as ações do diagrama da figura 5.10 estão representadas em mais alto nível, através da chamada de métodos e da avaliação de seus retornos. A lógica interna daqueles métodos está oculta no diagrama. Somente as representações da chamada e do retorno dos métodos são exibidas. Isso divide a ação em ações menores, diminuindo deste modo a complexidade e tornando a visualização da rotina mais simples e intuitiva.

No diagrama abaixo, da figura 5.11, acontece o contrário, além do método principal - *run()* - nenhum outro método foi desenvolvido. Assim, todas as ações que deveriam estar ocultas sob métodos, foram desenvolvidas em mais baixo nível, diretamente dentro do método principal. Isso não é uma boa prática de modelagem e nem de desenvolvimento, pois torna a lógica funcional complexa, prejudicando, além do entendimento do modelo, também a geração de código. Contudo, a intenção desta representação é validar o código VHDL gerado a partir deste diagrama UML complexo. Então, o diagrama abaixo promove um conjunto de ações de atribuição e alternativas que representa a funcionalidade do sistema modelado para válvula inteligente de forma compacta.

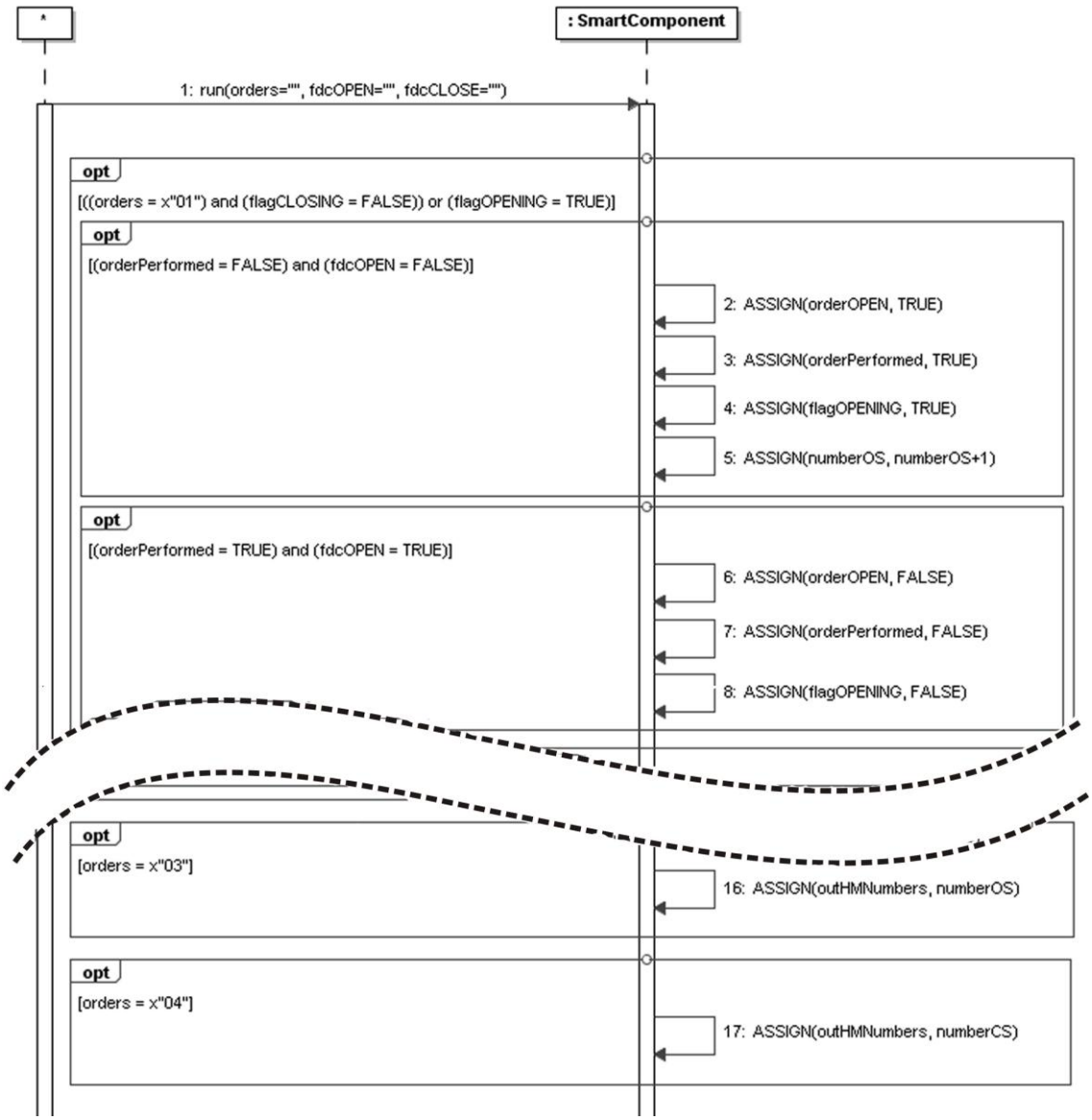

Figura 5.11: Diagrama de sequência parcial do modelo compacto da figura 5.9.

