UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Tiago Rogério Mück

PROJETO UNIFICADO DE COMPONENTES EM HARDWARE E SOFTWARE PARA SISTEMAS EMBARCADOS

Florianópolis(SC)

2013

Tiago Rogério Mück

PROJETO UNIFICADO DE COMPONENTES EM HARDWARE E SOFTWARE PARA SISTEMAS EMBARCADOS

Dissertação submetida ao Programa de Pós Graduação em Ciências da Computação para a obtenção do Grau de Mestre em Ciência da Computação.
Orientador: Prof. Dr. Antônio Augusto Fröhlich

Florianópolis(SC)

2013

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

Mück, Tiago Rogério
Projeto Unificado de Componentes em Hardware e Software para Sistemas Embarcados [dissertação] / Tiago Rogério Mück ; orientador, Antônio Augusto Fröhlich - Florianópolis, SC, 2013.
137 p. ; 21cm

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação.

Inclui referências

1. Ciência da Computação. 2. Projeto em nível de sistema. 3. C++. 4. Síntese de alto nível. 5. Programação orientada a aspectos. I. Fröhlich, Antônio Augusto. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Tiago Rogério Mück

PROJETO UNIFICADO DE COMPONENTES EM HARDWARE E SOFTWARE PARA SISTEMAS EMBARCADOS

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre em Ciência da Computação", e aprovada em sua forma final pelo Programa de Pós Graduação em Ciências da Computação.

Florianópolis(SC), 15 de Fevereiro 2013.

Prof. Dr. Ronaldo dos Santos Mello
Coordenador do Curso

Banca Examinadora:

Prof. Dr. Antônio Augusto Fröhlich
Orientador

Prof. Dr. Cesar Albenes Zeferino
Universidade do Vale do Itajaí

Prof. Dr. Eduardo Augusto Bezerra
Universidade Federal de Santa Catarina

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

RESUMO

O crescente aumento na complexidade dos sistemas embarcados está ocasionando uma migração para técnicas de projeto em níveis mais altos de abstração, o que tem levado a uma convergência entre as metodologias de desenvolvimento de hardware e software. Este trabalho tem como objetivo principal contribuir nesse cenário propondo uma estratégia de desenvolvimento unificada que possibilita a implementação de componentes em hardware e software a partir de uma única descrição na linguagem C++. As técnicas propostas se baseiam em conceitos de *programação orientada a objetos* (do inglês *Object-oriented Programming* — OOP) e *programação orientada a aspectos* (do inglês *Aspect-oriented Programming* — AOP) para guiar uma estratégia de engenharia de domínio que facilita a clara separação entre a estrutura e comportamento-base de um componente das características que são específicas de implementações em hardware ou software.

Certos aspectos de um componente, como, por exemplo, alocação de recursos e a interface de comunicação, são modelados de maneiras distintas dependendo da implementação-alvo (hardware ou software). Este trabalho mostra como tais aspectos podem ser fatorados e encapsulados em *programas de aspecto* que são aplicados às descrições iniciais apenas quando o particionamento final entre hardware e software é definido. Os mecanismos de aplicação de aspectos são definidos via *metaprogramação estática* utilizando os *templates* do C++. Dessa forma, a extração de implementações em hardware ou software a partir de uma *implementação unificada* em C++ é direta e se dá através de transformações no nível da linguagem suportadas por uma grande gama de compiladores e ferramentas de *síntese de alto-nível* (do inglês *High-level Synthesis* — HLS).

Para avaliar a abordagem proposta, foi desenvolvida uma plataforma flexível para implementação de *System-on-Chip*s (SoCs) em dispositivos lógico programáveis. A infraestrutura de hardware/software desenvolvida utiliza uma arquitetura baseadas em *Network-on-Chip*s (NoCs) para prover um mecanismo de comunicação transparente entre hardware e software. A avaliação dos mecanismos propostos foi feita através da implementação de um SoC para aplicações PABX. Os resultados mostraram que a estratégia proposta resulta em componentes flexíveis e reusáveis com uma eficiência muito próxima a de componentes implementados especificamente para software ou hardware.

Palavras-chave: Projeto em nível de sistema; Co-projeto de HW/SW; Síntese de alto nível; Programação orientada a aspectos; C++

ABSTRACT

The increasing complexity of current embedded systems is pushing their design to higher levels of abstraction, leading to a convergence between hardware and software design methodologies. In this work we aim at narrowing the gap between hardware and software design by introducing a strategy that handles both domains in a unified fashion.
We leverage on *Aspect-oriented Programming* (AOP) and *Object-oriented Programming* (OOP) techniques in order to provide *unified* C++ descriptions of embedded system components. Such unified descriptions can be obtained through a careful design process focused on isolating aspects that are specific of hardware and software scenarios. Aspects that differ significantly in each domain, such as resource allocation and communication interface, were isolated in *aspect programs* that are applied to the unified descriptions before they are compiled to software binaries or synthesized to dedicated hardware using *High-level Synthesis* (HLS) tools.
Furthermore, we propose a flexible FPGA-based SoC platform for the deployment of SoCs in a HLS-capable environment. The proposed hardware/software infrastructure relies on a *Network-on-Chip*-based architecture to provide transparent communication mechanisms for hardware and software components. The proposed unified design approach and its transparent communication mechanisms are evaluated through the implementation of a SoC for digital PABX systems. The results show that our strategy leads to reusable and flexible components at the cost of an acceptable overhead when compared to software-only C/C++ and hardware-only C++ implementations.
Keywords: System-level design; HW/SW co-design; High-level synthesis; Aspect-oriented system design; C++

LISTA DE FIGURAS

Figura 1 Complexidade da descrição de um SoC em vários níveis de abstração (BLACK et al., 2009) . . . 20
Figura 2 Definição de templates em diagramas de classe UML . . . 27
Figura 3 Implementações em *Register Transfer Level* (RTL) de um algorítimo de multiplicação/acumulação . . . 29
Figura 4 Fluxo da síntese de alto nível . . . 31
Figura 5 Modelo *double roof* descrevendo o fluxo de projeto ESL (GERSTLAUER et al., 2009) . . . 34
Figura 6 Volume de vendas de ferramentas de *High-level Synthesis* (HLS) ao longo dos anos (SMITH, 2012) . . . 35
Figura 7 Fluxo de projeto do SystemCoDesigner (KEINERT et al., 2009) 39
Figura 8 Fluxo de projeto do SCE (DÖMER et al., 2008) . . . 40
Figura 9 Fluxo de projeto de hoje. Desenvolvido no passado. . . . 43
Figura 10 Fluxo de projeto do futuro. Sendo desenvolvido atualmente. . . 43
Figura 11 Visão geral da decomposição de domínio em ADESD (FRÖHLICH, 2001) . . . 48
Figura 12 Incorporação de aspectos usando um adaptador de cenário . . . 49
Figura 13 Famílias de componentes responsáveis pelo escalonamento de tarefas no EPOS. As famílias `Task`, `Scheduler` e `Criterion` são mostradas em mais detalhes. . . . 50
Figura 14 Definição do componente `Scheduler` . . . 52
Figura 15 Implementação do componente `Scheduler` . . . 53
Figura 16 Adaptação do escalonador para hardware (FLOR; MÜCK; FRÖHLICH, 2011) . . . 59
Figura 17 Comunicação entre componentes em modelos no nível de transação (à esquerda) e orientados a objetos (à direita) . . . 64
Figura 18 Aplicação de aspectos de hardware e software utilizando um adaptador de cenário . . . 65
Figura 19 Possíveis mapeamentos de um modelo orientado a objetos para uma implementação física . . . 71
Figura 20 Comunicação entre-domínios usando proxies e agents. . . . 71
Figura 21 Mapeamento da definição de um componente . . . 72
Figura 22 Diagrama de classes UML do framework de comunicação . . . 74

Figura 23 Componentes híbridos síncronos (MARCONDES, 2009) 77
Figura 24 Componentes híbridos assíncronos (MARCONDES, 2009) ... 78
Figura 25 Componentes híbridos autônomos (MARCONDES, 2009) ... 78
Figura 26 Visão geral da arquitetura proposta 84
Figura 27 Estrutura do roteador da RTSNoC 85
Figura 28 Nodo de CPU(a) e IO(b) 85
Figura 29 Organização dos componentes do EPOS 87
Figura 30 Diagrama estilo UML da plataforma virtual. Os módulos `Virtual_Platform`, `CPU_Node` e `IO_Node` são usados apenas para encapsular outros módulos. 89
Figura 31 Interações entre proxies e agents em uma chamada do tipo `SW->HW->SW` 91
Figura 32 Gerenciamento de componentes no EPOS 92
Figura 33 Definição da interface para os agents em hardware 94
Figura 34 Passos utilizados para obter a implementação final dos componentes a partir de código C++ unificado 95
Figura 35 Latência da invocação de um método através de diferentes domínios. A comunicação `SW->SW` e `HW->HW` está alinhada com o eixo Y da direita. 104
Figura 36 Diagrama de blocos do sistema PABX 105
Figura 37 Definição do componente ADPCM 106
Figura 38 Definição do componente DTMF 106
Figura 39 Uso de memória de C/C++ apenas para software (`SW`) *vs.* C++ unificado e adaptado para software (`U`) 107
Figura 40 Tempos médio de execução normalizado. Os valores absolutos são mostrados acima das respectivas barras. 108
Figura 41 Uso médio de área da FPGA 110
Figura 42 Desempenho do C++ apenas p/ hardware *vs.* C++ unificado. Na figura (b), os valores são normalizados para comparar todos os componentes na mesma escala. 110
Figura 43 Processo de geração do sistema EPOS (POLPETA; FRÖHLICH, 2005) 116

LISTA DE TABELAS

Tabela 1 Comparação entre os trabalhos que empregam síntese a partir do nível de sistema 45
Tabela 2 Padrões comuns de comunicação entre componentes. O chamador (*Caller*) requisita operações do chamado (*Callee*). 63
Tabela 3 Ferramentas utilizadas para a avaliação 99
Tabela 4 Desempenho e tempo de simulação das plataformas 100
Tabela 5 Resultados da síntese dos componentes da plataforma em uma FPGA 101
Tabela 6 Utilização de memória do sistema operacional EPOS 101
Tabela 7 Memória utilizada pelos proxies/agents em software. Valores em *bytes*. 102
Tabela 8 Recursos da FPGA utilizados pelos proxies/agents em hardware 102
Tabela 9 Uso de área para um proxy e um agent de um componente que define dois métodos com um parâmetro de entrada e um parâmetro de retorno. No caso do software, o resultado inclui o consumo de memória do `Component_Manager` 103
Tabela 10 Tempo de execução do C/C++ apenas p/ software *vs.* C++ unificado 108
Tabela 11 Uso de recursos da FPGA do C++ apenas p/ hardware *vs.* C++ unificado 109
Tabela 12 Área utilizada pelos outros IPs do sistema 112
Tabela 13 Uso de memória e recursos da FPGA do SoC. No particionamento híbrido, apenas o componente `Scheduler` está em software. 112

LISTA DE ABREVIATURAS, SIGLAS E SÍMBOLOS

AD Analógico-Digital

ADESD *Application-driven Embedded System Design*

ADPCM *Adaptative Differential Pulse-Code Modulation*

AOP *Aspect-oriented Programming*

APDL *Application Description Level*

ARDL *Architecure Description Level*

codec Codificador/Decodificador

DTMF *Dual-Tone Multi-Frequency*

DA Digital-Analógico

DMA *Direct Memory Access*

DSE *Design Space exploration*

EDA *Electronic Design Automation*

EPOS *Embedded Parallel Operating System*

ESL *Electronic System-Level*

FBD *Family-based Design*

FF *Flip-flop*

FFT *Fast Fourier Transform*

FPGA *Field Programmable Gate Array*

HAL *Hardware Abstraction Layer*

HDL *Hardware Description Language*

HLS *High-level Synthesis*

IP *Intelectual Property*

ISA *Instruction-set Architecture*

ISR *Interrupt Service Routine*

ISS *Instruction Set Simulator*

JAXA Agência Espacial Japonesa

KPN *Kahn Process Network*

LUT *Lookup Table*

MDE *Model-driven Engineering*

MMU *Memory Management Unit*

MPSoC *Multiprocessor System-on-Chip*

NoC *Network-on-Chip*

OOP *Object-oriented Programming*

PABX *Private Automatic Branch Exchange*

PBD *Platform-based Design*

PeaCE *Ptolemy Extension as a Codesign Environment*

PLD *Programmable Logic Device*

RTL *Register Transfer Level*

RTOS *Real-time Operating System*

RTSNoC *Real-time Star Network-on-Chip*

SCE *System-on-Chip Environment*

SDF *Synchronous data-flow*

SER *Specify-explore-refine*

SDL *System Description Language*

SLDL *System-level Design Language*

SO Sistema Operacional

SoC *System-on-Chip*

STL *C++ Standard Template Library*

TCL *Tool Command Language*

TLM *Transaction-level Model*

TSC *Timestamp Counter*

UML *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO p. 19
1.1 Objetivos e delimitação do escopo . . . . p. 21
1.2 Organização da dissertação . . . . p. 22

2 FUNDAMENTOS p. 25
2.1 Metaprogramação estática em C++ . . . . p. 25
2.2 Síntese de alto-nível de hardware . . . . p. 28
2.3 Computação reconfigurável . . . . p. 30

3 PROJETO DE SISTEMAS EMBARCADOS p. 33
3.1 Projeto em nível de sistema . . . . p. 33
3.2 Metodologias baseadas em componentes pré-validados . . . . p. 35
3.3 Engenharia guiada por modelos . . . . p. 37
3.4 Metodologias baseadas na síntese de hardware e software . . p. 38
3.5 Verificação no nível de sistema . . . . p. 40
3.6 Interfaceamento entre hardware e software . . . . p. 41
3.7 Sumário e discussão . . . . p. 42

4 PROJETO DE SISTEMAS EMBARCADOS GUIADO PELA APLICAÇÃO p. 47
4.1 Exemplo de decomposição de domínio: escalonamento de tarefas . . . . p. 49
4.1.1 Implementação do escalonador . . . . p. 51
4.2 ADESD aplicada ao projeto de hardware e software . . . . . p. 53

5 PROJETO UNIFICADO DE HARDWARE E SOFTWARE p. 57
5.1 Um escalonador de tarefas sintetizável . . . . p. 57
5.1.1 Semântica dos ponteiros . . . . p. 58
5.1.2 Gerenciamento de alocação . . . . p. 59
5.1.3 Definição da interface . . . . p. 60
5.1.4 Outras considerações sobre HLS . . . . p. 60
5.2 Diferenças entre hardware e software . . . . p. 62
5.3 Implementação unificada de componentes de hardware e software . . . . p. 64
5.3.1 Encapsulamento de aspectos de hardware e software p. 65
5.3.1.1 Alocação de recursos . . . . p. 66

5.3.1.2 Interface de comunicação . . . . p. 67
5.3.1.3 Definição do cenário . . . . p. 68
5.3.1.4 Definição do adaptador de cenário . . . . p. 70
5.4 Integração entre componentes unificados . . . . p. 70
5.4.1 Framework de comunicação . . . . p. 73
5.4.2 Componentes híbridos *vs* Proxy+Agent: comunicação e microarquiteturas . . . . p. 77
5.5 Sumário e discussão . . . . p. 79

6 IMPLEMENTAÇÃO p. 83
6.1 Uma plataforma para o desenvolvimento de SoCs . . . . . . p. 83
6.1.1 RTSNoC . . . . p. 84
6.1.2 Nodos de CPU e IO . . . . p. 84
6.1.3 EPOS . . . . p. 86
6.1.4 A plataforma virtual . . . . p. 87
6.2 Comunicação entre hardware e software . . . . p. 90
6.3 Sumário do fluxo de implementação . . . . p. 94
6.3.1 Elementos de entrada . . . . p. 96
6.3.2 Geração de proxies e agents . . . . p. 96
6.3.3 Geração de software e hardware . . . . p. 97
6.3.4 Geração da plataforma final . . . . p. 97

7 AVALIAÇÃO p. 99
7.1 Avaliação da plataforma . . . . p. 99
7.1.1 Eficiência dos proxies e agents . . . . p. 101
7.2 Estudos de caso . . . . p. 104
7.2.1 Resultados . . . . p. 107
7.2.1.1 Software . . . . p. 107
7.2.1.2 Hardware . . . . p. 109
7.2.1.3 SoC com diferentes particionamentos . . . p. 111

8 CONSIDERAÇÕES FINAIS p. 113
8.1 Limitações . . . . p. 114
8.2 Trabalhos futuros . . . . p. 115

Referências Bibliográficas p. 119

APÊNDICE A – Código fonte p. 130

APÊNDICE B – Compilando implementações unificadas p. 132

APÊNDICE C – Produção científica p. 134

1 INTRODUÇÃO

Sistemas embarcados são sistemas computacionais projetados para exercer funções específicas dentro de um sistema maior (MARWEDEL, 2003). Estes sistemas são *embarcados* (ou *embutidos*) como parte de um sistema completo que, muitas vezes, possui partes mecânicas, sensores e atuadores, sendo capaz de interagir diretamente com ambiente a sua volta.

A aplicação de sistemas embarcados varia desde pequenos dispositivos portáteis, como relógios digitais e tocadores de MP3, até sistemas complexos, como sistemas de controle automotivo e aviônicos. De fato, sistemas embarcados constituem grande parte do destino final dos processadores e componentes produzidos pela indústria de semicondutores. Segundo um estudo feito ainda em 2005, 99% dos microprocessadores produzidos na época eram utilizados em sistemas embarcados e, naquele momento, o número de sistemas embarcados em uso já havia superado o número de habitantes no planeta (Paul Pop, 2005).

Além da crescente disseminação, sistemas embarcados também tem se tornado cada vez mais complexos à medida que os avanços da indústria de semicondutores permitem o uso de recursos computacionais mais sofisticados em uma gama cada vez maior de aplicações. As restrições impostas a tais sistemas, como desempenho, consumo de energia, custo e confiabilidade, estão cada vez mais rigorosas, tornando, desta forma, cada vez mais difícil a tarefa de projetar sistemas embarcados, aumentando os custos de desenvolvimento e o *time-to-market*.

Grande parte desta dificuldade no projeto destes sistemas se deve a sua especificidade, o que frequentemente induz um projeto integrado de hardware e software. Além disto, a implementação deste projeto pode ocorrer em uma gama considerável de arquiteturas distintas, desde simples microcontroladores de 8 bits, até sistemas complexos que integram todo o projeto em uma única pastilha de silício, conhecidos como *System-on-Chip*s (SoCs). De fato, SoCs tem se tornado os principais componentes de muitos dispositivos eletrônicos de consumo, como *smartphones* e *media players*, além de equipamentos médicos e de aviação.

Nesse cenário, o desenvolvimento de SoCs a partir de componentes previamente projetados e validados tem despontado como uma das alternativas de projeto que permite abstrair a complexidade e aumentar o reuso de componentes na concepção de novos sistemas embarcados. Contudo, conforme ressaltado por Bergamaschi e Cohn (2002), o desenvolvimento de uma arquitetura de SoC dedicada a uma aplicação consiste em um processo de engenharia complexo e custoso, podendo demandar em um demasiado tempo de projeto, inviabilizando o seu uso na prática. A Figura 1, extraída de Black et al. (2009), ilustra essa

complexidade. A figura mostra três exemplos de projetos de SoCs de diferentes gerações: os SoCs utilizados no passado, os SoCs sendo utilizados no presente e os SoCs que serão utilizados no futuro e estão sendo projetados atualmente. As barras em cada geração mostram a complexidade do projeto de hardware em diferentes níveis de abstração. Como pode ser observado na figura, uma descrição de um projeto atual no nível de transferência de registradores (do inglês *Register Transfer Level* — RTL) já está na ordem de um milhão de linhas de código, o que é praticamente impossível de ser gerenciado manualmente.

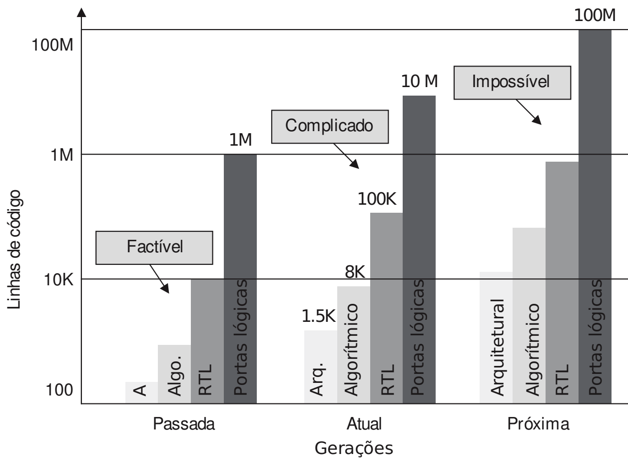

Figura 1: Complexidade da descrição de um SoC em vários níveis de abstração (BLACK et al., 2009)

Para lidar com tal complexidade, os projetos de sistemas embarcados tendem a ser feitos em níveis de abstração cada vez mais altos, incitando o desenvolvimento de metodologias de projeto no nível de sistema, também chamado de *Electronic System-Level* (ESL) *design*. A premissa básica do projeto ESL é modelar o sistema como um todo usando uma linguagem de alto nível e empregar um conjunto de técnicas de refinamento e síntese de alto nível com o objetivo de obter uma implementação final do sistema diretamente a partir do modelo inicial de forma (semi-)automática. Nesse cenário, uma convergência entre o projeto de hardware e software é necessária, pois uma abordagem unificada facilita a postergação das decisões sobre o particionamento entre hardware e software para as fases finais do projeto (GERSTLAUER et al., 2008).

Seguindo essa linha, nos últimos anos um grande esforço tem sido concentrado em ferramentas de *automação de projeto eletrônico* (do inglês *Electronic Design Automation* — EDA) com o objetivo de aproximar o de-

senvolvimento de hardware e software. Os avanços obtidos tem possibilitado a criação de fluxos de projeto baseados em síntese de hardware a partir de descrições comportamentais dos mesmos, viabilizando a descrição de hardware em linguagens como C e C++, e o uso técnicas de alto nível, como *programação orientada a objetos* (do inglês *Object-oriented Programming* — OOP). O foco dessas ferramentas, no entanto, é apenas síntese de hardware. Elas não introduzem uma metodologia clara para o desenvolvimento de componentes que possam ser implementados tanto em hardware quanto em software.

1.1 OBJETIVOS E DELIMITAÇÃO DO ESCOPO

Com o objetivo de estreitar as diferenças entre o desenvolvimento de hardware e software e contribuir nos avanços em direção a uma abordagem realmente unificada, este trabalho propõe técnicas e mecanismos que possibilitam a implementação de componentes em hardware e software a partir de uma única descrição na linguagem C++.

As técnicas propostas neste trabalho são construídas com base na *metodologia de projeto de sistemas embarcados guiado pela aplicação* (do inglês *Application-driven Embedded System Design* — ADESD) (POLPETA; FRÖHLICH, 2005). ADESD é uma metodologia de engenharia de domínio que se baseia em uma estratégia bem definida de decomposição do domínio através do *projeto baseado em famílias* (do inglês *Family-based Design* — FBD) (PARNAS, 1976) e OOP, focando também na aplicação de técnicas de *programação orientada a aspectos* (do inglês *Aspect-oriented Programming* — AOP) (KICZALES et al., 1997) já nos estágios iniciais do projeto. Desta maneira, ADESD guia um processo de engenharia de domínio de forma a criar famílias de componentes nas quais as dependências relativas ao cenário de execução são fatoradas em *programas de aspecto* e as relações externas entre os componentes são capturadas em um framework.

Este trabalho mostra como a estratégia de engenharia de domínio de ADESD facilita a clara separação entre a estrutura e comportamento-base de um componente das características que são específicas de implementações em hardware ou software. Certos aspectos de um componente, como, por exemplo, alocação de recursos e a interface de comunicação, são modelados de maneiras distintas dependendo da implementação-alvo (hardware ou software). Este trabalho mostra como tais aspectos podem ser encapsulados para serem utilizados em diferentes cenários e agregados às descrições iniciais apenas quando o particionamento final entre hardware e software é definido, só então gerando a implementação final do componente no cenário-alvo (hardware ou software).

De forma a permitir o desenvolvimento de descrições que possam ser eficientemente sintetizadas para hardware ou compiladas como software, tanto a *descrição unificada* dos componentes, *i.e.* a descrição independente de cenário, quanto os programas de aspecto que adaptam as mesmas para hardware ou software são definidos utilizando a linguagem C++. Além disso, os mecanismos de aplicação de aspectos são definidos via *metaprogramação estática* (CZARNECKI; EISENECKER, 2000) utilizando os *templates* do C++, uma funcionalidade intrínseca da própria linguagem. Dessa forma, a extração de implementações em hardware ou software a partir de uma implementação em nível de sistema (a *implementação unificada* em C++) é direta e se dá através de transformações no nível da linguagem, utilizando semânticas definidas pelo projetista que são independentes de fluxos de projeto e ferramentas específicas.

É importante ressaltar que este trabalho foca principalmente em diretivas gerais de implementação. O desenvolvimento de um fluxo completo para projeto de sistemas embarcados está fora do seu escopo. Dessa forma, questões como a exploração do espaço de projeto, integração entre etapas do fluxo e verificação, são abordadas apenas com a profundidade suficiente para dar suporte às ideias principais do trabalho.

1.2 ORGANIZAÇÃO DA DISSERTAÇÃO

O Capítulo 2 apresenta os conceitos básicos necessários para a compreensão dos mecanismos propostos neste trabalho, enquanto o Capítulo 3 apresenta a revisão bibliográfica. O Capítulo 3 foca nos conceitos de projeto ESL, apresentando os trabalhos relacionados a essa área.

Os capítulos 4 e 5 apresentam as técnicas e mecanismos que possibilitam a implementação de componentes em software e hardware a partir de uma única descrição na linguagem C++. No Capítulo 4, é feita uma introdução à ADESD e aos conceitos básicos utilizados para o encapsulamento e aplicação de aspectos. A estratégia de decomposição e engenharia de domínio é demonstrada por meio de um dos estudos de caso: um escalonador de recursos para um Sistema Operacional (SO). O Capítulo 5 apresenta, então, a principal contribuição deste trabalho, demostrando como as características específicas de hardware e software podem isoladas para produzir descrições unificadas de componentes.

O Capítulo 6 tem como objetivo abordar questões práticas relacionadas à implementação das ideias propostas no Capítulo 5, descrevendo, principalmente, a arquitetura de hardware/software utilizada para suportar a comunicação transparente entre hardware e software. A arquitetura descrita foi desenvolvida com o objetivo principal de prover uma plataforma flexível para a

implementação de SoCs em *Field Programmable Gate Array*s (FPGAs). O seu desenvolvimento foi iniciado anteriormente a este trabalho por um grupo do *Laboratório para Integração de Software e Hardware* (LISHA) e foi posteriormente expandida para suportar os mecanismos transparentes de comunicação propostos.

Para avaliar a eficiência da abordagem proposta, o Capítulo 7 apresenta a implementação de um sistema de *troca automática de ramais privados* (do inglês *Private Automatic Branch Exchange* — PABX), da qual três componentes forem reimplementados usando a estratégia de projeto unificado: um escalonador de recursos, um codec ADPCM e um detector de DTMF. Finalmente são apresentadas no Capítulo 8 as considerações finais e direções para trabalhos futuros.

2 FUNDAMENTOS

Ao longo deste trabalho, assume-se um conhecimento prévio a cerca dos conceitos básicos de OOP, *Unified Modeling Language* (UML), linguagem C++ e projeto de hardware digital. O leitor pode consultar Larman (2005), Czarnecki e Eisenecker (2000), Alexandrescu (2001) e Chu (2006) para uma explicação detalhada sobre esses conceitos. Neste capítulo, é feita apenas uma introdução aos conceitos mais específicos relacionados a proposta deste trabalho: síntese de alto nível de hardware e metaprogramação estática em C++. Ao final, também é feita uma revisão sobre computação reconfigurável e dispositivos lógico programáveis (do inglês *Programmable Logic Device* — PLD), como as FPGAs. Os avanços em FPGAs tem permitido a implantação de SoCs complexos e totalmente customizados sem a necessidade de passar pelo custoso processo de desenvolvimento e fabricação de circuitos integrados. Esta possibilidade está tornando as FPGAs uma tecnologia importante para o desenvolvimento de sistemas embarcados.

2.1 METAPROGRAMAÇÃO ESTÁTICA EM C++

De uma maneira geral, metaprogramação estática (CZARNECKI; EISENECKER, 2000) consiste em definir programas que são executados em *tempo de compilação*. Na linguagem C++ (e também em outras linguagens como Curl, D e XL), metaprogramação é feita utilizando *templates*[1].

Em um primeiro momento, templates podem ser vistos como um mecanismo para a criação de funções e classes genéricas. Assim, a tipologia de dados usadas pelas classes podem ser definidas como parâmetros. Por exemplo, uma classe que define um vetor de um tipo de dado genérico pode ser descrita da forma mostrada a seguir:

```
1 template <typename T, int SIZE>
2 class Vector {
3     T data[SIZE];
4     ...
5 };
6
7 Vector<int, 8>    int_vector; // vetor de 'int' contendo 8 elementos
8 Vector<float, 4> float_vector; // vetor de 'float' contendo 4 elementos
```

A declaração `template <typename T, int SIZE>` especifica que a classe `Vector` possui como parâmetro um tipo de dado, que é usado para definir o tipo do vetor, e um valor inteiro, que é usado para definir o tamanho

[1] o leitor pode consultar Czarnecki e Eisenecker (2000) e Alexandrescu (2001) para uma explicação detalhada sobre o uso de templates em C++.

do vetor. Classes ou funções que possuam parâmetros de template são também chamadas de `templates de classes`, pois o compilador gera o código que implementa a classe apenas quando a mesma é efetivamente utilizada em outro ponto do código. Para cada novo conjunto de parâmetros utilizados, o compilador gera, internamente, uma nova implementação específica para os parâmetros utilizados. Dessa forma, todos os tipos e valores utilizados como parâmetros do template devem ser constantes e conhecidos em tempo de compilação.

Um conceito chave para entender os mecanismos metaprogramados desse trabalho é a *especialização de templates*. Para ilustrar esse processo, será utilizado um problema muito comum: a implementação de vetores de valores booleanos. Muitos sistemas utilizam 16 ou 32 bits para implementar uma variável do tipo `bool`, mesmo que apenas um único bit seja necessário para armazenar o estado da variável. Dessa forma, pode-se economizar memória ao utilizar, por exemplo, uma única variável inteira de 32 bits para representar um vetor booleano de tamanho 32. Assim, um vetor do tipo `bool` requer uma implementação especializada. Essa especialização pode ser definida como mostrado a seguir:

```
1  // Implementação genérica
2  template <typename T, int SIZE>
3  class Vector {
4      T data[SIZE];
5      ...
6  };
7
8  // Especialização para o tipo bool de 32 bits
9  template <>
10 class Vector<bool,32> {
11     unsigned int data;
12     ...
13 };
14
15 Vector<int, 8>   int_vector ; // mapeia para a implementação genérica
16 Vector<float, 4> float_vector ;
17 Vector<bool, 32> bool_vector; //mapeia para a implementação especializada
```

A classe pode ser redefinida para um conjunto específico de valores para os parâmetros do templates. Sempre que os valores especificados forem utilizados, o compilador utiliza a implementação especializada da classe. Para todos os outros valores, o compilador gera uma nova implementação com base na definição genérica da classe.

