

Daniela Cristina Cascini Peixoto

LACCES: Uma Ferramenta para a Descrição da Arquitetura de Sistemas Embutidos

Dissertação apresentada ao Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte

Outubro de 2004

*Dedico este trabalho a meus pais,
Guiomar e Peixoto.*

Agradecimentos

Agradeço aos meus familiares, amigos e colegas que me deram o apoio necessário para a realização deste trabalho.

Gostaria de agradecer, particularmente, às seguintes pessoas:

Ao professor Diógenes pela orientação, amizade, competência e apoio demonstrados nesses cinco anos em que trabalhamos juntos.

Aos membros da banca, que generosamente deram o seu tempo no melhoramento desta dissertação.

Aos meus pais, pelo incentivo e pela colaboração nesses anos de estudos.

Aos amigos Luiz Filipe, Marcos Augusto, Fernanda, Breno Vitorino, Mateus, José Nacif, Otaviano e todos os demais amigos do LECOM (Laboratório de Engenharia de Computadores), pelo incentivo e pela colaboração no desenvolvimento deste trabalho.

Ao CNPq, Conselho Nacional de Pesquisa, pelo apoio financeiro deste trabalho.

Aos meus companheiros de trabalho do Synergia, agradeço pelo apoio e pela motivação.

Ao Fernando, pelo amor e pela compreensão nos momentos de dificuldade. Ele foi a minha âncora de apoio durante esse tempo de desenvolvimento do meu trabalho.

Minha gratidão também àqueles que, mesmo não tendo participado diretamente do trabalho, proporcionaram-me a paz de espírito necessária ao desenvolvimento de uma empreitada como esta. Destaco nesse grupo as minha amigas do vôlei: Amanda, Elaine e Graciane.

Resumo

Sistemas embutidos complexos não podem mais ser projetados sem alguma consideração da interação entre os domínios de hardware e de software.

As linguagens de descrição de hardware utilizadas atualmente como, por exemplo, VHDL e Verilog HDL possibilitam a descrição de sistemas no nível físico ou no nível de transferência entre registradores. Tais tipos de linguagens não possuem um nível de abstração suficiente para o projeto de sistemas complexos. Linguagens de descrição de arquitetura, ao contrário provêem um nível de abstração maior, focando nos aspectos estruturais do sistema, tais como componentes abstratos e suas interações.

O presente trabalho descreve um novo ambiente para especificar e capturar os elementos de projeto do nível de arquitetura de sistemas embutidos. A descrição da arquitetura é baseada na composição de componentes de hardware e de software com a adição de uma interface entre eles. Requisitos não funcionais especificados para os componentes e para as suas interfaces também podem ser modelados e verificados.

As idéias apresentadas neste trabalho foram implementadas em uma ferramenta. Esta ferramenta permite a descrição de um sistema embutido através de uma linguagem específica, LACCES. Ela também permite a simulação do comportamento e a verificação não funcional dos componentes de hardware e de software.

Abstract

Complex embedded systems can no longer be effectively designed without consideration of the interaction of hardware and software domains.

Current hardware description languages, such as VHDL and Verilog HDL focus on physical or register transfer logic levels and do not provide enough abstraction for complex system level designs. Architectural description languages provide higher levels of abstraction focusing on the structural aspects, such as abstract components and their interactions.

This work describes a new environment to specify and capture architectural design expertise of embedded systems. The architecture description is based on the composition of hardware and software components with the addition of an interface between them. Non-functional constraints for components and their interfaces can also be modeled and verified.

The ideas discussed in this work have been implemented in a tool. This tool enables the description of an embedded system through a specific language, LACCES. It can also allow the behavioral simulation and the non-functional verification of the hardware and software components.

Sumário

INTRODUÇÃO.....	11
1.1 MOTIVAÇÃO	11
1.2 REPRESENTAÇÕES.....	14
1.2.1 Arquitetura	14
1.2.2 Especificação	15
1.2.3 Modelo.....	15
1.3 OBJETIVO	16
1.4 LIMITES DO TRABALHO	16
1.5 VISÃO GERAL DE LACCES	17
1.5.1 Representação no Nível de Arquitetura	18
1.5.2 A Linguagem LACCES.....	22
1.6 ORGANIZAÇÃO DESTE DOCUMENTO	24
TRABALHOS RELACIONADOS.....	26
2.1 LINGUAGENS, FERRAMENTAS E AMBIENTES DE DESCRIÇÃO DE ARQUITETURA DE SOFTWARE E DE HARDWARE	26
2.1.1 Aesop.....	28
2.1.2 Acme.....	29
2.1.3 UniCon.....	29
2.1.4 Rapide	30
2.1.5 Wright.....	30
2.1.6 MICON.....	31
2.2 AMBIENTES DE PROJETO CONCORRENTE DE HARDWARE E SOFTWARE	32
2.2.1 Ptolemy.....	33
2.2.2 Lycos	33
2.2.3 POLIS.....	34
2.3 LINGUAGENS DE ESPECIFICAÇÃO DE SISTEMAS.....	34
2.3.1 SDL.....	35
2.3.2 SystemC.....	35
2.3.3 SpecC.....	36
2.3.4 Rosetta.....	37
2.3.5 VHDL	38
2.3.6 VERILOG HDL	38
2.3.7 Classificando as Linguagens de Especificação.....	38
2.4 REPRESENTAÇÃO INTERNA.....	39
2.4.1 SDS.....	39
2.5 MÉTODOS DE PROJETO DE SISTEMAS DE SOFTWARE EMBUTIDO.....	43
2.5.1 PECOS	43
2.5.2 Koala.....	44
PRIMITIVAS DO SISTEMA	45
3.1 COMPONENTES DE HARDWARE	45
3.1.1 Microcontrolador.....	46
3.1.2 Memória	47
3.1.3 CPU (Central Processing Unit).....	49
3.1.4 Conversores.....	50
3.1.5 Multiplicador.....	52

3.1.6	Controlador de LCD	53
3.1.7	Timer	53
3.1.8	Módulo LCD.....	56
3.1.9	RTC (Real Time Clock).....	57
3.1.10	Keypad	58
3.2	COMPONENTE DE SOFTWARE	59
3.2.1	Module.....	59
3.3	CONECTORES	59
3.3.1	Comunicação entre Componentes de Hardware.....	60
3.3.2	Comunicação entre Componentes de Hardware e Software.....	64
3.3.3	Comunicação entre Componentes de Software.....	66
A LINGUAGEM LACCES.....		68
4.1	ESTRUTURA DE LACCES	68
4.1.1	Representações.....	70
4.1.2	Sintaxe da Linguagem Estrutural de LACCES	72
4.1.3	Estendendo a Especificação Funcional com Propriedades Não Funcionais	74
4.2	COMPORTAMENTO DOS ELEMENTOS DE PROJETO.....	76
4.2.1	Processos.....	76
4.2.2	Métodos.....	77
4.2.3	Implementação	78
4.3	TIPOS DE DADOS.....	79
4.4	UNIDADES	80
DEFINIÇÃO FORMAL		82
5.1	SISTEMA.....	82
5.2	COMPUTAÇÃO.....	83
5.3	COMUNICAÇÃO	85
5.3.1	Refinamento da Comunicação	89
REPRESENTAÇÃO INTERNA		93
6.1	SEMÂNTICA DOS OBJETOS	94
6.2	DOMÍNIO DE FLUXO DE PROCESSO.....	96
6.3	DOMÍNIO DO FLUXO DE TEMPO E SEQUÊNCIA	99
6.4	DOMÍNIO DA ESTRUTURA LÓGICA	101
6.5	DOMÍNIO DA ESTRUTURA FÍSICA	103
6.6	CLASSES DE REFERÊNCIA	104
AMBIENTE DE PROJETO		106
7.1	PRÉ-PROCESSADOR.....	107
7.2	GERADOR DE CÓDIGO EM SYSTEMC	109
7.2.1	Visão Geral do Processo de Geração de Código.....	109
7.3	FERRAMENTA DE ANÁLISE	112
7.4	FERRAMENTA DE VERIFICAÇÃO DE TIPOS.....	113
VALIDAÇÃO DO AMBIENTE LACCES		114
8.1	ESTUDO DE CASO 1: UM TERMÔMETRO	114
8.1.1	Componentes.....	114
8.1.2	Conectores.....	116
8.1.3	Sistema	118
8.1.4	Simulação e Validação.....	119
8.2	ESTUDO DE CASO 2: DESCRIÇÃO DE UM SERVIDOR	119
8.2.1	Componentes.....	119

8.2.2	<i>Conectores</i>	120
8.2.3	<i>Sistema</i>	120
8.2.4	<i>Análise</i>	121
CONCLUSÃO		123
9.1	TRABALHOS FUTUROS	124
REFERÊNCIAS BIBLIOGRÁFICAS		126
APÊNDICES		130
A.1	GRAMÁTICA	130
A.2	UNIDADES DE MEDIDA.....	133
A.3	SEMÂNTICA DAS PROPRIEDADES	134

Lista de figuras

FIGURA 1 – Etapas de um projeto. [49].....	13
FIGURA 2 –Objetos e atividades do ambiente LACCES.	18
FIGURA 3 – Modelo tradicional de representação.	20
FIGURA 4 – Níveis de abstração.....	21
FIGURA 5 –Entidades principais de LACCES.	23
FIGURA 6 – Os 4 domínios de LACCES. [49].....	24
FIGURA 7 – Representação da arquitetura de um sistema.....	28
FIGURA 8 – Projeto de um sistema heterogêneo.	32
FIGURA 9 – Os domínios de SDS e os tipos de relacionamento entre os elementos.	41
FIGURA 10 – Modelo de informação de SDS.	43
FIGURA 11 - Exemplo de um MCU.	47
FIGURA 12– Descrição em LACCES de um sistema cliente-servidor.	71
FIGURA 13– Representação hierárquica do servidor.	72
FIGURA 14– Descrição de um sistema cliente-servidor com o detalhamento do servidor.	74
FIGURA 15– Gramática BNF parcial para uma instância estrutural simples da linguagem.	75
FIGURA 16– Sintaxe das propriedades.....	76
FIGURA 17 – Sintaxe do processo.	77
FIGURA 18– Sintaxe do método.	78
FIGURA 19– Sintaxe da implementação do comportamento.....	78
FIGURA 20 – Sintaxe da construção <i>Has</i>	79
FIGURA 21– Sintaxe da construção <i>Packet</i>	79
FIGURA 22 – Sintaxe da definição de unidades.	80
FIGURA 23– Unidade espaço.....	80
FIGURA 24– <i>Template</i> de um canal SPI.....	86
FIGURA 25 – Conector Virtual.	88
FIGURA 26– Descrição em LACCES do conector.	90
FIGURA 27– Implementação da comunicação entre um software e uma memória.....	91
FIGURA 28– Mapeamento do módulo de Software na CPU.....	92
FIGURA 29 – Diagrama das classes relacionadas à computação.....	94
FIGURA 30– Diagrama das classes relacionadas à comunicação.....	95
FIGURA 31 – Classes relacionadas à interconexão das classes de comunicação e computação.....	96
FIGURA 32– Classes do domínio de fluxo de processo.	97
FIGURA 33– Classes do domínio de fluxo de tempo e seqüência.	99
FIGURA 34– Classes do domínio de estrutura lógica.	102
FIGURA 35– Classes do domínio da estrutura física.	103
FIGURA 36– Representação do modelo de referência.....	105
FIGURA 37– Ferramentas disponíveis no ambiente de projeto.....	107
FIGURA 38– Descrição de uma arquitetura utilizando macros.	108
FIGURA 39– Código fonte da operação de escrita.....	111
FIGURA 40–Descrição em LACCES da CPU.....	115
FIGURA 41 – Componentes de hardware do termômetro.....	116
FIGURA 42 –Descrição em LACCES dos componentes de software.	117
FIGURA 43 –Descrição em LACCES dos SFRs.	117
FIGURA 44 –Comunicação entre componentes de hardware-software.	118
FIGURA 45 –Descrição em LACCES da comunicação hardware/software.....	119
FIGURA 46 – <i>Trace</i> da execução do Termômetro.....	120
FIGURA 47–Comportamento dos sub-componentes do servidor.	121
FIGURA 48–Mensagens geradas pela ferramenta de análise.....	122

Lista de tabelas

TABELA 1 – Classificação das linguagens de especificação.	39
TABELA 2 – Primitivas do nível físico de um MCU.	46
TABELA 3 – Primitivas de uma memória.....	49
TABELA 4 – Primitivas de uma CPU.....	50
TABELA 5 – Primitivas dos conversores.....	52
TABELA 6 – Primitivas do multiplicador.....	53
TABELA 7 – Primitivas do controlador de LCD.....	53
TABELA 8 – Primitivas do temporizador.	54
TABELA 9 – Primitivas do timer/counter.....	55
TABELA 10 – Primitivas do Evento.	55
TABELA 11 – Primitivas do PWM.....	56
TABELA 12 – Primitivas do Módulo LCD.	57
TABELA 13 – Primitivas do RTC.....	58
TABELA 14 – Primitivas do Keypad.....	58
TABELA 15 – Primitivas do Módulo.....	59
TABELA 16 – Classificação dos tipos de conectores.....	60
TABELA 17 – Primitivas do SPI.....	61
TABELA 18 – Primitivas do I ² C.	62
TABELA 19 – Primitivas da UART.....	64
TABELA 20 – Primitivas do conector Analógico.....	64
TABELA 21 – Primitivas do conector Paralelo.	65
TABELA 22 – Primitivas do conector FIFO.....	66
TABELA 23 – Primitivas do Semáforo.....	66
TABELA 24 – Primitivas do IPC.	67
TABELA 25 – Visão geral das construções primitivas da linguagem LACCES.....	69
TABELA 26 – Descrição dos arquivos produzidos.	110
TABELA 27 - Semântica dos predicados comportamentais de LACCES.....	135
TABELA 28 - Semântica dos predicados de tempo de LACCES.....	136
TABELA 29 - Semântica dos predicados estruturais de LACCES.....	137
TABELA 30 - Semântica dos predicados físicos de LACCES.....	137

Capítulo 1

Introdução

Este trabalho visa a confecção de uma ferramenta e de uma notação para a descrição da arquitetura de sistemas embutidos. A arquitetura é utilizada para descrever a estrutura de um sistema, ou seja, os componentes e os conectores que a compõem. Ela é útil, pois torna sistemas complexos tratáveis, caracterizando-os em um alto nível de abstração.

1.1 Motivação

Atualmente, o projeto de sistemas embutidos é bastante complexo e demanda um grande esforço de desenvolvimento. A complexidade é devida a conjunção dos seguintes fatores principais:

Heterogeneidade: como todos os sistemas de computação, um computador embutido é composto de elementos dos domínios de hardware e software.

Quando se iniciou o projeto de microprocessadores, no final da década de 70 e início da década de 80, muito do tempo de desenvolvimento de tais sistemas era gasto no projeto do hardware, em definir o mapeamento de memória, as entradas/saídas entre outras características. Quando o projeto do hardware estava completo, um programa comparativamente mais simples era desenvolvido, limitado no tamanho e na complexidade pela capacidade de armazenamento da memória utilizada. Atualmente, a maior parte do sistema de hardware está contida em um único *chip*, na forma de um microcontrolador. Soma-se a isso, o desenvolvimento da tecnologia das memórias que permite o uso de programas maiores e mais sofisticados [57].

A heterogeneidade existente nos sistemas embutidos, força os projetistas, ao tomarem alguma decisão, considerarem informações de vários domínios. Ao projetar um sistema é necessário levar em conta aspectos distintos do comportamento dos seus componentes, como também requisitos não funcionais, tais como: restrições de desempenho, custo, temporizações, consumo de energia e área de armazenamento. Isso introduz problemas no fluxo de projeto tradicional, pois a informação de diferentes domínios é tipicamente expressa utilizando

linguagens, formalismos e notações distintas.

Integrar esses domínios em uma única metodologia traz várias vantagens: acelera o processo de desenvolvimento do projeto e permite verificar as incompatibilidades existentes entre os elementos do sistema.

Níveis de Abstração: em um sistema embutido, soma-se ao problema do tamanho do projeto, as dificuldades dos projetistas em utilizar um nível de abstração mais elevado. Conseqüentemente, os projetistas utilizam o mesmo baixo nível de abstração para sistemas que estão cada vez mais complexos.

Questões de tempo: alguns sistemas embutidos operam com rígidas restrições de tempo, o que é uma característica de operar em tempo real. Esse estilo de funcionamento é diferente da operação dos computadores pessoais. Enquanto, por exemplo, pode-se tolerar a espera pela computação de um programa, não se pode esperar que o sistema de antiderrapagem de um carro decida se deve ser utilizado ou não os freios.

Os projetistas devem entender completamente os requisitos de tempo a serem atendidos pelo sistema e devem ser capazes de representá-los adequadamente na notação utilizada.

Confiabilidade: fornecedores de softwares destinados aos computadores pessoais vendem os mesmos sabendo que eles contêm erros (*bugs*). É importante vendê-los o quanto antes, e as correções somente são distribuídas a medida que os erros são descobertos. Fornecedores de sistemas embutidos não podem dar-se a esse luxo. Um erro de software em um sistema pode provocar danos físicos e, conseqüentemente, destruir a reputação do fabricante para sempre. Portanto, o projeto de um sistema embutido deve ser desenvolvido utilizando metodologias adequadas tanto para a construção do hardware quanto para o do software, além de utilizar ferramentas de teste para validações dos requisitos funcionais e não funcionais do sistema.

Mercado: o mercado em que os sistemas embutidos são vendidos é bastante competitivo. O desafio é ainda maior devido aos rápidos avanços da tecnologia. Os projetistas possuem como objetivo desenvolver produtos novos em um curto intervalo de tempo.

Avaliando-se todos esses fatores e características pode-se concluir que um **sistema embutido** é um sistema dedicado a tarefas específicas. Esses sistemas possuem restrições de tempo real, freqüentemente tem restrições de energia, são implementados usando um microcontrolador, utilizando software especializado, são confiáveis, autônomos, interativos com o homem ou com uma rede, operam em diversos ambientes e com diversas variáveis físicas, e são vendido em um mercado competitivo.

Devido a complexidade dos sistemas embutidos e a grande quantidade de informação a ser capturada, quanto mais cedo um erro de projeto for detectado, mais rápido será o seu ciclo de desenvolvimento. Além disso, o custo de correção do erro é bem menor do que o de manutenção do sistema.

Para evitar ter que lidar com uma grande quantidade de informação, usualmente é

utilizado um alto nível de abstração para representar os requisitos do projeto. Nesse nível, somente detalhes necessários do sistema são capturados, isto é, a funcionalidade e o comportamento que são independentes da implementação. Com essa informação, validações podem ser realizadas com a finalidade de verificar as propriedades modeladas e evitar posteriores manutenções que na maioria das vezes custam muito caro.

Após essa modelagem inicial, refinamentos sucessivos adicionam maiores informações ao sistema descrito até se chegar a uma implementação física, como mostrado na Figura 1. Essa metodologia de projeto é denominada *top-down*. Ela captura o comportamento dinâmico de um sistema ao refinar a sua visão conceitual em termos de componentes mais detalhados. Uma outra metodologia é a *bottom-up* que captura a visão estrutural e estática de um sistema, compondo os componentes com o objetivo de obter estruturas mais complexas. Ela permite também que detalhes de baixo nível sejam incorporados ao projeto.

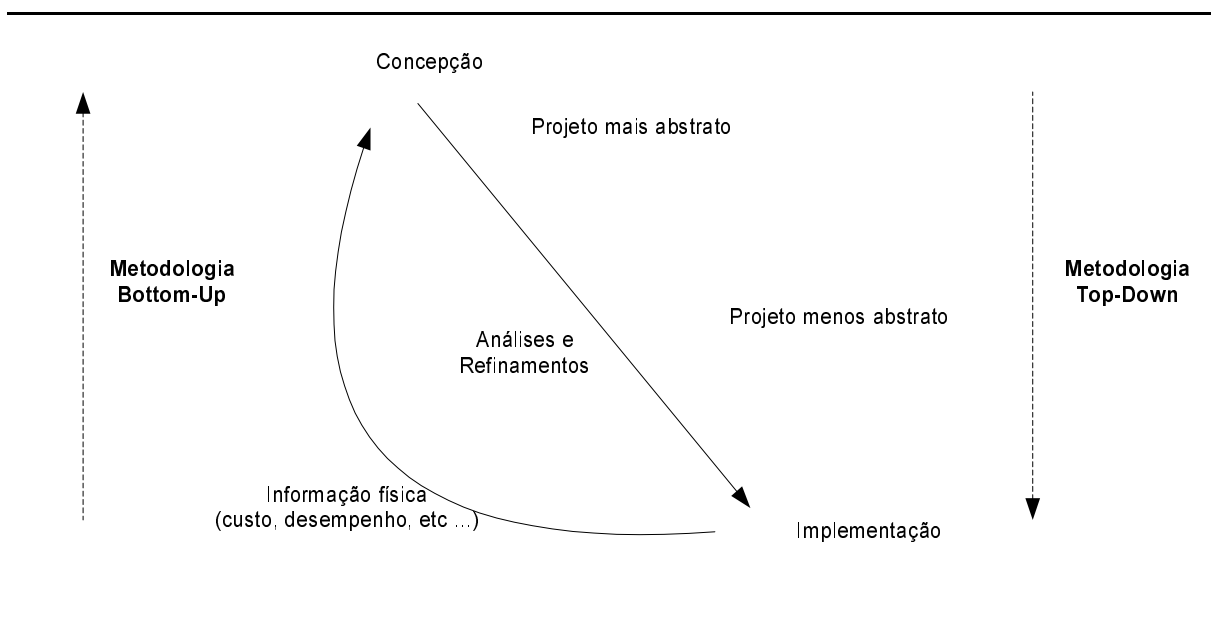


FIGURA 1 – Etapas de um projeto. [49]

É virtualmente impossível projetar um sistema começando por alguma descrição que é independente de uma arquitetura. Mesmo que os projetistas prefiram uma metodologia *top-down*, eles raramente a usam de maneira isolada. Os projetistas iniciam com pelo menos uma arquitetura em mente e usam seus conhecimentos sobre os componentes de nível físico para guiar as decisões de projeto. Além disso, eles já são capazes de atribuir funcionalidades aos componentes utilizados.

1.2 Representações

Uma representação de um projeto pode ser expressa em três visões distintas: arquitetura, especificação ou modelo.

1.2.1 Arquitetura

Um aspecto crítico no projeto de um sistema complexo é a sua arquitetura [21, 51]. Uma arquitetura é uma descrição estruturada de uma funcionalidade desejada. A estrutura é utilizada para lidar com a complexidade de alguns sistemas, tornando mais fácil o seu entendimento. Uma descrição de arquitetura descreve aspectos de alto nível, como a organização, a decomposição em componentes e a maneira como os componentes interagem.

A utilização da descrição de arquitetura é importante por duas razões. Primeiramente, uma descrição de arquitetura torna um sistema complexo mais tratável, pois o caracteriza em um alto nível de abstração. Em particular, a arquitetura expõe decisões de alto nível do projeto, permitindo que o projetista decida antes da implementação as propriedades do sistema. Propriedades típicas incluem protocolo de interação, desempenho, largura de banda e a localização da estrutura de armazenamento dos dados. Além disso, o projeto no nível de arquitetura permite análises especializadas antes do início do desenvolvimento do sistema propriamente dito.

Enquanto uma descrição de arquitetura utiliza um nível maior de abstração, as implementações correspondentes são escritas em linguagens de programação. Para um projetista de software, a atividade de compor um sistema a partir de subsistemas é substancialmente diferente da atividade de programar algoritmos e estruturas de dados. Projetistas de hardware confrontam com o mesmo problema; eles reconhecem a existência de diferentes níveis de abstração, cada um com os seus modelos, notações e técnicas de análise. Da mesma maneira, diferentes níveis de abstração requerem diferentes tipos de componentes, diferentes maneiras de compor componentes e diferentes tipos de raciocínio. A lacuna existente entre uma especificação e uma implementação é substancial. Preencher essa lacuna requer melhores modelos e notações para os passos intermediários. Esse é o principal objetivo de uma descrição de arquitetura.

Freqüentemente, as descrições de arquitetura são representadas de uma maneira informal através de diagramas de blocos e linhas, retratando a organização estrutural grosseira do sistema. Enquanto essas descrições provêem uma documentação muito rica, o nível de informalidade limita a sua utilidade. Não fica claro qual o significado da descrição da arquitetura, tornando-se impossível analisá-la quanto a sua consistência e outras propriedades.

Evidentemente, é necessária uma base mais rigorosa para descrever a arquitetura dos sistemas. Pelo menos, deve-se ser capaz de descrever qual o significado do bloco e da linha, ou

seja, a semântica utilizada na descrição. Além disso, deve-se ser capaz de validar a descrição em termos da consistência em relação a adequação dos componentes e das interconexões utilizadas e dos requisitos não funcionais.

1.2.2 Especificação

A especificação de um sistema baseia-se no processo de coletar os requisitos do cliente. Os requisitos consistem de características que definem os critérios de aceitação do produto [42]. O projetista deve traduzir esses requisitos, usualmente expressos de uma maneira informal, em uma representação mais formal adequada a manipulações automáticas.

Linguagens de especificação expressam de uma maneira formal o desejo dos clientes, ou seja, quais comportamentos que o projeto deve possuir. Essas linguagens são geralmente imperativas, sendo utilizadas para abstrair detalhes e apresentar uma descrição mais simples e clara de um comportamento complexo. Linguagens de programação são um bom exemplo de tais linguagens, onde um simples comando em alto-nível é traduzido em uma coleção de várias instruções.

1.2.3 Modelo

Enquanto uma especificação descreve os requisitos do projeto, um modelo descreve a sua implementação [17]. Linguagens de modelagem expressam o que o sistema será capaz de fazer. Um modelo de um determinado sistema diz, por exemplo, qual o atraso da entrada para a saída, enquanto uma especificação diz qual é o atraso máximo esperado.

Linguagens de modelagem são usualmente declarativas, e descrevem um sistema como uma coleção de comportamentos que são executados e que podem ser descritos no domínio de hardware ou de software. A arquitetura, por outro lado, descreve a estrutura de um sistema, ou seja, os componentes que a compõem.

Um modelo pode ser matemático, nesse caso, ele é visto como um conjunto de asserções sobre propriedades do sistema, tais como funcionalidade ou dimensões físicas.

Um modelo também pode ser construtivo, nesse caso, ele define um procedimento computacional que reproduz um conjunto de propriedades do sistema. Modelos construtivos são freqüentemente usados para descrever o comportamento de um sistema em resposta a estímulos externos. Modelos construtivos são também chamados de modelos executáveis.

Quando o modelo executável é claramente distinto do sistema que ele modela, ele é denominado simulação. Entretanto, em muitos sistemas eletrônicos, um modelo que se inicia como uma simulação transforma-se em uma implementação de software do sistema. Neste caso, a distinção existente entre o modelo e o sistema propriamente dito torna-se meio obscura.

Modelos executáveis são construídos com base em um modelo de computação (MOC). Um MOC descreve a sintaxe e semântica que governa a interação dos elementos em um modelo. Alguns dos modelos de computação mais conhecidos são: máquinas de estado finito, *Petri Nets*, eventos discretos, redes de fluxos de dados e sistemas de comunicação concorrente.

A maioria das linguagens de hardware tinham como objetivo inicial a modelagem do projeto e, historicamente, isso foi devido a necessidade de simular o comportamento. Inicialmente, os sistemas a serem simulados eram pequenos e as propriedades físicas podiam ser examinadas. Quando os circuitos tornaram-se complexos, essas simulações tornaram-se custosas e modelos HDL (*Hardware Description Language*), originalmente projetados para a simulação, passaram a ser interpretados como especificações.

1.3 Objetivo

A proposta principal deste trabalho consiste na confecção de uma notação e de uma ferramenta para a representação da arquitetura de sistemas embutidos. Para lidar com as limitações atuais do projeto de tais tipos de sistemas, foi definida uma notação que trata dos seguintes aspectos:

- Clara separação entre conceitos: computação e comunicação, estrutura e comportamento, e funcionalidade e tempo;
- Representação formal de um sistema;
- Validação funcional e não funcional;
- Habilidade de capturar uma grande variedade de propriedades, que incluem: funcionalidade, tempo, estrutura lógica e física.

Para que essa notação seja útil, foi implementado um ambiente que permite a descrição de sistemas embutidos utilizando a notação proposta. Esse ambiente traduz a descrição do sistema para uma representação interna, permitindo a simulação do comportamento e análise das suas propriedades estruturais, temporais e físicas.

1.4 Limites do Trabalho

Tão importante quanto enumerar os objetivos do trabalho é esclarecer os limites do mesmo, visando delimitar mais precisamente o escopo que está sendo tratado.

Em primeiro lugar, é importante ressaltar que o produto deste trabalho é um ambiente

para a descrição de arquitetura e não para a sua co-síntese. O que o trabalho propõe é uma linguagem e um ambiente para a descrição da arquitetura de um sistema embutido, além de permitir análise das informações descritas.

A segunda ressalva é referente às primitivas que compõem a linguagem. Apesar deste trabalho propor um conjunto básico de primitivas que permita aos projetistas descreverem a arquitetura de um sistema embutido, esse conjunto não deve ser encarado como uma proposta fixa e universal. Ele pode ser aumentado para lidar com outros tipos de ambientes que possuam componentes de hardware-software. Como será apresentado ao longo do texto, o conjunto ideal de primitivas depende diretamente dos objetivos que se pretende atingir com a arquitetura. Se os objetivos mudam, por consequência, as primitivas também sofrem alteração.

A terceira ressalva é que o foco deste trabalho é voltado para a descrição de sistemas embutidos. Mesmo que vários conceitos possam ser aplicados também no desenvolvimento de outras aplicações, a utilização desse ambiente nesses contextos possivelmente exigirá uma série de adaptações.

Finalmente, o dialeto utilizado pode não se adequar a descrição de arquitetura de sistemas legados. O nosso objetivo foi descrever uma notação que capturasse as necessidades atuais da indústria de sistemas embutidos.

1.5 Visão Geral de LACCES

LACCES (*Language of Components and Connectors for Embedded Systems*) [45] consiste de uma linguagem e de uma ferramenta que permite a descrição, simulação e análise da arquitetura de sistemas embutidos.

Uma arquitetura descrita na linguagem LACCES é traduzida para uma representação interna que é fortemente baseada em SDS [49].

A Figura 2 ilustra os objetos consumidos e as atividades realizadas no ambiente LACCES. Com exceção da representação interna, todos os outros elementos (objetos e atividades) foram criados e implementados a partir desta dissertação.

As seções seguintes dão uma visão geral dos objetivos e dos conceitos utilizados por esta nova linguagem.

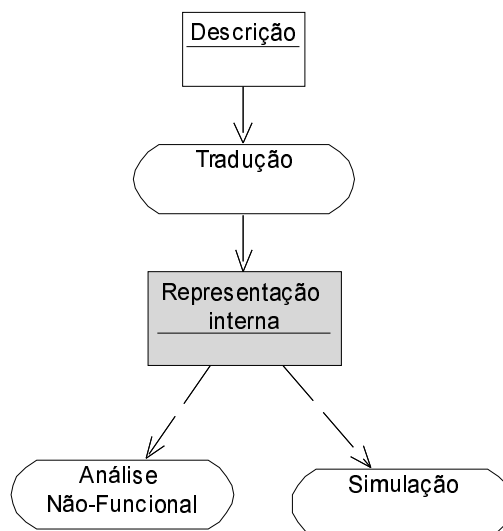


FIGURA 2 –Objetos e atividades do ambiente LACCES.

1.5.1 Representação no Nível de Arquitetura

LACCES tem por objetivo ajudar os projetistas na definição inicial da arquitetura de um sistema embutido, usando a abstração que eles acharem úteis e permitindo a simulação e a análise do projeto final.

Uma representação no nível de arquitetura adequada para o desenvolvimento de um sistema embutido deve atender a um conjunto de requisitos:

- Deve ser *abstrata* o suficiente, permitindo o encapsulamento e a descrição hierárquica do sistema;
- Deve permitir a *separação de conceitos* entre funcionalidade e tempo, comportamento e estrutura, computação e comunicação;
- Deve ser capaz de capturar as informações obtidas através da utilização de *metodologias de projeto top-down e bottom-up*;
- Deve permitir a *análise do sistema* projetado.

1.5.1.1 Abstração

Os sistemas tendem a ser complexos e os projetistas usualmente impõem algum tipo de particionamento lógico ou método estruturado, quebrando o sistema em módulos menores que são mais fáceis de serem descritos. Esses módulos usualmente são separados de acordo com a funcionalidade que eles provêem.

