UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Fauze Valério Polpeta

Uma Estratégia para a Geração de Sistemas Embutidos baseada na Metodologia Projeto de Sistemas Orientados à Aplicação

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Antônio Augusto Fröhlich, Prof. Dr.
Orientador

Florianópolis, Fevereiro de 2006

Uma Estratégia para a Geração de Sistemas Embutidos baseada na Metodologia Projeto de Sistemas Orientados à Aplicação

Fauze Valério Polpeta

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Raul Sidnei Wazlawick, Prof. Dr.

Banca Examinadora

Antônio Augusto Fröhlich, Prof. Dr.
Orientador

Carlos Eduardo Pereira, Prof. Dr.

Leandro Buss Becker, Prof. Dr.

Ricardo Pereira e Silva, Prof. Dr.

"Não sou do tamanho da minha altura.
Sou do tamanho daquilo que vejo,
e o que vejo são meus sonhos."

Fernando Pessoa

Ao meu eterno amor, Renata.

Agradecimentos

Agradeço primeiramente a minha esposa, Renata Fortunato Ávila Polpeta. Sua paciência, compreensão e força foram decisivas para a conclusão deste mestrado. Seu amor, imcomparável fonte de inspiração e alegria. Compartilho este momento inteiramente com você, minha querida. E claro, com nossa filha que está para chegar.

Aos meus colegas de trabalho do Laboratório de Integração Software Hardware (LISHA) que colaboraram com a concretização das idéias deste trabalho em várias discussões científicas. Especialmente, Arliones Stevert Hoeller Júnior, Lucas Francisco Wanner e Hugo Marcondes...verdadeiros "Lisheiros".

Agradeço meu orientador, Professor Antônio Augusto Fröhlich. Por sua dedicação, apoio e grande esforço em tornar, aqueles que passam pelo LISHA, grandes pesquisadores. Pelos momentos de vitória, derrotas, alegrias e discussões ao longo destes três anos. Estes momentos enriqueceram-me de conhecimento, os quais foram fundamentais para a realização deste trabalho e de outros que virão.

Ao Departamento de Pós-graducação em Ciências da Computação (PGCC) e à Universidade Federal de Santa Catarina (UFSC), pela oportunidade ímpar que foi a realização deste mestrado e a todas as pessoas que zelam por estas instituições.

E a Deus, por ter guiado meus passos durante esta jornada. Por ter me concedido a sabedoria e a força necessárias nos momentos em que fraquejei. Sinceramente, obrigado meu Pai.

Sumário

Lista de Figuras viii

Lista de Tabelas x

Lista de Siglas xi

Resumo xi

Abstract xii

1 Introdução 1

1.1 Motivações 3

1.2 Objetivo 8

1.3 Organização do Texto 8

2 Projeto Baseado em Plataformas 10

2.1 Definindo PBD 10

2.2 Projeto de SoCs Baseados em Plataformas 14

2.2.1 Plataformas de Hardware 19

2.2.2 Plataformas de Software 24

3 Projeto de Sistemas Orientados à Aplicação 28

3.1 A Metodologia AOSD 28

3.2 Mediadores de Hardware 31

4 Descrição e Especialização de Plataformas Segundo AOSD **36**

4.1 Descrição de Componentes 36

4.1.1 Propriedades Configuráveis 39

4.1.2 Dependências 41

4.1.3 Propriedades Não Funcionais 42

4.2 Especialização de Plataformas 42

4.2.1 Mediadores e Componentes de Hardware 48

5 SoCs Orientados À Aplicação **52**

5.1 SoCs Leon2 no Sistema EPOS 52

5.2 Ambiente de Geração de SoCs 56

5.3 Estudos de Caso 58

5.3.1 Aplicação Produtor-Consumidor 59

5.3.2 Tocador de Áudio PCM 78

6 Conclusão e Trabalhos Futuros **83**

Referências Bibliográficas **86**

Apêndice **95**

A DTD da linguagem de descrição de componentes **95**

B Descrição da família de abstrações Thread **98**

Lista de Figuras

1.1 Parametrização em VHDL e C++. 4

1.2 Instanciação de componentes de software e hardware. 6

2.1 Especialização de plataformas. 12

2.2 A plataforma de sistema. 13

2.3 Integração de IPs com interfaces padronizadas. 16

2.4 Integração de IPs usando wrappers. 17

2.5 Micro-arquitetura OpenRisc 22

2.6 Micro-arquitetura Leon2 24

3.1 Visão da decomposição de domínio proposta por AOSD 30

3.2 Família de mediadores de hardware de CPUs. 32

3.3 Fragmento de mediador de CPU da arquitetura IA-32. 34

4.1 Diagrama de ferrovia da descrição de famílias. 38

4.2 Diagrama de ferrovia da descrição de membros de famílias. 39

4.3 Diagramas de ferrovia da descrição de propriedades configuráveis. 40

4.4 Fluxograma do funcionalmento do mecanismo de especialização proposto. 44

4.5 Descrição da CPU da micro-arquitetura Leon2. 47

4.6 Especialização de plataformas segundo AOSD. 51

5.1 Fragmento do arquivo de configuração de propriedades configuráveis em C++. 54

5.2 Fragmento do arquivo de configuração de propriedades configuráveis em VHDL. 55
5.3 Fragmento do arquivo de configuração de propriedades configuráveis em Verilog. 56
5.4 Processo de geração do sistema EPOS. 57
5.5 Composição e validação de sistemas. 58
5.6 Código da aplicação Produtor-Consumidor. 60
5.7 Saída do Analisador para aplicação Produtor-Consumidor. 61
5.8 Especialização de plataformas para aplicação Produtor-Consumidor. . . . 63
5.9 Especificação dos componentes a serem instanciados no sistema. 66
5.10 Micro-arquitetura Leon2 especializada para aplicação Produtor-Consumidor. 68
5.11 Especialização de plataformas para aplicação Produtor-Consumidor com controle de erros de transmissão. 71
5.12 Especialização de plataformas para aplicação Produtor-Consumidor com Depuração. 73
5.13 Micro-arquitetura Leon2 especializada para aplicação Produtor-Consumidor com Unidade de Depuração. 74
5.14 Especialização de plataformas para aplicação Produtor-Consumidor sobre Ethernet. 76
5.15 Micro-arquitetura Leon2 especializada para aplicação Produtor-Consumidor sobre Ethernet. 77
5.16 Código fonte da aplicação Tocador de Áudio PCM. 80
5.17 Especialização de plataformas para aplicação Tocador de Áudio PCM. . . 81
5.18 Micro-arquitetura XMB_Leon2 especializada para aplicação Tocador de Áudio PCM. 82

Lista de Tabelas

5.1 Tamanho dos segmentos de código e dados dos sistemas operacionais EPOS, uCLinux e eCos para aplicação Produtor-Consumidor sobre Leon2. 69
5.2 Tamanho da micro-arquitetura Leon2 quando especializada para aplicação Produtor-Consumidor e quando sintetizada com todos os dispositivos. 69
5.3 Tempo consumido para executar 16.000 iterações de envio e recebimento na aplicação Produtor-Consumidor quando implementada sobre os sistemas operacionais EPOS e eCos. 70

Resumo

Atualmente, a crescente complexidade das aplicações embutidas tem demandado metodologias e estratégias que reduzam o impacto desta complexidade no desenvolvimento de sistemas embutidos. O reuso de componentes de software e hardware previamente projetados e verificados pode ser considerado um dos principais pilares das soluções propostas neste sentido. Todavia, este reuso pode não ser suficiente para determinar como os componentes devem ser integrados para que atendam os diferentes cenários de aplicação de sistemas embutidos. Alternativamente, padrões micro-arquiteturais conhecidos como *plataformas* podem ser utilizados para abstrair estes detalhes e oferecer ao projetista uma base de projeto que pode ser especializada de acordo com os requisitos da aplicação. Dentro deste contexto, o presente trabalho propõe uma estratégia que perfaz a especialização destas plataformas a partir de requisitos extraídos da aplicação. Isto é realizado através da descrição de componentes de hardware e software segundo as principais premissas da metodologia *Projeto de Sistema Orientados a Aplicação* (AOSD). Estas descrições formam uma base de dados sobre a qual uma ferramenta de inferência e configuração determina quais componentes devem compor a plataforma de hardware e o suporte operacional de sistemas embutidos orientados à aplicação. As etapas envolvidas neste processo são apresentadas em estudos de caso usando o sistema operacional EPOS—um sistema operacional experimental baseado na metodologia AOSD—e a plataforma de hardware LEON2.

Palavras-chave: Sistemas Embutidos, Projeto de Sistemas Orientados à Aplicação, Projeto Baseado em Plataformas, Soft-Cores, Engenharia de Software.

Abstract

The growing complexity of embedded applications demands methodologies and strategies that reduce the impact of this complexity in the development of embedded systems. The reuse of pre-designed and pre-verified components can be considered one of the main pillars of the solutions that are proposed in this direction. However, it is not enough to know how the components shall be integrated in order to support different application scenarios. Alternatively, micro-architectural templates, namely platforms, can be used to abstract such integration details and provide to the developer a project basis from which different systems can arise. In this context, the present work proposes a strategy that performs the specialization of theses platforms by using requisites that are extracted from the application. This is achieved through the deployment of the *Application-oriented System Design* methodology on the description of hardware and software components in a component database, which, in turn, is used by a configuration tool that is able to specify which of these components must be instantiated to properly fulfill the application's requirements. The several steps involved in this process are presented in detailed experiments that were made considering the EPOS system—an experimental AOSD-based operating system— and the LEON2 hardware platform.

Keywords: Embedded Systems, Application-oriented System Design, Platform-based Design, Soft-Cores, Software Engineering.

Capítulo 1

Introdução

Sistemas embutidos têm se tornado cada vez mais complexos e a busca por estratégias de projeto que permitam abstrair esta complexidade é tema recorrente de trabalhos realizados na área. Além do apelo científico, o fato de muitos destes trabalhos estarem comprometidos com a redução de tempo e esforço despendidos no desenvolvimento, faz com que muitas das estratégias investigadas pela academia estejam também relacionadas com as barreiras e tendências tecnológicas existentes na indústria. Dentre estas tendências, o desenvolvimento de sistemas em um único *chip* 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 embutidos. Todavia, conforme detalhado por Bergamaschi [1], o projeto destes sistemas, também conhecidos como SOCs (do inglês *System-on-a-Chip*), consiste de um custoso e complexo processo de engenharia. Mesmo quando há o reuso de componentes, diversas etapas de refinamento e validação são realizadas ciclicamente para assegurar que o sistema atenda os requisitos da aplicação para a qual foi projetado. Diante deste cenário é que o *Projeto Baseado em Plataformas* [2] ganha sua importância.

Firmando um compromisso entre a complexidade e o custo de desenvolvimento de sistemas computacionais embutidos, o *Projeto Baseado em Plataformas*, ou simplesmente PBD (do inglês *Platform-based Design*), propõe a utilização de padrões previamente validados que, a priori, podem ser vistos como bibliotecas de componentes

com suas respectivas regras de integração. Além de promoverem o reuso de componentes, estes 'padrões", também chamados de *plataformas*, têm por objetivo esconder do projetista a maneira pela qual os componentes devem ser organizados (ou integrados) para que suas funcionalidades sejam realizadas em um sistema. Em outras palavras, uma plataforma abstrai os detalhes micro-arquiteturais do sistema e com isto o projetista pode concentrar-se na especificação das aplicações.

Segundo Vincentelli [3], embora conceitualmente PBD reduza os custos envolvidos no projeto de um sistema embutido, esta abordagem de projeto ainda impõem alguns desafios para o seu sucesso. Um deles, é a própria especificação e implementação de plataformas cujas arquiteturas sejam (re-)utilizadas em uma gama justificável de projetos. O outro, consiste em definir metodologias e estratégias que possibilitam gerar, a partir destas plataformas, sistemas para aplicações específicas. Além da configuração e integração dos componentes de hardware, uma plataforma de software também deve ter seus componentes configurados e integrados para que satisfaça juntamente com o hardware os requisitos da aplicação. Neste ponto está centrada a proposta do presente trabalho: uma estratégia para a geração de sistemas embutidos a partir de plataformas previamente validadas usando a metodologia "Projeto de Sistemas Orientados a Aplicação" (AOSD) (do inglês *Application-oriented System Design*) [4].

Concebida no escopo de sistemas computacionais dedicados, a metodologia AOSD propõe estratégias para conduzir uma engenharia de domínio no sentido de identificar e abstrair entidades significativas de diferentes domínios. Para isto, paradigmas e mecanismos da engenharia de software são utilizados no projeto e implementação de componentes, os quais podem ser configurados e integrados para compor o suporte operacional adequado a diferentes aplicações. Neste trabalho, os mecanismos de inferência e configuração utilizados na composição deste suporte foram adaptados para que plataformas de hardware sejam também especializadas de acordo com os requisitos das aplicações. Desta forma, é possível guiar sistematicamente o projetista na seleção e configuração dos componentes que irão compor não só o suporte operacional mas também o hardware necessário para atender os requisitos de uma dada aplicação. O sistema gerado, de fato um SoC orientado à aplicação, pode ao final ser instanciado em uma FPGA para

avaliação e constatação de sua funcionalidade.

1.1 Motivações

Por algum tempo, grande parte das pesquisas relacionadas ao desenvolvimento de sistemas embutidos baseados em componentes estiveram focadas na parte que caracteriza o software destes sistemas. Talvez por um paradigma tecnológico, ou pelo fato dos sistemas comerciais atenderem as necessidades do mercado, a idéia por trás do desenvolvimento de um sistema embutido era fixar o hardware do sistema—em geral um microprocessador, memória e periféricos ou então um microcontrolador—e trabalhar encima de seu software. Dependendo da complexidade da aplicação, este software poderia resumir-se apenas a algumas rotinas de controle em *assembly* ou C, como também ser projetado segundo metodologias ou estratégias de desenvolvimento de componentes de software que, por exemplo, tratam da implementação de *drivers* de dispositivo, escalonadores e, dentre outros elementos, da composição de um sistema operacional a partir de requisitos extraídos da própria aplicação [5, 6, 4].

Atualmente, diante dos avanços da indústria de semicondutores e das técnicas de projeto e implementação de hardware, componentes conhecidos como IPs (do inglês *Intellectual Property*) têm ganhado um espaço cada vez maior no desenvolvimento de sistemas embutidos [7, 8, 9]. Mais especificamente no desenvolvimento do hardware destes sistemas, onde a classe de IPs conhecida como *soft-cores* [1] tem possibilitado a composição e rápida prototipagem de complexos sistemas de hardware em dispositivos conhecidos como PLDs (do inglês *Programmable Logic Device*); descaracterizando de certa forma o aspecto pouco "flexível" do hardware.

Além da rápida prototipação, componentes de hardware, quando implementados na forma de *soft-cores*, possuem propriedades que os colocam muito mais próximos da maneira com que componentes de software são utilizados no projeto de sistemas embutidos do que quando fixos em uma PCB (do inglês *Printed Circuit Board*). Isto, em parte, deve-se ao nível de abstração alcançado por linguagens de descrição de hardware,

[1] Há autores que preferem designar *soft-cores* como *núcleos leves*.

as quais incluem artifícios que permitem implementar componentes com níveis consideráveis de reuso e configurabilidade. A construção *Generics* de VHDL, por exemplo, possibilita ao projetista parametrizar um componente de hardware de maneira muito similar com que um componente de software é parametrizado em C++. A Figura 1.1 mostra trechos de código nestas duas linguagens nos quais uma entidade em VHDL e uma classe em C++ têm parametrizado o número de bits que os sinais (ou atributos) devem possuir.

```
// VHDL
entity Componente is
generic (numero_de_bits : positive);
  entrada : in  bit_vector (0 to numero_de_bits-1);
  saida   : out bit_vector (0 to numero_de_bits-1);
end Componente;

// C++
template<unsigned int numero_de_bits>
class Componente {
   public:
      typedef unsigned long long bit_vector;
   protected:
      bit_vector entrada : numero_de_bits;
      bit_vector saida   : numero_de_bits;
};
```

Figura 1.1: Parametrização em VHDL e C++.

Estruturalmente, um componente de hardware possui uma interface que é representada pelos sinais que exterioriza. Fisicamente, estes sinais podem ser vistos como os pinos de um circuito integrado e, em uma analogia com componentes de software, como o conjunto de métodos pelos quais suas funcionalidades são exportadas. Outro aspecto de sua estrutura é o comportamento, caracterizado em geral por algum tipo de processamento que o componente realiza. Na prática, o comportamento é implementado por processos que são desencadeados quando estímulos são aplicados na interface

do componente. Este processo pode ser comparado à execução dos métodos implementados por um componente de software. Uma chamada a um método de sua interface pode desencadear a execução de outras rotinas, as quais podem retornar valores e/ou modificar os atributos que definem seu estado interno. No hardware, este estado poderia, por exemplo, ser representado por um conjunto de registradores e, o retorno de uma rotina, pela alteração do nível de um sinal em sua interface.

No que se refere à instanciação de componentes de hardware a partir de *soft-cores*, também é possível identificar algumas semelhanças com o software. A figura 1.2 identifica as etapas deste processo em ambos os domínios. Como ilustrado, após a configuração dos componentes procede-se com a síntese do hardware e, do lado do software, com a compilação. Os resultados destas etapas são respectivamente uma *netlist* e o *código-objeto*. Salva a exceção de que uma *netlist* pode ser portável, o conteúdo destas saídas constituem representações em baixo nível dos componentes que serão na seqüência mapeados para seus respectivos ambientes de "execução". O componente de software passa pela etapa de "linkagem", na qual são resolvidos endereços necessários para que o programa gerado seja alocado e processado corretamente a partir da memória. O componente de hardware, por sua vez, passa pela etapa de *place-and-route*. Nesta etapa, o conjunto de conexões e portas definidos por uma *netlist* são arranjados de acordo com a tecnologia onde o componente de hardware será fisicamente instanciado. No caso de FPGAs a saída do *place-and-route* é um fluxo de bits (i.e. *bitstream*) que é utilizado na programação destes dispositivos.

Sobre o projeto de componentes de hardware e de componentes de software, a principal diferença é que, no domínio do hardware, fatores como sincronismo e propagação de sinais exigem uma série de refinamentos até que o componente seja passível de ser instanciado. Apesar de linguagens de descrição de hardware (e de descrição de sistemas) viabilizarem a abstração em alto-nível de alguns destes detalhes, a validação de um componente de hardware não deixa de ser um processo demorado e custoso. Logo, embora seja possível afirmar que um componente de hardware implementado como *soft-core* possui propriedades que permitem-no ser selecionado, configurado e instanciado de maneira muito similar à realizada com um componente de software, seu projeto despende

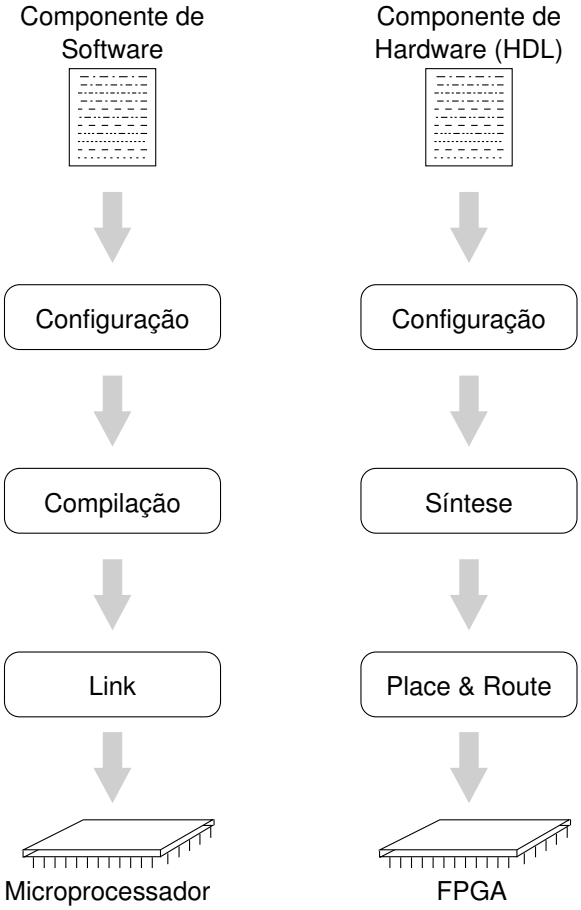

Figura 1.2: Instanciação de componentes de software e hardware.

um grande esforço até que se tenha isto como verdade.

Gupta e Gajski [10, 11] fazem um diagnóstico desta problemática e enfatizam que o reuso de componentes previamente projetados e validados é fundamental para abstrair a complexidade envolvida no projeto de sistemas embutidos. Todavia, mesmo a utilização de *soft-cores* e FPGAs pode não ser suficiente para reduzir satisfatoriamente o tempo e o esforço associados no desenvolvimento de um SOC. Dependendo dos requisitos da aplicação que este SOC deverá suportar, a especificação e validação de sua arquitetura pode ser caracterizada por um intrincado e custoso processo de engenharia.

Alternativamente, o *Projeto Baseado em Plataformas* [12, 13] reafirma a importância do reuso de componentes, porém, a fim de minimizar os contratempos do

projeto baseado em IPs, esta abordagem também propõe que o desenvolvimento de sistemas embutidos seja feito através da especialização de padrões conhecidos como plataformas. Do lado hardware, uma plataforma denominada *micro-arquitetura* inclui microprocessadores, memória e uma estrutura de comunicação que é utilizada para interconectar dispositivos adicionais. Sobre esta micro-arquitetura é feita a seleção, configuração e integração dos componentes de hardware de acordo com os requisitos da aplicação para a qual o sistema foi projetado. Do lado do software, a plataforma de software, normalmente a API de um sistema operacional, abstrai as funcionalidades do hardware e provê serviços para a aplicação.

Neste contexto, diante das semelhanças diagnosticadas entre componentes de hardware e software, a metodologia *Projeto de Sistemas Orientados à Aplicação*, embora originalmente associada apenas à plataforma de software de um sistema embutido, teve seus mecanismos de inferência e configuração adaptados para suportar também a especialização de plataformas de hardware e, portanto, a geração de sistemas orientados à aplicação. Neste processo, o artefato *Mediador de Hardware* [4] assume papel fundamental no mapeamento dos requisitos da aplicação em uma micro-arquitetura. Em um estudo sobre a portabilidade de sistemas operacionais baseados em componentes [14, 15], foi constatado que a interface software-hardware resultante da instanciação de mediadores de hardware pode ser vista como uma especificação dos requerimentos da aplicação para com o hardware. Logo, a associação destes mediadores a *soft-cores* foi traduzida na possibilidade de explorar componentes de hardware tal como componentes de software são originalmente explorados pela metodologia AOSD. O reuso e a configurabilidade encontradas nos *soft-cores* são as principais características desta classe de IPs que motivaram esta exploração. Para cada mediador instanciado, um componente de hardware de uma micro-arquitetura pré-definida é selecionado, configurado e também instanciado.

Seguindo estes princípios, tornou-se plausível elaborar uma estratégia de geração de sistemas orientados à aplicação que tem como sustentação teórico-científica o *Projeto Baseado em Plataformas* e a metodologia *Projeto de Sistemas Orientados à Aplicação*; contando também com o suporte técnico-financeiro dos projetos EPOS [16]

e PDSCE [17], ambos executados pelo Laboratório de Integração Software e Hardware (LISHA) da Universidade Federal de Santa Catarina (UFSC).

1.2 Objetivo

O objetivo deste trabalho é elaborar uma estratégia de geração de sistemas embutidos segundo a abordagem do *Projeto Baseado em Plataformas*. Nesta estratégia, a metodologia AOSD é utilizada como sustentação para que componentes de hardware, implementados na forma de *soft-cores*, sejam selecionados, configurados e integrados para atender juntamente com o sistema operacional os requisitos de aplicações dedicadas. Para uma aplicação codificada a partir de interfaces que especificam os serviços providos pelo sistema operacional, é possível guiar o projetista no processo de seleção dos componentes de hardware e software que irão compor o sistema a ser gerado.

Para cumprir este objetivo um levantamento do estado-da-arte foi realizado de maneira a contextualizar as premissas de AOSD no que tange a especialização de plataformas de hardware e software. A fim de avaliar o trabalho proposto e apurar resultados práticos, o sistema operacional EPOS [18]—fruto da proposta original de AOSD—juntamente com as ferramentas que perfazem sua geração, foi adaptado para que os *soft-cores* que compõem a plataforma de hardware LEON2 fossem utilizados na geração de SoCs prototipáveis em FPGAs.

1.3 Organização do Texto

No próximo capítulo, o *Projeto Baseado em Plataformas* é descrito com o objetivo de capturar o contexto no qual o presente trabalho está inserido. Primeiramente, são identificados os elementos que associam esta abordagem de projeto a sistemas computacionais embutidos. Na seqüência, estratégias de integração de IPs são relacionadas para caracterizar o reuso destes componentes em SoCs. Por fim, objetivando exemplificar o uso prático de PBD, são apresentadas questões relacionadas ao projeto e implementação de plataformas de hardware e software.

O Capítulo 3 apresenta a metodologia *Projeto de Sistemas Orientados à Aplicação*, tendo como propósito identificar os mecanismos de AOSD que são utilizados na estratégia de geração proposta neste trabalho. Neste mesmo capítulo, o artefato *Mediador de Hardware* tem sua importância justificada no escopo da metodologia AOSD e também no domínio de sistemas computacionais embutidos.

No Capítulo 4 a proposta central deste trabalho é apresentada. Primeiramente, é feito um correlacionamento das premissas de AOSD com a linguagem de descrição de componentes que permite sumarizar as características de componentes de hardware e software em uma base de dados XML. Na seqüência é demonstrado como esta informação é manipulada para que aplicações implementadas sobre os componentes descritos dêem origem a sistemas embutidos, mais especificamente, SOCs orientados à aplicação.

O capítulo seguinte descreve como a estratégia de geração de sistemas embutidos foi implementada na prática. A validação desta implementação é apresentada através dos resultados obtidos com estudos de caso nos quais foram gerados SOCs para um conjunto de aplicações.

Finalmente, no Capítulo 7, são apresentadas a conclusão do trabalho e a perspectiva sobre trabalhos futuros.

Capítulo 2

Projeto Baseado em Plataformas

Este capítulo tem por objetivo contextualizar o "Projeto Baseado em Plataformas". Primeiramente, a partir de uma análise empírica do conceito de plataformas, são identificados os elementos que associam esta abordagem de projeto a sistemas computacionais embutidos. Na seqüência, estratégias de integração de IPs são relacionadas para caracterizar o reuso destes componentes em SoCs. Por fim, objetivando exemplificar o uso prático de PBD, são apresentadas questões relacionadas ao projeto e implementação de plataformas de hardware e software.