Templates também podem ser *parcialmente especializados*. No exemplo anterior, não seria viável definir uma nova especialização de `Vector` para todos os possíveis tamanhos de vetores booleanos, logo, pode-se criar um novo template de classe, que especializa parcialmente `Vector` apenas para o tipo `bool`:

```
1  template <int SIZE>
2  class Vector<bool,SIZE> {
```

```
3     unsigned int data[SIZE/32];
4     ...
5 };
```

Tendo definido os principais conceitos relacionados aos templates do C++, a questão é como utiliza-los como um mecanismo efetivo de metaprogramação estática, ou seja, como definir programas que são executados em *tempo de compilação*. Um exemplo clássico utilizado para demonstrar o que é metaprogramação é a implementação de uma função que calcula o fatorial de um número. Esta função pode ser implementada em C++ como mostrado abaixo:

```
1 unsigned int factorial (unsigned int n) {
2     return (n==0)? 1 : n * factorial (n-1);
3 }
4
5 int x = factorial (4); // == (4 * 3 * 2 * 1 * 1) == 24
6 int y = factorial (0); // == 0! == 1
```

Este programa executa em tempo de execução para calcular o fatorial de 4 e 0. Como as entradas para a função de fatorial já são estáticas, a função de fatorial pode ser metaprogramada para calcular os resultados em tempo de compilação. Isso é feito através da especialização do template `Factorial` para parâmetro `0`, estabelecendo a condição de parada para a recursão:

```
1  template <int N>
2  struct Factorial {
3      enum { value = N * Factorial<N - 1>::value };
4  };
5
6  template <>
7  struct Factorial<0> {
8      enum { value = 1 };
9  };
10
11 // Factorial<4>::value == 24
12 // Factorial<0>::value == 1
13 int x = Factorial<4>::value; // == 24
14 int y = Factorial<0>::value; // == 1
```

Ao longo deste trabalho, diagramas de classe UML são utilizados para descrever estudos de caso e vários dos mecanismo propostos. O padrão UML não define uma maneira de expressar artefatos metaprogramados baseados em templates. Dessa forma, este trabalho usa uma notação própria, ilustrada no exemplo da figura 2.

No diagrama UML, os parâmetros de template são especificados junto ao nome da classe. No exemplo da figura 2, `Char_Vector` define um vetor do tipo `char`, herdando a implementação definida em `Vector`. A herança possui o rótulo `<char,SIZE>`, o que significa que `Char_Vector` está herdando de `Vector` usando o tipo `char` e o valor de `SIZE` (definido no escopo de `Char_Vector`) como valores para os parâmetros de template `T` e `SIZE`.

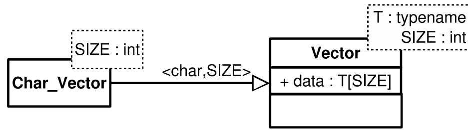

Figura 2: Definição de templates em diagramas de classe UML

2.2 SÍNTESE DE ALTO-NÍVEL DE HARDWARE

Síntese de alto-nível (do inglês *High-level Synthesis* — HLS), também chamado de síntese comportamental, ou também síntese ESL (*Electronic System-Level*), é um processo automático que interpreta uma descrição algorítmica de um comportamento e gera uma descrição do hardware que implementa tal comportamento no nível de transferência de registradores (do inglês *Register Transfer Level* — RTL). Uma descrição em RTL é normalmente composta por uma máquinas de estado controlando um *datapath* que consistem em unidades funcionais (somadores, multiplexadores, portas lógicas, etc.), banco de registradores, memórias e elementos de interconexão.

O estado-da-arte em ferramentas EDA (*e.g.* CatapultC (Calypto Design Systems, 2011), Synphony (Synopsys, 2011), C-to-Silicon (Cadence Design Systems, 2011), Cynthesizer (Forte Design Systems, 2011), AutoESL (Xilinx, 2012)) já suporta a síntese de hardware a partir de descrições algorítmicas em linguagens baseadas em C e C++. Além da descrição algorítmica, essas ferramentas também utilizam como entrada um conjunto de restrições de área e desempenho que vão definir as unidades funcionais que comporão a microarquitetura. Com base nessas entradas, as operações são escalonadas em diferentes ciclos de clock e unidades funcionais. A saída dessas ferramentas normalmente é uma descrição RTL em uma *linguagem de descrição de hardware* (do inglês *Hardware Description Language* — HDL) (*e.g.* VHDL (IEEE, 2000) ou Verilog (IEEE, 2001)) que é utilizada como entrada nos níveis mais baixos do fluxo de implementação.

Para exemplificar esse processo, considere o exemplo a seguir: um componente que recebe um conjunto de valores como entrada, multiplica esses valores por um fator e tem como saída a acumulação dos resultados. Em C++, esse comportamento pode ser implementado na função abaixo:

```
1 int mult_acc_array(int input[SIZE], int factor) {
2     int result = 0;
3 
4     mac: for (int i = 0; i < SIZE; ++i)
5         result += input[ i ] * factor;
6 
7     return result;
8 }
```

A função recebe como entrada um *array* e o fator. O corpo da função consiste em um *loop* que itera pelo array multiplicando e acumulando os valores. Em RTL, este mesmo algorítimo pode ser implementado utilizando diferentes microarquiteturas de acordo com a métricas de área e desempenho que pretende-se atingir. A figura 3 ilustra três dessa possíveis microarquiteturas.

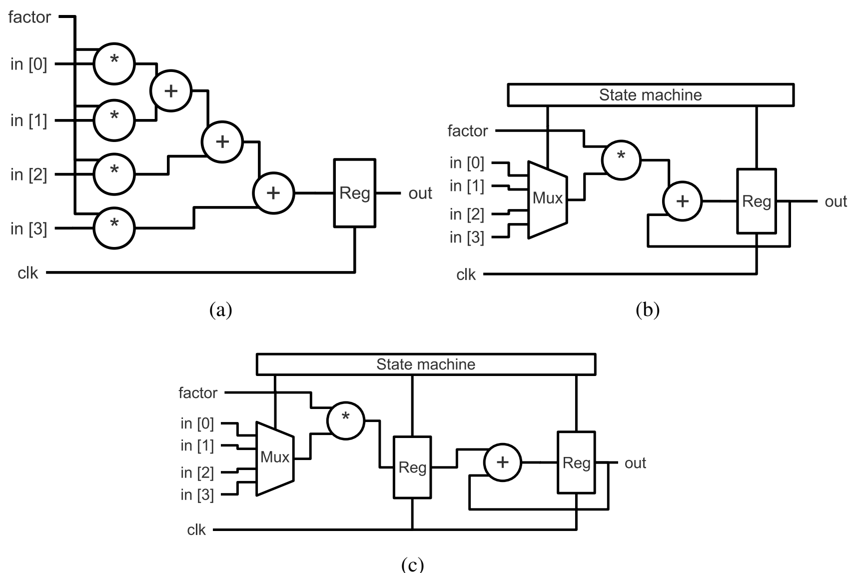

Figura 3: Implementações em RTL de um algorítimo de multiplicação/acumulação

A figura 3a ilustra uma microarquitetura totalmente *paralela*, na qual todas as iterações do loop são executadas no mesmo ciclo de clock. Isso é feito utilizando-se vários multiplicadores e somadores para gerar um novo resultado a cada ciclo. A figura 3b já mostra uma implementação *serial*, na qual cada iteração leva um ciclo de clock. Essa implementação requer menos área ao custo de ciclos extras. Uma implementação que utiliza um *pipeline* para melhorar a vazão do circuito é mostrada na figura 3c. Nesta, a multiplicação e a acumulação são executadas em ciclos diferentes.

Apesar de o processo de geração de código RTL ser automatizado, as ferramentas não são capazes de inferir automaticamente as microarquiteturas. O desenvolvedor deve "guiar" o processo de síntese de forma que a ferramenta gere a microarquitetura desejada. Este tipo de decisão é tomada baseada em

diretivas de síntese. O exemplo a seguir estende o exemplo anterior, definindo diretivas de modo que o circuito resultante seja o mesmo mostrado na figura 3a[2]:

```
1   #define SIZE 4
2
3   #pragma hls_design top
4   #pragma hls_pipeline_init_interval 1
5   int mult_acc_array(int input[SIZE], int factor) {
6
7       #pragma hls_resource input_resource variables="input,␣factor" \
8               map_to_module="mgc_ioport.mgc_in_wire␣"
9
10      #pragma hls_resource output_resource variables="mult_acc_array.return" \
11              map_to_module="mgc_ioport.mgc_out_stdreg"
12
13      int result = 0;
14
15      #pragma hls_unroll yes
16      mac: for (int i = 0; i < SIZE; ++i)
17          result += input[ i ] * factor;
18
19      return result;
20  }
```

As diretivas são definidas usando *pragmas*[3]. Neste exemplo, a diretiva `#pragma hls_unroll yes` define que o loop deve ser "desenrrolado", o que indica a execução paralela de todas as suas iterações. Estas diretivas também podem ser especificadas separadamente da descrição do algoritmo em C++. O exemplo a seguir define as mesmas diretivas mostradas anteriormente usando um *script* em *Tool Command Language* (TCL):

```
1   directive set -DESIGN_HIERARCHY mult_acc_array
2   directive set /mult_acc_array/core/main -PIPELINE_INIT_INTERVAL 1
3   directive set /mult_acc_array/core/mac -UNROLL yes
4   directive set /mult_acc_array/mult_acc_array.return:rsc -MAP_TO_MODULE
        mgc_ioport.mgc_out_stdreg
5   directive set /mult_acc_array/factor:rsc -MAP_TO_MODULE mgc_ioport.mgc_in_wire
6   directive set /mult_acc_array/input:rsc -MAP_TO_MODULE mgc_ioport.mgc_in_wire
```

De fato, essa é a opção mais utilizada pelos desenvolvedores, pois permite a definição de diferentes microarquiteturas sem que sejam feitas modificações na descrição C++ do algorítimo. A figura 4 sumariza o processo de síntese de alto nível. O leitor pode consultar Fingeroff (2010) para uma explicação detalhada sobre os conceitos relacionados à HLS.

2.3 COMPUTAÇÃO RECONFIGURÁVEL

Computação reconfigurável tem adquirido importância no cenário de desenvolvimento de sistemas embarcados. Por um lado, busca-se aumentar

[2] diretivas compatíveis com a ferramenta CatapultC (Calypto Design Systems, 2011)

[3] a diretiva `#pragma` é o método definido pelo padrão ANSI C/C++ para repassar informações adicionais ao compilador. A semântica da cada pragma varia segundo a plataforma e o compilador.

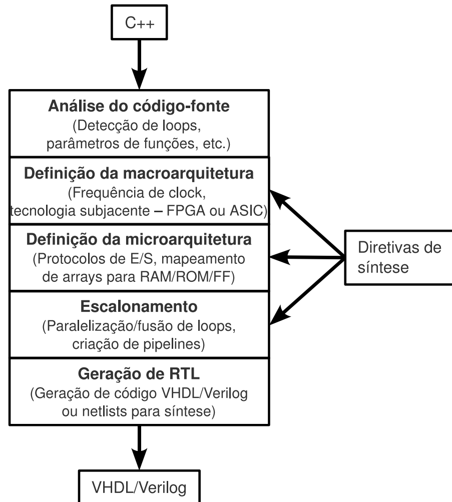

Figura 4: Fluxo da síntese de alto nível

o desempenho de funções implementando-as em hardware dedicado, por outro, busca-se manter a flexibilidade oferecida pelo software. Os dispositivos mais utilizados são as FPGAs. As FPGAs contem elementos programáveis chamados de *blocos lógicos*, e uma interconexão hierárquica que permite um roteamento configurável entre os sinais de entrada e saída dos blocos. Os blocos lógicos podem ser configurados para executar diversas funções combinacionais, desde simples portas lógicas AND e OR até funções complexas, ou funções sequenciais utilizando flip-flops. A maioria das FPGAs modernas também incluem elementos com funções dedicadas, como multiplicadores e blocos de memória SRAM, de forma a tornar mais eficientes os circuitos implementados nas mesmas. Um estudo detalhado sobre a organização e implementação física de tais dispositivos e sobre computação reconfigurável é apresentado por Compton e Hauck (2002).

Tal como ocorre no processo de desenvolvimento de software, o desenvolvimento de um componente de hardware para um dispositivo programável segue um fluxo de transformações que parte de uma descrição em RTL até chegar ao efetivo conjunto de bits de configuração do dispositivo (conhecido como *bitstream*). Inicialmente, uma HDL é utilizada para descrever o circuito que se deseja implementar em RTL. A síntese desta descrição é feita através do uso de uma ferramenta específica que irá produzir uma *netlist*, uma representação do circuito em termos de primitivas lógicas básicas. Esta representação pode ser então mapeada nas estruturas presentes no dispositivo reconfigurável específico,

em um processo denominado *place-and-route*, em alusão ao processo de alocar as diversas primitivas nos respectivos blocos lógicos presentes no dispositivo (*place*) e rotear as entradas e saídas dos blocos de forma a corresponder com o circuito sintetizado (*route*).

Mais recentemente, chips híbridos de FPGA surgiram no mercado. Estes agregam também dispositivos de hardware dedicado tais como processadores de uso geral e interfaces específicas de I/O, de forma a estender as funcionalidades das FPGAs. Exemplos são os dispositivos da família Zynq-7000 (Xilinx, 2012b) que agregam um processador ARM dual-core Cortex A9 e implementações dedicadas de alto desempenho de controladores de memória, PCIe, DMA e Gigabit Ethernet.

Os grandes fabricantes de FPGAs tem, de fato, feito um grande esforço para aumentar a sua penetração no mercado de sistemas embarcados baseados em SoCs. Este esforço tem sido direcionado principalmente na integração de soluções de HLS nos seus fluxos de projeto comerciais. A Xilinx, por exemplo, recentemente adquiriu a ferramenta de HLS AutoESL (Xilinx, 2012) e já disponibiliza um fluxo de projeto completo, partindo de C++/SystemC, até implementações completas em arquiteturas baseadas em barramentos nos seus dispositivos (CONG et al., 2011).

3 PROJETO DE SISTEMAS EMBARCADOS

Sistemas embarcados tem se tornado cada vez mais complexos à medida que os avanços da indústria de semicondutores permitem o uso de recursos computacionais mais sofisticados em uma gama cada vez maior de aplicações. Tais avanços também tem possibilitado a integração de boa parte das funcionalidades em uma única pastilha de silício, resultando nos sistemas conhecidos como SoCs. No entanto, o desenvolvimento de um SoC dedicado a uma aplicação consiste em um processo complexo e custoso. Para lidar com tal complexidade, os projetos de sistemas embarcados tendem a ser feitos em níveis de abstração cada vez mais altos, incitando o desenvolvimento de metodologias de projeto no nível de sistema, também chamado de ESL (*Electronic System-Level*) *design*, ou simplesmente *system-level design*.

Neste capítulo, é feita uma introdução ao projeto em nível de sistema, seguida por uma revisão dos trabalhos relacionados a essa área.

3.1 PROJETO EM NÍVEL DE SISTEMA

A premissa básica do projeto ESL é modelar o comportamento de todo o sistema usando uma linguagem de alto nível, como C e C++. Nos últimos anos, novas linguagens foram criadas especificamente para esse propósito, como, por exemplo, SystemC (PANDA, 2001) e SpecC (GAJSKI et al., 2000; FUJITA; NAKAMURA, 2001). Ambas as linguagens estendem C++ com mecanismos que permitem uma modelagem explicita de componentes, concorrência, temporização e sincronização, tipos de dados com precisão em nível de bits, além de um ambiente de simulação orientado a eventos, permitindo que os modelos sejam compilados e simulados para uma avaliação inicial do funcionamento do sistema. Linguagens que possuem essas capacidades também são conhecidas como *linguagens de descrição de sistemas* (do inglês *System Description Language* — SDL) ou *linguagens para projeto em nível de sistema* (do inglês *System-level Design Language* — SLDL).

De uma maneira geral, o fluxo de projeto ESL segue uma abordagem top-down que pode ser generalizada pelo modelo *double roof* (GERSTLAUER et al., 2009), mostrado na figura 5.

O fluxo de projeto representado por esse modelo começa com uma especificação ESL dada por um modelo comportamental no qual não é possível distinguir hardware de software. Em cada nível, um processo de síntese (flechas verticais) transforma uma especificação em uma implementação. Em ESL este processo de síntese é, então, o processo de seleção de uma arquitetura

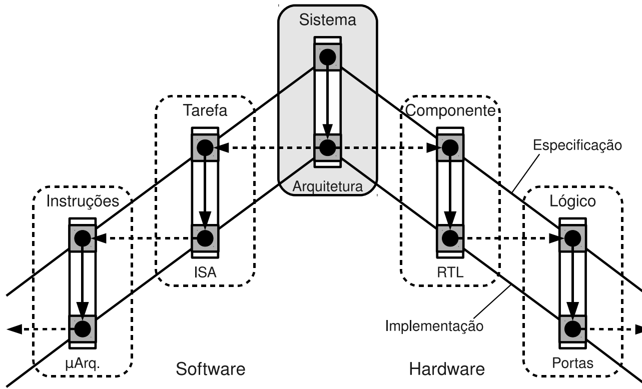

Figura 5: Modelo *double roof* descrevendo o fluxo de projeto ESL (GERSTLAUER et al., 2009)

adequada para a plataforma (com base em requisitos como área, latência, etc.), determinar um mapeamento do modelo comportamental para essa arquitetura e gerar um novo modelo refinado específico para a plataforma. Os componentes desse modelo refinado são então usados como especificação para os níveis de abstração mais baixos (setas horizontais), nos quais hardware e software seguem fluxos distintos.

A síntese nos níveis mais baixos segue um processo similar, no qual um componente comportamental é refinado para uma implementação estrutural. Por exemplo, em software, no nível de tarefas, processos ou *threads* alocados para um mesmo processador são traduzidos para o conjunto de instruções (do inglês *Instruction-set Architecture* — ISA) do processador, juntamente com um sistema operacional de tempo-real (do inglês *Real-time Operating System* — RTOS) ou um ambiente customizado para suporte em tempo de execução. Esse processo é feito utilizando um conjunto de cross-compiladores/ligadores específicos para a ISA e o RTOS. Por outro lado, no nível de componentes em hardware, os processos são sintetizados para implementações em RTL utilizando tecnologias de HLS.

No entanto, HLS começou a ganhar espaço apenas recentemente. Como pode ser visto na figura 6, o uso de ferramentas com esta tecnologia teve um crescimento pequeno até o ano de 2009. Assim, muitas das metodologias propostas até aquele momento têm o seu fluxo de hardware baseado no mapeamento de processos para componentes pré-existentes e já validados, conhecidos como *propriedades intelectuais* (do inglês *Intelectual Property* — IP) (GAJSKI, 1999). O desenvolvimento baseado em IPs é uma técnica promissora para o tra-

tamento da complexidade no projeto de sistemas embarcados (MARTIN et al., 2001), no entanto, o uso de IPs apresenta limitações, como a sua dependência arquitetural em relação a protocolos de comunicação específicos (WILTON; SALEH, 2001). Nessas situações, o projetista deve prover uma implementação manual em RTL do componente desejado.

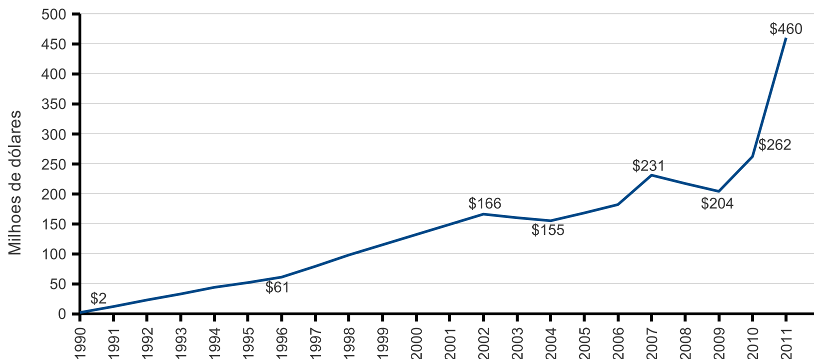

Figura 6: Volume de vendas de ferramentas de HLS ao longo dos anos (SMITH, 2012)

3.2 METODOLOGIAS BASEADAS EM COMPONENTES PRÉ-VALIDADOS

Várias metodologias e ferramentas de projeto foram propostas com o objetivo de prover um maior acoplamento entre os fluxos de projeto de hardware e software. Muitas dessas metodologias são baseadas na ideia de construir um sistema a partir de IPs.

Nesse contexto, Cesario et al. (2002) propõe o uso de projeto baseado em componentes para o desenvolvimento de plataformas SoC multiprocessadas. Este trabalho propõe uma metodologia unificada para a integração automática de IPs. Esta metodologia é utilizada por um fluxo de projeto chamado ROSES (DZIRI et al., 2004), que visa a geração do hardware, software, e de interfaces funcionais dos subsistemas de forma automática, a partir de um modelo arquitetural do sistema.

Também baseado no uso de componentes, o projeto Ptolemy (EKER et al., 2003) foca na modelagem de sistemas através da composição de componentes heterogêneos. O Ptolemy propõe um modelo de estrutura e um framework semântico que suporta diversos modelos de computação, juntamente com um

conjunto de ferramentas para simulação do modelo e posterior geração de código. O *Ptolemy Extension as a Codesign Environment* (PeaCE) (HA et al., 2008) estende o Ptolemy com um framework para síntese de sistemas embarcados de tempo real para aplicações multimídia. O PeaCE utiliza como entrada modelos compostos por grafos de *fluxo de dados síncrono* (do inglês *Synchronous data-flow* — SDF) e máquinas de estado finitas estendidas e gera código C a partir dos mesmos, além de suportar o mapeamento para IPs e processadores.

O *Projeto Baseado em Plataformas* (do inglês *Platform-based Design* — PBD) (SANGIOVANNI-VINCENTELLI; MARTIN, 2001) propõe o reuso de plataformas de hardware e software, padronizadas e previamente validadas, orientadas para determinados domínios de aplicação. A ideia principal é que, caso essa plataforma possa atender as restrições do projeto de um conjunto grande de aplicações, o custo da própria plataforma pode ser pulverizado dentre esse conjunto de aplicações, favorecendo assim o desenvolvimento das mesmas. No fluxo do projeto baseado em plataformas parte-se de uma instância da aplicação e é feita uma *exploração do espaço de projeto* (do inglês *Design Space exploration* — DSE) inicial para definir uma plataforma-base para o sistema. Após uma DSE da plataforma, é definida uma instância da mesma que satisfaça os requisitos da aplicação. Os frameworks *Metropolis* (BALARIN et al., 2003) e o seu sucessor *Metro-II* (DAVARE et al., 2007) seguem a metodologia PBD. Eles propõem o uso de meta-modelos formais como descrições iniciais dos projetos. Um ambiente que suporta simulação, validação formal e síntese é disponibilizado.

O trabalho de Mrabti et al. (2009) segue uma abordagem semelhante à PBD. Os autores propõem dois modelos de descrição, *Application Description Level* (APDL) e *Architecure Description Level* (ARDL), que são utilizados para descrever a aplicação e as plataformas, respectivamente. APDL é baseada em um modelo SDF que suporta canais com múltiplas entradas e saídas, além de mecanismos de arbitragem individuais para cada canal. Foi proposta uma ferramenta que mapeia os modelos APDL para componentes definidos usando ARDL e gera o código que implementa a aplicação, assim como o *Hardware Abstraction Layer* (HAL) específico da plataforma alvo.

Apesar de prover frameworks de integração para todo o fluxo de projeto, as metodologias citadas acima não definem uma maneira clara para projetar cada componente de forma que os mesmos sejam reusáveis. Além disso, o mapeamento de um modelo da aplicação pra componentes pré-existentes limita os possíveis particionamentos entre hardware e software. No caso específico de abordagens derivadas de PBD, Sangiovanni-Vincentelli et al. (2004) já alerta para os desafios existentes nas mesmas. O principal deles é especificar uma plataforma que seja reutilizável por uma gama considerável de aplicações, de

forma que os benefícios do uso desta plataforma possam efetivamente justificar os custos na tarefa de especificação e desenvolvimento.

3.3 ENGENHARIA GUIADA POR MODELOS

Outra abordagem interessante para tratar a complexidade no desenvolvimento de sistemas embarcados é o uso de técnicas de engenharia guiada por modelos (do inglês *Model-driven Engineering* — MDE). MDE propõe o desenvolvimento de sistemas computacionais a partir de um processo de transformação de uma especificação baseada em modelos para a sua implementação (SCHMIDT, 2006). Muitos dos trabalhos em MDE propõe uma estratégia baseada em modelos UML (*Unified Modeling Language*) (OMG, 2008). UML é uma linguagem padronizada que define um conjunto de diagramas para modelar vários aspectos estruturais e comportamentais de sistemas orientados a objetos.

Nesse cenário, o fluxo de projeto UPES/UPSoC (RICCOBENE et al., 2009) utiliza uma metodologia MDE baseada em PBD. Os autores propõem a modelagem da aplicação e da plataforma utilizando UML. Um modelo é proposto para mapear a aplicação na plataforma, gerando esqueletos para um modelo em nível de sistema. Neste nível, hardware e software são modelados utilizando, respectivamente, profiles UML para SystemC e *multithread*-C (com uma semântica *Pthread* (Open Group, 1995)), que são refinados posteriormente até a obtenção da implementação final.

O fluxo de projeto *Saturn* (MISCHKALLA; HE; MUELLER, 2010) também contribui nesse cenário. Os autores propuseram extensões de *SysML*, que, por sua vez, é uma extensão de UML para modelagem ESL (OMG, 2007). Os autores também desenvolveram uma ferramenta para geração de C++ para software e SystemC RTL para hardware. Apesar de, assim como no trabalho anterior, o sistema ser modelado utilizando uma única linguagem, software e hardware ainda são definidos e modelados explicitamente, obrigando um particionamento entre hardware e software já no nível de sistema. Além disso, no caso do *Saturn*, não está claro se a ferramenta proposta gera código apenas para a interface e integração entre os componentes, ou se o comportamento também é inferido a partir dos modelos em SysML.

Wehrmeister (2009) propõe o uso de técnicas de MDE aliada com conceitos de projeto orientado a aspectos e PBD para projetar componentes de sistemas de tempo-real embarcados e distribuídos. Através do uso de projeto orientado a aspectos, foi proposta a separação no tratamento de requisitos funcionais e não funcionais do sistema, promovendo uma melhor modularização e reuso dos artefatos produzidos. Adicionalmente, foi proposta uma ferramenta

para realizar a geração de código, suportando a transição automática das fases de especificação e implementação.

3.4 METODOLOGIAS BASEADAS NA SÍNTESE DE HARDWARE E SOFTWARE

Com o avanço das ferramentas de HLS, vários fluxos de projeto permitem a geração completa de hardware e software a partir da descrição em nível de sistema. Também graças aos avanços nas FPGAs, tais fluxos permitem a implantação de SoCs totalmente customizados para a aplicação-alvo, sem as limitações impostas por uma plataforma específica.

O trabalho de Ouadjaout e Houzet (2006) propõe um fluxo de projeto e uma ferramenta para a geração de sistemas a partir de modelos SystemC. A entrada para o fluxo é um modelo SystemC com precisão em nível de ciclos e uma biblioteca de definições de interface para componentes de hardware. A ferramenta proposta gera código C para o software e SytstemC sintetizável (OSCI, 2010) para o hardware, bem como a plataforma que integra todos os componentes. Os componentes de hardware são gerados como sendo aceleradores, dessa forma apenas comunicação entre componentes software→software e software→hardware é suportada.

Seguindo a mesma linha, a metodologia OSSS+R (SCHALLENBERG et al., 2009) propõe um conjunto de construções para suportar polimorfismo sintetizável e comunicação de alto nível em SystemC em RTL. Essa comunicação de alto nível gira em torno de objetos chamados *Shared Objects* (GRÜTTNER et al., 2010), que definem uma semântica de síntese para vários mecanismos distintos de comunicação (*e.g.* barramentos arbitrados, conexões ponto-a-ponto, etc.). No entanto, apesar de todo o sistema ser implementado de forma uniforme, o particionamento entre hardware e software ainda deve ser definido na fase inicial do projeto (MÜCK; FRÖHLICH, 2012). Além disso, a inclusão de construções não padronizadas na linguagem reduz a compatibilidade dos componentes do projeto com compiladores e ferramentas de síntese já existentes. Outra limitação, que também existe no trabalho de Ouadjaout, é o uso de um modelo temporizado[1] já no nível de sistema, o que aumenta a complexidade inicial do projeto, pois o projetista deve levar em conta as propriedades de sincronização da descrição algorítmica em relação ao protocolo de comunicação que será utilizado.

O SystemCoDesigner (KEINERT et al., 2009) já avança mais em relação aos trabalhos citados anteriormente no sentido de que o modelo de entrada

[1] em um modelo temporizado, as operações são totalmente ou parcialmente escalonadas entre diferentes ciclos de clock.

não precisa ser temporizado. O fluxo de projeto proposto é mostrado na figura 7. O SystemCoDesigner (KEINERT et al., 2009) é uma ferramenta que integra um fluxo baseado em HLS com DSE automática. A entrada da ferramenta é um modelo de fluxo de dados baseado em atores definido usando *SystemMOC* (FALK; HAUBELT; TEICH, 2006), uma extensão de SystemC. Algoritmos de DSE são aplicados no modelo inicial resultando em um conjunto de possíveis implementações para os atores. Uma vez que o particionamento final é definido, os atores do modelo podem ser convertidos para SystemC sintetizável, visando geração de hardware utilizando HLS, ou C++, visando geração de software. O SoC final é gerado automaticamente pela ferramenta, que se encarrega do escalonamento dos atores em software nos elementos de processamento disponíveis, e da geração da interconexão entre os atores em hardware. O fluxo de desenvolvimento proposto é bem completo e automatiza com sucesso grande parte do processo de desenvolvimento, contudo, como os próprios autores já estabeleceram, o SystemCoDesigner foca apenas em aplicações baseadas em fluxo de dados, como aplicações multimídia, e não são definidas diretivas em direção a uma aplicação mais geral da ferramenta.