Módulos abstratos são usados para representar um comportamento desejado. No nível de arquitetura, eles podem ser vistos como uma “caixa preta”. O projetista atribui um comportamento desejado ao módulo, o qual interage com o ambiente e com outros módulos através de portas.

Módulos abstratos podem ser hierárquicos, contendo sub-módulos. A hierarquia consiste na definição de um objeto pela agregação ou refinamento de outros objetos mais simples. Hierarquia é uma estratégia para lidar com a complexidade de sistemas, possuindo como vantagem o reuso de objetos.

A hierarquia pode ser dividida em duas metodologias principais:

- Composição: um objeto é composto de uma coleção de objetos mais simples, o que o torna um objeto mais complexo.
- Refinamento: um objeto possui diferentes visões, com um detalhamento maior entre elas.

Módulos comportamentais são módulos abstratos, onde a hierarquia é simples de ser entendida e pode ser considerada como uma composição de funções. Módulos físicos podem ser compostos de sub-módulos físicos e de sub-módulos comportamentais.

Com o objetivo de permitir que módulos físicos e comportamentais (abstratos) coexistam na mesma descrição de um sistema, eles devem ser descritos no mesmo nível de abstração. Para que isso seja possível, um encapsulamento é usado para filtrar os detalhes excessivos do módulo físico em uma representação mais abstrata.

1.5.1.2 Separação de Conceitos

Um paradigma no projeto de sistemas é a separação de conceitos no estágio inicial do desenvolvimento. Vários aspectos podem ser melhor explorado, provendo soluções alternativas para um mesmo projeto. Além disso, os conceitos são ortogonais, no sentido que eles podem ser manipulados independentemente uns dos outros. A separação ocorre entre os seguintes conceitos:

- Funcionalidade e tempo;
- Comunicação e computação;
- Funcionalidade e estrutura.

A separação existente entre a funcionalidade e o tempo é um paradigma bastante conhecido na comunidade que utiliza metodologias de alto nível. O tempo pode ser alterado sem afetar a funcionalidade. Daí, tempo e funcionalidade são conceitos distintos e ortogonais.

A separação entre comunicação e computação leva em consideração a complexidade de lidar com o que é executado por um módulo particular – como descrito por sua especificação - e como ele recebe dados e sinais, e envia os resultados. Em linguagens convencionais de programação de software é uma prática comum deixar a comunicação para o sistema operacional,

principalmente devido as limitadas possibilidades oferecidas. Entretanto, ao se escrever um módulo de software de tempo real ou embutido o programador necessita conhecer os detalhes de cada entrada e saída e usar bibliotecas específicas para permitir a comunicação e a sincronização. No projeto de hardware, esse problema torna-se mais complexo, pois o número de implementações possíveis é alto. Usualmente, os projetistas tomam as decisões em relação às entradas e saídas na etapa inicial do projeto, reduzindo, assim, o espaço de soluções disponíveis.

O modelo tradicional de um diagrama de blocos, tal como em VHDL [28] ou Verilog HDL [56], é mostrado na Figura 3. Ao atribuir valores para os sinais, de acordo com algum protocolo pré-definido, os processos P1 e P2 podem comunicar e trocar dados.

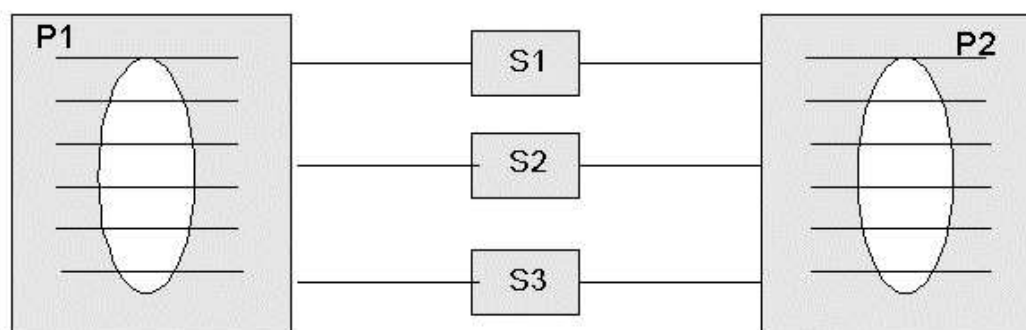


FIGURA 3 – Modelo tradicional de representação.

Nesse cenário, os processos P1 e P2 contêm código para realizar tanto a comunicação quanto a computação. A parte relacionada à comunicação está destacada em branco na figura. Por causa da comunicação e da computação estarem misturadas no código, elas não podem ser identificadas isoladamente por nenhuma ferramenta. Como resultado, não é possível trocar automaticamente o protocolo de comunicação, quando se alteram as restrições de projeto. Da mesma forma, também, torna-se impossível trocar o algoritmo de computação por um outro.

1.5.1.3 Diferentes Metodologias de Projeto

Projetistas humanos tendem a descrever os sistemas com conceitos gerais usando especificações abstratas. Através de refinamentos sucessivos, aumenta-se progressivamente a quantidade de informação descrita no projeto.

O método *top-down* é uma maneira bastante intuitiva de lidar com a complexidade de um sistema. Escondendo a informação indesejada ou desnecessária, o projeto pode ser mais facilmente manipulado. A maior vantagem do método *top-down* é a grande flexibilidade em explorar o conjunto de possíveis implementações. Isso leva a uma visão em pirâmide do projeto, onde uma decisão específica é modelada como o topo da pirâmide, e na base estão todas as possíveis soluções. Quanto mais detalhes são inseridos no projeto, o topo da pirâmide é movido

para baixo, reduzindo a altura e conseqüentemente o tamanho da base da pirâmide. Enquanto o processo de refinamento continua, o número de soluções possíveis é reduzido, gerando a implementação final.

A Figura 4 ilustra o princípio do refinamento sucessivo. Um sistema embutido, no nível de implementação, consiste de milhões de transistores. Tipicamente isso se reduz a alguns milhares de componentes no nível RTL. Entretanto, em um alto nível de abstração, o sistema é composto de alguns componentes, o qual inclui microprocessadores, memórias, hardwares específicos e barramentos.

Esta figura também ilustra o compromisso existente entre os níveis de abstração: quanto maior for o nível menor é a precisão, e vice-versa.

Mesmo se os projetistas preferirem um método *top-down*, eles raramente o usam isoladamente. O problema com descrições abstratas puramente funcionais é que elas são bem distantes da realização física. Devido a essa restrição, demanda-se um grande esforço para realizar o mapeamento entre as funções e os componentes de hardware, o que torna as atividades de projeto resultantes difíceis de serem automatizadas.

Uma possível solução para esse problema é utilizar a metodologia *bottom-up* conjuntamente com a *top-down*. A metodologia *bottom-up* possibilita que os projetistas incorporem detalhes de baixo nível em seus projetos. Esses detalhes permitem um maior refinamento do espaço de solução, nos estágios iniciais de desenvolvimento.

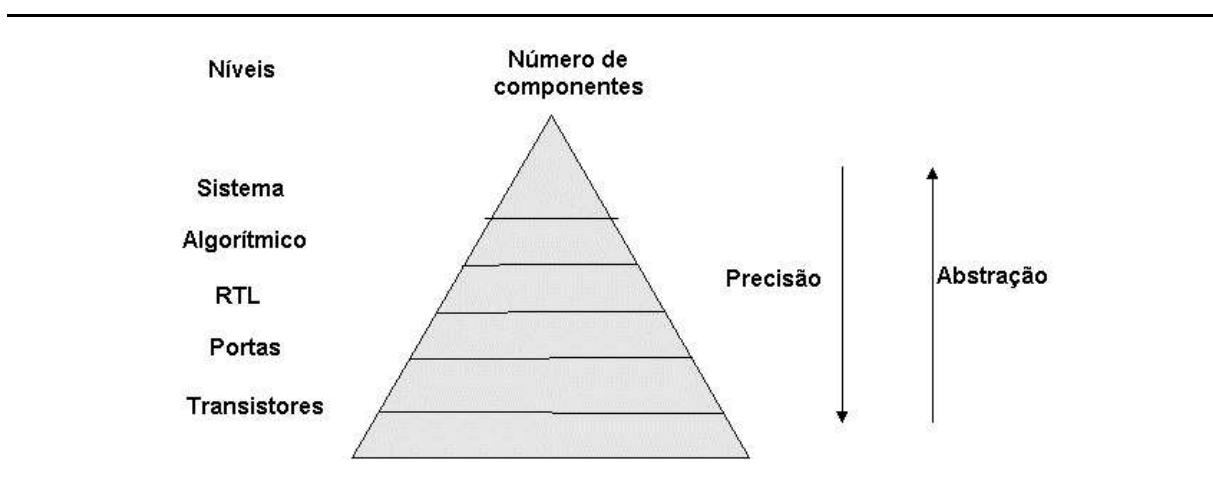


FIGURA 4 – Níveis de abstração.

Os requisitos de projeto, tais como custo e desempenho são candidatos usuais a esse conjunto de detalhes obtidos através da metodologia *bottom-up*. Além disso, outras características importantes podem ser descritas, de acordo com a finalidade do projeto, como, por exemplo, os tipos dos dispositivos de entrada e saída e as suas dimensões físicas.

Conclui-se que uma metodologia adequada para o projeto de sistemas embutidos deve

envolver tanto a utilização de métodos *top-down* quanto de métodos *bottom-up*.

1.5.1.4 Análise da Arquitetura

Em geral, linguagens informais tendem a ser mais intuitivas e expressivas do que linguagens formais. Entretanto, linguagens formais são geralmente mais adequadas para análises rigorosas.

Evidentemente, é necessária uma base formal para a descrição de uma arquitetura de qualquer tipo de sistema. Pelo menos, deve-se ser capaz de discernir qual o significado de um bloco ou de uma linha em um diagrama de blocos e linhas e, também, ser capaz de checar se toda a descrição é consistente, ou seja, se as partes adequam-se apropriadamente. Além disso, deve-se ter uma teoria coerente que permita raciocinar sobre as propriedades do sistema em um alto-nível de abstração.

Muito dos benefícios que um ambiente de desenvolvimento de sistemas provê para o projetista é derivado da capacidade que ele possui em permitir a análise e a avaliação do projeto. Análises efetivas possibilitam que o projetista explore as propriedades esperadas de um sistema, sinalize potenciais problemas, e ajude a determinar se o seu projeto será capaz de satisfazer os requisitos. Essa capacidade possibilita a avaliação das opções de projeto nos estágios iniciais do ciclo de desenvolvimento do sistema quando múltiplas alternativas de implementação podem ser exploradas relativamente a um custo desprezível.

Para verificar se um projeto está correto podem ser utilizadas técnicas de simulação e de análise. A simulação utiliza uma representação executável do sistema, permitindo que o projetista avalie o seu comportamento. A análise envolve a verificação das propriedades de alto-nível descritas para o sistema, avaliando a coerência das mesmas.

1.5.2 A Linguagem LACCES

A linguagem LACCES é usada para capturar tanto o conhecimento sobre o projeto de arquiteturas de sistemas embutidos quanto para especificar uma arquitetura. Um maior detalhamento da linguagem LACCES será provido nos capítulos seguintes.

A linguagem provê construções para representar duas classes fundamentais do conhecimento relacionadas à arquitetura de um sistema: o vocabulário e as restrições.

- O *vocabulário de projeto* é a forma mais básica do conhecimento do projeto que pode ser capturada através de LACCES, e possivelmente a mais valiosa. Os blocos básicos do sistema são especificados através do vocabulário, o qual descreve a seleção de *componentes*, *conectores* e *interfaces* que são utilizados no projeto do sistema. Os *componentes* representam elementos computacionais. Existem três tipos de componentes: software, hardware e genéricos. Os *conectores* são elementos que permitem a comunicação entre dois componentes. A *interface* define como o

componente e o conector interagem com o meio externo. Como um exemplo, o vocabulário disponível para o projeto de sistemas embutidos inclui componentes como memória e microcontrolador, e conectores como o SPI e I2C. A Figura 5 ilustra as entidades principais de LACCES.

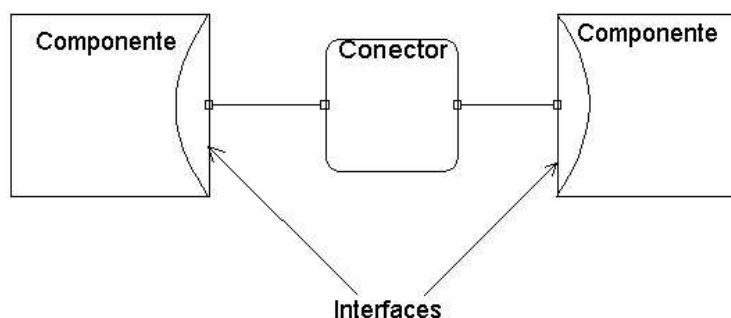


FIGURA 5 –Entidades principais de LACCES.

- As *restrições do projeto* especificam um conjunto de predicados que são capturados pelos projetistas através da utilização de uma metodologia *bottom-up* e/ou *top-down*. Esses predicados são classificados em 4 domínios: comportamental, temporal, estrutural e físico. Esses domínios estão de acordo com a separação existente entre os conceitos de funcionalidade, estrutura e tempo. O domínio físico foi adicionado para permitir a representação de restrições físicas impostas ao projeto. A Figura 6 ilustra esses 4 domínios como eixos de um grafo. Entre cada eixo do grafo existe uma ligação, que realiza o mapeamento entre os domínios. Por exemplo, uma funcionalidade é executada dentro de um intervalo de tempo, a qual possui uma estrutura que a implementa. Essa estrutura tem a sua realização física em um *chip*.

Além de prover construções para capturar o conhecimento abstrato do projeto, LACCES permite que o projetista avalie o comportamento do sistema através da simulação da arquitetura. As restrições de projeto também podem ser validadas através da linguagem.

A habilidade de permitir a descrição da arquitetura de sistemas embutidos constituídos de elementos de hardware e software, a simulação do comportamento da arquitetura e a verificação da consistência entre as restrições de projeto, fazem de LACCES uma linguagem única entre tantas existentes para o desenvolvimento de tais tipos de sistemas.

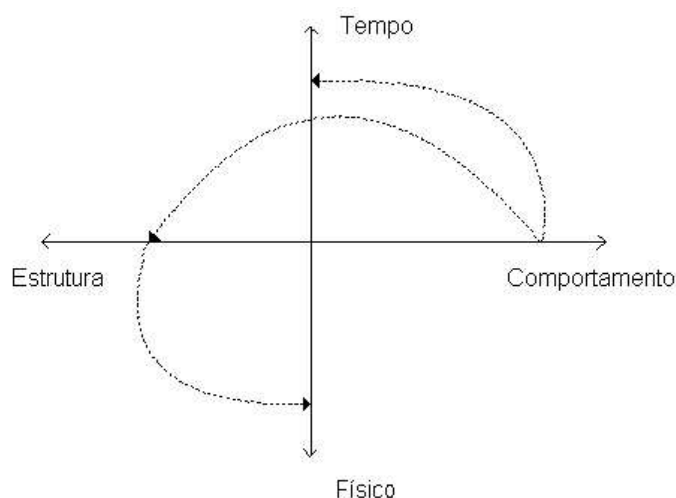


FIGURA 6 – Os 4 domínios de LACCES. [49]

1.6 Organização deste Documento

O trabalho de definição da notação e construção da ferramenta para descrição da arquitetura de sistemas embutidos seguiu uma seqüência bem definida de atividades, que foi utilizada como base para a estruturação dos capítulos deste documento.

A primeira atividade consistiu no estudo de conceitos relacionados à área de projeto de sistemas embutidos, envolvendo aspectos teóricos e diretrizes para representações de sistemas compostos de elementos de hardware e de software. O resultado desse estudo é descrito no **Capítulo 2**, e foi utilizado como fundamentação para todo o restante do trabalho.

Em seguida, foi feita a seleção das primitivas que irão compor a notação. Como veremos, essas primitivas são entidades atômicas que permitem representar um conjunto amplo de sistemas embutidos. O **Capítulo 3** detalha essas primitivas.

Uma vez especificado as primitivas, iniciou-se a construção da notação. A sintaxe e a semântica da linguagem foram definidas de acordo com as primitivas utilizadas. A descrição completa da sintaxe da notação encontra-se no **Capítulo 4**. O **Capítulo 5** apresenta a definição formal de cada construção.

Uma vez realizada a análise sintática e semântica do sistema descrito, gera-se uma representação interna do mesmo. A descrição de cada entidade desta representação é apresentada no **Capítulo 6**.

O **Capítulo 7** descreve as ferramentas auxiliares utilizadas no ambiente LACCES. Essas ferramentas permitem a validação funcional e não funcional da arquitetura especificada.

O **Capítulo 8** apresenta dois estudos de caso, os quais são utilizados para validar a linguagem e as ferramentas do ambiente LACCES.

Finalmente, o **Capítulo 9** apresenta as contribuições e conclusões observadas ao longo do trabalho, além de apontar sugestões de melhorias e novos trabalhos que podem ser desenvolvidos com base nos resultados aqui obtidos.

O documento vem acompanhado ainda de três apêndices, com informações que complementam alguns aspectos apontados ao longo do texto. O **Apêndice A.1** apresenta a listagem completa da sintaxe da linguagem descrita no **Capítulo 4**. Neste mesmo capítulo são apresentados alguns elementos auxiliares utilizados na construção da linguagem, os quais foram detalhados no **Apêndice A.2**. Por último, o **Apêndice A.3** apresenta a semântica de cada predicado utilizado na notação, os quais foram descritos no **Capítulo 3**.

Capítulo 2

Trabalhos Relacionados

Este capítulo apresenta um conjunto de trabalhos relacionados que tratam de problemas similares. Eles são divididos em cinco áreas principais. A primeira e a mais influente dessas áreas compreende linguagens, ferramentas e ambientes que permitem a descrição da arquitetura de software e de hardware. A segunda área compreende ambientes para o projeto concorrente de elementos de hardware e software. Essa área inclui ambientes para modelagem, simulação e prototipagem de sistemas heterogêneos. A terceira área compreende notações para especificação de projetos. A quarta área descreve uma representação interna capaz de capturar as propriedades de um sistema descrito em alto nível. Finalmente, a quinta área apresenta modelos para o projeto do software embutido.

2.1 Linguagens, Ferramentas e Ambientes de Descrição de Arquitetura de Software e de Hardware

A Arquitetura de software tem sido um campo de pesquisa muito ativo em Engenharia de Software [21, 51]. O seu objetivo é prover uma maneira formal de descrição e análise de sistemas de software de grande porte. De maneira abstrata, arquiteturas de software envolvem a descrição de elementos, padrões que guiam a composição, e restrições criadas sobre esses padrões.

Numerosas linguagens de descrição de arquitetura (LDAs) foram criadas com o objetivo de formalizar a descrição da estrutura e do comportamento de sistemas de software no nível de abstração de arquitetura. A maioria dessas LDAs oferecem um conjunto de ferramentas que permitem o projeto e a análise da arquitetura de sistemas de software. Exemplos, incluem: Aesop, Acme, UniCon, Rapide e Wright.

Entretanto, nenhuma dessas linguagens oferece o suporte para a validação das funcionalidades dos elementos dentro das restrições estruturais e temporais impostas na descrição do sistema. Além disso, a maioria dessas linguagens não permite a descrição de sistemas

heterogêneos, que incorporam elementos do domínio do hardware.

As arquiteturas de software são descritas através de um conjunto de sete entidades fundamentais. A notação utilizada para representar essas entidades pode variar de uma LDA para outra ou às vezes pode até nem existir. Entretanto, o conceito existe mesmo que implicitamente nas construções da linguagem.

- **Componentes** representam elementos computacionais e de armazenamento de dados em um sistema. Intuitivamente, correspondem aos blocos nos diagramas de blocos e linhas. Típicos exemplos de componentes são servidores, clientes, objetos e bancos de dados.
- **Conectores** representam a interação entre os componentes. Os conectores intermediam a comunicação e a sincronização entre os componentes. Informalmente, eles provêm a “cola” para o projeto de arquiteturas e, intuitivamente, eles correspondem as linhas nos diagramas de blocos e linhas. Exemplos incluem, chamada a procedimentos, protocolo cliente-servidor, ou uma ligação SQL entre uma base de dados e a aplicação.
- **Portas** representam a interface de um componente. Cada porta define um ponto de interação entre o componente e o seu ambiente.
- **Regras de comportamento (roles)** representam a interface de um conector. Cada regra define os participantes de uma interação.
- **Representações hierárquicas** de componentes e conectores.
- **Topologia** representa as conexões entre os componentes e os conectores.
- **Ligação (binding)** representa o mapeamento entre o sistema interno e a interface externa de um componente ou conector hierárquico.

A Figura 7 ilustra as entidades fundamentais na descrição de uma arquitetura.

Geralmente, as arquiteturas de hardware são definidas em RTL (*Register Transfer Level*), o que confunde o projeto da arquitetura com a sua implementação final. Existe uma variedade de linguagens que permitem a descrição nesse nível como, por exemplo, VHDL e Verilog HDL. A arquitetura de hardware pode também ser especificada em termos de ISA (*Instruction Set Architecture*), que define a arquitetura de computadores ou UCP (Unidade Central de Processamento).

Para o projeto de hardware, existem algumas ferramentas que permitem a descrição de arquiteturas de computadores. Entre elas, destaca-se a ferramenta denominada MICON.

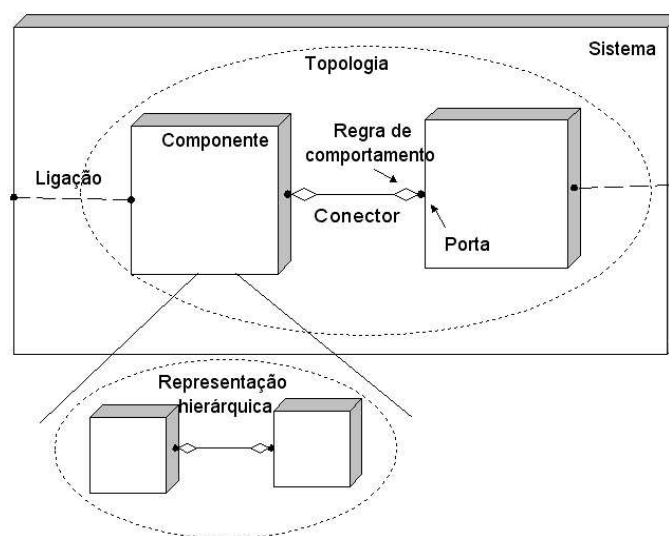


FIGURA 7 – Representação da arquitetura de um sistema.

2.1.1 Aesop

O sistema Aesop [19] é um ambiente de projeto de arquitetura de software genérico e configurável, que pode ser otimizado para uso específico de estilos de arquitetura. Estilos são padrões e idiomas recorrentes utilizados para descrever uma arquitetura de software específica como, por exemplo, arquitetura cliente-servidor, *pipeline* e de tempo real.

O modelo de definição de um estilo é baseado no princípio de subtipagem: um vocabulário específico de um estilo é criado através da subtipagem das classes básicas da linguagem (*Component*, *Connector*, *Port*, *Role*, *Representation* e *Binding*) ou de um dos seus subtipos (*Pipe* e *Filter*). Essas novas classes do estilo definem, através dos seus métodos, as restrições de configuração e o modo de visualização. Adicionalmente, as classes podem identificar uma coleção de ferramentas externas. Algumas delas podem vir a ser implementadas com o objetivo de realizar análise da arquitetura, enquanto outras são apenas referências para ferramentas externas de desenvolvimento de software. Uma vez que essas classes tenham sido escritas, o projetista gera um ambiente otimizado através da compilação e ligação de suas classes específicas de estilo com a infra-estrutura genérica do Aesop.

Cada um dos estilos definidos em Aesop limita o escopo do projeto, permitindo análise específicas para a arquitetura desenvolvida naquele estilo. Por exemplo, é possível analisar sistemas construídos usando o estilo *pipe-filter* por escalabilidade, desempenho, latência, e se há *deadlock*. Tais análises não são possíveis para uma arquitetura arbitrária ou construída em um outro estilo.

2.1.2 Acme

Acme [20] é uma linguagem genérica e extensível, que tem como objetivos a descrição e o intercâmbio de uma arquitetura de software. Acme possibilita o intercâmbio de descrições entre uma grande variedade de projetos de arquitetura e ferramentas de análise. A idéia de desenvolvimento de uma nova linguagem surgiu devido à proliferação de LDAs que operam isoladamente, tornando difícil o compartilhamento de recursos entre elas. Além disso, existem muitas características em comum entre as LDAs que são reimplementadas a cada novo projeto de uma linguagem. Exemplos incluem: ferramentas gráficas para visualização e manipulação de estruturas da arquitetura, e certos tipos de análise independentes do domínio, tais como, verificar se existem ciclos, ou se algum conector está incorreto.

Acme é baseada na premissa que existe pontos em comuns entre as LDAs o que possibilita o compartilhamento de informações. Esta linguagem tenta englobar essas similaridades e permite, também, a incorporação de informações específicas de cada LDA.

Para englobar essa variedade de informação, Acme permite que seja armazenada uma lista de propriedades juntamente com a estrutura da arquitetura. Do ponto de vista da linguagem, as propriedades são valores não interpretados. Elas somente tornam-se úteis quando uma ferramenta faz o seu uso para análise, tradução ou manipulação. Esse método possibilita que um subconjunto de ferramentas possa compartilhar dados que são compreendidos por elas, enquanto desconsideram a presença de outras informações que não estão contidas nos seus vocabulários.

2.1.3 UniCon

UniCon [50] consiste de uma linguagem de descrição e de um conjunto de ferramentas que permitem a geração e análise de sistemas de software a partir da descrição da arquitetura. O seu objetivo é produzir uma descrição de uma arquitetura compilável em um código executável.

A linguagem UniCon possui uma sintaxe que tem um conjunto de componentes, conectores e atributos já bem definidos e conhecidos. Componentes podem ser primitivos ou compostos, enquanto que os conectores só podem ser primitivos. Exemplos de componentes primitivos incluem *Filter*, *Proccess* e *SharedData*. Um atributo comum a todos os componentes é o nome do processador em que eles serão executados. Exemplos de conectores primitivos incluem *Unix Pipes*, *RPCs (Remote Procedure Calls)* e *SQL Queries*. Além dos tipos primitivos dos componentes e conectores serem pré-definidos pela linguagem, as interfaces desses elementos também são pré-definidas. As interfaces (*players*) do componente *Filter* são *StreamIn* e *StreamOut*. As interfaces (*roles*) do conector *Pipe* são *Source* e *Sink*. Uma interface *Source* de um conector só aceita a associação com uma interface do tipo *StreamOut* do *Filter*.

Devido a padronização dos componentes, conectores, atributos, interfaces e associações a linguagem é capaz de gerar código executável a partir da descrição de uma

arquitetura. Como resultado, UniCon consegue aumentar dramaticamente o nível de abstração no qual os sistemas são construídos.

2.1.4 Rapide

Rapide [34, 35] é uma linguagem de simulação concorrente baseada em eventos. Essa linguagem é utilizada para definir e simular o comportamento da arquitetura de sistemas concorrentes e distribuídos, constituídos de elementos de hardware e de software.

Rapide permite simular o comportamento de um sistema modelado, dado um *stream* de entrada. O comportamento resultante do sistema é representado como um conjunto parcialmente ordenado (Poset), que define a seqüência de eventos que podem ocorrer como resultado das entradas fornecidas ao sistema. Rapide, também, é capaz de modelar arquiteturas de sistemas dinâmicos, no qual o número de conexões e componentes variam quando o sistema é executado.

Uma arquitetura Rapide consiste de interfaces, conectores e restrições. As interfaces especificam o comportamento dos componentes do sistema. Os conectores definem a comunicação entre os componentes. As restrições restringem o comportamento das interfaces e das conexões.

Diferentemente das outras LDAs, Rapide não possui um elemento de projeto denominado componente. A interface provê a definição abstrata do comportamento externo visível do componente, isto é, do comportamento que é visível e que pode ser observado pela arquitetura que contém esses componentes. O projetista somente precisa especificar a interface, que é constituída de eventos que o componente pode observar ou gerar, transições de estados, restrições de comportamento e funções que ele provê e requer dos outros elementos da arquitetura. A interface opera da seguinte maneira: ela observa os eventos da arquitetura, reage executando as suas regras de transição e gera outros eventos que são enviados as interfaces dos outros componentes.

A comunicação entre as interfaces dos componentes pode ser síncrona, através da chamada de métodos, ou assíncrona, através da geração de eventos. O padrão de comunicação adotado por Rapide não possui verificação de compatibilidade entre interfaces de componentes e conectores.

2.1.5 Wright

Wright [5, 6] é uma linguagem utilizada para descrever e analisar formalmente conectores. O método usado na linguagem Wright descreve as regras de comportamentos como protocolos que interagem. Uma álgebra de processo é usada para modelar os traços de eventos de comunicação. Especificamente, é usado um subconjunto de CSP (*Communicating Sequential*

Processes) [26] para definir os protocolos das regras de comportamento, portas e da ligação. CSP tem como elementos base processos e eventos. Um processo é uma entidade lógica usada para representar componentes e conectores de uma arquitetura de software. Eventos podem ser primitivos ou podem possuir dados associados.

O uso de protocolos em Wright difere do seu uso tradicional de duas maneiras: primeiramente, os protocolos dos conectores especificam um conjunto de obrigações, ao invés de especificar um algoritmo que deve ser seguido pelos participantes. Isso permite que ocorram situações em que os usuários atuais dos protocolos (portas), possuam comportamentos relativamente diferentes daqueles especificados pelas classes dos conectores (via suas regras de comportamento), possibilitando, assim, o reuso dos conectores. Em segundo lugar, o método usado na linguagem Wright provê uma maneira específica de estruturar a descrição dos protocolos dos conectores, isto é, separando-os em regras de comportamento e ligações.

Descrevendo um sistema em Wright, torna-se possível realizar análise e checagem de compatibilidade entre os conectores, quando usados em um contexto particular. Ao contrário de Rapide, Wright permite a análise estática da arquitetura ao invés de uma simulação. Uma análise possível de ser realizada é a verificação se as partes que interagem no sistema estão livres de *deadlock*. Uma outra análise é a checagem de compatibilidade, que verifica se os processos que descrevem as portas sempre agem de uma maneira correspondente ao modo como os processos que descrevem as regras de comportamento são capazes de agir. É também possível realizar checagem de compatibilidade automaticamente. Isso é feito por uma ferramenta comercial projetada para checar as condições de refinamento para processos finitos em CSP.

Em particular, Wright não lida com restrições globais da arquitetura, tais como, sincronização global, escalonamento e análise global. Como também, não permite a construção de conectores complexos a partir de conectores mais simples.

2.1.6 MICON

MICON [8] é um sistema que integra uma coleção de programas que sintetizam sistemas de computadores a partir de uma especificação em alto nível. Os projetistas podem utilizar os seguintes subsistemas no momento de desenvolvimento do projeto: processadores, memórias, periféricos, interfaces de barramento e circuitos lógicos de apoio (osciladores, geradores de estados de espera, etc). Todos os tipos de subsistemas disponíveis em MICON estão armazenados em um banco de dados.

Para o projeto de um computador, o usuário deve especificar qual a funcionalidade requerida. Por exemplo, o usuário pode especificar o tipo do micro-processador, a quantidade e o tipo de memória, e o número e tipo de dispositivos de entrada e saída requeridos. O sistema utiliza o seu conhecimento sobre os componentes e micro-processadores para gerar um projeto que satisfaça os requisitos informados. Novos elementos são inseridos através de uma ferramenta

especial. Essa ferramenta especifica regras que detalham como cada dispositivo pode ser utilizado no projeto.

2.2 Ambientes de Projeto Concorrente de Hardware e Software

As metodologias de projetos concorrentes de hardware e software têm-se tornado comuns na literatura [1, 52,55]. Combinar o projeto desses domínios em uma única metodologia ou ferramenta possui várias vantagens. Uma delas é que ao incluir mais informações no projeto do sistema acelera o processo de desenvolvimento. A outra é que ao projetar, em uma única metodologia, sistemas constituídos de hardware e software, possibilita a verificação dinâmica dos compromissos entre desempenho, custo, etc.