5.3 Etapa 3 – Transformação UML-DERCS

A terceira etapa do processo segue o considerado em sua seção, transformando o modelo RT-UML em modelo DERCS. Neste ponto o diagrama de classes e os diagramas de sequência desenvolvidos nas etapas anteriores são convertidos, através de uma conversão entre modelos, para um modelo tipo DERCS, livre de ambiguidades e específico para o domínio de aplicação. Modelos UML normalmente possuem grande abstração de dados e não remetem à plataforma onde seriam utilizados. Então, essa transformação serve como uma etapa intermediária e ocorre para aproximar aquele modelo UML do domínio de aplicação para qual o código será gerado. Assim, essa fase do processo se caracteriza pela transformação de um modelo RT-UML, independente de plataforma (PIM), para um modelo DERCS, específico do domínio de sistemas embarcados de tempo-real distribuídos (PSM).

Essa etapa é totalmente transparente ao usuário/ou projetista, pois o processamento é realizado de forma oculta, resultando diretamente no código da plataforma alvo. Como foi mencionado no capítulo anterior, a ferramenta GenERTiCA, que suporta este trabalho, é um *plugin* da ferramenta comercial MagicDraw, sendo necessariamente utilizada como uma funcionalidade a mais desta última ferramenta. Desta forma, com o *plugin* adicionado e com o modelo pronto, o usuário precisa somente executar essa funcionalidade, e esperar pela conclusão do processo. A figura 5.12 demonstra como acessar o *plugin* da ferramenta GenERTiCA através do software MagicDraw de versão 16.0.

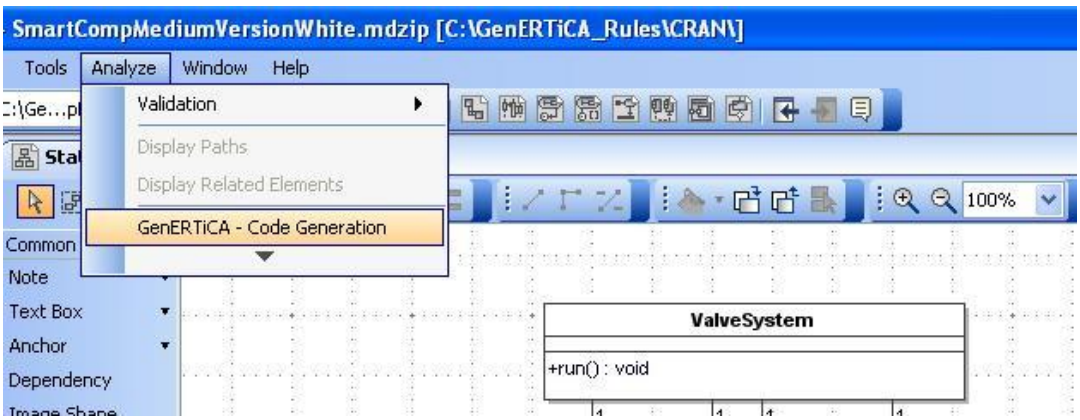

Figura 5.12: Acesso ao *plugin* da ferramenta GenERTiCA no MagicDraw.

Ao iniciar, GenERTiCA solicita o caminho onde está armazenado o arquivo com as regras de mapeamento que serão utilizadas. É diretamente a escolha deste arquivo que determina para qual linguagem alvo o código deve ser gerado. A figura 5.13 exemplifica este passo. Na sequência, deve ser selecionada a pasta onde será salvo o código resultante. Então, após este passo, o processo tem início, resultando em código ou em mensagens de erro, que auxiliam o usuário na identificação do problema.

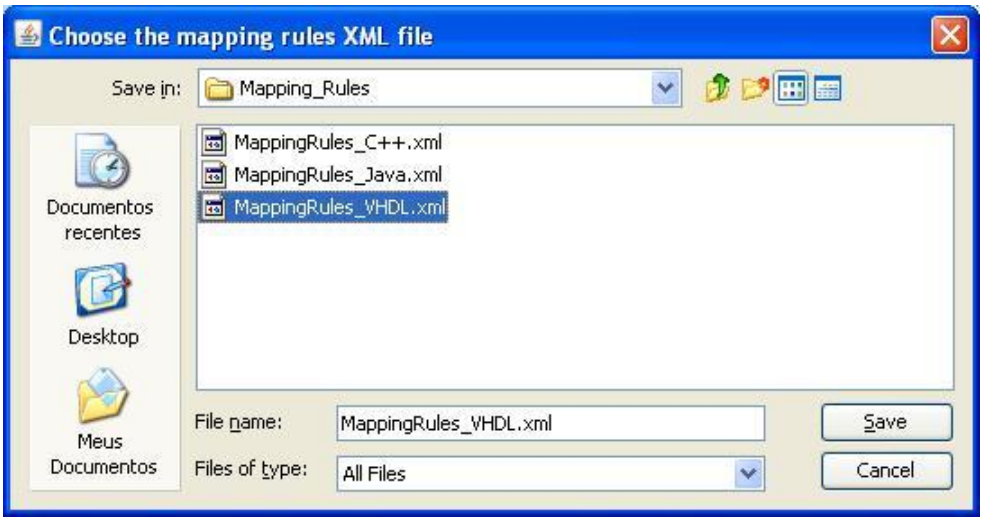

Figura 5.13: Escolha do arquivo de regras de mapeamento na execução de GenERTiCA.