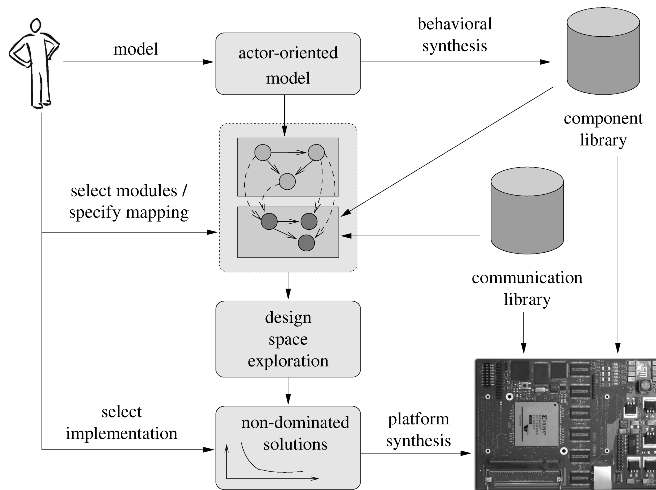

Figura 7: Fluxo de projeto do SystemCoDesigner (KEINERT et al., 2009)

Seguindo a mesma linha, o trabalho de Corre et al. (2012) estende o framework *Daedalus* (THOMPSON et al., 2007) com exploração automática do espaço de projeto. A partir de uma implementação paralela em C (seguindo o padrão *Pthread*), é gerado um modelo de redes de Kahn (do inglês *Kahn Process Network* — KPN) e profiles de desempenho da aplicação. Estes, jun-

tamente com a especificação da plataforma e um conjunto de requisitos, são usados como entrada para um algoritmo de DSE. Este trabalho foca uma arquitetura específica para FPGAs, denominada *Heterogeneous Multiprocessor System-on-Chip* (MPSoC) (H-MPSoC), composta por múltiplas unidades de processamento com memória local e comunicação utilizando troca de mensagens. Cada unidade de processamento pode ser auxiliada por aceleradores dedicados, gerados automaticamente a partir do modelo da aplicação utilizando HLS.

O *System-on-Chip Environment* (SCE) (DÖMER et al., 2008) oferece capacidades semelhantes aos trabalhos descritos anteriormente. O SCE é uma evolução de um projeto feito para a Agência Espacial Japonesa (JAXA) e é baseado em uma metodologia *Specify-explore-refine* (SER) (GERSTLAUER et al., 2008). Uma visão geral do seu fluxo de projeto é mostrado na figura 8. O SCE consiste em um grupo de ferramentas para simulação, verificação e implementação de SoCs. O ambiente de desenvolvimento utiliza como entrada modelos em SpecC e oferece um fluxo de desenvolvimento baseado no refinamento sucessivo de modelos. Guiado pelo projetista, o SCE gera automaticamente modelos no nível de transação (do inglês *Transaction-level Model* — TLM) (CAI; GAJSKI, 2003) descrevendo possíveis arquiteturas para o sistema, que são posteriormente refinados pela ferramenta até que seja obtida uma implementação com precisão em nível de ciclos.

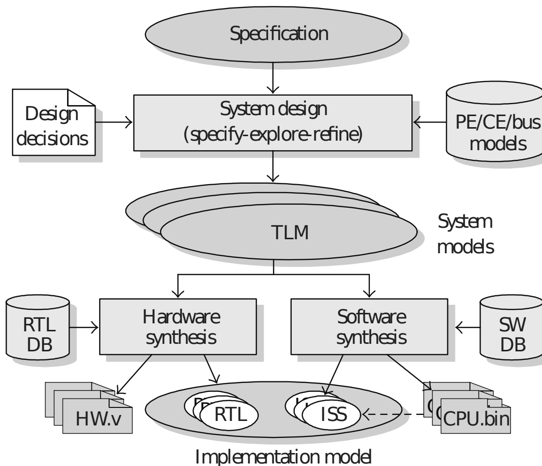

Figura 8: Fluxo de projeto do SCE (DÖMER et al., 2008)

3.5 VERIFICAÇÃO NO NÍVEL DE SISTEMA

Questões relacionadas a verificação da corretude de sistemas embarcados estão fora do escopo específico deste trabalho, no entanto, a possibilidade de gerar uma implementação diretamente a partir de uma descrição no nível de sistema tem criado uma nova vertente de pesquisa, que é a verificação formal de componentes descritos neste nível.

Como ferramentas de HLS permitem a descrição de componentes em hardware usando linguagens já consolidadas no domínio de software, técnicas de *model checking* de software podem ser aplicadas para verificar componentes que serão implementados em ambos os domínios. Em *model checking*, um programa é convertido para uma equação booleana que deve ser sempre verdadeira para um conjunto de propriedades definidas pelo desenvolvedor. O trabalho de Ludwich e Fröhlich (2012), por exemplo, propõe a definição de *contratos* compostos por invariantes de classe e pré- e pós-condições para métodos da forma como definido em *programação por contratos* (MEYER, 2003). O contrato é especificado em C++, a mesma linguagem considerada para a implementação dos componentes no nível de sistema. Tanto a implementação quanto a especificação são traduzidos para a representação interna da ferramenta *C Bounded Model Checker* (CBMC) (KRöNING, 2012) e, então, verificadas formalmente.

3.6 INTERFACEAMENTO ENTRE HARDWARE E SOFTWARE

Outros trabalhos focam basicamente no interfaceamento entre hardware e software e na comunicação transparente entre os domínios, sem o comprometimento de gerar e integrar todos os componentes do sistema a partir de descrições em alto nível.

O trabalho de Lange e Koch (2010) e Gantel et al. (2011), o projeto *HThread* (ANDERSON et al., 2007), o *ReconOS* (LÜBBERS; PLATZNER, 2008) e o sistema operacional BORPH (SO; BRODERSEN, 2008) seguem todos uma abordagem bastante semelhante para fornecer uma interface de comunicação unificada para ambos os domínios. Estes trabalhos são completamente focados em arquiteturas reconfiguráveis, como FPGAs. A ideia principal é fazer com que uma tarefa executada em hardware tenha a mesma semântica que uma thread em software, definindo uma interface de *system call* para os componentes em hardware. O trabalho de Lange e Koch (2010), por exemplo, propõe um conjunto de mecanismos para comunicação rápida e coerente entre hardware e software, o que inclui: tratamento prioritário das interrupções dos componentes em hardware; e um mecanismo de acesso direto

a memória integrado com a *unidade de gerenciamento de memória* (do inglês *Memory Management Unit* — MMU) do processador.

O trabalho de Gantel et al. (2011) avança um pouco mais em direção à modelagem em nível de sistema considerando a disponibilidade de reconfiguração dinâmica[2]. O trabalho propõe um fluxo de projeto no qual as threads a serem executas em hardware e software são extraídas de um modelo SDF. A principal contribuição do trabalho é a integração entre o escalonador de threads do SO e um mecanismo de escalonamento em hardware responsável por escalonar a execução de thread em partições reconfiguráveis na FPGA.

O trabalho de Rincón et al. (2009) segue uma abordagem distinta e é baseado nos mesmos conceitos definidos em plataformas de objetos distribuídos, tais como CORBA (OMG, 1997) e Java RMI (Oracle, 2010). Os autores implementaram uma estrutura baseada em *Network-on-Chip* (NoC) para suportar uma comunicação de alto desempenho e baixo consumo energético entre componentes em hardware e software. Foi proposta a implementação em hardware de um esquema de comunicação cliente/servidor no qual componentes requisitando uma operação são mapeados em tempo de execução para os componentes capazes de oferecer a operação requisitada. A implementação em hardware e software do processo de *marshalling*[3] é gerada automaticamente por uma ferramenta que analisa a interface dos componentes e gera os adaptadores adequados.

3.7 SUMÁRIO E DISCUSSÃO

Uma visão geral dos fluxos de projeto de sistemas embarcados pode ser vista nas figuras 9 e 10. A figura 9 representa os fluxos de projeto descritos nas seções 3.2 e 3.3. Estes foram fruto de pesquisa no passado e estão atualmente consolidados e sendo utilizados em larga escala para o projeto de sistemas embarcados. Nessas metodologias, os modelos no nível de sistema servem basicamente como uma especificação inicial e como um ponto de partida para a exploração do espaço de projeto. Esse processo leva em conta uma biblioteca de componentes pré-existentes e resulta na arquitetura do sistema. A partir da arquitetura, os desenvolvedores seguem fluxos distintos para o desenvolvimento do hardware e software remanescente.

A figura 10 sumariza a metodologia de projeto do futuro, que é o alvo dos fluxos de projeto e ferramentas do estado-da-arte sendo desenvolvidos

[2]a capacidade das FPGAs modernas de reconfigurar determinadas partes do chip em tempo de execução e sem afetar o restante do circuito.

[3]termo que, no contexto de computação distribuída, indica o processo de serializar um objeto, ou os parâmetros de uma chamada ao objeto, em um formato de dado adequado para transmissão

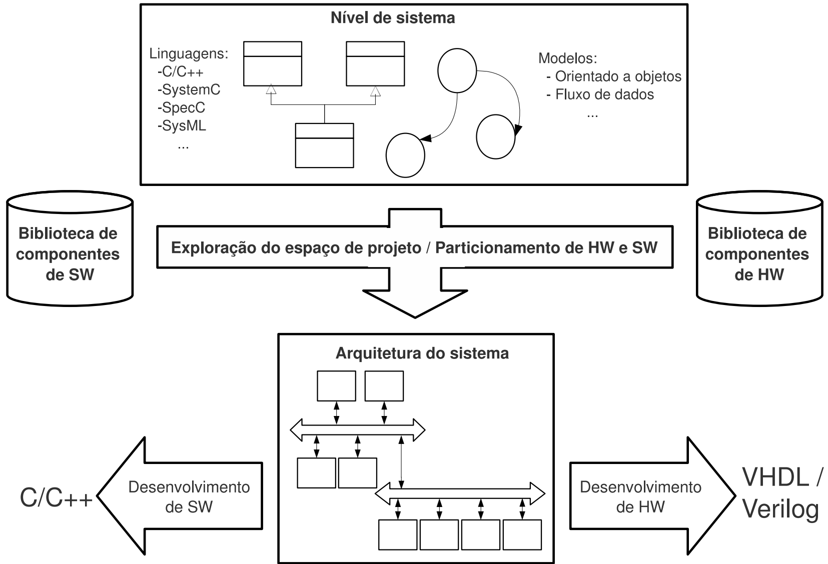

Figura 9: Fluxo de projeto de hoje. Desenvolvido no passado.

atualmente. Nessa metodologia, a implementação final seria gerada diretamente a partir da descrição no nível de sistema de forma, idealmente, automática.

De uma maneira geral, nos últimos anos, houve uma evolução significativa em direção a esse fluxo de projeto totalmente automatizado. Muitas das limitações das metodologias baseadas em IPs e plataformas fixas estão sendo superadas com a crescente introdução de ferramentas para HLS e dispositivos reconfiguráveis. No entanto, a maioria dos trabalhos já desenvolvidos focam principalmente nos processos de HLS e em ferramentas de integração para o fluxo de projeto como um todo, mas sem definir uma metodologia clara para projetar os componentes em si no nível de sistema.

Manter toda a “inteligência” do processo de projeto apenas nas ferramentas leva a limitações já discutidas em Gerstlauer et al. (2009). Neste trabalho os autores fazem uma comparação entre vários fluxos de projeto e ferramentas do estado-da-arte. Nenhum dos fluxos de projeto analisados apresenta uma solução completa, e a integração das ferramentas já desenvolvidas é complexa devido às diferenças significativas que existem entre os modelos de entrada e os modelos intermediários gerados pelos processos de refinamento de cada ferramenta. Nesse cenário, definir modelos com semânticas mais fortes pode potencialmente diminuir a complexidade de integração, além de possibilitar implementações no nível de sistema suscetíveis a uma gama maior de fluxos

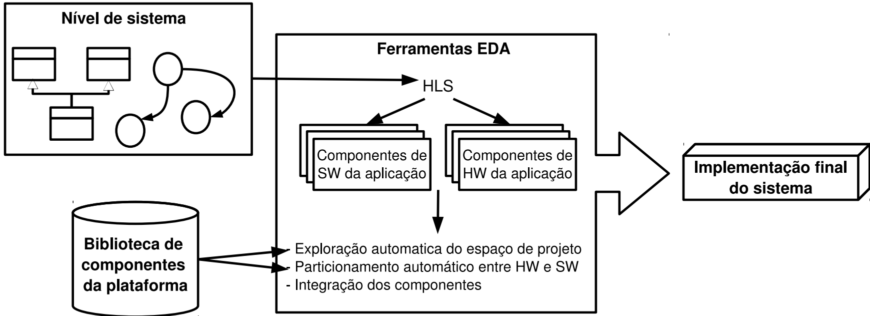

Figura 10: Fluxo de projeto do futuro. Sendo desenvolvido atualmente.

de projeto e ferramentas.

A tabela 1 compara os trabalhos da seção 3.4 com a proposta desta dissertação e mostra como a mesma avança nesse sentido. Na abordagem descrita neste trabalho, tanto a descrição unificada dos componentes, quanto os programas de aspecto que adaptam os mesmo para hardware ou software são definidos utilizando a linguagem C++. Além disso, os mecanismos de aplicação de aspectos também são baseados em funcionalidades intrínsecas da própria linguagem. Dessa forma, a extração de implementações em hardware ou software a partir de uma implementação em nível de sistema é direta e se dá através de transformações no nível da linguagem, utilizando semânticas definidas pelo projetista que são independentes de fluxos de projeto e ferramentas específicas. O foco em mecanismos na linguagem também permite a aplicação de técnicas de verificação formal, como a descrita na seção 3.5, para verificar tanto a corretude funcional dos componentes quanto a corretude dos mecanismos de obtenção de hardware e software.

Tabela 1: Comparação entre os trabalhos que empregam síntese a partir do nível de sistema

	Modelo inicial			Refinamento	Modelo intermediário		Automatização do modelo final	
	Linguagem	Modelo	Temporizado		Hardware	Software	DSE	Geração da plataforma
Quadjaout e Houzet	SystemC	Processos	Sim	Requer suporte de ferramentas EDA específicas	SystemC	C++	Automática	Manual
OSSS+R	SystemC (com extenções próprias)	Processos	Sim	Requer suporte de ferramentas EDA específicas	SystemC (com extenções próprias)	C++	Manual	Automática
Corre et al.	C (Pthread)	Processos	Não	Requer suporte de ferramentas EDA específicas	C++	C	Automática	Semi-automática
SystemCoDesigner	SystemMoC	Fluxo de dados	Não	Requer suporte de ferramentas EDA específicas	SystemC	C	Automática	Automática
SCE	SpecC	Processos	Não	Requer suporte de ferramentas EDA específicas	SystemC (com extenções próprias)	C	Manual	Semi-automática
Este trabalho	C++	Orientado a objetos	Não	Automático. Não requer ferramentas específicas	C++	C++	Manual	Manual

4 PROJETO DE SISTEMAS EMBARCADOS GUIADO PELA APLICAÇÃO

A *metodologia de projeto de sistemas embarcados guiado pela aplicação* (do inglês *Application-driven Embedded System Design* — ADESD) (FRÖHLICH, 2001; POLPETA; FRÖHLICH, 2005) é uma *metodologia de engenharia de domínio* que se baseia em uma estratégia bem definida de decomposição do domínio através do *projeto baseado em famílias* (do inglês *Family-based Design* — FBD) (PARNAS, 1976) e OOP, focando também na aplicação de técnicas de *programação orientada a aspectos* (do inglês *Aspect-oriented Programming* — AOP) (KICZALES et al., 1997) já nos estágios iniciais do projeto.

A Engenharia de Domínio suporta o desenvolvimento de novos sistemas de modo que as soluções específicas e dependentes de cada aplicação possam ser desenvolvidas a partir de um repositório contendo artefatos independentes das aplicações futuras. O desenvolvimento da solução específica ou dependente da aplicação deve ser sistematizado, de forma que os novos sistemas sejam elaborados com através da composição de artefatos do repositório (FRAKES; KANG, 2005).

Uma decomposição do domínio do problema guiada pela aplicação pode ser obtida, a princípio, seguindo as recomendações da decomposição orientada a objetos (BOOCH et al., 2007). A decomposição orientada a objetos utiliza uma análise de variabilidade pra identificar como uma entidade pode ser criada a partir de outra, com o objetivo de agrupar objetos com um comportamento semelhante. Além de levar ao problema da "classe-base frágil" (MIKHAJLOV; SEKERINSKI, 1998), esta política assume que especializações de uma abstração (*i.e.* as subclasses) só são aplicadas na presença da sua versão mais genérica (*i.e.* a superclasse). ADESD evita essa restrição promovendo a aplicação da análise de variabilidade segundo o FBD (PARNAS, 1976). Este método produz abstrações independentes organizadas como membros de uma família. Por exemplo, no contexto de redes de computadores, ao invés de modelar comutação de circuitos como uma especialização de comutação de pacotes, o que poderia utilizar de forma errônea os recursos de uma rede que opera no modo oposto, o projetista poderia modelar ambos como membros autônomos de uma família de protocolos de comunicação.

Outra importante diferença entre ADESD e a decomposição orientada a objetos tradicional, refere-se às dependências do cenário de aplicação dos componentes. A decomposição tradicional não enfatiza a diferenciação entre o que faz parte do comportamento primário do componente do que se refere ao *cenário de aplicação* do componente. Componentes que incorporam tais

dependências têm uma chance menor de serem reusados em diferentes cenários. ADESD reduz este tipo de dependência através da aplicação do conceito chave de AOP (KICZALES et al., 1997) ao processo de decomposição. As dependências do cenário são encapsuladas em artefatos especiais chamados de *aspectos*, e uma semântica de incorporação de aspectos é definida separadamente para descrever como os mesmos são aplicados aos componentes. Seguindo esse processo, o projetista pode distinguir claramente as variações que definem novos componentes daquelas que resultam em aspectos do cenário. Por exemplo, ao invés de modelar uma família de mecanismos de comunicação que suporta múltiplas *threads* e acesso concorrente, o projetista pode modelar concorrência como um aspecto de cenário que, quando ativado, bloquearia o mecanismo de comunicação (ou algumas de suas operações) quando em uma seção crítica.

Uma visão geral do processo de decomposição de ADESD pode ser visto na figura 11. Em suma, ADESD guia um processo de engenharia de domínio de forma a criar famílias de componentes nas quais as dependências relativas ao cenário de execução são fatoradas como aspectos e *propriedades configuráveis* (do inglês *configurable features*). As propriedades configuráveis representam variações sutis em um componente e podem ser ativadas para mudar ligeiramente o seu comportamento ou estrutura. Um exemplo é o tamanho do segmento de pilha a ser alocado para cada *thread* ou processo gerenciado pelo sistema.

Uma última questão relevante a ser considerada é como componentes interagem entre si. Em ADESD, as relações externas entre componentes são capturadas em um *framework*. Este framework estabelece a relação entre os componentes e os aspectos em um determinado cenário de execução, definindo um mecanismo que aplica os aspectos aos componentes de forma adequada.

A semântica de aplicação de aspectos em ADESD é definida por artefatos chamados de *adaptadores de cenário* (do inglês *scenario adapter*) (FRÖHLICH; SCHRÖDER-PREIKSCHAT, 2000). Os adaptadores de cenário foram desenvolvidos a partir da ideia de que componentes podem *entrar* e *sair* de um determinado cenário de execução, permitindo que ações sejam executadas nestes pontos. Tais ações são executadas antes e após cada uma das operações definidas pelos componentes de forma a estabelecer as condições requeridas pelo cenário. A figura 12 mostra a estrutura geral de um adaptador de cenário. Um artefato `Scenario` representa o cenário de execução e estabelece as operações de entrada/saída do cenário agregando todos os aspectos que o caracterizam. O adaptador de cenário então aplica o cenário ao componente-alvo. Como será mostrado a partir da seção 5.3.1, esta estrutura é definida na forma de um *framework metaprogramado* que é totalmente implementado em C++ usando OOP e técnicas de metaprogramação estática. Ao contrário do que é visto nas abordagens tradicionais de aplicação de aspectos, como o

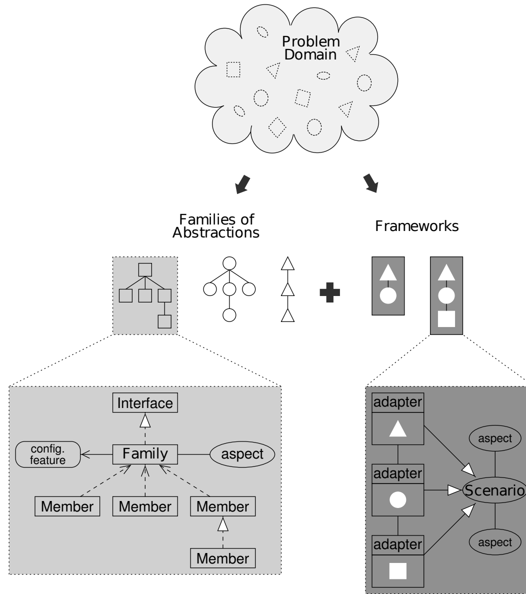

Figura 11: Visão geral da decomposição de domínio em ADESD (FRÖHLICH, 2001)

AspectC++ (SPINCZYK; GAL; SCHRÖDER-PREIKSCHAT, 2002), adaptadores de cenário não requerem suporte de ferramentas externas e podem ser usados como entrada em qualquer compilador compatível com C++ padrão.

4.1 EXEMPLO DE DECOMPOSIÇÃO DE DOMÍNIO: ESCALONAMENTO DE TAREFAS

Para ilustrar o processo de decomposição de domínio de ADESD, será mostrada a seguir a modelagem do escalonador de tarefas do sistema operacional *Embedded Parallel Operating System* (EPOS). O EPOS (The EPOS Project, 2011) é o principal estudo de caso utilizado para validar os artefatos propostos por ADESD. A escolha do seu escalonador de tarefas como exemplo é motivada pelo seu comportamento complexo. O escalonamento de tarefas envolve tanto operações síncronas (*e.g.* criar uma nova thread) quanto assíncronas (*e.g.* pré-emptar a execução de outro componente), e o seu comportamento afeta diretamente a eficiência de todo sistema.

A figura 13 mostra as principais entidades relacionas ao escalonamento

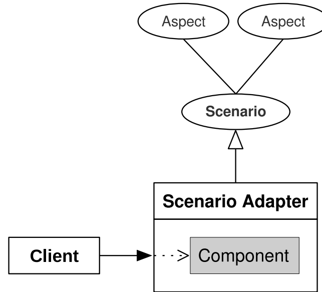

Figura 12: Incorporação de aspectos usando um adaptador de cenário

de tarefas e como elas foram organizadas em famílias de componentes seguindo o processo de decomposição de domínio de ADESD.

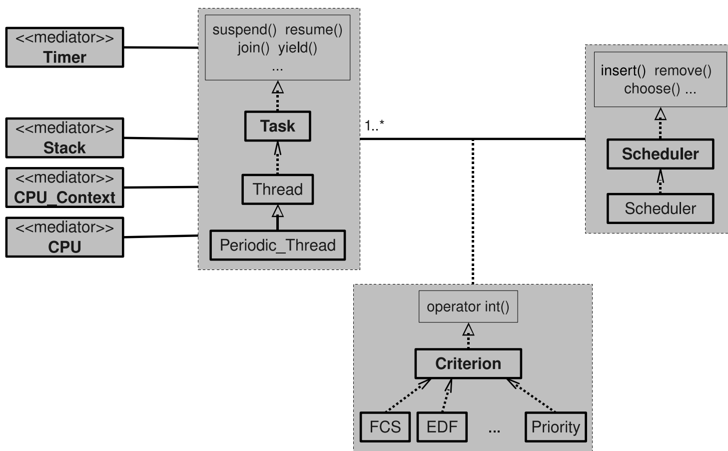

Figura 13: Famílias de componentes responsáveis pelo escalonamento de tarefas no EPOS. As famílias `Task`, `Scheduler` e `Criterion` são mostradas em mais detalhes.

Tarefas são representadas por componentes da família `Task`. Esses componentes definem o fluxo de execução de uma tarefa, implementando as funcionalidades tradicionais para esse tipo de abstração (*e.g.* tratamento de interrupções de tempo e troca de contexto). Esta família possui dois membros: `Thread` e `Periodic_Thread`. O segundo é uma especialização do primeiro e implementa tarefas periódicas, um tipo de tarefa comum em sistemas de tempo

real. Questões que são diretamente dependentes da arquitetura de hardware do sistema (*e.g.* a ISA da CPU e os periféricos) não são tratadas pela família `Task`, mas por outro tipo de família de componentes chamados de *mediadores de hardware* (POLPETA; FRÖHLICH, 2004). Os mediadores de hardware podem ser vistos como o equivalente em ADESD aos tradicionais drivers de dispositivos em sistemas Unix, mas não são construídos como uma HAL convencional. Os mediadores de hardware implementam uma interface entre os componentes de software e os dispositivos de hardware através de técnicas de metaprogramação estática. Deste modo, o código do mediador é dissolvido no componente em tempo de compilação, agregando ao componente apenas as funcionalidades do mediador que foram efetivamente utilizadas. Na figura 13, os artefatos marcados com «`mediator`» indicam famílias de mediadores de hardware que abstraem detalhes dependentes da arquitetura, como: salvamento e restauração de contexto, criação de uma nova pilha, configuração de interrupções de temporização, entre outras.

As famílias `Scheduler` e `Criterion` definem a estrutura que realiza o escalonamento das tarefas. O projeto e implementação tradicional de algoritmos de escalonamento são normalmente feitos através de uma hierarquia de classes especializadas a partir de uma classe abstrata que representa o escalonador. Tais classes podem ainda ser posteriormente especializadas para introduzir novas políticas de escalonamento ao sistema. De forma a reduzir o esforço na manutenção de código (normalmente presente nesse tipo de hierarquia de classes especializadas), assim como promover o reuso de componentes, o sistema de escalonamento do EPOS desacopla a política de escalonamento (família `Criterion`) da implementação do mecanismo de escalonamento (família `Scheduler`) e também do recurso que ele representa. Um componente da família `Scheduler` implementa apenas os mecanismos que realizam o ordenamento dos recursos, baseado na política selecionada. Nesse sentido, uma implementação em hardware do escalonador pode ser facilmente concebida pois um mesmo componente em hardware pode realizar diferentes políticas sem que nenhuma reconfiguração de hardware seja necessária. O confinamento da política em componentes da família `Criterion` é feita isolando o algoritmo de comparação dos elementos da fila de escalonamento, de forma análoga ao que é feito nas estruturas de dados definidas pela *C++ Standard Template Library* (STL) (STEPANOV; LEE, 1995).

4.1.1 Implementação do escalonador

Para obter um componente altamente reusável sem penalizar o desempenho, a implementação do escalonador em si faz um uso extensivo de técnicas

de metaprogramação estática (CZARNECKI; EISENECKER, 2000). A figura 14 mostra em mais detalhes a definição do componente Scheduler e como ele se relaciona com uma thread. O escalonador é definido através da classe Scheduler<T>, que possui como parâmetro de template o tipo do recurso que será escalonado. A associação entre a classe Thread e a classe Scheduler<T> na figura 14 possui o rótulo <Thread>, o que significa que a classe Thread está associada com uma instancia da classe Scheduler que usa o tipo Thread como valor para o template T.

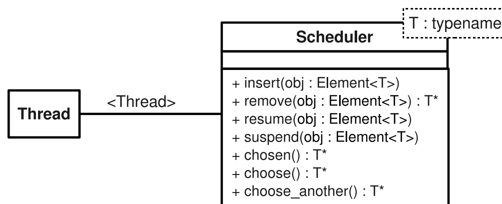

Figura 14: Definição do componente Scheduler

A figura 15 estende a figura 14 com mais detalhes de implementação do escalonador. A implementação segue uma abordagem baseada em listas duplamente encadeadas para armazenar e ordenar os recursos a serem escalonados. No entanto um ponto importante a ser destacado nessa implementação é o fato de que os aspectos de encadeamento e armazenamento são fatorados e implementados separadamente, sendo apenas integrados em tempo de compilação através de metaprogramação estática. Ou seja, a hierarquia de classes List◁—Ordered_List◁—Scheduling_List não é responsável pela alocação dos nodos internos e pelos objetos que ela armazena. Os algoritmos de manipulação e ordenação de listas são implementados de forma a lidar com referências para esses objetos. O principal objetivo dessa abordagem é evitar a alocação/desalocação excessiva de nodos quando o mesmo objeto é inserido e removido da lista várias vezes. Por exemplo, a thread pré-aloca um nodo no momento da sua criação, e este mesmo elemento é repassado ao escalonador sempre que a thread for inserida ou removida da fila de escalonamento (objetos Element<T> na assinatura dos métodos da classe Scheduler). Além disso, como será mostrado nas seções seguintes, essa abordagem de implementação também permite a fatoração de alocação de recursos como um aspecto que pode ser implementado de forma distinta em hardware e software.

Como pode ser visto na figura 15 um nodo da lista é definido pelo template de classe Scheduling_Element<T,Rank>. O segundo parâmetro, Rank, define o critério de ordenação do elemento na lista e mapeia diretamente para um dos critérios definidos dentro da família Criterion,

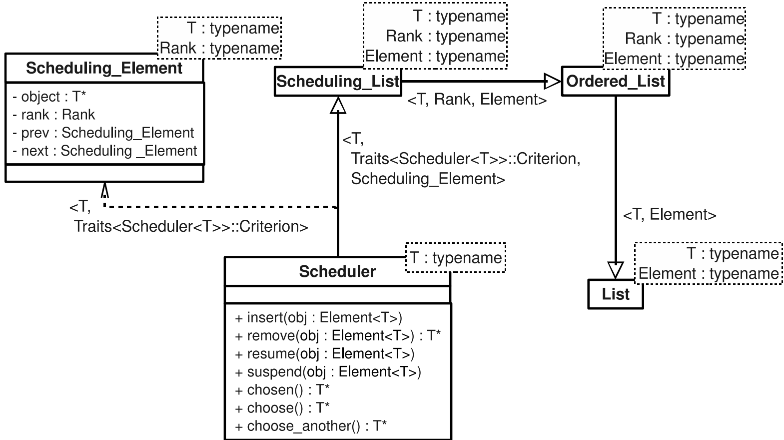

Figura 15: Implementação do componente `Scheduler`

tal como mostrado anteriormente. `Rank` também é um parâmetro da classe `Scheduling_List`. `Scheduler`, por sua vez, herda de `Scheduling_List` definindo `Scheduling_Element` como o tipo do nodo e `Traits<Scheduler<T> >::Criterion` como critério de ordenação.