As metodologias de projeto concorrente de hardware e software podem ser caracterizadas pelas atividades que integram os componentes desses domínios. A Figura 8 ilustra como as várias tarefas de projeto estão relacionadas. Essas atividades são caracterizadas como:

- **Co-Simulação de Hw/Sw** [27]: simulação simultânea de componentes de software e de hardware. Esse tipo de simulação requer um ambiente que compreenda a semântica dos componentes de hardware e de software e como uma ação em um domínio afeta o outro. O objetivo da co-simulação é avaliar o desempenho do sistema ou verificar se a sua funcionalidade está de acordo com o que foi especificado.

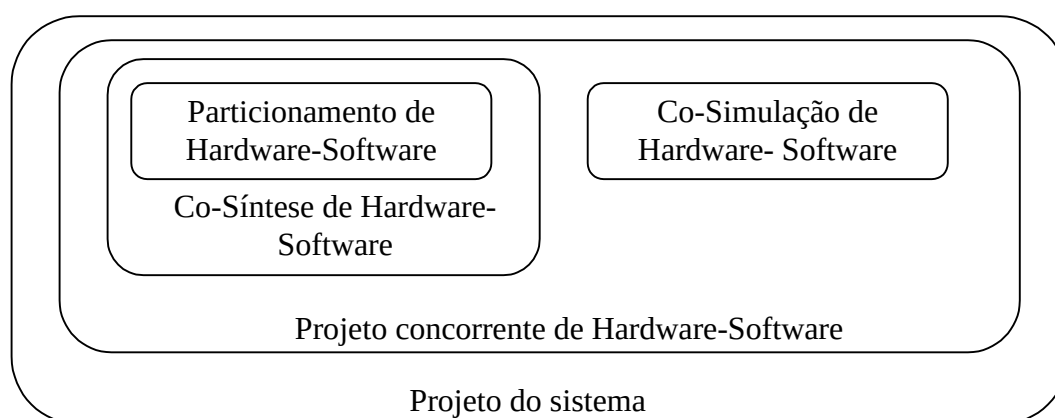


FIGURA 8 – Projeto de um sistema heterogêneo.

- **Co-Síntese de Hw/Sw [24]:** compreende a síntese concorrente de componentes de hardware e de software. As ferramentas que realizam esse tipo de síntese devem entender como decisões de projeto em um domínio afetam as opções disponíveis no outro domínio. É também necessário possuir um discernimento de como o custo total do sistema e o seu desempenho são afetados pela escolha de uma implementação em hardware ou em software.
- **Particionamento de Hw/Sw:** A síntese concorrente de sistemas de hardware e software inclui o particionamento desses componentes. Isso significa saber se a metodologia de projeto permite escolher entre usar hardware ou software para implementar uma determinada funcionalidade. Funcionalidades que possuem um grande impacto no desempenho de um sistema são preferencialmente implementadas em hardware. Por outro lado, se a funcionalidade será modificada, uma implementação em software é preferida.

2.2.1 Ptolemy

Ptolemy [31] consiste de um ambiente para simulação e prototipagem rápida de sistemas heterogêneos. Esse ambiente utiliza tecnologia orientada a objetos para modelar e integrar cada subsistema de uma maneira natural e eficiente.

Uma abstração básica em Ptolemy é o domínio, que implementa um modelo de computação apropriado para cada tipo particular de sistema. Exemplos de domínios incluem: síncrono, de eventos discretos e de máquinas de estado finito, entre outros. Os domínios podem ser mesclados para a implementação de um sistema.

A infraestrutura de software corrente para o projeto Ptolemy chama-se Ptolemy II [11]. Ela consiste de um conjunto de pacotes de Java que permitem o projeto e a modelagem de sistemas concorrentes e heterogêneos.

Ptolemy II é baseado em atores, que são classes em java. Atores encapsulam uma *thread* de controle e possuem uma interface para interação com outros atores. A interface inclui portas, que representam pontos de interação, e parâmetros, que são utilizados para configurar o modo de operação de um ator. Os atores podem ser polimórficos, operando em mais de um domínio.

Em um ambiente baseado em atores a comunicação ocorre através de troca de mensagens através de canais de comunicação.

2.2.2 Lycos

Lycos [36] é um ambiente de co-síntese que pode ser usado para o particionamento de

uma aplicação de hardware e software em uma arquitetura constituída de uma CPU e de um componente de hardware.

Lycos não permite a representação abstrata do sistema. Dada uma especificação em um subconjunto de VHDL ou C [47], Lycos traduz em um modelo de computação independente da aplicação denominado Quenya [33]. Esse modelo é baseado em grafos de fluxos de dados e controle (CDFGs).

Lycos possibilita a escolha entre os diferentes modelos e algoritmos de particionamento, um deles é o algoritmo chamado PACE. Além disso, esse ambiente permite a exploração do espaço do projeto, com o objetivo de se obter o melhor particionamento possível para cada arquitetura alvo em termos de tempo de execução, área, custo, etc.

2.2.3 POLIS

POLIS [13] é um ambiente para o projeto concorrente de hardware e software. A especificação do sistema em POLIS é feita em ESTEREL [10], que é uma linguagem de especificação para sistemas reativos de tempo real. Assim como Lycos, POLIS também não permite a representação do sistema em um alto-nível de abstração.

A especificação em ESTEREL é traduzida para uma representação interna chamada CFSMs (*Codesign Finite State Machines*) [14]. Essa representação consiste de máquinas de estado finito que se comunicam através de áreas comuns de armazenamento, que são implementadas como registradores em hardware e como posições de memória em software.

Após a tradução, as CFSMs são atribuídas a módulos de software ou de hardware. Essa divisão é feita pelo usuário. Porém, POLIS provê ferramentas que permitem estimar o custo de implementação em cada um desses domínios.

Os elementos de hardware e software são gerados automaticamente a partir da divisão escolhida pelo usuário. Os elementos de hardware são sintetizados em circuitos síncronos e os de software são sintetizados em uma coleção de procedimentos, cada um implementando uma CFSMs.

2.3 Linguagens de Especificação de Sistemas

Linguagens de especificação permitem capturar o comportamento do sistema a ser desenvolvido. Entre essas linguagens destacam-se:

- Linguagens para a especificação de projetos descritos em um alto nível de

abstração: Uma solução conhecida em ciência da computação para lidar com a complexidade de sistemas é mover os conceitos para um nível de abstração mais alto. A partir desse nível, a metodologia de projeto consiste de um refinamento sucessivo até a sua implementação final. Devido a crescente complexidade dos sistemas, a especificação em alto nível tem-se tornado uma questão crítica. Várias metodologias foram desenvolvidas para lidar com o projeto em um alto nível de abstração. Exemplos incluem: SDL, SystemC, SpecC e Rosetta.

- Linguagens para a especificação de sistemas digitais: as linguagens dominantes são VHDL e Verilog HDL.

2.3.1 SDL

SDL (*Specification and Description Language*) [30] é uma linguagem de programação de alto nível adequada para o desenvolvimento de sistemas reativos. Padronizada pela ITU (*International Telecommunication Union*) através da recomendação Z.100, combina uma sintaxe formal com uma representação gráfica intuitiva, possuindo um alto nível de abstração em sua notação. Inicialmente voltada para o projeto de sistemas de telecomunicações e protocolos de comunicação, SDL evoluiu ao longo dos anos e atualmente é usada para o desenvolvimento de sistemas em diversas outras áreas.

Sistemas descritos em SDL consistem de muitos agentes (processos) executando concorrentemente, e que se comunicam assincronamente entre si e com o ambiente através de sinais discretos (transmitidos por canais). Cada agente é descrito por uma máquina de estados finitos estendida (EFSM). As máquinas de estados finitos são nomeadas estendidas desde que variáveis e temporizadores também possam ser definidos.

A precisão e a formalidade de SDL provêm a possibilidade de geração de código para linguagens de baixo nível, tais como C/C++ e Java. Além disso, é possível construir ferramentas para a simulação de sistemas em SDL e para a validação de características formais, como detecção de *deadlocks*.

Conseqüentemente, SDL é uma linguagem adequada para a especificação do comportamento independente de implementação e também para a descrição do comportamento realmente implementado. Entretanto, SDL não permite a representação de propriedades não funcionais dos elementos de projeto, tais como estrutura lógica e física.

2.3.2 SystemC

SystemC [7, 23, 40, 41] é uma linguagem que permite a especificação e o projeto de sistemas de hardware-software em vários níveis de abstração. Ela também permite a criação de especificações executáveis do projeto.

A linguagem SystemC foi influenciada por VHDL e Verilog HDL, sendo constituída de um subconjunto de C++ [53, 54] com algumas bibliotecas adicionais. Ela provê um conjunto de construções que não estão presentes em C++ e que são necessárias para modelar uma arquitetura de um sistema: temporização em hardware, concorrência e comportamento reativo.

As estruturas principais de SystemC são módulos, portas, interfaces, canais, processos e eventos.

Módulos representam os blocos básicos da linguagem. Eles permitem que sistemas complexos sejam descritos hierarquicamente e que a representação interna dos dados e dos algoritmos seja encapsulada. Um módulo contém portas, processos, dados internos, canais e hierarquicamente outros módulos.

As portas permitem a comunicação do módulo com seu meio externo. Elas acessam o canal através de interfaces, pode-se dizer que a porta está associada ao canal através de uma interface. Uma interface especifica um conjunto de operações que o canal provê. Enquanto as interfaces e portas descrevem quais funções são adequadas para a comunicação, os canais definem como essas funções são implementadas.

Em SystemC, os processos são as unidades básicas de funcionalidade. Eles provêem o mecanismo necessário para simular um comportamento concorrente. Os processos não podem invocar outros processos. Portanto, uma modelagem hierárquica de processos não é permitida.

Um evento é um objeto que determina se a execução de um processo deve ser acionada ou reiniciada e quando isso deve ocorrer. Em outras palavras, um evento é usado para representar uma condição que deve acontecer durante o curso da simulação e para controlar o acionamento dos processos.

Embora SystemC permita a especificação de sistemas heterogêneos em vários níveis de abstração, não é capaz de representar aspectos específicos do projeto, como temporização mais detalhada, restrições temporais e físicas. Por exemplo, capacidade de armazenamento de uma memória ou a energia consumida.

2.3.3 SpecC

SpecC [16, 22] é uma linguagem de especificação no nível de sistema, sendo uma extensão de ANSI-C. Além das construções de ANSI-C, SpecC inclui extensões para o projeto de hardware. Como em SystemC, SpecC permite a execução e o refinamento do sistema especificado.

Em SpecC, como nas outras linguagens de projeto no nível de sistema, a computação e a comunicação são claramente separadas. A computação é encapsulada em comportamentos e a comunicação é encapsulada em canais.

Em SpecC um programa é uma coleção de classes. Existem três tipos de classes:

comportamento, canais e interfaces. Ao contrário de SystemC, o comportamento em SpecC pode ser composto hierarquicamente. Além disso, SpecC permite especificar a forma do escalonamento estático do seu comportamento. O projetista especifica se o comportamento descrito em uma classe será executado de forma paralela, seqüencial, *pipeline* ou como uma máquina de estados.

A interface declara os métodos que são implementados em um canal. A transferência de dados se dá através de canais ou variáveis. Cada comportamento possui um conjunto de portas que se conectam às outras portas de outro comportamento.

Em SpecC, o comportamento representa ao mesmo tempo a estrutura e o comportamento especificado. Em SystemC, há uma clara separação entre esses conceitos. A estrutura é modelada através de módulos, enquanto o comportamento é modelado usando processos. SpecC, também, não possui o mecanismo de sensibilidade estática, que em SystemC especifica quais eventos disparam um processo.

2.3.4 Rosetta

Rosetta [3, 4] é uma linguagem de especificação no nível de sistema. Essa linguagem permite a especificação de sistemas heterogêneos através da construção de arquiteturas que envolvem componentes e conectores e da especificação de múltiplas visões para um mesmo componente. Através de Rosetta é possível definir as restrições do projeto de um sistema e não o seu comportamento. Logo, uma especificação em Rosetta não pode ser executada. A finalidade principal dessa linguagem é poder analisar matematicamente o desempenho do sistema projetado.

A unidade básica de especificação em Rosetta é a faceta. Uma faceta modela diferentes visões de um sistema ou de um componente. Cada faceta é uma coleção parametrizada de declarações e de definições especificadas usando uma teoria de um domínio. Como em Ptolemy, cada domínio representa um modelo de computação, possuindo um vocabulário para a especificação de propriedades particulares do sistema. Exemplos de domínios incluem o *lógico*, que provê um conjunto básico de expressões matemáticas, tipos e operações; e *baseado em estado*, que descreve o comportamento de um componente ao definir as propriedades no estado corrente e no estado seguinte, a operação $x@s$ (valor de x no estado s), por exemplo, é válida nesse domínio.

Quando um componente é especificado em Rosetta, o projetista escolhe um domínio apropriado para construir o seu modelo. Em seguida, ele descreve a faceta correspondente, adicionando declarações e termos que utilizam o modelo de computação pré-definido pelo domínio. Um componente pode ser constituído de um conjunto de facetas que interagem entre si. As interações definem quando teoremas de um domínio implicam em teoremas em outros domínios.

2.3.5 VHDL

VHDL (*VHSIC Hardware Description Language*) é uma linguagem que permite a especificação e a simulação de projetos digitais. Baseada em ADA, VHDL também possui modularidade, herança e sobrecarga de operadores. Programadores podem definir bibliotecas de estrutura de dados e operadores e seletivamente importá-los ou exportá-los.

O tipo básico em VHDL é o bit. Algumas bibliotecas estendem esse conjunto, permitindo incluir tipos compostos como byte, inteiro, e palavras de tamanho definido pelo usuário. Um programa em VHDL define uma entidade e um ou mais corpos de arquitetura. A entidade especifica as portas de entrada/saída que o comportamento utiliza para comunicar com o mundo exterior. O corpo da arquitetura pode ser descrito em uma linguagem estrutural, comportamental ou de fluxo de dados.

A comunicação entre os processos é realizada pela troca de valores através dos sinais, que lembram variáveis globais em linguagens de programação, ou pode ser abstraída pela definição de funções de enviar e receber que podem ser simuladas, mas para elas não existem nenhuma primitiva de sincronização. Em outras palavras, fica a cargo do projetista decidir e descrever esse mecanismo. O sincronismo existe em dois níveis, no nível de processo, pelo uso de uma lista de sensibilidade que lista as variáveis e os sinais que disparam a execução do processo, e no nível de comando pelo uso do comando *wait*.

2.3.6 VERILOG HDL

Verilog foi influenciada pela linguagem C e permite a especificação e simulação de sistemas digitais. Ao contrário de VHDL, a modularização não é bem definida e a sobrecarga de operadores também não é permitida. A maior limitação de Verilog é a falta de uma sintaxe fortemente tipada. Em contraste com outras linguagens, ela oferece uma primitiva de tratamento de exceção através do comando *disable* o qual desabilita o bloco corrente e transfere o controle para o próximo comando que segue o bloco. Similar a VHDL, a comunicação pode ser simulada usando sinais globais e comandos explícitos de baixo nível, como por exemplo,

```
@(negedge) clock 10 q = d;
```

que significa que dez unidade de tempo após a borda negativa do clock, o valor de d será atribuído a q.

2.3.7 Classificando as Linguagens de Especificação

O conjunto de linguagens de especificação de hardware não possui características importantes de nível de sistema. Porém, as outras linguagens de especificação incorporam características importantes. Entretanto, elas não possuem construções que permitam a

especificação de propriedades “verdadeiras” do nível de sistema, tais como, requisitos não funcionais, descrições estruturais e temporização mais detalhada.

A Tabela 1 apresenta a classificação dessas linguagens de acordo com a sua execução e a possibilidade de síntese.

SystemC e SpecC são sintetizáveis utilizando uma metodologia que mapeia a especificação para uma linguagem de síntese, como Verilog HDL ou VHDL.

Linguagem	Sintetizável	Executável
SDL	Não	Sim
SystemC	Metodologia	Sim
SpecC	Metodologia	Sim
Rosetta	Não	Não
VHDL	Sim	Sim
Verilog HDL	Sim	Sim

TABELA 1 – Classificação das linguagens de especificação.

2.4 Representação Interna

Geralmente, para a representação interna de um sistema são utilizadas estruturas de dados hierárquicas, onde a funcionalidade, temporização e, às vezes, recursos são representados. A representação interna de LACCES é baseada em SDS.

2.4.1 SDS

SDS (*System Data Structure*) [49] é uma representação interna capaz de capturar os requisitos dos clientes e o conhecimento dos projetistas sobre o sistema especificado em uma estrutura de dados unificada, formal e compreensível. Essa estrutura de dados pode ser examinada por ferramentas de análise e síntese. Embora, a completeza de um projeto no nível de sistema é uma condição necessária para a síntese comportamental, SDS não garante tal propriedade, mas permite que sejam capturadas especificações parciais e eventualmente, quando o projetista prover mais detalhes, seja representado um projeto completo para ser sintetizado.

SDS permite inúmeras metodologias de projeto de sistema, incluindo a *top-down* e *bottom-up*. A metodologia *top-down* captura o comportamento dinâmico, ao refinar uma visão conceitual do sistema em termos de componentes mais detalhados. O comportamento dinâmico é representado em SDS através de dois domínios independentes: o de fluxo de processo e de fluxo de tempo.

A metodologia *bottom-up* captura a visão estrutural e estática de um sistema ao compor componentes com o objetivo de obter elementos que sejam mais complexos. Isso é

representado em SDS através dos domínios de estrutura lógica e de estrutura física.

SDS permite uma separação clara de conceitos nos estágios iniciais de desenvolvimento. Na verdade os conceitos tornam-se ortogonais na modelagem utilizada, na maneira que eles são independentes uns dos outros e podem ser relacionados uns com os outros através de ligações. Por exemplo, o comportamento temporal pode mudar sem alterar o comportamento funcional. Vários modelos não provêm esta separação.

Em [18] um modelo de gráfico em Y é utilizado para representar um projeto de sistema. Esse modelo utiliza três domínios distintos - estrutural, funcional e físico, que são representados por cada eixo do gráfico. O processo de síntese é representado como uma ligação entre os eixos. Entretanto, nesse modelo não existe uma representação clara para o tempo e a concorrência.

A representação interna de SDS é definida através de um modelo de informação que abstrai de implementações particulares e provê a base para um modelo geral. Esse modelo foi adaptado para ser utilizado na representação interna da arquitetura de um sistema embutido descrito em LACCES.

O objeto fundamental em SDS é o componente que representa uma entidade física. O componente é descrito em termos de quatro domínios e de um conjunto de ligações entre esses domínios. Uma ligação entre os domínios é uma representação formal explícita da descrição dos objetos nos diferentes domínios. Essa ligação é necessária, pois como os domínios são ortogonais, não existe uma relação implícita entre eles.

Cada domínio é composto de um conjunto de modelos, enlaces e referências. Os modelos representam objetos que possuem semântica relacionada a operações, os elementos de enlace representam objetos com semântica relacionada a conexões, e as referências são utilizadas para conectar esses dois elementos. As referências definem uma rede que utilizam pinos para representar a conexão. Pinos são objetos específicos associados aos modelos e aos elementos de enlace, representando suas entradas e saídas. A Figura 9 mostra o relacionamento entre os elementos de modelo e de enlace em cada domínio. As referências possuem tipos simples e nenhum relacionamento.

O domínio de fluxo de processo define os aspectos operacionais do projeto, capturando a funcionalidade das computações e as coordenações. Os objetos principais desse domínio são: **processos**, **canais** e **informação**.

Os **processos** representam a funcionalidade de um componente. Em SDS existem dois tipos de processos: algoritmo e hardware. Algoritmos representam processos com especificações funcionais e que não possuem implementação física. A especificação funcional é obtida usando uma metodologia *top-down*. Hardwares representam processos com uma implementação física. Eles são obtidos usando a metodologia *bottom-up*.

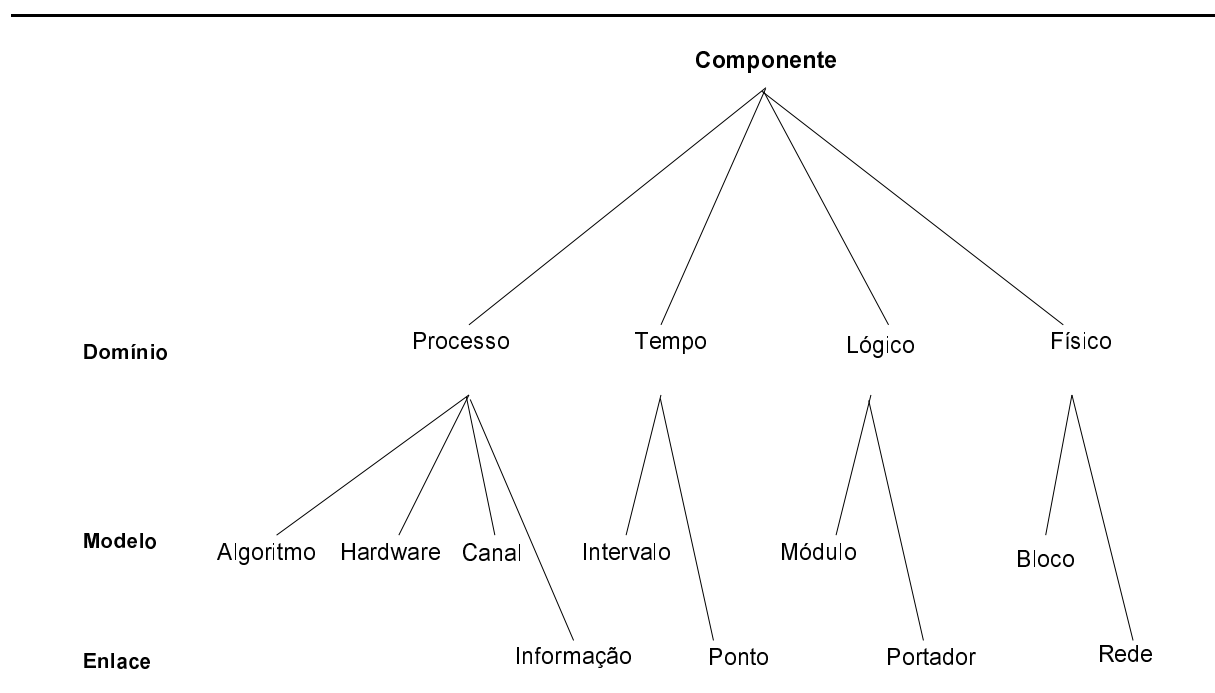


FIGURA 9 – Os domínios de SDS e os tipos de relacionamento entre os elementos.

Os **canais** capturam os mecanismos de coordenação a serem utilizados e, portanto, são ortogonais aos mecanismos de computação. Com essa separação, diferentes mecanismos de coordenação podem ser trocados sem modificar os de computação. Canais podem possuir uma especificação funcional (protocolo) e uma implementação física. Canais também podem possuir persistência, para a modelagem do armazenamento e da recuperação de informações. Por exemplo, uma memória e um banco de dados são modelados como canais persistentes.

A **informação** representa o dado ou o sinal de sincronização transferido entre dois processos interconectados por um canal.

O domínio do tempo define um ordenamento parcial entre operações e as informações trocadas entre os processos. Esse domínio é composto de dois componentes: **pontos** e **intervalos**.

Pontos possuem um tempo de duração infinitesimal e são considerados instantâneos. Pontos servem como delimitadores dos intervalos de tempo e, às vezes, podem possuir uma semântica especial. Tipos de pontos incluem: simples, OR (fluxo condicional) e AND (composição paralela).

Intervalos representam intervalos de tempo que podem possuir uma expressão definindo a sua duração. Os tipos de intervalos são: *Causal*, que representa um intervalo de tempo genérico; *Constraint*, que representa um intervalo com alguma restrição específica (por exemplo, maior ou igual a um determinado valor); *Delay*, que representa um intervalo de tempo obtido de um componente físico; e *Loop*, que representa um intervalo de um bloco de iteração (*loop*, *while*, *wait*).

O domínio da estrutura lógica captura a visão estrutural do sistema e denota uma unidade topológica. Um objeto nesse domínio não possui nenhum significado além da sua organização lógica especificada pelo projetista numa metodologia *top-down* ou imposta pelos requisitos ou restrições físicas em uma metodologia *bottom-up*. Esse domínio deve ser visto como um campo intermediário em que o comportamento e os componentes físicos são relacionados. Ele é composto de dois componentes: **módulos** e **portadores**.

Os **módulos** e os **portadores** representam unidades lógicas relacionadas com a sua implementação física, podendo não existir uma correspondência de um para um entre eles. Ambos podem ser hierárquicos, correspondendo a uma agregação de outros módulos e portadores. Entretanto, somente o módulo possui persistência, podendo ser usado para representar memórias ou registradores.

O domínio da estrutura física descreve as restrições físicas imposta pelos requisitos. A especificação do sistema pode conter uma informação física específica, dada por uma referência a um catálogo que lista um conjunto de componentes físicos, que podem ser utilizados para a sua implementação, ou por um conjunto de restrições físicas. Esse domínio representa atributos como tamanho, custo e peso. Ele é composto por dois componentes: **blocos** e **redes**.

Blocos representam componentes físicos. **Redes** representam conexões físicas como, por exemplo, cabos. Blocos e redes são considerados caixas pretas, e como tais, não possuem hierarquia. A representação interna do bloco pode ser descrita através de outra representação como DDS (*Design Data Structure*) [12]. Essa suposição limita claramente o que corresponde ao projeto de um sistema e o que corresponde ao projeto de um componente.

Cada componente em SDS é caracterizado por um conjunto de primitivas. Essas primitivas são utilizadas para capturar características importantes de sistemas de hardware/software. Entretanto, para cada tipo de sistema específico uma adaptação nesse conjunto de primitivas é necessária, como ocorre, por exemplo, no caso de sistemas embutidos ou sistemas de tempo real.

Para interligar esses quatro domínios existem dois tipos de relacionamentos: comportamentais e de implementação. Os relacionamentos comportamentais interligam os domínios do fluxo de processo, de tempo e de estrutura lógica. Os relacionamentos de implementação interligam os domínios de estrutura lógica e física.

A Figura 10 apresenta um exemplo do modelo de representação utilizado por SDS. T1 e T2 são dois processos interconectados através de um canal C1. O processo T1 envia uma mensagem para o canal C1 que a envia para o processo T2. As mensagens MSG1 e MSG2 são objetos de informação que possuem uma rede que as interconectam com os processos T1 e T2 e com o canal. Essa interconexão ocorre através dos pinos PT1, PC1, PC2 e PT2, respectivamente. Esse diagrama também representa os dois mapeamentos existentes entre os quatro domínios.

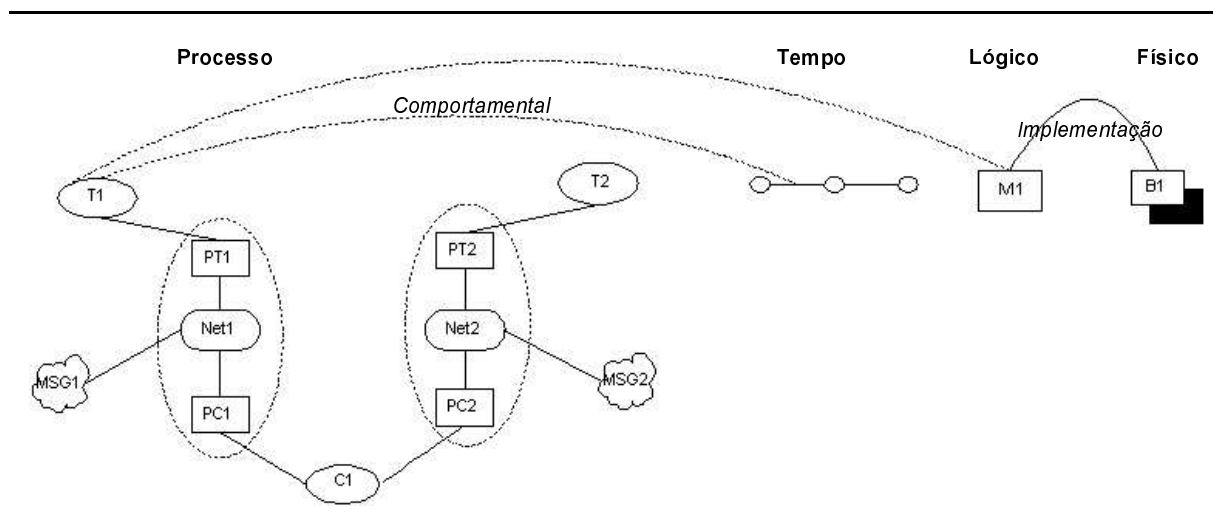


FIGURA 10 – Modelo de informação de SDS.

2.5 Métodos de Projeto de Sistemas de Software Embutido

Sistemas de arquitetura utilizam componentes, portas e conectores como elementos principais de representação. Entretanto, a maioria dessas metodologias não levam em consideração propriedades específicas de sistemas de software para dispositivos embutidos.

2.5.1 PECOS

PECOS (*PE*rvasive *C*omponent *S*ystem) [44, 58] é um modelo que provê uma tecnologia baseada em componentes destinada ao desenvolvimento de sistemas de software embutido. PECOS é subdividido em dois modelos: estrutural e de execução. O modelo estrutural define as entidades, suas características e propriedades. O modelo de execução define a semântica de execução dos componentes, ou seja, o sincronismo e a forma de execução.

As entidades principais desse modelo são componentes, portas e conectores. Existem três tipos de componentes em PECOS: passivos, ativos e eventos. Componentes passivos são componentes que não controlam a sua própria execução, são utilizados como parte do comportamento de outros componentes. Componentes ativos controlam a sua própria execução. Componentes de eventos são componentes acionados pelo acontecimento de um evento. Um componente composto é responsável por prover um escalonador que especifica a ordem que o seu comportamento e o de seus sub-componentes devam ser executados.

A semântica dos conectores é o de uma variável compartilhada. Uma porta de um

componente acessa uma variável que representa o conector. Em PECOS, não é possível representar estruturas mais complexas de comunicação como protocolos. Além disso, os conectores não possuem uma decomposição hierárquica.

Um conjunto de meta-informações pode ser associada a um componente através de propriedades, tais como o consumo de memória ou o seu tempo de execução. Essas informações são utilizadas por diferentes ferramentas de PECOS.

Os comportamentos dos componentes não são especificados na linguagem, eles são implementados diretamente em C++ ou Java. Portanto, um sistema especificado em PECOS é mapeado em uma hierarquia de classes na linguagem alvo. A hierarquia corresponde diretamente à especificação hierárquica dos componentes, ou seja, cada componente é mapeado em uma classe e cada conector é mapeado em uma variável compartilhada.

2.5.2 Koala

Koala [39] é um modelo utilizado para descrição de software embutido. Esse modelo é constituído de componentes e configurações.

Os componentes comunicam-se com o mundo exterior através de uma interface, que especifica as funcionalidades providas e as requeridas. Como em Java, uma interface em Koala é um pequeno conjunto de funções semanticamente relacionadas. As interfaces são tipos instanciados nas construções de *provides* e *requires*. As conexões entre os componentes se dão entre as instâncias dessas interfaces, pode-se dizer que elas representam as “portas” dos componentes.

Com o objetivo de gerar um código eficiente das especificações, técnicas de avaliação parcial são empregadas em certas expressões. Ao contrário de PECOS, Koala não possui um modelo de execução e também não suporta a descrição de escalonadores. Além disso, Koala não leva em consideração requisitos não funcionais tais como temporização e consumo de memória.

Capítulo 3

Primitivas do Sistema

Este capítulo apresenta um conjunto primitivas [45] utilizadas para representar uma grande quantidade de sistemas embutidos. Essas primitivas são entidades atômicas, não sendo decompostas em estruturas mais primárias.

As primitivas foram selecionadas tendo-se em mente os requisitos de uma representação no nível de arquitetura, os quais foram descritos no **Capítulo 1**, e servem de base para a definição dos predicados da linguagem LACCES.

As primitivas foram agrupadas de acordo com os elementos de computação ou de comunicação que elas caracterizam. Para cada um desses elementos, as primitivas foram classificadas de acordo com o domínio em que elas se aplicam.

Os elementos de computação e comunicação apresentados neste capítulo representam um conjunto mínimo de entidades necessárias para a descrição de um sistema embutido. Eles são os mais importantes encontrados nas aplicações atuais. Além disso, eles podem ser agregados com o objetivo de compor sistemas mais complexos.

- A seção 3.1 apresenta as primitivas dos componentes de hardware;
- A seção 3.2 apresenta as primitivas dos componentes de software;
- A seção 3.3 apresenta as primitivas dos conectores que permitem a interconexão entre dois componentes.