2.1 Definindo PBD

É possível afirmar que estratégias de projeto baseadas em plataformas são comumente utilizadas em nosso cotidiano. Em geral, toda vez que produzimos algo a partir de um alicerce previamente padronizado e validado, estamos de certa forma praticando PBD (do inglês *Platform-based Design*). Na indústria automobilística por exemplo, este alicerce pode ser representado pelo chassis de veículos. Ou seja, ainda que automóveis de uma mesma linha (e.g. Corsa Sedan, Corsa Hat) difiram quanto ao modelo, motor ou acessórios, eles são geralmente projetados e montados sobre o mesmo chassis. Com isto, é possível reduzir sensivelmente tempo e custo associados ao projeto de veículos da mesma linha. Esta estratégia, adotada pelo setor automobilístico e por diversos

outros segmentos da indústria, caracteriza o principal propósito de PBD: o reuso sistemático de componentes e decisões de projeto anteriormente validadas através de alicerces bem definidos que chamamos de plataformas.

Na computação, embora o projeto baseado em plataformas esteja sendo atualmente promovido a metodologia, é comum identificar ao longo da história situações que associam o desenvolvimento de sistemas computacionais a esta abordagem de projeto. O próprio PC segue algumas das premissas básicas de PBD. A família X86, por exemplo, está "sedimentada" sobre um conjunto de atributos comuns a todos os seus membros— em geral, um processador, barramento(s) (e.g. ISA, PCI), gerenciador de interrupções e suporte a dispositivos de entrada e saída (e.g. mouse, teclado, vídeo, etc)— Vistos por um fabricante, estes atributos especificam um modelo a ser seguido, mas, sob a ótica de PBD, são componentes abstraídos por uma plataforma que, por sua vez, pode ser facilmente extendida para atender requisitos específicos de um computador pessoal. A título de exemplo, PCs utilizados na renderização de imagens são comumente acrescidos de placas gráfico-aceleradoras que agilizam este trabalho.

Seguindo este raciocínio, Vincentelli [2] define *plataforma* como uma camada de abstração a partir da qual um certo número de refinamentos pode ser realizado de forma a especializá-la para uma aplicação. Estes refinamentos são caracterizados tanto pelo grau de parametrização e configuração da plataforma, como também pela sua flexibilidade em agregar novos componentes. A especialização de uma plataforma é também conhecida como *projeto derivativo* (do inglês *derivative design*), e foca essencialmente a co-verificação do sistema [19], ou seja, a constatação de que os componentes e a plataforma escolhida "acomodaram" adequadamente o sistema projetado, atendendo assim os requisitos da aplicação que o mesmo deve suportar. Para o caso de um automóvel fabricado a partir de um chassis já consolidado, a co-verificação consiste em assegurar que a estabilidade e a dirigibilidade estão dentro dos níveis especificados para o novo veículo. Para um PC utilizado na renderização de imagens, *benchmarks* poderiam ser usados para averiguar se o desempenho alcançado com uma ou outra placa gráfico-aceleradora ficou dentro do esperado.

Como ilustrado na figura 2.1, a especialização de plataformas pode dar

origem a componentes que serão abstraídos (ou integrados) por uma outra plataforma, a qual representa um nível de abstração superior no fluxo de projeto do sistema. No âmbito de sistemas computacionais embutidos, o alicerce de projeto a ser explorado no desenvolvimento de um novo sistema é em geral constituído por duas plataformas. Do lado do hardware, uma plataforma denominada *micro-arquitetura* constitui a estrutura básica de componentes que é utilizada para implementar as funcionalidades que definirão a arquitetura do sistema [13]. Estas funcionalidades são abstraídas por uma camada de software—normalmente um sistema operacional—e então providas através de uma interface de alto-nível que é comumente referenciada como a API de software do sistema. No contexto de PBD, esta API (do inglês *Application Programming Interface*) constitui a plataforma de software a partir da qual o programa de aplicação é implementado. Por meio desta, torna-se possível re-utilizar o código da aplicação quando a arquitetura do sistema é modificada em função de métricas ou requisitos que o mesmo deve atender.

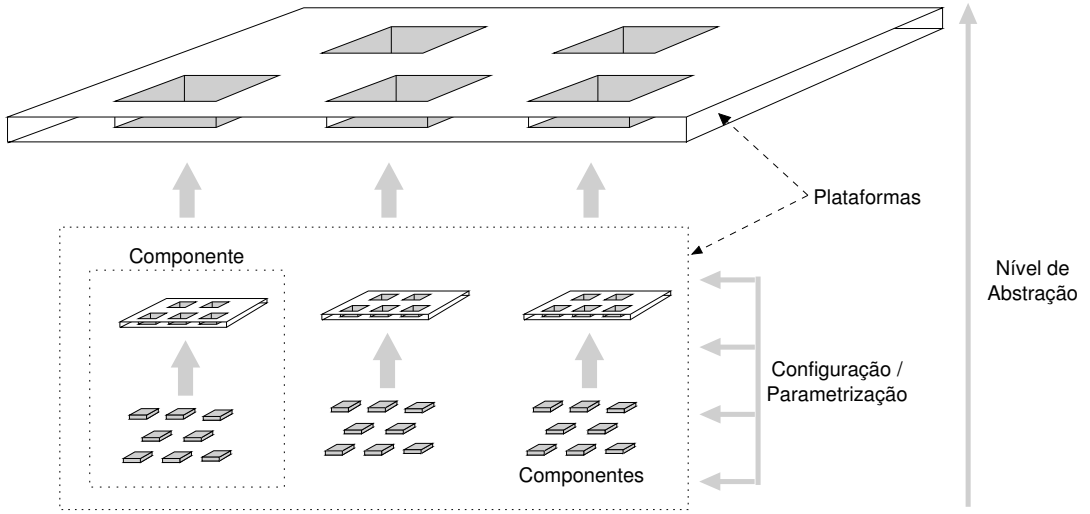

Figura 2.1: Especialização de plataformas.

A conjunção de uma arquitetura de hardware com os serviços da API que a abstrai constitui a *plataforma do sistema*. Como ilustrado na Figura 2.2 a concepção desta plataforma pode ser vista como um processo de "afunilamento" no qual duas macro-etapas de refinamento são realizadas: a *exploração do domínio arquitetural* e a *especificação da API de software*. A exploração do domínio arquitetural consiste em avaliar

dentre um campo de possibilidades uma micro-arquitetura e, por conseguinte, a respectiva especialização desta micro-arquitetura que atenda os requisitos da aplicação para qual o sistema está sendo projetado. A especificação da API de software, por sua vez, objetiva identificar um sistema operacional que abstraia a arquitetura de hardware e, por conseguinte, quais serviços deste sistema operacional devem compor o suporte operacional adequado à aplicação.

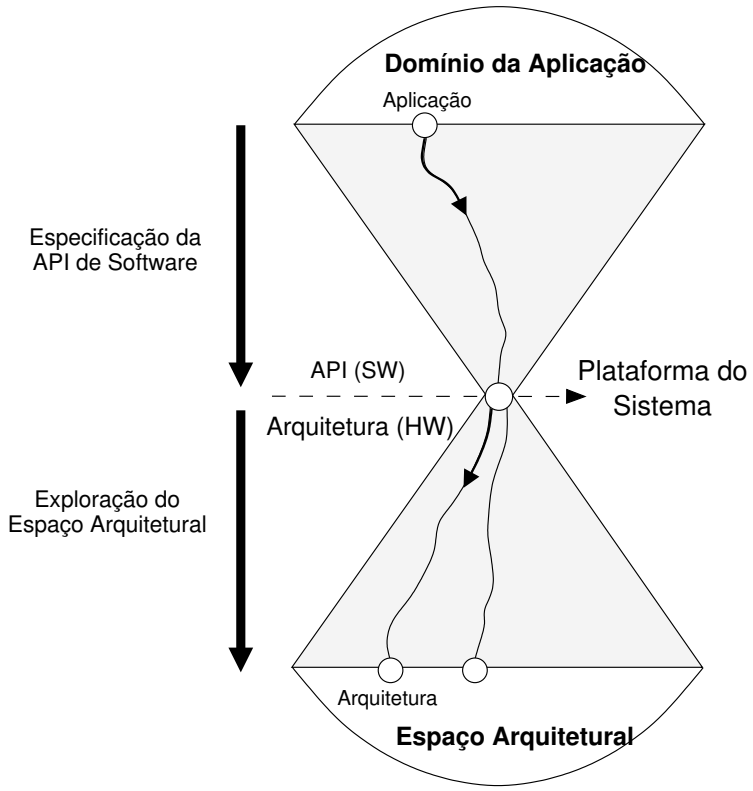

Figura 2.2: A plataforma de sistema. [2]

O (re-)uso de componentes de hardware e software previamente projetados e validados é fundamental para a geração de sistemas embutidos segundo PBD. Tanto a plataforma arquitetural como o sistema operacional que implementa a API de software podem incluir novos componentes a fim de que o domínio de aplicações coberto por estas seja ampliado. Para que isto seja possível, estes componentes devem seguir alguns padrões que comumente são ditados pelas próprias plataformas. O emprego de um novo componente de hardware na especialização de uma micro-arquitetura, por exemplo,

será viável somente se o padrão de interface seguido pelo mesmo for equivalente ao adotado pela plataforma e, conseqüentemente, pelos demais componentes. Adicionalmente, modelos de custo podem ser usados com o propósito de atribuir aos componentes—e também às próprias plataformas—valores que os quantifiquem em relação diferentes requisitos e/ou métricas de projeto. A partir destes modelos, e dependendo da granularidade dos componentes, é possível obter estimativas sobre o tamanho, performance, potência e outros fatores que impactam diretamente no sistema gerado. Em [20], Liu apresenta um modelo para estimar estaticamente, isto é, sem simulações, o consumo de energia de sistemas baseados em IPs.

Embora o uso de plataformas possa limitar a gama de aplicações que podem ser desenvolvidas sobre as mesmas, os níveis de flexibilidade e produtividade passíveis de serem alcançados justificam seu mérito. A seleção e configuração de plataformas e componentes podem ser sistematicamente automatizadas com o auxílio de ferramentas que perfazem a análise dos requisitos da aplicação e os traduzem em um conjunto regras que definirão a arquitetura do sistema. Logo, como a maior parte das decisões de projeto está implícita em plataformas e componentes já validados, o desenvolvimento fica praticamente centrado na aplicação e na subseqüente co-verificação do sistema gerado. Neste sentido, é possível afirmar que o projeto baseado em plataformas abre precedentes para o surgimento de estratégias que guiam o projetista na composição e geração automática de sistemas computacionais embutidos.

2.2 Projeto de SoCs Baseados em Plataformas

A abstração de complexidade é essêncial para que a tecnologia dos SOCs seja explorada eficientemente. Isto em parte pode ser alcançado através do reuso de componentes [21], porém a integração destes em um único *chip* impõe novos desafios ao projeto de sistemas embutidos. A simples escolha de quais componentes devem compor um sistema não é suficiente para definir sua arquitetura e a forma com que serão tratadas questões relacionadas à performance, potência, propagação de sinais no hardware, etc. Seguindo a proposta de PBD esta problemática pode ser sensivelmente reduzida, uma

vez que muitos dos detalhes de projeto e implementação estão condicionados à arquitetura de uma plataforma pré-definida. Todavia, a escolha dos componentes de um SoC é merecedora de um certo cuidado por parte do projetista. Atualmente, o termo IP tem sido amplamente utilizado para designar componentes previamente projetados e verificados que podem ser empregados em diversos sistemas computacionais mas, a inexistência de um padrão único de interconexão de IPs influi diretamente na reusabilidade destes componentes no projeto de SoCs. A inclusão (ou integração) de um IP a uma plataforma somente é possível se este for compatível com o padrão seguido pela plataforma ou se o mesmo for adaptado para que esta compatibilidade seja alcançada. Neste sentido, podemos resumir a integração de IPs a três estratégias distintas [22], as quais são descritas a seguir:

Integração baseada em padrões

Esta estratégia considera o uso de IPs cujas interfaces seguem um mesmo padrão. Com isto a integração é em tese realizada sem qualquer alteração na estrutura dos IPs. Do lado do software, podemos considerar componentes cujas interfaces, ou seguem padrões de especificação tais como POSIX [23], ou estão de acordo com a modelagem de um *framework* onde foram originalmente concebidas. No que se refere ao hardware, a padronização da interface de um componente não implica somente na assinatura de métodos ou, mais especificamente, nos sinais exteriorizados por este componente. Através deste conjunto de sinais um protocolo de comunicação é intrinsicamente implementado. Conseqüentemente, a integração de vários componentes de hardware que seguem um mesmo padrão de interface requer em geral a geração da *"lógica de cola"* (i.e. *glue-logic*) que implementa o canal (ou canais) de comunicação entre os componentes (Figura 2.3).

A padronização das interfaces de componentes de hardware é realizada segundo dois "modelos" distintos de *lógica de cola*. O primeiro deles é caracterizado por *barramentos*, cabendo destacar como exemplos os padrões AMBA [24], CoreConnect [25] e Wishbone [26]. Contratempos deste modelo são caracterizados pela possibili-

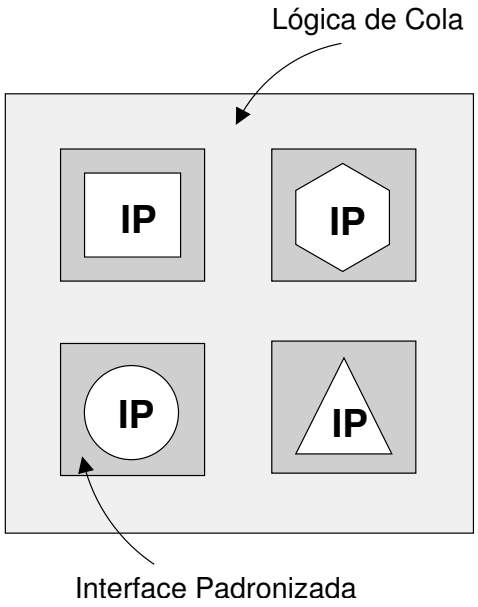

Figura 2.3: Integração de IPs com interfaces padronizadas.

dade de ainda serem necessários adaptadores de barramentos simplificados e também pela depreciação da performance quando um único barramento é utilizado no sistema sem que haja discriminação dos dispositivos de alta ou baixa latência. Neste caso, além de otimizações de acesso e de uso do barramento (e.g. *burst transfers*), barramentos secundários podem ser implementados para tratar os dispositivos com diferentes latências.

O segundo modelo de *lógica de cola* é baseado na implementação de *redes intra-chip* [27, 28], nas quais cada IP possui uma interface não-dependente de barramento que agrega somente as funcionalidades de comunicação necessária à sua integração com os demais componentes. Também conhecidas como NoCs (do inglês *Network-on-a-Chip*), estas redes possibilitam que um IP se comunique com outros IPs sem que haja a interferência de árbitros de barramentos. Os padrões VCI [29] e OCP [30] podem ser citados como os de maior expressivade no projeto de NoCs.

Adaptação de interface.

Também conhecida como *síntese de comunicação*, esta estratégia é caracterizada pelo emprego de construções conhecidas como *wrappers*, as quais viabilizam

a integração de IPs cujas interfaces seguem especificações distintas. O padrão de projeto conhecido como *adaptador* [31, 32] pode ser visto como uma definição clássica desta forma de integração, sobretudo no âmbito de componentes de software. No domínio do hardware, embora existam trabalhos que propõe o uso de padrões de projeto aos moldes de como estes são empregados no desenvolvimento de software [33], um *wrapper* constitui em geral um elemento de hardware que é fisicamente inserido entre um componente e um ponto de conexão da NoC ou barramento do sistema, realizando assim as adaptações dos sinais e do protocolo de comunicação. Um *wrapper* CoreConnect-AMBA, por exemplo, viabiliza a integração de IPs projetados segundo o padrão CoreConnect em um sistema cujo padrão de barramento é AMBA (Figura 2.4).

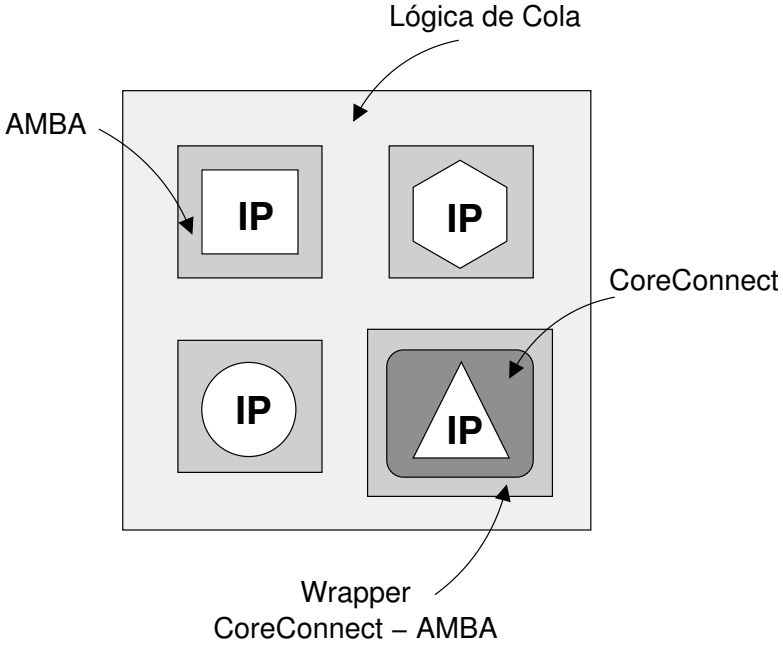

Figura 2.4: Integração de IPs usando wrappers.

Conceitualmente, esta estratégia é adequada para quando os componentes a serem integrados são *firm-cores* ou *hard-cores*, uma vez que a alteração de suas interfaces são complexas ou inviáveis. Por outro lado, nada impede que esta estratégia seja também utilizada para integrar *soft-cores*. De fato, quando um componente não implementa de maneira eficiente a separação de responsabilidades entre a interface e as

construções que definem seu comportamento, o esforço necessário para alterá-lo pode ser maior do que o exigido para integrá-lo através de um *wrapper*. Isto é comumente observado em descrições de componentes de hardware em nível RTL (do inglês *Register-transfer Level*).

Especialização de IPs

Integrar componentes através de especializações pressupõe que os IPs sejam *soft-cores* projetados e implementados segundo alguma estratégia de modelagem; em geral, mecanismos e paradigmas já consagrados pela comunidade de software e que agora vêm sendo extendidos para o projeto de hardware em um esforço de explorar, neste domínio, as potencialidades alcançadas com componentes de software. Apesar de linguagens como VHDL [34] possibilitarem níveis consideráveis de reuso e configurabilidade, muito deste esforço está focado especificamente em trazer para o domínio do hardware o paradigma de *Orientação a Objetos* [35, 36]. Com isto, a integração de um IP poderia ser feita, por exemplo, através da especialização de uma super-classe em uma sub-classe que realiza a interface compatível com a da plataforma onde o componente será incluído.

Não obstante, o uso do paradigma de orientação a objetos na modelagem do hardware tem se tornado uma realidade somente no contexto do projeto em nível de sistema onde, o advento de linguagens de descrição de sistema como SYSTEMC [37], tem consolidado a possibilidade de abstrair componentes de hardware em uma abordagem de alto-nível. Porém, grande é o esforço para mapear um componente descrito em nível de sistema para a sua respectiva representação RTL e com isto viabilizar sua síntese. Como enfatizado por Martin [38], mesmo com o suporte de compiladores e ferramentas cada vez mais eficientes, é justamente o uso conjunto e racional de linguagens de descrição de sistema e de linguagens de descrição de hardware que atribui ao projeto um menor risco de fracasso.

2.2.1 Plataformas de Hardware

Em geral, as plataformas de hardware existentes para sistema embutidos incluem microprocessadores, memória e uma estrutura de comunicação que é utilizada para interconectar componentes adicionais. Dependendo do domínio de aplicações em que serão empregadas é possível projetar plataformas que incluem recursos que possibilitam ao projetista atender de maneira pontual os requisitos de sistemas implementados sobre as mesmas. A plaforma *Nexperia* [39] da Philips caracteriza um bom exemplo de micro-arquitetura que foi projetada para atender um domínio específico de aplicações, no caso, o de sistemas multimídia. Para isto, inclui um micro-processador TriMedia [40] e uma série de outros dispositivos dedicados à codificação de decodificação de áudio e vídeo digitais.

Diversos SOCs têm sido criados a partir de especializações da *Nexperia*, dentre os quais cabe citar o PNX8500 [41], utilizado em terminais de acesso de TV Digital (i.e. *set-top-boxes*), e os da série PNX78XX, unidades centrais de gravadores de CDs e DVDs. Todavia, como investigado por Goering [42] o desenvolvimento desta plataforma, bem como a concretização de sua primeira especialização, não foram triviais. Durante um período de três anos foi necessário o envolvimento de 300 (trezentos) engenheiros para que a *Nexperia* e o PNX8500 fossem concluídos. Tal experiência corrobora a realidade de que os principais desafios de PBD são a especificação e implementação de plataformas que possam ser (re-)utilizadas em uma gama justificável de projetos e também a carência de estratégias para especializá-las em curtos espaços de tempo [3].

O uso de PLDs constitui umas das principais alternativas para flexibilizar a geração de SOCs. Em uma micro-arquitetura estes dispositivos são utilizados para instanciar componentes de hardware que não foram originalmente incluídos nesta plataforma. Implementados como *soft-cores* ou *firm-cores*, estes componentes constituem portanto uma forma de ampliar o domínio de aplicações coberto pelo conjunto de sistemas gerados a partir de uma micro-arquitetura. Exemplificando, a possibilidade de instanciar um decodificador MP3 em um SOC nos permite afirmar que este cobre um nicho específico de aplicações de áudio. É claro que esta funcionalidade também poderia

ser realizada em software, mas isto exigiria uma capacidade de processamento que, tal como o decodificador, pode não ter sido incluída na plataforma originalmente.

Algumas plataformas podem ser inteiramente implementadas sobre uma PLD. Dentro do que Schirrmeister classifica como *totalmente programáveis* [43], estas plataformas consideram em geral FPGAs como a tecnologia final para a instanciação do sistema. Um exemplo são as VIRTEXII-PRO da Xilinx [44], as quais combinam lógica programável a um micro-processador IBM PowerPC fixo (i.e. *hard-wired*). Sobre estas plataformas projetistas integram diferentes IPs de maneira a compor o sistema almejado. Ferramentas conhecidas como *Construtores de* SOC*s* (i.e. *SoC Builders*) podem ser utilizadas para automatizar a geração da *lógica de cola* necessária para interconexão dos componentes que, normalmente, devem seguir um mesmo padrão de interface. As outras classes de plataformas de hardware indentificadas por Schirrmeister são:

- *Plataformas de aplicação completa*: as plataformas desta classe são otimizadas para domínios específicos de aplicações e podem ser entregues aos projetistas já com algum suporte para o desenvolvimento de sistemas sobre as mesmas. Um exemplo é a própria *Nexperia* da Philips, a qual possui um *kit* de desenvolvimento com compiladores, sistema operacional, exemplos e documentação para o desenvolvimento de software;

- *Plataformas baseadas em processador*: são plataformas que incluem um ou mais micro-processadores específicos e focam justamente a execução do software de sistema sobre os mesmos. Um exemplo desta classe de plataformas é o micro-processador *Xtensa6* da empresa Tensilica [45]. Além de poder ser conectado a outros componentes de hardware, o *Xtensa* pode ter seu conjunto de instruções otimizado de acordo com o domínio de aplicações onde será empregado;

- *Plataformas baseadas em comunicação*: as plataformas desta classe provêm ao projetista uma estrutura de comunicação na qual diferentes componentes são conectados para compor um sistema para uma aplicação. A matriz de interconexão *SiliconBackplane III* da Sonics caracteriza um bom exemplo de plataforma baseada

em comunicação [46]. Além de sua estrutura central de comunicação, esta plataforma inclui adapatadores de IPs denominados *agentes configuráveis* que agilizam agilizam trabalho de integração dos componentes.

Evidentemente, algumas plataformas podem ser enquadradas em mais de uma das classes apresentadas. Micro-arquiteturas desenvolvidas inteiramente a partir de *soft-cores* cujas especializações são instanciáveis em FPGAs podem ser implementadas em torno de um ou mais processadores e ainda contarem com uma estrutura de comunicação que permite integrar componentes para que diferentes domínios de aplicações sejam cobertos pela plataforma. Dentre estas, cabe destacar as micro-arquiteturas LEON2 [47] e OPENRISC [48] que, embora sejam constantemente designadas como SOCs, são de fato plataformas a partir das quais diferentes sistemas podem ser gerados. A seguir estas duas micro-arquiteturas são apresentadas para enfatizar o emprego de *soft-cores* em plataformas de hardware.

OpenRisc

A micro-arquitetura OPENRISC é mantida pelo grupo OpenCores, uma iniciativa da comunidade de hardware para o desenvolvimento de *soft-cores* de domínio público. O objetivo é obter uma plataforma que seja facilmente extendida com novos componentes de hardware e assim versátil o suficiente para suportar diferentes domínios de aplicações. Também conhecida como OR1K, esta plataforma inclui um processador RISC de 32 bits com pipeline de 5 estágios, *cache* de dados e instruções, suporte a memória virtual e algumas funções básicas de `DSP`. A Figura 2.5 ilustra o diagrama de componentes desta micro-arquitetura. Além do micro-processador e de uma `MMU`, a OR1K também inclui um controlador de interrupções programável (`PIC`), uma unidade de depuração (`DBG`), um contador de tempo (`TIMER`) e uma unidade avançada de gerenciamento de energia (`PM`). Com a exceção do contador de tempo estes outros dispositivos possuem interfaces externas para que suas funcionalidades sejam realizadas em conjunto com outros componentes. A `PM I/F` é usada para o gerenciamento do consumo de energia dos periféricos; a `DB I/G` tem a ela conectados dispositivos que exteriorizam dados

para a sua depuração; e a `INT I/F` implementa as linhas de interrupção de periféricos conectados ao núcleo.