O critério que será utilizado depende de configurações estáticas definidas em templates especiais chamados de *Traits* (atributos ou características, em português) (MYERS, 1995). Na figura 15, quando o escalonador é usado com a classe `Thread`, `Traits<Scheduler<T> >` mapeia para `Traits<Scheduler<Thread> >`, cuja definição é mostrada a seguir:

```
1  template <> struct Traits<Scheduler<Thread>> {
2      typedef Scheduling_Criteria:: Priority   Criterion ;
3  };
```

Essa é uma maneira conveniente de definir a associação entre tipos, valores e funções. Para usar o escalonador na gerência de outro tipo de recurso, basta adicionar uma nova especialização do template `Traits` contento a definição de qual política de escalonamento será usada junto ao novo recurso.

4.2 ADESD APLICADA AO PROJETO DE HARDWARE E SOFTWARE

ADESD foi inicialmente proposta no escopo do desenvolvimento de software e aplicada ao domínio de sistemas operacionais dedicados (FRÖHLICH, 2001). No entanto, logo se percebeu que os mesmos conceitos também poderiam ser eficientemente aplicados no projeto de sistemas embarcados como

um todo, englobando tanto aspectos de hardware quanto de software. Isso foi inicialmente explorado por Polpeta e Fröhlich (2004), que propuseram uma estratégia para a seleção, configuração e adaptação de componentes de acordo com desenvolvimento baseado em IPs. Esta abordagem especifica uma micro-arquitetura baseada em componentes de software e componentes de hardware sintetizáveis em dispositivos de lógica programável. Uma ferramenta identifica os mediadores de hardware (ver seção 4.1) instanciados pela aplicação e utiliza essa informação para guiar a geração do hardware em si.

O trabalho de Polpeta e Fröhlich (2004) foi posteriormente estendido com o conceito de *Componentes Híbridos* (MARCONDES; FRÖHLICH, 2009), na qual foi desenvolvida uma arquitetura comum para a implementação e comunicação entre componentes nos domínios de hardware e/ou software. A estratégia proposta permite que cada componente possa ser visto pelo projetista como um componente livre de sua realização em software ou hardware, permitindo este possa migrar entre ambos os domínios durante qualquer etapa do projeto, sem que com isso seja necessário realizar uma reengenharia do sistema, criando assim uma arquitetura mais flexível para o desenvolvimento de SoCs. A partir dessa arquitetura também foi desenvolvida *HyRA* (MÜCK; FRÖHLICH, 2011a), uma arquitetura flexível para o desenvolvimento de *rádios definidos por software*[1], que aplica o conceito de componentes híbridos no domínio de processamento digital de sinais.

Os trabalhos anteriores definem uma metodologia clara sobre como novos componentes devem ser projetados para serem reusados em uma gama maior de aplicações e estabelecem mecanismos que permitem a prorrogação do particionamento entre hardware e software para as fases finais do desenvolvimento do sistema. Contudo, há ainda uma disparidade entre os processos de desenvolvimento de software e hardware. A aplicação dos conceitos fundamentais de ADESD (*i.e.* organização dos componentes em famílias e isolamento das dependências do cenário em aspectos) é barrada no nível dos mediadores de hardware e não é estendida ao desenvolvimento do hardware em si, que segue o fluxo de desenvolvimento RTL utilizando uma linguagem como VHDL e Verilog.

Uma proposta de aplicar os conceitos de ADESD diretamente em hardware foi feita, em um primeiro momento, nas fases iniciais do desenvolvimento do presente trabalho. Aproveitando-se das funcionalidades que SystemC herdou do C++, foi proposta uma maneira de utilizar adaptadores de cenário como um mecanismo para melhorar a reusabilidade de projetos RTL utilizando-se SystemC (MÜCK et al., 2012). Apesar de aproximar o desenvolvimento de software do desenvolvimento de hardware, nessa abordagem, no entanto, o

[1] do inglês *software-defined radio*, são sistemas nos quais partes da camada física de um protocolo de comunicação (*e.g.* modulação e demodulação) podem ser implementadas em software

hardware ainda é desenvolvido em um nível mais baixo.

Segundo o que foi mostrado no Capítulo 3, o desenvolvimento de sistemas embarcados está sendo deslocado para o nível ESL, no qual as funcionalidades de um componente são descritas de forma que a sua implementação em hardware e software possa ser inferida da forma mais automatizada possível. As ferramentas EDA recentes avançam de forma significativa nesse sentido, viabilizando a descrição de hardware em linguagens como C e C++. O foco dessas ferramentas, no entanto, é apenas síntese de hardware. Elas não introduzem uma metodologia clara para o desenvolvimento de componentes que possam ser implementados tanto em hardware quanto em software. Nesse sentido, o presente trabalho visa demonstrar como os artefatos propostos por ADESD podem ser utilizados para implementações em C++ que possam ser utilizadas tanto em software quanto como entrada para um processo de HLS.

5 PROJETO UNIFICADO DE HARDWARE E SOFTWARE

Este capítulo apresenta as técnicas e mecanismos que possibilitam a implementação de componentes em software e hardware a partir de uma única descrição na linguagem C++. Como será visto a seguir, o “projeto unificado” de hardware e software se dá por meio de um cuidadoso processo de engenharia de domínio que guia a implementação de componentes de forma que características especificas de hardware e software possam ser encapsuladas separadamente. Essas características são, então, aplicadas à implementação base, ou “unificada”, do componente apenas nas fases finais do fluxo de projeto.

As ferramentas EDA impõem certas limitações no desenvolvimento de hardware usando C++, de forma que um componente sintetizável deve ser implementado usando o subconjunto da linguagem suportado pela ferramenta e seguindo certas diretrizes para que o hardware possa ser inferido corretamente. Em um primeiro momento, as questões relacionadas ao processo de HLS serão apresentadas por meio de um estudo de caso: uma versão sintetizável do escalonador mostrado no Capítulo 4 (FLOR; MÜCK; FRÖHLICH, 2011). Dessa forma, ficarão mais claros os pontos que efetivamente diferenciam uma implementação em C++ visando software de uma visando hardware. Em seguida, nas seções 5.2 e 5.3, será mostrado como essas particularidades de cada domínio podem ser isoladas utilizando os mecanismos de encapsulamento e aplicação de aspectos propostos por ADESD.

5.1 UM ESCALONADOR DE TAREFAS SINTETIZÁVEL

Várias ferramentas trabalham com C++ padrão ANSI[1] e permitem a utilização de construções avançadas da linguagem, tais como ponteiros, classes e *structs*, arrays multidimensionais e metaprogramação via templates. Isso permitiu que grande parte do código que implementa a estrutura mostrada na figura 15 pudesse ser reutilizada sem um grande número de alterações. As alterações realizadas foram ditadas pelas restrições impostas pelas ferramentas ESL que estavam disponíveis durante o desenvolvimento deste trabalho: CatapultC (Calypto Design Systems, 2011) e AutoESL (Xilinx, 2012). A seguir é detalhado cada um dos tipos de alterações realizados sobre o código C++ original do escalonador.

[1] *e.g.* CatapultC (Calypto Design Systems, 2011), Synphony (Synopsys, 2011), C-to-Silicon (Cadence Design Systems, 2011) e AutoESL (Xilinx, 2012)

5.1.1 Semântica dos ponteiros

Basicamente, ponteiros não têm um significado intrínseco em hardware. Durante o fluxo de síntese de alto nível, estruturas do tipo *array* presentes no código são mapeadas para bancos de registradores ou memórias, e as variáveis do tipo ponteiro são mapeadas para índices dos arrays aos quais se referem. Dessa forma, em um código C++ sintetizável, ponteiros nulos ou inválidos não podem existir. A ferramenta deve ser capaz de determinar estaticamente todas as estruturas para as quais um determinado ponteiro pode apontar.

No entanto, é um estilo de codificação comum utilizar-se ponteiros nulos para reportar falhas. Na implementação original do escalonador, esse estilo é utilizado pela implementação das listas mostradas na figura 15. Para contornar essa questão, o código foi modificado para utilizar um tipo de dado conhecido como *option type*[2]. Esse tipo de dado pode ser visto como um contêiner para um valor genérico, e agrega uma variável de estado que indica se o valor é válido ou inválido. Um *option type* foi implementado em C++ na classe `SafePointer<T>`, que possui os seguintes construtores:

```
1  template<typename T> class SafePointer {
2  ...
3          SafePointer(): _exists(false) {}
4          SafePointer(T thing, bool exists = true): _exists(exists), _thing(thing) {}
5  ...
6  };
```

Um construtor representa a ausência de um valor no contêiner, enquanto o outro representa a sua presença. Através da substituição de ponteiros simples (`T*`) no código do escalonador por valores `SafePointer<T*>`, foi possível evitar complemente a passagem de ponteiros inválidos. Por exemplo, o trecho de código abaixo mostra a implementação do método de busca da classe `List<T>`, na versão original e na versão sintetizável:

```
1  //código original
2  Element * search(const Object_Type * obj) {
3      Element * e = _head;
4      for (; e && (e->object() != obj); e = e->next());
5      return e;
6  }
7
8  //código modificado
9  SafePointer<Element*> search(const Object_Type &obj) {
10         Element * e = _head;
11     for (; e && (e->object() != obj); e = e->next());
12     return SafePointer<Element*>(e, e != 0);
13 }
```

[2]um tipo muito comum em linguagens de programação funcional. Mais informações podem ser encontradas em Odersky, Spoon e Venners (2008)

5.1.2 Gerenciamento de alocação

Uma das limitações da implementação de algoritmos em hardware, é a impossibilidade de utilizar alocação dinâmica de memória. A quantidade de memória a ser utilizada pelo algoritmo precisa ser um valor conhecido em tempo de síntese. Alocar memória dinamicamente em uma FPGA, por exemplo, significaria reconfigurar suas partes dinamicamente, uma tarefa atualmente não suportada pelas ferramentas de síntese. Por outro lado, todas as estruturas de dados do escalonador são independentes da forma com que seus elementos estão alocados. Como explicado na seção 4.1, a classe `Thread` é responsável pela alocação dos elementos para a lista encadeada e repassa um ponteiro desse elemento para o escalonador.

Fazer a alocação dos elementos na classe `Thread`, contudo, acaba impondo uma dificuldade adicional quando o escalonador é considerado isoladamente em hardware. Como mencionando anteriormente, ponteiros não possuem um significado em hardware, ou seja, um ponteiro criado no escopo da classe `Thread` não fará sentido no escalonador em hardware pois a ferramenta de síntese não tem acesso direto ao espaço de endereçamento do sistema operacional. Dessa forma, um alocador foi implementado separadamente e associado ao escalonador de forma a manter a memória que contém os elementos das filas no mesmo "escopo de síntese" do escalonador. O alocador implementado é mostrado na parte mais interna da figura 16 (`Allcation Management`).

O alocador funciona como um invólucro para uma instância do escalonador, e exporta publicamente uma interface bastante semelhante à do objeto envolvido. Duas grandes diferenças podem ser notadas: as operações possuem semântica de passagem por valor, e nas inserções e remoções são feitas alocações e liberações de espaço no armazenamento subjacente.

O armazenamento para os elementos das filas de escalonamento é realizado em uma estrutura de dados do tipo array com um tamanho fixo. A operação de inserção retorna um índice que indica a posição que o elemento das listas foi alocado. Esse índice é usado nas chamadas subsequentes do escalonador para identificar o objeto sendo escalonado.

5.1.3 Definição da interface

Outra restrição da ferramenta utilizada se refere à forma como é feita a inferência das portas de entrada e saída do bloco de hardware sendo modelado. Em descrições C++, o projetista deve designar uma única função como a

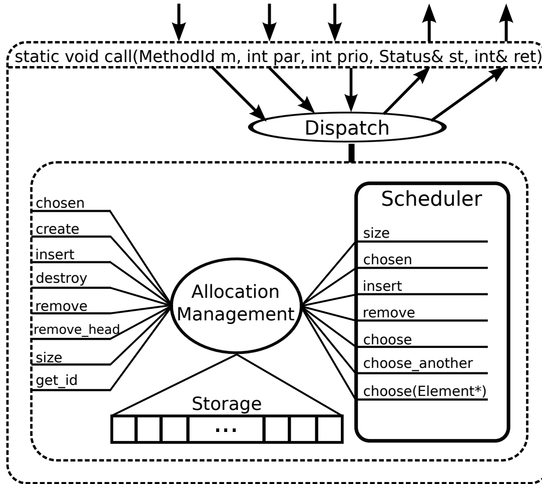

Figura 16: Adaptação do escalonador para hardware (FLOR; MÜCK; FRÖHLICH, 2011)

função "raiz"[3] do projeto. A ferramenta de síntese irá, então, inferir, a partir dos parâmetros presentes na assinatura e da forma como eles são utilizados, as portas de entrada e saída do bloco de hardware sendo modelado, sendo que os tipos dos parâmetros definirão o tamanho das portas.

O escalonador, no entanto, é implementado seguindo uma abordagem orientada a objetos, e sua interface possui um método para cada operação oferecida. A parte mais externa da figura 16 mostra como a interface de métodos do escalonador foi adaptada em uma função que define a interface do bloco em hardware. Basicamente, a união de todos os possíveis parâmetros dos métodos é manifestada na assinatura da função *call*. A função *call* também possui um parâmetro extra (*MethodId*) que identifica qual operação será realizada pelo bloco de hardware.

5.1.4 Outras considerações sobre HLS

As modificações no escalonador foram guiadas pelas características das ferramentas disponíveis (Calypto Design Systems, 2011; Xilinx, 2012), no entanto, muitas dessas características são comuns às ferramentas ESL que suportam HLS baseada em C++. De uma maneira geral, as ferramentas limi-

[3]no contexto de projeto digital de hardware, o termo *top level* é mais comum

tam o uso de funcionalidades do C++ que requerem algum tipo de estrutura dinâmica para serem implementadas em software (*e.g. heaps*, pilhas, tabelas de métodos virtuais), como os operadores *new* e *delete*, recursão e polimorfismo dinâmico. Anteriormente, foi mostrado como a alocação de recursos foi tratada no caso específico do escalonador, e na seção 5.3.1 será mostrado como essa funcionalidade pode ser implementada como um aspecto. Recursão e polimorfismo, no entanto, possuem implementações alternativas e sintetizáveis que podem ser feitas utilizando metaprogramação estática. Polimorfismo estático, por exemplo, pode ser implementado como mostrado abaixo:

```
1  template <typename derived_class> class Base {
2      void operation() {
3          static_cast<derived_class*>(this)->operation();
4      }
5  };
6
7  class Derived : public Base<Derived> {
8      void operation();
9  };
```

Classes podem derivar de uma instanciação de um template de uma classe-base usando elas mesmas como parâmetro para o template. Isso também é conhecido como CRTP (do inglês *Curiously Recurring Template Pattern*) (COPLIEN, 1995), e permite a determinação estática de chamadas para métodos virtuais de uma classe-base.

Outra questão que deve ser mencionada é o uso de tipos de dados com precisão em nível de bit. As ferramentas de síntese normalmente fornecem bibliotecas específicas (*e.g.* os tipos `ac_int/ac_fixed` do CatapultC) que permitem ao projetista utilizar apenas a quantidade de bits necessária para cada algoritmo. Contudo, enquanto precisão em nível de bits resulta na geração de um hardware mais eficiente, o seu uso em software gera um custo adicional pois muitos compiladores e ISAs suportam tipos inteiros com uma largura fixa (*e.g.* 8/16/32/64 bits), o que requer operações binárias (*e.g.* deslocamento bit-a-bit e máscaras) adicionais para emular o comportamento com precisão em nível de bit.

Nos estudos de caso neste trabalho, foram mantidos os tipos de dados padrão do C++ com o objetivo de minimizar as diferenças entre o C++ para hardware e o C++ para software e assegurar a compatibilidade com uma gama maior de ferramentas de síntese/compiladores. No entanto, uma possível maneira de utilizar precisão em nível de bits e ao mesmo tempo manter certa uniformidade no código, seria utilizar as declarações `typedef` do C++ para definir os tipos de dados suportados em cada domínio. Por exemplo, um tipo inteiro de 5 bits pode ser definido em hardware usando `typedef ac_int<5> int5`, e em software usando `typedef char int5`, pois `char` é o tipo nativo de menor largura com o tamanho especificado. Entretanto, o projetista deve tomar um cuidado extra para manter a consistência semântica ao misturar tipos

com diferentes larguras.

Finalmente, as questões remanescentes são referentes ao processo de síntese de alto nível em si. No capítulo 6 será explicado de forma um pouco mais detalhada os passos utilizados pela ferramenta utilizada na síntese dos estudos de caso. Mas, de uma maneira geral, um mesmo algoritmo descrito em C++ pode gerar diferentes implementações em hardware. Por exemplo, um laço pode ter cada uma das suas iterações executadas em um único ciclo de clock, ou pode ser completamente desenrolado de forma que todas as iterações executem no mesmo ciclo, aumentando o desempenho a um custo adicional de área de silício, devido às unidades funcionais extras para executar todas as iterações ao mesmo tempo. Este tipo de decisão é normalmente tomada baseada em *diretivas de síntese* que são especificadas separadamente da descrição do algoritmo em C++. É importante destacar que a definição e refinamento dessas diretivas é parte do processo de exploração do espaço de projeto e não é parte do escopo desse trabalho.

5.2 DIFERENÇAS ENTRE HARDWARE E SOFTWARE

Como pode ser visto na seção 5.1, ferramentas de HLS possibilitam a síntese de hardware a partir de algoritmos descritos em linguagens como C++. Quando feitas para este fim, as implementações dos algoritmos em si não são muito diferentes do que é feito em implementações visando software. Contudo, como foi possível observar, a nível de componente, considerando a integração em um sistema completo, surgem várias características que diferem as implementações para hardware das implementações para software. Esta seção estabelece quais dessas características serão consideradas e, em seguida serão apresentadas as estratégias para separá-las apropriadamente de forma a obter descrições unificadas em C++.

A tabela 2 sumariza os padrões mais comuns de comunicação entre componentes. No domínio de software, componentes podem ser objetos que se comunicam por meio de invocação de métodos (considerando uma abordagem baseada em OOP), enquanto que no domínio de hardware, componentes comunicam-se utilizando sinais de entrada e saída e protocolos de *handshaking* específicos. Para comunicação entre domínios distintos, o software deve prover uma HAL e rotinas para o tratamento de interrupções (do inglês *Interrupt Service Routine* — ISR) apropriadas, enquanto que o componente em hardware deve estar ciente que esta requisitando operações implementadas em software.

Em algumas abordagens em nível de sistema, como TLM, componentes comunicam-se por meio de leituras e escritas em *canais*. Em processos posteriores de refinamento, estes canais podem ser mapeados para barramentos,

Tabela 2: Padrões comuns de comunicação entre componentes. O chamador (*Caller*) requisita operações do chamado (*Callee*).

Direção	Tipo de comunicação	
	Caller	*Callee*
SW→HW	HAL envia comandos para o hardware através de uma infraestrutura de comunicação(*e.g.* um barramento ou uma NoC)	A interface de comunicação recebe os comandos e dispara as operações
HW→SW	O hardware interrompe o software e espera que a operação seja completada. Pode haver a transferência de dados entre o componente e a memória do sistema (*e.g. Direct Memory Access* (DMA)).	Uma ISR chama a operação requisitada e notifica o hardware quando esta for completada
SW→SW	Interface de chamada de funções/métodos	
HW→HW	Sinais/*Handshaking*	

conexões ponto-a-ponto, ou chamadas de funções, dependendo de como serão implementados os componentes que estão se comunicando (em hardware ou software). Apenas para ilustrar algumas diferenças entre abordagens OOP e TLM, a figura 17 mostra um sistema simples modelado usando ambas as abordagens.

O modelo OOP é mais expressivo e claro na maneira que componentes e interações são definidas. Um componente pode ser implementado tanto como um objeto global (`C1` e `C3`) quanto como parte de outro objeto (`C2`). Por outro lado, a própria estrutura dos modelos TLM induz um mapeamento mais natural para implementações físicas. Em OOP a estrutura original acaba sendo "desmontada" caso diferentes objetos em uma mesma hierarquia de classes representem componentes que deverão ser implementados em diferentes domínios. Por exemplo, no modelo orientado a objetos da figura 17, `C2` está "dentro" de `C1`, mas, na implementação final, `C1` poderia ser implementado como um componente em hardware, enquanto `C2` poderia ser executado como software em um processador. Como a estratégia proposta nesse trabalho é baseada em OOP, o mesmo demonstra uma maneira de tornar estes diferentes mapeamentos transparentes para o projetista.

A outra característica discutida que distingue hardware de software é a *alocação de recursos*. O hardware é "congelado" no sentido que não é possível "alocar mais hardware", dessa forma muitas característica comuns em software, como alocação dinâmica de memória, não estão disponíveis. Portando, em

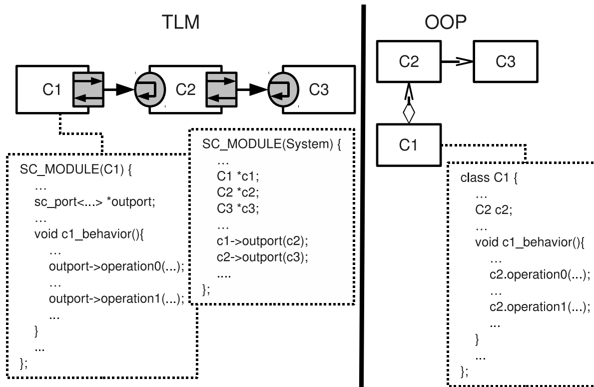

Figura 17: Comunicação entre componentes em modelos no nível de transação (à esquerda) e orientados a objetos (à direita)

descrições apropriadas para síntese de hardware, todas as estruturas de dados devem residir em memória estaticamente alocada. Alguns trabalhos focam nesse problema e apresentam propostas baseadas nas capacidades de reconfiguração dinâmica das FPGAs para suportar a instanciação de hardware em tempo de execução (ABEL, 2010) e para integrar componentes em hardware com objetos presentes na memória principal do sistema (LÜBBERS; PLATZNER, 2009). No entanto, este não é o foco deste trabalho. Este trabalho tem como objetivo principal a definição de estratégias para a implementação de componentes de forma que as características mencionadas possam ser encapsuladas como aspectos.

5.3 IMPLEMENTAÇÃO UNIFICADA DE COMPONENTES DE HARDWARE E SOFTWARE

O primeiro ponto é definir o que é, de fato, uma *implementação unificada* de um componente. Código C++ unificado e apropriado para implementação automática em hardware e software deve ser desenvolvido seguindo um cuidadoso processo para que o código não agregue aspectos específicos de hardware ou software. Estas características serão mais tarde incorporadas aos componentes utilizando um mecanismo de aplicação de aspectos. Isto deve ser levado em conta ao implementar um componente, caso contrário, esse tipo de adaptação não será possível. Como mencionado anteriormente, este trabalho leva em conta as seguintes diferenças básicas entre hardware e software: a

interface de comunicação e *alocação de recursos*.

Ao assumir que o estado interno de um componente pode ser influenciado apenas através da invocação dos métodos na sua interface pública (*e.g.* não permitir o uso de memória compartilhada entre os componentes), a adaptação da interface de comunicação é simples. Na perspectiva do software, a interface de métodos pode ser usada diretamente sem adaptações. Na perspectiva do hardware, ele dever ser adaptada para satisfazer os requisitos da ferramenta de síntese. A seção 5.3.1 dá mais detalhes sobre o encapsulamento da interface de métodos.

Por outro lado, o encapsulamento de alocação de recursos como um aspecto requer uma implementação que permita que essa característica seja tratada de forma externa ao componente. Nas seções 4.1 e 5.1, foi mostrado a decomposição de domínio e implementação do escalonador do EPOS, ilustrando uma abordagem de projeto que favorece a separação de mecanismos de alocação como aspecto.

Assumindo que os componentes foram modelados levando em conta as diretivas apresentadas até então, as próximas seções mostram como os mecanismos de separação de aspectos propostos por ADESD podem ser usados para obter componentes suscetíveis a geração de tanto software quanto hardware.

5.3.1 Encapsulamento de aspectos de hardware e software

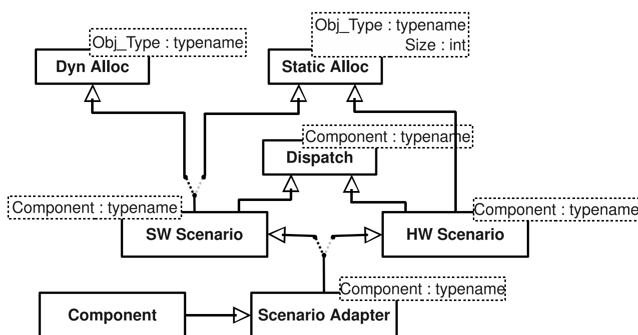

Figura 18: Aplicação de aspectos de hardware e software utilizando um adaptador de cenário

Anteriormente foram definidos dois aspectos que devem ser tratados separadamente de acordo com o domínio de implementação do componente: *alocação de recursos* e *interface de comunicação*. A figura 18 mostra como

os *adaptadores de cenário* foram utilizados para isolar estes aspectos. Como mencionado no Capítulo 4, o conjunto de aspectos que fazem parte de um cenário são incorporados por um artefato que define o cenário em si. O adaptador de cenário então redefine o componente-base, envolvendo o comportamento original com operações específicas do cenário. Esta estrutura é definida na forma de um *framework metaprogramado* que é totalmente implementado em C++ usando OOP e técnicas de metaprogramação estática. Cada parte da figura 18 é explicada em mais detalhes a seguir.

5.3.1.1 Alocação de recursos

A classe *Static Alloc* generaliza a estrutura mostrada na seção 5.1 e define um aspecto de alocação estática para lidar com a ausência de alocação dinâmica em hardware, sendo, então, parte do *cenário hardware* (representado pela classe `HW Scenario` na figura 18). Todas as operações do componente devem passar por esse alocador que é utilizado para reservar e liberar, sob demanda, o espaço de armazenamento. Por exemplo, os métodos `insert` e `remove` da classe `Scheduler` (figura 15) são redefinidos no seu adaptador de cenário como mostrado abaixo:

```
1  Link insert(Object obj, Criterion crit) {
2      Link link = Scenario::allocate(obj, crit);
3      Component::insert(Scenario::get(link));
4      return link;
5  }
6  Object remove(Link link) {
7      Object obj = Component::remove(Scenario::get(link));
8      Scenario::free(link);
9      return obj;
10 }
```

`Scenario::allocate(obj,crit)` cria um elemento para a lista de escalonamento e recebe como argumento o objeto sendo escalonado e o seu critério de escalonamento. Após a criação do elemento da lista, ele é obtido através do método `Scenario::get` e inserido no escalonador (referenciado usando o alias `Component`). Nas implementações em hardware, os métodos `Scenario::*` são mapeados para operações implementadas no aspecto `Static Alloc`. Nesse cenário, a quantidade de memória disponível para nodos e objetos é definida em tempo de síntese e requisições de alocação são simplesmente mapeadas para um espaço livre. O trecho de código abaixo sumariza a implementação do aspecto `Static Alloc`. Os métodos `allocate` são também templates que permitem que diferentes argumentos sejam repassados ao construtor do objeto sendo alocado.

```
1  template<typename Obj_Type, int size>
2  class Static_Alloc {
3   ...
```

```
4      Link allocate () {...}
5
6      template<typename T0>
7      Link allocate(T0 &t0) {
8          Link link = search();
9          //este array armazena os objetos
10         elements[link] = Obj_Type(t0);
11         return link;
12     }
13
14     template<typename T0, typename T1>
15     Link allocate(T0 &t0, T1 &t1) {...}
16 ...
17     void free(Link link) {
18         //apenas marca a local correspondente como disponível
19         alloc_bitmap[link] = false;
20     }
21
22     Obj_Type* get(Link link){
23         return &elements[link];
24     }
25 ...
26 private:
27     inline Idx_Type search(){
28         Idx_Type idx = Idx_Type();
29         for(; idx < Max; ++i)
30             if (alloc_bitmap[idx] == false) {
31                 _alloc_bitmap[idx] = true;
32                 return idx;
33             }
34         return idx; //== Max
35     }
36 };
```

Nas implementações em software, alocação dinâmica está disponível, logo, no *cenário software* (representado pela classe `SW Scenario`) a alocação pode ser tratada tanto pelo aspecto `Static Alloc` como pelo aspecto `Dynamic Alloc`, como mostrado na figura 18.

Uma abordagem, similar a descrita neste trabalho, que também viabiliza alocadores externos para contêineres como listas, é utilizada pela STL (STEPANOV; LEE, 1995). A princípio, este trabalho poderia ter sido feito com base na STL, contudo as implementações atuais da mesma não são sintetizáveis, o que motivou a definição de alocadores próprios.

5.3.1.2 Interface de comunicação

O aspecto `Dispatch` é usado, em conjunto com `HW Scenario`, para definir um novo ponto de entrada para o componente de forma que ele seja compatível com os requisitos das ferramentas de HLS. Nas ferramentas CatapultC e AutoESL, por exemplo, a interface visível do bloco RTL resultante é inferida a partir da *assinatura de uma única função*. Essa função de despacho é definida pelo `Dispatch` e possui um *id da operação* como seu primeiro parâmetro, interpreta o seu valor, executa conversões de tipos necessárias, e chama os

métodos apropriados do componente. O valor retornado pelas chamadas ao componente também é verificado, convertido se necessário, e atribuído a um dos parâmetros de saída da função de despacho.

`Dispatch` deve ser especializado manualmente para cada componente pois não foi possível descrever a funcionalidade de despacho de forma genérica utilizando metaprogramação estática. Além disso, cada ferramenta de síntese requer uma codificação específica para a função de despacho, que também é influenciada pelo protocolo de IO utilizado pela plataforma de hardware a qual o componente final será acoplado. Essa variabilidade será abordada na seção 5.4, que foca nas questões relacionadas à integração e comunicação entre componentes no sistema.

O mecanismo de despacho, a princípio, não é necessário no cenário software, pois neste, todas as operações são requisitadas através da invocação direta de métodos. Contudo, como mostra a figura 18, esse aspecto também é incorporado por `SW Scenario`. Em sistemas nos quais um componente em software possa atender chamadas de um componente em hardware, um mecanismo de despacho em software acaba sendo necessário pois o hardware não invoca diretamente os métodos do componente em software. A comunicação geralmente ocorre através de troca de mensagens e interrupções (tabela 2). Essas questões também serão tratadas com mais detalhes na seção 5.4.