3.1 Componentes de Hardware

Em LACCES, os componentes de hardware são COTS (*Components Off-The-Shelf*). Eles representam abstrações de dispositivos, onde somente o comportamento visível é considerado para o projeto e não a implementação da sua estrutura interna. Características físicas e estruturais de nível de arquitetura podem ser associadas aos COTS. Elas são capturadas utilizando uma metodologia de projeto *bottom-up*.

Como já foi dito, as primitivas classificam-se em 4 domínios: comportamental, temporal, estrutural e físico. As primitivas de nível físico estão presentes em todos os componentes que possam ser implementados em *chips*. Elas consistem do custo, que representa o valor monetário do componente; informações específicas do dispositivo - nome do fabricante, nome da família, tipo de empacotamento, dimensões físicas e a quantidade de energia necessária para o seu funcionamento.

A Tabela 2 resume as primitivas de nível físico.

Nome	Descrição
<i>Chip name</i>	Nome da família do chip
<i>Package</i>	Tipo de empacotamento utilizado, por exemplo, TSOP, BGA, VSOP.
<i>Manufacturer name</i>	Nome do fabricante do dispositivo.
<i>Power consumption</i>	Energia consumida para o funcionamento do dispositivo.
<i>Dimensions</i>	Área ou espaço disponível para a implementação. Podem ser expressos como valores de 2D (duas dimensões), 3D (três dimensões) e fator de forma.
<i>Costs</i>	Valor monetário do dispositivo.

TABELA 2 – Primitivas do nível físico de um MCU.

3.1.1 Microcontrolador

Um microcontrolador (MCU) é um tipo particular de microprocessador, otimizado para realizar funções de controle a um baixo custo, possuindo um tamanho bastante reduzido.

Em LACCES, um microcontrolador pode ser constituído dos seguintes sub-componentes: Memória, RTC (*Real Time Clock*), ADC (*Analogue to Digital Converter*), DAC (*Digital to Analogue Converter*), Multiplicador, Controlador de LCD (*Liquid Crystal Display*), CPU (*Central Processing Unit*) e Temporizador (*Timer*).

Somente as funcionalidades dos periféricos podem ser acessadas pelos dispositivos externos ao MCU. Os periféricos se comunicam com a CPU através de registradores. A Figura 11 ilustra um exemplo de um MCU.

Em LACCES, o MCU só possui primitivas de nível físico. As primitivas comportamentais, temporais e estruturais são descritas pela agregação das primitivas de seus sub-componentes.

Como já foi dito, em LACCES, as primitivas físicas são utilizadas somente para descrever *chips*. Logo, os componentes internos ao MCU não possuem tais tipos de primitivas.

A Tabela 2 apresenta as primitivas físicas aplicáveis ao MCU.

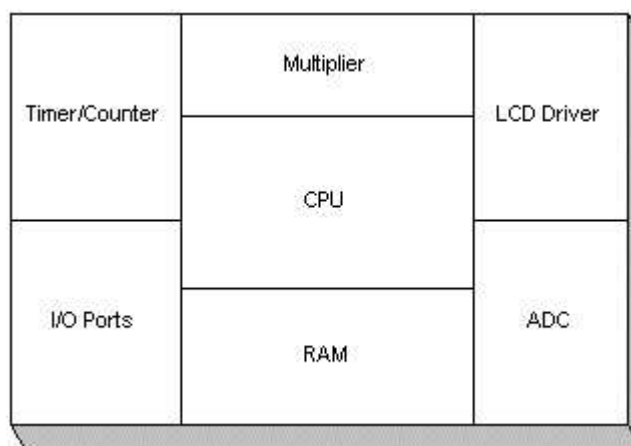


FIGURA 11 - Exemplo de um MCU.

3.1.2 Memória

Uma memória é um componente de hardware utilizado para armazenar dados e instruções. Em LACCES, a memória pode ser utilizada tanto para descrever uma memória interna quanto uma externa ao microcontrolador.

O comportamento de uma memória consiste das primitivas básicas de leitura e escrita. Em memórias seriais, essas primitivas podem se comportar como operações de leitura e escrita sequenciais. Além dessas primitivas, existem outras que também podem ser utilizadas na descrição do comportamento de uma memória. A primitiva *reset* estabelece o estado inicial da memória. Essa condição é atingida pela alteração de um sinal de uma das portas de entrada. Quando esse sinal for verdadeiro a operação que estava sendo executada é interrompida e as saídas têm atribuído a seus valores alta impedância. A primitiva *erase* consiste em apagar um determinado conteúdo da memória, por exemplo, um setor ou um bloco. Uma vez que a operação de *erase* tiver sido executada, todos os bits dos dados apagados tem seu valor alterado de '0' para '1'. A primitiva *refresh* é utilizada para recarregar os dados de memórias dinâmicas. Essa operação deve ser executada periodicamente para manter a consistência dos dados armazenados. A primitiva *lock* é utilizada como um mecanismo de proteção, não permitindo que a memória seja escrita ou apagada. A primitiva *read-write* consiste da leitura de uma posição de memória e, em seguida, da escrita de um dado nesta posição.

O desempenho de uma memória é primariamente determinado pela frequência na qual os dados podem ser lidos ou escritos. Uma medida de desempenho conveniente é o tempo médio requerido para ler ou escrever uma quantidade de informação da memória, por exemplo, uma palavra. Isso é chamado de tempo de acesso (*access time*). Esse tempo é calculado a partir do momento em que é recebida uma operação de leitura ou escrita, até o tempo em que o dado requisitado está disponível na saída ou foi totalmente escrito na posição informada.

Além do tempo de acesso, existe um tempo mínimo para a execução de duas operações subseqüentes na memória. Esse tempo é descrito através da primitiva *cycle time*. Em memórias dinâmicas, por exemplo, não pode ser realizado um novo acesso a memória sem que seja executada anteriormente uma operação de *refresh*. Isso significa que o tempo para a execução de duas operações diferentes deve ser maior do que o tempo de acesso. Pode-se dizer que o tempo de acesso é menor ou igual ao tempo de ciclo.

O circuito de acesso aos dados da memória tem um efeito significativo no seu custo total. Uma maneira de diminuir esse custo é utilizar uma organização mais adequada [25]. A organização mais simples e mais cara é a de um vetor, nesse caso as memórias são chamadas de 1D (uma dimensão). Cada célula de uma memória 1D está associada a um endereço. Memórias 2D (duas dimensões) são organizadas como matrizes. O endereçamento é dividido em dois componentes X e Y. Memórias 2.5 D (duas dimensões e meia) são implementadas como vetor de vetores. Nesse caso, as memórias são constituídas de setores, páginas e blocos.

Além da organização, outras primitivas estruturais compreendem a capacidade de armazenamento, que representa a área adequada para guardar os dados, o tipo da memória: RAM, ROM, etc, e, para memórias internas a um MCU, se ela é de programa e/ou dados.

As memórias podem ser utilizadas como componentes internos ou externos ao MCU. No caso de memórias internas, as primitivas de nível físico não se aplicam.

A Tabela 3 resume as primitivas de memória.

Domínio	Nome	Descrição
Comportamento	<i>Read</i>	Operação de leitura de uma posição de memória. Em memórias seriais, essa operação pode ler mais de uma posição de memória em um único acesso.
	<i>Write</i>	Operação de escrita em uma posição da memória. Em memórias seriais, essa operação pode escrever mais de um dado em um único acesso.
	<i>Erase</i>	Operação que apaga o conteúdo da memória.
	<i>Reset</i>	Operação que estabelece o estado inicial da memória.
	<i>ReadWrite</i>	Operação que lê uma posição da memória e, em seguida, escreve um dado no endereço lido.
	<i>Refresh</i>	Operação que consiste no refrescamento dos dados da memória.
	<i>Lock</i>	Define o tamanho da memória que não pode ser apagada e nem escrita. (<i>Software Protection</i>)
Tempo	<i>Access time</i>	Tempo médio necessário para ler/escrever uma quantidade fixa de informação na memória.
	<i>Cycle time</i>	Tempo necessário para realizar uma operação na memória.
Estrutura	<i>Organization</i>	Caracteriza como o arranjo da memória é organizado: 1D (uma dimensão), 2D (duas dimensões) ou 2.5 (duas dimensões e meia).
	<i>Capacity</i>	Capacidade de armazenamento da memória.

	<i>Device type</i>	Tipo de memória: ROM, PROM, EEPROM, EPROM, RAM, DRAM, SRAM, SDRAM ou Flash.
	<i>Memory Area</i>	Indica se a memória interna ao MCU é uma memória de programa e/ou de dados.
Físico	<i>Chip name</i>	Nome da família do chip.
	<i>Package</i>	Tipo de empacotamento utilizado, por exemplo, TSOP, BGA, VSOP.
	<i>Manufacturer name</i>	Nome do fabricante do dispositivo.
	<i>Power consumption</i>	Energia consumida para o funcionamento do dispositivo.
	<i>Dimensions</i>	Área ou espaço disponível para a implementação. Podem ser expressos como valores de 2D (duas dimensões), 3D (três dimensões) e fator de forma.
	<i>Costs</i>	Valor monetário do dispositivo.

TABELA 3 – Primitivas de uma memória

3.1.3 CPU (*Central Processing Unit*)

A CPU é o cerne do microcontrolador. Em LACCES, o comportamento de uma CPU é descrito pelas primitivas de interrupção, de *reset* e de uma primitiva que representa o seu conjunto de instruções. Uma interrupção é uma entrada externa a CPU, que pode ser usada para forçar o processador a prover uma resposta rápida a eventos externos. Fontes de interrupções podem sinalizar situações de emergência – a temperatura excedeu um limite máximo ou a fonte de energia foi perdida. Alternativamente, elas podem sinalizar uma rotina, possivelmente eventos periódicos – uma tecla foi pressionada, um dado foi recebido na porta de entrada, ou uma conversão analógica para digital foi executada. A primitiva de *reset* consiste de uma operação que reinicializa a CPU. A primitiva *code* compreende a execução de um conjunto de instruções. Essas instruções não foram listadas separadamente, pois não é objetivo de LACCES descrever a arquitetura interna de uma CPU.

A primitiva de tempo da CPU corresponde ao ciclo de instrução, ou seja, ao tempo médio de execução de uma instrução.

A estrutura de uma CPU é definida pelo tipo de sua arquitetura e pelo tamanho da palavra. A arquitetura de uma CPU pode ser *Von Neumann* ou *Harvard*. Em uma arquitetura *Von Neumann* a memória de programa e a de dados compartilham o mesmo barramento de endereçamento e de dados, os quais estão contidos no mesmo mapeamento de memória. Em uma arquitetura *Harvard* o mapeamento da memória de dados e de programa é separado, existindo um barramento de dados e um de endereçamento para cada uma dessas memórias. O tamanho da palavra corresponde ao número em bits que o processador pode operar. Esse número reflete em todo o sistema do microcontrolador como, por exemplo, no tamanho do barramento de dados.

A Tabela 4 resume as primitivas da CPU. O domínio físico somente se aplica para a descrição de chips de processadores que possuem memória externa, como por exemplo o microprocessador DragonBall da Motorola.

Domínio	Nome	Descrição
Comportamento	<i>Reset</i>	A condição de <i>reset</i> é aquela na qual é atribuído ao PC o endereço inicial do programa, definido pelo vetor de <i>reset</i> . Existem várias fontes de <i>reset</i> como, <i>watchdog</i> , <i>reset</i> externo, um endereçamento ilegal, um código ilegal de uma operação, etc.
	<i>Interrupt</i>	Característica do microcontrolador para interagir com periféricos e com o mundo externo. Uma interrupção pode ser mascarada ou não mascarada. As interrupções podem, também, possuir prioridades de execução.
	<i>Code</i>	Conjunto de instruções da CPU.
Tempo	<i>Instruction Cycle</i>	Tempo médio de execução de uma instrução.
Estrutura	<i>Architecture type</i>	Tipo da arquitetura: <i>Harvard</i> ou <i>Von Neumann</i> .
	<i>Word-length</i>	Número de bits da ALU do microcontrolador.
Físico	<i>Chip name</i>	Nome da família do chip.
	<i>Package</i>	Tipo de empacotamento utilizado, por exemplo, TSOP, BGA, VSOP.
	<i>Manufacturer name</i>	Nome do fabricante do dispositivo.
	<i>Power consumption</i>	Energia consumida para o funcionamento do dispositivo.
	<i>Dimensions</i>	Área ou espaço disponível para a implementação. Podem ser expressos como valores de 2D (duas dimensões), 3D (três dimensões) e fator de forma.
	<i>Costs</i>	Valor monetário do dispositivo.

TABELA 4 – Primitivas de uma CPU.

3.1.4 Conversores

Embora os microcontroladores sejam dispositivos essencialmente digitais, eles gastam uma boa quantidade do seu tempo lidando com sinais analógicos. Eles devem ser capazes de converter sinais de entrada analógicos para um forma digital, por exemplo de um microfone ou de um sensor de temperatura, ou um sinal de entrada digital para um sinal de saída analógico, por exemplo, ao controlar um alto-falante.

Os conversores provêm a interface entre o mundo analógico e o digital. Um conversor analógico-digital (ADC) é um circuito eletrônico cuja saída digital é proporcional a sua entrada analógica. Um conversor digital-analógico (DAC) gera um sinal de saída analógico que representa um sinal digital de entrada.

O comportamento dos conversores é dado pela primitiva de conversão. O tempo que o conversor gasta para realizar uma conversão corresponde à primitiva de tempo do dispositivo.

As primitivas estruturais dos conversores correspondem à resolução, taxa de amostragem, precisão e arquitetura.

A resolução de um conversor consiste na menor alteração na entrada do conversor que gera uma alteração em sua saída. A resolução pode ser descrita de três maneiras: número de

bits, percentagem ou tensão. Por exemplo, um conversor DA (Digital-Analógico) de 8-bits com um intervalo de tensão da saída de 5 V, possui 8 bits de resolução, ou 0,39% ($100\%/256$) de resolução, ou ainda uma resolução de 19,5 mV ($5V/256$).

A frequência de amostragem do sinal de entrada é dada pela primitiva de taxa de amostragem. A frequência máxima que um conversor pode amostrar depende do tempo que ele gasta para realizar uma conversão. Se essa conversão ocorre em T segundos, então sua taxa de amostragem máxima não pode ser mais rápida do que $1/T$ amostras por segundo.

A precisão corresponde à diferença entre o valor medido e o valor verdadeiro da entrada.

Dependendo da resolução, aplicação e tecnologia, existem diferentes tipos de arquiteturas para os conversores. No caso dos conversores AD (Analógico-Digital), as principais arquiteturas [29] são:

- *Flash*: Os conversores *flash* são bastante rápidos, pois os bits de saída são determinados em paralelo. Esse método requer um grande número de comparadores, limitando a resolução do conversor a 8 bits (256 comparadores).
- *SAR (Successive Approximation Register)*: Os conversores SAR implementam, basicamente, um algoritmo de pesquisa binária. Eles consistem de um conversor DA e de um comparador. Um conversor SAR compara a saída do conversor DA com a tensão de entrada. Essa arquitetura é mais lenta do que a *Flash*, porque a comparação é feita em série. Entretanto, a sua resolução varia de 8 a 16 bits.
- *Integrating*: Um grande número de ADCs utilizam a técnica de integração, que mede o tempo para carregar e descarregar um capacitor, o que possibilita determinar a tensão de entrada. A desvantagem dessa metodologia é uma baixa taxa de conversão. A sua resolução varia de 12 a 24 bits.

A arquitetura de conversores DA é baseada em resistores, capacitores e corrente. Para uma aplicação que requer uma alta velocidade e uma baixa resolução, uma implementação baseada em resistores (*R-String*) é adequada. Para uma aplicação que requer uma alta resolução e uma baixa largura de banda, uma implementação baseada em capacitores (*Charge redistribution*) é adequada. Para uma aplicação que requer uma alta velocidade e uma resolução média, uma arquitetura baseada em corrente (*Current-Steering*) é adequada.

Os conversores podem ser utilizados como componentes internos ou externos ao MCU. No caso de conversores internos, as primitivas de nível físico não se aplicam.

A Tabela 5 resume as primitivas dos conversores.

Domínio	Nome	Descrição
Comportamento	<i>AD/DA Conversion</i>	Operação de conversão de um sinal analógico para um digital, no caso de um ADC; ou de um sinal digital para um analógico, no caso de um DAC.
Tempo	<i>Conversion Time</i>	Tempo gasto na conversão.
Estrutura	<i>Resolution</i>	Menor alteração na entrada que pode provocar uma alteração na saída.
	<i>Sampling Rate</i>	Frequência de amostragem do sinal de entrada.
	<i>Accuracy</i>	Precisão do dispositivo (por exemplo, +/- 2 bits menos significativos).
	<i>Architecture</i>	Tipo da estrutura do circuito utilizado. Alguns parâmetros dos conversores são dependentes da arquitetura utilizada, como, por exemplo, a resolução e a taxa de amostragem.
Físico	<i>Chip name</i>	Nome da família do chip.
	<i>Package</i>	Tipo de empacotamento utilizado, por exemplo, TSOP, BGA, VSOP.
	<i>Manufacturer name</i>	Nome do fabricante do dispositivo.
	<i>Power consumption</i>	Energia consumida para o funcionamento do dispositivo.
	<i>Dimensions</i>	Área ou espaço disponível para a implementação. Podem ser expressos como valores de 2D (duas dimensões), 3D (três dimensões) e fator de forma.
	<i>Costs</i>	Valor monetário do dispositivo.

TABELA 5 – Primitivas dos conversores.

3.1.5 Multiplicador

O hardware que realiza a multiplicação pode ser um periférico que não faz parte da CPU. Isso significa, que as suas atividades não interferem nas atividades da CPU. A multiplicação pode ser feita usando operadores com ou sem sinal, possuindo um tamanho variado (16 x 16 bits, 8 x 16 bits, etc).

Em LACCES, um multiplicador somente pode ser utilizado como um componente interno ao MCU.

A Tabela 6 resume as primitivas do multiplicador.

Domínio	Nome	Descrição
Comportamento	<i>Multiplication</i>	Operação de multiplicação.
Tempo	<i>Multiplication delay</i>	Tempo gasto para realizar uma operação de multiplicação.
Estrutura	<i>Operand Length</i>	Tamanho das entradas e da saída (exemplo, 16x16x16, 8x8x16, etc).

	Type	Tipos das entradas de dados (ex: unsigned, signed).
--	------	---

TABELA 6 – Primitivas do multiplicador.

3.1.6 Controlador de LCD

Os controladores de LCD geram os sinais de segmentos e sinais de comuns necessários para controlar o dispositivo de LCD. Os sinais de segmentos correspondem ao menor elemento (*pixel*) que podem estar ou não visíveis. Os sinais comuns denotam quantos segmentos estão conectados a um mesmo terminal de segmentos.

A capacidade de exibição de um controlador de LCD é dada pelo número total de sinais de segmentos e de comuns que ele pode controlar. A estrutura do dado exibido, por exemplo, de um dígito, é dada pelas dimensões da matriz de exibição. Essa matriz determina a “largura” e “altura” do dado exibido. Por exemplo, em um controlador de LCD com 40 sinais de segmentos e 16 sinais de comuns (*display capacity*), pode-se utilizar 5 sinais de segmentos e 8 sinais de comuns (*matrix*) para exibir cada dado de uma linha.

Em LACCES, o controlador de LCD somente pode ser utilizado como um componente interno ao MCU.

A Tabela 7 resume as primitivas do controlador de LCD.

Domínio	Nome	Descrição
Comportamento	<i>Pixel On/Off</i>	Atribui valores aos sinais de segmentos e comuns.
Tempo	<i>Cycle Time</i>	Tempo necessário para atribuição dos valores aos sinais de segmentos e comuns.
Estrutura	<i>Matrix</i>	Número de sinais de segmentos e comuns necessários para exibição de um dado.
	<i>Display Capacity</i>	Número total de sinais de segmentos e comuns.

TABELA 7 – Primitivas do controlador de LCD.

3.1.7 Timer

Quase todos os dispositivos embutidos possuem uma forte dependência de tempo. Isso pode implicar que a resposta a um evento externo deve ser realizada dentro de um certo intervalo máximo de tempo, ou que certas medidas relacionadas com o tempo devem ser realizadas, ou, ainda, que certas saídas dependentes do tempo devem ser geradas.

Para lidar com a questão do tempo, utilizam-se temporizadores (*Timers*). Esses dispositivos realizam as seguintes atividades: contagem de tempo, captura de eventos, geração de

interrupções, comparação de valores e geração de sinais PWMs.

Para contemplar todas essas atividades, considerou-se que o temporizador é subdividido nos seguintes sub-componentes: *timer/counter*, *event* e PWM.

As primitivas comportamentais, temporais e estruturais do temporizador são descritas pela agregação das primitivas de seus sub-componentes.

As primitivas estruturais, que se aplicam a todos os sub-componentes do temporizador, compreendem a fonte de clock e o divisor de sua frequência.

Os temporizadores podem ser utilizados como componentes internos ou externos ao MCU. No caso de temporizadores internos, as primitivas de nível físico não se aplicam.

A Tabela 8 resume as primitivas do temporizador.

Domínio	Nome	Descrição
Estrutura	<i>Prescaler</i>	Divisor da frequência do <i>clock</i>
	<i>Clock source</i>	Fonte de <i>clock</i> utilizada, podendo ser síncrona (a mesmo utilizado pela CPU) ou assíncrona (gerada por um oscilador).
Físico	<i>Chip name</i>	Nome da família do <i>chip</i> .
	<i>Package</i>	Tipo de empacotamento utilizado, por exemplo, TSOP, BGA, VSOP.
	<i>Manufacturer name</i>	Nome do fabricante do dispositivo.
	<i>Power consumption</i>	Energia consumida para o funcionamento do dispositivo.
	<i>Dimensions</i>	Área ou espaço disponível para a implementação. Podem ser expressos como valores de 2D (duas dimensões), 3D (três dimensões) e fator de forma.
	<i>Costs</i>	Valor monetário do dispositivo.

TABELA 8 – Primitivas do temporizador.

3.1.7.1 *Timer/Counter*

Esse sub-componente do temporizador realiza a contagem de pulsos, objetos ou eventos. Ele armazena a informação em um registrador, que pode ser acessado para leitura, escrita e que, também, pode ter o seu valor apagado. Uma interrupção pode ser gerada quando ocorrer o *overflow* do registrador.

A Tabela 9 resume as primitivas do Temporizador/Contador.

Domínio	Nome	Descrição
Comportamento	<i>Interrupt</i>	Interrupção gerada quando ocorre um <i>overflow</i> do temporizador/contador.
	<i>Read</i>	Leitura do valor do temporizador/contador.

	<i>Write</i>	Escrita de um valor no temporizador/contador.
	<i>Clear</i>	Apaga o valor armazenado no temporizador/contador.
Tempo	<i>Cycle Time</i>	Tempo de execução das operações de leitura, escrita e de apagar o valor contido no temporizador/contador.
Estrutura	<i>Resolution</i>	Número máximo de bits que podem ser armazenados no temporizador/contador

TABELA 9 – Primitivas do timer/counter.

3.1.7.2 Events (*Input Capture e Output Compare*)

O componente evento é subdividido em uma estrutura de captura e outra de comparação. A estrutura de captura é responsável por capturar eventos externos, dando aos mesmos um rótulo de tempo, o qual indica o exato momento que tais eventos ocorreram. Esse rótulo pode ser usado para calcular a frequência e os ciclos de trabalho (*duty-cycles*) de sinais externos, para criar um registro de eventos ou para calcular outras características da aplicação. A captura pode gerar uma interrupção se o acontecimento do evento estiver configurado para tal.

A estrutura de comparação realiza a comparação do valor armazenado em dois registradores. Essa estrutura possui por objetivo a geração de um atraso de tempo. Quando os valores dos registradores tornam-se iguais, geralmente é gerada uma interrupção, mas na maioria das vezes o atraso de tempo é utilizado para iniciar um evento externo. O processador não precisa nem saber que o período de atraso terminou. Em alguns controladores, a saída do comparador é conectada diretamente a um pino externo ao processador.

Os valores capturados e os comparados podem ser lidos, entretanto, somente pode-se escrever no valor a ser comparado.

A Tabela 10 resume as primitivas do evento.

Domínio	Nome	Descrição
Comportamento	<i>Interrupt</i>	Interrupção gerada quando ocorre um evento (por exemplo, quando ocorre a captura de um dado, ou se a comparação de dois valores for verdadeira).
	<i>Capture</i>	Captura do momento da ocorrência de um evento.
	<i>Compare</i>	Comparação entre o valor do temporizador/contador e de um valor carregado no registrador de comparação.
	<i>Read</i>	Leitura dos valores de comparação e capturados.
	<i>Write</i>	Escrita do valor a ser comparado.
Tempo	<i>Access Time</i>	Tempo de acesso (leitura/escrita) aos registradores.
Estrutura	<i>Resolution</i>	Número máximo de bits utilizados para a captura e comparação.

TABELA 10 – Primitivas do Evento.

3.1.7.3 PWM (Pulse Width Modulation)

PWM é uma forma de modulação digital que varia o ciclo de trabalho (tempo ligado) de um sinal digital de período constante. O ciclo de trabalho é dependente do período e determina a percentagem de tempo que o sinal permanece em 1.

O número de bits gerados pelo PWM é dado por sua resolução.

A Tabela 11 resume as primitivas do PWM.

Domínio	Nome	Descrição
Comportamento	<i>Waveform Generation</i>	Geração de um sinal PWM de acordo com o período e o <i>duty-cycle</i> pré-definidos.
Tempo	<i>Period</i>	Período de um sinal PWM.
	<i>Duty-Cycle</i>	É a razão entre o tempo em que o sinal é 1 e o período.
Estrutura	<i>Resolution</i>	Quantidade de bits gerados.

TABELA 11 – Primitivas do PWM.

3.1.8 Módulo LCD

O módulo de LCD é um dispositivo utilizado para a exibição dos dados. Existem basicamente dois tipos de LCDs: caracter e pixel.

A capacidade do dispositivo é determinada pela quantidade de sinais comuns e segmentos que ele possui.

O módulo de LCD é um componente externo ao MCU. A Tabela 12 resume as primitivas do módulo de LCD.

Domínio	Nome	Descrição
Comportamento	<i>Display</i>	Exibição dos dados.
Tempo	<i>Display Delay</i>	Tempo gasto para a exibição dos dados.
Estrutura	<i>Capacity</i>	Número de caracteres ou de pixels que o terminal é capaz de exibir (por exemplo, 2 x16, 4 x16).
	<i>Display Type</i>	Tipo do terminal: caracter ou pixel
Físico	<i>Chip name</i>	Nome da família do chip.
	<i>Package</i>	Tipo de empacotamento utilizado, por exemplo, TSOP, BGA, VSOP.
	<i>Manufacturer name</i>	Nome do fabricante do dispositivo.
	<i>Power consumption</i>	Energia consumida para o funcionamento do dispositivo.

	<i>Dimensions</i>	Área ou espaço disponível para a implementação. Podem ser expressos como valores de 2D (duas dimensões), 3D (três dimensões) e fator de forma.
	<i>Costs</i>	Valor monetário do dispositivo.

TABELA 12 – Primitivas do Módulo LCD.

3.1.9 RTC (*Real Time Clock*)

O RTC provê um relógio de tempo real, com uma base de tempo própria, que fornece a representação do tempo e da data corrente para o processador do sistema. O tempo é contado em segundos, minutos e horas. A data é contada em dias da semana, dias do mês, mês e ano. Esses valores são armazenados em registradores que podem ser acessados através de operações de leitura e escrita. O tempo gasto para acessar esses registradores é dado pela primitiva de tempo de acesso.

Além da representação do tempo, o RTC, também, provê um alarme que é acionado quando o valor armazenado no registrador de alarme (uma hora ou uma data) é igual ao valor do registrador de contagem de tempo.

Uma interrupção pode ser gerada de acordo com a ocorrência de um evento. Por exemplo, o alarme foi acionado, ou o contador foi incrementado em um segundo, minuto, hora, dia ou ano, ou ocorreu um *overflow* do contador.

Uma característica estrutural é a capacidade de armazenamento dos registradores. Esse valor determina a resolução do RTC.

O RTC pode ser utilizado como componente interno ou externo ao MCU. No caso de um RTC interno, as primitivas de nível físico não se aplicam. A Tabela 13 resume as primitivas do RTC.

Domínio	Nome	Descrição
Comportamento	<i>Interrupt</i>	Interrupção gerada quando ocorre um evento.
	<i>Alarm</i>	Aciona o alarme quando o valor programado no registrador de contagem de tempo é igual ao armazenado no registrador de alarme.
	<i>Read</i>	Leitura dos dados dos registradores.
	<i>Write</i>	Escrita dos dados nos registradores
Tempo	<i>Access Time</i>	Tempo de acesso (leitura/escrita) aos registradores.

Estrutura	<i>Resolution</i>	Número máximo de bits dos registradores utilizados para o armazenamento das informações do RTC.
Físico	<i>Chip name</i>	Nome da família do chip.
	<i>Package</i>	Tipo de empacotamento utilizado, por exemplo, TSOP, BGA, VSOP.
	<i>Manufacturer name</i>	Nome do fabricante do dispositivo.
	<i>Power consumption</i>	Energia consumida para o funcionamento do dispositivo.
	<i>Dimensions</i>	Área ou espaço disponível para a implementação. Podem ser expressos como valores de 2D (duas dimensões), 3D (três dimensões) e fator de forma.
	<i>Costs</i>	Valor monetário do dispositivo.

TABELA 13 – Primitivas do RTC.

3.1.10 Keypad

Ao contrário dos dispositivos apresentados até o momento, o Keypad não é um componente de prateleira. Em LACCES, ele é utilizado para representar uma interação entre o ambiente externo e o sistema embutido. O Keypad consiste de um conjunto de chaves que ao serem pressionadas geram uma interrupção no MCU. A característica temporal do Keypad é dependente do tempo de varredura do software que processa a informação, quando uma chave é pressionada.

O Keypad é um componente externo ao MCU. A Tabela 14 resume as suas primitivas.

Domínio	Nome	Descrição
Comportamento	<i>Interrupt</i>	Interrupção gerada quando uma tecla é pressionada e solta.
Tempo	-	Tempo de varredura do software que processa a informação.
Estrutura	<i>Number of Keys</i>	Número de teclas que o dispositivo possui.
Físico	<i>Power consumption</i>	Energia consumida para o funcionamento do dispositivo.
	<i>Dimensions</i>	Dimensões do dispositivo.
	<i>Costs</i>	Valor monetário do dispositivo.
	<i>Number of Keys</i>	Número de chaves que implementam o Keypad.

TABELA 14 – Primitivas do Keypad.

3.2 Componente de Software

3.2.1 Module

O comportamento de um componente de software é representado através de processos, os quais são definidos pelos projetistas.

Em LACCES, o nível físico dos componentes de software não é descrito. Esse nível depende da CPU em que o software será executado. A Tabela 15 resume as primitivas do Módulo.

Domínio	Nome	Descrição
Comportamento	-	Definido pelo projetista.
Tempo	<i>Execution Time</i>	Tempo de execução de um comportamento específico. Inclui o tempo de entrada, saída e de computação.
Estrutura	<i>Data Length</i>	Tamanho da área de dados necessária para a execução de um comportamento específico.
	<i>Code Length</i>	Tamanho da área de memória necessária para armazenar o código de um comportamento específico.

TABELA 15 – Primitivas do Módulo.

3.3 Conectores

Existem oito tipos de conectores em LACCES, que são classificados de acordo com os componentes de software ou de hardware que eles interconectam.

A Tabela 16 apresenta a classificação dos tipos de conectores. As primitivas de cada um deles são descritas nas seções seguintes.

Comunicação/ Conector	Hw-Hw	Hw-Sw	Sw-Sw
SPI			
I2C			
UART			
Analógico			
Paralelo			
FIFO			

Semáforo			
IPC			

TABELA 16 – Classificação dos tipos de conectores.

3.3.1 Comunicação entre Componentes de Hardware

3.3.1.1 Conectores Digitais

Existem basicamente duas maneiras de transmitir dados digitais: paralelamente ou serialmente. Na transmissão paralela, cada bit da palavra de dados, que está sendo transmitida, possui a sua própria conexão. Além disso, todos os bits de cada palavra são transmitidos simultaneamente. Isso implica em uma transmissão de alta velocidade, ao custo de um grande número de interconexões.