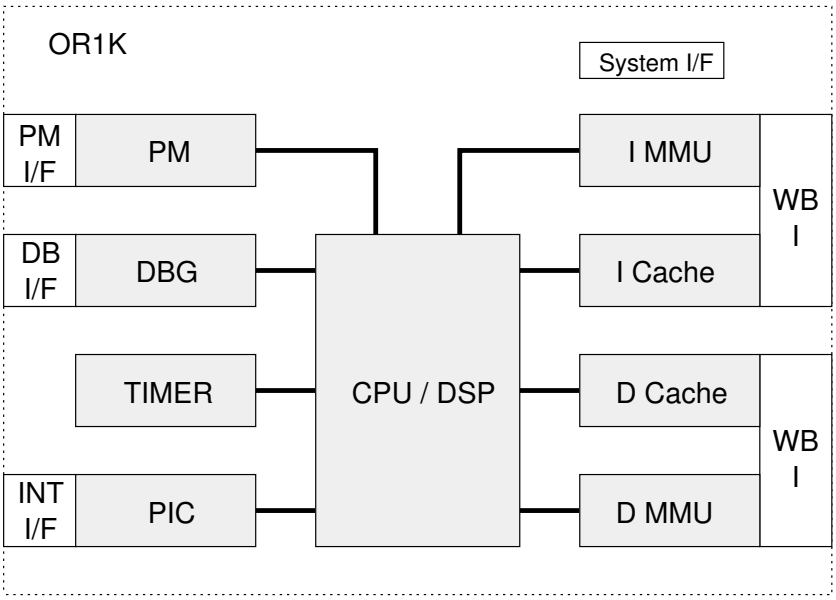

Figura 2.5: Micro-arquitetura OpenRisc

A inclusão de novos componentes à OR1K respeita o padrão de interface WISHBONE e pode ser através das interfaces `WB I`, as quais mapeiam em memória os dispositivos. No entanto, mesmo o fato de seguir um padrão ainda não atribuiu a esta micro-arquitetura a versatilidade esperada por seus projetista. Embora existam inúmeros desenvolvedores trabalhando em aprimoramentos e novos componentes, o próprio caráter público do projeto talvez seja o maior entrave para a rápida especialização desta micro-arquitetura e também para a instanciação do sistema gerado. Não existe um grande cuidado com a organização e manutenção do código-fonte, sendo necessário muitas vezes intervenções do desenvolvedor para garantir primeiramente a síntese e posteriormente o funcionamento do sistema. Outro problema é que a linguagem na qual a plataforma está implementada não facilita a modularidade e configuração de componentes. Em VERILOG não há o conceito de pacotes, não existem diretivas para a configuração e replicação de estruturas e o recurso de parametrização é muito pobre, sendo necessário sobrescrever constantes ao longo do processo de síntese para que os componentes sejam

pré-processados corretamente.

Leon2

Esta micro-arquitetura foi originalmente projetada em 1997 por Jiri Gaisler em um projeto de código aberto da Agência Espacial Européia (ESA). O principal objetivo, desenvolver um processador de alta-performance para uso em sistemas aero-espaciais que primasse pela compatibilidade de software, baixo custo e, sobretudo, a tolerância a falhas [7]. Para permitir a rápida prototipação e o porte para diferentes tecnologias, a micro-arquitetura LEON2 foi implementada inteiramente a partir de *soft-cores*, o que lhe atribui adicionalmente grande flexibilidade na configuração. Além de seus componentes possuirem propriedades que são ajustáveis de acordo com as decisões do projetista, a parametrização da *lógica de cola* usada para integrá-los também permite especificar quais deles serão instanciados no sistema.

A Figura 2.6 ilustra a micro-arquitetura LEON2. A plataforma conta com um micro-processador SPARCV8 e, conectados diretamente a este, há uma `MMU`; uma interface para unidades de ponto flutuante (`FPU`); uma interface para co-processadores (`CP`); e uma memória local que é usada para extender sua *cache*. A *lógica de cola* utilizada para interconexão de dispositivos periféricos é baseada no padrão de barramento AMBA. Originalmente, estes periféricos incluem um bloco de memória intra-chip de tamanho configurável; uma ponte `PCI`; uma interface de rede (`NIC`); e uma unidade de depuração através da qual é possível realizar a execução passo-a-passo de programas e inspecionar o conteúdo da memória e registradores. Os demais periféricos da LEON2 são contadores de tempo, interfaces seriais (`UART`), portas de I/O e o controlador de interrupções (`IC`); estão conectados ao barramento APB que, ao contrário do barramento AHB, é utilizado para a conexão de dispositivos de alta latência.

Comparada à micro-arquitetura OPENRISC, a LEON2 oferece níveis muito superiores de configurabilidade. Além de estar sob os cuidados de uma instituição privada que a disponibiliza também em versões comerciais, os recursos da linguagem VHDL para a modularização e parametrização dos componentes são extesivamente

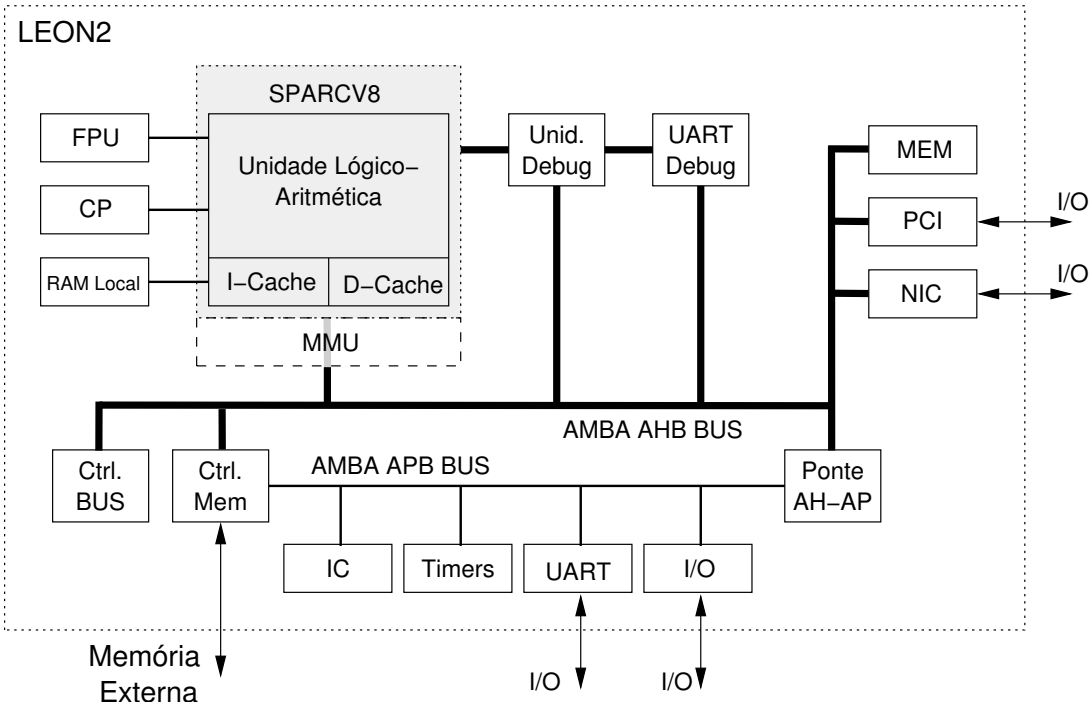

Figura 2.6: Micro-arquitetura Leon2

explorados. Adicionalmente, em um esforço para garantir sua portabilidade, a plataforma inclui implementações de blocos memória genéricos que facilitam a sua instanciação de suas especializações em diferentes tecnologias de FPGAs.

2.2.2 Plataformas de Software

Conceitualmente a plataforma de software de um sistema embutido é exteriorizada por uma API pela qual uma ou mais aplicações podem explorar de forma organizada os recursos providos pelo hardware. Esta API é realizada em geral por um sistema operacional e portanto abstrai não só detalhes do hardware mas também as próprias funcionalidades providas pelo sistema operacional. Espera-se que uma aplicação desenvolvida a partir desta API seja executada sem qualquer modificação em todas as plataformas de hardware suportadas pelo sistema operacional que a realiza. Isto torna *portabilidade* fator determinante no desenvolvimento de sistemas operacionais.

Estratégias de portabilidade estão tradicionalmente concentradas em dois flancos: *Máquinas Virtuais* (VM) e *Camadas de Abstração de Hardware* (HAL). Sobre máquinas virtuais, cabe ressaltar que o sistema operacional se estende do hardware à aplicação [49] e, portanto, uma máquina virtual constitui de fato a porção de software do sistema operacional que é dependente do hardware, ao passo que promove a portabilidade dos componentes sobre ela implementados. A principal deficiência deste mecanismo é o *overhead* gerado nas operação de tradução de código intermediário para código nativo. Ainda sim, sistemas como o JAVAOS [50] têm promovido a *Máquina Virtual Java* (JVM) como um atraente mecanismo de portabilidade, mas não diferente de qualquer outro sistema baseado em máquinas virtuais, a JVM deve ser re-implementada para cada nova plataforma de hardware.

Camadas de abstração de hardware, por sua vez, constituem um substrato dependente de arquitetura que encapsula detalhes específicos da plataforma e disponibiliza para o sistema operacional, através de uma interface comum, os recursos implementados pela plataforma de hardware. Ainda que não gere o *overhead* característico das máquinas virtuais, uma HAL pode incluir dependências arquiteturais que podem depreciar a portabilidade do sistema operacional. Um exemplo clássico de tais dependências é encontrada no gerente de memória de sistemas UNIX [51]. A chamada de sistema *brk*, usada no re-dimensionamento da área de dados de um processo, parte sempre do princípio de que um modelo paginado de memória está em uso e, em função disto, de que a plataforma de hardware dispõe de uma MMU (do inglês *Memory Management Unit*). Isto compromete diretamente o porte destes sistemas para plataformas que não dispõem deste componente de hardware e o modelo paginado não é necessário.

O sistema UCLINUX [52] caracteriza um exemplo de sistema operacional embutido cuja portabilidade está fundamentada em uma HAL. Além de constituir uma camada monolítica de software, a HAL deste sistema está comprometida em suportar funcionalidades que foram herdadas do sistema operacional LINUX. Dentre estas, a abstração de um sistema arquivos impacta não só no tamanho do sistema operacional gerado mas também em toda a infraestrutura de inicialização do sistema e carga das aplicações.

Como evidenciado por Neville [53] a problemática de HALs e VMs é ainda maior quando as arquiteturas de hardware a serem abstraídas são baseadas em SoCs. A diversidade de dispositivos existentes nestas plataformas o fez concluir que as técnicas tradicionais de especificação de interfaces software-hardware estão muito aquém do almejado "*plug-and-play*". Tal afirmação ganha mais força quando o hardware a ser abstraído é instanciado em dispositivos de lógica programável como as FPGAs. Com estes dispositivos é possível modificar a arquitetura do hardware em um curto espaços de tempo [54], dificultando assim ainda mais a portabilidade dos componentes do sistema operacional e, conseqüentemente, da aplicação implementada sobre este.

Componentes de software constituem a saída adotada por muitos projetistas para não só minimizar os contratempos das tradicionais estratégias portabilidade mas também para viabilizar a composição de sistemas operacionais que incluam apenas os serviços requisitados por uma aplicação. Todavia, uma vez que o projeto destes componentes não esteja apoiado na engenharia de software e/ou em metodologias específicas de desenvolvimento, a manutenção do sistema e sua própria utilização podem ser comprometidas.

O sistema eCos [55] da RedHat possibilita a seleção das abstrações que devem integrar o sistema operacional. Todavia, seus componentes não são modelados segundo qualquer paradigma ou metodologia da engenharia de software e os diferentes aspectos de cenário nos quais podem ser utilizados não são fatorados. Conseqüentemente, o número de componentes que implementam as abstrações do sistema operacional pode ser tão grande quando o número de cenários de aplicação em que estes podem ser utilizados. Isto dificulta a seleção dos componentes uma vez que o sistema eCos não possui um mecanismo de inferência de componentes que possibilite extrair do código-fonte da aplicação informações que guiem o projetista na seleção dos componentes do sistema. Outra conseqüencia direta de sua modelagem esta realacionada à interface software-hardware. Tal como o uClinux, este sistema tem sua portabilidade fundamentada em uma HAL, a qual embora seja baseada em componentes de software, não é gerada de acordo com os requisitos da aplicação, podendo assim agregar código desnecessário ao sistema. A título de exemplo, o sistema eCos tem como certa a existência de uma UART em SoCs

gerados a partir da micro-arquitetura LEON2.

Alternativamente, metodologias que promovem a engenharia de domínio no projeto de componentes de software têm sido empregradas no desenvolvimento de sistemas operacionais [56, 57]. A metodologia *Projeto de Sistemas Orientados à Aplicação* (AOSD) combina princípios de *Programação Orientada a Objetos* (POO) [58] com *Programação Orientada a Aspectos* (POA) [59] e *Metaprogramação Estática* (MPE) [60] para guiar o desenvolvimento de componentes de software adaptáveis a diferentes cenários de aplicação. Um sistema operacional projetado segundo as premissas de AOSD oferece ao projetista famílias de abstrações cujos membros são selecionados, configurados e instanciados de acordo com os requisitos da aplicação. A modelagem destas famílias ainda oferece meios para se proceder com a inferência automática destes componentes, fazendo com que o desenvolvimento esteja centrado na própria aplicação.

Objetivando demonstrar em maiores detalhes as potencialidades de metodologias de projeto de componentes de software, o próximo capítulo apresenta em maiores detalhes a metodologia AOSD cujos conceitos foram fundamentais para a realização deste trabalho.

Capítulo 3

Projeto de Sistemas Orientados à Aplicação

Neste capítulo são apresentados os principais conceitos que definem a metodologia "Projeto de Sistemas Orientados à Aplicação" (AOSD). Na seqüência o artefato de software *Mediador de Hardware* é descrito com o objetivo de evidenciar seu papel em sistemas operacionais desenvolvidos segundo a metodologia AOSD.

3.1 A Metodologia AOSD

Projeto de Sistema Orientados à Aplicação (AOSD) [4] propõe estratégias para proceder com uma engenharia de domínio no sentido de identificar e abstrair através de componentes de software as entidades significativas de um dado domínio de aplicação. Em princípio, esta decomposição pode ser alcançada usando o paradigma de orientação a objeto [58], porém, alguns pontos devem ser considerados. Primeiramente, esta abordagem propõe a organização de uma hierarquia de classes que caracteriza como os objetos são especializados. Além de levar ao problema da "classe-base frágil" [61], a especialização e o uso de uma abstração se dá apenas na presença de sua versão mais "genérica" (i.e. *super-classes*). Aplicando análise de variabilidade segundo *Projeto Baseado em Famílias* [62], AOSD promove a modelagem de abstrações independentes organiza-

das como membros de famílias. Certamente, os membros destas famílias podem ainda ser modelados como especializações, mas como sugerido por Habermann [49], isto não é uma regra, já que a especialização de um objeto pode ser modelada como um membro de família. Desta maneira, a AOSD contorna as restrições da decomposição orientada a objetos, favorecendo de fato a decomposição orientada à aplicação.

Uma segunda e importante diferença entre estas duas abordagens de decomposição está relacionada às dependências que emanam do ambiente no qual as abstrações serão utilizadas. A análise de variabilidade proposta pela decomposição orientada a objetos não leva em conta tais dependências e, conseqüentemente, as abstrações que as incorporam tem uma pequena chance de serem reutilizadas em um novo cenário. Considerando que um sistema operacional orientado à aplicação esteja comprometido com tais mudanças de cenário, tais dependências poderiam depreciar sua modelagem toda vez que uma nova aplicação fosse concebida. A fim de reduzir dependências de ambiente e permitir que abstrações sejam usadas em diferentes cenários, AOSD agrega ao processo de decomposição o conceito chave da *Programação Orientada a Aspectos* [59]: a separação de aspectos. Através deste conceito é possível identificar as variações de cenários que, ao invés de justificarem a crição de novos membros de famílias, são tratadas como aspectos de cenário. Por exemplo, ao invés de se modelar um novo membro para uma família de mecanismos de comunicação que esteja apto a operar em um ambiente *multithreading*, modela-se *multithreading* como um aspecto de cenário. Este aspecto, quando ativado, bloquearia o mecanismo de comunicação (ou algumas de suas operações) quando em uma seção crítica.

Com base nestas premissas, AOSD tem guiado a engenharia do domínio de sistemas operacionais (Figura 3.1), dando origem a componentes que estão fundamentados em quatro elementos básicos de AOSD: famílias de abstrações; adaptadores de cenário, interfaces infladas e *propriedades configuráveis* (no inglês *configurable features*).

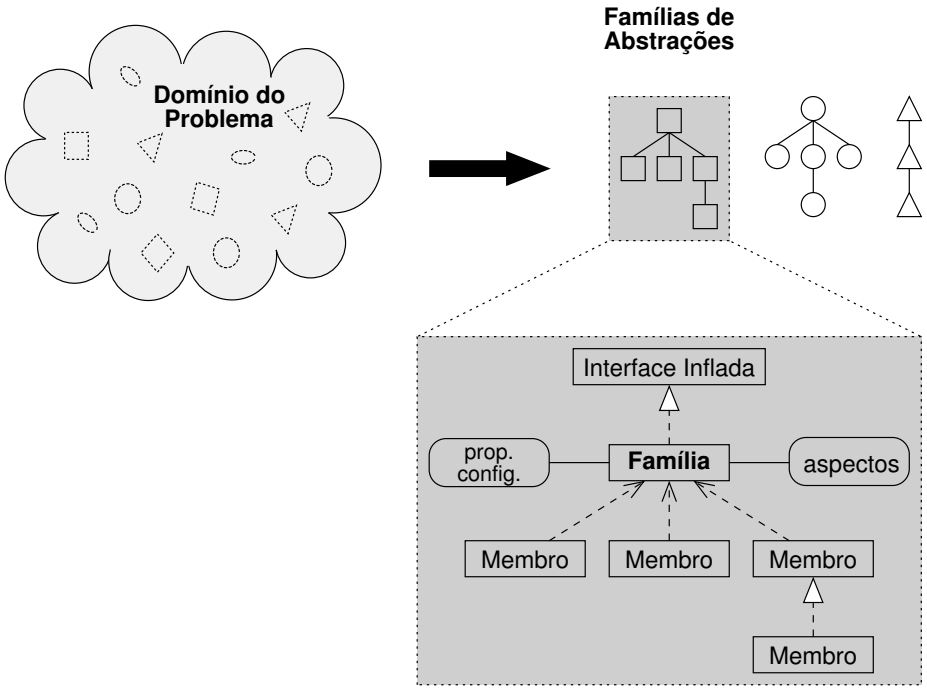

Figura 3.1: Visão da decomposição de domínio proposta por AOSD

Famílias de abstrações

Durante a decomposição de domínio, abstrações são definidas a partir de entidades do domínio e agrupadas em famílias de acordo com suas propriedades comuns. Ao mesmo tempo, a separação de aspectos é usada para que estas abstrações sejam modeladas como componentes independentes do cenário em que serão aplicadas.

Adaptadores de cenário

Como citado anteriormente, AOSD promove a fatoração de *aspectos* que mantêm assim as abstrações do sistema operacional independentes do cenário em que serão empregadas. Entretanto, para que esta estratégia tenha resultado, alguns mecanismos devem ser utilizados para que os aspectos fatorados sejam aplicados transparentemente sobre as abstrações do sistema. O caminho tradicional para se garantir isto é através do uso de *aspect weavers* [63], entretanto, os adaptadores de cenário [64, 65] promovidos pela metodologia oferece as mesmas potencialidades sem que ferramentas

externas sejam utilizadas. Estes adaptadores adequam uma abstração a um dado cenário perfazendo as devidas adaptações entre ambiente e componente sem gerar *overhead*.

Interfaces infladas

Interfaces infladas contemplam as propriedades de todos os membros de uma família, criando uma visão única da família como se esta fosse um "super componente". Isto permite aos programadores de aplicação escreverem suas aplicações a partir de uma interface única, deixando a decisão de qual membro da família será usado até o momento em que o sistema será gerado. A associação de interfaces infladas a um membro específico da família pode assim ser feita automaticamente por ferramentas de configuração que identificam quais propriedades da família foram utilizadas, escolhem o membro que implementa estas propriedades e por conseguinte agregam o membro escolhido ao sistema operacional.

Propriedades configuráveis

Outro importante elemento de AOSD são as *propriedades configuráveis* cujo propósito é designar propriedades dos componentes que podem ser configuradas em função dos requisitos da aplicação. Um exemplo é o tamanho do segmento de pilha a ser alocado para cada *thread* ou processo gerenciados pelo sistema. *Propriedades configuráveis* não estão restritas apenas à especificação de valores. Adicionalmente, *propriedades configuráveis* podem incorporar construções genericamente programadas [66]. Quando ativadas, estas *propriedades configuráveis* realizam funcionalidades que um dado componente é capaz de prover em um cenário específico de aplicação.

3.2 Mediadores de Hardware

Como apresentado, a proposta original de AOSD é guiar o projeto e implementação de sistemas operacionais orientados à aplicação a partir de componentes de software que podem ser adaptados a diferentes cenários de execução. Objetivando

manter estes componentes “alheios” a detalhes arquiteturais das plataformas de hardware sobre as quais as aplicações serão executadas, AOSD sugere que as abstrações do sistema interajam com o hardware através de *Mediadores de Hardware*. Como artefato de portabilidade, a principal idéia por trás destes mediadores é o de não constituirem uma camada de abstração de hardware universal ou uma máquina virtual (VM), mas sim, sustentar um “*contrato de interface*” entre as abstrações do sistema operacional e os componentes do hardware.

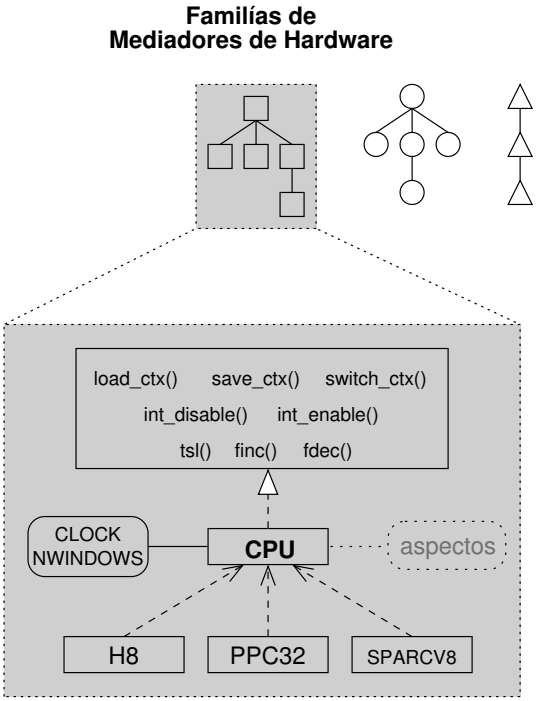

Figura 3.2: Família de mediadores de hardware de CPUs.

Diferentemente das tradicionais HALs (do inglês *Hardware Abstraction Layer*), mediadores de hardware não constituem uma camada monolítica de software que encapsula todos os recursos da plataforma, cada componente de hardware é mediado via seu próprio mediador, garantindo assim a portabilidade das abstrações que o utilizam sem gerar dependências desnecessárias. Tal como abstrações em AOSD, mediadores de hardware são organizados em família cujos membros representam entidades significativas do domínio de hardware. A Figura 3.2 exemplifica a família de mediadores `CPU`, a qual

inclui os membros `H8`, `PPC32` e `SPARCV8`. Como ilustrado, a interface implementada por estes membros provê métodos para carregamento, salvamento e troca de contexto; métodos para desativar e ativar as interrupções e métodos para realizar atomicamente operações de incremento e decremento. Nenhum aspecto de cenário é aplicado sobre a família e duas *propriedades configuráveis* são implementadas: *CLOCK* e *NWINDOWS*. A propriedade configurável *CLOCK* contém a freqüência do relógio da `CPU` e a propriedade *NWINDOWS*, peculiar ao membro SPARCV8, especifica o número de janelas de registradores implementadas por esta `CPU`.

Esta estratégia de modelagem garante que o sistema incluirá somente código-objeto dos mediadores necessários para dar suporte à aplicação. Além disto, segundo a proposta de AOSD, propriedades ortogonais aos componentes são fatoradas como *aspectos* que podem ser aplicados aos membros das famílias sem afetar suas interfaces e sem agregar código desnecessário. Um exemplo da aplicação de aspectos sobre mediadores é a necessidade de famílias como `UART` (do inglês *Universal Asynchronous Receiver-Transmitter*) e `NIC` (do inglês *Network Interface Card*) operarem em modo exclusivo de acesso. Isto pode ser alcançado através da "aplicação" de um aspecto de controle de acesso sobre estas famílias.

O elemento *propriedade configurável* atribui a mediadores de hardware grande flexibilidade na composição de interfaces software-hardware que estão comprometidas em atender os requisitos da aplicação. Além de abstrairem componentes de hardware, mediadores podem suprir funcionalidades que o hardware não implementa, ou que devem ser realizadas pelo software por decisões de projeto. Um exemplo é a geração de códigos `CRC` por uma *propriedade configurável* implementada por um mediador de um dispositivo de rede—também chamados de `NIC`s (do inglês *Network Interface Card*. A fim de diagnosticar a integridade dos quadros de dados (i.e. *frames*) em uma rede *ethernet*, geradores de código `CRC` podem ser utilizados; algumas NICs implementam estes geradores em hardware; em outros casos, o software é o responsável por suprir esta funcionalidade.

Com relação a implementação de mediadores de hardware, os recursos de meta-programação estática providos pela linguagem de programação C++ [67] favo-

rece a implementação de mediadores na forma de classes parametrizadas cujos métodos são declarados `inline` e definidos a partir de instruções *assembly*. Com isto, mesmo o *overhead* decorrente de chamadas de função é evitado, maximizando assim a performance do sistema. A Figura 3.3 ilustra a implementação do método `tsc` do mediador de `CPU` da arquitetura `IA-32`, que tem por objetivo retornar o valor corrente do registrador *time-stamp counter*. A invocação deste método na forma

```
register unsigned long long tsc = IA32::tsc();
```

produziria uma única instrução de máquina: `rdtsc`. Neste exemplo é possível evidenciar que o uso de construções meta-programadas na implementação de mediadores de hardware faz com que o código a ser gerado para estes componentes seja na verdade dissolvido no código das abstrações tão logo o contrato de interface para com o sistema seja atendido. Em outras palavras, um mediador de hardware entrega a funcionalidade de um componente de hardware através de uma interface orientada ao sistema operacional. A existência de uma interface software-hardware é vislumbrada somente durante a modelagem do sistema, mas não no código gerado para o mesmo.

```
class IA32 {
    // ...
  public:
      static unsigned long long tsc() {
          unsigned long long tsc;
          __asm__ __volatile__ ("rdtsc" : "=A" (tsc) : );
          return tsc;
      }
    // ...
};
```

Figura 3.3: Fragmento de mediador de CPU da arquitetura IA-32.