5.3.1.3 Definição do cenário

Metaprogramação com templates é utilizada para definir a relação entre componentes, cenários e aspectos. No caso do cenário hardware, a incorporação é feita por `HW Scenario` simplesmente usando herança múltipla. Já para o cenário software, a agregação de aspectos no cenário é feita utilizando uma herança condicional metaprogramada. Isso é especificado na figura 18 através da notação $\dot{\vee}$, que indica que apenas uma das duas possíveis heranças pode ocorrer ao mesmo tempo. O código abaixo mostra como a herança condicional é definida na declaração de `SW Scenario`:

```
1 template <typename Component> public SW_Scenario :
2     public
3         IF<Traits<SW_Scenario<Component> >::static_alloc,
4             Static_Alloc,
5             Dyn_Alloc>::Result,
6     public
7         IF<Traits<Component>::has_dispatcher,
8             Dispatch<Component>,
9             Dispatch<void>>::Result
```

O aspecto base do cenário é o resultado do metaprograma `IF`, que depende de configurações definidas nos *traits* do componente (o conceito de *traits* é abordado também na seção 4.1). O trecho de código abaixo mostra a

classe Traits usada na definição do cenário:

```
1  // Configuração padrão para o cenário software
2  template <> struct Traits<SW_Scenario<void> > {
3      static const bool static_alloc = false;
4  };
5  // Configuração específica aplicada a 'Component'
6  template <> struct Traits<SW_Scenario<Component> >:
7                     public Traits<SW_Scenario<void> > {
8      static const bool static_alloc = true;
9  };
```

A especialização de Traits para SW_Scenario<void> define a configuração padrão para o cenário software, que se refere ao uso de alocação dinâmica (static_alloc = false). A especialização para SW_Scenario<Component> herda a configuração padrão, mas redefine static_alloc para true. Em outras palavras, alocação estática será usada ao invés da dinâmica sempre que o cenário software for aplicado a Component.

A implementação do metaprograma IF que escolhe entre as possíveis opções é mostrada abaixo:

```
1  template<bool condition, typename Then, typename Else>
2  struct IF { typedef Then Result; };
3
4  template<typename Then, typename Else>
5  struct IF<false, Then, Else> { typedef Else Result; };
```

A sua implementação é análoga ao exemplo do fatorial mostrado anteriormente e usa especialização parcial de templates. Se o valor de condition for igual a true, o valor de Result é definido como sendo o tipo especificado por Then. Isso é codificado na definição-base do template. O template é parcialmente especializado para o caso em que condition é igual a false, definido Result como sendo o tipo especificado por Else.

Uma última observação a respeito da definição dos cenários é o fato do próprio componente ser utilizado como parâmetro de template para o cenário, permitindo que um cenário seja especializado para diferentes componentes, como exemplificado abaixo:

```
1  template<class T> Scenario {...};
2
3  template<> Scenario<Component_0> {...};
4  ...
5  template<> Scenario<Component_n> {...};
```

A primeira definição descreve a implementação genérica do cenário. No entanto, certos componentes podem requerer diferentes definições do mesmo cenário. Esta questões podem ser tratadas de forma clara e uniforme através da especialização do cenário para componentes específicos. Outro ponto importante é que essas especializações também podem ocorrer a nível de família. Por exemplo, uma determinada família de componentes não requer qualquer mecanismo de alocação de recursos, dessa forma, basta prover uma especialização de SW Scenario e HW Scenario para a família inteira de forma que os

aspectos de alocação não sejam herdados. Ou seja, `SW Scenario` será vazio e não terá nenhum efeito sob o componente, enquanto `HW Scenario` incorporará apenas o aspecto `Dispatch`.

5.3.1.4 Definição do adaptador de cenário

O adaptador de cenário incorpora os cenários usando a herança condicional descrita anteriormente. O trecho de código abaixo mostra a definição do adaptador de cenário:

```
1  template <typename Component> class Scenario_Adapter :
2      private
3          IF<Traits<Component>::hardware,
4              HW_Scenario<Component>,
5              SW_Scenario<Component> >::Result,
6      private
7          Component
8  {
9      ...
10 };
11
12 // Traits  de um componente
13 template <> struct Traits<Component> {
14     static  const bool hardware = true;
15 };
```

O adaptador herda o comportamento do componente diretamente da sua implementação unificada, enquanto o cenário-base é escolhido através do metaprograma `IF` de acordo com o que está definido nos *traits* do componente. Componentes que podem ser implementados como hardware ou software possuem uma configuração chamada `hardware`, que é usada para definir o domínio para o qual o componente será adaptado. O corpo da implementação do adaptador de cenário contém as adaptações necessárias aos métodos do componente, como descrito na seção 5.3.1.1

5.4 INTEGRAÇÃO ENTRE COMPONENTES UNIFICADOS

Até o momento, os mecanismos de adaptação têm levado em conta os componentes de forma isolada. Ao considerar o sistema como um todo, com componentes podendo migrar livremente entre hardware e software, uma outra questão importante pode ser levantada. Como discutido na seção 5.2, a forma como a hierarquia entre os componentes é definida em um modelo orientado a objeto requer que esse modelo seja "desmontado" quando diferentes objetos em uma mesma hierarquia de classe representam componentes que serão implementados em diferentes domínios. A figura 19 resgata o modelo da figura 17 e ilustra dois possíveis mapeamentos do mesmo para plataforma

física.

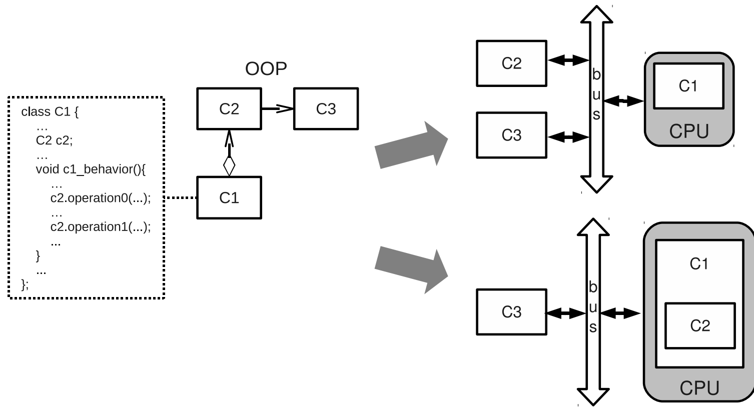

Figura 19: Possíveis mapeamentos de um modelo orientado a objetos para uma implementação física

`C2` é um atributo de `C1`. Considerando que `C1` esteja no topo da hierarquia em um determinado processo de HLS/compilação, `C2` seria sintetizado/compilado juntamente com `C1` (o segundo mapeamento na figura 19). No entanto, caso a intenção do projetista seja ter `C1` e `C2` como componentes fisicamente separados em diferentes domínios (o primeiro mapeamento na figura 19), então um mecanismo adicional é necessário para expressar essa possível mudança quando um componente é instanciado.

Uma maneira de tratar essa questão é usar os conceitos das plataformas de objetos distribuídos (RINCÓN et al., 2009). A figura 20 ilustra interações entre-domínios de `C1` e `C2`. O componente chamado é representado no domínio do chamador por um *proxy*. Quando uma operação é invocada usando um proxy, os argumentos da operação são serializados na forma de uma mensagem de requisição e enviados através de um canal de comunicação (*channel*) para a implementação real do componente. Um *agent* recebe as requisições, deserializa os argumentos e executa a invocação da operação localmente. Todo o processo é então repetido na direção oposta, produzindo mensagens de resposta que carregam eventuais valores de retorno das operações.

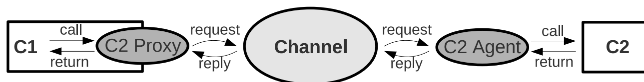

Figura 20: Comunicação entre-domínios usando proxies e agents.

Para integrar esses mecanismos de invocação remota na descrição dos componentes, o primeiro passo é definir uma maneira de deixa-los transparente quando o componente é usado. Uma solução eficiente é utilizar metaprogramas para substituir a definição do componente pela definição de um proxy ou agent à medida que for necessário. A figura 21 mostra como um componente e seus proxies/agents podem ser definidos e em que ponto que o mapeamento ocorre.

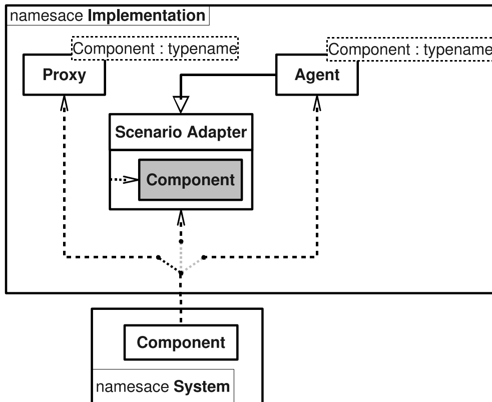

Figura 21: Mapeamento da definição de um componente

Os *namespaces* do C++[4] são utilizados para separar as implementações dos componentes do que é efetivamente visível. A implementação de todos os componentes e os artefatos de adaptação é feita no `namespace Implementation`. As definições no `namespace System` são as usadas para instanciar os componentes e devem mapear para a definição correta dentro de `Implementation`. Por exemplo, considerando o modelo mostrado na figura 19, a declaração de `C2` dentro de `C1` deve ser feita usando o tipo `System::C2`, que terá os seguintes possíveis mapeamentos, dependendo do particionamento final entre hardware e software:

- um proxy para software, se `C1` for sintetizado como um IP de hardware e `C2` estiver em software;
- um proxy para hardware, se `C1` estiver em software e `C2` for um IP em hardware;
- `Implementation::Scenario_Adapter<C2>`, se `C1` e `C2` estiverem no mesmo domínio. Neste caso `C2` é adaptado para o mesmo domínio de `C1`.

[4]um mecanismo usado para definir um escopo para declarações de tipos e objetos

Além disso, `System::C2` é mapeado para um agent quando ele é o componente atualmente sendo compilado/sintetizado e recebe invocação de métodos através de outros componentes em diferentes domínios.

A realização desse tipo de mapeamento usando metaprogramação segue a mesma abordagem já mostrada anteriormente para o metaprograma `IF`. Usando os traits dos componentes, o mapeamento mencionado acima para `System::C2` poder ser feito como mostrado abaixo:

```
1  namespace System {
2  typedef MAP<Implementation::Scenario_Adapter<C2>, Implementation::Agent<C2>,
          Implementation::Proxy<C2>,
3                 false,
4                 Traits <C2>::hardware>::Result
5           C2;
6  }
```

`MAP` seleciona entre `Implementation::Scenario_Adapter<C2>`, `Implementation::Agent<C2>` e `Implementation::Proxy<C2>`, de acordo com os valores dos dois últimos parâmetros. A implementação do metaprograma `MAP` é mostrada abaixo:

```
1  template<typename Implementation, typename Agent, typename Proxy,
2           bool top_level, bool hardware>
3  struct MAP {
4      typedef IF<top_level,
5                    Agent,
6                 IF<hardware==Traits<System>::hardware_domain,
7                    Implementation,
8                 Proxy>::Result >::Result
9      Result;
10 };
```

O valor de `top_level` é igual a `true` quando um componente é o que está sendo atualmente sintetizado e uma interface de comunicação é necessária, por isso o mapeamento para um agent. O metaprograma usa o valor de `Traits<System>::hardware_domain` para saber se o código está sendo submetido a uma ferramenta de síntese de hardware ou ao um compilador de software. No caso do exemplo anterior, se `Traits<System>::hardware_domain` for igual ao parâmetro `hardware`, que na instanciação do metaprograma recebe o valor de `Traits<C2>::hardware`, então o mapeamento é feito diretamente para a implementação, caso contrário o mapeamento é feito para o proxy.

5.4.1 Framework de comunicação

A implementação dos canais, proxies e agents mostrados anteriormente pode ser realizada de diferentes formas e depende diretamente da arquitetura de hardware subjacente (*e.g.* usando barramentos, DMA, NoCs) e das funcionalidades do RTOS. Por exemplo, quando utiliza-se um canal de comunicação baseado em barramentos, um proxy em hardware pode ser implementado como

um *slave*[5] com registradores mapeados em memória e, então, notifica a CPU através de uma interrupção quando uma mensagem requisitando uma operação está pronta para ser lida. Alternativamente, em uma implementação baseada em NoCs, um interface orientada a pacotes deverá ser usada para transmitir mensagens contendo requisições.

Esta variabilidade é relacionada à arquitetura de hardware/software da plataforma-alvo e não deve afetar os componentes no nível de sistema. Assim, é importante separar a implementação específica do componente, que faz parte do modelo comportamental, da implementação específica da plataforma subjacente. Para lidar com essa questão, foi definido um framework metaprogramado em C++ que suporta diferentes implementações de proxies e agents. O framework é mostrado na figura 22.

A figura 22 estende a figura 21, ilustrando como as funcionalidades dos proxies e agents podem ser fatoradas e encapsuladas.

O template de classe `Serializer` implementa as partes independentes de plataforma dos mecanismos de serialização e deserialização. O valor do parâmetro é usado para selecionar qual implementação será usada: a especialização para software ou a especialização para hardware. Essa classe é então estendida pelos templates `Proxy_Common` e `Agent_Common`. As implementações dependentes de plataforma são definidas por especializações dos templates `Platform`, que indica a plataforma, e `hardware`, que indica a implementação para hardware ou software. `Proxy<Component>` e `Agent<Component>` definem os artefatos para cada componente e herdam a implementação específica de plataforma final segundo as definições nos `Traits` do sistema.

A seção 5.3.1.2 apresentou o aspecto `Dispatch`, que é usado para definir um novo ponto de entrada para o componente. Ao definir o framework, algumas funcionalidades são fatoradas entre os artefatos `Dispatch`, `Agent_Common` e `Agent`. Nesse caso, fica a cargo do agent a definição da função que estabelece a interface do componente e faz a deserialização dos parâmetros de cada operação. `Dispatch` fica responsável apenas por definir uma estrutura do tipo *switch* para selecionar qual método local será invocado com base na informação repassada pelo agent.

[5] o *master* (normalmente a CPU) comanda o barramento, sendo que nenhum *slave* envia pacotes se não forem solicitados explicitamente pelo *master*

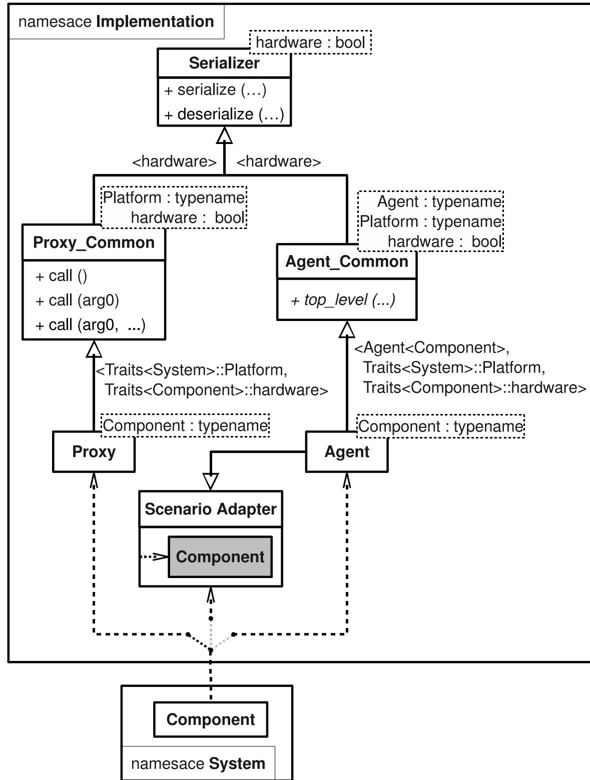

Figura 22: Diagrama de classes UML do framework de comunicação

A relação entre os agent e os aspectos de despacho fica mais clara ao observar os exemplos de implementação a seguir. O primeiro mostra uma possível implementação para `Agent_Common`:

```
1  template<typename Agent>
2  class Agent_Common<Agent, Platforms::CPP, true> :
3          public Serializer<true> {
4  ..
5    #pragma hls_design top
6    void top_level (...) {
7       //deserialização dos parametros de entrada usando Serializer::deserialize
8       ...
9       static_cast<Agent*>(this)−>dispatch(...);
10
11      // serialização  dos parametros de saída usando Serializer::serialize
12      ...
13   }
14 };
```

Por questões de simplicidade, a assinatura e a implementação completa do método `top_level` foram omitidas. Esses detalhes são dependentes da ferramenta de síntese (se for um agent para um componente em hardware), do RTOS (se for um agent para software), do protocolo de entrada/saída do componente e da ferramenta de síntese utilizada. Cada uma dessa variações resulta em uma especialização diferente. No exemplo acima, é definida a especialização em hardware para uma plataforma fictícia `CPP`, utilizando uma única função em C++ para definir a interface do componente. Algumas ferramentas de síntese, como o Cynthesizer (Forte Design Systems, 2011), requerem que a interface seja definida utilizando um modulo em SystemC. Este novo estilo de interface pode ser definido através de uma nova especialização de `Agent_Common`:

```
1  #pragma hls_design top
2  template<typename Dispatcher>
3  class Agent_Common<Agent, Platforms::SystemC, true> :
4          public Serializer<true>,
5          public sc_module {
6
7    //declaração explicita  dos sinais  da interface
8    sc_in<int>  data_in;
9    sc_out<char> ...
10
11   SC_CTOR(Agent_Common){
12        SC_THREAD(top_level);
13   }
14
15   //Análoga a versão em C++
16   void top_level() ;
17 };
```

Neste exemplo simplificado, a interface do bloco em hardware é extraída dos sinais definidos na declaração do modulo. A função que implementa o comportamento do agent é análoga a função original em C++, mas neste caso ela deve ser registrada como um processo em SystemC (`SC_THREAD`, no caso do exemplo).

Nos exemplos acima, a mesma abordagem utilizada para implementar

polimorfismo estático é usada para ter acesso à implementação da função de despacho implementada em `Dispatch`. A coerção `static_cast<Agent*>(this)` em `Agent_Common` é possível, pois `Agent<Component>` herda de `Agent_Common`. `Agent<Component>` também herda de `Scenario_Adapter<Component>`, que, por sua vez, herda de `Dispatch<Component>` através do cenário. Logo, um objeto do tipo `Agent_Common` pode assumir o tipo `Agent<Component>`, que pode assumir o tipo `Dispatch<Component>`. A mesma relação é utilizada na implementação do método `dispatch`, que precisa ter acesso a interface de métodos do componente, devidamente adaptada pelo adaptador de cenário:

```
1 template<>
2 void Dispatch<Component>::dispatch(...){
3     switch(op){
4     case OP_0:
5         static_cast<Scenario_Adapter<Component>*>(this)->operation0(); break;
6     case OP_1: ...
7     }
8 }
```

5.4.2 Componentes híbridos *vs* Proxy+Agent: comunicação e microarquiteturas

Em um trabalho anterior, Marcondes e Fröhlich (2009) já haviam partido dos conceitos de ADESD para desenvolver uma arquitetura de *Componentes Híbridos*. Naquele trabalho foi desenvolvida uma arquitetura comum para a implementação e comunicação entre componentes nos domínios de hardware e/ou software. Componentes são divididos em três categorias, segundo a forma como as operações são oferecidas aos clientes: componentes síncronos, componentes assíncronos e componentes autônomos. Foram fixados três modelos de comunicação SW->HW entre os componentes, definindo um padrão de iterações para cada uma das categorias acima. A figuras 23-25 mostram diagramas UML de atividades que descrevem estes modelos de comunicação.

Componentes síncronos (figura 23) realizam atividades apenas quando seus métodos são explicitamente invocados, de modo que o componente que requer o serviço é bloqueado até que a tarefa seja finalizada. Um componente híbrido implementado em hardware é sempre representado, em software, pelo seu respectivo *mediador de hardware*(ver seção 4.1). Assim, quando um componente síncrono é implementado em hardware, o seu mediador irá bloquear o cliente até que o hardware termine o serviço solicitado. Isto pode ser facilmente implementado pelo mediador através do método de “pooling” em um registrador de estado do hardware (mecanismo conhecido como “busy-waiting”) ou através da suspensão do fluxo de execução que está aguardando o serviço, que é reativado através de mecanismos de interrupção do hardware (mecanismo

conhecido como “idle-waiting”).

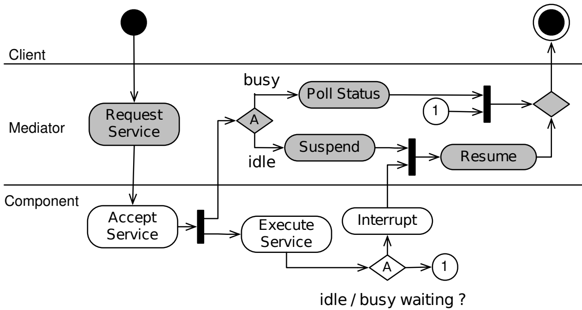

Figura 23: Componentes híbridos síncronos (MARCONDES, 2009)

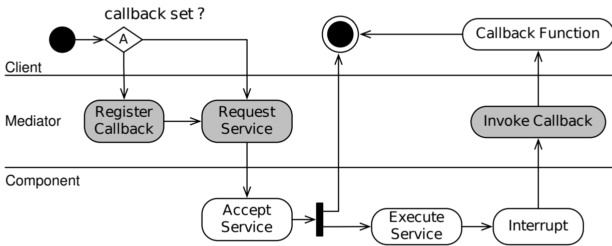

Figura 24: Componentes híbridos assíncronos (MARCONDES, 2009)

Componentes assíncronos (figura 24) são componentes que permitem que seu serviço seja executado de forma assíncrona. Seus métodos são invocados explicitamente, contudo eles não bloqueiam o componente cliente que solicitou o serviço enquanto o mesmo é executado. O componente cliente solicita o serviço através de seu mediador, que irá acionar o componente em hardware, e então retorna o fluxo de execução para o cliente. Uma vez finalizada a requisição, o componente em hardware gera uma interrupção que irá acionar um tratador de seu mediador, responsável por acionar o mecanismo de chamada de retorno utilizada pelo sistema.

Componentes autônomos (figura 25) executam seus serviços de forma independente, sem necessitar de uma chamada explícita de um componente cliente. componentes autônomos implementados em hardware são implementados através de redirecionamento de eventos para o mesmo, através de seu mediador de hardware, que irá receber os eventos destinados ao componente, e repassá-los ao hardware de acordo com a sua implementação (i.e. notificar

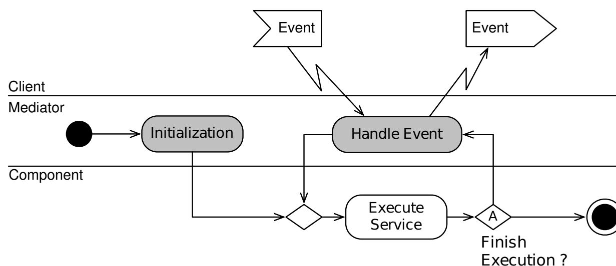

Figura 25: Componentes híbridos autônomos (MARCONDES, 2009)

através de um registrador mapeado em memória). Os eventos gerados pelo componente autônomo são criados também através do seu mediador, que os recebe do hardware através do mecanismo de interrupções.

No contexto da presente dissertação, as microarquiteturas fixam padrões de comunicação que cobrem uma gama maior tipos de comportamentos que os componentes podem apresentar. O mecanismo de proxy/agent adotado neste trabalho suporta basicamente a categoria de componentes síncronos. No entanto optou-se por adota-los pois esta estrutura oferece uma maior flexibilidade em relação aos diferentes domínios nos quais os componentes podem ser implementados. As microarquiteturas descritas anteriormente definem que a comunicação com um componente híbrido no domínio de hardware é realizada sempre por intermédio de seu mediador de hardware. Desta forma, quando outro componente em hardware precisa requisitar um serviço, o mesmo é realizado através da geração de uma interrupção, que irá ser tratada por um método específico do mediador, responsável por identificar o serviço e requisitá-lo, da mesma forma que é realizada através do componente em software. Proxies/agents, por outro lado, são definidos tando no domínio de hardware quanto de software, e permitem que um componente em hardware invoque diretamente operações de outro componente em hardware, sem a necessidade de criar um nível extra de indireção.

5.5 SUMÁRIO E DISCUSSÃO

Este capítulo mostrou que uma única implementação em C++, chamada de *implementação unificada*, pode ser usada para gerar tanto hardware quanto software desde que ela seja desenvolvida seguindo processo cuidadoso com diretivas de projeto bem definidas. Por “processo cuidadoso”, assume-se um processo que leva em conta desde o início que as dependências relativas ao

hardware ou software serão aplicadas mais tarde através de um adaptador de cenário. Estas dependências e sua influência na implementação unificada são sumarizadas abaixo:
Interface de comunicação: limitar a interação entre componentes a chamadas para métodos permite a criação de um mecanismo externo genérico que pode ser usado pelas ferramentas de síntese para inferir a interface final do hardware.
Alocação de recursos: componentes que requerem alocação de recursos podem ser desenvolvidos para trabalhar apenas com referências para os recursos esperados, como exemplificado no exemplo do escalonador. O processo de alocação em si pode, então, ser feito externamente usando a abordagem mais apropriada para o domínio de implementação.

Neste trabalho o conceito de aspectos e adaptadores de cenário foi aplicado para incorporar as dependências de hardware e software, sendo todos os mecanismos implementados utilizando metaprogramação com templates em C++. No entanto, outros métodos têm sido usados para atingir propósitos semelhantes. Por exemplo, macros do pré-processador da linguagem C[6] (*e.g.* blocos `#define` e `#ifdef`) permitem que código específico de hardware ou software seja introduzido ou removido de forma condicional durante o processo de compilação. Essa é uma abordagem comum para distinguir C++ executável de C++ sintetizável em muitos projetos industriais sendo desenvolvidos atualmente. Entretanto, os templates do C++ oferecem duas vantagens significativas em relação às já defasadas macros da linguagem C: 1) macros são basicamente um mecanismo de substituição textual e não oferecem nenhum tipo de verificação sintática e checagem de tipo, enquanto templates são *type-safe*, ou seja, associar valores a parâmetros de tipos não compatíveis resulta em erros de compilação; 2) a implementação de um template poder ser parcialmente ou totalmente especializada para diferentes parâmetros. Especialização de templates é o que, de fato, torna possível os mecanismos metaprogramados apresentados neste capítulo. Metaprogramação usando templates tem a suas desvantagens, no entanto. Como os templates são executados internamente pelo compilador C++, para depurar metaprogramas de forma iterativa seria necessário um depurador para o processo de compilação em si. Para identificar erros nos metaprogramas, o desenvolvedor fica limitado a analise manual do código fonte e às mensagens de erro emitidas pelo compilador.

Como visto na capítulo 3, outra abordagem para gerar hardware ou software a partir de uma única implementação seria depender exclusivamente de mecanismos implementados nas ferramentas EDA (SCHALLENBERG et al., 2009; KEINERT et al., 2009; DÖMER et al., 2008). Apesar de o resultado ser um fluxo de projeto mais automatizado que parte de um modelo em uma linguagem mais abstrata que C++, é criada uma forte dependência entre o

[6]um programa que recebe texto e efetua apenas conversões léxicas nele

estilo do modelo de entrada e ferramentas bem específicas. Nesse sentido, prover uma forma sistemática de descrever a semântica do processo no próprio modelo de entrada pode ser o caminho para reduzir esse tipo de limitação. C++ padrão ANSI e as suas capacidades OOP são extensivamente suportadas por ferramentas de síntese de alto nível e compiladores de software, dessa forma mantendo a abordagem proposta nesse capítulo aplicável a uma grande gama de fluxos de projeto e ferramentas.

Uma outra questão importante que deve ser mencionada é o fato de que nem sempre a implementação mais direta de um algoritmo, usada para executa-lo em software, é adequada para síntese de hardware. Em alguns casos, o algoritmo em C++ deve ser implementado de forma a expressar algumas questões arquiteturais que se espera obter no hardware final, caso contrário não é possível utilizar síntese de alto nível para obter resultados comparáveis aos que podem ser obtidos através de codificação manual em RTL. Um exemplo que ilustra essa situação é a implementação da transformada rápida de Fourier (do inglês *Fast Fourier Transform* — FFT)[7] mostrada em um trabalho relacionado (BUTT; LAVAGNO, 2012). É mostrada uma implementação na qual os acessos ao array que armazena os dados usados pelo algoritmo são reordenados para evitar conflitos no acesso aos blocos de memória para os quais o array foi mapeado. Essa reordenação aumenta o número de otimizações que podem ser aplicadas pela ferramenta de síntese, resultando em uma microarquitetura mais eficiente.

O fato de se utilizar uma implementação-base diferente do algoritmo para gerar hardware mais eficiente, entretanto, não inviabiliza o conceito de uma *implementação unificada*, pois essa mesma implementação ainda pode ser utilizada em software. Essa questão deve ser levada em conta pelo projetista que, mesmo durante e desenvolvimento de implementações unificadas dos componentes, deve ter em mente o tipo hardware/software que será obtido ao aplicar o componente em seu domínio final de implementação.

Um ponto final que pode ser discutido é também, como mencionado na seção 5.1.4, a exploração do espaço microarquitetural do componente em hardware através de diferentes diretivas de síntese (*e.g.* paralelização/pipelining de loops), resultando várias possíveis implementações para o mesmo algoritmo descrito em C++. Embora o tratamento desse tipo de diretiva não esteja entre os objetivos deste trabalho, é também interessante considerar como as técnicas propostas poderiam ser utilizadas para encapsular múltiplos conjuntos de possíveis implementações. Por exemplo, diretivas definidas usando *pragmas* poderiam ser especificadas como um aspecto. De fato, este aspecto teria que ser especializado para cada componente, ferramenta EDA e o tipo de

[7]algoritmo utilizado em processamento digital de sinais para converter um sinal do domínio do tempo para o domínio da frequência

microarquitetura do hardware, já que cada um desses elementos irá exigir um conjunto diferente de pragmas. Uma abordagem semelhante nesta linha foi proposta no escopo da linguagem LARA (CARDOSO et al., 2012), na qual os autores definem um compilador de aspectos cujo backend pode ser trocado para gerar diretivas específicas a partir de uma linguagem comum.

Além das configuração das diretivas de síntese, outra parte importante da exploração do espaço de projeto está na definição de quais componentes devem ser implementados como software ou hardware. Neste trabalho, foi proposta a utilização de *Traits* como um mecanismo que permite a definição do particionamento entre software/hardware. As configurações nos *Traits*, no entanto, são atualmente definidas pelo projetista de forma manual. Idealmente essa definição deve ser automatizada em um processo que escolhe um particionamento que satisfaz os requisitos da aplicação. Nesse escopo o trabalho de Cancian (2011) propôs um modelo evolucionário multiobjetivo para a exploração automática do espaço de projeto. Este trabalho foi desenvolvido em sinergia com a metodologia ADESD, de modo que o processo de exploração do espaço de projeto usa como entrada componentes compostos em um framework metaprogramado e gera automaticamente os *Traits* finais do sistema. Em trabalhos futuros, espera-se integrar os artefatos propostos nesta dissertação com o método de exploração do espaço de projeto proposto por Cancian.