5.4 Etapa 4 – Geração do código

Depois de inicializado o processo de geração de código, a ferramenta GenERTiCA transforma internamente o modelo UML em modelo DERCS e, automaticamente, dá início a transformação modelo-texto que gera o código. Com o modelo DERCS pronto, GenERTiCA assume o arquivo de regras de mapeamento e processa os *scripts* responsáveis por gerar os elementos de código, indo buscar no modelo DERCS os elementos e informações necessários para formação do código na linguagem alvo.

No caso da linguagem VHDL, o resultante deste processo é um conjunto de arquivos de extensão **vhd**, característicos da linguagem alvo, e um arquivo de *log*, gerado pela ferramenta GenERTiCA durante o processo, em qual ficam registradas todas as ações realizadas, desde o início até a conclusão do processo. A figura 5.14 exibe os arquivos resultantes da geração de código para o modelo da figura 5.8. É gerado um arquivo VHDL para cada classe do sistema, inclusive para o recurso de *scheduler*. O conteúdo dos arquivos de cada classe foi preenchido com dados oriundos do modelo. Já o recurso de *scheduler*, para ter seu código gerado corretamente, precisaria ter sido implementado também como uma classe do sistema, ou ainda, poderia ser um recurso de código inserido diretamente pela ferramenta GenERTiCA, na etapa de entrelaçamento de aspectos, caso o aspecto *Timing Package* tivesse sido desenvolvido e incluído no modelo.

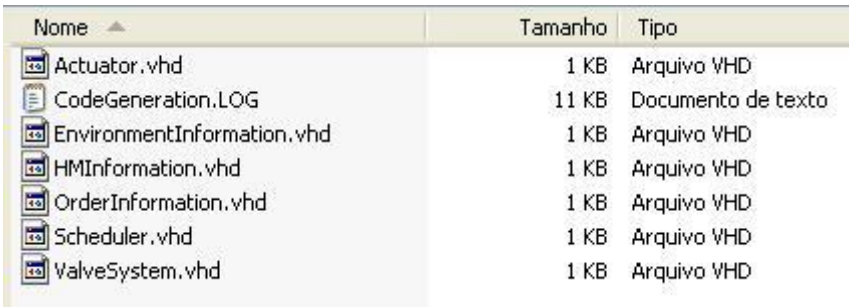

Figura 5.14: Resultante do processo de geração de código sobre o modelo da figura 5.8.

O código da figura 5.15 é o resultado do processo de geração de código realizado sobre o modelo da figura 5.9, com a utilização do conjunto de regras de mapeamento para linguagem VHDL, desenvolvidos para extensão da ferramenta GenERTiCA como proposta desta dissertação. Sendo o modelo da figura 5.9, a representação do sistema em uma única classe UML, com atributos e operações que realizam a funcionalidade do sistema, o código gerado é apenas um arquivo da linguagem VHDL, contendo o conjunto *Entity* e *Architecture*, que declara o componente e descreve sua arquitetura, e um *Process* da linguagem alvo, que representa o método *run()* da figura 5.11, com o código VHDL que executa todas suas ações.

```
-- Generated Code
01 library IEEE;
02 use IEEE.STD_LOGIC_1164.ALL;
03 use IEEE.NUMERIC_STD.ALL;

04 entity SmartComponent is
05 Port (
06   clock : in STD_LOGIC;
07   reset : in STD_LOGIC;
08   orders : inout STD_LOGIC_VECTOR(7 downto 0);
09   outHMNumbers : inout INTEGER;
10   fdcOPEN : inout BOOLEAN;
11   fdcCLOSE : inout BOOLEAN;
12   orderOPEN : inout BOOLEAN;
13   orderCLOSE : inout BOOLEAN
14   );
15 end SmartComponent;

16 architecture Behavioral of SmartComponent is
17 signal numberOS : INTEGER;
18 signal numberCS : INTEGER;
19 signal flagOPENING : BOOLEAN;
20 signal flagCLOSING : BOOLEAN;
21 signal orderPerformed : BOOLEAN;

22 begin

23 run: process(clock, reset, orders, fdcOPEN, fdcCLOSE)
24 begin
25   if (reset='1') then
26     -- variables initialization
27     numberOS <= 0;
28     numberCS <= 0;
29     flagOPENING <= FALSE;
30     flagCLOSING <= FALSE;
31     orderPerformed <= FALSE;
32   elsif (clock'EVENT and clock='1') then
33     if ((orders = x"01") and (flagCLOSING = FALSE)) or (flagOPENING = TRUE) then
34       if (orderPerformed = FALSE) and (fdcOPEN = FALSE) then
35         orderOPEN <= TRUE;
36         orderPerformed <= TRUE;
37         flagOPENING <= TRUE;
38         numberOS <= numberOS+1;
39       end if;
40       if (orderPerformed = TRUE) and (fdcOPEN = TRUE) then
41         orderOPEN <= FALSE;
42         orderPerformed <= FALSE;
43         flagOPENING <= FALSE;
44       end if;
45     end if;
46     if ((orders = x"02") and (flagOPENING = FALSE)) or (flagCLOSING = TRUE) then
47       if (orderPerformed = FALSE) and (fdcCLOSE = FALSE) then
48         orderCLOSE <= TRUE;
49         orderPerformed <= TRUE;
50         flagCLOSING <= TRUE;
51         numberCS <= numberCS+1;
52       end if;
53       if (orderPerformed = TRUE) and (fdcCLOSE = TRUE) then
54         orderCLOSE <= FALSE;
55         orderPerformed <= FALSE;
56         flagCLOSING <= FALSE;
57       end if;
58     end if;
59     if orders = x"03" then
60       outHMNumbers <= numberOS;
61     end if;
62     if orders = x"04" then
63       outHMNumbers <= numberCS;
64     end if;
65   end if;
66 end process;
67 end Behavioral;
```