Na transmissão serial, cada bit é enviado em uma única conexão de dados. Esse tipo de transmissão gasta mais tempo, porém o número de interconexões é drasticamente reduzido.

No mundo dos sistemas embutidos, a interconexão serial é muito importante. Além de economizar espaço nas placas de circuito impresso, o tempo gasto na transmissão dos dados, comparado com uma comunicação paralela, não é um grande problema.

Para que os dispositivos eletrônicos comuniquem-se com sucesso, eles devem concordar em como os dados são enviados e recebidos, ou seja, eles devem falar a mesma língua. Um protocolo de comunicação é essencialmente um conjunto de regras que governam como a comunicação de dados pode ser estabelecida.

Os protocolos e padrões seriais mais importantes no mundo dos sistemas embutidos são SPI, I2C e UART.

3.3.1.1.1 SPI (Serial Peripheral Interface)

O protocolo SPI foi criado pela Motorola e consiste de um padrão de comunicação serial entre circuitos integrados. O SPI é essencialmente um barramento serial de “3+n” fios, onde “n” corresponde ao número de dispositivos escravos. Esses três fios carregam a informação necessária para a comunicação. Dois dos três fios são utilizados para a transferência de dados (um para cada direção) e o terceiro fio é um *clock* serial. Todas as linhas do barramento SPI são unidirecionais.

Os dispositivos conectados ao barramento SPI podem ser classificados como mestres ou escravos. O dispositivo mestre é o responsável por iniciar a transferência de informações; pela geração do *clock*, que é usado para sincronizar a transferência de dados; e pela geração dos sinais de controle. Um dispositivo escravo é controlado pelo mestre através de uma linha de seleção e só está ativo quando for selecionado. Os outros escravos não selecionados não interferem nas atividades do barramento. Geralmente, uma linha separada para a seleção é necessária para cada

dispositivo escravo. Um mesmo dispositivo pode possuir a funcionalidade de um mestre ou escravo. Mas em um momento específico do tempo, somente um mestre pode controlar o barramento.

O barramento SPI possui várias velocidades de transferência de dados, que é em função do *clock* do sistema. Uma informação pode ser transmitida na razão de próximo a 0 bps até a 20 Mbps .

A transferência de dados é realizada em blocos de 8 ou 16 bits. Um bit de dado é transferido em cada ciclo de *clock*. Podem ser transferidos tantos blocos de dados quantos forem necessários, para isso deve-se manter o dispositivo selecionado.

Quatro modos de *clock* são definidos para o barramento SPI, os quais são determinados pelo valor da polaridade e da fase. A polaridade determina se o estado de inatividade (*idle state*) do *clock* será o nível zero ou um. A fase determina qual transição do *clock* (borda de subida ou de descida) corresponde a novos dados no barramento.

As primitivas do SPI são resumidas na Tabela 17.

Domínio	Nome	Descrição
Comportamento	<i>Send</i>	Operação de envio de um bloco de dados.
	<i>Receive</i>	Operação de recebimento de um bloco de dados.
	<i>Mode</i>	Configuração do modo de funcionamento da fase e da polaridade do <i>clock</i> . Em alguns casos, a fase ou a polaridade são alteradas entre as transmissões para permitir que o dispositivo mestre possa ser comunicar com um escravo que possui diferentes requisitos de comunicação.
Tempo	<i>Word Time</i>	Tempo gasto para transferir os dados
Estrutura	<i>Data rate</i>	Quantidade de dados transferidos por segundo.
	<i>Data format</i>	Define se os primeiros bits do bloco a ser transmitido são os mais significativos ou os menos significativos.
Físico	<i>Wires</i>	Número de fios necessários para a conexão entre o mestre e o escravo.

TABELA 17 – Primitivas do SPI.

Em LACCES, o comportamento de um conector do tipo SPI consiste, basicamente, na transferência de informações entre o mestre e o escravo. Como foi dito acima, o escravo deve ser primeiramente selecionado pelo mestre para que ocorra a comunicação. A tarefa de seleção do escravo está implícito no comportamento do conector.

3.3.1.1.2 *I²C (Inter-Integrated Circuit)*

O protocolo *I²C* consiste de um padrão desenvolvido pela Philips que provê um meio de comunicação simples entre circuitos integrados (IC) e que utiliza um pequeno número de fios. O barramento *I²C* compõe-se de duas linhas de comunicação – uma de dados bidirecional e outra

de *clock*.

Similarmente ao SPI, os dispositivos conectados ao barramento I²C classificam-se como mestres ou escravos. O I²C pode ser considerado um barramento “verdadeiro”, pois permite a existência de mais de um mestre. Além disso, ele inclui mecanismos para detecção de colisão e prevenção da corrupção dos dados se dois ou mais mestres iniciam simultaneamente a transferência de dados. O mestre pode agir como um transmissor ou um receptor.

O I²C utiliza um protocolo de comunicação “endereçável”, o qual permite que o mestre se comunique com o escravo endereçado no pacote de dados, para isso o mestre utiliza 7 ou 10 bits de endereçamento. Cada IC possui um endereço que é estabelecido pela Philips.

O mestre gera todos os sinais de *clock* para a comunicação com o escravo. A comunicação inicia-se quando o mestre gera um sinal de início e o barramento não está ocupado. Em seguida, ele envia o endereço do IC que ele deseja se comunicar e uma mensagem dizendo se serão enviados ou recebidos dados do dispositivo escravo. Os bits mais significativos do bloco são enviados primeiro. O mestre, então, espera um bit de confirmação do receptor da mensagem, que reconhece o endereço enviado e informa que está pronto para se comunicar. Em seguida, o mestre envia ou recebe 8 bits de dados tantas vezes quanto desejar. Depois de cada 8-bits de dados, o receptor gera um bit de confirmação. Quando todos os dados forem enviados, o mestre gera uma condição de parada.

O I²C possui três velocidades: modo padrão (100 Kbps), modo rápido (400 Kbps) e modo de alta velocidade (3,4 Mbps).

As primitivas do I²C podem ser resumidas na Tabela 18.

Domínio	Nome	Descrição
Comportamento	<i>Send</i>	Operação de envio de um bloco de dados.
	<i>Receive</i>	Operação de recebimento de um bloco de dados.
	<i>Ack</i>	Operação de confirmação de uma transmissão com sucesso.
Tempo	<i>Word Time</i>	Tempo gasto para transferir os dados
Estrutura	<i>Device address</i>	Formato do endereçamento utilizado pelo mestre (7 ou 10 bits).
	<i>Data Rate</i>	Quantidade de bits de dados transferidos por segundo.
Físico	<i>Wires</i>	Número de fios necessários para a conexão entre o mestre e o escravo.

TABELA 18 – Primitivas do I²C.

Similarmente ao protocolo SPI, o comportamento de um conector do tipo I²C consiste, basicamente, na transferência de informações entre o mestre e o escravo e da sua confirmação. Porém, o modo de seleção do escravo difere-se do SPI. As tarefas do mestre de iniciar e finalizar uma comunicação estão implícitas no comportamento do conector.

3.3.1.1.3 *UART (Universal Asynchronous Receiver Transmitter)*

UART é um padrão de comunicação antigo, implementado nos anos 60. O padrão UART é simples, universal, serial e muito lento. A transmissão é assíncrona, ou seja, o transmissor não precisa enviar o sinal de *clock*. Ao invés disso, o transmissor e o receptor devem concordar com os parâmetros de tempo. Bits especiais são inseridos em cada palavra para sincronizar o transmissor e o receptor.

Um bit chamado início é adicionado ao início de cada palavra a ser transmitida. O bit de início é usado para alertar o receptor que uma palavra de dados será enviada, e para forçar que o *clock* do receptor sincronize-se com o *clock* do transmissor.

Após o bit de início, 5 a 8 bits de dados são enviados. Cada bit enviado é transmitido na mesma quantidade de tempo de todos os outros bits. Os bits menos significativos do bloco são enviados primeiro. Quando todos o bits de dados são enviados, o transmissor pode adicionar ou não um bit de paridade ímpar ou par. O bit de paridade pode ser usado pelo receptor para realizar verificações de erros simples. Finalmente, pelo menos 1 bit de parada é enviado pelo transmissor, podendo ser também 1.5 ou 2 bits.

Como uma UART pode ser implementada em um *chip*, foram acrescentadas à sua primitiva de nível físico (*wires*) as primitivas de nível físico de um *chip*. As primitivas de uma UART são resumidas na Tabela 19.

Domínio	Nome	Descrição
Comportamento	<i>Send</i>	Operação de envio de um bloco de dados.
	<i>Receive</i>	Operação de recebimento de um bloco de dados.
Tempo	<i>Word Time</i>	Tempo gasto para transferir os dados.
Estrutura	<i>Baud rate</i>	Quantidade de bits transferidos por segundo, inclui os bits de inicialização, de finalização e o de paridade do bloco.
	<i>Data Rate</i>	Quantidade de bits de dados transferidos.
	<i>Data Length</i>	Tamanho da palavra de dados (5 a 8 bits).
	<i>Stop bit</i>	Número de bits de parada: 1, 1.5 ou 2 bits.
	<i>Parity</i>	Paridade utilizada pelos blocos de dados enviados, podendo ser ímpar, par ou nenhuma.
Físico	<i>Wires</i>	Número de fios necessários para a conexão entre o mestre e o escravo.
	<i>Chip name</i>	Nome da família do chip.
	<i>Package</i>	Tipo de empacotamento utilizado, por exemplo, TSOP, BGA, VSOP.
	<i>Manufacturer name</i>	Nome do fabricante do dispositivo.
	<i>Power consumption</i>	Energia consumida para o funcionamento do dispositivo.
	<i>Dimensions</i>	Área ou espaço disponível para a implementação. Podem ser expressos como valores de 2D (duas dimensões), 3D (três dimensões) e fator de forma.

	<i>Costs</i>	Valor monetário do dispositivo.
--	--------------	---------------------------------

TABELA 19 – Primitivas da UART.

3.3.1.2 Conector Analógico

Embora os microcontroladores sejam dispositivos essencialmente digitais, eles gastam bastante tempo comunicando-se com dispositivos analógicos, como, por exemplo, sensores. Essa comunicação ocorre através de conectores analógicos.

Em LACCES, os conectores analógicos são utilizados para conectar entradas de um conversor AD ou saídas de um conversor DA a um dispositivo de hardware.

Esse conector é caracterizado pelo tipo de sinal utilizado na comunicação, que pode ser corrente ou tensão. O tipo de sinal mais utilizado é a tensão. A corrente é freqüentemente utilizada para transmitir sinais em ambientes ruidosos, pois ela é menos afetada por tal distúrbio.

Para os sinais que utilizam tensão, existem as estruturas de conectores *single-ended* e *differential*. Conectores *single-ended* utilizam um fio para cada sinal a ser enviado, e todos os sinais referenciam um sinal terra em comum. Esse tipo de conector é sensível a ruído.

Um conector *differential* utiliza dois fios para cada sinal. A medida da tensão é a diferença entre a tensão nesses dois fios. Esses tipos de conectores possuem um desempenho maior e são menos suscetíveis a ruídos do que os conectores do tipo *single-ended*.

As primitivas de um conector analógico são resumidas na Tabela 20.

Domínio	Nome	Descrição
Comportamento	<i>Send</i>	Operação de envio de dados.
	<i>Receive</i>	Operação de recebimento de dados.
Tempo	<i>Word Time</i>	Tempo gasto para transferir os dados
Estrutura	<i>Frequency</i>	Frequência do sinal.
	<i>Connection Type</i>	<i>Single-Ended</i> ou <i>Differential</i> .
Físico	<i>Wires</i>	Número de fios necessários para a conexão.
	<i>Signal Type</i>	Corrente ou tensão.

TABELA 20 – Primitivas do conector Analógico.

3.3.2 Comunicação entre Componentes de Hardware e Software

3.3.2.1 Paralelo

A transmissão paralela de dados é modelada como um registrador que permite a escrita e a leitura dos bits de dados transmitidos pelos componentes.

As primitivas de um conector paralelo são resumidas na Tabela 21.

Domínio	Nome	Descrição
Comportamento	<i>Send</i>	Operação de envio de dados.
	<i>Receive</i>	Operação de recebimento de dados.
Tempo	<i>Word Time</i>	Tempo gasto para transferir os dados
Estrutura	<i>Data Rate</i>	Quantidade de dados transferidos por segundo.
	<i>Data Length</i>	Número de bits.
Físico	<i>Wires</i>	Número de fios necessários para a conexão.

TABELA 21 – Primitivas do conector Paralelo.

3.3.2.2 *FIFO*

A FIFO é uma fila que armazena os dados transferidos entre os componentes. Um dado enviado é acrescentado à fila e um dado lido é retirado da mesma.

A FIFO possui dois tipos de acesso: um que bloqueia e outro que não bloqueia o processo do componente que executa a operação.

Se a FIFO estiver vazia e está sendo utilizada a forma de acesso que bloqueia, uma leitura irá fazer com que o processo suspenda a sua operação até que mais dados estejam disponíveis na fila. Na versão em que não há bloqueio, o processo não é suspenso, ele continua a sua execução sem conseguir ler os dados da fila.

Da mesma forma, se a FIFO estiver cheia e está sendo utilizada a forma de acesso que bloqueia, uma escrita irá fazer com que o processo suspenda a sua operação até que alguns espaço esteja disponível. Na versão em que não há bloqueio, o funcionamento é igual à da leitura em que não há bloqueio.

Como uma FIFO pode ser implementada em um chip, foram acrescentadas às suas primitivas, as primitivas de nível físico de um *chip*. As primitivas da FIFO são resumidas na Tabela 22.

Domínio	Nome	Descrição
Comportamento	<i>Read</i>	Operação de leitura de um dado do início da fila, retirando-o da mesma.
	<i>Write</i>	Operação de escrita do dado no final da fila.
	<i>Blocking</i>	Indica se o processo que está executando a operação irá bloquear ou não.
Tempo	<i>Cycle Time</i>	Tempo necessário para executar a operação de leitura e escrita sem bloquear.
Estrutura	<i>Size</i>	Tamanho da fila.
	<i>Data Types</i>	Tipos dos dados armazenados na fila.
Físico	<i>Chip name</i>	Nome da família do chip.

	<i>Package</i>	Tipo de empacotamento utilizado, por exemplo, TSOP, BGA, VSOP.
	<i>Manufacturer name</i>	Nome do fabricante do dispositivo.
	<i>Power consumption</i>	Energia consumida para o funcionamento do dispositivo.
	<i>Dimensions</i>	Área ou espaço disponível para a implementação. Podem ser expressos como valores de 2D (duas dimensões), 3D (três dimensões) e fator de forma.
	<i>Costs</i>	Valor monetário do dispositivo.

TABELA 22 – Primitivas do conector FIFO.

3.3.3 Comunicação entre Componentes de Software

3.3.3.1 Semáforo

O conector Semáforo é utilizado na modelagem de sessões críticas. Os recursos compartilhados são acessados através de duas operações atômicas: *wait* e *signal*.

Existem dois tipos de semáforos: inteiro ou binário. Em um semáforo inteiro, o valor inteiro da variável do semáforo pode variar em um intervalo sobre um domínio irrestrito. Um semáforo binário é um semáforo inteiro cujo valor do intervalo é de 0 ou 1.

Um processo, antes de entrar em uma sessão crítica, tenta bloquear o acesso ao recurso através da operação *wait*. Se o recurso já estiver sendo utilizado, então o processo corrente, que está tentando acessá-lo, irá suspender a sua execução até que o acesso a esse recurso esteja disponível (operação *signal*).

As primitivas de um Semáforo são resumidas na Tabela 23

Domínio	Nome	Descrição
Comportamento	<i>Wait</i>	Operação que tenta utilizar o semáforo: o processo é bloqueado se o semáforo já estiver sendo utilizado.
	<i>TryWait</i>	Operação que verifica se o semáforo está sendo utilizado ou não. Essa operação não bloqueia o processo.
	<i>Signal</i>	Operação que libera o semáforo para outro processo utilizá-lo
	<i>GetValue</i>	Operação que recupera o valor do semáforo.
Tempo	<i>Access Time</i>	Tempo de acesso ao recurso compartilhado, que pode estar armazenado em uma memória ou em um registrador.
Estrutura	<i>Type</i>	Binário ou Inteiro.

TABELA 23 – Primitivas do Semáforo.

3.3.3.2 IPC (Inter-Process Communication)

IPC provê um mecanismo que permite que os processos comuniquem através de mensagens, sem a necessidade de utilizar variáveis compartilhadas.

Neste tipo de conector, as mensagens são enviadas e recebidas através de *mailboxes*.

Um *mailbox* pode ser visto como um objeto abstrato no qual as mensagens podem ser inseridas e retiradas pelos processos que participam da comunicação. Cada processo que deseja se comunicar com outro processo, deve criar um *mailbox* para a comunicação. A conexão só é estabelecida entre dois processos se eles compartilham o mesmo *mailbox*.

As primitivas de um IPC são resumidas na Tabela 24.

Domínio	Nome	Descrição
Comportamento	<i>Send</i>	Operação de envio de uma mensagem para o mailbox.
	<i>Receive</i>	Operação de leitura e retirada da mensagem do mailbox.
Tempo	<i>Cycle time</i>	Tempo para executar a operação de envio e de recebimento da msg.
Estrutura	<i>Message size</i>	Tamanho do <i>mailbox</i> .

TABELA 24 – Primitivas do IPC.

Capítulo 4

A Linguagem LACCES

A linguagem LACCES define os conceitos necessários para a captura do conhecimento do projetista acerca da arquitetura do sistema embutido a ser especificada.

Neste capítulo, o enfoque é dado para a especificação da estrutura e das construções da linguagem LACCES.

- A seção 4.1 apresenta as principais construções utilizadas para a especificação da arquitetura de um sistema embutido. Uma visão simplificada da gramática da linguagem é apresentada.
- A seção 4.2 descreve a estrutura necessária para a descrição do comportamento de componentes e conectores da linguagem.
- A seção 4.3 especifica os tipos de dados existentes na linguagem.
- A seção 4.4 apresenta como descrever as unidades de medida para as propriedades dos sistemas.

4.1 Estrutura de LACCES

Ao selecionar um conjunto apropriado de construções para uma linguagem, torna-se clara as preocupações e os objetivos do seu desenvolvedor. No caso de LACCES, ela precisa ser expressiva o suficiente para descrever a arquitetura de um sistema embutido, e ainda ser flexível e extensível o suficiente para permitir a descrição de sistemas em diferentes níveis de abstrações. LACCES soluciona essas preocupações ao prover uma linguagem simples e genérica que permite descrever a estrutura da arquitetura como fundamento para a sua especificação. No contexto de LACCES, a estrutura da arquitetura define a composição de um sistema como um conjunto de componentes e a maneira como esses componentes interagem. Essa estrutura forma o contexto necessário para se fazer todas as especificações e projetos subsequentes do sistema.

Como detalhado na Tabela 25, as construções estruturais básicas de LACCES são: *component*, *connector* e *system*. Essas construções são genericamente referenciadas como

elementos de projeto. A interface (*Interface*) de um componente descreve o seu comportamento provido (*Provides*) e o requerido (*Requires*). *Ports* e *Roles* representam o ponto de acesso dos componentes e conectores, respectivamente, ao ambiente externo. *Attachments* definem um ponto de interação entre um *port* e um *role*. Como resultado, um conjunto de *attachments* associados a um sistema define a sua topologia. Componentes, conectores e sistemas podem ser hierárquicos, sendo decompostos através da construção *Representation*. Um *mapeamento abstrato* define um mapeamento de um elemento externo a uma representação, com os seus elementos internos. Um *binding* é um tipo especial de mapeamento abstrato em LACCES que define a equivalência de duas entidades. Essas onze construções compõem as construções básicas de LACCES, as quais permitem capturar a estrutura da arquitetura de sistemas embutidos.

Selecionar essa coleção de construções básicas provou ser um passo crítico e desafiante para a definição desta linguagem. Elas foram selecionadas baseadas na combinação de: observações da prática de projetistas de software e de hardware que utilizam amplamente de blocos e linhas (representando componentes e conectores, respectivamente), estudo das técnicas informais e intuitivas utilizadas pelos projetistas (genericamente diagramas de blocos e linhas, associados a uma linguagem de texto natural e informal), e experiência da utilização de linguagens de descrição de arquitetura e de linguagens de interconexão de módulos (MIL) [15, 46].

Um dos primeiros passos no projeto de um sistema embutido é atribuir funcionalidades, responsabilidades e capacidades do sistema alvo a um conjunto de componentes. Os componentes representam os elementos primários de computação e de armazenamento do sistema.

Construção	Finalidade
<i>System</i>	Descreve um conjunto de componentes, conectores e suas interações.
<i>Component</i>	Representa o elemento computacional e de armazenamento de dados de um sistema.
<i>Connector</i>	Mediador da interação entre os componentes.
<i>Interface</i>	Especifica o comportamento e as portas de um componente do sistema.
<i>Port</i>	Define um ponto de acesso ao componente
<i>Role</i>	Define um ponto de acesso ao conector.
<i>Provides</i>	Conjunto de funcionalidades providas pelo componente.
<i>Requires</i>	Conjunto de funcionalidades requeridas pelo componente.
<i>Representation</i>	Permite o encapsulamento e a decomposição hierárquica dos elementos de projeto.
<i>Attachment</i>	Define um relacionamento entre uma porta e um <i>role</i> . O conjunto de <i>attachments</i> em um sistema define a sua topologia.
<i>Bindings</i>	Define um relacionamento entre um elemento de projeto e os elementos hierárquicos contidos na sua representação.

TABELA 25 – Visão geral das construções primitivas da linguagem LACCES.

Um componente possui uma interface e um corpo. A interface de um componente define um conjunto de portas, funcionalidades providas e requeridas pelo componente. As interfaces em linguagens como ADA (ex: *package specifications*), a parte pública das classes de C++, ou as interfaces descritas nas entidades de VHDL, especificam somente a assinatura das capacidades providas (ex, funções ou portas). Nenhuma define características requeridas de outros componentes e nem o seu comportamento independentemente de uma instância do módulo. Uma propriedade de LACCES é que um componente pode prover múltiplas interfaces para o seu ambiente. Isso permite separar a especificação do que o componente pode fazer de como ele é encapsulado. O corpo do componente descreve o comportamento, que é observado externamente, e propriedades não funcionais, como, por exemplo, custo e desempenho.

A decomposição de um sistema em seus componentes constituintes é somente o primeiro passo no projeto e na especificação da sua arquitetura. Um outro passo fundamental é a definição da maneira como esses componentes interagem. Os conectores solucionam o problema de representar essas interações entre os componentes.

Como os componentes, os conectores possuem explicitamente uma interface e um corpo. A interface é definida através de um conjunto de *roles*. Cada *role* define uma forma de participação do conector na comunicação. Os *roles* são associados internamente através de *RoleBindings*. Esse mapeamento define para qual *role* de saída deve ser enviada a informação que foi recebida em um *role* de entrada. O corpo do conector descreve o comportamento, que é observado externamente, e propriedades não funcionais, como, por exemplo, custo e desempenho.

Especificações individuais de componentes e conectores não são úteis se utilizadas isoladamente. Elas tornam-se interessantes quando compõem sistemas. LACCES provê a construção *System* para representar configurações de componentes e conectores. Um sistema inclui um conjunto de componentes, um conjunto de conectores e um conjunto de *attachments* que descrevem a topologia do sistema.

A Figura 12 ilustra um simples exemplo de dois componentes de software que se comunicam através do envio de mensagens.

4.1.1 Representações

O projeto de arquiteturas complexas requer descrições hierárquicas, o que a torna tratável do ponto de vista dos projetistas, possibilitando o seu entendimento. LACCES permite a decomposição hierárquica das arquiteturas. Especificamente, componentes, conectores e sistemas podem ser descritos por representações menos abstratas e que provêem mais detalhes. Uma representação estabelece e força uma fronteira entre a estrutura, interface, e propriedades que um elemento de projeto exporta para o ambiente e os seus detalhes internos. Uma representação consiste de duas partes: um sistema que descreve a estrutura da representação, e um conjunto de *bindings* que definem o mapeamento entre a estrutura do elemento de alto-nível, que encapsula a

representação, e os elementos menos abstratos, definidos internamente na representação do sistema.

Para ilustrar, vamos estender o exemplo do cliente-servidor, como descrito na Figura 12. Suponha que o servidor, neste exemplo, seja mais complexo e consista de três sub-componentes que interagem – um gerenciador de conexões, um gerenciador de segurança e um banco de dados. A Figura 13 ilustra essa extensão graficamente e a Figura 14 provê a especificação textual resumida em LACCES.

```

System simpleCS {
  Component Client : Module {
    Interface IntCliente {
      port out sendReq : int;
      Requires{
        ReceiveRequest();
      }
    }
    SendRequest();
  }
  Component Server : Module {
    Interface IntServer {
      port in recReq : int;
      Provides{
        ReceiveRequest();
      }
    }
    ReceiveRequest() sensitive recReq;
  }
  Connector RPCConn: IPC {
    Role in caller :int;
    Role out callee :int;
    RoleBindings{
      caller to callee;
    }
    Send(int data) assoc caller;
    int Receive() assoc callee;
  }
  Attachments{
    Client.sendReq to RPCConn.caller;
    RPCConn.callee to Server.recReq;
  }
}

```

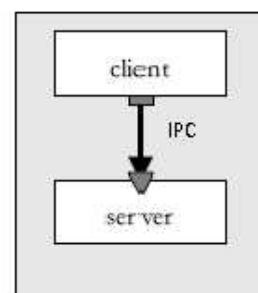


FIGURA 12– Descrição em LACCES de um sistema cliente-servidor.

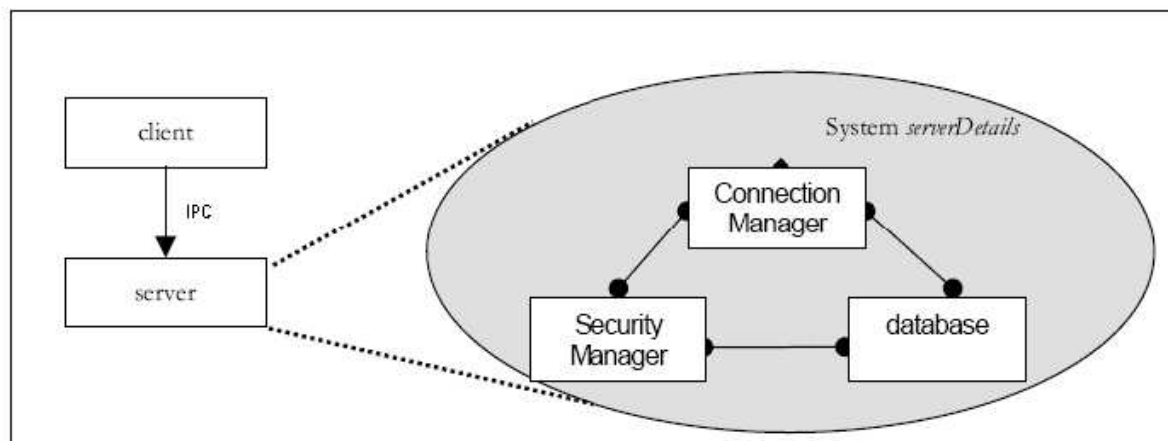


FIGURA 13— Representação hierárquica do servidor.

Adicionando uma representação para o servidor, possibilita ao projetista prover detalhes de baixo nível, sem comprometer sua abstração em alto-nível, como um único elemento de projeto. Ao trabalhar em um nível mais abstrato é apropriado esconder a complexidade do servidor, simplificando o conhecimento sobre ele como um componente que provê um serviço específico.

4.1.2 Sintaxe da Linguagem Estrutural de LACCES

A discussão até este ponto apresenta uma descrição breve das construções de LACCES para especificar a estrutura da arquitetura e as suas características. Para referência, uma curta gramática BNF (*Backus-Naur Form*) [2] é apresentada na Figura 15.

A simplicidade das construções centrais da linguagem reflete-se na sua gramática. Essa mínima especificação da linguagem omite quatro partes – a descrição do comportamento, os tipos de dados, a definição das propriedades e de suas unidades. A descrição da sintaxe será estendida em seguida, ao passo que as outras construções forem detalhadas. Para referência, a gramática BNF completa de LACCES encontra-se no **Apêndice A.1**.

```

System simpleCS {
  Component Client : Module {
    .....
  }
  Component Server : Module {
    Interface IntServer {
      port in recReq : int;
      Provides{
        ReceiveRequest();
      }
    }
    ReceiveRequest() sensitive recReq;
  Representation serverDetails {
    System serverDetailsSys {
      Component connectionManager : Module{
        Interface connMan{
          port in externalSocket :int;
          port out securityCheck :int;
          port out QueryInf :int;
          .....
        }
        .....
      }
      Component securityManager : Module{
        Interface connMan{
          port in securityAuthorization:int;
          port out credentialQuery: int;
          ....
        }
        .....
      }
      Component database : Module{
        Interface dataCon{
          port in securityManagement :int;
          port in SQLQuery :int;
        }
        .....
      }
    }
    Connector SQLQuery : IPC{
      Roles{
        in Caller :int;
        out Callee :int;
      }
      .....
    }
    Connector clearRequest : IPC {
      Roles {
        in Requestor :int;
        out Grantor :int;
      }
      ....
    }
  }
}

```

```

Connector securitQuery: IPC {
  Roles{
    in SecurityManager :int;
    out Requestor :int;
  }
  .....
}

Attachments{
  connectionManager.securityCheck to
  clearRequest.Requestor;
  clearRequest.Grantor to
  securityManager.securityAuthorization;
  connectionManager.QueryInf to
  SQLQuery.Caller;
  SQLQuery.Callee to
  database.SQLQuery;
  securityManager.credentialQuery to
  securitQuery.securityManager;
  securitQuery.requestor to
  database.securityManagement;
}
}
Bindings{
  connectionManager.externalSocket to
  Server.recReq;
}
}
}
.....

```

FIGURA 14– Descrição de um sistema cliente-servidor com o detalhamento do servidor.

4.1.3 Estendendo a Especificação Funcional com Propriedades Não Funcionais

Embora a estrutura e a topologia sejam aspectos críticos na arquitetura de um sistema, eles não são suficientes para a sua descrição completa. Existem vários outros aspectos do projeto de um sistema que podem e devem ser capturados pela especificação da arquitetura. Esses aspectos variam do tipo do sistema alvo que está sendo descrito e dos tipos de análises que serão realizadas. Um sistema de tempo real deve permitir que sejam descritas restrições de tempo real, como por exemplo, *deadlines*. Sistemas constituídos de clientes e servidores, de outra maneira, devem ser capazes de modelar os protocolos de comunicação.

Como esses dois exemplos ilustram, os projetistas também necessitam especificar informações não-funcionais. LACCES provê um conjunto de propriedades que podem ser associadas aos componentes e conectores do sistema. Uma propriedade em LACCES é uma construção que pode possuir uma unidade. Uma funcionalidade pode estar relacionada a uma propriedade, nesse caso o seu valor só será conhecido dinamicamente. A Figura 16 ilustra a

sintaxe das propriedades.