Além de promoverem portabilidade, mediadores de hardware também exercem um papel fundamental na estratégia de geração proposta neste trabalho. O próximo capítulo apresenta esta estratégia e descreve como mediadores de hardware são uti-

lizados na especialização de plataformas de hardware considerando requisitos extraídos das aplicações para as quais sistemas devem ser gerados.

Capítulo 4

Descrição e Especialização de Plataformas Segundo AOSD

Conforme apresentado no Capítulo 2, um sistema embutido orientado à aplicação pode ser visto como um produto da especialização de duas plataformas distintas: a plataforma micro-arquitetural e a API de um sistema operacional. Objetivando automatizar este processo empregando conceitos da metodologia Projeto de Sistemas Orientados à Aplicação (AOSD) optou-se pela utilização de uma linguagem de descrição que permitisse descrever os componentes das plataformas em uma base de dados XML a partir da qual um mecanismo de gestão e manipulação de conhecimento guia o projetista na geração de sistemas orientados à aplicação. O presente capítulo apresenta a linguagem utilizada na descrição destes componentes e o mecanismo idealizado para a especialização de plataformas pré-projetadas.

4.1 Descrição de Componentes

Visando originalmente a implementação de um ambiente de geração de sistemas operacionais dedicados, uma linguagem de descrição de componentes baseada em XML foi proposta por Tondello [68] para descrever componentes projetados segundo as principais premissas da metodologia AOSD. De acordo com o que foi abordado no

Capítulo 3, estes componentes estão organizados em famílias cujas interfaces são conhecidas como *interfaces infladas*. A DTD (do inglês *Document Type Definition*) que define a estrutura XML a ser seguida na descrição de famílias e seus membros é apresentada no apêndice da página 95. Neste trabalho, as descrições de componentes realizadas a partir desta linguagem constituem uma base de dados que fornece informações a respeito das interfaces e funcionalidades realizadas pelos componentes das plataformas como também as relações de dependência entre estes.

Considerando a DTD supracitada, temos que a descrição de uma família é feita segundo o elemento

```
family (common*, interface, member+, (feature | dependency | trait)*)
```

Este elemento também pode ser representado graficamente por um diagrama de ferrovia ilustrado na Figura 4.1. Como pode ser observado, o descritivo de uma família inicia-se com sua interface inflada. Através do elemento *common* são descritos os tipos e constantes comuns a todos os membros da família e, através do elemento *interface*, são descritos os construtores, métodos, tipos e constantes realizados por cada membro da família. A descrição de uma família ainda contém a descrição individual de cada um de seus membros; um conjunto opcional de elementos *trait* e/ou *feature*, usados para descrever suas *propriedades configuráveis* e, com o propósito de explicitar suas dependências para com outros componentes, pode incluir também o elemento *dependency*. Tanto as dependências como também as *propriedades configuráveis* de uma família são aplicáveis a todos os membros da família.

A descrição de membros de famílias é feita segundo o elemento

```
member (super*, interface, property*, (feature | dependency | trait)*)
```

O diagrama de ferrovia que representa esta construção é apresentado na Figura 4.2. O elemento *super* é utilizado para descrever relações de herança entre os membros de uma mesma família caso existam; o elemento *interface* descreve a interface que o membro realiza, ou seja, uma realização parcial ou total da interface inflada da família; e as construções *trait*, *feature* e *dependency* descrevem, respectivamente, *propriedades configuráveis* e relações de dependência pertinentes exclusivamente ao membro. Como pode ser observado, a descrição de um membro ainda pode conter o elemento *property*, o qual descreve

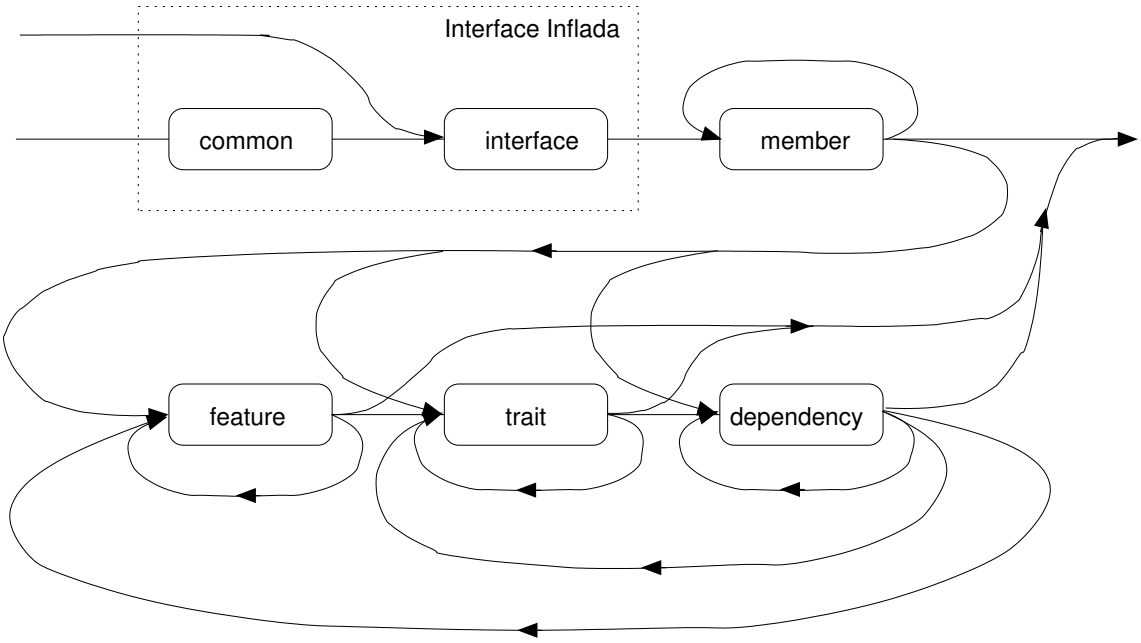

Figura 4.1: Diagrama de ferrovia da descrição de famílias.

as propriedades não funcionais (no inglês *non-functional properties*) do componente. Estas propriedades podem ser implementadas por meio de modelos de custo que quantificam o componente em termos de consumo de energia, tamanho, performance, etc. Estes modelos consideram mensurações realizadas em experimentos envolvendo as plataformas e também estimativas de complexidade e esforço computacional realizadas pelo projetista.

O apêndice da página 98 mostra a descrição da família *Thread*. Esta descrição inclui informações que permitem identificar quais membros a família possui—*Exclusive_Thread*, *Cooperative_Thread*, *Concurrent_Thread* e *Priority_Thread*—e como sua interface é realizada por estes membros. Neste caso, a organização hierárquica que estabelece esta realização segue um modelo incremental [49] e, portanto o membro *Priority_Thread* realiza todos os métodos da interface inflada da família. Adicionalmente, podem ser observadas as propriedades funcionais e dependências desta família; estes dados, bem como as construções da linguagem que foram utilizadas para descrevê-los, são apresentados a seguir.

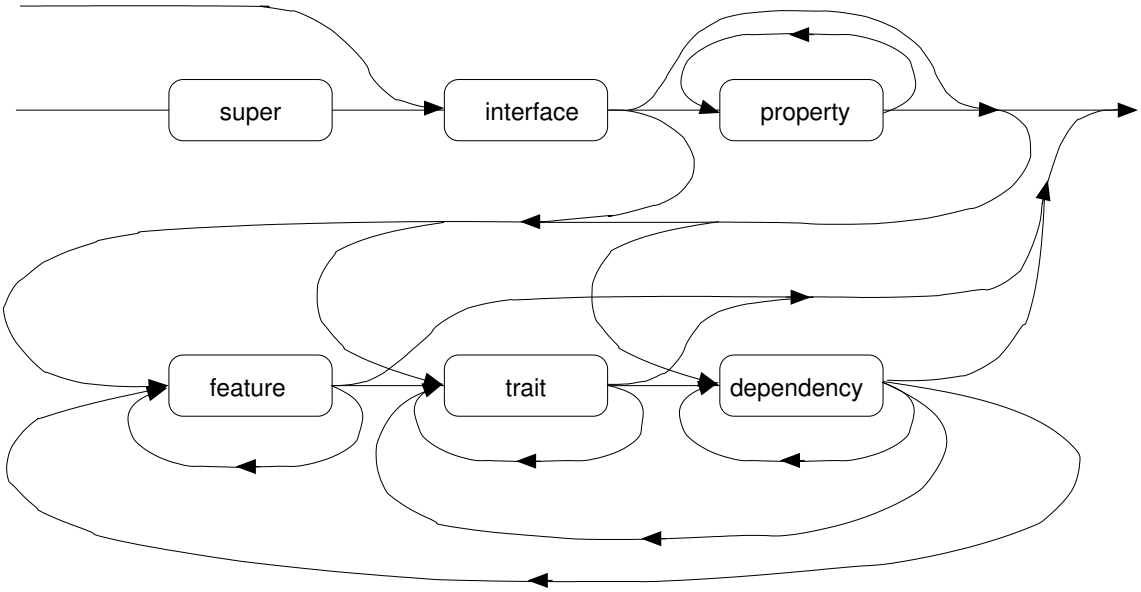

Figura 4.2: Diagrama de ferrovia da descrição de membros de famílias.

4.1.1 Propriedades Configuráveis

Sobre a descrição de *propriedades configuráveis*, deve-se traçar um paralelo entre o papel destes elementos em AOSD e os elementos da linguagem que são utilizados para descrevê-las. Na linguagem, o elemento *trait* refere-se a parâmetros que devem ser quantificados para caracterizar as funcionalidades dos componentes. Em AOSD, um `trait` pode, por exemplo, especificar o *quantum* do escalonador de *threads* do sistema operacional. O elemento *feature*, por sua vez, denota funcionalidades que são realizada pelos componentes e também para indicar se estas estão, por padrão, ativadas ou desativadas quando estes componentes são instanciados. Por exemplo, em nossa modelagem de *threads*, a ativação de uma `feature` denominada *active_scheduler* faz com que o tempo que uma *thread* está em execução influa em sua permanência na `CPU`.

Os diagramas de ferrovia que, representam os elementos da DTD utilizados para descrever *propriedades configuráveis*, são mostrados na Figura 4.3. Como pode ser observado, a descrição de um `trait` contém os dados referentes ao seu nome, tipo (e.g. *int*, *long*, *char*) e um valor padrão de configuração. A descrição de uma `feature`, por sua vez, contém o nome, o valor padrão de ativação ou desativação (i.e.

true ou *false*) e também pode designar `traits` e dependências que condicionam sua ativação. Em termos gerais, isto significada que "ligar" uma funcionalidade "extra" de um componente pode exigir parâmetros que a configurem como também a realização de outras funcionalidades.

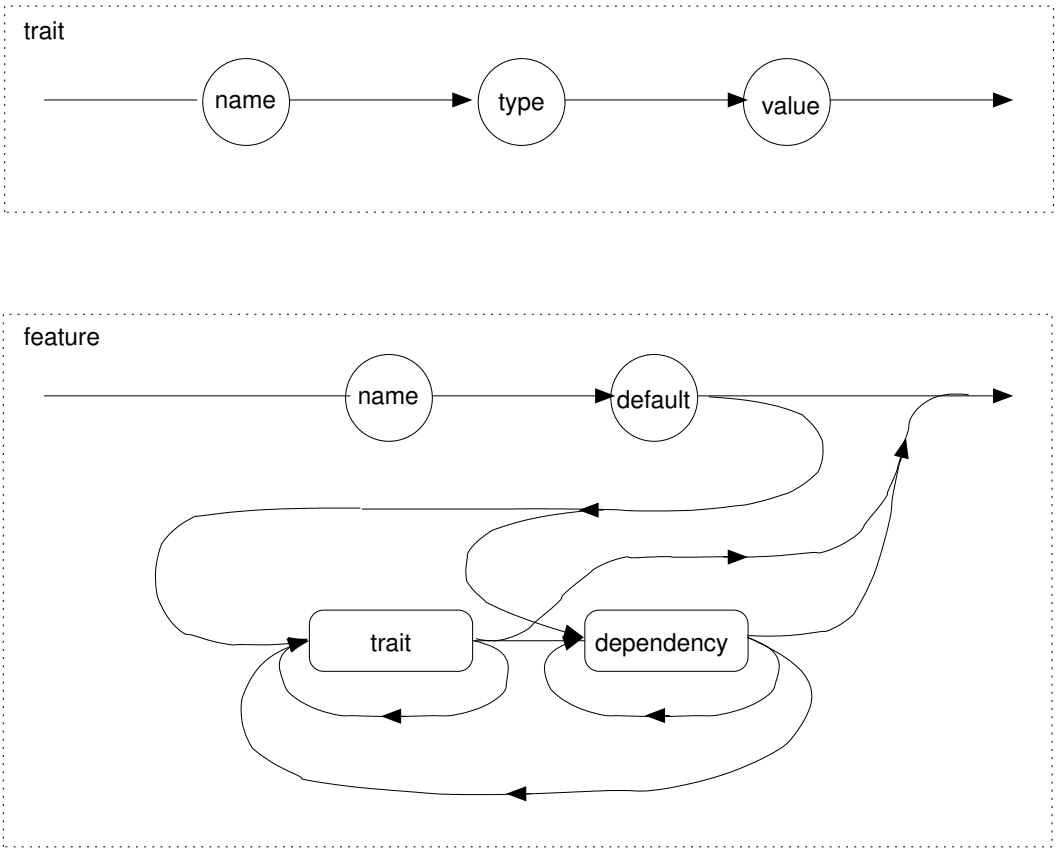

Figura 4.3: Diagramas de ferrovia da descrição de propriedades configuráveis.

Um ponto que deve ser ressaltado sobre *propriedades configuráveis* é que estas exercem papel não só na parametrização do sistema mas também no processo de inferência de componentes. Por exemplo, o uso de um componente pode estar condicionado à ativação de uma *propriedade configurável* pertinente a uma outra família ou membro. Este fato já justifica a integração ao sistema de um componente que implementa esta *propriedade configurável*. Assim, *propriedades configuráveis* também reduzem o espaço de escolha de componentes quando há entre estes certa equivalência de interfaces ou mesmo outras *propriedades configuráveis*. Considerando novamente a ativação da

`feature` *active_scheduler* na família *Thread*, temos que os únicos membros desta família que são passíveis de serem selecionados são *Concurrent_Thread* e *Priority_Thread*, pois ambos têm relação com o tempo de `CPU` (i.e. *quantum*), sendo diferenciados apenas pelo fato de que um esquema de prioridades também influirá nas decisões do escalonador caso o segundo seja selecionado.

4.1.2 Dependências

A especificação de dependências entre os componentes usando a linguagem de descrição em questão é feita através de expressões lógicas textuais nas quais os operandos são uma composição de identificadores de famílias (i.e. nomes), membros, `traits` e `features`. Por exemplo, a expressão lógica que condiciona a ativação da `feature` *active_scheduler* à seleção dos membros *Concurrent_Thread* e *Priority_Thread* é:

```
<feature name="active_scheduler" default="true">
  <trait name="QUANTUM" type="unsigned␣int" value="10000"/>
  <dependency requisit=
     "(::Thread::Concurrent_Thread␣||␣::Thread::Priority_Thread)"/>
</feature>
```

Outra situação que justifica a especificação de dependências entre membros e famílias é que a partir da aplicação, ou de seu código-fonte propriamente dito, pode não ser possível extrair a informação necessária para inferir todos os componentes de um sistema. Referências a métodos das interfaces infladas na aplicação possibilitam inferir os membros que os realizam, mas não como e através de quais outros componentes os realizam [1]. Portanto, é necessário explicitar nas descrições dos componentes quais outros componentes devem ser selecionados—e posteriormente instanciados—para que o sistema realize de fato as funcionalidades que deve realizar. Por exemplo, o uso de *threads* na aplicação leva-nos também a requisitar a família *Address_Space* e, conseqüentemente,

[1] Um mecanismo de análise incremental tem sido estudado para viabilizar a implementação de um máquina de inferência com esta potencialidade.

a seleção de um de seus membros. Isto porque, independentemente de qual membro da família *Thread* foi utilizado, tem-se como um requisito funcional a instanciação de um espaço de endereçamento para a devida alocação do sistema operacional e da aplicação em memória.

4.1.3 Propriedades Não Funcionais

Evidentemente, no que tange a avaliação de requisitos, ainda é possível que membros de uma mesma família possuam interfaces e também *propriedades configuráveis* idênticas, levando-os a serem considerados funcionalmente equivalentes. Neste caso, a linguagem prevê o uso de *modelos de custo* através dos quais é possível deduzir qual componente é mais adequado (ou "barato") dentre seus equivalentes. Isto é feito através do elemento *property*, o qual é o ponto de partida para implementação destes modelos e assim especificar as propriedades não funcionais dos membros de cada família. Em [69], Hoeller propõe um modelo de gerência do consumo de energia de cada componente segundo a metodologia AOSD. No presente trabalho, para que o mecanismo de inferência tivesse poder de decisão quando se deparasse com situações de equivalência funcional dentro de uma mesma família, o modelo de custo utilizado consiste de estimativas a respeito do tamanho ocupado pelos membros quando instanciados. As unidades de medida utilizadas foram *bytes* para componentes de software e LUTs (do inglês *Look-up Tables*) para componentes de hardware.

4.2 Especialização de Plataformas

Conceitualmente, a especialização de plataformas seguindo as premissas da metodologia AOSD consiste em atender um contrato de pré- e pós-condições para obter, como resultado, a *configuração estável* de um sistema que será gerado (ou sintetizado) a partir de um repositório de componentes sobre o qual os requisitos da aplicação são mapeados. Por *configuração* podemos assumir uma lista de todos os componentes de sistema selecionados para uma aplicação e uma outra lista de parâmetros que os con-

figuram para este cenário específico de execução. O fator *estável*, por sua vez, pode ser traduzido no atendimento de todos os requisitos da aplicação. Isto é alcançado justamente com a seleção dos componentes, pela quantificação, ativação ou desativação de *propriedades configuráveis* e também pela satisfação de propriedades não funcionais.

Em termos práticos, o mecanismo de gestão e manipulação de conhecimento que viabiliza a especialização de plataformas seguindo as premissas da metodologia AOSD age em função dos elementos da linguagem que descrevem o que cada componente prove (ou realiza)—interfaces, *propriedades configuráveis* e *propriedades não funcionais*—e também em função dos elementos que descrevem o que cada componente requer. A Figura 4.4 apresenta o fluxograma que define este mecanismo. Primeiramente são extraídos do código-fonte da aplicação referências às interfaces infladas dos componentes do sistema operacional que foram utilizados pela mesma. A partir destas referências e utilizando a base de dados descritiva dos componentes das paltaformas é possível identificar quais famílias e, por conseguinte, quais membros destas famílias de componentes foram utilizados na aplicação. Conhecendo estes membros, o projetista pode fazer uma *seleção inicial* dos componentes que irão compor seu sistema. Estes componentes, por sua vez, podem possuir requisitos, os quais estão expressos em suas descrições através de expressões lógicas conforme detalhado na Seção 4.1.2. Estes requisitos são traduzidos em uma lista de dependências que devem ser resolvidas pelo projetista através da seleção (ou remoção) de outros componentes, quantificação de `traits` e com a ativação ou desativação de `features`. A resolução de dependências, por sua vez, também pode originar novos requisitos que, de forma similar, são apresentados ao projetista que deve também resolvê-los. Este processo encerra-se quando não houverem mais dependências para serem resolvidas. Como já mencionado, ao final, tem-se uma lista dos componentes inferidos direta (i.e. *seleção inicial*) e/ou indiretamente a partir de uma aplicação e uma outra lista de parâmetros que os configuram para este cenário específico de execução; nesta segunda lista estão indicadas as `features` e `traits` dos componentes com seus respectivos valores de ativação (ou desativação) e/ou configuração.

A aplicabilidade deste mecanismo esta condicionada ao fato de que os componentes das plataformas devem ser descritos considerando que os mesmos foram

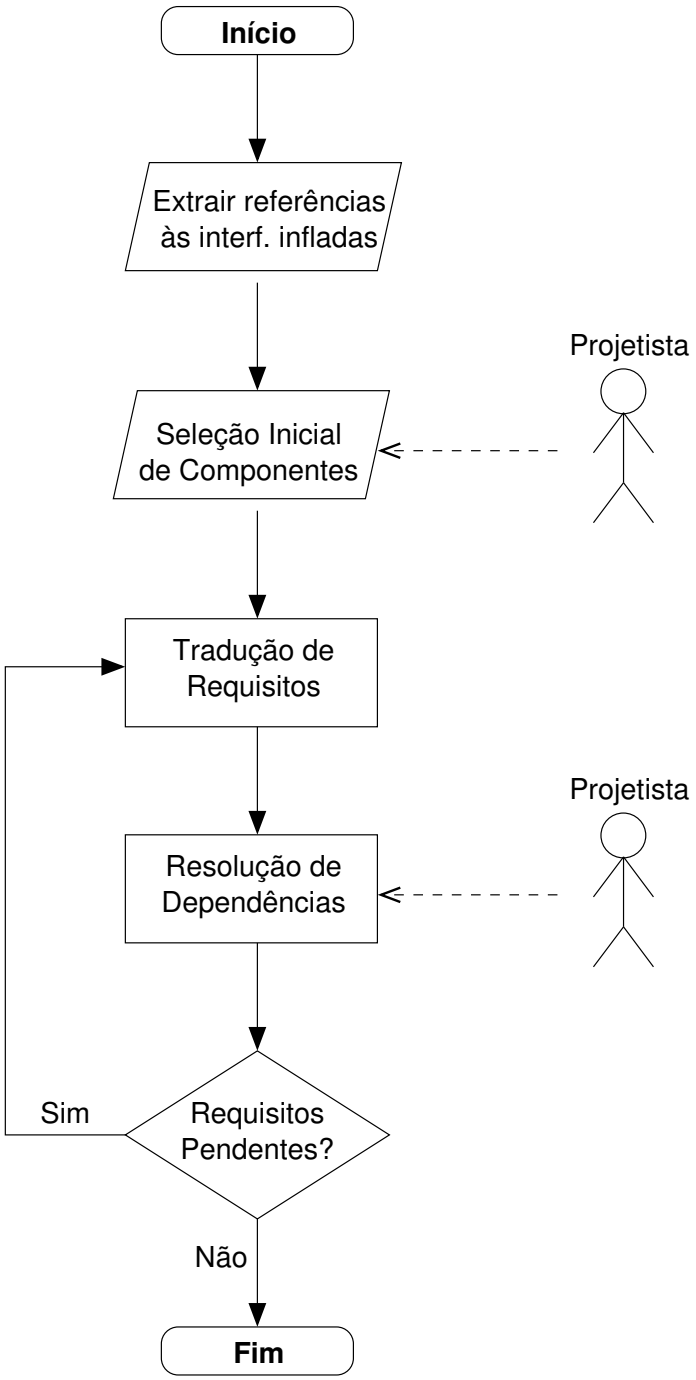

Figura 4.4: Fluxograma do funcionalmento do mecanismo de especialização proposto.

projetados segundo a metodologia AOSD ou que, ao menos, incluam características que permitam descrevê-los segundo as principais premissas desta metodologia. Frente a isto, embora AOSD ainda não cubra o projeto de componentes de hardware, foi imprescindí-

vel identificar uma micro-arquitetura cujos componentes pudessem ser descritos na linguagem apresentada na Seção 4.1 e que fossem flexíveis quanto à configuração e ao reuso em diferentes aplicações. Estas exigências levaram à busca por uma micro-arquitetura implementada a partir de *soft-cores*, haja vista a possibilidade de selecionar, configurar e instanciar estes componentes de maneira muito similar à realizada com componentes de software.

Em um primeiro momento, optou-se pela micro-arquitetura OPENRISC, mas fatores já apresentados no Capítulo 2, e que podem ser resumidos na baixa modularidade e nas dificuldades em se obter um protótipo funcional, levaram ao seu descarte. A segunda opção foi a LEON2, cujo o projeto segue uma linha diferente da seguida pela OPENRISC. Além de ser escrito em VHDL, linguagem que provê recursos de configuração mais poderosos do que os providos por VERILOG, o código-fonte desta micro-arquitetura facilita a identificação das unidades (i.e. arquivos) que implementam os componentes que a integram. Estes componentes, além de serem configuráveis em diferentes aspectos, podem ser facilmente removidos (ou incluídos) em um processo de especialização orientado à aplicação. Foram estes fatores, dentre outros mencionados no Capítulo 2, que levaram a escolha da LEON2.

No que cabe a descrição dos componentes de hardware da LEON2, alguns pontos devem ser observados. Uma vez que o padrão de interconexão seguido pela plataforma esconde os detalhes de interface dos componentes de hardware que a integram, verificou-se que a descrição destes componentes fica restrita à suas *propriedades configuráveis* e dependências. Mesmo que estas interfaces fossem descritas, os "sinais" (ou "pinos") que as definem agregam uma informação cuja semântica está comumente relacionada ao barramento ou NOC ao qual o componente será conectado do que propriamente às funcionalidades que implementa.

Frente a isto, *mediadores de hardware* assumem um papel fundamental na estratégia de geração proposta neste trabalho. Associando mediadores de hardware aos componentes da micro-arquitetura foi possível determinar como esta deveria ser especializada para diferentes aplicações. Isto porque o *contrato de interface* mantido entre as abstrações do sistema operacional e mediadores de hardware pode ser estendido de tal

maneira que os próprios mediadores sejam responsáveis pela realização, em alto-nível, das interfaces dos dispositivos que abstraem. Na prática, isto reflete no fato de que tão logo um mediador de hardware seja selecionado para atender requisitos de abstrações do sistema operacional—e portanto requisitos da aplicação—o componente de hardware abstraído por ele também deve ser selecionado.

A Figura 4.5 mostra um fragmento da descrição do *soft-core* que implementa a CPU da micro-arquitetura LEON2. Esta CPU é um SPARCV8 e, como pode ser observado, possui duas `features` e um `trait`, o qual especifica o número de janelas de registradores a serem implementadas pela `CPU`. Quanto as `features`, a primeira delas, *mul_div_inst*, especifica se a CPU deve incluir, em seu *datapath*, suporte a instruções de multiplicação e divisão complexas (i.e. 32x32 bits). A segunda `feature`, *mul_inst_pipe* está, por sua vez, condicionada à ativação da primeira e especifica a implementação de registradores de *pipeline* como suporte às instruções de multiplicação e divisão.