Figura 5.15: Código gerado para o estudo de caso do componente inteligente.

Para validação da quantidade de código gerado, foi realizada uma comparação entre o número de linhas de código gerado para cada um dos modelos propostos para este estudo de caso. A Tabela 5.1 apresenta a comparação entre a quantidade de código gerado para cada modelo. Designado como Completo está o modelo integral da figura 5.3, que representa a modelagem UML ideal para este tipo de sistema. Apresentados como Médio e Compacto estão os modelos das figuras 5.8 e 5.9, que representam a modelagem do mesmo sistema da válvula inteligente, porém de forma reduzida e para fins de geração de código de acordo com as regras de modelagem definidas neste trabalho.

A primeira coluna da tabela contém os modelos que foram comparados. A segunda coluna representa o número de linhas de código VHDL automaticamente geradas para cada modelo. A terceira coluna representa linhas de código que foram geradas de forma equivocada. Pode-se observar que somente foram geradas linhas de código errôneas para o modelo Completo, aquele que foi desenvolvido integralmente sob regras normais de modelagem. Neste caso, a ferramenta suporte interpretou de forma equivocada algum elemento do modelo, inserindo trechos de código em excesso, ou deixando de inserir alguns itens. Isto é possível de acontecer, visto que o modelador não seguiu as regras de modelagem propostas, que pretendem evitar justamente esse tipo de problema. A quarta coluna representa o número de linhas que faltaram ser adicionados ao modelo. Para o primeiro modelo, o Completo, é compreensível que existam linhas faltando já que foram implementadas estruturas que não estão mapeadas. Para o modelo Médio, as linhas faltantes se devem principalmente à inserção das estruturas de associação. O mapeamento destas estruturas não ficou completo e, assim, seu código é gerado com deficiência. A coluna de Percentagem mede a quantidade de código válido em relação ao total de linhas geradas.

Apesar da simplicidade do modelo Compacto, os resultados obtidos são encorajadores. Para este componente, sessenta e sete (67) linhas de código VHDL foram automaticamente geradas, cobrindo completamente (100%) o código necessário para aplicação.

Tabela 5.1: Quantidade de código gerado para cada modelo.

Modelo	**Nº de linhas geradas**	**Linhas erradas**	**Linhas faltando**	**Percentagem**
Completo	1211	118	807	57 %
Médio	242	0	26	90 %
Compacto	67	0	0	100 %

Para verificar a qualidade e a viabilidade real do resultado da geração de código para VHDL, o código da figura 5.15 foi testado sobre o ambiente de desenvolvimento, simulação e síntese para FPGAs, ISE 13.4 da Xilinx, com configuração para executar sobre um FPGA Xilinx Spartan3A (Device: xc3s700a, Package: fg484 e Speed: -4). O código foi compilado e sintetizado perfeitamente, conforme resultados das Tabelas 5.2 e 5.3. Para um teste definitivo, o *bitstream* gerado poderia ser carregado diretamente em um kit de desenvolvimento com o FPGA em questão. Contudo, esta etapa não foi incluída na dissertação, restando como trabalho de validação futura.

Tabela 5.2: Consumo de área do projeto da figura 5.15.

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	101	11,776	1%	
Number of 4 input LUTs	88	11,776	1%	
Number of occupied Slices	78	5,888	1%	
Number of Slices containing only related logic	78	78	100%	
Number of Slices containing unrelated logic	0	78	0%	
Total Number of 4 input LUTs	150	11,776	1%	
Number used as logic	88			
Number used as a route-thru	62			
Number of bonded IOBs	46	372	12%	
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	3.06			

Tabela 5.3: Resultado da análise e validação de *Timing*.

Timing Summary:

Speed Grade: -4

Minimum period: 5.442ns (Maximum Frequency: 183.756MHz)
Minimum input arrival time before clock: 7.284ns
Maximum output required time after clock: 5.558ns
Maximum combinational path delay: No path found

As Tabelas 5.2 e 5.3 exibem os dados resultantes das etapas de síntese e implementação do código VHDL do projeto da figura 5.15, coletados a partir do sumário de síntese retornado pela plataforma ISE 13.4 da Xilinx. A Tabela 5.2 detalha o consumo de área de projeto e a Tabela 5.3 mostra o resultado da análise de *Timing*, que caracteriza o desempenho do circuito gerado no FPGA.