[] Produção opcional
| Separa escolhas alternativas
(...) seqüência de zero ou mais elementos*
(...)+ seqüência de um ou mais elementos
(...)? seqüência de zero ou um elemento

```

System          ::= System Name '{'(System_Body)*}'
System_Body     ::= ComponentDecl
                  |ConnectorDecl
                  |CompInterfaceDecl
                  |ConnInterfaceDecl
                  |RepresentationDecl
                  |AttachmentDecl
                  |BindingsDecl
ComponentDecl   ::= Component Name ':' ComponentType '{'(CompInterfaceDecl)+
                  (ProcDecl|PropertyDecl)* (RepresentationDecl)? '}'
ConnectorDecl   ::= Connector Name ':' ConnectorType '{' ConnInterfaceDecl
                  (FuncDecl|PropertyDecl)* (RepresentationDecl)? '}'
CompInterfaceDecl ::= Interface Name '{'(PortDecl)+ IntBehaviorDecl '}'
ConnInterfaceDecl ::= (RoleDecl)+ (RoleBindings '{'(RoleName to RoleName ';')+
                  '}')?
RoleDecl        ::= Role direction Name (',' Name)* ':' Type ';'
PortDecl        ::= Port direction [PortBind] Name (',' Name)* ':' Type ';'
PortBind        ::= '['Num']
ConnectorType   ::= HW_PARALLEL | SW_PARALLEL | SPI | I2C | HW_FIFO | SW_FIFO
                  |UART | ANALOGUE | SEMAPHORE | IPC | VIRTUAL
ComponentType   ::= MEMORY | MCU | LCD_MODULE | RTC | ADC | DAC | CPU
                  |MULTIPLIER | LDC_DRIVER | TIMER | EVENT | PWM
                  |TIMER_COUNTER | MODULE | KEYPAD | GENERIC
direction       ::= in | out | inout
Name            ::= [A-Za-z][A-Za-z0-9_\-]*
Type            ::= INT | BIT | INT '['(Num)?']' | BIT '['(Num)?']' | BYTE |
                  BYTE '['(Num)?']' | ANALOGUE | Name
Num             ::= [0-9]+
RepresentationDecl ::= Representation '{'System [BindingsDecl]}'
AttachmentDecl  ::= Attachments '{'(PortName to RoleName ';')+ '}'
BindingsDecl    ::= Bindings '{'(Name to Name ';')+ '}'
IntBehaviorDecl ::= Provides '{' (ProcDecl)+ '}' Requires '{' (Proc';')+ '}'
                  | Provides '{' (ProcDecl)+ '}' | Requires '{' (Proc';')+
                  '}'

```

FIGURA 15– Gramática BNF parcial para uma instância estrutural simples da linguagem.

```

PropertyDecl      ::= Name Operator UnitsConst
Operator          ::= '=' | '>' | '<=' | '!='
Exp               ::= primary | set | range | list
primary           ::= Name | Num
set               ::= '{' (ValueList)? '}'
range             ::= '[' (ValueList)? ']'
list              ::= '(' (ValueList)? ')'
ValueList         ::= primary (',' primary)* | primary
                  ...'primary
UnitsConst        ::= Exp [Name] ';' | ProcDecl

```

FIGURA 16– Sintaxe das propriedades.

O conjunto de propriedades é fixo na linguagem, o que permite que seja feita a sua análise. Outras propriedades podem ser descritas pelo projetista, entretanto seus valores não são interpretados por LACCES. As propriedades fixas na linguagem baseiam-se nas primitivas definidas no **Capítulo 3**.

4.2 Comportamento dos Elementos de Projeto

O comportamento dos elementos de projeto pode ser descrito através de processos ou métodos. Os processos são utilizados para descrever o comportamento concorrente dos componentes. Os métodos são utilizados para mediar a comunicação e a sincronização entre os componentes.

4.2.1 Processos

Em linguagens típicas de programação, métodos são executados sequencialmente, enquanto o controle é transferido de um método para outro. Métodos são úteis para modelar sistemas sequenciais. Sistemas eletrônicos são inerentemente paralelos, com muitas atividades ocorrendo simultaneamente. Modelar essas atividades paralelas através de uma linguagem sequencial pode ser bastante difícil. LACCES provê processos que permitem descrever as atividades concorrentes do sistema.

Um processo representa uma unidade funcional de um componente, estando contido na sua interface ou no seu corpo do mesmo. Na interface do componente existem dois tipos de processos, os que ele provê e os que ele requer. Os processos providos são aqueles que o componente disponibiliza para o ambiente externo, e os requeridos são processos de outros componentes que geram resultados necessários para a execução do seu comportamento. Um processo provido pode ser uma funcionalidade do seu comportamento interno ou uma funcionalidade específica da interface. Um processo requerido corresponde a um processo provido na interface de outro componente.

Processos são sensíveis a eventos (mudança de valores) que ocorrem nas portas do componente. Esses eventos disparam a execução de qualquer processo sensível a eles. Em LACCES, o projetista deve especificar para cada processo o conjunto de portas, associadas a eventos, que podem disparar a sua execução.

A sintaxe de um processo em LACCES é a seguinte:

```

ProcDecl      ::= Proc (SensDecl)? (ImplDecl)? (HasDecl)? ';'
Proc          ::= Name '(' ' ' ')'
SensDecl      ::= sensitive (PortName) ( ',' PortName)*

```

FIGURA 17 – Sintaxe do processo.

4.2.2 Métodos

Os métodos são descritos nos conectores e especificam os mecanismos de comunicação e sincronização entre os processos. Um método é invocado por um processo para realizar a comunicação com o seu ambiente externo.

Cada método de um conector está associado a um ou mais *roles*. Isso permite especificar quais funcionalidades estão disponíveis através de cada *role* do conector.

A sintaxe de um método em LACCES é a seguinte:

FuncDecl	::= [Type] Func (AssocDecl)? (ImplDecl)? (HasDecl)? ';' ;
AssocDecl	::= assoc RoleName (',' RoleName)*
Func	::= Name '(' ParameterList ')'
ParameterList	::= Type Name (',' Type Name)*

FIGURA 18– Sintaxe do método.

4.2.3 Implementação

O comportamento de um processo ou de um método definido pelo usuário pode ser implementado diretamente no código de alguma linguagem de programação. Esse código torna-se disponível para LACCES através da construção *Implementation*.

Essa construção permite especificar características da implementação do comportamento, tais como o tipo da linguagem utilizada, o nome do arquivo que contém o código, se esse código é fonte ou executável. Essa informação será útil para a posterior simulação da arquitetura.

A sintaxe da construção *Implementation* em LACCES é a seguinte:

ImplDecl	::= Implementation '{' in FileName ';' ; FileType TypeName ';' // Executable ou Source [ImplType ImplName ';'] // Hardware, Software ou None [With Name ';'] // Para processo hierárquicos [Language Name ';'] // Nome da linguagem. }'
----------	---

FIGURA 19– Sintaxe da implementação do comportamento.

Além da especificação das características de implementação, o projetista também pode definir propriedades não funcionais associadas ao comportamento. Isso é possível através da construção *Has*:

HasDecl	::= has {(PropertyDecl)+}
---------	----------------------------------

FIGURA 20 – Sintaxe da construção *Has*.

4.3 Tipos de Dados

Os tipos de dados disponíveis em LACCES são:

- **Int**: tipo inteiro cujo tamanho é dependente da máquina, usualmente é representado através de 32 bits.
- **Bit**: tipo de dado que possui dois valores - '0' (falso) ou '1' (verdadeiro).
- **Byte**: tipo de dado constituído de 8 bits.
- **Int[]**: tipo utilizado para representar um inteiro cujo tamanho em bits varia.
- **Bit[]**: vetor de bits.
- **Byte[]**: vetor de bytes.
- **Analogue**: tipo utilizado para representar um dado analógico.

Além desses tipos, LACCES permite que o projetista defina estruturas de dados, representando pacotes.

Um pacote é uma estrutura de dados análoga a um registro (*Record*), sendo composto de uma sequência de itens de dados heterogêneos e concatenados. A sintaxe para a definição de um pacote é a seguinte:

PacketDecl	::= Packet Name ':' ProtocolType '{'(PacketElem)+'}'
PacketElem	::= Name (',' Name)*: Type ';' ;
ProtocolType	::= SPI I2C UART Name

FIGURA 21– Sintaxe da construção *Packet*.

Os pacotes devem ser especificados como parte de todos os sistemas descritos em LACCES que necessitam de tipos de dados definidos pelo usuário. As definições dos pacotes podem estar contidas no mesmo arquivo da descrição do sistema ou em um arquivo separado e que é incluído no arquivo principal.

4.4 Unidades

A descrição de uma arquitetura requer propriedades cujos valores podem possuir diferentes tipos de unidades de medida, tais como bits, bytes e nanosegundos. LACCES permite que o projetista defina os valores das suas propriedades em unidades que sejam convenientes para a legibilidade de seu projeto. Enquanto isso, a linguagem realiza as conversões necessárias entre as unidades.

A sintaxe para a definição de unidade é:

```
UnitsDefinition      ::= Units Name '(' BaseUnit (DerivedUnit)+ ')'
BaseUnit            ::= Name
DerivedUnit         ::= Name '=' Exp [Name] ';'

```

FIGURA 22 – Sintaxe da definição de unidades.

A Figura 23 ilustra a definição da unidade espaço:

```
Units space (
    bit /* bit é a unidade básica */
    bits = bit;
    byte = 8 bits;
    bytes = byte;
    KB = 1024 bytes;
    MB = 1024 KB;
    GB = 1024 MB;
}

```

FIGURA 23– Unidade espaço.

As unidades de medida devem ser definidas como parte de todos os sistemas descritos em LACCES. As definições das unidades podem estar contidas no mesmo arquivo da especificação do sistema ou em um arquivo separado e que é incluído no arquivo principal. As definições de todas as unidades disponíveis para o projeto do sistema encontram-se no **Apêndice A.2**.

Capítulo 5

Definição Formal

Este capítulo apresenta uma definição teórica dos principais conceitos da notação LACCES, utilizando grafos. O propósito de tal formalismo é prover uma definição sólida para cada elemento da linguagem.

- A seção 5.1 apresenta a definição do sistema;
- A seção 5.2 apresenta a definição relacionada aos elementos de computação do sistema;
- A seção 5.3 apresenta a definição relacionada aos elementos de comunicação do sistema.

5.1 Sistema

Um sistema é visto como uma coleção de componentes concorrentes e comunicantes que cooperam entre si com o objetivo de realizar uma tarefa mais complexa. A comunicação possibilita que eles troquem informações e sincronizem-se.

Em LACCES, a concorrência é limitada ao nível do sistema. Dentro de cada componente, as funcionalidades são executadas sequencialmente em uma certa ordem. Pode-se dizer que o sistema comporta-se como uma arquitetura globalmente assíncrona e localmente síncrona (GALS), ou seja, cada componente é localmente síncrono, enquanto a comunicação é primariamente assíncrona.

Definição 5.1: Um **sistema** é um grafo bipartido $S = (C_p, C_n, L)$, onde

C_p é o conjunto de vértices que representa nós de computação.

C_n é o conjunto de vértices que representa nós de comunicação.

L é o conjunto de arestas entre os nós de computação e de comunicação, onde $L \subseteq (C_p \times C_n \cup C_n \times C_p)$.

Embora a completude da arquitetura de um sistema seja necessária para a sua simulação, LACCES não garante tal propriedade, mas permite que sejam capturadas descrições parciais e eventualmente quando o projetista prover mais detalhes, seja representado um projeto completo.

5.2 Computação

Como já foi dito anteriormente, os elementos de computação do sistema são representados através de componentes.

De acordo com a definição apresentada do sistema, um componente pode ser definido da seguinte maneira:

Definição 5.2: Um **componente** é um nó de computação $C_{pi} \in C_p$. Ele é um elemento ativo que possui um tipo, denotado por $\text{tipo}(C_{pi})$, onde $\text{tipo}(C_{pi}) \in \{Memory, CPU, LCD_Module, LCD_Driver, RTC, ADC, DAC, MCU, Timer, Event, PWM, Timer_Counter, Keypad, Multiplier, Module, Generic\}$. O conjunto dos tipos dos nós de computação pode ser subdividido em três subconjuntos, com $\text{tipo}(C_{pi}) = \{C_{pH} \cup C_{pS} \cup C_{pN}\}$, onde:

C_{pH} consiste do conjunto dos nós de computação que representam dispositivos de hardware C_{pH} , onde $\text{tipo}(C_{pH}) \in \{Memory, CPU, LCD_Module, LCD_Driver, RTC, ADC, DAC, MCU, Timer, Event, PWM, Timer_Counter, Keypad, Multiplier\}$;

C_{pS} consiste do conjunto dos nós de computação que representam elementos de software C_{pS} , onde $\text{tipo}(C_{pS}) \in \{Module\}$;

C_{pN} consiste do conjunto dos nós de computação cuja implementação ainda não foi definida C_{pN} , onde $\text{tipo}(C_{pN}) \in \{Generic\}$.

Os tipos dos componentes foram escolhidos tendo-se em vista os elementos de computação descritos no **Capítulo 3**. O tipo *Generic* foi utilizado para permitir que o projetista defina um componente diferente dos especificados na linguagem e cuja implementação, em hardware ou em software, não é conhecida no momento do projeto.

Um nó de computação é hierárquico, podendo ser constituído de outros nós de comunicação e de computação. Um componente *Generic* pode possuir sub-componentes de hardware, software ou genérico.

Um nó de computação C_{pi} é constituído de um corpo e de uma interface.

Definição 5.3: O **corpo** de um nó de computação é um conjunto B_{cmp} , com $B_{cmp} = \{B_{cp} \cup B_{tp} \cup B_{ep} \cup B_{fp}\}$, onde:

B_{cp} consiste do conjunto de operações básicas que o nó de computação realiza, representando o seu comportamento visível externamente, isto é, o comportamento que é visível e que pode ser observado pela arquitetura que contém esse nó. Essas operações são descritas através de processos;

B_{tp} consiste do conjunto de propriedades temporais associadas ao nó de computação;

B_{ep} consiste do conjunto de propriedade estruturais associadas ao nó de computação;

B_{fp} consiste do conjunto de propriedade físicas associadas ao nó de computação.

Definição 5.4: Uma **interface** de um nó de computação é um conjunto I_{cmp} , com $I_{cmp} = \{P \cup F_p \cup F_R\}$ onde:

P define um conjunto de portas que representam o ponto de interação do nó com o meio externo. Cada porta $P_i \in P$ é caracterizada por um nome, por um tipo e por uma forma de acesso. O tipo é denotado por $tipo(P_i)$, com $tipo(P_i) \in \{int, bit, byte, int[x], bit[x], byte[x], analogue, <PacketName>\}$, onde $x > 0$ e $PacketName$ é o nome da estrutura definida pelo usuário. A forma de acesso é denotada por $modo(P_i)$, com $modo(P_i) \in \{in, out, inout\}$.

F_p consiste do conjunto de operações providas pelo nó de computação, que são dependentes do comportamento nó.

F_R consiste do conjunto de operações requeridas pelo nó de computação, com $F_R \subseteq F_p'$, onde F_p' representa o conjunto das funcionalidades providas pelo nó de computação interconectado ao nó que está sendo descrito.

A interface de um nó de computação especifica o que está visível, para o mundo externo, da sua estrutura interna. Um nó de computação pode possuir mais de uma interface.

Os processos providos por um nó de computação são os processos descritos no seu comportamento ou os processos que são implementados na sua interface. Por exemplo, a memória M25P40 provê a operação de leitura de um byte, dado um endereço. Esta operação só existe em sua interface. Ela é implementada através de uma operação de leitura interna que acessa a sua matriz de armazenamento.

5.3 Comunicação

Como já foi dito anteriormente, os elementos de comunicação permitem que dois componentes comuniquem-se e sincronizem-se. A comunicação na linguagem é ponto a ponto, ou seja, em LACCES não existe a idéia de barramento e nem as primitivas que são necessárias para o seu funcionamento: detecção de colisão, prevenção de corrupção de dados, etc.

Nesta linguagem, a comunicação é representada através de dois conceitos distintos: **canal** e **conector**.

Definição 5.5: Um **canal** é um *template*, ou seja, uma macro parametrizada, para a especificação de padrões de comunicação recorrentes.

Os *templates* definem estruturas sintáticas que podem ser expandidas com o objetivo de produzir novas declarações. Eles são utilizados através de instanciação, fornecendo-se os parâmetros corretos.

Definição 5.6: Um **conector** é uma entidade abstrata que corresponde a uma instância de um canal.

Pode-se dizer que em uma modelagem orientada a objetos, os canais correspondem às “classes” e os conectores correspondem aos “objetos”.

Em LACCES, essa diferenciação foi utilizada para permitir o reuso dos elementos de comunicação. Isso é possível devido ao compartilhamento de predicados entre os mesmos tipos de elementos. Por exemplo, os conectores do tipo SPI sempre possuem os predicados *mode* e *data rate*. Entretanto, esse compartilhamento não ocorre para os elementos de computação. Por exemplo, uma memória Flash pode ou não possuir o comportamento *lock*. A Figura 24 apresenta um exemplo de *template* de um canal SPI e como é feita a sua instanciação.

```

Channel SPI ( type, time, rate, format, num ) {
    role in MOSI_sender : type;
    role out MISO_sender : type;
    role out MOSI_receiver : type;
    role in MISO_receiver : type;
    RoleBindings{
        MOSI_sender to MOSI_receiver;
        MISO_sender to MISO_receiver;
    }
    send(type data) assoc MOSI_sender, MISO_receiver;
    type receive() assoc MOSI_receiver, MISO_sender;

    word_time = time;
    data_rate = rate;
    data_format = format;
    wire = num;
}

Connectors{
    ...
    uses SPI(SPI_Pack, 5 ns, 20 Mbps, MSB, 3) as SPI_connector;
}

```

FIGURA 24– *Template de um canal SPI.*

De acordo com a definição apresentada do sistema, um conector pode ser definido da seguinte maneira:

Definição 5.7: Um **conector** é um nó de comunicação $C_{ni} \in C_n$. Ele é um elemento passivo que possui um tipo, denotado por $\text{tipo}(C_{ni})$, onde $\text{tipo}(C_{ni}) \in \{SPI, I2C, UART, Analogue, Sw_Parallel, Hw_Parallel, Sw_FIFO, Hw_FIFO, Semaphore, IPC, Virtual\}$. Os tipos dos nós de comunicação podem ser subdivididos em três subconjuntos, com $\text{tipo}(C_{ni}) = \{C_{nH} \cup C_{nS} \cup C_{nHS}\}$, onde:

C_{nH} consiste do conjunto dos nós de comunicação entre componentes de hardware, onde $\text{tipo}(C_{nH}) \in \{SPI, I2C, UART, Analogue, Hw_Parallel, Hw_FIFO\}$;

C_{nS} consiste do conjunto dos nós de comunicação entre componentes de software, onde $\text{tipo}(C_{nS}) \in \{Sw_Parallel, Sw_FIFO, Semaphore, IPC\}$;

C_{nHS} consiste do conjunto dos nós de comunicação hierárquicos e que podem ser utilizados para a comunicação entre componentes de hardware e de software, onde $\text{tipo}(C_{nHS}) \in \{Virtual\}$.

Assim como os tipos dos componentes, os tipos dos conectores foram escolhidos tendo-se em vista os elementos de comunicação descritos no **Capítulo 3**.

No **Capítulo 3** foram definidas que as primitivas de uma FIFO ou de uma estrutura paralela podem ser utilizadas para a comunicação entre componentes de hardware e de software. Entretanto, como os níveis de abstrações desses domínios são distintos, é necessária uma construção sintática que seja capaz de representar essa diferença, para que isso não fique implícito na linguagem e precise ser inferido pelo contexto. Além disso, um conector mais refinado pode ser necessário para a conexão entre componentes de hardware do que entre componentes de software. Por isso, são utilizados dois tipos de conectores para cada elemento de comunicação: FIFO de software, FIFO de hardware, Paralelo de software e Paralelo de hardware. Pode-se dizer que esses tipos definidos são especializações dos tipos primitivos FIFO e *Parallel*. Eles possuem o mesmo conjunto de primitivas comportamentais, estruturais e temporais definidos por esses tipos primitivos. Entretanto, as primitivas físicas só estão presentes para os conectores FIFO e Paralelo de hardware, os quais possuem uma realização física.

Um nó de comunicação C_{ni} é constituído de um corpo e de uma interface.

Definição 5.8: O **corpo** de um nó de comunicação é um conjunto B_{cnn} , com $B_{cnn} = (B_{cn} \cup B_{tn} \cup B_{en} \cup B_{fn})$, onde:

B_{cn} consiste do conjunto de operações básicas que o nó de comunicação realiza, representando o seu comportamento visível externamente, isto é, o comportamento que é visível e que pode ser observado pela arquitetura que contém esse nó. Essas operações são descritas através de métodos;

B_{tn} consiste do conjunto de propriedades temporais associadas ao nó de comunicação;

B_{en} consiste do conjunto de propriedade estruturais associadas ao nó de comunicação;

B_{fn} consiste do conjunto de propriedade físicas associadas ao nó de comunicação.

A interface de um nó de comunicação é constituída por um conjunto de *roles*. Os *roles* definem unidades semânticas visíveis através das quais os conectores mediam a interação entre os componentes.

Definição 5.9: A **interface** de um nó de comunicação é um conjunto R , com $R = \{R_P \cup R_B\}$, onde:

R_P define um conjunto de *roles* que representam o ponto de interação do nó com o meio externo. Cada *role* $R_i \in R_P$ é caracterizado por um nome, por um tipo e por uma forma de acesso. O tipo é denotado por $\text{tipo}(R_i)$, com $\text{tipo}(R_i) \in \{int, bit, byte, int[x], bit[x], byte[x], analogue, \langle PacketName \rangle\}$, onde $x > 0$ e *PacketName* é o nome da estrutura definida pelo usuário. A forma de acesso é denotada por $\text{modo}(R_i)$, com $\text{modo}(R_i) \in \{in, out, inout\}$.

R_B consiste no conjunto que define as conexões internas existentes entre os *roles* do nó de comunicação, $R_B = (R_{B'} \times R_{B''})$, onde $R_{B'} \cup R_{B''} \subseteq R_P$.

Um nó de comunicação é hierárquico, podendo ser constituído de outros nós de comunicação e de computação. Um nó hierárquico é definido utilizando o tipo *Virtual*. O nome *Virtual* foi escolhido para deixar claro que não se trata de uma implementação real. Conectores virtuais são implementados através de conectores e componentes mais básicos e não hierárquicos, os quais são chamados nós folhas. As folhas devem ser dos tipos pré-definidos em LACCES.

Um nó de comunicação hierárquico em LACCES pode ser usado para representar a comunicação entre elementos de hardware e software. Pode-se dizer que ele realiza o papel de um adaptador de comunicação, ou seja, permite a comunicação entre componentes que possuem interfaces distintas.

A Figura 25 ilustra o papel de um conector virtual. Nesta figura, o componente de software *DataManager* que possui uma interface FIFO, necessita comunicar com uma memória paralela *Mem*. Para permitir essa comunicação foi utilizado um conector virtual hierárquico constituído dos seguintes elementos:

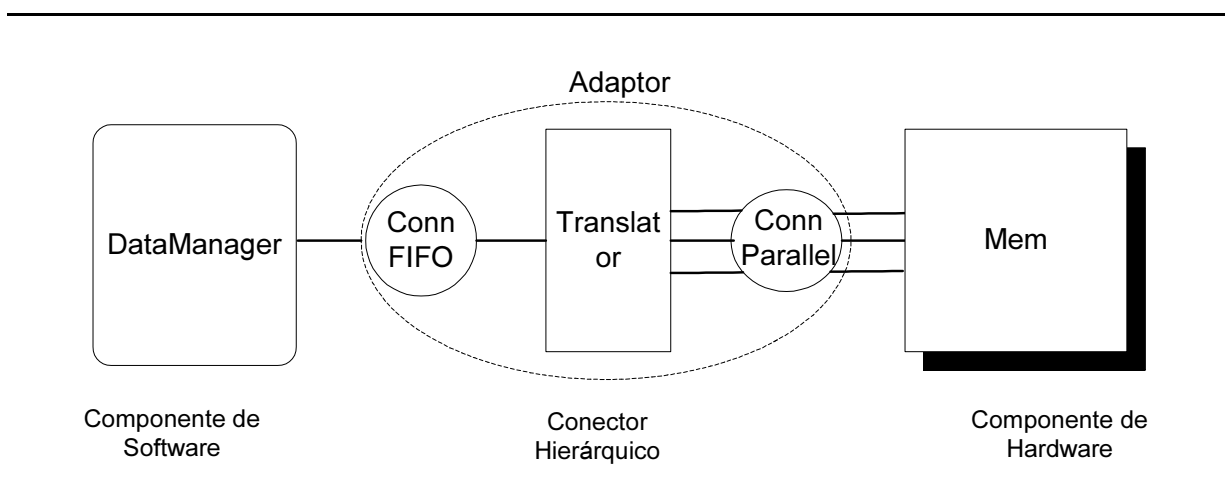


FIGURA 25 – Conector Virtual.

- O conector *ConnFIFO* do tipo *Sw_FIFO*.
- O conector *ConnParallel* do tipo *Hw_Parallel*.
- O componente genérico *Translator* que realiza o processamento e a tradução dos dados recebidos através do *ConnFIFO*, e os envia para o conector *ConnParallel*.

A comunicação ocorre quando algum componente aciona um comportamento do conector. Esse acionamento se dá através da chamada de um método de comunicação, por exemplo, *send* ou *receive*, provido pelo *role* do conector que está ligado à porta do componente.

No exemplo acima, o componente de software, *DataManager*, aciona o método *write(x)* do conector *ConnFIFO*. O conector armazena o dado a ser enviado e gera um evento, notificando o componente *Translator* que um dado foi recebido e está pronto para ser lido. Esse evento decorre da alteração do valor armazenado no conector. O componente *Translator* aciona o método *read()* do conector *ConnFIFO*, para ler o dado. Esse dado é então processado e traduzido para ser transmitido paralelamente. A transmissão paralela ocorre da mesma forma: o componente *Translator* aciona o método *send(x)* do conector paralelo para o envio dos dados, em seguida o conector notifica a memória *Mem*, que faz a leitura dos dados através o método *receive()*. A Figura 26 apresenta o código do conector.

5.3.1 Refinamento da Comunicação

No processo de transição de uma arquitetura abstrata para uma arquitetura mais detalhada, não é só refinada as suas estruturas internas - seus tipos de dados ou temporizações - mas também os seus elementos: componentes e conectores.

O refinamento da comunicação consiste em expandir um elemento abstrato na sua implementação real. No exemplo 26, a comunicação abstrata é representada através do conector *Virtual*. Esse conector abstrato é refinado em uma implementação no MCU. A Figura 27 ilustra o conector refinado.

Os elementos de software desse sistema embutido são implementados na CPU do MCU – neste caso, o componente *DataManager* e o conector *ConnFIFO*. Considerando-se que no refinamento, o projetista decidiu que o componente *Translator* é de software, ele também será implementado na CPU.

Para fazer a “ponte” entre os componentes de software e os componentes de hardware são utilizados *bindings*, que associam as portas do componente de software às portas da CPU. Neste exemplo, todos os dados enviados e recebidos pelo componente *Translator* para a memória pelo serão através das portas da CPU.

```

Connector Adaptor : Virtual{
  role in myfifo : int;
  role out myparallel[3]: bit;
  Representation {
    System Adaptor_Details{
      Component Translator : Generic{
        Interface TranslatorInterface{
          port in fifo_in: int;
          port out parallel_out[3]: bit;
          Provides{
            Translator();
          }
        }
        Translator() sensitive fifo_in
        Implementation{
          in code;
          FileType executable;
          ImplType none;
          Language SystemC;
        };
      };
    }
    Connector ConnFIFO : Sw_FIFO{
      role in myfifo : int;
      role out datafifo : int;
      RoleBindings {
        myfifo to datafifo;
      }
      int read() assoc datafifo;
      write(int x) assoc myfifo;
    }
    Connector ConnParallel : Hw_Parallel{
      role in sender[3] :bit;
      role out receiver[3] :bit;
      RoleBindings {
        sender[0 ... 2] to receiver[0 ... 2];
      }
      send(bit x) assoc sender[0 ... 2];
      bit receive()assoc receiver[0 ... 2];
    }
    Attachments{
      ConnFIFO.datafifo to Translator.fifo_in;
      Translator.parallel_out[0...2] to ConnParallel.sender[0 ... 2];
    }
  }
  Bindings{
    ConnFIFO.myfifo to myfifo;
    ConnParallel.receiver[0 ... 2] to myparallel[0 ... 2];
  }
}
}

```

FIGURA 26– Descrição em LACCES do conector.

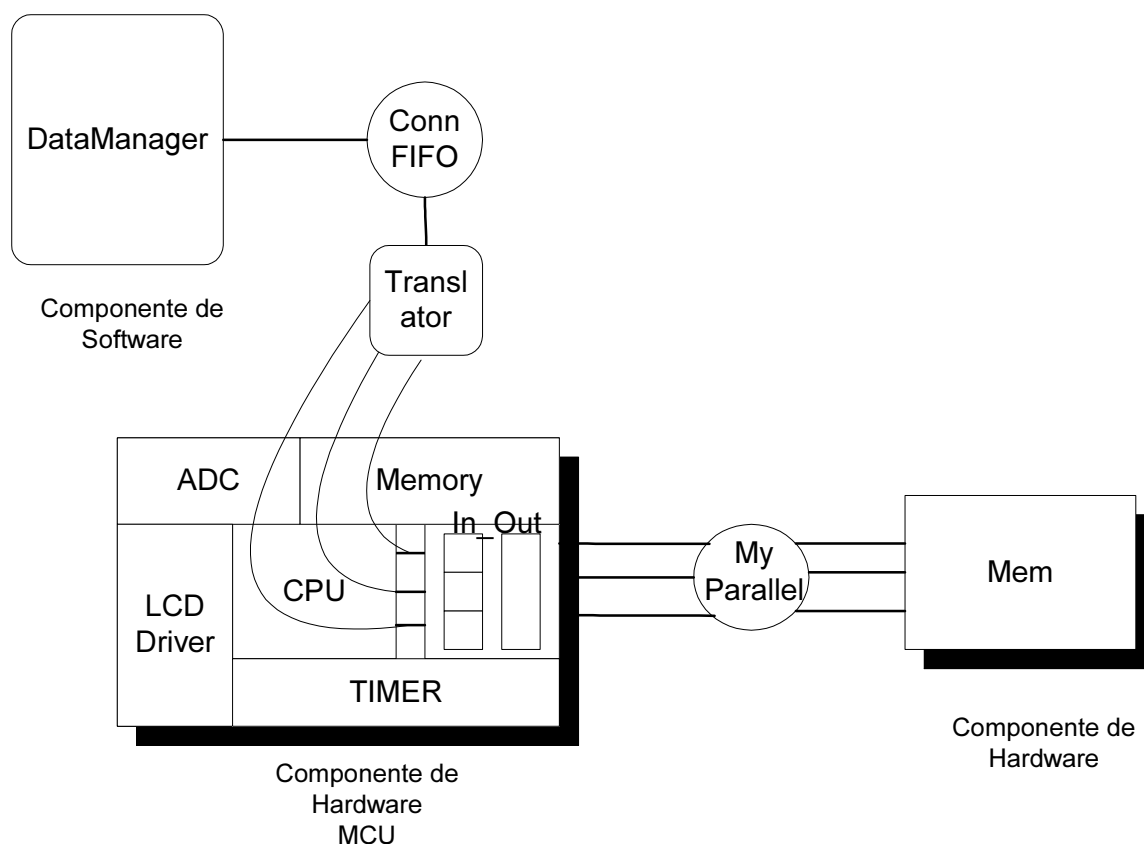


FIGURA 27– Implementação da comunicação entre um software e uma memória.

O conector *ConnParallel* foi refinado em um componente *In_Out*, que representa a interface paralela do MCU, e no conector paralelo *MyParallel*. O componente *In_Out* é constituído de um registrador, que é responsável pelo armazenamento dos dados a serem transmitidos, e de um componente genérico, que realiza a leitura dos dados armazenados no registrador e os envia para o conector *MyParallel*.

A Figura 28 ilustra a descrição resumida em LACCES deste exemplo.

```

System Sensor {
  Component Translator : Module{
    Interface TranslatorInterface{
      port in fifo_in: int;
      port out parallel_out[3]: bit;
      ...
    }
    ...
  }
  Component MSP : MCU{
    Interface MCUInterface{
      port inout TI[3]: bit;
      port in r : bit;
      ...
      Provides {
        reset() sensitive r;
        ...
      }
    }
    Representation {
      System MCU_Details{
        Component MSPCPU : CPU{
          Interface CPUInt{
            port inout p[3]: bit;
            ...
          }
          ...
        }
        ...
      }
    }
  }

  Bindings{
    Translator.parallel_out[0 ... 2] to MSP.MSPCPU.p[0 ... 2];
  }
}

```

FIGURA 28– Mapeamento do módulo de Software na CPU.

Capítulo 6

Representação Interna

Este capítulo apresenta a descrição da representação interna utilizada por LACCES. Essa representação é fortemente baseada em SDS, tanto que ela é denominada LSDS (*LACCES-SDS*).

LSDS é uma versão de SDS para o mundo de sistemas embutidos. Como foi apresentado no **Capítulo 2**, SDS é uma representação que consiste de um conjunto de classes que modelam cada um dos quatro domínios de uma especificação de um sistema heterogêneo. SDS permite que sejam explicitadas as características em comum e as diferenças dos objetos que compõem o sistema.

A semântica dos tipos das estruturas de representação do modelo SDS foi mantida, entretanto, foram modificados alguns dos seus atributos. Essas alterações foram realizadas de tal maneira que fosse preservada a consistência e a compatibilidade de LSDS com a versão inicial de SDS.