6 IMPLEMENTAÇÃO

Este capítulo tem como objetivo abordar questões práticas importantes relacionadas à implementação das ideias propostas no Capítulo 5, descrevendo, principalmente, a arquitetura de hardware/software utilizada para suportar a comunicação transparente entre componentes. A seção 6.1 descreve a arquitetura de hardware/software da plataforma desenvolvida e a seção 6.2 descreve a implementação dos artefatos do framework de comunicação do Capítulo 5 para a plataforma descrita. Ao final é apresentado um sumário do fluxo de desenvolvimento utilizado.

6.1 UMA PLATAFORMA PARA O DESENVOLVIMENTO DE SOCS

A arquitetura de hardware/software descrita a seguir foi desenvolvida com o objetivo principal de prover uma plataforma flexível para a implementação de SoCs em FPGAs, oferecendo suporte aos mecanismos transparentes de comunicação propostos no capítulo 5. A figura 26 mostra uma visão geral da arquitetura proposta. Ela consiste em vários nodos conectados a roteadores, formando uma *Network-on-Chip* (NoC). Uma arquitetura baseada em NoCs foi escolhida tendo em mente a flexibilidade e escalabilidade desse tipo de interconexão. Enquanto MPSoCs baseados em barramentos são suficientes para sistemas mais centrados no software, nos quais componentes de hardware funcionam apenas como aceleradores passivos, eles não são uma boa opção para projetos mais heterogêneos, nos quais componentes de hardware podem ter um papel ativo (MICHELI et al., 2010).

Na arquitetura proposta, cada IP de hardware gerado a partir de um componente unificado é integrado como um nodo conectado em um roteador na rede (os `App nodes`). Componentes de software são compilados juntamente com um RTOS e são executados em um nodo de CPU (`CPU node`) que consiste basicamente em um *softcore*[1] e memórias. Periféricos de entrada e saída compartilhados são encapsulados em um nodo separado (`IO node`) do modo que eles podem ser acessados tanto pelos componentes de hardware quanto pelos componentes de software. Cada um desses serão detalhados a seguir.

[1] um processador implementado na lógica programável de uma FPGA

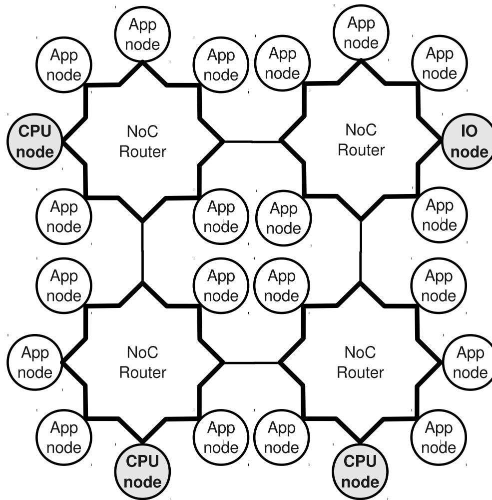

Figura 26: Visão geral da arquitetura proposta

6.1.1 RTSNoC

A *Real-time Star Network-on-Chip* (RTSNoC) (BEREJUCK, 2011) é o componente principal da arquitetura. A rede consiste em um conjunto de roteadores com uma topologia do tipo estrela que podem ser arranjados para formar uma rede *mesh* 2-D. A figura 27 mostra a estrutura de um roteador da rede.

Cada roteador possui oito canais bidirecionais que podem ser conectados a nodos da rede ou a outros roteadores. A RTSNoC foi projetada tendo em mente a sua aplicação em sistemas de tempo-real. Por essa razão, os árbitros dos roteadores implementam um algoritmo de escalonamento dinâmico baseado em prioridades, de modo que é possível estabelecer limites para a latência de comunicação entre dois nodos na rede.

6.1.2 Nodos de CPU e IO

As figuras 28a e 28b mostram a estrutura geral de um nodo de CPU e de um nodo de IO. A estrutura interna dos nodos é baseada na última versão da família de barramentos AMBA, o AXI4 (IEEE, 2010), que está se tornando um dos principais padrões industriais em interconexões baseadas em barramentos.

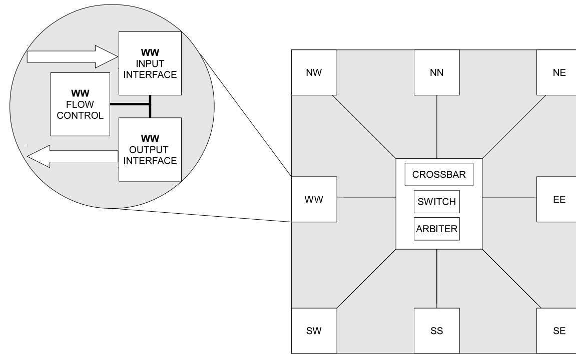

Figura 27: Estrutura do roteador da RTSNoC

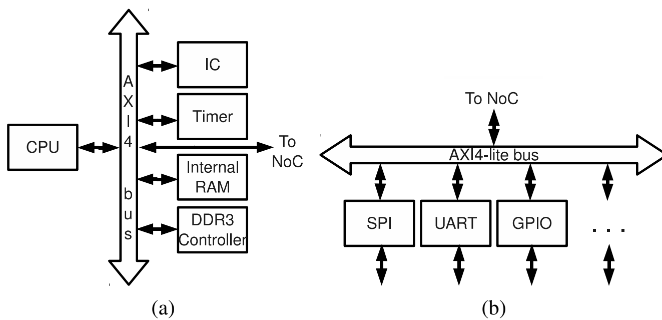

Figura 28: Nodo de CPU(a) e IO(b)

Na medida do possível, a implementação atual desses nodos foi baseada em IPs de código aberto, implementados em RTL e independentes do dispositivo alvo, com o objetivo maximizar a reusabilidade. Um sumário dos IPs sendo atualmente utilizados é mostrado abaixo:

Softcore Plasma[2] o Plasma é um softcore que oferece compatibilidade binária com processadores da arquitetura MIPS I. Ele é implementado em VHDL e possui um pipeline configurável de dois ou três estágios;

Barramento AXI4[3] uma implementação multiplexada do barramento AXI4. Suporta um dispositivo do tipo *master* e um número parametrizável de dispositivos do tipo *slave*;

Interfaces RTSNoC[3] adaptadores *AXI4 Slave -> RTSNoC* e *RTSNoC -> AXI4 Master* que permitem a conexão dos nodos de CPU e IO à NoC;

Timer[3] um timer de 32 bits. Utilizado como *Timestamp Counter* (TSC) e também pode ser configurado para gerar interrupções periódicas;

PIC[2] um controlador de interrupções programável;

RAM interna[4] IP de memória interna implementada utilizando BRAM da FPGA. Contém o *bootloader* durante a inicialização do sistema e pode ser usado posteriormente como uma cache;

IO básica[2] um dispositivo para controle de pinos de GPIO e um dispositivo de UART;

Controlador DDR3[4] controlador pra memória externa do tipo SRAM DDR3. Devido à complexidade desse IP, optou-se por usar uma implementação proprietária disponibilizada para FPGAs fabricadas pela Xilinx.

6.1.3 EPOS

Os componentes implementados como software são executados nos nodos de CPU com o suporte de um RTOS. A implementação atual utiliza o sistema operacional EPOS. O EPOS é um sistema operacional voltado para aplicações embarcadas. Ele foi desenvolvido segundo os conceitos de ADESD e é implementado em C++, fazendo uso extensivo de metaprogramação estática

[2] IP obtido no repositório de código aberto OpenCores: http://opencores.org/
[3] IP desenvolvido ao longo do trabalho
[4] IP gerado pela ferramenta *CoreGen* da Xilinx

para alcançar um alto grau de flexibilidade e ao mesmo tempo atender os requisitos de um sistema operacional embarcado (como desempenho, previsibilidade e tamanho de código).

A figura 29 apresenta uma visão geral do EPOS. Seguindo os conceitos de ADESD, o EPOS é dividido entre famílias de abstrações do sistema, que implementam as funcionalidade básicas do SO (*e.g.* gerenciamento de tarefas, memória e energia), e famílias de mediadores de hardware. Como já explicado anteriormente, os mediadores de hardware abstraem os dispositivos de hardware e mantem as abstrações do sistema independentes da plataforma. Os mediadores estão divididos em duas categorias: os mediadores em `architecture` abstraem questões relativas à ISA do processador (*e.g.* troca de contexto, proteção de memória, gerenciamento da pilha) enquanto os mediadores em `machine` abstraem o restante dos dispositivos presentes na plataforma.

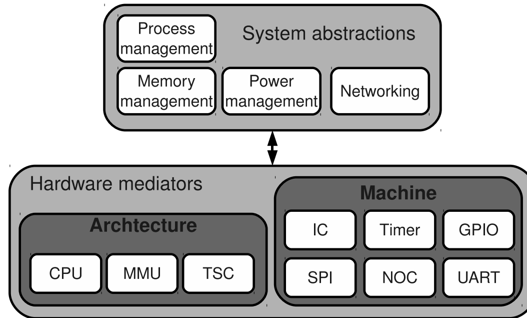

Figura 29: Organização dos componentes do EPOS

6.1.4 A plataforma virtual

Além de uma plataforma física, modelos de simulação dos componentes de hardware podem ser gerados e integrados para construir uma *plataforma virtual* do sistema. Plataformas virtuais estão presentes em muitos fluxos de projeto de SoCs (SCHALLENBERG et al., 2009; KEINERT et al., 2009; DÖMER et al., 2008) como uma alternativa mais rápida em relação às custosas simulações em RTL (CHIANG; YEH; TSENG, 2011). Plataformas virtuais são normalmente compostas por *simuladores de conjuntos de instruções* (do inglês *Instruction Set Simulator* — ISS) de CPUs e modelos dos componentes de hardware em níveis de abstração variados, desde TLM até modelos com precisão a nível de ciclos.

Para prover um ambiente de prototipação rápida usando os mecanismos propostos nesse trabalho, foi desenvolvida uma plataforma virtual em SystemC para simular a arquitetura de hardware descrita anteriormente. Os modelos dos componentes estão no nível TLM, tendo em vista a velocidade da simulação. A figura 30 mostra um diagrama que ilustra como a plataforma foi implementada.

Em SystemC TLM, componentes são definidos por classes que herdam do tipo `sc_module`. Por questões de simplicidade, essas classes são denotadas apenas com o estereótipo «`sc_module`» na figura 30. Os componentes comunicam-se através de canais, ou portas, que são associados para interconectar os componentes. Cada canal deve especificar uma interface que o componente-alvo deve implementar para poder ser conectado ao canal. Considerando, por exemplo, o componente `AXI4`, que modela um barramento. Ele possui portas que requerem a interface `B_Slave` (definidas pela composição estereotipada com «`sc_port`») e está associado a componentes que implementam esta interface (associações estereotipadas com «`B_Slave`». `AXI4`, por sua vez, implementa a interface `B_Controller`.

Na prática, esta estrutura implica nas seguintes operações sempre que, por exemplo, o processador requisitar uma palavra da memória:

1. `MIPS32` chama o método `read` (definido na interface `B_Controller`) de `AXI4`, utilizando uma porta que está associada ao componente `AXI4`;

2. a implementação de `read` em `AXI4` decodifica o endereço e identifica a porta que está associada ao componente de memória (*e.g.* `Internal_Mem`);

3. um procedimento análogo ao procedimento 1 é executado, retornando das chamadas aninhadas com o dado lido.

Seguindo os conceitos de TLM, a principio não há temporização no sistema. No entanto, a noção de tempo é importante para poder-se ter uma visão do desempenho da aplicação na plataforma real. Nesse sentido, os componentes foram temporizados utilizando a abordagem mostrada na implementação do método `read` de `Internal_Mem`:

```
1  void Internal_Mem::read(unsigned int *data, unsigned int address, int size) {
2      unsigned long int ptr = (unsigned long int) m + (address % mem_size);
3      *data = (unsigned int)ntohl(*(unsigned int*)ptr);
4
5      sc_core::wait(sc_time(CLK_PERIOD, SC_NS));
6  }
```

Uma chamada para `sc_core::wait` faz com que o kernel de simulação SystemC “passe o tempo” sem gerar eventos, o que aconteceria caso um sinal de clock fosse adicionado para sincronizar os componentes. No caso da implementação acima, uma memória interna, implementada como BRAM em

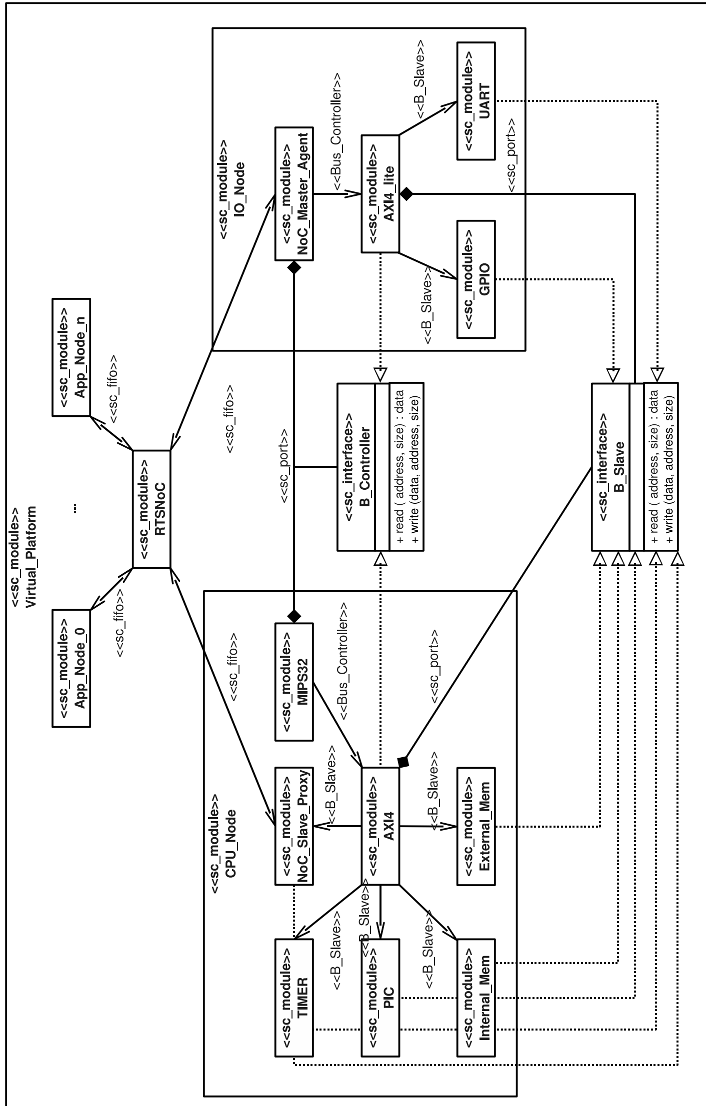

Figura 30: Diagrama estilo UML da plataforma virtual. Os módulos `Virtual_Platform`, `CPU_Node` e `IO_Node` são usados apenas para encapsular outros módulos.

uma FPGA, tem um atraso de apenas um ciclo de clock, logo, `wait` é chamado com um valor referente ao período do clock na plataforma física.

A plataforma virtual aproveita-se do fato de o CatapultC (ferramenta de HLS usada ao longo do trabalho) gerar modelos em SystemC dos componentes implementados em C++. Um wrapper simples é utilizado para conectar esses modelos ao modelo da RTSNoC. No entanto, todos os outros IPs tiveram seus modelos desenvolvidos manualmente a partir das implementações RTL pré-existentes, incluindo um ISS para o MIPS32, que simula o comportamento do softcore Plasma. Nesse sentido, vale a pena mencionar outras abordagens que poderão ser utilizadas em trabalhos futuros. Por exemplo, o trabalho de Lorenz et al. (2012) pode ser utilizado para gerar modelos com anotações de desempenho a partir de implementações RTL já existentes. A linguagem *ArchC* (AZEVEDO et al., 2005) também poderia ser utilizada para obter ISSs dos processadores. ArchC é uma linguagem baseada em SystemC que provê mecanismos para descrever arquiteturas de processadores. A linguagem é acompanhada por um conjunto de ferramentas para a geração de compiladores e modelos em vários níveis de abstração.

6.2 COMUNICAÇÃO ENTRE HARDWARE E SOFTWARE

Na seção 5.4 do capítulo anterior foi apresentado o mecanismo utilizado para viabilizar a comunicação transparente entre hardware e software. A implementação dos canais, proxies e agents mostrados anteriormente pode ser realizada de diferentes formas e depende diretamente da arquitetura de hardware subjacente (*e.g.* usando barramentos, DMA, NoCs) e das funcionalidades do RTOS.

Para ilustrar a implementação de proxies e agents segundo usando o EPOS na plataforma baseada na RTSNoC e como os artefatos da plataforma são mapeados para componentes do framework mostrado na figura 22, será utilizado um exemplo baseado no modelo simples mostrado nas figuras 17 e 19 do capítulo anterior. Assume-se o seguinte particionamento e sequência de operações:

- `C1` em software, `C2` em hardware, e `C3` em software;
- `C1` chama o método `C2::op0`. Dentro de `C2::op0`, `C2` chama `C3::op1`;
- ambos os método possuem um argumento e um valor de retorno.

Considerando as interações acima e as características da plataforma-alvo, a figura 31 ilustra as interações que ocorrem entre os componentes em

software, no nodo de CPU, e em hardware, no nodo que contém a implementação de C2. Cada operação está descrita a seguir:

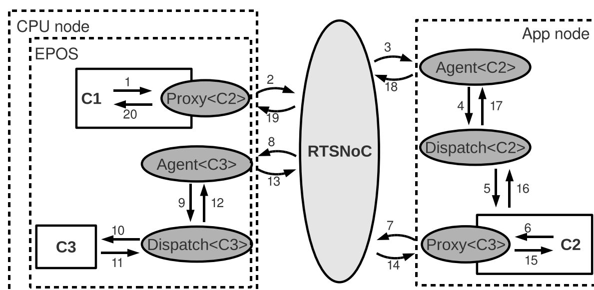

Figura 31: Interações entre proxies e agents em uma chamada do tipo SW->HW->SW

1 C1 chama C2::op0 através do seu proxy;

2 Proxy<C2> serializa as chamadas em pacotes, que são enviados pela NoC; o proxy fica bloqueado esperando o pacote que contem o valor de retorno;

3 o nodo da rede que contém a implementação em hardware de C2 recebe os pacotes;

4 Agent<C2> deserializa os pacotes e encaminha a chamada para o método de despacho implementado em Dispatch<C2>;

5 a invocação de C2::op0 é feita localmente em C2;

6 C2 chama C3::op1 através do seu proxy;

7 Proxy<C3> serializa as chamadas em pacotes, que são enviados pela NoC; o proxy fica bloqueado esperando o pacote que contem o valor de retorno;

8 o SO identifica uma interrupção da NoC e recebe os pacotes; os pacotes são identificados e encaminhados para Agent<C3>;

9 Agent<C3>, que estava bloqueado esperando por pacotes, deserializa os pacotes e encaminha a chamada para o método de despacho implementado em Dispatch<C3>;

10, 11, 12, 13 a invocação é feita localmente em C3; o valor retornado é envidado através da NoC por Agent<C3>; o mesmo volta a ficar bloqueado esperando por novas chamadas;

14, 15 `Proxy<C3>` recebe o pacote contendo o valor de retorno; a chamada à `C3::op1` retorna;

16, 17, 18 `C2::op0` retorna; `Agent<C2>` envia o valor de retorno pela NoC e volta a ficar bloqueado esperando por pacotes contendo uma nova chamada;

19, 20 o SO identifica uma interrupção da NoC e recebe os pacotes; os pacotes são identificados e encaminhados para `Proxy<C2>`, que é desbloqueado e a chamada à `C2::op0` retorna.

Para implementar a sequência de operações em software, um mecanismo de gerenciamento de componentes foi implementado no EPOS. Esses mecanismos são mostrados na figura 32.

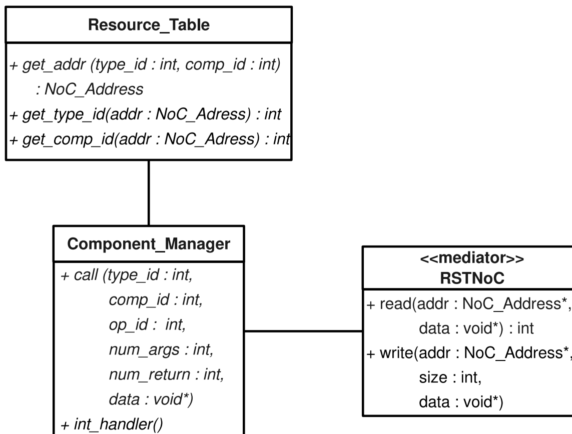

Figura 32: Gerenciamento de componentes no EPOS

O `Component_Manager` tem duas funções: 1) enviar, pela RTSNoC, pacotes contendo chamadas para os componentes em hardware; 2) receber pacotes da RTSNoC e repassar para proxies bloqueados, quando contiverem um valor de retorno, ou agents, quando contiverem dados para uma chamada. Dessa maneira, o `Component_Manager` mantém uma lista de todos os proxies para componentes em hardware e agents para componentes em software. Cada componente é associado a um ID único que é mapeado para um endereço físico na NoC por uma tabela estática de recursos.

Durante as operações **1** e **2**, o `Component_Manager` é requisitado por `Proxy_Common`. O trecho de código abaixo mostra parte da especialização de `Proxy_Common` para a plataforma utilizada:

```
1  template<>
2  class Proxy_Common<RTSNoC, false> {
3  protected:
4      // sem argumento, com retorno
5      template<unsigned int OP, typename RET>
6      RET call_r() {...}
7
8      // um argumento, com retorno
9      template<unsigned int OP, typename RET, typename ARG0>
10     RET call_r(ARG0 &arg0){
11         void *data = Serializer<false>:: serialize (arg0);
12         Component_Manager::call(_type_id, _comp_id, OP, 1, 1, data);
13         return Serializer<false>::deserialize<RET>(data);
14     }
15
16     // dois argumentos, com retorno
17     template<unsigned int OP, typename RET, typename ARG0, typename ARG1>
18     RET call_r(ARG0 &arg0, ARG1 &arg1){...}
19
20     ...
21 };
```

`Proxy_Common` implementa vários métodos que suportam diversos padrões de assinatura. Em cada método, ele serializa os argumentos da chamada (utilizando `Serializer`) e encaminha a mesma para `Component_Manager`. O `Component_Manager` constrói, então, os pacotes e envia para o nodo em hardware utilizando o mediador de hardware da RTSNoC. Caso o método sendo invocado possua valores de retorno, o `Component_Manager` fica bloqueado até que seja identificada uma interrupção da NoC indicando a chegada de pacotes (operações **19 e 20**). Um tratador de interrupção (definido por `Component_Manager::int_handler`) lê todos os pacotes pendentes. Quando um pacote contiver um valor de retorno, o `Component_Manager` (no contexto do tratador de interrupções) procura em uma lista que identifica proxies bloqueados, sinalizando a eles que o retorno foi recebido. Após retornar para o contexto normal de execução, a chamada a `Component_Manager::call` retorne para o proxy do componente com o valor de retorno da chamada.

No lado do hardware, a especialização de `Agent_Common` para `<RTSNOC,true>` implementa as operações **3**, **4** e **16-18**. A sua função nesse ponto é definir o ponto de entrada que é usado para inferir a interface do IP resultante em hardware. A figura 33 mostra a assinatura do método que define o ponto de entrada.

Para a ferramenta usada (CatapultC) o ponto de entrada é definido usando objetos do tipo `ac_channel`. Esse tipo é uma abstração disponibilizada pela ferramenta que define operações bloqueantes de leitura e escrita. Essas operações são mapeadas pela ferramenta para um protocolo de handshaking de duas fases que pode ser facilmente atrelado a uma das portas da NoC. O restante da implementação do método `top_level` segue o processo descrito na seção 5.4 do capítulo anterior, com o aspecto `Dispatch` sendo responsável por fazer a invocação local utilizando os dados repassados pelo agent.

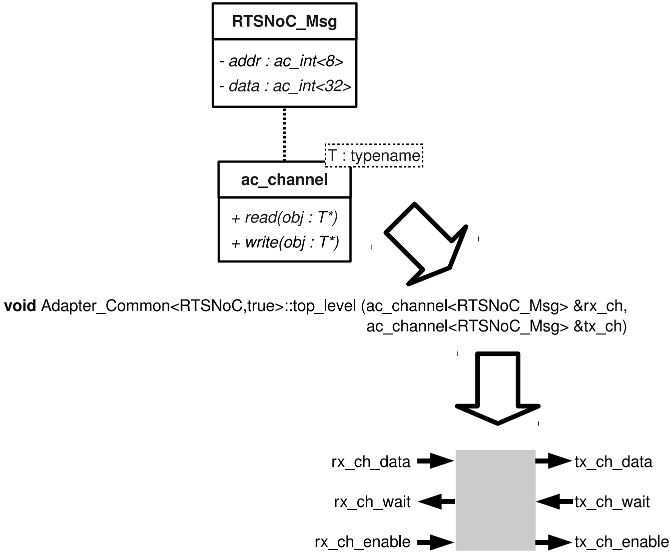

Figura 33: Definição da interface para os agents em hardware

A implementação de proxies em hardware (operações **6, 7, 14-15**) e dos agents em software (operações **8-13**) é análoga às suas versões em software e hardware. A diferença está no fato de que não há um gerenciador de componentes global. A montagem dos pacotes a serem enviados é feita localmente em cada proxy na implementação de `Proxy_Common<RTSNOC, true>`. No caso do exemplo apresentado, `Proxy<C3>` está no mesmo nodo que `C2` e o seu agent, logo ambos usam os mesmos objetos `ac_channel` para enviar pacotes pela NoC. No caso do agent em software, a diferença está no fato de que o mesmo recebe dados via chamadas sucessivas ao método `Agent_Common<RTSNoC,false>::top_level`. Essas chamadas são feitas no contexto do tratador de interrupções e, uma vez que todos os dados necessário são recebidos, `top_level` faz a chamada local para o método requisitado.

6.3 SUMÁRIO DO FLUXO DE IMPLEMENTAÇÃO

Apesar de o objetivo desse trabalho não ser propor um fluxo completo para o desenvolvimento de sistemas embarcados, para a implementação e avaliação das ideias propostas foi utilizado um fluxo de implementação simples, que apenas integra os mecanismos propostos com o conjunto de ferramentas

disponível. A figura 34 mostra os passos utilizados para sair de uma descrição unificada em C++ e obter componentes integrados na plataforma. Esses passos são descritos abaixo.

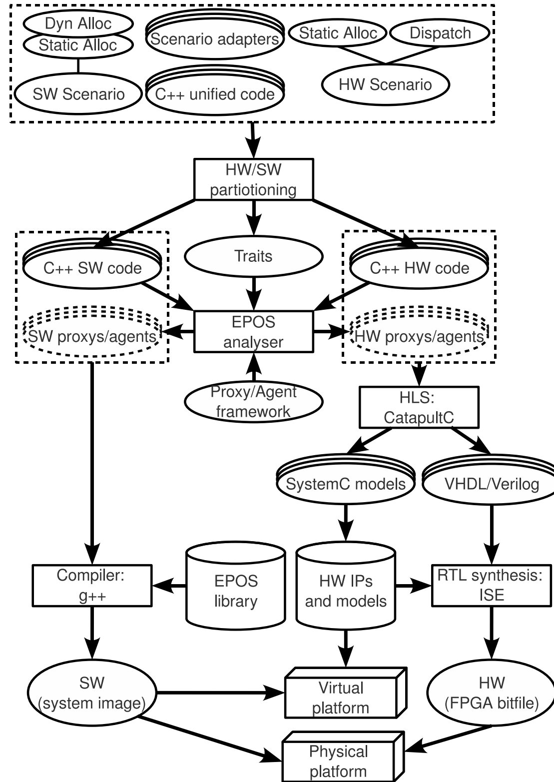

Figura 34: Passos utilizados para obter a implementação final dos componentes a partir de código C++ unificado

6.3.1 Elementos de entrada

Os primeiros passos mostram os artefatos que fazem parte da abordagem proposta. Aspectos, cenários, componentes e os seus respectivos adaptadores compõe as entradas para o fluxo de implementação. Uma vez que o particionamento entre hardware e software é definido, obtêm-se os `Traits` que definem a configuração do sistema. As implementações específicas para hardware e

software são obtidas automaticamente. Isso se deve ao fato de o processo de adaptação e mapeamento entre os componentes ser definido utilizando metaprogramação na própria linguagem de entrada.

Como ressaltado no Capítulo 5, seção 5.5, o particionamento entre hardware e software e a definição dos `Traits` é atualmente manual. No entanto, em trabalhos futuros, um algorítimo de exploração automática do espaço de projeto, como o proposto por Cancian (2011), pode ser empregado nesta fase com o objetivo de automatizar o particionamento entre hardware e software e a definição dos `Traits`.

6.3.2 Geração de proxies e agents

Devido à hierarquia modular do framework de comunicação, a obtenção de proxies e agents para cada componente é um processo muito simples que pode ser facilmente automatizado. O trecho de código abaixo mostra a implementação desses artefatos para o componente `C2` mostrado nos exemplos anteriores:

```
1  template<>
2  class Proxy<C2> : public Proxy_Common<Traits<System>::Platform,
3                                        Traits <C2>::hardware>
4  {
5      unsigned int op0(unsigned int arg){
6          return call_r<OP0_ID,unsigned int>(arg);
7      }
8
9  }
10
11 template<>
12 void Dispatch<C2>::dispatch(int op, unsigned int *data){
13     switch(op){
14     case OP0_ID:
15         unsigned int tmp =
16         static_cast<Scenario_Adapter<C2>*>(this)->op0(data[0]);
17         data[0] = tmp;
18         break;
19     default: break;
20     }
21 }
```

O proxy apenas redefine os métodos da interface do componente e faz as chamadas utilizando os mecanismos implementados na sua classe-base. No caso do agent, a implementação especifica para o componente está, na verdade, dentro do aspecto `Dispatch`. Nesse caso, é possível fazer a especialização apenas do método de despacho.

Apesar de esses mecanismos serem atualmente implementados manualmente, não há impedimentos para que esse passo venha a ser automatizado. O ferramental que integra o EPOS, por exemplo, inclui um analisador sintático que pode ser usado par obter as assinaturas dos métodos de cada compo-

nente (SCHULTER et al., 2007). Essa possibilidade de automatização está ilustrada no fluxo da figura 34.

6.3.3 Geração de software e hardware

Durante este trabalho, utilizou-se o CatapultC (Calypto Design Systems, 2011) para fazer a síntese de hardware. A entrada para o processo consiste em código C++ acompanhado das diretivas de síntese. Como explicado na seção 5.1.4 do capítulo anterior, essas diretivas são utilizadas para guiar a ferramenta no processo de geração da microarquitetura em RTL a partir do algoritmo em C++. O resultado desse processo consiste em descrições no nível RTL em VHDL ou Verilog e modelos em SystemC para simulação.