A baixa taxa de ocupação de área do FPGA (1%) se deve a simplicidade do projeto, e o desempenho do sistema poderia ser estimado a partir da frequência máxima de execução deste circuito sob a plataforma em questão.

6 CONCLUSÕES E TRABALHOS FUTUROS

Na busca por processos de desenvolvimento mais acelerados, a linguagem UML se destacou por permitir a especificação de sistemas de forma gráfica, facilitando o projeto de sistemas complexos por tornar o processo mais intuitivo e por elevar o nível de abstração de sistemas, permitindo a abstração de aspectos irrelevantes ao projeto naquele momento (i.e. são modelados componentes, sem definir se serão desenvolvidos em hardware ou software). Pela praticidade e potencialidade, a linguagem UML se tornou um padrão para modelagem de sistemas em alto nível de abstração, e logo surgiram processos que permitem gerar a partir de modelos UML, código para linguagens de descrição de software embarcado (C, Java) e para linguagens tradicionais de descrição de hardware (HDL).

Nossa proposta é focada na geração de descrições de hardware para linguagem VHDL a partir de modelos UML. Descrições de hardware em VHDL estão sendo bastante utilizadas no desenvolvimento de sistemas embarcados porque permitem que o comportamento descrito para o sistema (modelado) seja verificado e validado antes de ser efetivamente construído, resultando em um processo de produção de hardware mais rápido e menos propenso a erros de projeto. No entanto, apesar de existirem diversos trabalhos e ferramentas comerciais direcionadas à geração de código convencional a partir de modelos UML, transformações de UML para VHDL ainda não é um assunto bem difundido. Existem poucas pesquisas sobre o tema e, até o momento da elaboração deste trabalho, nenhuma ferramenta comercial dedicada especificamente para conversão de especificações UML em VHDL.

Em face deste desafio, este trabalho propõe uma abordagem metodológica para geração automática de código fonte VHDL a partir de modelos UML de sistemas embarcados para ser utilizado em sistemas com FPGAs. É proposto um processo de engenharia que cobre desde as fases de análise de requisitos e modelagem do sistema em UML, até a geração do código fonte na linguagem VHDL, onde o foco é gerar na forma de descrições de hardware, todas aquelas funções lógicas de um sistema embarcado que normalmente são desenvolvidas em software.

Para atingir este objetivo é proposto neste trabalho um conjunto de regras de mapeamento que estende a funcionalidade da ferramenta GenERTiCA, permitindo a geração de código VHDL a partir de modelos UML para sistemas embarcados. Foram pesquisados e desenvolvidos conceitos para representar o mapeamento entre a linguagem gráfica de especificação de sistemas UML e linguagem estruturada para descrição hardware VHDL. Adicionalmente, *scripts* capazes de realizar o mapeamento entre as estruturas e os elementos das linguagens também foram desenvolvidos na forma de regras de mapeamentos. Regras estas, que ampliam a funcionalidade da

ferramenta GenERTiCA, permitindo-lhe gerar código VHDL, além de código fonte de software.

A metodologia proposta foi demonstrada a partir do estudo de caso de um sistema que implementa um Componente (Dispositivo) Inteligente. Este estudo de caso foi desenvolvido a partir de um sistema real existente no Centro de Pesquisas em Automação de Nancy - França (CRAN). Foi motivado pela intenção de adicionar funcionalidades e inteligência em um componente de campo industrial - uma válvula de controle de fluidos - a fim de torná-la inteligente para realizar atividade de prognóstico e manuteção preventiva através do suporte de um FPGA.

Regras de mapeamento foram desenvolvidas e verificadas permitindo a geração de código fonte VHDL para os seguintes elementos e estruturas da linguagem UML: classes, atributos, métodos, elementos de comportamento, troca de mensagens e associações entre classes. Além disso, apesar de não haverem sido implementadas, também foram desenvolvidos e descritos os conceitos para o desenvolvimento de regras de mapeamento para estruturas de herança e de polimorfismo. Embora a simplicidade do estudo de caso validado, os resultados obtidos foram bastante encorajadores. Para sistemas simples, a eficiência das regras permitiu a geração de 100% do código necessário ao sistema. Resultados que são considerados satisfatórios e uma metodologia que demonstra potencial.

Dentro de nossas expectativas, os resultados obtidos também foram bem aceitos na comunidade internacional. Artigos sobre o tema, com os resultados apresentados aqui, foram submetidos e aprovados nas seguintes conferências: *8nd IEEE International Conference on Industrial Informatics (INDIN 2010)* (MOREIRA, T.G. et. al., 2010a) e *7th IFIP Conference on Distributed and Parallel Embedded Systems (DIPES 2010)* (MOREIRA, T.G. et. al., 2010b).

Para finalizar as conclusões deste trabalho são feitas algumas considerações a respeito de melhorias que podem ser adotadas como próximos passos para continuação do trabalho desta dissertação:

- Desenvolver regras de mapeamento para os conceitos de herança e de polimorfismo é uma tarefa bastante importante, pois permitirá a modelagem de sistemas complexos;
- Formular os conceitos e desenvolver regras para o mapeamento de requisitos não funcionais através do uso de aspectos;
- Realizar outros estudos de caso para testar e validar as regras;
- Executar testes em FPGAs para validar o código e viabilizar a possibilidade de otimizações no código resultante;
- Aplicar metodologia proposta em sistemas reais complexos, tais como sistemas industriais de prognóstico e manutenção preventiva.