Este capítulo é estruturado da seguinte maneira:

- A seção 6.1 descreve a semântica relacionada aos principais elementos da representação interna do sistema;
- A seção 6.2 apresenta a semântica detalhada das classes e atributos do domínio do fluxo de processo;
- A seção 6.3 apresenta a semântica detalhada das classes e atributos do domínio do fluxo de tempo e seqüência;
- A seção 6.4 apresenta a semântica detalhada das classes e atributos do domínio da estrutura lógica;
- A seção 6.5 apresenta a semântica detalhada das classes e atributos do domínio da estrutura física;
- A seção 6.6 apresenta a semântica detalhada das classes relacionadas com a interconexão entre os elementos de comunicação e computação.

6.1 Semântica dos Objetos

Cada componente e conector descrito em LACCES pode ser representado em LSDS em 4 domínios: domínio do fluxo de processo (PFD), domínio da estrutura lógica (LSD), domínio da estrutura física (PSD) e domínio do fluxo de tempo e sequência (TSD).

Utilizando a mesma notação formal que foi empregada para definir a semântica dos conceitos de LACCES, o domínio do fluxo de processo possui três classes de nós de computação e uma classe de nó de conexão, os outros domínios possuem uma classe de nó de computação e uma de conexão. As classes de nós de computação são agrupadas na classe pai *Model* e as classes de nós de conexão são agrupadas na classe pai *Link*.

Model: Super-tipo dos objetos cuja semântica está relacionada com a execução de operações. A partir desta classe são derivados os tipos dos nós de computação dos quatro domínios: *Node* para o PFD, *Module* para o LSD, *Range* para o TSD e *Block* para o PSD. A Figura 29 ilustra o modelo de classes utilizando a notação UML [9].

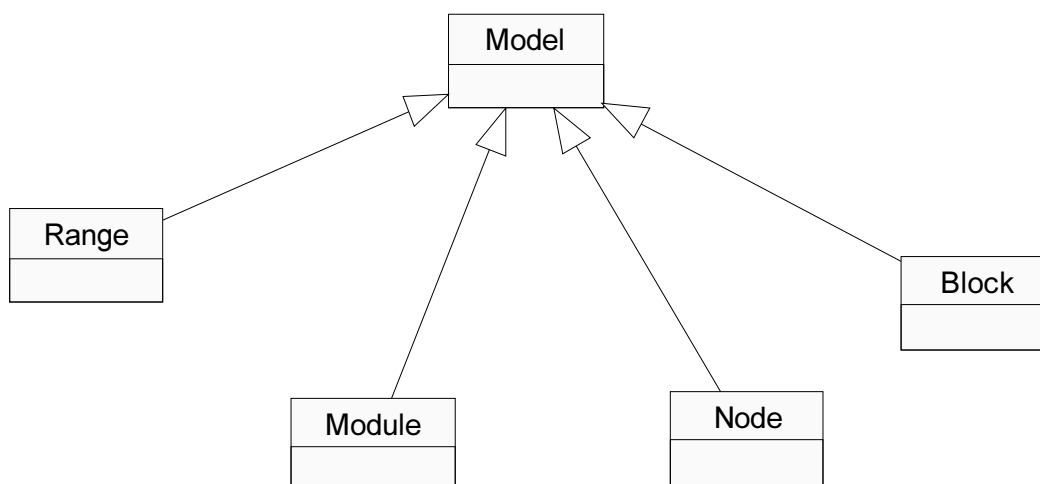


FIGURA 29 – Diagrama das classes relacionadas à computação.

Link: Super-tipo dos objetos cuja semântica está relacionada com a conexão. A partir dessa classe são derivados os tipos dos nós de conexão dos quatro domínios: *Information* para o PFD, *Point* para o TSD, *Carrier* para o LSD e *Net* para o PSD. A Figura 30 ilustra o modelo de classes utilizando a notação UML.

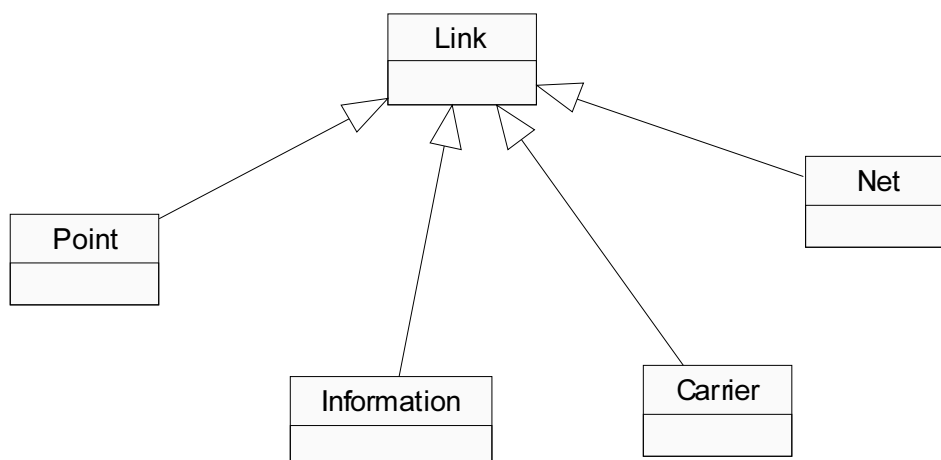


FIGURA 30– Diagrama das classes relacionadas à comunicação.

Reference: Super-tipo cuja semântica está relacionada com a interconexão dos objetos de *Model* e *Link*. Esta classe foi introduzida para modelar e representar explicitamente a ligação entre os elementos de comunicação e computação de cada domínio. Em um modelo de grafos, os objetos da classe *Reference* são representados como um conjunto de arestas. A partir dessa classe são derivadas as seguintes subclasses: *RefNet* e *RefPin*.

A classe *RefNet* define uma lista de “redes” que são utilizadas para interconectar os objetos de *Model* e *Link* através de pinos. Pinos são representados pela classe *RefPin* e são necessários para modelar as entradas e saídas de cada elemento. Esse termo refere-se aos pinos de uma representação física somente se o objeto descrito for do domínio físico, nos outros domínios esse conceito refere-se aos pontos de entrada e saída. As “redes” existem para os domínios do fluxo de processo, estrutura lógica e estrutura física. A Figura 31 ilustra o modelo de classes utilizando a notação UML.

Para descrever o mapeamento entre os domínios são utilizados *bindings*. Existem dois tipos de *bindings*: no primeiro, os objetos do domínio comportamental são relacionados aos objetos dos domínios de tempo e de estrutura. No segundo, os objetos do domínio estrutural são relacionados aos objetos do domínio físico. Em LSDS, esses mapeamentos consistem de atributos das classes *Node* e *Module*. Em SDS, eles consistem de estruturas de dados específicas.

Como será visto nas seções seguintes, os objetos do PFD, TSD e LSD podem ser hierárquicos e os do PSD não. A hierarquia define uma árvore, onde a raiz representa o elemento de mais alto nível.

Os *bindings* entre elementos de diferentes domínios devem respeitar a hierarquia, ou seja, um elemento de mais alto nível não pode ter um sucessor mapeado em um elemento mais abstrato do que aquele em que ele está mapeado.

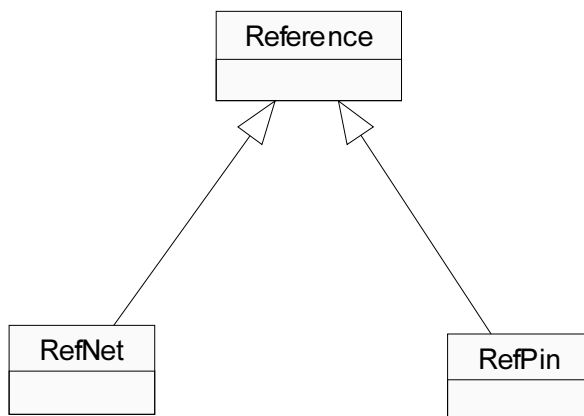


FIGURA 31 – Classes relacionadas à interconexão das classes de comunicação e computação.

6.2 Domínio de Fluxo de Processo

O domínio do fluxo de processo é constituído da classe pai *Node* e das classes filhas *Channel*, *Process* e *ComposedProcess*. A Figura 32 ilustra o modelo de classes para esse domínio. Esse modelo e todos os outros modelos de classes apresentados neste capítulo abstraem atributos que são específicos da implementação, tornando-os mais fáceis de serem entendidos.

A classe *Node* contém a seguinte informação comum à suas classes filhas:

- *Type*: define o tipo do nó. Seu valor corresponde ao tipo do componente ou do conector definido na descrição da arquitetura.
- *BB_Proc*: mapeamento do comportamento (*Node*) na estrutura (*Module*).
- *B_Behavior*: mapeamento do comportamento (*Node*) no tempo (*Range*).
- *Port_ref*: representa o conjunto de portas que acionam o seu comportamento.

Um *ComposedProcess* é constituído de processos genéricos (*Process*) que podem ser especializados em processos de hardware (*Hardware*) ou de software (*Algorithm*). Essa classe possui a seguinte informação:

- *Proc_ref*: Lista de referência às instâncias de processos que o compõem.
- *Channel_ref*: Lista de referência às instâncias de canais que interconectam as instâncias de processos que o compõem.
- *Info_ref*: Lista de informações transferidas entre os processos que o compõem.

- *Con_ref*: Lista de referência às instâncias de *Channels* dos seus sub-connectores.
- *Comp_ref*: Lista de referência às instâncias de *ComposedProcesses* dos seus sub-componentes.
- *Mode*: modo de configuração do *ComposedProcess*.
- *Config*: tupla que determina a área da memória que não pode ser apagada e nem escrita.

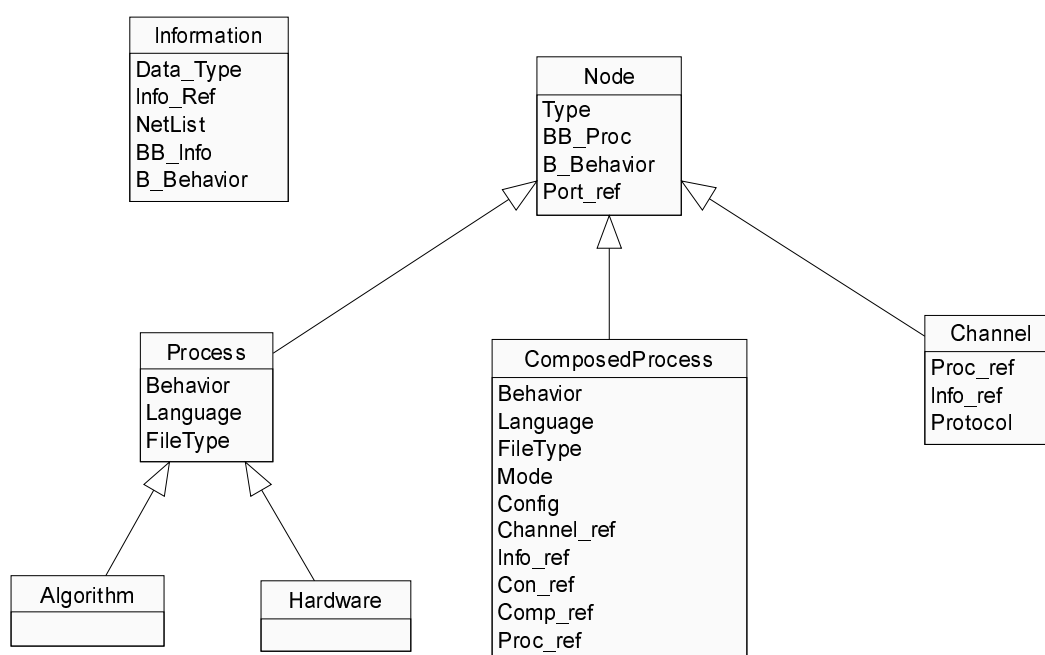


FIGURA 32– Classes do domínio de fluxo de processo.

Um objeto da classe *Process* representa um processo genérico que não possui uma implementação definida, ou seja, ainda não se sabe se ele será um processo de hardware ou de software.

Os seguintes atributos são comuns à classe *Process* e à classe *ComposedProcess*.

- *Behavior*: nome do arquivo que contém a implementação do processo.
- *Language*: nome da linguagem utilizada para implementar o processo.
- *FileType*: indica se o arquivo que contém a implementação do processo é um arquivo com código fonte ou executável.

Um objeto da classe *Algorithm* representa um processo que possui uma especificação

funcional e que não possui uma implementação física. Uma especificação funcional é geralmente obtida através da utilização de uma metodologia *top-down*.

Um objeto da classe *Hardware* representa um processo que será implementado fisicamente e que é obtido através de uma metodologia *bottom-up*.

Um objeto da classe *Channel* representa a comunicação e a sincronização entre dois processos, podendo possuir uma especificação funcional e uma implementação física. Canais podem ser persistentes, sendo utilizados para modelar o armazenamento de informações. Um canal persistente requer um comportamento específico para lidar com objetos persistentes. Esse comportamento inclui a habilidade de recuperar e armazenar informações. Uma memória é um exemplo de canal persistente em LDS.

A classe *Channel* possui os seguintes atributos:

- *Proc_ref*: Lista de referência às instâncias de processos que compõem o canal.
- *Info_ref*: Lista de referência às instâncias de informações que compõem o canal.
- *Protocol*: Nome do protocolo utilizado pelo canal.

A classe *Information* corresponde à informação transferida entre dois processos através de um canal. Essa classe possui os seguintes atributos:

- *Data_Type*: Tipo do dado transmitido.
- *Info_Ref*: Lista de referência às informações que a compõem.
- *NetList*: Lista de objetos do tipo *PFNet* que conectam a informação a instâncias de *ComposedProcess*, *Process* ou *Channel*.
- *BB_Info*: Mapeamento da informação (*Information*) na estrutura (*Carrier*).
- *B_Behavior*: Mapeamento da informação (*Information*) no tempo (*Range*).

Uma instância da informação pode ser hierárquica, contendo instâncias de sub-informações. Essa hierarquia denota a estrutura do tipo de dados.

Cada componente descrito em LACCES é mapeado em LDS em um *ComposedProcess*. Um componente possui uma interface e um comportamento. A interface é mapeada em um *Channel*, já que ela está associada ao protocolo de comunicação. O comportamento é mapeado em um *ComposedProcess*. A única exceção é para a memória que representa uma entidade persistente, nesse caso, o seu comportamento também é mapeado em um *Channel*.

Cada conector descrito em LACCES é mapeado em um *Channel*. A funcionalidade do conector é mapeada em processos que o compõem. Para conectores do tipo *Virtual* é feita a planarização dos seus sub-elementos.

6.3 Domínio do Fluxo de Tempo e Seqüência

O domínio do fluxo de tempo e seqüência define uma ordem parcial nas operações e informações trocadas entre os elementos do PFD. Esse domínio é composto de dois tipos de objetos: intervalos (*Ranges*) e pontos (*Points*). Intervalos representam intervalos de tempo. Pontos servem como delimitadores de intervalos de tempo e podem possuir uma semântica especial. Os pontos possuem uma duração infinitesimal e conseqüentemente são considerados instantâneos. A Figura 33 ilustra o modelo de classes para este domínio.

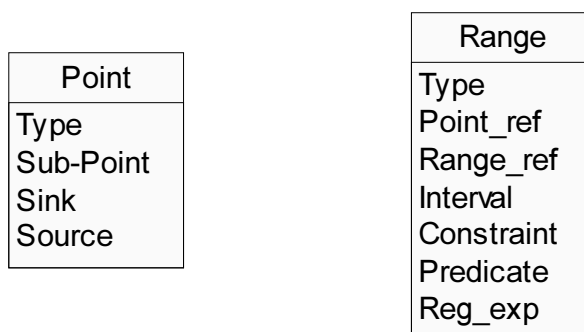


FIGURA 33– Classes do domínio de fluxo de tempo e seqüência.

Um objeto do tipo *Range* possui o seguinte conjunto de atributos:

- *Type*: Existem quatro tipos de intervalos:
 - o *Causal*: Separa dois eventos instantâneos. Todos os intervalos de um sistema podem ser considerados causais, a razão para isso advém da necessidade de ser capaz de representar explicitamente um intervalo de tempo desconhecido, o qual, usualmente, é capturado durante a especificação de um sistema.
 - o *Constraint*: Representa um intervalo de tempo com uma duração especificada pelo projetista. Esse tempo é um requisito do projeto e pode ser definido usando um operador relacional. Intervalos do tipo *constraint* capturam uma informação *top-down*.
 - o *Delay*: Similar a um intervalo do tipo *Constraint*, porém seus atributos são conhecidos e medidos. Esses atributos podem ser obtidos através do *datasheet* ou outro documento que detalha as características do dispositivo físico. Intervalos do tipo *delay* capturam uma informação *bottom-up*.

- o *Loop*: Corresponde a um intervalo do tipo *causal* que se repete, como se fosse um *loop for*.
- *Point_ref*: Lista de referência às instâncias de pontos que o compõem.
- *Range_ref*: Lista de referência às instâncias de intervalos que o compõem.
- *Interval*: Predicado que define a duração do intervalo.
- *Constraint*: Para intervalos do tipo *Constraint*, determina qual operador relacional que se aplica: igual, maior ou igual, maior, menor, menor ou igual.
- *Predicate*: Indica se o intervalo está associado a um evento síncrono ou assíncrono.
- *Reg_exp*: Determina o número de vezes que o intervalo se repete.

Um objeto do tipo *Point* possui a seguinte informação:

- *Type*: Existem cinco tipos de pontos:
 - o *or_fork* e *or_join*: Pontos do tipo *or-fork* e *or-join* são usados para representar uma execução condicional. Um ponto do tipo *or-fork* possui um intervalo de entrada e dois de saída. Cada intervalo de saída está associado a uma condição. O fluxo de execução segue o intervalo cuja condição seja satisfeita. O ponto do tipo *or-join* possui dois intervalos de entradas e um de saída.
 - o *and_fork* e *and_join*: Pontos do tipo *and-fork* e *and-join* representam execução paralela. Se não existe nenhum predicado associado a um intervalo de saída de um *and-fork*, impedindo a sua execução, ele é executado. O predicado é associado a um intervalo para modelar eventos síncronos ou assíncronos, os quais podem ocorrer a qualquer momento no sistema.
 - o *simple*: Pontos do tipo simples representam uma execução seqüencial sem alteração do fluxo de controle.
- *Sub_point*: Referência ao primeiro ou ao último ponto de um sub-intervalo.
- *Sink*: Lista de intervalos que sucedem o ponto corrente.
- *Source*: Lista de intervalos que antecedem o ponto corrente.

6.4 Domínio da Estrutura Lógica

O LSD captura a estrutura lógica ou a topologia do sistema. Um objeto neste domínio não possui nenhum significado além da sua organização lógica especificada pelo projetista utilizando uma metodologia *top-down* ou imposta pelos requisitos ou restrições físicas em uma metodologia *bottom-up*. O significado é associado aos objetos deste domínio quando eles são ligados aos objetos do PFD. Na verdade, o LSD deve ser visto como uma ponte entre o comportamento e a implementação física.

Esse domínio é constituído das classes principais: *Module* e *Carrier*. *Modules* representam aspectos estruturais dos processos e canais. *Carriers* representam aspectos estruturais da informação.

Modules podem possuir persistência e podem ser usados para modelar registradores e memórias. A Figura 34 ilustra o modelo de classes para este domínio.

Um objeto do tipo *Module* possui os seguintes atributos:

- *Carrier_ref*: Lista de referência às instâncias de *carriers* que o compõem.
- *Module_ref*: Lista de referência às instâncias de módulos que o compõem.
- *Persistency*: Tupla onde o primeiro elemento indica se o modulo é ou não persistente e o segundo representa a sua capacidade.
- *Type*: Indica o tipo da arquitetura utilizada pelo módulo.
- *BR_Proc*: Mapeamento da estrutura (*Module*) na implementação física (*Block*).
- *Org*: Lista de tuplas que armazenam a organização do módulo.
- *Matrix*: Para um módulo do tipo LCD, indica a organização de segmentos e comuns necessários para a exibição dos dados.
- *Accuracy*: Indica a precisão do módulo.
- *NumDef*: Informação numérica do módulo, como por exemplo, o número de teclas de um *Keypad* ou o *Prescaler* de um *Timer*.
- *Speed*: Taxa de operação do módulo.
- *Mode*: Para memórias internas ao microcontrolador, indica se ela é usada para o armazenamento de dados, programa ou ambos.

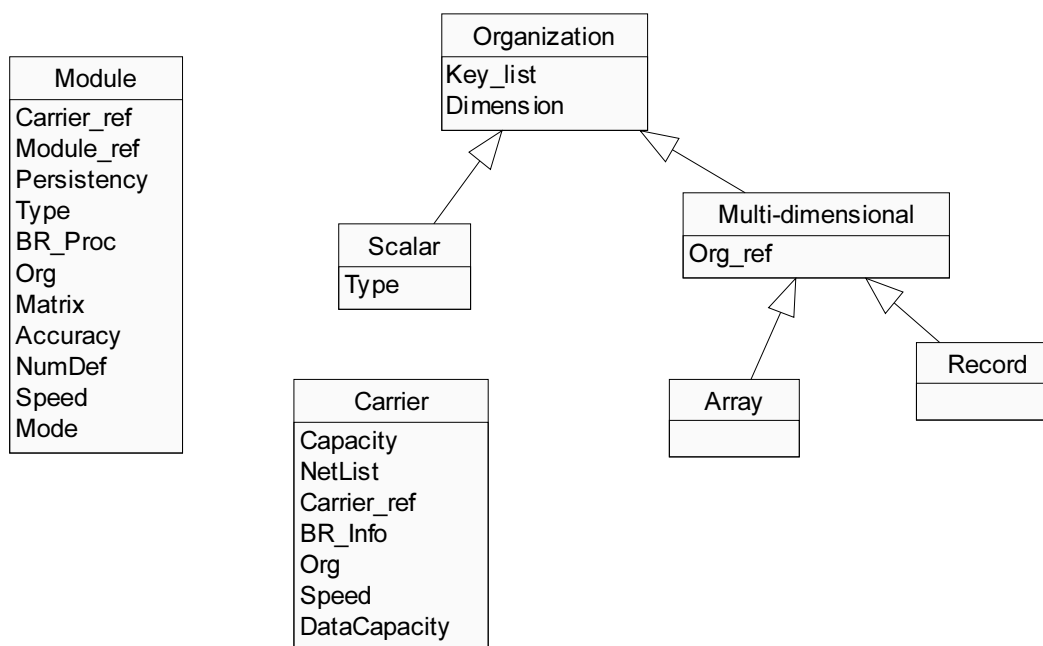


FIGURA 34– Classes do domínio de estrutura lógica.

A classe *Carrier* possui os seguintes atributos:

- *Capacity*: Capacidade de transferir informação em termos de *bandwidth*.
- *NetList*: Lista de objetos do tipo *LSNet* que conectam um *carrier* a instâncias de módulos.
- *Carrier_ref*: Lista de referência às instâncias de *carriers* que o compõem.
- *BR_Info*: mapeamento do *Carrier* na implementação física (*Net*).
- *Org*: Referência a um objeto do tipo *Organization*.
- *Speed*: Para canais analógicos, indica a frequência do sinal.
- *DataCapacity*: Taxa de transferência dos dados.

A classe *Organization* representa a organização do *Carrier* nos tipos *Scalar* ou *Multi-dimensional* (*Array* e *Record*). Essa classe possui os seguintes atributos:

- *Dimension*: Lista de tuplas que definem o intervalo do tipo. Os elementos da tupla indicam o início e o fim do intervalo.
- *Key_list*: Nome que identifica cada elemento.

O tipo *Scalar* é um tipo primitivo e não possui hierarquia. Ele contém o atributo *type*.

O tipo *Multi-dimensional* possui o atributo *Org_ref* que é uma lista de referência aos

objetos do tipo *Scalar* ou *Multi-dimensional*. Ele é constituído dos seguintes sub-tipos: *Array* e *Record*. Um *Array* é um conjunto ordenado de elementos do mesmo tipo. Um *Record* é um conjunto de elementos de tipos diferentes.

6.5 Domínio da Estrutura Física

O domínio da estrutura física representa restrições físicas impostas pelos requisitos do projeto e informações físicas específicas que estão descritas no *datasheet* do dispositivo. Esse domínio é constituído de dois tipos de objetos: *Block* e *Net*. *Blocks* representam entidades físicas e são considerados como “caixas pretas” sem hierarquia, tais como *chips*. Isso torna clara a separação existente entre o projeto de um sistema e o projeto de um componente. A representação interna de um componente físico está além do escopo desta dissertação. Objetos do tipo *Net* representam conexões físicas, tais como cabos ou fios. A Figura 35 ilustra o modelo de classes para este domínio.

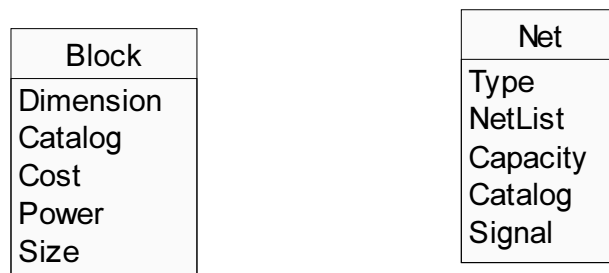


FIGURA 35– Classes do domínio da estrutura física.

Objetos do PSD possuem um grande número de atributos devido a proximidade da implementação física. Entretanto, nem toda informação é capturada pela representação interna, somente a que é relevante para o comportamento do sistema. Além disso, um detalhamento maior do componente pode ser obtido através do seu *datasheet*.

Um objeto do tipo *Block* possui a seguinte informação:

- *Dimension*: Tripla que descreve as suas dimensões físicas.
- *Cost*: Custo monetário.
- *Power*: Tupla que indica a potência dissipada ao executar uma função associada ao *Block*.

- *Catalog*: Informações do *datasheet*.
- *Size*: Para um *Keypad*, representa o número de teclas.

Um objeto do tipo *Net* possui a seguinte informação:

- *Type*: Tipo do elemento de comunicação.
- *Catalog*: Informações do *datasheet*.
- *Capacity*: descreve a capacidade de transferência de informação em termos de *bandwidth*.
- *Signal*: tipo do sinal associado a canais analógicos- corrente ou tensão.
- *NetList*: Lista de objetos do tipo PSNet que conectam um *Net* a instâncias de blocos.

6.6 Classes de Referência

A classe *RefNet* é utilizada para interconectar os objetos da classe *Link* e os da classe *Model*. A Figura 36 ilustra o modelo de classes.

Existem três classes filhas da classe *RefNet*:

- *PFNet*: interconecta *Nodes* do PFD.
- *LSNet*: interconecta *Modules* do LSD.
- *PSNet*: interconecta *Blocks* do PSD.

A classe *RefNet* possui o atributo *source* e *sink* que são referências aos pinos utilizados na interconexão.

Os pinos possuem uma classe pai chamada *RefPin*. Essa classe contém os seguintes atributos:

- *Type*: tipo da informação que o pino pode transportar. Esse valor é obtido através dos tipos das portas e dos *roles* dos elementos de projeto de LACCES.
- *Mode*: define a direção da transferência dos dados, podendo ser de entrada e/ou de saída.
- *RoleBind*: Referência a um pino de interconexão interna de um *Channel*.
- *Bind*: Referência ao pino do elemento pai.
- *NumBinds*: Indica o número máximo de *binds* associados a um pino.

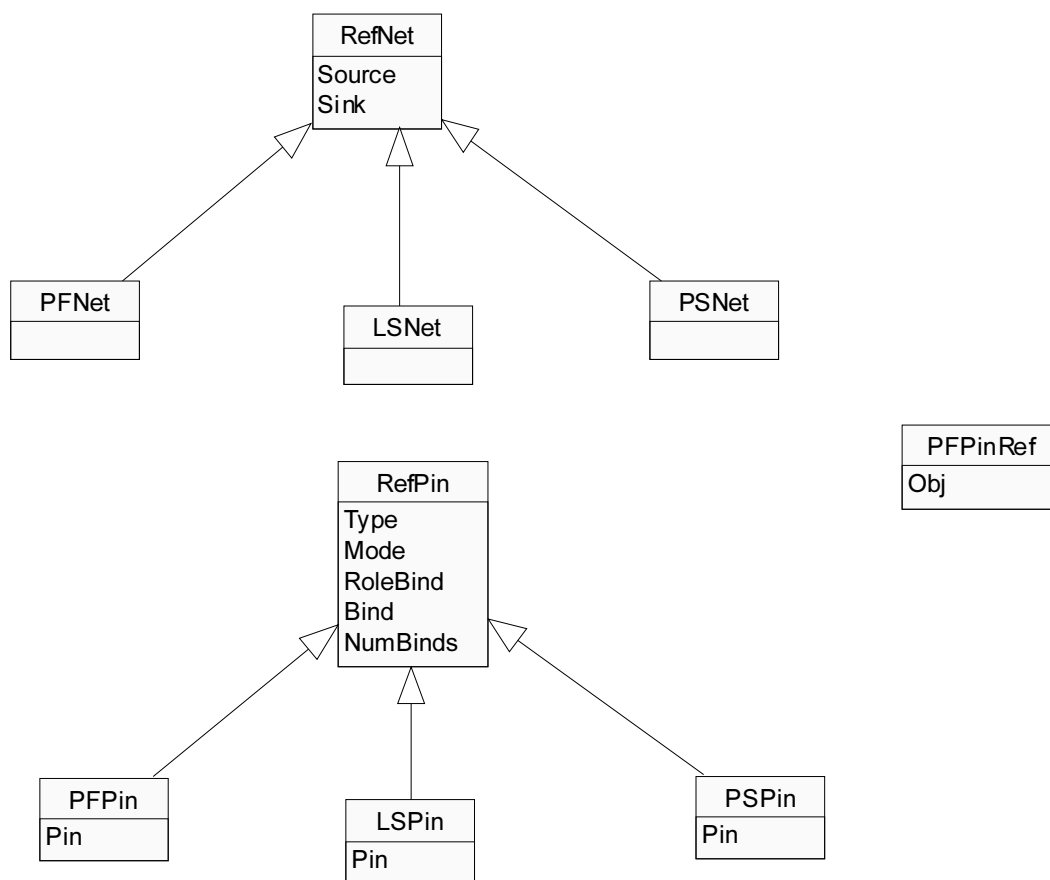


FIGURA 36– Representação do modelo de referência.

Os tipos de pinos são:

- *PFPin*: pino associado ao *Node* do PFD.
- *LSPin*: pino associado ao *Module* do LSD.
- *PSPin*: pino associado ao *Block* do PSD.

Para representar a sensibilidade dos processos, criou-se o tipo *PFPinRef* cujo atributo *Obj* representa uma referência a um *RefPin*.

Capítulo 7

Ambiente de Projeto

A infra-estrutura de LACCES permite que a arquitetura de um sistema embutido seja descrita e validada. O cerne dessa infra-estrutura consiste na representação interna do sistema em LSDS.

As seguintes ferramentas estão disponíveis para o projetista:

- Um **pré-processador** que realiza a conversão de macros e inclusão de arquivos.
- Um **parser** que lê uma descrição textual da arquitetura expressa na linguagem LACCES, e converte para a representação interna LSDS.
- Uma **ferramenta de análise** que avalia as propriedades não funcionais descritas para os componentes e conectores.
- Uma **ferramenta que reporta erros** e alerta os projetistas sobre eventuais problemas.
- Um **gerador de código em SystemC** que gera um arquivo na linguagem SystemC para a simulação do comportamento do sistema embutido.
- Uma **ferramenta de verificação de tipos** que analisa se o projeto está correto em relação aos tipos de elementos e conectores utilizados.

A Figura 37 descreve a arquitetura de LACCES. Como indica a figura, todas as ferramentas do ambiente LACCES compartilham um mesmo componente que corresponde a representação interna do sistema. Esse conjunto básico de ferramentas permite capturar e analisar a especificação da arquitetura de um sistema embutido, avaliar se os projetos estão corretos e reportar qualquer anomalia encontrada durante o processo de avaliação.

Nesta configuração, primeiramente, o ambiente lê e realiza a análise sintática e semântica de uma especificação em LACCES, gerando uma representação interna em LSDS. Em seguida, é realizada a conversão para SystemC. Após a conversão, é feita a análise das propriedades não funcionais descritas para o sistema. Qualquer erro encontrado durante esse processo, é reportado ao projetista.

Tudo isso é feito invocando a ferramenta através de linha de comando no sistema operacional Linux. As saídas são geradas em arquivos.