O processo geração do software é mais simples. Os componentes em software são compilados juntamente com o EPOS, gerando o binário final.

6.3.4 Geração da plataforma final

As descrições RTL e SystemC são associadas com o restante dos componentes (*e.g.* nodos de CPU/IO, RTSNoC) que compõe a plataforma física e virtual, respectivamente. Esse processo é, atualmente, manual. No entanto, ele consiste em apenas codificar o *top level* da implementação da plataforma, definido a associação entre os nodos e as portas da RTSNoC, sendo passível de automatização.

7 AVALIAÇÃO

Este capítulo tem dois objetivos: 1) fazer uma avaliação sintética da plataforma descrita no Capítulo 6, na qual são apresentadas noções mais genéricas em relação a eficiência da implementação obtida; 2) aplicar os mecanismos orientados a aspectos em estudos de caso para demonstrar que as implementações unificadas e adaptadas como descrito no Capítulo 5 são comparáveis às dedicadas para hardware ou software, em termos de área e desempenho. Estas duas questões são abordadas nas seções 7.1 e 7.2, respectivamente.

Todos os experimentos deste capítulo foram conduzidos utilizando a plataforma ML605 da da Xilinx (Xilinx, 2012a). Esta plataforma contém uma FPGA *Virtex6 XC6VLX240T* que foi utilizada para implementar a arquitetura de hardware descrita no Capítulo 6. A tabela 3 sumariza as características do conjunto de ferramentas utilizadas para a geração do hardware e do software.

Tabela 3: Ferramentas utilizadas para a avaliação

	Software	Hardware			
Alvo	MIPS I ISA	Xilinx Virtex6 XC6VLX240T			
		6-input LUTs	Flip-flops	36 Kb BRAM	DSP slices
		150720	301440	416	768
Ferramenta	gcc 4.0.2	C++ -> RTL: CatapultC UV 2011 RTL -> FPGA: Xilix ISE 13.4			
Configurações das ferramentas	Otimizações nível *02*	Minimizar a área do circuito tendo como alvo uma frequência de 100 MHz			

Para geração da plataforma virtual, foi utilizando o *gcc 4.3.3* e a versão 2.2 da biblioteca SystemC. As simulações RTL foram executadas usando o *ModelSim 6.5c* (Mentor Graphics, 2011). Todas as simulações foram executadas em um sistema *i686* (Intel Core2 Quad CPU Q9550 @ 2.83GHz) com *Linux kernel 2.6.28*.

7.1 AVALIAÇÃO DA PLATAFORMA

Foram feitos alguns experimentos com o objetivo de prover uma visão geral da plataforma desenvolvida. Primeiramente, foi medido o tempo de execução de algumas aplicações sendo executas no SoC. Para essa avaliação

inicial, foram utilizados *benchmarks* do *MiBench* (GUTHAUS et al., 2001). O *MiBench* é um conjunto de benchmarks que reúne diversas aplicações comuns em sistemas embarcados. Os seguintes aplicativos foram selecionados:

Dijkstra: constrói um grafo e calcula o caminho mais curto entre cada par de nodos no grafo utilizando o algoritmo de Dijkstra. O algoritmo de Dijkstra é bem conhecido e computa uma solução em um tempo $O(n^2)$;

SHA: algoritmo usado em funções criptográficas. Gera um *hash* de 160 bits a partir de uma entrada;

FFT: calcula uma transformada discreta de Fourier de 4096 pontos utilizando uma implementação da FFT;

Susan: um conjunto de algoritmos de reconhecimento de padrões em imagens. Foram utilizados os algoritmos de reconhecimentos de cantos (*S-Corners*) e bordas (*S-Edges*) aplicados em uma imagem em formato cru de $76x95$.

A tabela 4 mostra o tempo de execução das aplicações sendo executadas tanto na plataforma virtual quanto na plataforma física, assim como o tempo de simulação de ambas as plataformas. Tanto a plataforma virtual quanto a física foram simuladas com uma frequência de clock de 100MHz.

Tabela 4: Desempenho e tempo de simulação das plataformas

Benchmark	Tempo de execução (ms)		Tempo de simulação	
	Plat. virtual	Plat. física	Plat. virtual(s)	RTL(min)
SHA	33.31	36.21	6.9	32.1
Dijkstra	2074.86	2290.38	555.3	2493.3
FFT	5180.96	6042.62	1350.4	6231.9
S-Corners	42.38	47.96	8.9	42.2
S-Edges	54.59	61.90	11.5	52.1

Os resultados mostram que a diferença média entre o tempo de execução na plataforma real e na plataforma física é de apenas 11%, enquanto que o tempo de simulação é mais de 275 vezes maior nas simulações RTL da plataforma real. Esses dados ilustram a eficiência do uso de plataformas virtuais para estimativas iniciais de desempenho do sistema. A diferença de 11% no tempo de execução, no entanto, pode inviabilizar o uso da plataforma virtual em aplicações de tempo-real críticas. Apesar de não ser o objetivo deste trabalho o desenvolvimento de plataformas virtuais com estrita precisão temporal, acredita-se que é possível diminuir essa diferença encontrada. Uma

grande parte dessa diferença vem, principalmente do ISS desenvolvido para a arquitetura MIPS32, que ainda pode ser significativamente melhorado de modo que o seu comportamento seja mais próximo do softcore Plasma usado na implementação em RTL.

Como referência, as tabelas 5 e 6 mostram os resultados da síntese da plataforma para uma FPGA e o consumo de memória do sistema operacional EPOS.

Tabela 5: Resultados da síntese dos componentes da plataforma em uma FPGA

	RTSNoC (1 router)	Nodo de CPU	Nodo de IO
Freq. máxima	151.2 MHz	105.9 MHz	290.5 MHz
6-input LUTs	2256	7775	537
Flip-flops	689	6834	409
36Kb BRAM	0	2	0
Uso médio de área	0.30%	1.90%	0.11%

Tabela 6: Utilização de memória do sistema operacional EPOS

	Tamanho (bytes)
Código	9756
Dados	381
Total	10137

O uso de área da FPGA é mostrado em termos do número de *Lookup Table*s (LUTs), *Flip-flop*s (FFs) e blocos de memória SRAM utilizados. Devido à ausência de uma métrica única para mostrar a área usada por um componente na FPGA, a linha *"Uso médio de área"* da tabela 6 mostra a média aritmética de cada recurso utilizado ponderado pelo seu total na FPGA. O valor resultante procura estimar o total de área da FPGA (em %) necessário e é usado nesse trabalho como uma métrica unificada para estimativa de área.

7.1.1 Eficiência dos proxies e agents

Nesta seção, foi feita uma avaliação parcial da atual implementação dos mecanismos de comunicação propostos. A infraestrutura implementada foi avaliada em termos de latência e utilização de recursos.

As tabelas 7 e 8 mostram a área total utilizada em software (tamanho do código) e hardware (recursos da FPGA). Os componentes foram sintetiza-

dos/compilados utilizando as mesmas ferramentas e configurações descritas na tabela 3. Nesta avaliação, no entanto, a frequência do SoC foi fixada em 100 MHz, tendo em vista frequência máxima obtida pelo nodo da CPU (105.9 MHz). Dessa forma, as ferramentas de síntese foram configuradas para minimizar a área do circuito considerando a frequência-alvo.

Tabela 7: Memória utilizada pelos proxies/agents em software. Valores em *bytes*.

	Proxy	Agent	*Component_Manager*
Código	64	364	2422
Dados	4	44	74
Total	68(46)	408(136)	2496

Tabela 8: Recursos da FPGA utilizados pelos proxies/agents em hardware

	Proxy	Agent
6-input LUTs	45(10)	61(16)
Flip-flops	89(2)	147(4)
36Kb BRAM	0	0
Uso médio de área	0.02%	0.03%

Em ambas as tabelas, proxies a agents suportam um único método com um argumento de 4 bytes, enquanto os valores entre parênteses indicam quantos recursos extras são necessários para suportar um método adicional. Por exemplo, um agent em hardware para um componente com dois métodos requer 77 $(61+16)$ LUTs para ser implementado.

Para verificar a significância dos valores obtidos em relação a outros trabalhos, a tabela 9 compara os resultados obtidos com dados extraídos diretamente de outros trabalhos. Estes trabalhos (RINCÓN et al., 2009; GRACIOLI; FRÖHLICH, 2010) são descritos em mais detalhes no capítulo 3 e proveem uma infraestrutura de proxy/agents similar à deste trabalho. O trabalho de Gracioli e Fröhlich (2010) também é baseado no sistema operacional EPOS e propõe os mecanismos apenas para o domínio do software. O trabalho de Rincón et al. (2009) foca principalmente na implementação de agents em hardware no nível RTL, sendo que resultados referentes aos mecanismos em software não são apresentados pelos autores.

Na comparação do software, o resultado obtido foi 10% menor que o do trabalho de referência. Levando em conta que este tinha como alvo uma arquitetura de 8 bits, que requer menos memória para armazenar instruções,

Tabela 9: Uso de área para um proxy e um agent de um componente que define dois métodos com um parâmetro de entrada e um parâmetro de retorno. No caso do software, o resultado inclui o consumo de memória do `Component_Manager`

Trabalho	Tipo de plataforma	HW agent (LUTs/FFs)	SW proxy (código+dados)
Este trabalho	32-bits MIPS / RTSNoC	80 / 154	2636
Rincón et al. (2009)	32-bits PPC / IPIF bus	202 / 354	-
Gracioli e Fröhlich (2010)	8-bits AVR	-	2911

pode-se concluir que a infraestrutura proposta no presente trabalho resulta em um baixo sobrecusto em termos de consumo de memória.

No lado do hardware, a diferença foi ainda mais significativa, mas neste caso ela pode ter sido induzida pela diferença entre as infraestruturas de comunicação dos trabalhos. Os agents propostos neste trabalho possuem uma interface simples para serem acoplados na RTSNoC, enquanto os agents de Rincón et al. (2009) possuem uma interface mais complexa compatível com o protocolo IPIF (XILINX, 2005). Os resultados também são referentes a uma FPGA com LUTs de 4 entradas, enquanto neste trabalho foi usada uma FPGA com LUTs de 6 entradas, o que resulta em um circuito ligeiramente menor. No entanto, apesar destas diferenças, é importante ressaltar que foi possível obter, a partir de um framework metaprogramado em C++, resultados comparáveis a uma implementação feita diretamente em RTL.

O gráfico da figura 35 mostra o número de ciclos necessários para a invocação de um método entre hardware e software, em ambas as direções (`SW->HW` e `HW->SW`). Como referência, também foi incluído o caso "sem sobrecusto", no qual ambos os componentes são implementados no mesmo domínio, mas de modo que um não tenha sido "absorvido" pelo outro durante o processo de compilação/síntese. Em software, isso significa que as chamadas não são feitas *inline*, e, em hardware, significa uma interação entre componentes em nodos distintos da NoC.

Para a interação SW->HW, uma invocação de um método com um único argumento de 4 bytes leva 1769 ciclos, enquanto cada argumento adicional aumenta a latência em cerca de 135 ciclos. No caso oposto, a mesma invocação leva 2015 ciclos, e cada argumento adicional adiciona cerca de 500 ciclos. A figura 35 também mostra a latência de uma invocação similar no trabalho de Rincón et al. (2009). Nesse caso, o trabalho relacionado apresenta uma latência significativamente inferior. Isso é o resultado do baixo desempenho do `Component_Manager` em software, influenciado pelo desempenho do softcore Plasma, que é significativamente menos poderoso que o proces-

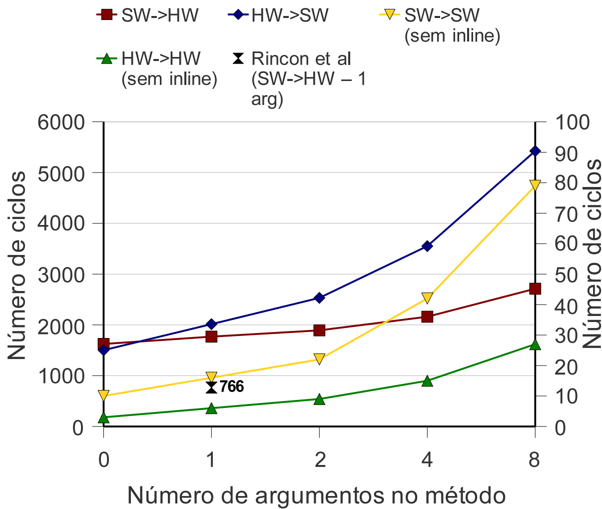

Figura 35: Latência da invocação de um método através de diferentes domínios. A comunicação `SW->SW` e `HW->HW` está alinhada com o eixo Y da direita.

sador utilizado no trabalho relacionado (um IBM PowerPC integrado com a FPGA). O Plasma é um softcore de baixo desempenho até mesmo quando comparado a outras implementações na mesma categoria (KRANENBURG, 2009), contudo optou-se pela sua utilização devido ao suporte pré-existente do EPOS para a arquitetura MIPS32. De modo a melhorar os resultados obtidos em implementações futuras, está previsto o porte do EPOS para a arquitetura MicroBlaze (XILINX, 2012) e a utilização do softcore aeMB (AESTE, 2012), que oferece uma melhora significativa de área e desempenho em relação ao Plasma (KRANENBURG, 2009).

7.2 ESTUDOS DE CASO

Para avaliar a abordagem proposta, foi feito uma análise de uma aplicação PABX e algumas de suas partes foram reimplementadas utilizando as diretrizes de implementação propostas. O componente principal é uma matriz de comutação que controla a interconexão entre vários canais de entrada/saída de dados. Estes canais são conectados a linhas de telefone (por meio de um conversor Analógico-Digital (AD)/Digital-Analógico (DA)), geradores de tons, e detectores de tons. O sistema também suporta a transmissão de dados de voz

através da rede *Ethernet*. A figura 36 mostra um diagrama de blocos da parte digital do sistema PABX da forma como ele foi implementado como um SoC na FPGA. Os componentes selecionados para serem implementados utilizando C++ unificado estão destacados e aparecem tanto em hardware quanto em software, pois eles podem ser migrados de um domínio para o outro dependendo do particionamento final de hardware e software. Estes componentes são descritos em mais detalhes a seguir:

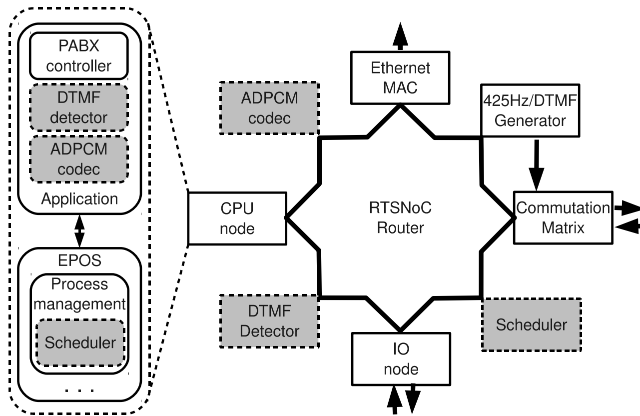

Figura 36: Diagrama de blocos do sistema PABX

Escalonador do EPOS: o estudo de caso descrito no capítulo 5. Foram avaliadas as três versões do escalonador: a versão original, apenas compilável para software (seção 4.1); a primeira versão sintetizável (seção 5.1); a versão unificada, adaptada pelo adaptador de cenário para software e hardware (seção 5.3.1). Na avaliação da versão unificada, foi utilizado o aspecto de alocação dinâmica na adaptação para software e alocação estática na adaptação para hardware. Em todos os casos, foi assumido um número máximo de oito threads no sistema e um critério de escalonamento do tipo *round-robin*. O código fonte das três implementações desse estudo de caso pode ser visto no apêndice A.

Codec ADPCM: um Codificador/Decodificador (codec) ADPCM padrão IMA (Interactive Multimedia Association (IMA), 1992) que é usado para reduzir o trafego de dados transmitido pela rede. O codec faz compressão de dados usando um algoritmo adaptativo (*Adaptative Differential Pulse-Code Modulation*) que converte amostras de 16 bits de um sinal em amostras de 4 bits. A implementação unificada em C++ foi feita a partir de uma implementação pré-existente na linguagem C. A figura 37 mostra a estrutura do codec. Os algoritmos de codificação e decodificação são independentes e implementados em classes diferentes. Ambos os algoritmos, no entanto, compartilham as

mesmas LUTs que são definidas em uma classe comum (*ADPCM_Common*). A classe *ADPCM_Codec* apenas encapsula ambas as implementações na forma de um único componente codec. O código fonte das três implementações desse estudo de caso pode ser visto no apêndice A.

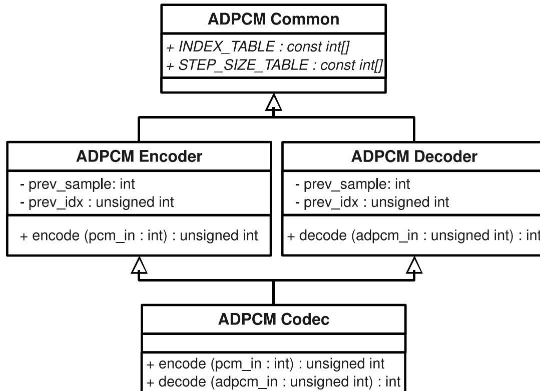

Figura 37: Definição do componente ADPCM

Detector de DTMF: um detector de DTMF (*Dual-Tone Multi-Frequency*) é um dos blocos mais básicos de um sistema PABX. O detector DTMF recebe amostras de sinais das linhas telefônicas conectadas na central e detecta os tons emitidos. A figura 38 mostra a sua definição. O ponto de entrada da implementação original na linguagem C era uma única função que recebia um ponteiro para um quadro de amostras e retornava o tom detectado. Esta foi refatorada para obter a implementação unificada que define duas operações: *add_sample* atualiza uma amostra no quadro atual; e *do_dtmf* implementa o pseudocódigo na figura 38 para efetuar a detecção. Para cada tom, é usado o algoritmo de *goertzel* para verificar se o quadro de amostras contém ou não os componentes de frequência que definem o tom, então LUTs utilizadas para analisar o resultado e retornar um valor ASCII representando o tom. O código fonte das três implementações desse estudo de caso pode ser visto no apêndice A.

7.2.1 Resultados

Para demonstrar que as implementações unificadas são comparáveis às dedicadas em termos de eficiência, os componentes adaptados para software usando o adaptador de cenário são comparados com as suas respectivas im-

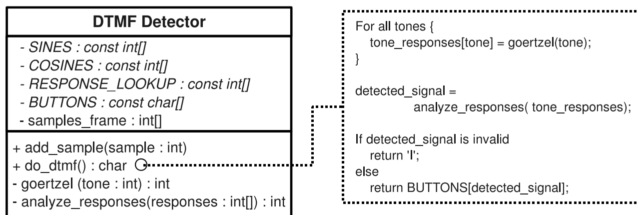

Figura 38: Definição do componente DTMF

plementações originais na linguagem C (C++ no caso do escalonador). Os componentes adaptados para hardware usando o adaptador de cenário também são comparados com as suas respectivas versões que foram feitas sob medida para uma ferramenta de HLS.

7.2.1.1 Software

A tabela 10 e figuras 39 e 40 comparam as implementações originais, apenas para software, com as implementações adaptadas para software usando um adaptador de cenário. Os componentes foram compilados usando as ferramentas e configurações descritas na tabela 3.

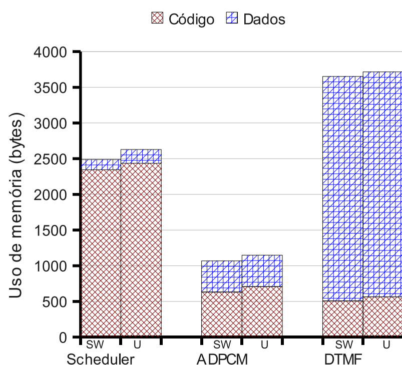

Figura 39: Uso de memória de C/C++ apenas para software (`SW`) *vs.* C++ unificado e adaptado para software (`U`)

A figura 39 mostra um aumento médio de cerca de 4.9% no uso de memória. No caso do escalonador este aumento pode ser explicado pela

Tabela 10: Tempo de execução do C/C++ apenas p/ software *vs.* C++ unificado

Componente		Tempo de execução (μs)	
		Software	Unificado
Scheduler	insert	6.0	6.0
	remove	2.9	3.3
	suspend	3.0	3.1
	resume	6.0	6.0
	choose	8.4	8.5
ADPCM	encode	4.2	4.2
	decode	3.4	3.6
DTMF	do_dtmf	5878.3	6182.9

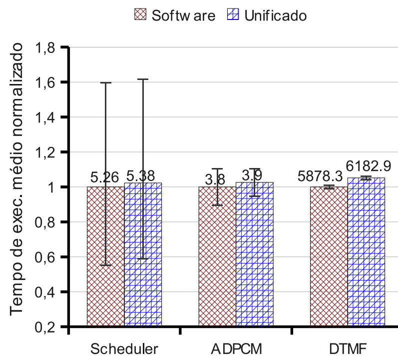

Figura 40: Tempos médio de execução normalizado. Os valores absolutos são mostrados acima das respectivas barras.

introdução dos *option types* e o uso de um mecanismo mais genérico para alocação de memória (os aspectos *Static/Dyn Alloc*). No restante dos casos, a maior parte do sobrecusto vem do código adicional que foi necessário para encapsular o comportamento dos componentes em classes mais reusáveis com uma interface de métodos mais clara.

A tabela 10 e a figura 40 mostram o tempo de execução de cada operação dos componentes e comparam os valores médios. O tempo de execução do escalonador e do codec ADPCM foi cerca de 2.5% maior nas implementações unificadas. Para o detector de DTMF, a diferença aumenta para 5%. O detector de DTMF original utilizava apenas uma chamada para o método `do_dtmf` para analisar um quadro de amostras, enquanto a versão refatorada requer várias chamadas para o método `add_sample` antes de executar a mesma tarefa, o que resultou em um aumento mais significativo do tempo de execução.

A figura 40 também mostra barras de erro que indicam a variação encontrada nos tempos de execução. No caso do escalonador, o tempo pode variar significativamente de acordo com o número de threads na fila de escalonamento em um dado momento.

7.2.1.2 Hardware

A tabela 11 e figuras 41 e 42 comparam as implementações feitas sob medida para hardware com as implementações adaptadas para hardware usando um adaptador de cenário. Os componentes foram sintetizados usando as mesmas ferramentas e configurações descritas na tabela 3.

Tabela 11: Uso de recursos da FPGA do C++ apenas p/ hardware *vs.* C++ unificado

Recurso	Scheduler		ADPCM		DTMF	
	Hardware	Unificado	Hardware	Unificado	Hardware	Unificado
6-input LUT	2119	2540	524	615	387	443
Flip-flops	1849	2766	208	368	331	431
36Kb BRAM	0	0	1	1	2	1
DSP slice	0	0	0	0	6	6

A tabela 11 mostra a quantidade de recursos utilizados na FPGA para cada componente; a figura 41 plota o *uso médio de área* seção 7.1). Os resultados variam significativamente em cada estudo de caso. A área aparentemente menor na implementação unificada do detector de DTMF é resultado de um mapeamento diferente dos recursos entre os blocos internos de RAM e flip-

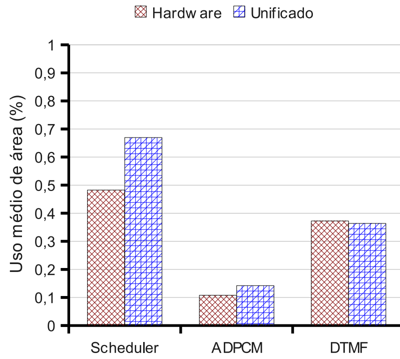

Figura 41: Uso médio de área da FPGA

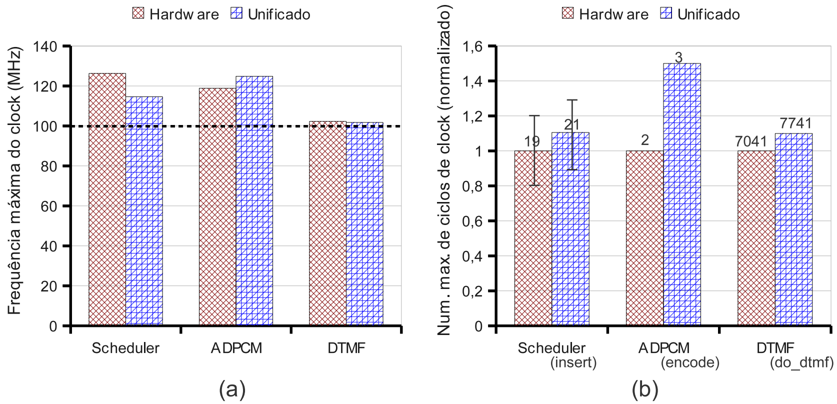

Figura 42: Desempenho do C++ apenas p/ hardware *vs.* C++ unificado. Na figura (b), os valores são normalizados para comparar todos os componentes na mesma escala.

flops, o que gerou um uso médio de área menor. Nos outros dois casos, a implementação unificada do escalonador e do ADPCM exigiu cerca de 39% e 27% mais área, respectivamente. No caso do ADPCM, as implementações unificadas e as otimizadas para hardware são, internamente, muito similares. Dessa forma, conclui-se que a área adicional foi resultado do uso do aspecto *Dispatch* e dos proxies/agents. Estes artefatos implementam um mecanismo genérico para analisar e disparar chamadas de métodos, enquanto na implementação em hardware, a interface é feita sob medida para o componente. Esse sobrecusto pode ser considerado significativo e aumenta com o número de operações que devem ser suportadas.

A figura 42a mostra a frequência máxima atingida por cada componente. Todas as implementações atingiram frequência similares e acima do limite de 100 MHz estabelecido. A figura 42b mostra o número máximo de ciclos de clock necessários para executar uma operação. Para as implementações unificadas do escalonador e do ADPCM, o sobrecusto ocasionado pelo uso dos agents foi proporcional ao número de argumentos em cada operação (dois ciclos para `Scheduler::insert` e um ciclo para `ADPCM_Codec::encode`. O alto sobrecusto absoluto na implementação do detector de DTMF é o resultado das várias invocações de `DTMF_Detector::add_sample`, necessárias para encher o seu buffer interno com um quadro de amostras. Na implementação sob medida para hardware, essa operação é feita através de uma interface de *streaming*.

7.2.1.3 SoC com diferentes particionamentos

Para concluir a análise, foi avaliado o impacto que os mecanismos propostos tiveram no sistema final. A tabela 13 mostra o uso de memória e de recursos da FPGA considerando os seguintes particionamentos do SoC PABX: todos os estudos de caso em software; todos os estudos de caso em hardware; e um particionamento híbrido, no qual apenas o componente `Scheduler` está em software. Na tabela 13, os valores para a coluna `Sistema` correspondem à área total (memória/FPGA) subtraída pela área dos componentes no respectivo domínio. Por exemplo, o consumo de memória em software mostrado na coluna `Sistema` para um determinado particionamento é igual ao valor mostrado na coluna `Total`, subtraído pelo consumo de memória de todos os componentes implementados em software naquele particionamento. Este valor permite que seja analisado o impacto dos mecanismos de gerenciamento de componentes no SoC completo. Como referência, a tabela mostra 12 a utilização de recursos dos outros IPs do SoC PABX.

Como pode ser visto na tabela 13, mover todos os casos para hardware

Tabela 12: Área utilizada pelos outros IPs do sistema

	Cmm. matrix	425Hz/DTMF gen.	Eth MAC
6-input LUTs	93	519	3424
Flip-flops	108	544	4967
36 Kb Block RAM	0	0	0
Uso médio de Área	0.03	0.13	1.11

Tabela 13: Uso de memória e recursos da FPGA do SoC. No particionamento híbrido, apenas o componente `Scheduler` está em software.

Particionamento	Memória (código+dados)		Uso médio de área da FPGA	
	Total	Sistema	Total	Sistema
Tudo em SW	19553 b	12201 b	3.70%	3.70%
Tudo em HW	16479 b	16479 b	5.05%	3.75%
Híbrido	19093 b	16293 b	4.24%	3.74%

reduziu a memória necessária. Na comparação entre os valores na coluna `Sistema` pode-se observar que, no particionamento `Tudo em HW`, a introdução do `Component Manager` e dos proxies para os três componentes resultou num sobrecusto de 4278 bytes. Após mover o escalonador de volta para o software, no particionamento `Híbrido`, este sobrecusto é reduzido para 4092. Desses dois valores, pode-se obter o custo final do proxy do escalonador: 186 bytes.

No lado do hardware, a área utilizada na FPGA quando todos os casos estão em hardware aumenta de 3.7% para cerca de 5%. Fazendo uma análise análoga a anterior, pode-se verificar que o sobrecusto do agents nos componentes em hardware representa apenas 0.05% da área total do dispositivo, resultando em um impacto negligenciável no sistema como um todo.

8 CONSIDERAÇÕES FINAIS

De uma maneira geral, nos últimos anos, o desenvolvimento de sistemas embarcados tem sido deslocado para o nível de sistema, no qual as funcionalidades de um componente são descritas de forma que a sua implementação em hardware ou software possa ser inferida da forma mais automatizada possível. As ferramentas EDA recentes avançam de forma significativa nesse sentido, viabilizando a descrição de hardware em linguagens como C e C++. No entanto, a maioria dos trabalhos já desenvolvidos foca principalmente nos processos de HLS e em ferramentas de integração para o fluxo de projeto como um todo, mas sem definir uma metodologia clara para projetar os componentes em si no nível de sistema.

De fato, focar apenas na evolução do ferramental resulta em várias dificuldades. Como já mostrado no Capítulo 3, a integração de soluções propostas em diferentes fluxos de projeto seria benéfica, mas esta é uma tarefa difícil devido às diferenças significativas que existem entre os modelos de entrada e os modelos intermediários gerados pelos processos de refinamento de cada ferramenta. Nesse cenário, definir modelos com semânticas mais fortes pode potencialmente diminuir a complexidade de integração, além de possibilitar implementações no nível de sistema suscetíveis a uma gama maior de fluxos de projeto e ferramentas.

O trabalho descrito nesta dissertação avança nesse sentido. Esta dissertação explorou a aplicação dos conceitos de ADESD com o objetivo de produzir implementações unificadas de componentes de hardware e software. As características específicas de ambos os domínios foram identificadas e separadas em *aspectos* que são aplicados às implementações unificadas apenas nas etapas finais de geração do sistema. A principal contribuição deste trabalho em relação aos demais está no fato de que, tanto *a descrição unificada dos componentes, quanto os programas de aspecto que adaptam os mesmos para hardware ou software, são definidos utilizando a linguagem C++*. Além disso, *os mecanismos de aplicação de aspectos são definidos via metaprogramação estática*, utilizando funcionalidades intrínsecas da própria linguagem. Dessa forma, a extração de implementações em hardware ou software a partir de uma implementação em nível de sistema (a *implementação unificada* em C++) é direta e se dá através de transformações no nível da linguagem, utilizando semânticas definidas pelo projetista que são independentes de fluxos de projeto e ferramentas específicas.