A maneira encontrada para indicar, na descrição dos componentes, a associação entre mediadores e componentes de hardware foi através de uma `feature` denominada *synthesizable*. Esta `feature` é inserida na descrição dos mediadores de hardware para os quais existe no repositório de componentes o *soft-core* que implementa o componente de hardware abstraído; sua ativação está condicionada à própria inclusão do *soft-core* na configuração do sistema. A título de exemplo, na descrição do mediador de hardware para o SOFT-CORE de CPU, cuja descrição foi ilustrada na Figura 4.5, esta `feature` é indicada da seguinte forma:

```
<feature name="synthesizable" default="true">
  <dependency requisit="::CPU_IP::SPARCV8_CPU_IP"/>
</feature>
```

Excepcionalmente, em micro-arquiteturas “rígidas” ou que possuem *hard cores*, a seleção de um componente de hardware desta classe de IP fica restrita à própria presença do mediador de hardware responsável por abstraí-lo.

A associação de mediadores e componentes de hardware não está res-

```
<family name="CPU_IP" type="hardware">

  <member name="SPARCV8_CPU_IP" cost="10">

    <feature name="muldiv_inst" default="true"/>

    <feature name="mul_inst_pipe" default="false">
      <dependency requisit=
       "HARDWARE::CPU_IP::SPARCV8_CPU_IP::FEATURE::muldiv_inst"/>
    </feature>

    <trait name="REGISTER_WINDOWS" type="int" value="8"/>

    ...
  </member>

</family>
```

Figura 4.5: Descrição da CPU da micro-arquitetura Leon2.

trita apenas à `feature` *synthesizable*. Sua ativação pode desencadear outras dependências que tem por objetivo manter acopladas as propriedades que são funcionalmente importantes tanto para o componente de hardware como para o mediador que o abstrai. A título de exemplo, o número de janelas de registradores implementadas pela `CPU` SPARCV8—indicado na descrição deste componente pelo `trait` *REGISTER_WINDOWS*—consiste de uma informação que é utilizada em operações de salvamento e restauração de contexto. Por este motivo, o mediador de hardware para este componente também possui um `trait` denominado *NWINDOWS* que, por sua vez, carrega o mesmo conteúdo semântico de *REGISTER_WINDOWS*. Logo, uma vez que a `feature` *synthesizable* for ativada neste mediador mediador de hardware, uma dependência que assegure esta igualdade deve ser satisfeita. Esta dependência é expressa da seguinte forma:

```
<feature name="synthesizable" default="true">
  <dependency requisit=
     "::CPU::SPARCV8::NWINDOWS␣="
     "::CPU\_IP::SPARCV8\_CPU\_IP::REGISTER\_WINDOWS"/>
</feature>
```

A seguinte sub-seção procura ilustrar os diferentes cenários e condições em que componentes de hardware são selecionados a partir de seus mediadores de hardware para compor sistemas orientados à aplicação.

4.2.1 Mediadores e Componentes de Hardware

Para melhor contextualizar o uso de mediadores na especialização de plataformas de hardware (ou micro-arquiteturas) considere uma família de mediadores de hardware para `NIC`s (do inglês *Network Interface Card*) cujos membros são funcionalmente equivalentes. A partir de uma aplicação que utiliza abstrações do sistema operacional para estabelecer comunicação através de uma interface de rede, pode-se deduzir que um membro da família `NIC` deve ser selecionado; mas, como são funcionalmente equivalentes, a decisão de qual membro especificamente é tomada pelo projetista [2]. Esta situação caracteriza o que chamamos de *Seleção Combinada de IP*. Os dispositivos de hardware são inferidos a partir de requisitos extraídos da aplicação e também através de decisões pontuais do projetista.

Outro cenário, nomeado *Seleção Discreta de IP*, está relacionado à inferência de componentes de hardware considerando os requisitos extraídos da aplicação—nenhuma decisão explícita do programador precisa ser tomada no que se refere a qual componente de hardware selecionar. Um bom exemplo para este cenário pode ser deduzido a partir do modelo de gerenciamento de memória a ser implementado pelo sistema operacional. Uma vez que o programador da aplicação usa abstrações do sistema operacional que implementam um modelo *paginado*, uma `MMU` (do inglês *Memory Management Unit*) será selecionada para síntese. Inversamente, se um modelo *flat* é adotado, o sistema

[2] Na existência de um modelo de custo, este poderia ser utilizado para auxiliar nesta decisão.

não irá dispor de uma MMU.

Uma terceira situação, conhecida como *Seleção Explícita de IP*, representa a liberdade que o projetista possui para escolher os componentes de hardware que serão instanciados no sistema. De fato, esta estratégia de seleção é sempre tomada quando o programador especifica qual micro-arquitetura será especializada pois, comumente uma CPU, um barramento e um controlador de memória são componentes intrinsecamente pertinentes à micro-arquitetura.

Não obstante, o uso de mediadores de hardware na especialização de plataformas não está relacionado apenas com a seleção de IPs. Como visto anteriormente, *propriedades configuráveis* também estão presentes na modelagem e descrição de mediadores e componentes de hardware, sendo "ajustadas" para que uma micro-arquitetura suporte diferentes aplicações. Por exemplo, uma *propriedade configurável* de um mediador de NIC pode designar a geração de códigos CRC que, em outro momento, poderiam ser gerados pelo próprio hardware. Neste caso, a feature que implementa esta funcionalidade tem como requisito a geração destes códigos pelo componente de hardware abstraído pelo respectivo mediador.

Outros dois exemplos sobre o uso de *propriedades configuráveis* para adequar micro-arquitetura à diferentes cenários de aplicação são o dimensionamento da memória cache da CPU para minimizar o número de acessos à memória principal [70] e também a exploração da escalabilidade da arquitetura SPARCV8. Nesta arquitetura, modificando o número de janelas de registradores implementadas pela CPU é possível reduzir o tráfego de dados entre o processador e a memória de pilha de um processo ou *thread*. Como este tráfego é comum na passagem e retorno de parâmetros em chamadas de funções, o desempenho pode ser melhorado.

Adicionalmente, é importante reafirmar o papel exercido pelas *propriedades configuráveis* na inferência de componentes. Tendo em vista que estes elementos podem representar funcionalidades que satisfazem requisitos específicos de uma aplicação, apenas os componentes que implementam estas *propriedades configuráveis* seriam passíveis de serem selecionados. Para exemplificar esta situação no domínio do hardware considere uma abstração do sistema operacional denominada *Serial_Communicator*

cujo propósito é prover suporte à comunicação serial ponto-a-ponto usando dispositivos `UART`s (do inglês *Universal Asynchronous Receiver/Transmitter*). Visando controle de erros durante a comunicação, uma `feature` desta abstração denominada *data_integrity* tem como requisito a geração de códigos de paridade. Uma vez ativada, somente UARTs que implementam esta funcionalidade seriam selecionáveis.

A Figura 4.6 apresenta um digrama que ilustra a especialização de plataformas de software e hardware considerando a estratégia proposta neste trabalho. As linhas verticais representam as diferentes classes de componentes que podem compor o sistema. As colunas horizontais mostram os três cenários de seleção de IPs abordados (i.e. Explícita, Discreta e Combinada). Como ilustrado as *propriedades configuráveis* são aplicáveis sobre as abstrações do sistema operacional, mediadores e componentes de hardware. A área hachurada representa funcionalidades que, dependendo dos requisitos da aplicação, podem ser implementadas pelos mediadores de hardware ou pelo próprio hardware. Cabe também ressaltar que por ilustrar um processo de geração no qual as dependências entre os componentes são resolvidas iterativamente, o diagrama da Figura 4.6 não objetiva ilustrar a sequência exata em que os componentes são inferidos e/ou selecionados para um dado sistema. Em geral, após escrever a aplicação e escolher a micro-arquitetura de hardware a partir qual o sistema será gerado, o projetista realiza uma seleção inicial de componentes que foram diretamente referenciados no código-fonte da aplicação e a seguir participa de um processo de resolução de dependências.

O capítulo seguinte apresenta como o mecanismo gestão e manipulação de conhecimento apresentado neste capítulo foi implementado e avaliado em estudos de caso experimentais que envolveram a especialização de uma plataforma de software e de uma plataforma de hardware no sentido de gerar SoCs orientados à aplicação.

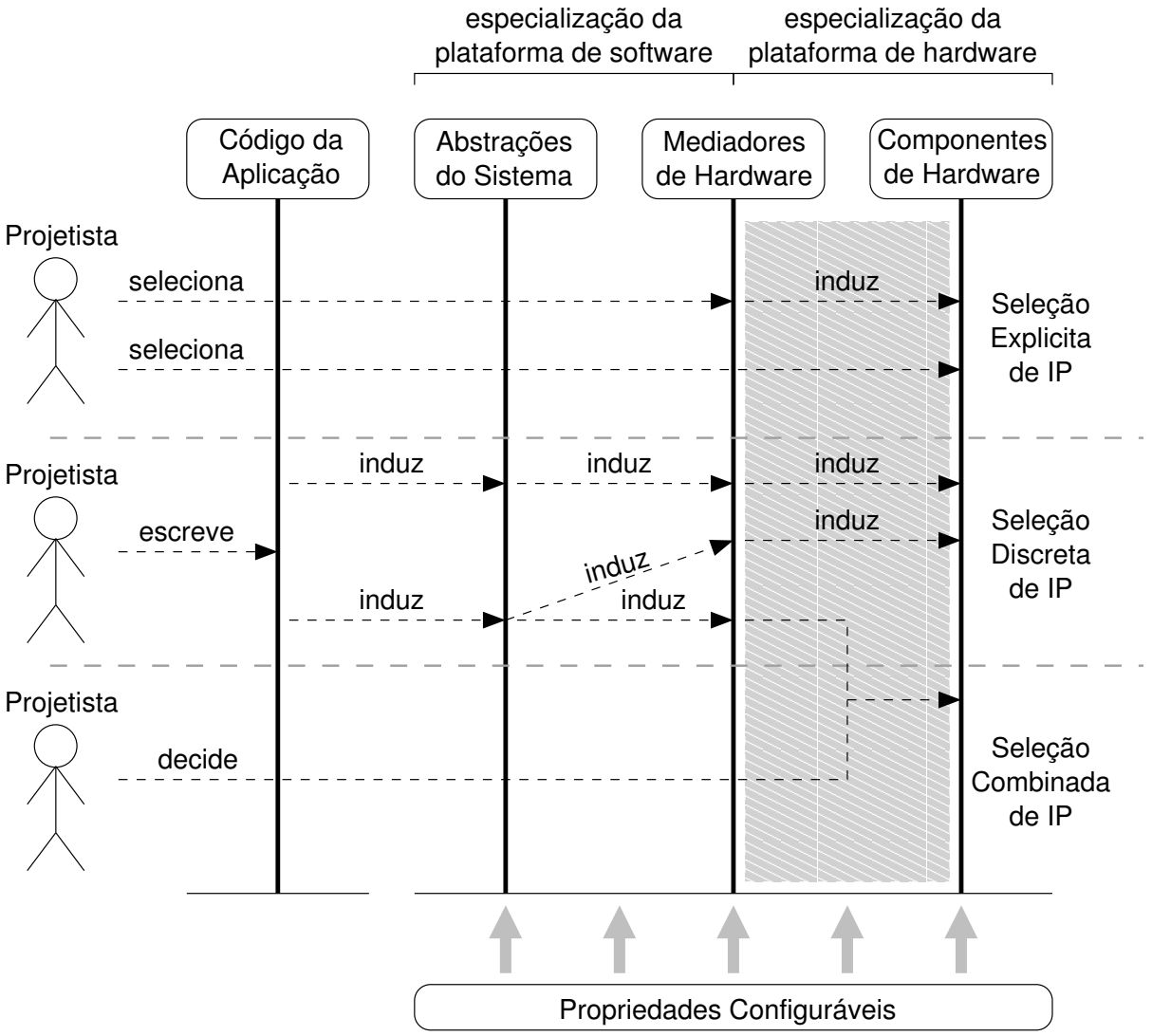

Figura 4.6: Especialização de plataformas segundo AOSD.

Capítulo 5

SoCs Orientados À Aplicação

Este capítulo apresenta resultados que demonstram a aplicação prática da estratégia de geração de sistemas embutidos proposta neste trabalho. As plataformas de software e hardware utilizadas nos experimentos foram, respectivamente, o sistema operacional EPOS e a micro-arquitetura LEON2. A partir destas duas plataformas foi possível gerar SOCs a partir dos requisitos extraídos de aplicações dedicadas. Durante esta fase de experimentação, contou-se com o suporte de uma ferramenta de configuração de componentes desenvolvida no âmbito dos projetos EPOS [16] e PDSCE [17], a qual, por sua vez, foi estendida com uma implementação da estratégia de geração proposta.

5.1 SoCs Leon2 no Sistema EPOS

O sistema EPOS (do inglês *Embedded Parallel Operating System*) [4] foi desenvolvido para prover suporte operacional para aplicações computacionais dedicadas. Tem na metodologia AOSD a base para o desenvolvimento de famílias de componentes que podem ser adaptados para atender aos requisitos de aplicações em diferentes cenários. Com o objetivo de manter a portabilidade de suas abstrações de sistema—a princípio, nenhuma das abstrações do sistema operacional interage diretamente com o hardware—o sistema EPOS faz uso de mediadores de hardware. Uma aplicação escrita sobre este sistema operacional pode ser submetida a uma ferramenta que procura por re-

ferências às interfaces infladas das famílias de componentes que o implementam. Esta ferramenta, conhecida como *Analisador*, gera uma saída na forma de declarações parciais de interfaces de componentes, incluindo métodos, constantes e tipos que foram utilizados explicitamente pela aplicação.

Os dados produzidos pelo *Analisador* servem de entrada para o algoritmo que implementa a etapa de configuração da estratégia proposta neste trabalho e, portanto, a fim de executa-lá na prática, os componentes do sistema operacional EPOS foram descritos na linguagem apresentada no Capítulo 4. De forma similar, os componentes da micro-arquitetura LEON2 (Figura 2.6, Capítulo 2) também foram descritos na mesma linguagem para que, a cerca das mesmas ferramentas de inferência e configuração de componentes, a plataforma de hardware também fosse especializada. Como visto anteriormente, estas descrições constituem uma base de dados XML que descreve as interfaces e funcionalidades realizadas pelos componentes das plataformas como também as relações de dependência entre estes.

Dando seqüência na especialização das plataformas, a ferramenta denominada *Configurador* é a que implementa o algoritmo supracitado e portanto é a responsável por relacionar a informação extraída da aplicação com a base de dados formada pelas descrições dos componentes. O resultado deste processo é um conjunto de requisitos que, traduzido em uma lista de dependências, sugere ao projetista quais componentes devem ser selecionados para que o sistema inclua, a princípio, realizações dos métodos de interface inflada que foram referenciados pela aplicação. Uma vez selecionados, estes componentes podem designar novas dependências, as quais, nas etapas seguintes da configuração, são resolvidas não só pela seleção de outros componentes, mas pela manipulação de suas *propriedades configuráveis* (i.e. `traits` e `features`).

Resolvidas as dependências, temos o que foi denominado uma *configuração estável* do sistema (Seção 4.2, Capítulo 4). Para o EPOS esta *configuração* é traduzida em um conjunto de asserções que serão usadas para associar interfaces infladas às abstrações do sistema e também para ativar os aspectos de cenário [1]. Para a

[1]Estes aspectos são eventualmente identificados como requisitos da aplicação ou emanados do cenário de execução no qual o sistema a ser gerado está inserido.

```
template <> struct Configurable_Features<SPARCV8> {

    //Traits
    static const unsigned int CLOCK = 54340000;
    static const unsigned int NWINDOWS = 8;

    //Features
    static const bool synthesizable = true;
};
```

Figura 5.1: Fragmento do arquivo de configuração de propriedades configuráveis em C++.

LEON2, optou-se por utilizar o mesmo tipo de asserções uma vez que as ferramentas de síntese de hardware utilizadas suportam estas construções e as interpretam como regras de pré-processamento da linguagem na qual esta micro-arquitetura foi implementada (i.e. VHDL).

Além das asserções, o *Configurador* também produz arquivos nos quais estão indicadas, com seus respectivos valores de configuração e/ou ativação, as *propriedades configuráveis* dos componentes. Para o EPOS produz-se um cabeçalho em C++ no qual, para cada membro das famílias de aspectos, abstrações e mediadores de hardware, são declaradas constantes estáticas que definem os valores destas propriedades. A Figura 5.1 mostra um fragmento deste arquivo ilustrando a declaração das *propriedades configuráveis* para o mediador de hardware da CPU SPARCV8. Do lado hardware, o *Configurador* gera dois arquivos de conteúdo semântico similar, mas um deles em VHDL e outro em VERILOG. O motivo desta duplicidade, é que nada impede que estendamos uma micro-arquitetura implementada originalmente em uma destas HDLs com IPs implementados em outra. As Figuras 5.2 e 5.3 apresentam fragmentos destes arquivos para a LEON2. Nestes fragmentos estão ilustradas *propriedades configuráveis* do componente de hardware *SPARCV8_CPU_IP*, o qual implementa a CPU desta micro-arquitetura.

A etapa de geração do sistema é realizada pela ferramenta denominada

```
library IEEE;
use IEEE.std_logic_1164.all;

package configurable_features is

//Traits
constant
HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_REGISTER_WINDOWS : integer := 8;
constant
HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_FAST_JUMPING : boolean := true;
constant
HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_LOAD_DELAY : integer := 1;
constant
HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_FAST_DECODING : boolean := true;
...

//Features
constant
HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_MULDIV_INST : boolean := true ;
constant
HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_MUL_INST_PIPE : boolean := false;
...
end;
```

Figura 5.2: Fragmento do arquivo de configuração de propriedades configuráveis em VHDL.

Gerador que, para o software, usa as asserções produzidas pelo *Configurador* como parâmetros de um *framework* estaticamente meta-programado e desencadeia na seqüencia a compilação do sistema operacional. De forma análoga, para o hardware as asserções são usadas como parâmetros de configuração da *lógica de cola* que será utilizada para integrar os componentes da micro-arquitetura que foram selecionados na etapa de configuração. Cabe também ao *Gerador* desencadear a síntese do hardware. Uma visão de todo o processo descrito nesta seção é apresentada na Figura 5.4. Ao final, tem-se um

```
`ifndef __configurable_features_v
`define __configurable_features_v

//Traits
`define HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_REGISTER_WINDOWS 8
`define HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_FAST_JUMPING TRUE
`define HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_LOAD_DELAY 1
`define HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_FAST_DECODING TRUE
...

//Features
`define HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_MULDIV_INST TRUE
`define HARDWARE_z_CPU_IP_z_SPARCV8_CPU_IP_z_MUL_INST_PIPE FALSE
...
`endif
```

Figura 5.3: Fragmento do arquivo de configuração de propriedades configuráveis em Verilog.

SoC orientado aplicação composto por uma instância do sistema operacional EPOS e um *bitstream* do hardware que é produto da especialização da micro-arquitetura LEON2

5.2 Ambiente de Geração de SoCs

As ferramentas descritas na seção anterior foram integradas em um ambiente único de desenvolvimento. Introduzido originalmente por Tondello [71], este ambiente provê uma interface para a configuração de componentes descritos na linguagem apresentada no Capítulo 4. O uso destes recursos de interface com as ferramentas que implementam a estratégia de geração foi alcançado com a implementação de um mecanismo de composição e validação de sistemas cujo funcionamento é baseado na máquina de estados apresentada na Figura 4.4 do Capítulo 4. Em um primeiro momento, este mecanismo interpreta e materializa as descrições dos componentes em um repositório a partir do qual o projetista pode realizar a seleção dos componentes que devem integrar o

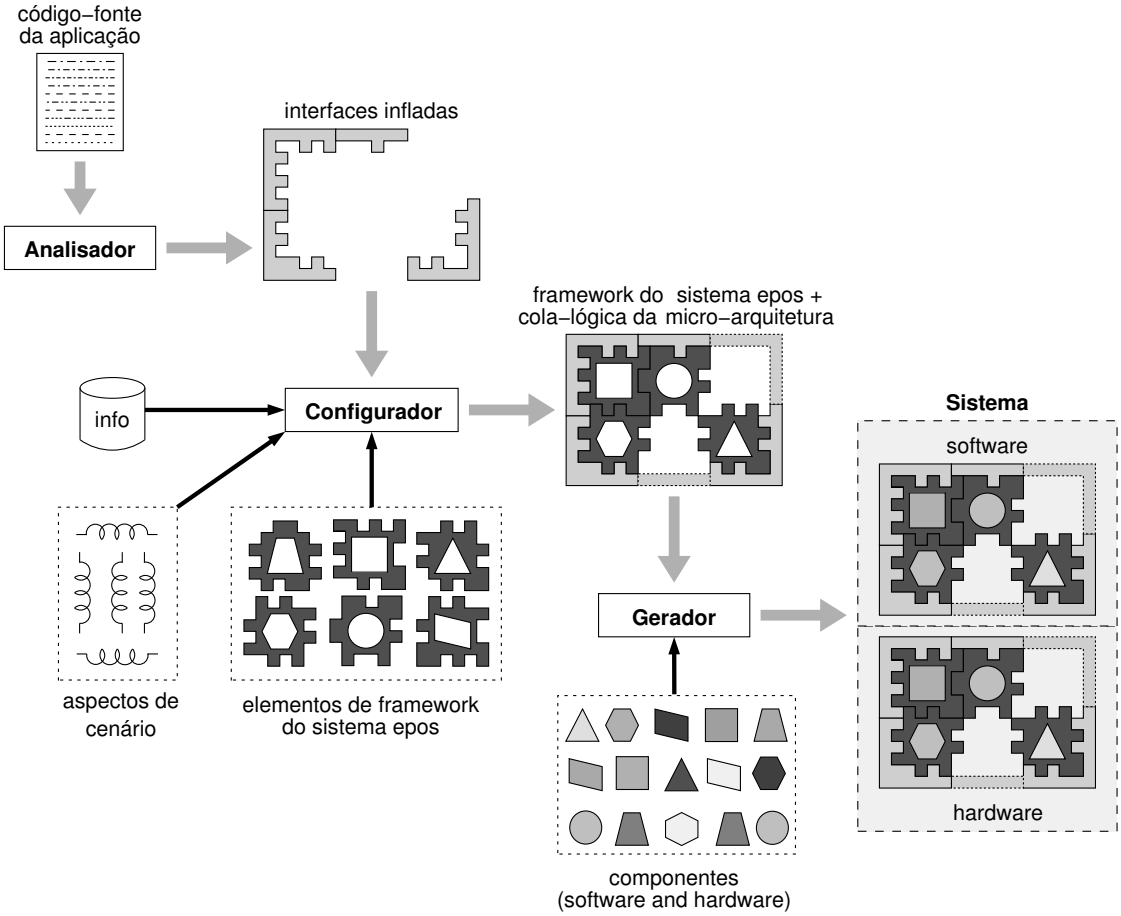

Figura 5.4: Processo de geração do sistema EPOS.

sistema. Cada componente selecionado é então copiado para um repositório de configuração sobre o qual uma rotina de validação verifica ciclicamente quais dependências destes componentes não foram satisfeitas. Estas dependências são apresentadas ao projetista que, por sua vez, as resolve com a seleção (ou remoção) de outros componentes e/ou com a configuração daqueles que se encontram no respectivo repositório. A Figura 5.5 retrata este cenário no qual uma *configuração estável* do sistema é caracterizada pela inexistência de dependências não satisfeitas.

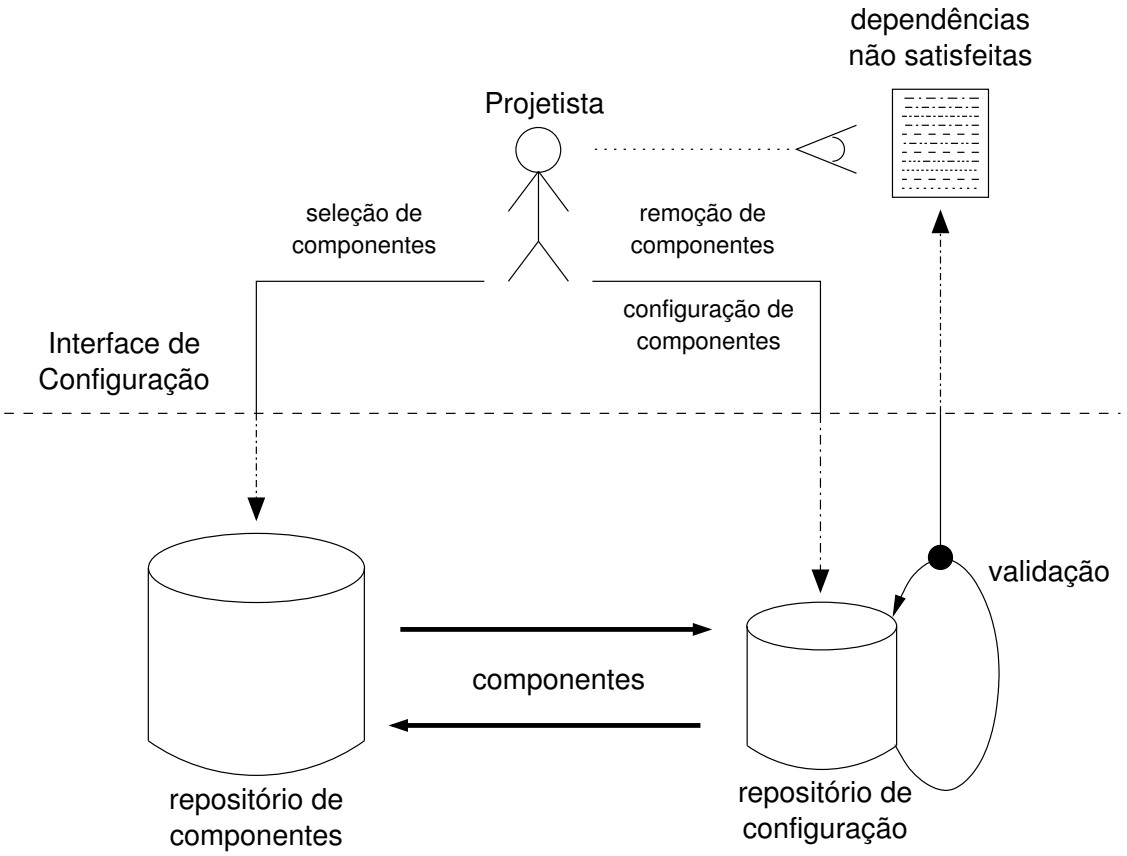

Figura 5.5: Composição e validação de sistemas.