REFERÊNCIAS

AKEHURST, D.; HOWELLS, G.; MCDOANLD-MAIER, K.; BORDBAR, B. Compiling UML State Diagrams into VHDL: An Experiment in Using Model Driven Development. **Forum on Specification and Design Languages**, Barcelona, 2007. p. 219-224.

ANDERSSON, P.; HÖST, M. UML and System C – A comparison and mapping rules for automatic code generation. **International Forum on Specification and Design Languages**, Barcelona: Springer Netherlands, 2007. p.199-299.

APACHE. **Apache Velocity Project**. 2011. Disponível em: <http://velocity.apache.org/>. Acesso em: Nov. 2011.

ASHENDEN, P. J. **The Designer's Guide to VHDL**. San Francisco: Morgan Kaufmann, 1996.

AXELSSON, J. Real-world modeling in UML. **13th International Conference on Software and Systems Engineering and their Applications (ICSSEA),** Paris, 2000.

BALARIN, F.; WATANABE, Y.; HSIEH, H.; LAVAGNO, L.; PASSERONE, C.; SANGIOVANNI-VINCENTELLI, A. Metropolis: An Integrated Electronic System Design Environment. **IEEE Computer**, Los Alamitos, v.36, n.4, p.45–52, 2003.

BARR, M. **Real men program in C**. Embedded Systems Design. TechInsights (United Business Media). (1 August 2009) p. 2. Disponível em: <http://eetimes.com/electronics-blogs/industrial-control-designline-blog/4027479/Real-men-program-in-C>. Acesso em: Nov. 2011.

BERTAGNOLLI, S. C. **FRIDA: um método para elicitação e modelagem de rnfs.** 2004. Doutorado em Ciências da Computação - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

BJÖRKLUND, D.; LILIUS, J. From UML behavioral descriptions to efficient synthesizable VHDL. **20th IEEE Norchip Conference,** Copenhagen, 2002.

CARRO, L.; WAGNER, F. R. Sistemas Computacionais Embarcados. **Jornadas de atualização em informática**. Campinas: SBC, 2003. n.22, p.45-94.

CHARFI, A.; MRAIDHA, C.; GERARD, S.; TERRIER, F.; BOULET, P. Toward optimized code generation through model-based optimization. **Design, Automation & Test in Europe Conference & Exhibition (DATE),** Dresden, 2010. p.1313-1316.

CHEN, R.; SGORI, M.; LAVAGNO, L.; MARTIN, G.; SANGIOVANNI-VICENTELLI, A. UML and Platform-based Design. In: LAVAGNO, L.; MARTIN, G.; SELIC, B. (Ed.). **UML for Real: Design of Embedded Real-Time Systems.** Dordrecht: Kluwer Academic, 2003. p. 107-126.

CÔTÉ, C.; ZILIC, Z. Automated SystemC to VHDL translation in hardware/software codesign. **9th International Conference on Electronics, Circuits and Systems (ICECS),** Dubrovnik, 2002. p.717-720.

CRAN. **Centre de Recherche en Automatique de Nancy**. 2011. Disponível em: <http://www.cran.uhp-nancy.fr/>. Acesso em: Nov. 2011.

DAVEAU, J. M.; MARCHIORO, G. F.; VALDERRAMA, C. A.; JERRAYA, A. VHDL Generation from SDL specification hardware description languages and their applications. **Computer Hardware Description Languages and Their Applications (CHDL'97),** Toledo, 1997.

DAVIS II, J.; GOEL, M.; HYLANDS, C.; KIENHUIS, B.; LEE, E. A.; LIU, J.; MULIADI, L.; NEUENDORFFER, S.; REEKIE, J.; SMYTH, N.; TSAY, J.; XIONG, Y. **Overview of the Ptolemy project**, 1999.

DIJKSTRA, E. W. **A Discipline of Programming**. Englewood Cliffs, New Jersey: Prentice Hall, p.1976. 217.

DOUGLASS, B. P. **Real-Time UML: Developing Efficient Objects for Embedded Systems.** Boston: Addison-Wesley, 1999.

ECKER, W. An object-oriented view of structural VHDL description. **VHDL International Users Forum Spring '96 Conference,** Santa Clara, 1996.

FERNANDES, J. M.; ADAMSKI, M.; PROENÇA, A. J. VHDL generation from hierarchical Petri net specifications of parallel controllers. **IEE Proceedings: Computers and Digital Techniques**, [S.1], v.144, n.2, p.127-137, 1997.

FILIBA, T.; LEUNG, M-K.; NAGPAL, V. **VHDL Code Generation in the Ptolemy II Environment.** Technical report, Electrical Engineering and Computer Sciences, UC Berkeley , 2006.

FISCHER, S.; WYTREBOWICZ, J.; BUDKOWSKI, S. Hardware/Software Co-Design of Communication Protocols. **22nd EUROMICRO Conference**, Prague, 1996.