Com o objetivo de validar a abordagem proposta, ao longo do trabalho também foi desenvolvida uma plataforma flexível baseada em uma NoC, focando na implementação de SoCs em FPGAs. Esta plataforma foi utilizada

para demonstrar como componentes de hardware e software podem ser facilmente integrados em um fluxo de projeto baseado em um processo de HLS. Essa integração se dá por meio de um mecanismo de invocação remota que permite a comunicação transparente entre hardware e software. Foi também demostrado como estes mecanismos podem ser isolados em um framework metaprogramado em C++ e integrados aos mecanismos de incorporação de aspectos.

Para avaliar os artefatos propostos, alguns blocos funcionais de um sistema PABX foram reimplementados na forma de componentes unificados, seguindo os preceitos de ADESD descritos nesse trabalho: o escalonador de tarefas do EPOS, um codec ADPCM e um detector de DTMF. As implementações unificadas foram comparadas com implementações codificadas especificamente para hardware e software. O resultado dessa comparação mostrou que a flexibilidade e reusabilidade proporcionada pela abordagem unificada não ocasionou um impacto significativo na eficiência do sistema.

8.1 LIMITAÇÕES

Do ponto de vista conceitual, o uso de C++, da forma como foi proposta neste trabalho, não exime o projetista da necessidade de considerar os possíveis domínios de implementação para os seus componentes. Em alguns casos, o algoritmo em C++ deve ser implementado de forma a expressar algumas questões arquiteturais que se espera obter no hardware final, caso contrário, não seria possível utilizar síntese de alto nível para obter resultados comparáveis aos que podem ser obtidos através de codificação manual em RTL. Essa questão é ligeiramente minimizada em trabalhos que usam abstrações num nível mais alto (*e.g.* SpecC no SCE (DÖMER et al., 2008) e SystemMOC no SystemCoDesigner (KEINERT et al., 2009)).

Outra questão relevante está nas possíveis dificuldades no uso de metaprogramação utilizando templates. A sintaxe do C++ para definição de templates nem sempre favorece a legibilidade do código e, além disso, a depuração de metaprogramas pode ser um processo trabalhoso. Como os metaprogramas são executados internamente pelo compilador C++, para depura-los de forma iterativa seria necessário um depurador para o processo de compilação em si. Para identificar erros nos metaprogramas, o desenvolvedor fica limitado à análise manual do código fonte e às mensagens de erro emitidas pelo compilador.

Também foram identificadas algumas limitações do ponto de vista técnico. Estabelecer a comunicação de componentes apenas através de chamadas de métodos autocontidas (*e.g.* não permitir o uso de memória compartilhada entre os componentes) não favorece componentes com interfaces que requerem

alta vazão de dados, como foi o caso do detector de DTMF. Novos mecanismos devem ser considerados para oferecer um suporte mais adequado para aplicações com uma alta vazão de dados entre os componentes. A principal preocupação com esse aspecto está relacionada com a integração de interfaces de *streaming* e mecanismos de DMA com a abordagem orientada a objetos proposta nesse trabalho.

8.2 TRABALHOS FUTUROS

Uma primeira vertente de trabalhos futuros diz respeito à melhoria dos mecanismos propostos com o objetivo de eliminar as limitações identificadas. Nesse sentido, o próximo passo é a aplicação da abordagem proposta em um número maior de estudos de caso. Isto trará novas perspectivas sobre a aplicabilidade dos mecanismos desenvolvidos, contribuindo no processo de obtenção de soluções para as questões relacionadas, por exemplo, à comunicação mais eficiente entre componentes com uma alta vazão de dados. Além disso, através de novos estudos de caso, espera-se identificar outros aspectos de hardware e software que possam ser separados e encapsulados usando os mecanismos propostos.

Outra importante vertente de trabalhos futuros gira em torno da integração dos mecanismos desenvolvidos em um fluxo de projeto completo, suportado por um conjunto de ferramentas que permita a geração automática do sistema, como ilustrado pela figura 10, replicada abaixo:

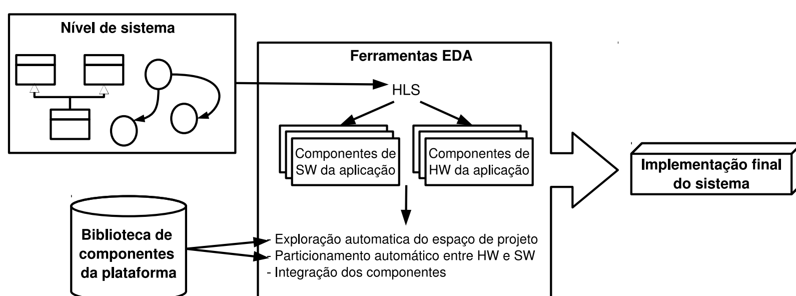

Figura 10: Fluxo de projeto do futuro. Sendo desenvolvido atualmente.

Nesse sentido, a próximo passo é a integração com a ferramenta de geração de sistemas embarcados do EPOS, proposta por Polpeta e Fröhlich (2005). O fluxo de geração do sistema nesta ferramenta está ilustrado na figura 43.

O `Analisador` identifica os componentes usados pela aplicação e ex-

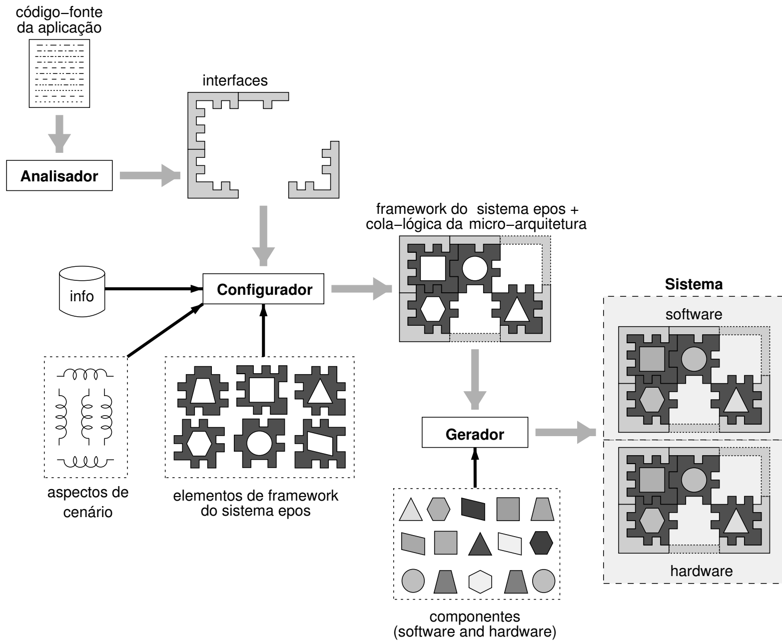

Figura 43: Processo de geração do sistema EPOS (POLPETA; FRÖHLICH, 2005)

trai as interfaces exigidas. O `Configurador` usa as informações e consulta um banco de dados que contém informações de todos os componentes, a suas dependências e regras de composição. Com base nessas informações, são gerados os *traits* do sistema, configurando como que framework metaprogramado adapta os componentes. O `Gerador`, dispara, então, os processos de compilação de software e síntese de hardware. Além da automatização do processo de geração do sistema, esta ferramenta também já foi estendida no trabalho de Cancian (2011), que incorporou ao `Configurador` um modelo evolucionário multiobjetivo para a exploração automática do espaço de projeto.

REFERÊNCIAS BIBLIOGRÁFICAS

ABEL, N. Design and Implementation of an Object-Oriented Framework for Dynamic Partial Reconfiguration. In: *Proc. of the International Conference on Field Programmable Logic and Applications*. [S.l.: s.n.], 2010. p. 240 –243.

AESTE. *AEMB Core*. 2012. Http://web.aeste.my/aemb/.

ALEXANDRESCU, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. [S.l.]: Addison-Wesley, 2001. (C++ in-Depth Series).

ANDERSON, E. et al. Supporting High Level Language Semantics within Hardware Resident Threads. In: *Proc. of the International Conference on Field Programmable Logic and Applications*. [S.l.: s.n.], 2007. p. 98 –103.

AZEVEDO, R. et al. The archc architecture description language and tools. *Int. J. Parallel Program.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 33, n. 5, p. 453–484, out. 2005.

BALARIN, F. et al. Metropolis: an integrated electronic system design environment. *Computer*, v. 36, p. 45–52, April 2003.

BEREJUCK, M. D. *Dynamic Reconfiguration Support for FPGA-based Real-time Systems*. Florianópolis, Brazil, 2011. PhD qualifying report.

BERGAMASCHI, R. A.; COHN, J. The A to Z of SoCs. In: *Proc. of the 2002 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA: ACM, 2002. p. 790–798. ISBN 0-7803-7607-2.

BLACK, D. et al. *SystemC: From the Ground Up*. [S.l.]: Springer, 2009. ISBN 9780387699578.

BOOCH, G. et al. *Object-oriented analysis and design with applications*. [S.l.]: Addison-Wesley, 2007.

BUTT, S. A.; LAVAGNO, L. Designing parameterized signal processing ips for high level synthesis in a model based design environment. In: *Proc. of the 8th IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis*. [S.l.: s.n.], 2012. (CODES+ISSS '12), p. 295–304.

Cadence Design Systems. *C-to-Silicon Compiler*. 2011. Http://www.cadence.com.

CAI, L.; GAJSKI, D. Transaction level modeling: an overview. In: *Proc. of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. Newport Beach, USA: [s.n.], 2003. p. 19–24.

Calypto Design Systems. *CatapultC Synthesis*. 2011. Http://www.calypto.com/.

CANCIAN, R. L. *Um Modelo Evolucionário Multiobjetivo para Exploraçâo do Espaço de Projeto em Sistemas Embarcados*. Tese (Doutorado) — Universidade Federal de Santa Catarina, Florianópolis, Brazil, 2011.

CARDOSO, J. M. P. et al. Specifying Compiler Strategies for FPGA-based Systems. In: *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2012. (FCCM '12), p. 192–199. ISBN 978-0-7695-4699-5.

CESARIO, W. et al. Multiprocessor SoC platforms: a component-based design approach. *IEEE Design & Test of Computers*, v. 19, n. 6, p. 52–63, 2002.

CHIANG, M.-C.; YEH, T.-C.; TSENG, G.-F. A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, v. 30, n. 4, p. 593 –606, april 2011.

CHU, P. P. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. 1. ed. [S.l.]: Wiley-IEEE Press, 2006.

COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 34, n. 2, p. 171–210, jun. 2002.

CONG, J. et al. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, v. 30, n. 4, p. 473 –491, april 2011. ISSN 0278-0070.

COPLIEN, J. O. Curiously recurring template patterns. *C++ Rep.*, New York, USA, v. 7, p. 24–27, February 1995.

CORRE, Y. et al. A framework for high-level synthesis of heterogeneous MP-SoC. In: *Proc of the great lakes symposium on VLSI*. New York, NY, USA: ACM, 2012. (GLSVLSI '12), p. 283–286.

CZARNECKI, K.; EISENECKER, U. W. *Generative programming: methods, tools, and applications*. New York, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

DAVARE, A. et al. A Next-Generation Design Framework for Platform-based Design. In: *Proc. of the Design & Verification Conference & Exhibition*. [S.l.: s.n.], 2007.

DÖMER, R. et al. System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design. *EURASIP J. Embedded Syst.*, New York, United States, v. 2008, p. 5:1–5:13, January 2008.

DZIRI, M.-A. et al. Unified component integration flow for multi-processor SoC design and validation. In: *Proc. of the Design, Automation and Test in Europe Conference and Exhibition*. [S.l.: s.n.], 2004. v. 2, p. 1132–1137 Vol.2.

EKER, J. et al. Taming Heterogeneity - The Ptolemy Approach. In: *Proc. of the IEEE*. [S.l.: s.n.], 2003. p. 127–144.

FALK, J.; HAUBELT, C.; TEICH, J. Efficient Representation and Simulation of Model-Based Designs in SystemC. In: *Proc. of the Forum on Design Languages*. [S.l.: s.n.], 2006. p. 129–134.

FINGEROFF, M. *High-Level Synthesis Blue Book*. [S.l.]: Xlibris Corporation, 2010.

FLOR, J. ao P. P.; MÜCK, T. R.; FRÖHLICH, A. A. High-level Design and Synthesis of a Resource Scheduler. In: *Proc. of the 18th IEEE International Conference on Electronics, Circuits, and Systems*. Beirut, Lebanon: [s.n.], 2011. p. 736–739.

Forte Design Systems. *Cynthesizer*. 2011. Http://www.forteds.com.

FRAKES, W. B.; KANG, K. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, v. 31, p. 529–536, 2005.

FRÖHLICH, A. A. *Application-Oriented Operating Systems*. Sankt Augustin, Germany: GMD - Forschungszentrum Informationstechnik, 2001. 200 p. (GMD Research Series, 17).

FRÖHLICH, A. A.; SCHRÖDER-PREIKSCHAT, W. Scenario Adapters: Efficiently Adapting Components. In: *Proc. of the 4th World Multiconference on Systemics, Cybernetics and Informatics*. Orlando, USA: [s.n.], 2000.

FUJITA, M.; NAKAMURA, H. The standard SpecC language. In: *Proc. of the 14th International Symposium on Systems Synthesis*. New York, NY, USA: ACM, 2001. (ISSS '01), p. 81–86.

GAJSKI, D. D. IP-based design methodology. In: *Proc. of the 36th annual ACM/IEEE Design Automation Conference*. New York, NY, USA: ACM, 1999. (DAC '99), p. 43.

GAJSKI, D. D. et al. *SpecC: Specification Language and Methodology*. 1. ed. [S.l.]: Springer, 2000. ISBN 0792378229.

GANTEL, L. et al. Dataflow programming model for reconfigurable computing. In: *Proc. of th 2011 6th Internationale Workshop on Reconfigurable Communication-centric Systems-on-Chip*. [S.l.: s.n.], 2011. p. 1 –8.

GERSTLAUER, A. et al. Electronic system-level synthesis methodologies. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, IEEE Press, Piscataway, NJ, USA, v. 28, n. 10, p. 1517–1530, out. 2009.

GERSTLAUER, A. et al. Specify-explore-refine (SER): from specification to implementation. In: *Proc. of the 45th annual Design Automation Conference*. New York, NY, USA: ACM, 2008. (DAC '08), p. 586–591.

GRACIOLI, G.; FRÖHLICH, A. A. ELUS: A dynamic software reconfiguration infrastructure for embedded systems. In: *Proc. of the IEEE 17th International Conference on Telecommunications*. [S.l.: s.n.], 2010. p. 981 –988.

GRÜTTNER, K. et al. Towards a Synthesis Semantics for SystemC Channels. In: *Proc. of the 8th IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2010. (CODES/ISSS '10), p. 163–172.

GUTHAUS, M. R. et al. Mibench: A free, commercially representative embedded benchmark suite. In: *Proc. of the IEEE Int. Workshop on Workload Characterization*. Washington, DC, USA: IEEE Computer Society, 2001. p. 3–14. Http://www.eecs.umich.edu/mibench/.

HA, S. et al. PeaCE: A hardware-software codesign environment for multimedia embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, New York, USA, v. 12, p. 24:1–24:25, May 2008.

IEEE. *Std 1076-2000: IEEE Standard VHDL Language Reference Manual*. [S.l.], 2000.

IEEE. *Std 1364-2001: IEEE Standard Verilog Hardware Description Language*. [S.l.], 2001.

IEEE. *AMBA AXI Protocol Specification (Rev 2.0)*. 2010. Http://www.arm.com.

Interactive Multimedia Association (IMA). *Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems*. [S.l.], 1992. <http://www.cs.columbia.edu/h̃gs/audio/dvi/IMA_ADPCM.pdf>.

KEINERT, J. et al. SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Trans. Des. Autom. Electron. Syst.*, New York, USA, v. 14, p. 1:1–1:23, January 2009.

KICZALES, G. et al. Aspect-Oriented Programming. In: *Proc. of the European Conference on Object-oriented Programming*. Jyväskylä, Finland: [s.n.], 1997. v. 1241, p. 220–242.

KRANENBURG, T. *Design of a Portable and Customizable Microprocessor for Rapid System Prototyping*. Dissertação (Mestrado) — Delft University of Technology, Delft, The Netherlands, 2009.

KRöNING, D. *The CBMC homepage*. 2012. <http://www.cprover.org/cbmc/>.

LANGE, H.; KOCH, A. Architectures and Execution Models for Hardware/Software Compilation and Their System-Level Realization. *IEEE Transactions on Computers*, v. 59, n. 10, p. 1363 –1377, oct. 2010.

LARMAN, C. *Applying UML And Patterns: An Introduction To Object-Oriented Analysis And Design And Iterative Development*. [S.l.]: Prentice Hall PTR, 2005.

LORENZ, D. et al. From rtl ip to functional system-level models with extra-functional properties. In: *Proc. of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2012. (CODES+ISSS '12), p. 547–556.

LÜBBERS, E.; PLATZNER, M. A portable abstraction layer for hardware threads. In: *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. [S.l.: s.n.], 2008. p. 17 –22.

LÜBBERS, E.; PLATZNER, M. ReconOS: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst.*, New York, NY, USA, v. 9, n. 1, p. 8:1–8:33, out. 2009.

LUDWICH, M. K.; FRöHLICH, A. A. System-Level Verification of Embedded Operating Systems Components. In: *Proc of the Brazilian Symposium on Computing System Engineering*. Natal, Brazil: [s.n.], 2012.

MARCONDES, H. *Uma Arquitetura de Componentes Híbridos de Hardware e Software para Sistemas Embarcados*. Dissertação (Mestrado) — Federal University of Santa Catarina, Florianópolis, Brazil, 2009.

MARCONDES, H.; FRÖHLICH, A. A. A Hybrid Hardware and Software Component Architecture for Embedded System Design. In: *Proc. of the International Embedded Systems Symposium*. Langenargen, Germany: [s.n.], 2009. p. 259–270.

MARTIN, G. et al. Component selection and matching for IP-based design. In: *Proc. of the Conference on Design, Automation and Test in Europe*. Piscataway, NJ, USA: IEEE Press, 2001. (DATE '01), p. 40–46.

MARWEDEL, P. *Embedded System Design*. [S.l.]: Kluwer Academic Publishers, 2003. ISBN 1-4020-7690-9.

Mentor Graphics. *ModelSim*. 2011. Http://model.com/.

MEYER, B. A framework for proving contract-equipped classes. In: *Proc. of the 10th International Workshop on Abstract State Machines: Advances in Theory and Applications*. Berlin, Heidelberg: Springer-Verlag, 2003. (ASM'03), p. 108–125. ISBN 3-540-00624-9.

MICHELI, G. D. et al. Networks on Chips: from research to products. In: *Proc. of the 47th Design Automation Conference*. New York, NY, USA: ACM, 2010. (DAC '10), p. 300–305. ISBN 978-1-4503-0002-5.

MIKHAJLOV, L.; SEKERINSKI, E. A study of the fragile base class problem. In: *Proceedings of the 12th European Conference on Object-Oriented Programming*. [S.l.]: Springer-Verlag, 1998. p. 355–382. ISBN 3-540-64737-6.

MISCHKALLA, F.; HE, D.; MUELLER, W. Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems. In: *Proc. of the Conference on Design, Automation and Test in Europe*. Dresden, Germany: [s.n.], 2010. p. 1201–1206.

MRABTI, A. E. et al. Abstract Description of System Application and Hardware Architecture for Hardware/Software Code Generation. In: *Proc. of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. [S.l.: s.n.], 2009. (DSD '09), p. 567 –574.

MÜCK, T. R.; FRÖHLICH, A. A. A Run-time Memory Management Approach for Scratch-pad-based Embedded Systems. In: *Proc. of the 15th IEEE International Conference on Emerging Techonologies and Factory Automation*. Bilbao, Spain: [s.n.], 2010. p. 1–4. ISBN 978-1-4244-6849-2.

MÜCK, T. R.; FRÖHLICH, A. A. Hyra: A software-defined radio architecture for wireless embedded systems. In: *Proc. of the 10th International Conference on Networks*. St. Maarten, The Netherlands Antilles: [s.n.], 2011. p. 246–251.

MÜCK, T. R.; FRÖHLICH, A. A. Run-time Scratch-pad Memory Management for Embedded Systems. In: *Proc. of the 37th Annual Conference of the IEEE Industrial Electronics Society*. Melbourne, Australia: [s.n.], 2011. p. 2748–2753. ISBN 978-1-61284-971-3.

MÜCK, T. R.; FRÖHLICH, A. A. On AOP techniques for C++-based HW/SW component implementation. In: *Proc. of the 19th IEEE International Conference on Electronics, Circuits, and Systems*. Seville, Spain: [s.n.], 2012. p. 536–539. ISBN 978-1-4673-1259-2.

MÜCK, T. R. et al. A Case Study of AOP and OOP applied to digital hardware design. In: *Proc. of the Brazilian Symposium on Computing System Engineering*. Florianópolis, Brazil: [s.n.], 2011. p. 146–157. ISBN 978-1-4673-0427-6.

MÜCK, T. R. et al. Implementing OS Components in Hardware using AOP. *SIGOPS Operating Systems Review*, v. 46, n. 1, p. 64–72, 2012.

MYERS, N. C. Traits: a new and useful template technique. *C++ Report*, SIGS Publications, Inc., New York, NY, USA, June 1995. <http://www.cantrip.org/traits.html>.

ODERSKY, M.; SPOON, L.; VENNERS, B. *Programming in Scala*. [S.l.]: Artima Inc., 2008.

OMG. *Common Object Request Broker Architecture (CORBA)*. [S.l.], 1997. <http://www.corba.org/>.

OMG. *Systems Modeling Language (SysML)*. [S.l.], 2007. <http://www.sysml.org>.

OMG. *Unified Modeling Language (UML)*. [S.l.], 2008. <http://www.uml.org>.

Open Group. *POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)*. [S.l.], 1995. <http://www.opengroup.org>.

Oracle. *Java Remote Method Invocation (RMI)*. [S.l.], 2010. <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html>.

OSCI. *SystemC Synthesizable Subset Draft 1.3*. [S.l.], 2010. <http://www.systemc.org/>.

OUADJAOUT, S.; HOUZET, D. Generation of embedded hardware/software from SystemC. *EURASIP J. Embedded Syst.*, Hindawi Publishing Corp., New York, NY, United States, v. 2006, n. 1, p. 19–19, jan. 2006. ISSN 1687-3955.

PANDA, P. R. SystemC: a modeling platform supporting multiple design abstractions. In: *Proc. of the 14th international symposium on Systems synthesis*. Montréal, Canada: [s.n.], 2001. p. 75–80.

PARNAS, D. L. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2, n. 1, p. 1–9, Mar. 1976.

Paul Pop. Embedded Systems Design: Optimization Challenges. In: *CPAIOR*. [S.l.: s.n.], 2005. p. 16–16.

POLPETA, F. V.; FRÖHLICH, A. A. Hardware mediators: a portability artifact for component-based systems. In: *Proc. of the International Conference on Embedded and Ubiquitous Computing*. [S.l.]: Springer, 2004. p. 271–280.

POLPETA, F. V.; FRÖHLICH, A. A. On the Automatic Generation of SoC-based Embedded Systems. In: *Proc. of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*. Catania, Italy: [s.n.], 2005.

RICCOBENE, E. et al. SystemC/C-based model-driven design for embedded systems. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 8, n. 4, p. 30:1–30:37, jul. 2009. ISSN 1539-9087.

RINCÓN, F. et al. Transparent IP Cores Integration Based on the Distributed Object Paradigm. In: *Intelligent Technical Systems*. [S.l.: s.n.], 2009, (Lecture Notes in Electrical Engineering, v. 38). p. 131–144.

SANGIOVANNI-VINCENTELLI, A. et al. Benefits and challenges for Platform-based Design. In: *Proc. of the 41st Annual Conference on Design Automation*. San Diego, USA: [s.n.], 2004. (DAC '04), p. 409–414.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Design & Test of Computers*, v. 18, p. 23–33, 2001.

SCHALLENBERG, A. et al. OSSS+R: a framework for application level modelling and synthesis of reconfigurable systems. In: *Proc. of the Conference on Design, Automation and Test in Europe*. Nice, France: [s.n.], 2009. p. 970–975.

SCHMIDT, D. C. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, v. 39, n. 2, p. 25 – 31, feb. 2006.

SCHULTER, A. et al. A Tool for Supporting and Automating the Development of Component-based Embedded Systems. *Journal of Object Technology*, v. 6, n. 9, p. 399–416, Oct 2007. ISSN 1660-1769.

SMITH, G. *How to Achieve 44% Design Cost Reduction*. Jun 2012. Keynote in the 49th Design Automation Conference.

SO, H. K.-H.; BRODERSEN, R. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Trans. Embed. Comput. Syst.*, New York, USA, v. 7, p. 14:1–14:28, January 2008. ISSN 1539-9087.

SPINCZYK, O.; GAL, A.; SCHRÖDER-PREIKSCHAT, W. AspectC++: an aspect-oriented extension to the C++ programming language. In: *Proc. of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. Sydney, Australia: [s.n.], 2002. p. 53–60.

STEINER, R.; MüCK, T. R.; FRöHLICH, A. A. A Configurable Medium Access Control Protocol for IEEE 802.15.4 Networks. In: *Proc. of the 2010 Int. Congress on Ultra Modern Telecommunications and Control Systems*. Moscow, Russia: [s.n.], 2010. p. 301–308. ISBN 978-1-4244-7285-7.

STEINER, R.; MüCK, T. R.; FRöHLICH, A. A. C-MAC: a Configurable Medium Access Control Protocol for Sensor Networks. In: *Proc. of the 9th IEEE Sensors*. Waikoloa, HI, USA: [s.n.], 2010. p. 845–848. ISBN 978-1-4244-8168-2.

STEPANOV, A.; LEE, M. *The Standard Template Library*. [S.l.], 1995.

Synopsys. *Synphony C Compiler*. 2011. Http://www.synopsys.com.

The EPOS Project. *Embedded Parallel Operating System*. 2011. Http://epos.lisha.ufsc.br/.

THOMPSON, M. et al. A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In: *Proc. of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2007. (CODES+ISSS '07), p. 9–14.

WEHRMEISTER, M. A. *An Aspect-Oriented Model-Driven Engineering Approach for Distributed Embedded Real-Time Systems*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil, 2009.

WILTON, S.; SALEH, R. Programmable logic IP cores in SoC design: opportunities and challenges. In: *Proc. of the IEEE Conference on Custom Integrated Circuits*. [S.l.: s.n.], 2001. p. 63 –66.

XILINX. *OPB IPIF (v3.01c)*. 2005. Http://www.xilinx.com/support/documentation/ip_documentation/opb_ipif.pdf.

Xilinx. *AutoESL High-Level Synthesis*. 2012. Http://www.xilinx.com/tools/autoesl.htm.

XILINX. *MicroBlaze Soft Processor Core*. 2012. Http://www.xilinx.com/tools/microblaze.htm.

Xilinx. *Virtex-6 FPGA ML605 Evaluation Kit*. 2012. Http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm.

Xilinx. *Zynq-7000 Extensible Processing Platform*. 2012. Http://www.xilinx.com/products/silicon-devices/epp/zynq-7000/index.htm.

APÊNDICE A – Código fonte

Este apêndice está disponível online através do link:
http://www.lisha.ufsc.br/pub/Muck_MSC_2013_apA.pdf

APÊNDICE B – Compilando implementações unificadas

Este apêndice está disponível online através do link:
http://www.lisha.ufsc.br/pub/Muck_MSC_2013_apB.pdf

APÊNDICE C – Produção científica

Este apêndice lista os artigos produzidos e publicados ao longo do presente mestrado. Apesar de serem auto-contidos e estarem fora do contexto principal do presente trabalho, alguns dos esforços anteriores geraram contribuições importantes e foram incluídos nesta dissertação de forma que a mesma descreva integralmente os resultados obtidos durante todo o curso do mestrado.

- MÜCK, T. R.; FRÖHLICH, A. A. A Run-time Memory Management Approach for Scratch-pad-based Embedded Systems. In: *Proc. of the 15th IEEE International Conference on Emerging Techonologies and Factory Automation*. Bilbao, Spain: [s.n.], 2010. p. 1–4. ISBN 978-1-4244-6849-2.
- STEINER, R.; MüCK, T. R.; FRöHLICH, A. A. A Configurable Medium Access Control Protocol for IEEE 802.15.4 Networks. In: *Proc. of the 2010 Int. Congress on Ultra Modern Telecommunications and Control Systems*. Moscow, Russia: [s.n.], 2010. p. 301–308. ISBN 978-1-4244-7285-7.
- STEINER, R.; MüCK, T. R.; FRöHLICH, A. A. C-MAC: a Configurable Medium Access Control Protocol for Sensor Networks. In: *Proc. of the 9th IEEE Sensors*. Waikoloa, HI, USA: [s.n.], 2010. p. 845–848. ISBN 978-1-4244-8168-2.
- MÜCK, T. R.; FRÖHLICH, A. A. Hyra: A software-defined radio architecture for wireless embedded systems. In: *Proc. of the 10th International Conference on Networks*. St. Maarten, The Netherlands Antilles: [s.n.], 2011. p. 246–251.
- MÜCK, T. R. et al. A Case Study of AOP and OOP applied to digital hardware design. In: *Proc. of the Brazilian Symposium on Computing System Engineering*. Florianópolis, Brazil: [s.n.], 2011. p. 146–157. ISBN 978-1-4673-0427-6.
- FLOR, J. ao P. P.; MÜCK, T. R.; FRÖHLICH, A. A. High-level Design and Synthesis of a Resource Scheduler. In: *Proc. of the 18th IEEE International Conference on Electronics, Circuits, and Systems*. Beirut, Lebanon: [s.n.], 2011. p. 736–739.
- MÜCK, T. R.; FRÖHLICH, A. A. Run-time Scratch-pad Memory Management for Embedded Systems. In: *Proc. of the 37th Annual Conference of the IEEE Industrial Electronics Society*. Melbourne, Australia: [s.n.], 2011. p. 2748–2753. ISBN 978-1-61284-971-3.
- MÜCK, T. R. et al. Implementing OS Components in Hardware using AOP. *SIGOPS Operating Systems Review*, v. 46, n. 1, p. 64–72, 2012.

- MÜCK, T. R.; FRÖHLICH, A. A. On AOP techniques for C++-based HW/SW component implementation. In: *Proc. of the 19th IEEE International Conference on Electronics, Circuits, and Systems*. Seville, Spain: [s.n.], 2012. p. 536–539. ISBN 978-1-4673-1259-2.