5.3 Estudos de Caso

Visando demonstrar a aplicação prática da estratégia de geração de sistemas embutidos proposta neste trabalho, são apresentados a seguir os resultados de experimentos que consistiram em gerar SOCs a partir de requisitos extraídos de aplicações específicas. Para isto, contou-se com o suporte do ambiente de desenvolvimento descrito na seção anterior e, como mencionado anteriormente, as plataformas de software e hardware utilizadas nos experimentos foram, respectivamente, o sistema operacional EPOS e a micro-arquitetura LEON2. Os SOCs obtidos a partir da especialização destas duas plataformas foram instanciados e avaliados funcionalmente no kit de desenvolvimento *Xilinx Multimedia Board* [72]. Este kit dispõe de uma FPGA Virtex2, 8 MBytes SDRAM e uma série de interfaces físicas para conectar ao "mundo" externo os dispositivos instanciados

na FPGA. O mesmo também possui um decodificador de áudio PCM (do inglês *Pulse Code Modulation*), o qual foi usado em uma aplicação que implementa um tocador de áudio. Além deste tocador, em uma primeira fase de experimentação, o clássico problema de sincronismo conhecido como *Produtor-Consumidor* foi implementado e estendido de diferentes maneiras.

5.3.1 Aplicação Produtor-Consumidor

O código de aplicação que implementa o problema *Produtor-Consumidor* é ilustrado na Figura 5.6. Como pode ser observado, há duas *threads* concorrentes (i.e. *Consumidor* e *Produtor*). O comportamento destas *threads* é definido pelo envio e recebimento de dados utilizando um dispositivo de comunicação serial, o qual, no código apresentado, é abstraído pela abstração do sistema operacional denominada *Serial_Communicator*. Esta abstração provê ao programador uma interface de comunicação de alto-nível para dispositivos seriais. Nesta aplicação, por uma decisão de projeto, o dispositivo foi uma `UART`. Já o controle de concorrência sobre o *buffer* de envio e recebimento de dados, este é realizado através de semáforos implementados pela abstração *Semaphore*. Além disto, tendo em vista que um mesmo *Serial_Communicator* (ou uma mesma `UART`) é utilizado para o envio e recebimento de dados, faz-se necessário aplicar o aspecto *Atomic* sobre esta abstração para garantir atomicidade no acesso a este dispositivo. Na prática, isto significa que chamadas a métodos implementados por membros da família *Communicator* sejam executados de forma atômica. Na aplicação em questão, estes métodos são os métodos *send()* e *receive()* da abstração *Serial_Communicator*. A ativação deste aspecto é feita pelo projetista e não agrega código-fonte visível à aplicação, uma vez que o *framework* do sistema EPOS perfaz, através dos adaptadores de cenário, a integração transparente do código-fonte dos aspectos ao sistema.

```
char buffer[BUF_SIZE];
Semaphore empty(BUF_SIZE), full(0);
Serial_Communicator COM;

int producer() {
    int count_p, out = count_p = 0;
    while (count_p < COUNT) {
        empty.p();
        COM->send(&buffer[out],1);
        out = (out + 1) % BUF_SIZE;
        full.v();
        count_p++;
    }
}

int consumer() {
    int count_c, in = count_c = 0;
    while (count_c < COUNT) {
        full.p();
        COM->receive(&buffer[in],1);
        in = (in + 1) % BUF_SIZE;
        empty.v();
        count_c++;
    }
}

int main() {
    Thread * prod = new Thread(&producer);
    Thread * cons = new Thread(&consumer);
    prod->join();
    cons->join();
}
```

Figura 5.6: Código da aplicação Produtor-Consumidor.

Submetendo o código-fonte da aplicação *Produtor-Consumidor* ao *Analisador* (Seção 5.1) temos como saída as asserções apresentadas na Figura 5.7. Esta saída permite identificar quais abstrações do sistema operacional foram utilizadas pela aplicação, e quais métodos destas abstrações foram referenciados no código-fonte. São elas: a abstração *Semaphore*, da qual foram referenciados o construtor (i.e. *constructor(int)*), o destrutor (i.e. *destructor()*) e os métodos *p()* e *v()*; a abstração *Serial_Communicator*, que teve referenciados o construtor, o destrutor e os métodos *send(unsigned char*, unsigned)* e *receive(unsigned char*, unsigned)*; e a abstração *Thread*, da qual foram referenciados o construtor (i.e. *constructor(int (*)())*) e o método *join()*.

```
Semaphore {
    constructor(unsigned int)
    p()
    v()
    destructor()
}
Serial_Communicator {
    constructor()
    send(unsigned char*, unsigned int)
    receive(unsigned char*, unsigned int)
    destructor()
}
Thread {
    constructor(int (*)())
    join()
}
Heap {
    alloc(unsigned int)
    free(void*)
}
```

Figura 5.7: Saída do Analisador para aplicação Produtor-Consumidor.

Sobre a informação produzida pelo *Analisador* cabe dizer que a identificação dos construtores e destrutores para as abstrações *Semaphore* e *Serial_Communicator* é uma conseqüência da instanciação estática dos objetos *empty*, *full* e *COM*. No caso das *threads*, como os objetos *prod* e *cons* são instanciados dinamicamente através do operador *new*, e não são destruídos explicitamente com o operador *delete*, o destrutor da classe não foi identificado pelo *Analisador* [2]. No que cabe a família de abstrações *Thread*, vale ainda observar que o analisador não determina qual membro da família foi referenciado pela aplicação—*Exclusive_Thread*, *Cooperative_Thread*, *Concurrent_Thread* ou *Priority_Thread*. Como o programador utilizou o próprio identificador da família na declaração dos objetos *prod* e *cons*, a ferramenta apenas faz a ressalva de que a mesma foi utilizada e portanto um de seus membros deverá ser incluído na configuração do sistema.

Os dados extraídos pelo *Analisador* do código-fonte da aplicação *Produtor-Consumidor* serviram como ponto de partida para a especialização das plataformas EPOS e LEON2, e, portanto, para a geração de um SOC orientado a esta aplicação específica. Submetendo-os ao *Configurador*, teve início a etapa de configuração, a qual, como descrito na Seção 5.2, envolve a participação do projetista na composição e validação do sistema. As inferências e sugestões da ferramenta como também as decisões tomadas pelo projetista durante este processo estão ilustradas no diagrama da Figura 5.8. Estes eventos são detalhados a seguir respeitando a ordem em que ocorrem:

1. O uso da micro-arquitetura LEON2 permite ao configurador inferir que, tanto o componente de hardware que implementa uma `CPU` SPARCV8 como também o mediador de hardware que a abstrai, devem ser instanciados no sistema. Temos aqui o que foi denominado *Seleção Explícita de IP*;

2. Nas seqüência a ferramenta sugere ao projetista a seleção dos componentes *Semaphore*, *Serial_Communicator*, *Concurrent_Thread*, *Alarm*, *Flat_Addr_Space* e *Heap*. Sobre estas sugestões, cabe esclarecer o seguinte:

[2]Na implementação do sistema EPOS cada thread executa implicitamente a rotina “exit” ao final de sua execução para que os recursos alocados durante sua execução sejam liberados caso ainda não o foram.

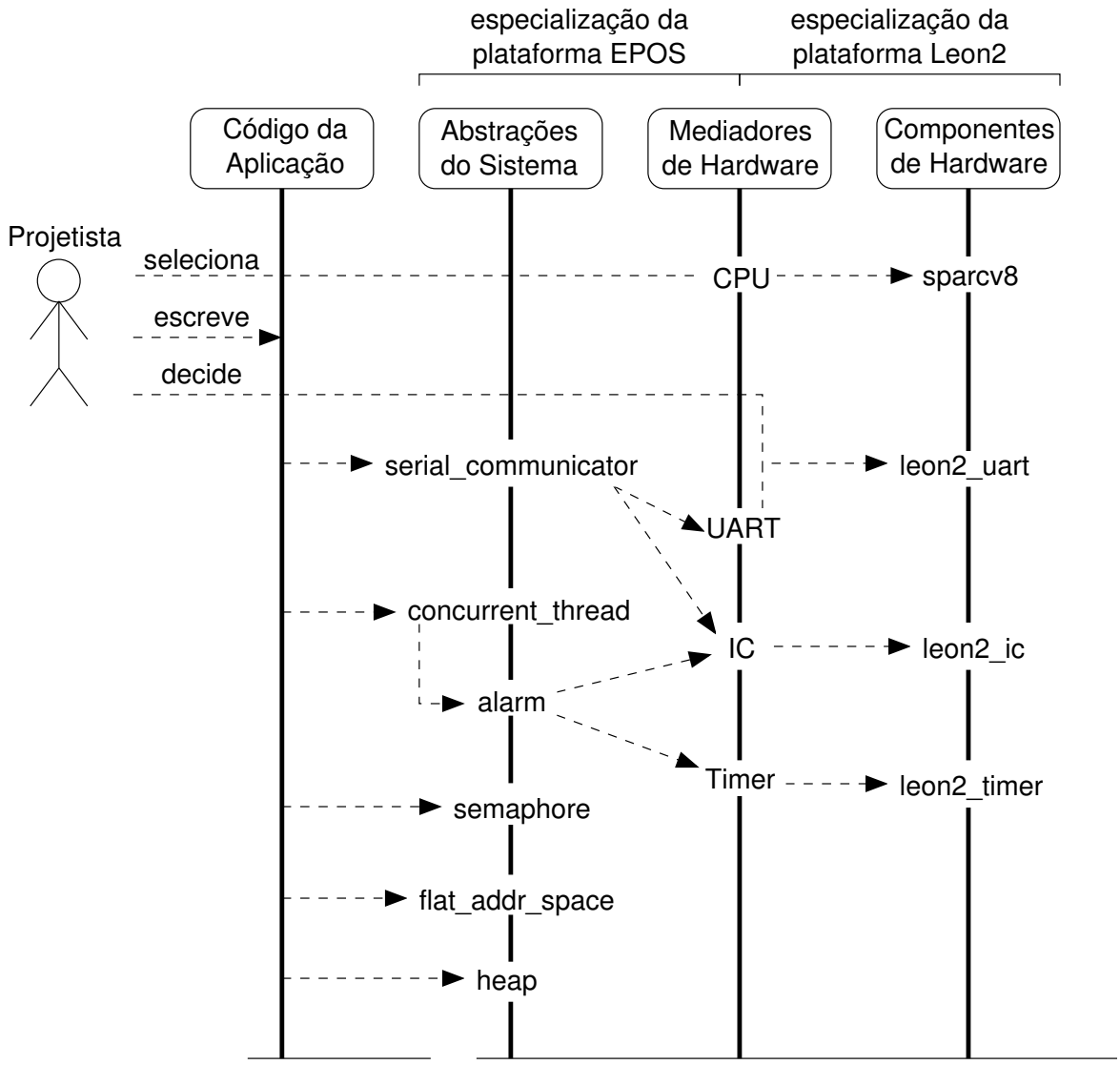

Figura 5.8: Especialização de plataformas para aplicação Produtor-Consumidor.

(a) A sugestão e, por conseguinte, a seleção dos componentes *Semaphore* e *Serial_Communicator* são conseqüências do uso explícito destes componentes no código-fonte da aplicação;

(b) A inclusão de *Concurrent_Thread* é resultado de uma inferência que considerou quais métodos da interface inflada da família *Thread* foram utilizados na aplicação. No caso, a referência ao método *join()*, faz com que o *Configurador* descarte a inclusão dos membros *Exclusive_Thread* e *Cooperative_Thread*

desta família, uma vez que estes membro não implementam este métodos. Já a decisão entre *Concurrent_Thread* e *Priority_Thread* é feita com base em dois fatores. Primeiramente, não há indícios de que um esquema de prioridades foi utilizado pelas *threads* na aplicação e, segundo, o modelo de custo do sistema EPOS, embora simplificado, permite deduzir que o membro *Concurrent_Thread* é mais "barato" que o membro emphPriority_Thread;

(c) O uso da abstração *Concurrent_Thread*, por sua vez, é dependente do membro *Alarm* da família de abstrações *Timepiece*. Este componente implementa uma interface de alto-nível para o gerenciamento do contador de tempo que é utilizado para o escalonamento das *threads* do sistema;

(d) Como não há indícios de um ambiente *multitasking* em que seja necessário o mapeamento lógico-físico de endereços, a abstração do sistema denominada *Flat_Addr_Space* é incluída. Esta abstração implementa uma interface de gerenciamento de memória na qual os endereços apontam diretamente para a memória física do sistema. No que tange o hardware, isto implica na não inclusão do componente de hardware que implementa a `MMU` da micro-arquitetura LEON2 e do mediador de hardware que a abstrai;

(e) O uso do operador *new* para a instanciação dos objetos referentes às *threads* produtor e consumidor faz com que seja incluída na configuração do sistema a abstração *Heap*, a qual implementa um mecanismo de gerenciamento da memória alocada dinamicamente pela aplicação.

3. Selecionadas as abstrações do sistema operacional, cabe ainda ao *Configurador* e ao projetista resolverem as dependências destas abstrações para com o hardware do sistema. Isto é feito com a composição de uma interface sotftware-hardware a partir de membros das famílias de mediadores de hardware que o *Configurador* identifica como requisitos da aplicação. A partir desta interface, é possível definir quais os componentes de hardware da micro-arquitetura devem ser instanciados. Para a aplicação em questão, isto ocorreu da seguinte forma:

(a) A abstração *Serial_Communicator* tem como dependência um dispositivo `UART`. A resolução desta dependência é feita com a inclusão de um membro da família de mediadores *UART* mas, a decisão de qual membro especificamente, cabe ao projetista. Na aplicação em questão, o projetista optou pelo membro *leon2_uart* o que também implicou na seleção do componente de hardware que implementa este dispositivo; caracterizando uma *Seleção Combinada de IP*;

(b) Outra dependência da abstração *Serial_Communicator* é o controlador de interrupções (i.e. `IC`). Isto porque, esta abstração considera o uso de interrupções para sinalizar o recebimento de dados. Como o controlador de interrupções é peculiar à micro-arquitetura LEON2, o mediador e o componente de hardware associados a este dispositivo foram automaticamente selecionados pela ferramenta, caracterizando, portanto, uma *Seleção Discreta de IP*;

(c) A abstração *Alarm*, por sua vez, possui duas dependências. A primeira delas é um contador de tempo (i.e. `Timer`) que será utilizado pelo mecanismo de escalonamento implementado pelas *threads*. A segunda dependência é o controlador de interrupções (i.e. `IC`) que, neste caso, é utilizado para sinalizar o sistema operacional sobre as interrupções geradas pelo contador de tempo. Como pode ser observado no diagrama de especialização (Figura 5.8), a seleção dos mediadores e componentes de hardware associados a estes dois dispositivos foi realizada automaticamente pelo *Configurador* (i.e. *Seleção Discreta de IP*).

O conjunto de asserções produzido pelo *Configurador* para a aplicação *Produtor-Consumidor*, e que foi utilizado pelo *Gerador* na compilação do sistema operacional e síntese do hardware é ilustrado na Figura 5.9. Em suma, este arquivo é uma especificação de quais componentes de software e hardware serão instanciados no sistema. Dois tipos de asserções são distinguidos: *CONF* e *BIND*. O primeiro tipo especifica membros de famílias que devem integrar sistema; o segundo estabelece qual membro é (ou pode ser) declarado na aplicação a partir do próprio identificador da família. Por

exemplo, uma instância do tipo *Concurrent_Thread* pode ser declarada na aplicação na forma *Concurrent_Thread thread(...);* ou simplesmente *Thread thread(...);*.

```
//ABSTRACTIONS
#assert CONF_ADDRESS_SPACE      (Flat_AS)
#assert BIND_ADDRESS_SPACE      (Flat_AS)
#assert CONF_THREAD             (Concurrent_Thread)
#assert BIND_THREAD             (Concurrent_Thread)
#assert CONF_TIMEPIECE          (Alarm)
#assert BIND_TIMEPIECE          (Alarm)
#assert CONF_COMMUNICATOR       (Serial_Communicator)
#assert BIND_COMMUNICATOR       (Serial_Communicator)
#assert CONF_SYNCHRONIZER       (Semaphore)
#assert BIND_SYNCHRONIZER       (Semaphore)
#assert CONF_MEMORY             (Heap)
#assert BIND_MEMORY             (Heap)

//MEDIATORS
#assert CONF_UART               (LEON2_UART)
#assert BIND_UART               (LEON2_UART)
#assert CONF_TIMER              (LEON2_Timer)
#assert BIND_TIMER              (LEON2_Timer)
#assert CONF_IC                 (LEON2_IC)
#assert BIND_IC                 (LEON2_IC)
#assert CONF_CPU                (SPARCV8)
#assert BIND_CPU                (SPARCV8)

//IPs
#assert CONF_CPU_IP             (SPARCV8_CPU_IP)
#assert CONF_TIMER_IP           (LEON2_Timer_IP)
#assert CONF_UART_IP            (LEON2_UART_IP)
#assert CONF_IC_IP              (LEON2_IC_IP)
```

Figura 5.9: Especificação dos componentes a serem instanciados no sistema.

A Figura 5.10 apresenta o diagrama de blocos do hardware resultante da especialização da micro-arquitetura LEON2. Os barramentos `APB` e `AHB`, a ponte de barramento `AH-AP`, o controlador de barramento `Ctrl. Bus` e o controlador de memória `Ctrl. Mem` podem ser vistos como componentes intrínsecos da micro-arquitetura LEON2 e são inferidos a partir do momento em que a mesma é utilizada. Os demais componentes, `SPARCV8`, `IC`, `UART`, `Timer`, são instanciados em função das asserções que os referenciam; as quais são respectivamente: (SPARCV8_CPU_IP), (LEON2_TIMER_IP), (LEON2_UART_IP) e (LEON2_IC_IP).

A fim de agregar maior relevância a este primeiro experimento, os resultados obtidos foram comparados com números extraídos dos sistemas operacionais ECOS e UCLINUX, os quais foram também adaptados para suportar a aplicação Produtor-Consumidor sobre a micro-arquitetura LEON2. A Tabela 5.1 apresenta os tamanhos dos segmentos de código das instâncias destes dois sistemas operacionais juntamente com os do sistema EPOS. Como já enfatizado no Capítulo 2, é comum observar em sistemas operacionais tradicionais um "descuido" para com os requisitos da aplicação—sobretudo quando portados para micro-arquitetura "flexíveis" como a LEON2—o que leva quase sempre a dependências desnecessárias e à agregação de código extra. No UCLINUX, a abstração de um sistema arquivos impacta não só no tamanho do sistema operacional gerado mas também em toda a infraestrutura de inicialização do sistema e de carga das aplicações. O sistema ECOS, por sua vez, também agrega algumas dependências desnecessárias em seu código-fonte. Como já exemplificado, sua HAL tem como certa a existência de uma `UART` em SOCs gerados a partir da micro-arquitetura LEON2. Além de afetar o tamanho e a performance final do sistema, fatores como estes, podem também elevar sensivelmente o custo de desenvolvimento e dificultar eventuais manutenções e/ou atualizações.

Outras estatísticas colhidas dizem respeito à especialização da micro-arquitetura. Através dos dados apresentados na Tabela 5.2 é possível notar a significativa diferença de tamanho entre o hardware gerado para a aplicação Produtor-Consumidor e o hardware gerado a partir de uma "especialização completa" da micro-arquitetura

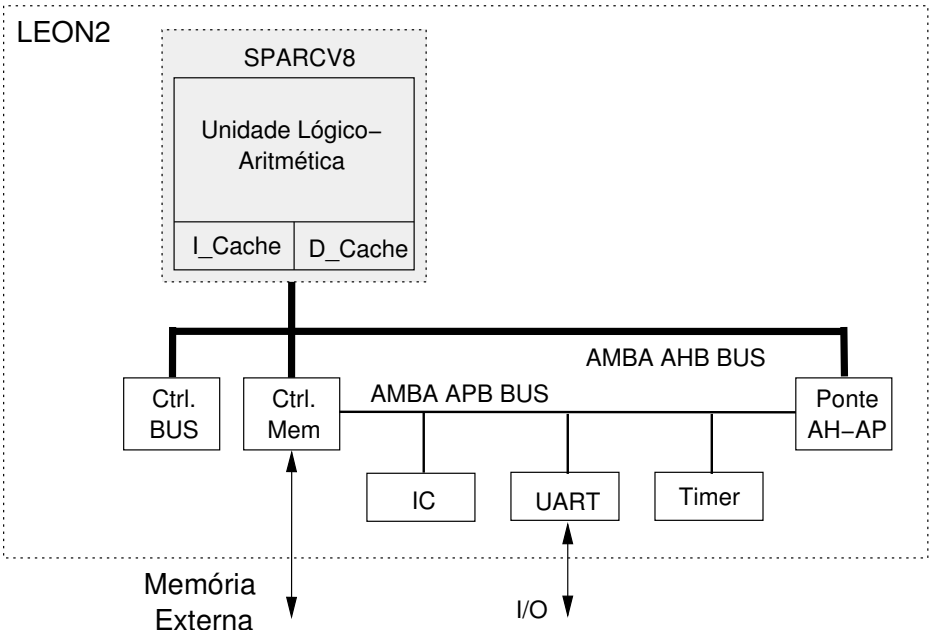

Figura 5.10: Micro-arquitetura Leon2 especializada para aplicação Produtor-Consumidor.

LEON2 [3]. Por "especialização completa" entenda-se a síntese da micro-arquitetura com todos os dispositivos que a mesma integra originalmente.

A aplicação em questão também foi comparada em termos de performance nos sistemas operacionais EPOS e ECOS. As condições de experimentação foram as mesmas: LEON2 a 54 Mhz com um conector de *loopback* de dados conectado fisicamente à porta serial para eliminar a necessidade de outro equipamento na comunicação ponto-a-ponto. O tempo de execução foi mensurado após a realização de 16.000 operações de envio e recebimento pelas *threads* [4]. Os resultados obtidos são mostrados na Tabela 5.3. Estes números não consideram o tempo de transmissão de dados, mas apenas o tempo de processamento da aplicação. Com a `UART` operando na taxa de 38,400 bps, o tempo de transmissão é de aproximadamente 16 segundos.

A diferença de performance entre os sistemas operacionais ECOS e EPOS para a presente aplicação pode ser interpretada como uma conseqüência de pelo menos dois fatores relacionados à implementação destes sistemas. Primeiramente, o uso

[3]Devido algumas restrições arquiteturais o co-processador e a ponte PCI não foram incluídos.

[4]O número de iterações foi quantificado de acordo com a resolução do timer destes sistemas

Sistema Operacional	.text (bytes)	.data (bytes)	.bss (bytes)	Total (bytes)
EPOS	8.988	28	8.400	17.416
eCos	17.152	796	34.040	51.988
uClinux	840.712	44.700	72.649	958.061

Tabela 5.1: Tamanho dos segmentos de código e dados dos sistemas operacionais EPOS, uCLinux e eCos para aplicação Produtor-Consumidor sobre Leon2.

Aplicação	Tamanho (LUTs)	Área ocupada na FPGA Virtex2
Produtor-Consumidor	6.792	31,0 %
Especialização Completa	14.582	67,0 %

Tabela 5.2: Tamanho da micro-arquitetura Leon2 quando especializada para aplicação Produtor-Consumidor e quando sintetizada com todos os dispositivos.

de construções estaticamente meta-programadas no EPOS reduz o número de chamadas de funções no código gerado para este sistema operacional e, conseqüentemente, o tempo para executá-lo. Outro fator é a estratégia adotada para realizar a troca de contexto da `CPU` na arquitetura SPARCV8. No sistema operacional ECOS, a fim de reaproveitar o código das rotinas que tratam o *overflow* e *underflow* das janelas de registradores, a troca de contexto é feita através do lançamento destas *traps*. Respectivamente, o tratamento de *overflow* e *underflow* consiste em copiar para a memória o conteúdo dos registradores das janelas atualmente ativas—contexto a ser salvo—e copiar da memória um conjunto de valores que preenche as janelas de registradores anteriormente ativas—contexto a ser restaurado. No sistema operacional EPOS, o conhecimento de quais janelas estão ativas (i.e. em uso) no contexto a ser salvo e no contexto a ser restaurado é obtido a partir de um cálculo sobre o conteúdo dos registradores `CWP` (do inglês *Current Window Pointer*) e `WMI` (do inglês *Window Invalid Mask*). Isto elimina o *overhead* de tratamento de *traps* durante a troca de contexto que foi observado no ECOS.

A seguir são apresentadas algumas variações da aplicação Produtor-

Sistema Operacional	Duração (ms)
EPOS	45,42
eCos	132,67

Tabela 5.3: Tempo consumido para executar 16.000 iterações de envio e recebimento na aplicação Produtor-Consumidor quando implementada sobre os sistemas operacionais EPOS e eCos.

Consumidor nas quais novos requisitos foram introduzidos com o propósito de caracterizar diferentes especializações das plataformas EPOS e LEON2.

Produtor-Consumidor com Controle de Erros de Transmissão

Neste segundo experimento, foi considerado que o projetista solicitou, na forma de um requisito adicional da aplicação Produtor-Consumidor, a ativação de uma *propriedade configurável* que provê o controle de erros na transmissão dos dados. Em geral, este controle pode ser implementado em software pelos membros da família *Communicator* ou pelos mediadores de hardware que abstraem os dispositivos de comunicação. No caso específico de uma `UART`, a maneira tradicional de implementá-lo é em hardware através da análise de paridade dos bits.

A realização desta *propriedade configurável* pelo sistema a ser gerado está condicionada à ativação de uma `feature` denominada *data_integrity* na família de abstrações *Communicator*. Para o membro *Serial_Communicator* esta `feature` também tem como requisito a ativação de outra `feature`: *parity_check* no mediador de hardware que abstrai o componente *leon2_uart*. Por sua vez, a ativação de *parity_check* leva o projetista a escolher o tipo de paridade a ser implementada: par ou ímpar.