FLEISCHMANN, J.; BUCHENRIEDER, K.; KRESS, R. A Hardware/Software Prototyping Environment for Dynamically Reconfigurable Embedded Systems. **6th International Workshop on HW/SW co-design**, Washington, 1998. p.105-109.

FREITAS, E. P. de. **Metodologia Orientada a Aspectos para a Especificação de Sistemas Tempo-Real Embarcados e Distribuídos.** 2007. Mestrado em Ciências da Computação - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

FREITAS, E.P.; WEHRMEISTER, M.A.; SILVA JR, E.T.; CARVALHO, F.; PEREIRA, C.E.; WAGNER, F.R. DERAF - A High-level Aspects Framework for Distributed Embedded Real-Time Systems Design. In: **EA-AOSD'07 - 10th International Workshop on Early Aspects**, Vancouver, v. 4765, p. 55-74, 2007.

GéRARD, S.; SELIC, B. The UML – MARTE Standardized Profile. World congress of the international federation of automatic control. **Proceedings...** [S.l.: s.n.], 2008. p.6909–6913.

GUO, Z.; NAJJAR, W.; BUYUKKURT, B. Efficient hardware code generation for FPGAs. **ACM Trans. on Architecture and Code Optimization**, [S.1], v. 5, n. 1, p. 1–26, 2008.

HECHT, M.V.; PIVETA, E.; PIMENTA, M.; PRICE, R.T. Aspect-Oriented Code Generation. **XX Simpósio Brasileiro de Engenharia de Software**, Porto Alegre, 2006.

IUNG, B.; NEUNREUTHER, E.; MOREL, G. Engineering process of integrated distributed shop floor architecture based on interoperable field components. **International Journal of Computer Integrated Manufacturing,** [S.1], v. 14, n. 3, p.246-262, 2001.

KICZALES, G. et al. Aspect-Oriented Programming. European conference for object-oriented programming, ECOOP, 1997. **Proceedings...** Berlin: Springer-Verlang, 1997. p. 220-240.

LE BEUX, S.; MARQUET, P.; HONORE, A.; DEKEYSER, J-L. A Model Driven Engineering Design Flow to Generate VHDL**. International ModEasy'07 Workshop,** Barcelona, 2007.

LONG, Q.; LIU, Z.; LI, X.; JIFENG, H. Consistent Code Generation from UML Models. **Australian Software Engineering Conference (ASWEC'05)**, Brisbane, 2005.

MARTIN, G.; MÜLLER, W. **UML for SoC Design**. Netherlands: Springer, 2005.

MCUMBER, W.E.; CHENG, B.H. UML-Based Analysis of Embedded Systems Using a Mapping to VHDL. **4th IEEE International Symposium on High-Assurance Systems Engineering,** Washington D.C., 1999.

MISCHKALLA, F.; DA HE; MUELLER, W. Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems. **Design, Automation & Test in Europe Conference & Exhibition (DATE),** Dresden, 2010.

MOREIRA, T.G.; WEHRMEISTER, M.A.; PEREIRA, C.E.; PETIN, J.-F.; LEVRAT, E. Automatic code generation for embedded systems: From UML specifications to VHDL code. **8th IEEE International Conference on Industrial Informatics (INDIN)**, Osaka, p.1085-1090, 2010a.

MOREIRA, T.G.; WEHRMEISTER, M.A.; PEREIRA, C.E.; PETIN, J.-F.; LEVRAT, E. Generating VHDL Source Code from UML Models of Embedded Systems. **IFIP Advances in Information and Communication Technology**, Brisbane, p.125-136, 2010b.

NARAYAN, S.; VAHID, F.; GAJSKI, D. Translating System Specifications to VHDL. **IEEE European Design Automation Conference**, Amsterdam, 1991.

NASCIMENTO, F. A.; S. OLIVEIRA., M. F.; WEHRMEISTER, M. A.; PEREIRA, C. E.; WAGNER, F. R. MDA-based Approach for Embedded Software Generation from a UML/MOF Repository. **Symposium on integrated Circuits and Systems Design**, New York, 2006.

NEBEL, W.; OPPENHEIMER, F.; SCHUMACHER, G.; KABOUS, L.; RADETSKI, M.; PUTZKE-RÖMING, W. Object-oriented Specification and Design of Embedded Hard Real-Time Systems. **16th IFIP World Congress**, [S.1], 2000, p. 504-515.

OMG UML. **Unified Modeling Language.** 2010. Disponível em: <http://www.uml.org/>. Acesso em: Out. 2010.

OMG MDA. **Model Driven Architecture.** 2010. Disponível em: <http://www.omg.org/mda/>. Acesso em: Out. 2010.

OMG MARTE. **The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems.** 2010. Disponível em: < http://www.omgmarte.org/ >. Acesso em: Out. 2010.

OMG OCL. **Documents Associated With Object Constraint Language, Version 2.2**. 2010. Disponível em: <http://www.omg.org/spec/OCL/2.2/>. Acesso em: Out. 2010.

OMG QVT. **Documents Associated With MOF 2.0 Query/View/Transformation, V1.0.** 2010. Disponível em: <http://www.omg.org/spec/QVT/1.0/>. Acesso em: Out. 2010.

OMG MOF. **Meta Object Facility**. 2011. Disponível em: <http://www.omg.org/mof/>. Acesso em: Nov. 2011.

OMG XMI. **XMI Specification: MOF 2.0/XMI Mapping, V2.1.1**. 2010. Disponível em: <http://www.omg.org/spec/XMI/2.1.1/>. Acesso em: Out. 2010.

PARKINSON, M.F.; TAYLOR, P.M.; PARAMESWARAN, S. C to VHDL converter in a codesign environment. **Proceedings of VHDL International Users Forum**, Oakland, 1994.

PEREIRA, C.E.; CARRO, L. Distributed real-time embedded systems: Recent advances, future trends and their impact in manufacturing plant control. **Annual Reviews in Control**, Saint Etienne, v. 31, p. 81-92, 2007.

PERRY, D. L. **VHDL – Programming by Example**. 4. ed. McGraw-Hill, 2002.

PETIN, J-F.; IUNG, B.; MOREL, G. Distributed intelligent actuation and measurement (IAM) system within an integrated shop-floor organization. **Computers in Industry**, [S.1], v. 37, n. 3, p. 197-211, 1998.

POHL, C.; FUEST, R.; PORRMANN, M. vMAGIC - Automatic Code Generation for VHDL. **International Journal of Reconfigurable Computing**, [S.1], 2010.

PORRES, I. **A toolkit for manipulating UML models.** Technical Report 441, Turku Centre for Computer Science, 2002.

RASHID, A.; SAWYER, P.; MOREIRA, A.; ARAUJO, J. Early Aspects: A model for Aspect-Oriented Requirements Engineering. In: IEEE JOINT INTERNATIONAL CONFERENCE ON REQUIREMENTS ENGINEERING, 2002. **Proceedings...** Washington: IEEE Computer Society, 2002, p. 199-202.

RICCOBENE, E.; SCANDURRA, P.; ROSTI, A.; BOCCHIO, S. Model-driven design environment for embedded systems. **43rd ACM/IEEE Design Automation Conference**, San Francisco, 2006.

RIEDER, M.; STEINER, R.; BERTHOUZOZ, C.; CORTHAY, F.; STERREN, T. Synthesized UML, a practical approach to map UML to VHDL. **2nd International Workshop Rapid Integration of Software Engineering techniques**, Heraklion Crete, 2005.

SANGIOVANNI-VINCENTELLI, A. The Tides of EDA**. IEEE Design & Test of Computers**, [S.l.], v.20, n.6, 2003, p.59–75.

SOMMERVILLE, I. **Engenharia de Software.** 6ªed. São Paulo: Addison-Wesley, 2003.

STANKOVIC, J. A. et al. VEST: An Aspect-Based Composition Tool for Real-Time System. **9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)**, [S.1], 2003, p. 58-59.

TRANCHERO, M.; REYNERI, L. M. Automatic Generation of VHDL Code for Self-Timed Circuits from Simulink Specifications. **14th IEEE International Conference on Electronics, Circuits and Systems (ICECS)**, Marrakech, 2007.

TURLEY, J. **The Two Percent Solution**. Embedded Systems Design. TechInsights (18 December 2002). Disponível em: <http://eetimes.com/discussion/other/4024488/The-Two-Percent-Solution>. Acesso em: Nov. 2011.

VIDAL, J.; LAMOTTE, F.; GOGNIAT, G.; SOULARD, P; DIGUET, J-P. A codesign approach for embedded system modeling and code generation with UML and MARTE. In Design, **Automation & Test in Europe Conference & Exhibition (DATE)**, [S.1], 2009, p. 226-231.

VHDL. **Linguagem para descrição de hardware - Wikipédia.** 2010. Disponível em: <http://pt.wikipedia.org/wiki/VHDL>. Acesso em: Mar. 2010.

WEHRMEISTER, M. A. **Framework Orientado a Objetos para Projeto de Hardware e Software Embarcados para Sistemas Tempo-Real.** 2005. Dissertação (Mestrado em Ciências da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre.

WEHRMEISTER, M.A.; FREITAS, E.P.; PEREIRA, C.E.; RAMMIG, F. GenERTiCA: A Tool for Code Generation and Aspects Weaving. 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing, ISORC, 2008. **Proceedings...** [S.l.]: IEEE Computer Society, 2008.

WEHRMEISTER, M.A. **An Aspect-Oriented Model-Driven Engineering Approach for Distributed Embedded Real-Time Systems.** 2009. 201f. Tese (Doutorado em Ciências da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre.

WOLF, W. H. **Computers as Components : principles of embedded computing system design.** San Francisco: Morgan Kaufmann, 2001.

WOOD, S. K., et al. A Model-Driven development Approach to Mapping UML State Diagrams to Synthesizable VHDL. **IEEE Transactions on Computers**, [S.1], v. 57, n. 10, p. 1357-1371, 2008.

W3C. **eXtensible Markup Language (XML) 1.0 (Fifth Edition)**. 2008. Disponível em: <http://www.w3.org/TR/xml/>. Acesso em: Nov. 2011.