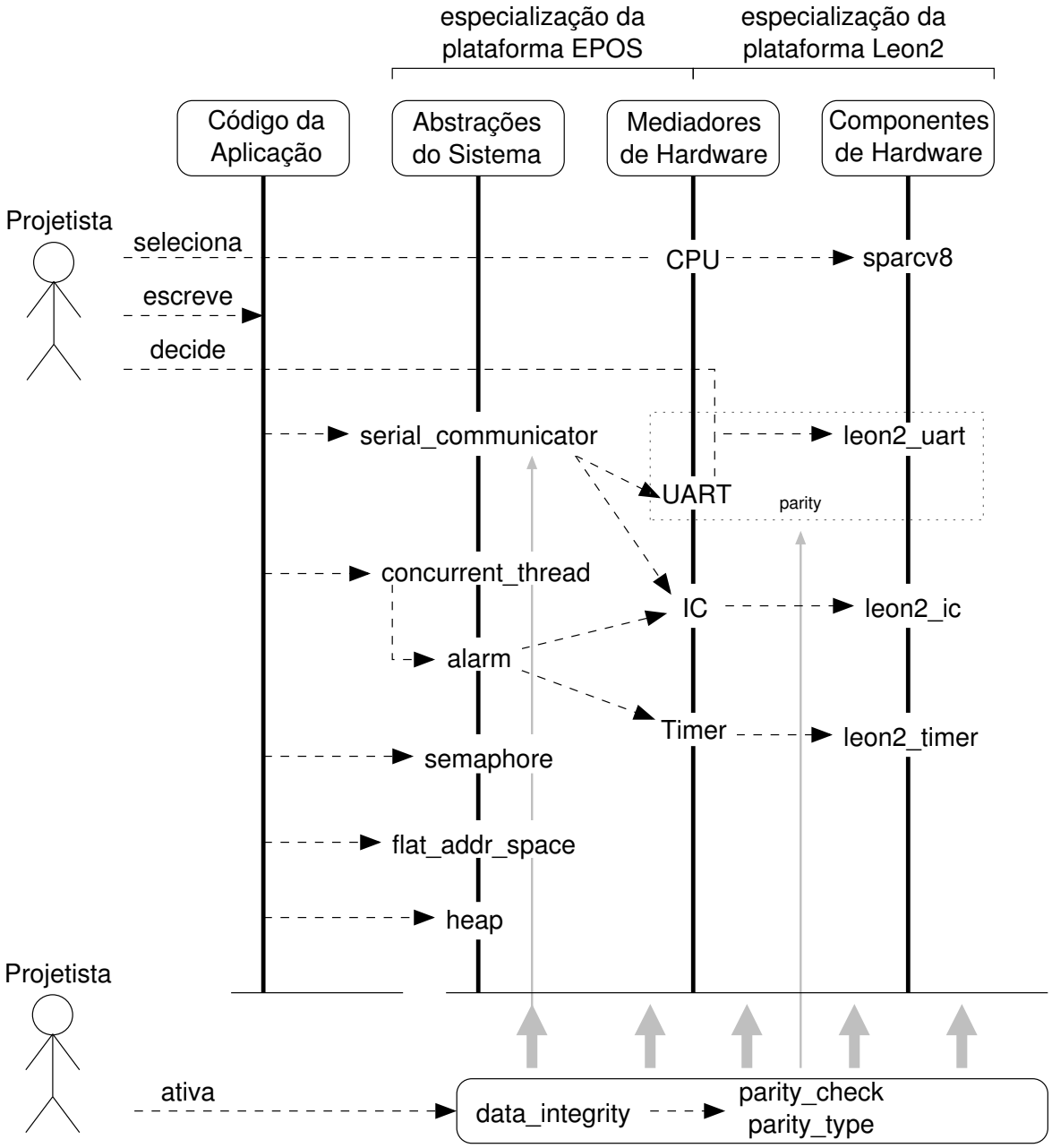

Figura 5.11: Especialização de plataformas para aplicação Produtor-Consumidor com controle de erros de transmissão.

Em termos de tamanho, o SoC gerado para a aplicação Produtor-Consumidor com controle de integridade de dados é exatamente igual ao gerado para a aplicação Produtor-Consumidor originalmente. O número de bytes do sistema operacional e o número de LUTs do hardware gerados não foram alterados pois a análise de paridade de bits é intrínsica ao componente que a implementa (i.e. *leon2_uart*), e o sistema operacional, sempre que inicializado, perfaz o ligamento ou desligamento desta *propriedade configurável*. No entanto, por não serem funcionalmente equivalentes, o diagrama que ilustra a especialização das plataformas é diferenciado. Conforme ilustrado na Figura 5.11, a ativação da `feature` *data_integrity* pelo projetista, desencadeia a ativação da textttfeature *parity_check* e a valoração do `trait` *parity_type*.

Produtor-Consumidor com Depuração

Este experimento teve como objetivo a utilização da unidade de depuração da micro-arquitetura LEON2. Através de algumas instruções inseridas em pontos estratégicos do código do sistema é possível interromper sua execução para que o mesmo seja executado passo-passo e ainda seja possível visualizar e alterar o conteúdo da memória e registradores. Após um estudo de como empregar de forma transparente este dispositivo, ou seja, sem que o projetista fosse o responsável por programá-lo ou ativá-lo explicitamente no código da aplicação, optou-se pela implementação do aspecto *OnChip* que, por sua vez, é membro da família de aspectos *Debugging*. Uma vez aplicado sobre uma abstração ou mediador de hardware do sistema operacional, este aspecto faz com que, logo após a chamada a qualquer rotina implementada por estes componentes, a unidade depuração seja acionada, e a execução passo-a-passo seja realizada. Ao se atingir o fim da rotina, a execução passo-a-passo é encerrada e o sistema retoma o fluxo de execução.

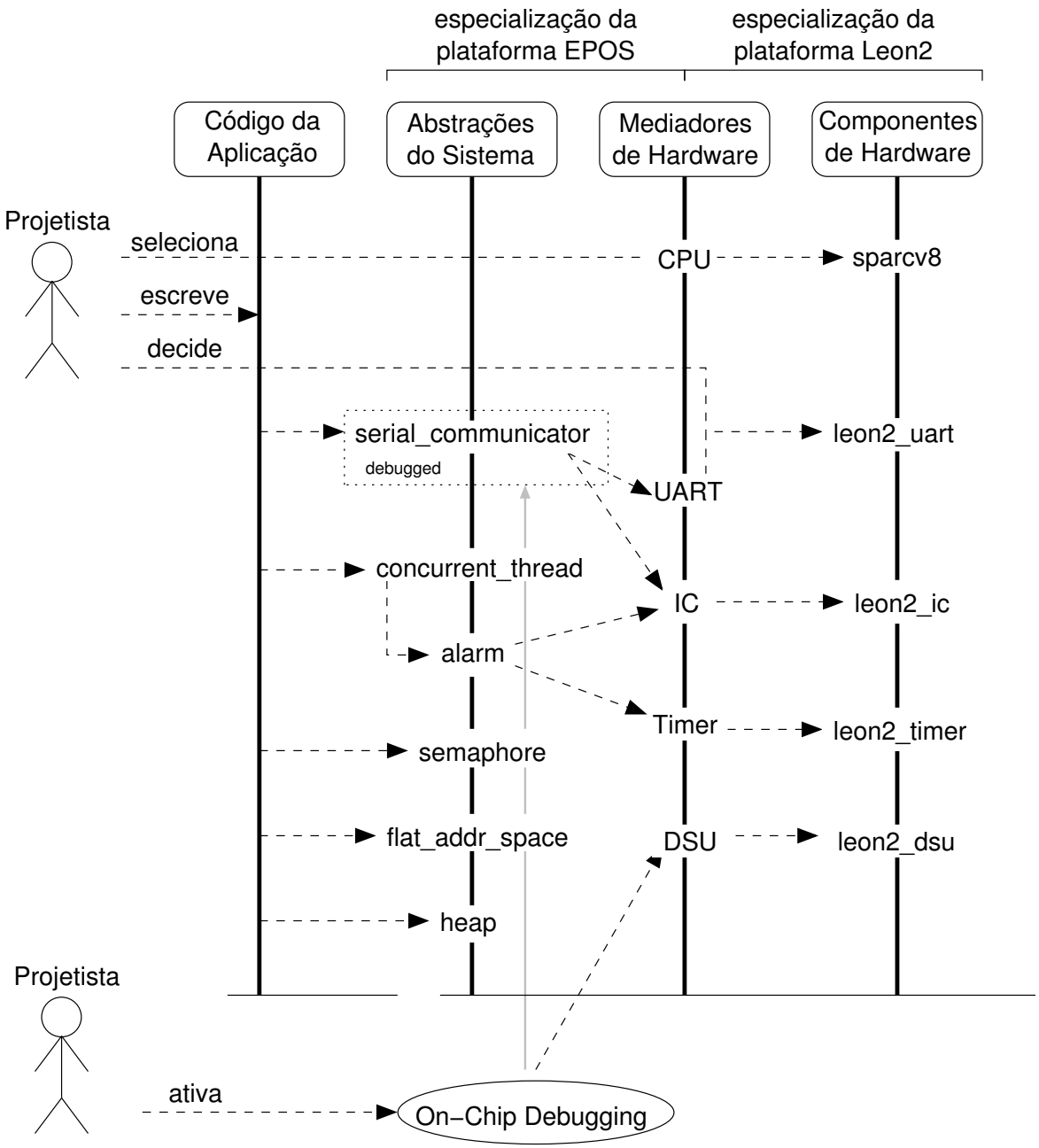

Figura 5.12: Especialização de plataformas para aplicação Produtor-Consumidor com Depuração.

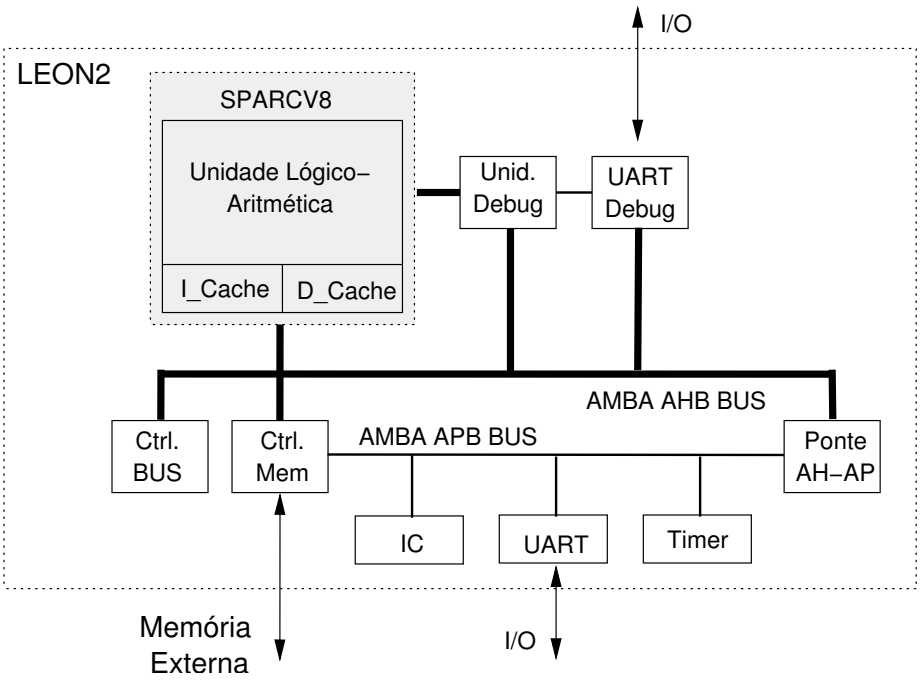

Figura 5.13: Micro-arquitetura Leon2 especializada para aplicação Produtor-Consumidor com Unidade de Depuração.

A inferência do componente de hardware que implementa a unidade depuração da micro-arquitetura LEON2 foi tratada como uma dependência entre o aspecto *OnChip* e a família de mediadores denominada DSU (do inglês *Debug Support Unit*). Esta família possui o membro *leon2_dsu* que, por sua vez, mantém uma dependência para com o componente de hardware que implementa a unidade de depuração da micro-arquitetura em questão.

Nesta variação da aplicação Produtor-Consumidor o adaptador de cenário *OnChip* foi aplicado sobre a abstração *Serial_Communicator* e portanto imediatamente após as rotinas *send()* e *receive()* serem chamadas na aplicação o sistema entrava em modo de depuração. As Figuras 5.12 e 5.13 ilustram respectivamente a especialização das plataformas quando o suporte a depuração é utilizado e o diagrama de blocos do hardware resultante. Neste experimento a instância do sistema operacional EPOS foi de 17.716 bytes e a instância da micro-arquitetura LEON2 foi de 10.428 LUTs.

Produtor-Consumidor sobre Ethernet

Neste experimento, por uma decisão de projeto, ao invés do comunicador serial (i.e. *Serial_Communicator*), a abstração utilizada para prover uma interface de comunicação foi a *Ethernet_Communicator* e, portanto, um `NIC` *Ethernet* foi utilizado para a transmissão dos dados ao invés de uma `UART`. As Figuras 5.12 e 5.13 ilustram respectivamente a especialização das plataformas e o diagrama de blocos do hardware resultante. Neste experimento a instância do sistema operacional EPOS foi de 19.924 bytes e a instância da micro-arquitetura LEON2 foi de 8.302 LUTs.

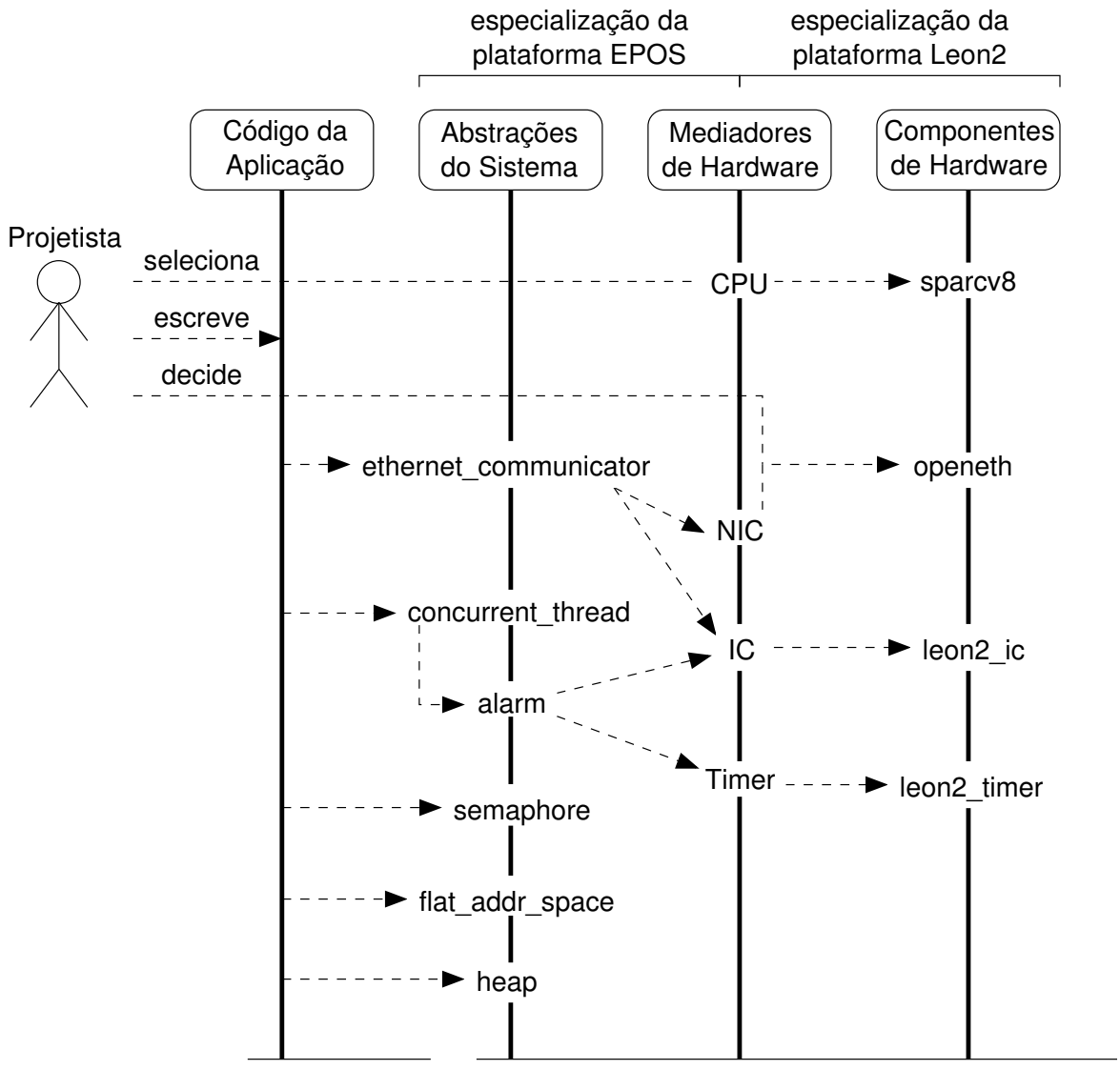

Figura 5.14: Especialização de plataformas para aplicação Produtor-Consumidor sobre Ethernet.

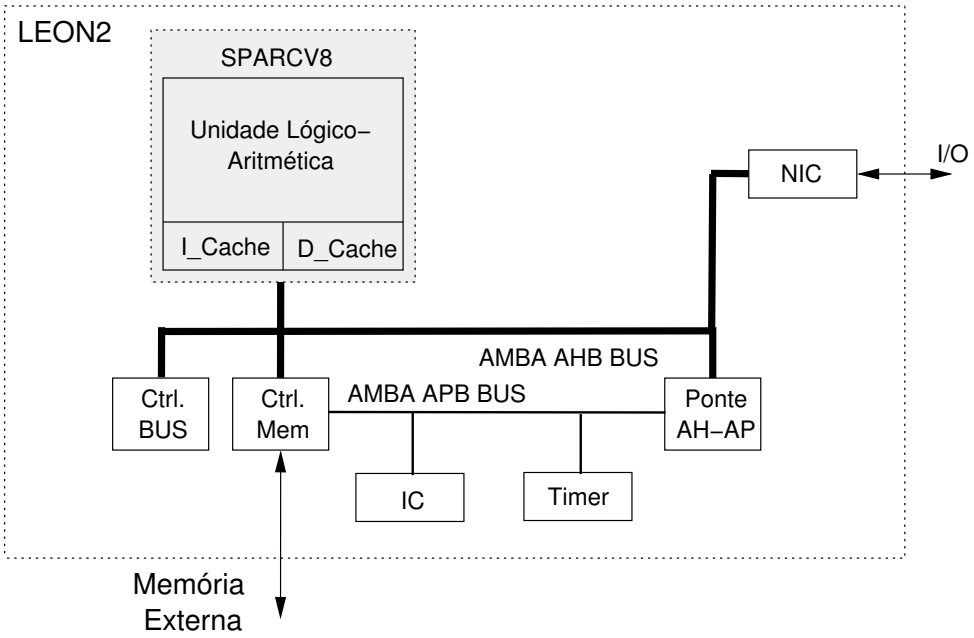

Figura 5.15: Micro-arquitetura Leon2 especializada para aplicação Produtor-Consumidor sobre Ethernet.

5.3.2 Tocador de Áudio PCM

Como mencionado anteriormente, o kit de desenvolvimento *Xilinx Multimedia Board* utilizado para instanciar os SOCs gerados durante os experimentos possui um decodificador de áudio PCM, mais precisamente, um AC97 [73]. A fim de implementar uma aplicação de áudio que utiliza este dispositivo, um *wrapper* que o conecta ao barramento APB da micro-arquitetura LEON2 foi adicionado ao repositório de componentes [74]; quanto a este novo componente cabem as seguintes considerações:

- Na visão da estratégia de geração proposta, o *wrapper* constitui mais um componente de hardware e portanto é descrito como tal através da linguagem de descrição de componentes apresentada no Capítulo 4. Neste sentido, uma família de componentes de hardware denominada *Audio_Decoder_IP* foi criada com o membro *W_AC97*;

- A inclusão de um *wrapper* na micro-arquitetura LEON2 pode ser vista como uma simples extensão desta. Todavia, no contexto da aplicação Tocador de Áudio, a plataforma de hardware também possui um componente extra que é o decodificador AC97 do kit de desenvolvimento. Por este motivo, a micro-arquitetura a ser especializada é na verdade uma conjunção do kit de desenvolvimento com a LEON2 e assim foi denominada XMB_LEON2.

- Na visão do sistema operacional EPOS não há qualquer distinção se o mediador de hardware está abstraindo um componente instanciado na FPGA ou externo a esta, todavia para que o requisito da aplicação fosse mapeado no domínio do hardware, isto é, para que o *wrapper* fosse selecionado para a síntese, o mediador de hardware que abstrai o decodificador AC_97 foi descrito com a seguinte dependência em sua *feature synthesizable*:

```
<feature name="synthesizable" default="true">
  <dependency requisit="::Audio_Decoder_IP::W_AC97"/>
</feature>
```

A aplicação, cujo código-fonte é apresentado na Figura 5.6, consistiu de um tocador PCM que recebe via `UART` o áudio a ser processado pelo decodificador AC97. Para isto, primeiramente é feito o recebimento do conjunto de informações que caracteriza o áudio a ser processado, ou seja, a taxa de amostragem (i.e. *sample rate*), número de bits por amostra (i.e. *bit depth*), modo (i.e. stereo ou mono) e a duração em segundos. Além de serem usados na decodificação dos dados, estas informações são utilizadas no cálculo do tamanho do áudio a ser recebido pela interface serial. De posse deste tamanho, um *buffer* é alocado e, através do método *receive* do "comunicador", o aúdio é recebido e armazenado neste *buffer*. Finalizada a recepção, este áudio é então submetido ao processamento. Para isto, foi utilizada a rotina *play()* da abstração *PCM_Audio_Player* que, por sua vez, prove para o programador uma interface simplificada para o processamento de áudio PCM.

As Figuras 5.17 e 5.18 ilustram respectivamente a especialização das plataformas e o diagrama de blocos do hardware resultante para esta aplicação. A instância do sistema operacional EPOS foi de 18.132 bytes e a instância da micro-arquitetura LEON2 foi de 7.120 LUTs.

```
int main() {

    Serial_Communicator COM;
    PCM_Audio_Player audio_player;

    Audio_Player::Sample_Rate sample_rate;
    Audio_Player::Bit_Depth   bit_depth;
    Audio_Player::Sound_Mode  mode;
    unsigned int              duration;
    unsigned int              audio_length = 0;
    unsigned char *           audio;

    for(;;) {
      COM->receive(&sample_rate,1);
      COM->receive(&bit_depth,1);
      COM->receive(&mode,1);
      COM->receive(&duration,4);

      audio_length = (sample_rate * (bit_depth/8)) * duration;
      if (mode == Audio_Player::STEREO)
        audio_length *= 2;

      audio = new unsigned char[audio_length];
      COM->receive(audio,audio_length);

      audio_player.play(audio,
                        duration,
                        sample_rate,
                        bit_depth,
                        duration);

      delete[] audio;
    }
}
```

Figura 5.16: Código fonte da aplicação Tocador de Áudio PCM.

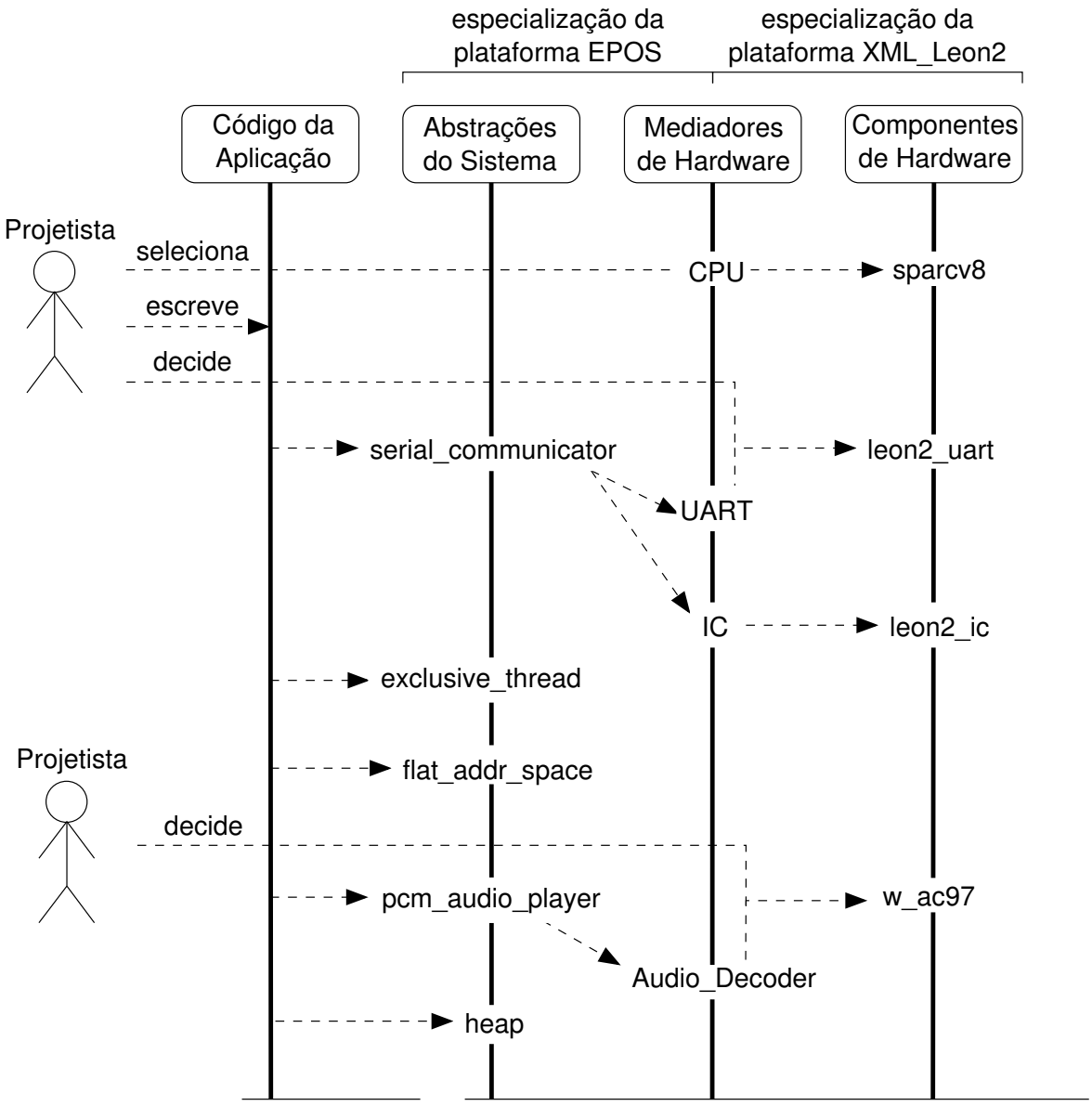

Figura 5.17: Especialização de plataformas para aplicação Tocador de Áudio PCM.

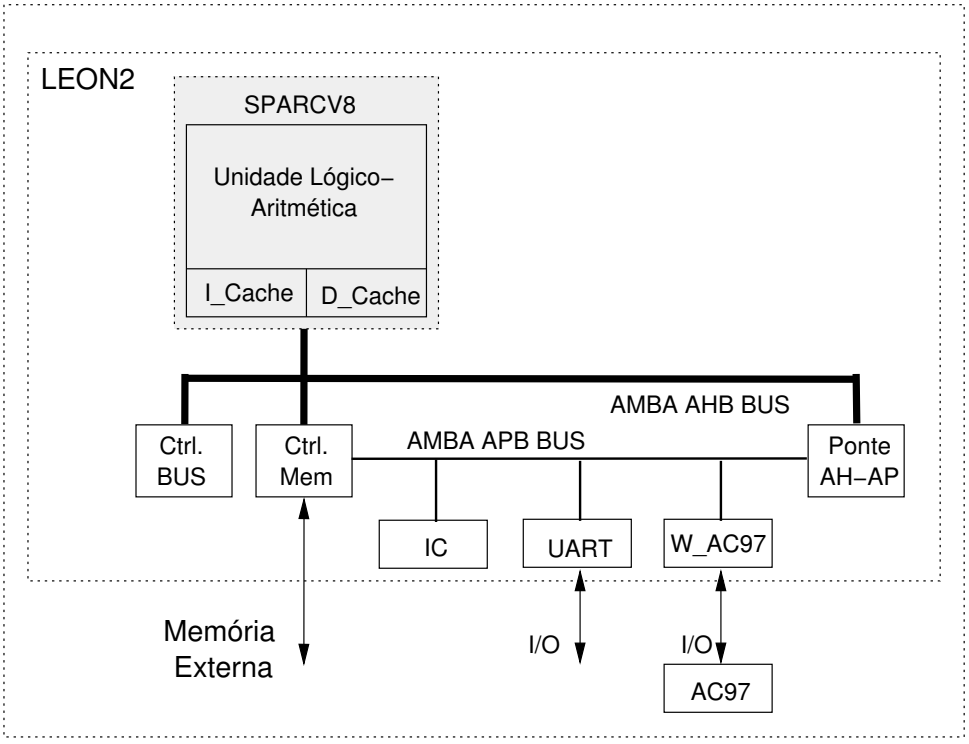

Figura 5.18: Micro-arquitetura XMB_Leon2 especializada para aplicação Tocador de Áudio PCM.

Capítulo 6

Conclusão e Trabalhos Futuros

O uso de plataformas de hardware e software previamente projetadas e validadas constitui uma importante alternativa para o desenvolvimento de sistemas computacionais embutidos, uma vez que estas abstraem muito da complexidade envolvida na integração de seus componentes. Neste trabalho, a descrição de plataformas de hardware segundo as principais premissas da metodologia AOSD serviu de base para a elaboração de uma estratégia de geração de sistemas computacionais embutidos. Esta estratégia foi validada com a implementação de um mecanismo de gestão e manipulação de conhecimento que guia o projetista na confecção de sistemas orientados à aplicação a partir de informações extraídas do código-fonte de suas aplicações. Neste contexto, além de serem promovidos como artefato de portabilidade, mediadores de hardware foram utilizados para mapear requisitos da aplicação em componentes de hardware implementados na forma de *soft-cores*. Esta extensão da aplicabilidade de mediadores de hardware os tornam um poderoso artefato de *co-design* e foi constitui o alicerce para a geração de SOCs orientados à aplicação.

Os experimentos práticos realizados demonstraram a aplicação prática da estratégia na geração de SOCs usando o sistema operacional EPOS e a microarquitetura LEON2. Nestes experimentos fica caracterizado como as principais premissas da metodologia são empregadas na especialização destas plataformas. Tradicionalmente, o projeto de um sistema embutido utilizando sistemas operacionais como ECOS

e uCLINUX esbarra em suposições (ou imposições) que estes sistemas fazem a respeito de dispositivos e de serviços que serão utilizados pelo projetista, levando quase sempre a inclusão de código ou mesmo hardware extra ao sistema. A estratégia proposta sugere que o sistema operacional e a plataforma de hardware sejam especializados de acordo com a aplicação para a qual o sistema embutido foi projetado. As comparações realizadas demonstraram uma grande redução no tamanho dos segmentos de código e do hardware pertinentes aos SOCs gerados segundo esta abordagem.

Uma das limitações que ainda recaem sobre o mecanismo apresentado é que o mesmo não é capaz de avaliar qual micro-arquitetura de hardware deve ser adotada para uma dada aplicação. Embora extendê-lo dependa da existência de um conjunto de micro-arquiteturas para o qual o sistema operacional EPOS tenha sido portado, uma primeira implementação desta funcionalidade etaria embasada em um algoritmo que realizasse a identificação de quais micro-arquiteturas possuem todos os dispositivos que foram classificados como requisitos da aplicação. As micro-arquiteturas assim identificadas definiriam, portanto, o espaço de exploração arquitetural do projetista. Um outro ponto, ainda que não seja uma limitação, diz respeito ao projeto das micro-arquiteturas. Como diagnosticado durante o trabalho, a aplicabilidade deste mecanismo está condicionada ao fato de que os componentes das plataformas devem ser descritos considerando que os mesmos foram projetados segundo a metodologia AOSD ou que, ao menos, incluam características que permitam descrevê-los segundo desta forma.

Como já mencionado, no escopo da metodologia AOSD, o presente trabalho promoveu a aplicação das principais premissas desta metodologia no domínio do hardware. Para isto não foram necessárias quaisquer extensões da metodologia, o que possibilita concluir que componentes de hardware têm, de fato, sido desenvolvidos de maneira muito similar à que componentes de software podem ser desenvolvidos. Logo, embora hajam algumas limitações linguísticas e de paradigmas em relação a fatores como propagação de sinais e sincronismo, com base nos resultados apurados é factível conjecturar sobre a possibilidade de estender a metodologia AOSD para que a mesma cubra também o projeto de componentes e plataformas de hardware.

A possibilidade de estender a estratégia proposta neste trabalho tem

sido investigada em diferentes flancos. Um deles é extender a linguagem de descrição de componentes com modelos de custos que permitam avaliar requisitos relacionados ao consumo de energia e performance do sistema. Informações que qualifiquem as plataformas nestes domínios constituem um importante elemento a cerca da decisão de qual micro-arquitetura utilizar para uma dada aplicação. A escolha não seria tomada apenas com base nos componentes funcionais que a plataforma deve possuir mas também com base no impacto destes componentes sobre o sistema.

Outra proposta de trabalho futuro é a confecção de mediadores hardware cujas funcionalidades possam ser instanciadas tanto em software como em hardware. Dependendo dos requisitos de uma aplicação em tempo de execução, ou seja, de sua demanda por uma funcionalidade que é contemplada por uma dada *propriedade configurável*, o próprio sistema pode designar qual implementação (hardware ou software) deve ser utilizada ou ainda, objetivando tirar maior proveito do paralelismo existente no hardware, instanciar em uma FPGA várias das unidadades funcionais que contemplam esta *propriedade configurável*.

Por fim, cabe ressaltar a importância do presente trabalho para o projeto PDSCE, onde, juntamente com a metodologia AOSD e o sistema operacional EPOS, deu sustenbilidade à idéia de elaborar uma plataforma de desenvolvimento de sistemas computacionais embutidos que possibilite a projetistas e/ou programadores concentrarem-se na implementação de suas aplicações a partir de uma biblioteca de componentes de software e hardware que podem ser sistematicamente integrados em um sistema orientado à aplicação.

Referências Bibliográficas

[1] Reinaldo A. Bergamaschi and John Cohn. The A to Z of SoCs. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 790–798, New York, NY, USA, 2002. ACM Press.

[2] Alberto Sangiovanni-Vincentelli. Defining Platform-based Design. *EEDesign of EETimes*, February 2002.

[3] Alberto Sangiovanni-Vincentelli, Luca Carloni, Fernando De Bernardinis, and Marco Sgroi. Benefits and challenges for platform-based design. In *Proceedings of the 41st annual conference on Design automation*, pages 409–414, San Diego, CA, USA, 2004. ACM Press.

[4] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Aug. 2001.

[5] Carsten Böke. Combining two customization approaches: Extending the customization tool terecs for software synthesis of real-time execution platforms. In *Proceedings of the Workshop on Architectures of Embedded Systems (AES 2000)*, Karlsruhe, Germany, 2000.

[6] Lovic Gauthier, Sungjoo Yoo, and Ahmed A. Jerraya. Automatic generation and targeting of application specific operating systems and embedded systems software. In *Design Automation and Testing in Europe*, Munich, Germany, March 13-16 2001. IEEE CS Press.

[7] S. Habinc. Using VHDL Cores in System-On-A-Chip Developments. In *ESA SP-439: European Space Components Conference : ESCCON 2000*, pages 159–+, June 2000.

[8] Miron Abramovici, Charles Stroud, and Marty Emmert. Using embedded FPGAs for SoC yield improvement. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 713–724, New York, NY, USA, 2002. ACM Press.

[9] Nobuyuki Ohba and Kohji Takano. An SoC design methodology using FPGAs and embedded microprocessors. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 747–752, New York, NY, USA, 2004. ACM Press.

[10] Rajesh K. Gupta and Yervant Zorian. Introducing Core-Based System Design. *IEEE Des. Test*, 14(4):15–25, 1997.

[11] Daniel D. Gajski. IP-Based Design Methodology. In *DAC '99: Proceedings of the 36th Annual Conference on Design Automation (DAC'99)*, pages 43–43, Washington, DC, USA, 1999. IEEE Computer Society.

[12] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Des. Test*, 18(6):23–33, 2001.

[13] Kurt Keutzer, Sharad Malik, A. Richard Newton, Jan M. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonolization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[14] Fauze Valério Polpeta and Antônio Augusto Fröhlich. Hardware Mediators: a Portability Artifact for Component-Based Systems. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, volume 3207 of *LNCS*, pages 271–280, Aizu,Japan, Aug. 2004. Springer.

[15] Fauze Valério Polpeta. Portabilidade em sistemas operacionais baseados em componentes de software. Trabalho individual de acompanhamento de dissertação de mestrado, 06 2004.

[16] EPOS Project. Embedded Parallel Operating System, August 2005. URL:<http://epos.lisha.ufsc.br>.

[17] PDSCE Project. Plataforma de Desenvolvimento de Sistemas Computacionais Embutidos, August 2005. URL:<http://www.lisha.ufsc.br/news/pdsce.html>.

[18] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. High Performance Application-oriented Operating Systems – The EPOS Approach. In *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, pages 3–9, Natal, Brazil, Sep. 1999.

[19] Grant Martin and Jean-Yves Brunel. Platform-based Co-Design and Co-Development: Experience, Methodology and Trends. In *EDP '02: Proceedings of the Ninth IEEE/DATC Electronic Design Processes Workshop*, 2002.

[20] X. Liu and C. Papaefthymiou. A static power estimation methodolodgy for IP-based design. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 280–289, Piscataway, NJ, USA, 2001. IEEE Press.

[21] Reinaldo A. Bergamaschi, Subhrajit Bhattacharya, Ronoldo Wagner, Colleen Fellenz, Michael Muhlada, William R. Lee, Foster White, and Jean-Marc Daveau. Automating the Design of SOCs Using Cores. *IEEE Des. Test*, 18(5):32–45, 2001.

[22] Flávio R. Wagner, Wander O. Cesário, Luigi Carro, and Ahmed A. Jerraya. Strategies for the integration of hardware and software IP components in embedded systems-on-chip. *Integr. VLSI J.*, 37(4):223–252, 2004.

[23] Hal Jespersen. POSIX Retrospective. *StandardView*, 3(1):2–10, 1995.

[24] Advanced RISC Machines Limited - ARM. *The de facto Standard for On-Chip Bus*, online document edition, 2003. http://www.arm.com/ products/ solutions/ AMBAHomePage.html.

[25] IBM Microeletronics. *CoreConnect Bus Architecture*, online document edition, 1999. http://www.ibm.com/ chips/ products/ coreconnect/.

[26] Silicore Corporation. *The WISHBONE Service Center*, online document edition, Sep. 2003. http://www.silicore.net/ wishbone.htm/.

[27] Giovanni De Micheli and Luca Benini. Networks on Chip: A New Paradigm for Systems on Chip Design. In *DATE '02: Proceedings of the conference on Design, Automation and Test in Europe*, pages 418–419, 2002.

[28] Umit Y. Ogras, Jingcao Hu, and Radu Marculescu. Key research problems in NoC design: a holistic perspective. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 69–74, New York, NY, USA, 2005. ACM Press.

[29] Virtual Socket Interface Alliance. *Virtual Component Interface*, 2005. http://www.vsia.org.

[30] OCP International Partnership. *Open core protocol*, 2005. http://www.ocpip.org.

[31] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Abstraction and Reuse in Object-Oriented Designs. In O. M. Nierstrasz, editor, *ECOOP'93: Object-Oriented Programming - Proceedings of the 7th European Conference*, pages 406–431, Berlin, Heidelberg, 1993. Springer.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[33] Robertas Damaševičius, Giedrius Majauskas, and Vytautas Stuikys. Application of Design Patterns for Hardware Design. In *DAC '03: Proceedings of the 40th*

conference on Design automation, pages 48–53, New York, NY, USA, 2003. ACM Press.

[34] Moe Shahdad. An overview of VHDL language and technology. In *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 320–326, Piscataway, NJ, USA, 1986. IEEE Press.

[35] G. Schumacher and W. Nebel. Object-oriented hardware modelling—where to apply and what are the objects? In *EURO-DAC '96/EURO-VHDL '96: Proceedings of the conference on European design automation*, pages 428–433, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[36] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, Marc Edwards, and Yaron Kashai. A framework for object oriented hardware specification, verification, and synthesis. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 413–418, New York, NY, USA, 2001. ACM Press.

[37] SystemC Community. *SystemC*, 2005. http://www.systemc.org.

[38] Grant Martin. Systemc and the future of design languages: Opportunities for users and research. In *SBCCI '03: Proceedings of the 16th symposium on Integrated circuits and systems design*, page 61, Washington, DC, USA, 2003. IEEE Computer Society.

[39] Philips Semiconductors. *Nexperia: Highly integrated, programmable system-on-chip (SoC)*, 2005. http://www.semiconductors.philips.com/products/nexperia/.

[40] Selliah Rathnam and Gert Slavenburg. An Architectural Overview of the Programmable Multimedia Processor, TM-1. In *COMPCON '96: Proceedings of the 41st IEEE International Computer Conference*, page 319, Washington, DC, USA, 1996. IEEE Computer Society.

[41] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems. *IEEE Design and Test of Computers*, 18(5):21–31, september/october 2001.

[42] Richard Goering. Platform-based design: A choice, not a panacea. *EETimes Magazine*, November 2002. http://www.eetimes.com/story/OEG20020911S0061.

[43] Frank Schirrmeister, Martin Meindl, and Stan Krolikoski. IP Authoring and Integration for HW/SW Co-design and Reuse - Lessons Learned. In *Proceeding of the Ninth IEEE/DATC Eletronic Desing Process Workshop*, Montery, CA, USA, April 2002.

[44] Xilinx Inc. *Virtex-II Pro/ProX FPGAs Overview* , 2005. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex.

[45] Tensilica. *The Xtensa 6 Processor for SOC Design*, 2005. http://www.tensilica.com/products/xtensa_6.htm.

[46] SONIC Smart Interconnects. *SiliconBackplane III*, 2005. http://www.sonicsinc.com/sonics/products/siliconbackplaneIII/.

[47] Gaisler Research Laboratory. *LEON2 XST User's Manual*, 1.0.22 edition, May 2004.

[48] Damjan Lampret. *OpenRISC 1200 IP Core Specification*. OpenCores, 0.6 edition, jun 2001.

[49] A. N. Habermann, Lawrence Flon, and Lee Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Commun. ACM*, 19(5):266–272, 1976.

[50] Peter W. Madanty. JavaOS: A Standalone Java Environment. Technical report, Sun Microsystems White Paper, May 1996.

[51] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1987.

[52] D. Jeff Dionne, Greg Ungerer, David McCullough, Ryan McDonald, and Mike Durrant. *uClinux: Embedded Linux/Microcontroller Project*, 2005. http://www.uclinux.org.

[53] George Neville-Neil and Telle Whitney. SoC: Software, Hardware, Nightmare, Bliss. *ACM Queue*, 1(2):24, 2003.

[54] Rob A. Rutenbar, Max Baron, Thomas Daniel, Rajeev Jayaraman, Zvi Or-Bach, Jonathan Rose, and Carl Sechen. (When) Will FPGAs kill ASICs? In *Proceedings of the 38th conference on Design automation*, pages 321–322. ACM Press, 2001.

[55] Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall PTR, 1 edition, 2002.

[56] Lothar Baum. Towards Generating Customized Run-time Platforms from Generic Components. In *Proceedings of the 11th Conference on Advanced Systems Engineering*, Heidelberg, Germany, June 1999.

[57] Constantinos A. Constantinides, Atef Bader, Tzilla H. Elrad, P. Netinant, and Mohamed E. Fayad. Designing an Aspect-oriented Framework in an Object-oriented Environment. *ACM Comput. Surv.*, 32(1es):41, 2000.

[58] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition, 1994.

[59] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[60] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[61] Leonid Mikhajlov and Emil Sekerinski. A Study of the Fragile Base Class Problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 355–382. Springer-Verlag, 1998.

[62] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, Mar. 1976.

[63] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[64] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., July 2000.

[65] Tiago Stein D'Agostini and Antônio Augusto Fröhlich. Bridging AOP to SMP: turning GCC into a metalanguage preprocessor. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1563–1564, New York, NY, USA, 2005. ACM Press.

[66] David R. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of the First International Joint' Conference of ISSAC and AAECC*, number 358 in Lecture Notes in Computer Science, pages 13–25, Rome, Italy, July 1989. Springer.

[67] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 3 edition, 2000.

[68] Gustavo Fortes Tondello and Antônio Augusto Fröhlich. On the Automatic Configuration of Application-Oriented Operating Systems. In *3rd ACS/IEEE International Conference on Computer Systems and Applications*, Cairo, Egypt, Jan. 2005.

[69] Arliones Stevert Hoeller Junior. Application-oriented power management for embedded systems. Technical report, Universidade Federal de Santa Catarina, Florianópolis, Brazil, Dec 2004.

[70] Pablo Viana, Edna Barros, Sandro Rigo, Rodolfo Azevedo, and Guido Araujo. Exploring Memory Hierarchy with ArchC. In *Proceedings of the 15th Symposium on*

Computer Architecture and High Performance Computing, São Paulo, Brazil, October 2003. IEEE CS.

[71] Gustavo Fortes Tondello. Uma ferramenta de gerenciamento de conhecimento de configuração de componentes de software segundo a metodologia de Projeto de Sistemas Orientados à Aplicação. B.sc. thesis, Federal University of Santa Catarina, Feb. 2004.

[72] Xilinx Inc. *MicroBlaze and Multimedia Development Board User Guide*, ug020 (v1.0) edition, 2002.

[73] National Semiconductor. *AC'97 Rev 2.1 Codec with Sample Rate Conversion and National 3D Sound*, 2000.

[74] Eduardo Billo. Playing MP3 on LEON2/uClinux. Technical report, Unicamp, 2003. http://www.lsc.ic.unicamp.br / billo /mp3onleon.htm.

Apêndice A

DTD da linguagem de descrição de componentes

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT family (common*, interface, member+, (feature | dependency | trait)*)>
<!ATTLIST family name   ID #REQUIRED
                 type   (abstraction | mediator |
                         aspect | hardware) #REQUIRED
                 class  (uniform | incremental |
                         combined | dissociated) #REQUIRED>

<!ELEMENT common (type*, constant*)>

<!ELEMENT interface (type*, constant*, constructor*, method*)>

<!ELEMENT member (super*,interface,property*,(feature | dependency | trait)*)>
<!ATTLIST member name   ID #REQUIRED
                 type   (exclusive | inclusive) "inclusive"
                 arch   (IA32 | PPC32 | ARM | AVR8 | SPARCV8 | OR32 ) #IMPLIED
                 mach   (PC | Khomp | iPAQ | ATMega | LEON2 | OR1K ) #IMPLIED
                 cost   CDATA #REQUIRED>
```

```
<!ELEMENT feature (trait | dependency)*>
<!ATTLIST feature name    CDATA #REQUIRED
                  default (true|false) #REQUIRED>

<!ELEMENT property EMPTY>
<!ATTLIST property name   CDATA #REQUIRED>

<!ELEMENT dependency EMPTY>
<!ATTLIST dependency requisit CDATA #REQUIRED>

<!ELEMENT trait EMPTY>
<!ATTLIST trait name  CDATA #REQUIRED
                type  CDATA #REQUIRED
                value CDATA #REQUIRED>

<!ELEMENT type EMPTY>
<!ATTLIST type name CDATA #REQUIRED
               type (class | structure | enumeration | union |
                     synonym | CDATA) #REQUIRED
               value CDATA #IMPLIED>

<!ELEMENT constant EMPTY>
<!ATTLIST constant name  CDATA #REQUIRED
                   type  CDATA #REQUIRED
                   value CDATA #REQUIRED>

<!ELEMENT super EMPTY>
<!ATTLIST super name CDATA #REQUIRED>

<!ELEMENT constructor (parameter*)>
```

```
<!ELEMENT method (parameter*)>
<!ATTLIST method name       CDATA #REQUIRED
                 return     CDATA #IMPLIED
                 qualifier  (class | polymorphic | abstract) #IMPLIED>

<!ELEMENT parameter EMPTY>
<!ATTLIST parameter name    CDATA #REQUIRED
                    type    CDATA #REQUIRED
                    default CDATA #IMPLIED>
```

Apêndice B

Descrição da família de abstrações

Thread

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE family SYSTEM "family.dtd">

<family name="Thread" type="abstraction" class="incremental">

  <common>

    <type name="Self" type="enumeration" value="SELF"/>

    <type name="State" type="enumeration" value=
    "RUNNING,READY,SUSPENDED,WAITING,FINISHING"/>

    <type name="Priority" type="enumeration" value=
    "HIGH=0,NORMAL=15,LOW=31"/>

  </common>
```

```
<interface>

  <constructor>
    <parameter name="entry" type="int␣(*)()"/>
    <parameter name="..." type=’""’/>
    <parameter name="state" type="const␣State␣&amp;" default="READY"/>
    <parameter name="priority" type="const␣Priority␣&amp;" default="NORMAL"/>
  </constructor>

  <method name="state" return="volatile␣const␣State␣&amp;"></method>
  <method name="priority" return="Priority"></method>
  <method name="priority">
    <parameter name="priority" type="const␣Priority␣&amp;"/>
  </method>

  <method name="join" return="int"></method>
  <method name="pass"></method>
  <method name="suspend"></method>
  <method name="resume"></method>

  <method name="yield" qualifier="class"></method>
  <method name="exit" qualifier="class">
    <parameter name="status" type="int" default="0"/>
  </method>

</interface>
```

```
  <member name="Exclusive_Thread" type="exclusive" cost="2">

    <interface>

      <constructor>
        <parameter name="entry" type="int␣(*)()"/>
        <parameter name="..." type='""'/>
      </constructor>

      <method name="exit" qualifier="class">
        <parameter name="status" type="int" default="0"/>
      </method>

    </interface>

  </member>

<!-- ************************************************************ -->

  <member name="Cooperative_Thread" type="exclusive" cost="4">

    <interface>

      <super name="Exclusive_Thread"/>

      <constructor>
        <parameter name="entry" type="int␣(*)()"/>
        <parameter name="..." type='""'/>
      </constructor>

      <method name="state" return="volatile␣const␣State␣&amp;"></method>

      <method name="pass"></method>

      <method name="exit" qualifier="class">
        <parameter name="status" type="int" default="0"/>
      </method>

    </interface>

  </member>
```

```
<member name="Concurrent_Thread" type="exclusive" cost="8">

  <interface>

    <super name="Cooperative_Thread"/>

    <constructor>
      <parameter name="entry" type="int␣(*)()"/>
      <parameter name="..." type='""'/>
      <parameter name="state" type="const␣State␣&amp;" default="READY"/>
    </constructor>

    <method name="state" return="volatile␣const␣State␣&amp;"></method>

    <method name="join" return="int"></method>
    <method name="pass"></method>
    <method name="suspend"></method>
    <method name="resume"></method>

    <method name="yield" qualifier="class"></method>
    <method name="exit" qualifier="class">
      <parameter name="status" type="int" default="0"/>
    </method>

  </interface>

  <dependency requisit="ABSTRACTION::Timepiece::Alarm"/>

</member>
```

```
<member name="Priority_Thread" type="exclusive" cost="10">

  <interface>

    <super name="Concurrent_Thread"/>

    <constructor>
      <parameter name="entry" type="int␣(*)()"/>
      <parameter name="..." type='""'/>
      <parameter name="state" type="const␣State␣&amp;" default="READY"/>
      <parameter name="priority" type="const␣Priority␣&amp;" default="NORMAL"/>
    </constructor>

    <method name="state" return="volatile␣const␣State␣&amp;"></method>
    <method name="priority" return="Priority"></method>
    <method name="priority">
      <parameter name="priority" type="const␣Priority␣&amp;"/>
    </method>

    <method name="join" return="int"></method>
    <method name="pass"></method>
    <method name="suspend"></method>
    <method name="resume"></method>

    <method name="yield" qualifier="class"></method>
    <method name="exit" qualifier="class">
      <parameter name="status" type="int" default="0"/>
    </method>

  </interface>

  <dependency requisit="ABSTRACTION::Timepiece::Alarm"/>

</member>
```

```
  <feature name="active_scheduler" default="true">
    <trait name="QUANTUM" type="unsigned␣int" value="10000"/>
    <dependency requisit=
    "(::Thread::Concurrent_Thread␣||␣::Thread::Priority_Thread)"/>
  </feature>

  <dependency requisit="::Address_Space"/>

</family>
```