Essa infra-estrutura de LACCES consiste de 15 módulos de código fonte escrito em C++, 2 especificações em FLEX [43], 1 especificação em YACC [32], e um total de 15 arquivos incluídos (.h). Isto dá um total de aproximadamente 9.500 linhas de código de todos os tipos.

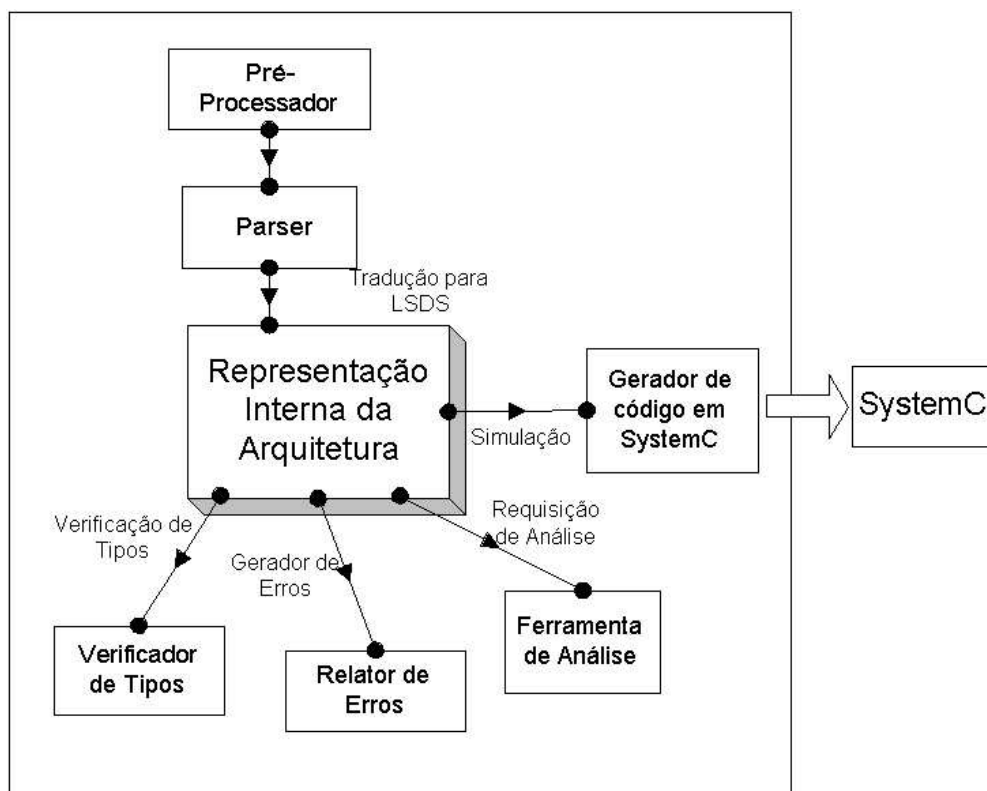


FIGURA 37– Ferramentas disponíveis no ambiente de projeto.

O *parser* foi descrito nos **Capítulos 3, 4, 5 e 6**. Neste capítulo serão descritas as outras ferramentas disponíveis para o projetista.

7.1 Pré-Processador

O pré-processador expande um conjunto de macros pré-definidas na linguagem, facilitando a utilização e o entendimento da mesma. As macros definidas são:

- **#include** <arqname> : inclui o arquivo <arqname> no arquivo corrente.
- **uses** <name> **as** <newname>: permite que uma cópia do componente ou conector previamente descrito, denominado <name>, seja utilizado no arquivo corrente com o novo nome <newname>.

LACCES também permite que sejam definidas macros parametrizadas para os elementos de comunicação. Para isso deve-se utilizar a construção:

Channel <name> (<par₁>, <par₂>, ..., <par_n>), onde <name> é o nome do canal e <par_n> são os parâmetros.

Com a utilização dessas macros, a definição de um sistema se resume a algumas linhas de código, contendo somente a instanciação dos componentes e conectores, e a definição das suas interconexões. A Figura 38 ilustra a utilização das macros para a definição de um sistema constituído de um microcontrolador MSP430 e uma memória SPI.

```
#include units.adl;
#include packet.adl;

System SimpleExample{
  Components{
    uses M25P40 as my_memory;
    uses MSP430 as my_mcu;
  }
  Connectors{
    uses SPI(SPI_Pack, 5 ns, 20 MBps, MSB, 3) as my_connector;
  }
  Attachments{
    my_mcu.MOSI to my_connector.MOSI_sender;
    my_mcu.MISO to my_connector.MISO_sender;
    my_connector.MOSI_receiver to my_memory.D;
    my_connector.MISO_receiver to my_memory.Q;
  }
}
```

FIGURA 38– Descrição de uma arquitetura utilizando macros.

LACCES possui uma biblioteca que contém descrições de arquitetura de alguns componentes e conectores. Os componentes presentes nesta biblioteca incluem uma versão simplificada de um microcontrolador da família MSP430 e uma memória SPI, denominada M25P40. Além disso, também estão disponíveis todos os *templates* de canais. Outros componentes podem e devem ser incluídos nesta biblioteca, aumentando, assim, o reuso dos elementos de projeto.

7.2 Gerador de Código em SystemC

Uma das ferramentas mais importante do ambiente LACCES é o gerador de código em SystemC. Essa ferramenta gera uma especificação executável que é utilizada para simular e validar a funcionalidade do sistema antes da sua implementação, evitando inconsistências e erros durante as fases iniciais de desenvolvimento.

Há muitos benefícios em criar uma especificação executável de um sistema complexo em fases iniciais de desenvolvimento:

- Uma especificação executável evita inconsistências e erros e ajuda a garantir a totalidade da especificação. Isso é verdade, pois quando se cria uma especificação executável, está se criando basicamente um programa que exhibe o mesmo comportamento do sistema. O processo de criar o programa revela inconsistências e erros, e o processo de testar o programa ajuda a garantir a totalidade da especificação.
- Uma especificação executável garante uma interpretação não-ambígua da especificação. Toda vez que os implementadores estão em dúvida sobre o projeto, eles podem executar a especificação para determinar o que é suposto que o sistema esteja fazendo.
- Uma especificação executável ajuda a validar a funcionalidade do sistema antes da implementação começar.
- Uma especificação executável ajuda a criar modelos iniciais de desempenho.
- Uma especificação executável serve como modelo de referência para uma bancada de teste (*testbench*) dos refinamentos do projeto.

SystemC foi escolhida como alvo para a geração de código por causa do seu ambiente de simulação, e por também ser uma ferramenta gratuita, que possui uma boa documentação. Essa linguagem está sendo amplamente utilizada no desenvolvimento de projetos de sistemas integrados de hardware-software em diversos níveis de abstração.

7.2.1 Visão Geral do Processo de Geração de Código

O processo de geração de código se inicia após ter sido realizada a conversão da descrição da arquitetura para a representação interna LSDS. O processo de geração e a estrutura dos arquivos gerados serão descritos nas seções seguintes.

7.2.1.1 Mapeamento LSDS - SystemC

O código SystemC gerado é estruturado em um conjunto de classes C++ e é dividido em duas bibliotecas principais:

- Uma biblioteca é gerada pela ferramenta de conversão, contendo a especificação do comportamento do sistema descrito em LACCES.
- A outra biblioteca contém a implementação das primitivas providas pelo ambiente LACCES.

Quando a geração de código tem sucesso, quatro arquivos são produzidos, como mostrado na Tabela 26. Dois arquivos são da biblioteca de primitivas de LACCES e os outros dois arquivos contêm o código SystemC propriamente dito.

Nome do Arquivo	Descrição
NomeSistema.cpp	Contém a rotina principal, <i>sc_main</i> , onde é feita a instanciação do sistema, declaração de portas e sinais.
NomeSistema.h	Contém a declaração das classes que representam a arquitetura descrita em LACCES, seus processos, portas e sinais. O nome do arquivo depende do nome dado ao sistema na sua descrição inicial.
liblaccs.h	Declaração de todos os métodos primitivos disponíveis na linguagem.
liblaccs.cpp	Implementação dos métodos primitivos declarados em liblaccs.h.

TABELA 26 – Descrição dos arquivos produzidos.

Quando esses arquivos são compilados juntamente com a biblioteca SystemC por ferramentas padrões de C++, uma aplicação executável é gerada. Essa aplicação comporta-se da mesma forma que o sistema descrito em LACCES que a originou e, por isso, pode ser usada para simular e validar o comportamento da arquitetura.

7.2.1.2 Arquivo NomeSistema.cpp

O sistema descrito pode ser dividido em duas partes: o ambiente externo, com o qual ele interage, e o próprio sistema.

Geralmente para simular um sistema, sinais devem ser enviados para ele para se iniciar computações, prover dados, etc. Raramente o sistema não é controlado pelo seu ambiente externo. Por isso, para que a simulação ocorra corretamente, o projetista deve especificar como esse ambiente se comporta em relação aos sinais que são necessários para a sua simulação. O projetista deve fazer essa especificação no método *sc_main* que está contido no arquivo NomeSistema.cpp. O gerador de código apenas escreve nesse método a declaração do sistema, suas portas e sinais. Os estímulos externos devem ser providos pelo projetista.

7.2.1.3 Arquivo NomeSistema.h

O arquivo NomeSistema.h contém a declaração das classes que correspondem aos componentes e conectores descritos na arquitetura do sistema.

7.2.1.4 Arquivo liblaccs.h e liblaccs.cpp

Os arquivos liblaccs.h e liblaccs.cpp contêm a declaração e implementação, respectivamente, das primitivas comportamentais disponíveis na linguagem. Essas primitivas estão descritas no **Apêndice A.3**. Todas as outras funcionalidades, que não se encontram nessa lista, devem ser providas pelo projetista, para que a simulação ocorra corretamente.

Durante a conversão para a representação interna, a ferramenta gera uma listagem de todas as funcionalidades que não estão disponíveis. O projetista deve avaliar esse conjunto de dados e prover as informações necessárias.

A implementação das primitivas comportamentais foi simplificada ao máximo, para que somente seja validada a funcionalidade do sistema. Basicamente, cada implementação contém uma espera pelo seu tempo de execução, o qual pode ter sido especificado pelo projetista na descrição LACCES, a geração do *trace* de execução e, quando necessário, a geração de um valor de retorno.

A Figura 39 ilustra a implementação da operação de escrita.

NomeProjeto.cpp

```
void write()
{
    wait(5.0, SC_NS);
    write_();
}
```

liblaccs.cpp

```
void write_(){
    cout << "at time: " << sc_time_stamp() << " Process Write " << endl;
}
```

FIGURA 39— Código fonte da operação de escrita.

7.3 Ferramenta de Análise

Descrições de arquitetura devem ser “abertas” para ferramentas de análise. Isto é, a linguagem de descrição de arquitetura deve ser capaz de interagir com qualquer ferramenta de análise que trabalhe com informações no nível de arquitetura. Essas ferramentas podem verificar propriedades tais como desempenho e quantidade de armazenamento.

A linguagem deve ser capaz de armazenar as informações necessárias para a análise, como expressões que podem ou não ser interpretadas, entregar essas informações para a ferramenta, receber os resultados, e incorporá-los na descrição da arquitetura, como, por exemplo, gerando mensagens de erro ou interagindo com o usuário.

A representação LSDS possui um papel fundamental para a realização dessa atividade. Ela consegue capturar todos os predicados que são necessários para a análise, além de permitir que outras ferramentas utilizem as informações armazenadas.

LACCES possui uma ferramenta de análise que realiza as seguintes verificações:

- Verificação da alocação da memória: Para cada componente de software que esteja mapeado em uma CPU, a ferramenta verifica se existe memória suficiente no sistema para o seu armazenamento. Caso não exista, o sistema verifica áreas de armazenamento em memórias externas. Atualmente, LACCES realiza essa análise para arquiteturas que utilizam microcontroladores da família MSP430, através da ferramenta *mcp430-gcc* [38].
- Verificação do mapeamento das primitivas:
 - o Cada primitiva comportamental deve ter um tempo associado. Elas também são mapeadas na estrutura e em uma implementação física, no caso de COTS.
 - o Para memórias, verifica se o tempo de acesso é menor ou igual ao tempo de ciclo.
 - o Para temporizadores, verifica se o tempo de trabalho é menor que o período.
 - o Para conectores físicos, verifica se o número de fios definidos é igual ao número de portas descritas na arquitetura.
 - o Para conectores analógicos, verifica se o número de fios é o dobro do número de portas para o tipo *differential* e igual para o tipo *single_ended*.
 - o Verifica a relação entre a taxa de conversão e a taxa de amostragem do sinal em conversores DA/AD.
- Composição do sistema: verifica se a temporização dos conectores e dos

componentes estão corretas, ou seja, um conector não pode ser mais lento do que os componentes que ele interconecta.

Existem inúmeras outras análises que podem ser realizadas, entretanto, algumas delas são bastante específicas. Poderia ser feita uma análise dos predicados comportamentais ou estruturais, pelos valores descritos em seus tipos. Por exemplo, uma memória ROM não pode ter a operação de escrita. Um conversor AD do tipo *flash* deve possuir uma resolução de no máximo 8 bits. Uma análise bastante interessante seria a de estimar o consumo de energia de acordo com o tempo gasto na simulação.

Todas as incompatibilidades encontradas pela ferramenta de análise são reportadas ao projetista através de mensagens de alerta. Esses defeitos não impedem a simulação da arquitetura.

7.4 Ferramenta de Verificação de Tipos

Um problema similar à verificação de tipos em linguagens de programação surge em três pontos em uma linguagem de arquitetura:

- Os tipos de componentes e conectores devem ser coerentes: componentes de software devem estar conectados a componentes de software, o mesmo valendo para componentes de hardware. A comunicação entre componentes de software e de hardware só é possível por intermédio de uma CPU.
- A interface e o tipo dos componentes devem ser compatíveis com o tipo conector utilizado. Ou seja, componentes de hardware com interface SPI só podem se comunicar através de um conector do tipo SPI.
- Alguns componentes de hardware não existem independentemente de outros componentes, como por exemplo, o *Multiplier* ou o *LCD_Driver*. Eles só podem ser definidos dentro de um MCU. Enquanto, outros só existem independentemente, como o *Keypad* e o *LCD_Module*.

Todos os defeitos encontrados pela ferramenta de verificação de tipos são emitidos como mensagens de erro.

Capítulo 8

Validação do Ambiente LACCES

Nesta dissertação foi afirmado que é possível capturar um conjunto significativo e expressivo do conhecimento dos projetistas acerca da arquitetura de um sistema embutido, em uma linguagem conceitual que é constituída de componentes e de conectores. A discussão nos capítulos anteriores mostra que isso é possível.

Neste capítulo são apresentados dois estudos de caso que são utilizados para validar a linguagem e as ferramentas do ambiente LACCES. O primeiro estudo de caso detalha o projeto de um termômetro e é utilizado para validar a simulação comportamental realizada pelo ambiente LACCES. O segundo estudo de caso descreve um sistema cliente-servidor e é utilizado para validar a análise não funcional realizada por este ambiente.

8.1 Estudo de Caso 1: Um termômetro

Este sistema realiza a leitura do valor da temperatura ambiente a cada 100 ms. Esse valor é processado por um componente de software e o resultado deste processamento é exibido em um LCD.

8.1.1 Componentes

Uma das primeiras tarefas que um projetista deve realizar na descrição de uma arquitetura é decompor funcionalmente o sistema em componentes. Neste estudo de caso, os componentes consistem de um microcontrolador, um módulo de LCD e dois componentes de software.

O microcontrolador utilizado foi uma versão simplificada dos microcontroladores da família MSP430. Ele é constituído dos seguintes sub-componentes:

- Uma CPU de 16 bits.

- Um controlador de LCD, com 4 sinais de comuns e 30 sinais de segmentos.
- Uma memória RAM, com capacidade de 512B, tempo de acesso de 50 ns e tempo de ciclo de 50 ns.
- Um ADC, com resolução de 16 bits e tempo de conversão de 2 μ s.

A Figura 40 ilustra a descrição da CPU do microcontrolador.

O módulo LCD contém 4 sinais de comuns e 30 sinais de segmentos. O sensor e a bateria são considerados como elementos externos ao ambiente de projeto.

A Figura 41 ilustra a decomposição do sistema nos seus componentes de hardware.

O tratamento das interrupções é um ponto crucial neste exemplo. Para conseguir modelar a estrutura de tratamento existente nos microcontroladores, foram utilizados dois componentes de software: um controlador de interrupções (*Interrupt Controller*) e um componente (*Provider*) responsável pelo tratamento da interrupção gerada pelo ADC. Este exemplo mostra como um problema complexo no projeto de sistemas embutidos pode ser modelado de forma bastante simples em LACCES.

O componente *Interrupt Controller* encapsula toda a estrutura de controle de interrupções. Ele é responsável pela priorização e pela permutação dos sinais recebidos.

```

Component MCPU : CPU{
  Interface CPUInt{
    port in ADC_int: bit;
    port in res: bit;
    port in p[16]: bit;
    port out lcd[16]: bit;
    port inout mem[16]: bit;
    Provides {
      reset();
    }
  }
  reset() sensitive res;
  word_length = 16 bits;
  architecture_type = Von Neumann;
}

```

FIGURA 40–Descrição em LACCES da CPU.

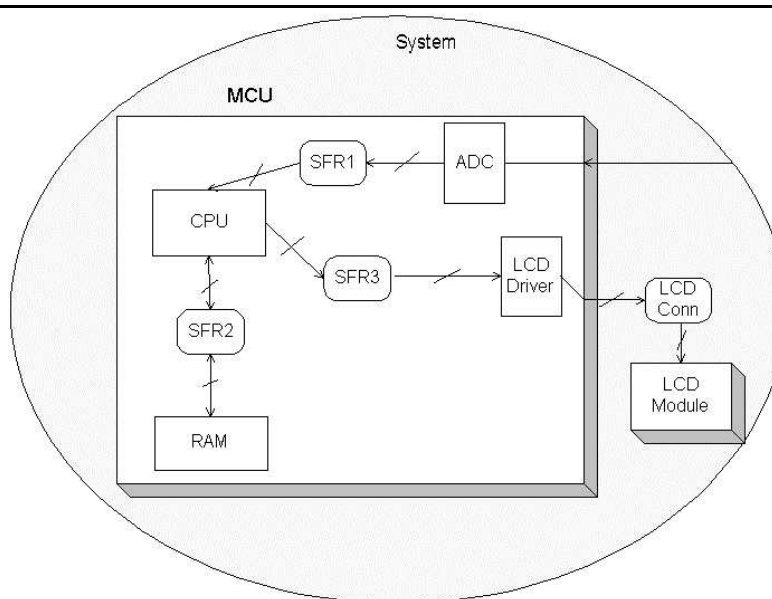


FIGURA 41 – Componentes de hardware do termômetro.

O componente *Provider* possui um processo que corresponde a uma rotina de tratamento de interrupções - ISR (*Interrupt Service Routine*).

O comportamento esperado deste sistema é o seguinte: ao finalizar uma conversão, o ADC gera uma interrupção na CPU sinalizando que existe um dado pronto para ser processado. O controlador de interrupções verifica qual tipo de interrupção foi gerada e aciona a rotina de tratamento adequada. Para cada valor de temperatura convertido pelo ADC, o processo de tratamento de interrupção realiza uma consulta na memória para validar os valores e, em seguida, envia o resultado para ser exibido no LCD.

A Figura 42 ilustra a descrição dos componentes de software deste sistema.

8.1.2 Conectores

Após a decomposição funcional ter sido realizada e as responsabilidades terem sido atribuídas aos componentes, é necessário conectá-los. Isso é feito associando cada saída de um componente a entrada de outro componente, através de conectores.

Os conectores que foram utilizados neste exemplo são instâncias de canais paralelos. Conforme foi dito anteriormente, canais paralelos operam como registradores.

No projeto do microcontrolador são utilizados registradores específicos para permitir a comunicação entre a CPU e os periféricos. Esses registradores são chamados de *Special Function Registers (SFRs)* ou *Control Registers*. A Figura 43 ilustra a descrição de um SFR.

```

Component Provider: Module{
  Interface ProvInt{
    port in sfr: bit;
    port inout mem[16]: bit;
    port out lcd[16]: bit;
    Provides {
      conversion_temp();
    }
  }
  conversion_temp() sensitive sfr
  implementation{
    in conversion.cc;
    FileType source;
    Language SystemC;
  };
}

Component InterruptController: Module{
  Interface InterruptContInt{
    port in inter: bit;
    port out sfr: bit;
    Provides {
      ControlInterrupt();
    }
  }
  ControlInterrupt() sensitive inter
  implementation{
    in interrupt.cc;
    FileType source;
    Language SystemC;
  };
}

```

FIGURA 42 –Descrição em LACCES dos componentes de software.

```

Connector SFR_1 : Hw_Parallel{
  role in din[16] :bit;
  role out dout[16] :bit;
  RoleBindings{
    din[0 ... 15] to dout [0 ... 15];
  }
  send(bit x) assoc din[0 ... 15];
  bit receive() assoc dout[0 ... 15];
  word_time = 3 ns;
  data_length = 16 bits;
  wires = 16;
}

```

FIGURA 43 –Descrição em LACCES dos SFRs.

8.1.3 Sistema

Componentes e conectores isoladamente não possuem nenhuma funcionalidade. Eles somente tornam-se úteis quando são utilizados na composição de sistemas.

Neste exemplo, a conexão dos componentes de hardware é direta, feita através de conectores paralelos. Para realizar a comunicação entre os componentes de software e de hardware são necessários *bindings* especiais, que interconectam o módulo de software às portas correspondentes da CPU. A Figura 44 ilustra a conexão entre os componentes de software e o de hardware e a Figura 45 apresenta o código correspondente em LACCES.

A porta de entrada *inter* do componente *Interrupt Controller*, está associada à porta de interrupção *ADC_int* da CPU. O processo *ControlInterrupt*, que implementa o comportamento deste componente, é sensível a eventos na porta *inter*. Quando a CPU recebe uma interrupção do ADC esse processo é acionado.

As portas de saída do componente *Provider* estão conectadas às portas da CPU que dão acesso à memória e ao controlador de LCD.

Com a descrição da arquitetura pronta, a próxima etapa do desenvolvimento do sistema Termômetro é a geração de código em SystemC. A primeira vez que o código LACCES é compilado, o compilador exibe os diversos erros de sintaxe encontrados. Quando esses erros são resolvidos pelo projetista, o compilador começa a mostrar então os erros semânticos. Somente quando todos os erros são eliminados, o código em SystemC é gerado e é realizada a análise da arquitetura.

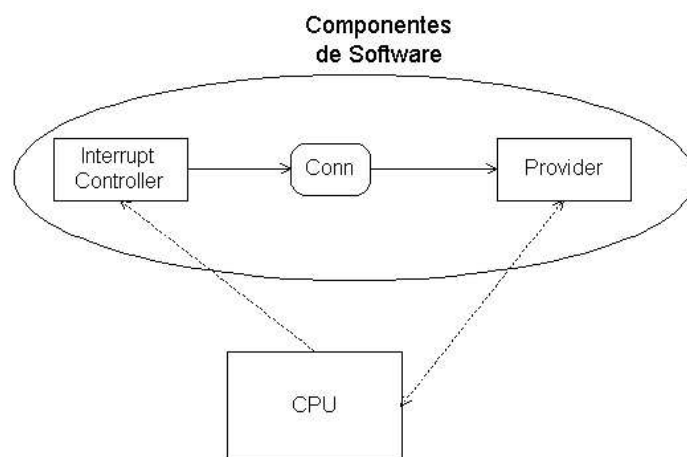


FIGURA 44 –Comunicação entre componentes de hardware-software.

```

Bindings{
    Provider.mem[0 ... 15] to MSP430.MCPU.mem[0 ... 15];
    Provider.lcd[0 ... 15] to MSP430.MCPU.lcd[0 ... 15];
    InterruptController.inter to MSP430.MCPU.ADC_int;
}

```

FIGURA 45 –Descrição em LACCES da comunicação hardware/software.

8.1.4 Simulação e Validação

Quando o código gerado é compilado juntamente com a biblioteca de SystemC, uma aplicação executável é produzida. Essa aplicação deve apresentar o mesmo comportamento que o sistema especificado. Então ela pode ser usada para validar o sistema e verificar se o comportamento especificado é o esperado.

O código gerado por LACCES para o sistema Termômetro foi compilado e a aplicação executável resultante foi usada para simulá-lo.

A aplicação executável gera o arquivo de trace como mostrado na Figura 46, quando executado por 1000 unidades de tempo. Esse arquivo ilustra claramente a seqüência da execução dos processos da arquitetura. Os valores gerados pelos processos, durante a simulação do sistema, são aleatórios.

O projetista deve avaliar esse *trace*, verificando se o comportamento do sistema está correto.

8.2 Estudo de Caso 2: Descrição de um servidor

Este estudo de caso apresenta a descrição de um componente de software hierárquico que executa em um sistema embutido. Esse componente acessa a memória para obter dados necessários ao seu processamento.

8.2.1 Componentes

O componente de software deste sistema consiste de um componente *Server* que é decomposto em três sub-componentes: um gerenciador de conexões, um gerenciador de segurança e um banco de dados. A descrição resumida deste exemplo encontra-se no **Capítulo 4**, nas Figuras 13 e 14. As funcionalidades dos sub-componentes de software são descritas na Figura 47.

Os componentes de hardware consistem de um microcontrolador e de uma memória

FLASH de 32 KB. A descrição da arquitetura deste microcontrolador é similar a do estudo de caso anterior, acrescida de uma outra memória interna: uma ROM de 2KB.

8.2.2 Conectores

Os componentes de software se comunicam através de IPCs. Os conectores de hardware são SFRs, os quais foram descritos na seção anterior.

8.2.3 Sistema

A comunicação entre os componentes de hardware é direta, feita através de conectores paralelos. Para comunicação entre os componentes de software e de hardware são necessários *bindings* especiais que interconectam o módulo de software às portas correspondentes da CPU. Neste exemplo, o componente *database* acessa a memória através da CPU para obter os dados do cliente.

```

ADCP: at time: 0 ns Process ADconversion output :10113
SFR_1: at time: 0 ns Process Send output :31051
SFR_1: at time: 3 ns Process Receive output :9084
MCPU: at time: 3 ns Process Interrupt
ControlInterrupt : at time: 4 ns Process ControlInterrupt
Conn: at time: 4 ns Process Send output :32225
Conn: at time: 7 ns Process Receive output :25089
Provider : at time: 8 ns Process Conversion_temp: Request data Mem
SFR_2: at time: 8 ns Process Send output :16838
SFR_2: at time: 11 ns Process Receive output :16212
RAMData : at time: 61 ns Process Read output :26966
SFR_2: at time: 61 ns Process Send output :16838
SFR_2: at time: 64 ns Process Receive output :16212
Provider : at time: 65 ns Process Conversion_temp: Generate LCD Data
SFR_3 : at time: 65 ns Process Send output :25137
SFR_3 : at time: 68 ns Process Receive output :20495
MLCD_Driver: at time: 69 ns Process Pixel_on_off
LCDConn : at time: 69 ns Process Send output :11367
LCDConn : at time: 72 ns Process Receive output :17031
MLCD_Module : at time: 73 ns Process Display output :19882

```

FIGURA 46 –Trace da execução do Termômetro.

Component connectionManager:

- Validate_client();
- View_Info();
- NewClient();

Component securityManager:

- Verify_Password();
- Verify_Permissions();

Component database:

- GetInfo();
 - Insertion();
 - Exclusion();
-

FIGURA 47–Comportamento dos sub-componentes do servidor.

8.2.4 Análise

A ferramenta de análise realiza um conjunto de validações. A mais importante é a verificação da área de armazenamento necessária para a execução dos processos de software.

Primeiramente, LACCES compila cada processo dos componentes de software associados à CPU, na arquitetura alvo do MSP430.

Após a geração dos executáveis e leitura das áreas de armazenamento de dados e de programa, LACCES verifica se as capacidades das memórias do microcontrolador são suficientes. Neste exemplo, a área necessária para o armazenamento do código (4.094 B) é maior do que a área provida pela memória ROM, interna à MCU. Já a capacidade da memória RAM (512 B) é suficiente para o armazenamento dos dados (28 B).

LACCES emite mensagens, avisando ao projetista sobre as áreas de armazenamento de dados e de código. Como a capacidade da memória ROM não foi suficiente, a ferramenta verifica, para as memórias externas conectadas ao MCU, se existe uma área de armazenamento para a diferença entre o requisitado e o disponível (2.046 B). O resultado dessa verificação é emitido como uma terceira mensagem.

A Figura 48 ilustra as informações exibidas pela ferramenta de análise após a execução deste estudo de caso.

MCU: ROMProgram memory area is not sufficient for software code	
Code Size	Area
4094	2048
MCU: RAMData memory area is sufficient for software data	
Data Size	Area
28	512
FLASH_ext: External memory area is sufficient for software	
Size	Area
2046	32768

FIGURA 48–Mensagens geradas pela ferramenta de análise.

Capítulo 9

Conclusão

Nesta dissertação foi apresentada LACCES, uma linguagem e uma ferramenta, que permite a descrição, simulação e análise da arquitetura de sistemas embutidos.

A linguagem LACCES é usada para capturar tanto o conhecimento sobre o projeto de arquiteturas de sistemas embutidos quanto para especificar uma arquitetura. Esta linguagem pode contribuir para:

- Reduzir dramaticamente os custos, o tempo e as dificuldades encontradas nos projetos de sistemas embutidos. Isso se deve principalmente pela capacidade de modelagem em diversos níveis de abstração e pela possibilidade de descrição de sistemas heterogêneos em um mesmo ambiente.
- Incorporar metodologias distintas de projeto, incluindo *bottom-up* e *top-down*, o que permite capturar informações utilizadas em diferentes níveis de abstração.
- Separar conceitos ortogonais já na descrição da arquitetura do projeto, o que possibilita a exploração de soluções alternativas e eliminação da redundância. Além disso, conceitos ortogonais podem ser manipulados independentemente um dos outros.
- Incorporar restrições e requisitos impostos pelo projetista aos diversos aspectos do sistema - comportamento, estrutura lógica e física. Os requisitos atribuídos ao sistema usualmente servem como guia durante a síntese.
- Permitir a decisão de aspectos críticos nos estágios iniciais de desenvolvimento do projeto. Aspectos que não são necessários em uma representação em alto nível podem ser abstraídos.

LACCES foi desenvolvida a partir de uma lista de primitivas capaz de capturar propriedades dos domínios do comportamento, tempo, estrutura lógica e física de uma grande gama de sistemas. Cada primitiva foi levantada tendo-se em mente a cobertura de cada domínio, sendo consistentes entre si e compatíveis para a descrição de diversos tipos de sistemas embutidos.

Para representar de forma coesa essa separação de conceitos foi utilizada uma

representação interna que é uma adaptação para o mundo de sistemas embutidos da representação interna denominada SDS. A partir desta representação, torna-se possível analisar o sistema quanto a compatibilidade das primitivas nos quatros domínios e extrair o seu comportamento para simulação.

Um conjunto de exemplos e estudos de caso foram apresentados para ilustrar como a técnica, a linguagem e o ambiente descritos podem ser utilizados para efetivamente capturar uma descrição de arquitetura.

Por sua flexibilidade, espera-se que a linguagem aqui proposta possa ser utilizada como ponto de partida no desenvolvimento de projetos de sistemas embutidos, contribuindo para que se tenha como resultado um produto que atenda a todos os requisitos do projetista.

9.1 Trabalhos Futuros

Este trabalho apresenta um ambiente para a descrição da arquitetura de sistemas embutidos. Seria interessante estendê-lo para permitir a co-simulação e a co-síntese da arquitetura. Isso evitaria a necessidade de migração para uma outra linguagem, o que acarreta uma série de problemas, tais como: inserção de erros, necessidade de conversão dos testes, já que os teste que são válidos para um modelo, também devem ser válidos para o modelo convertido, e incompatibilidade entre os modelos, já que o modelo convertido torna-se o foco do projeto, e o modelo original rapidamente torna-se desatualizado.

Quanto à ferramenta LACCES, vários trabalhos podem ser ainda realizados. Em primeiro lugar, obviamente, a construção de uma interface gráfica para a ferramenta, que aumentaria bastante a sua usabilidade.

Além disso, algumas extensões podem ser realizadas para que sua utilização possa trazer ainda mais benefícios para os projetistas. Uma delas seria permitir a descrição de barramentos, possibilitando a execução de operações como detecção de colisão e prevenção de corrupção de dados. No estágio atual, somente conexões ponto a ponto podem ser descritas. A inclusão desses novos conceitos diminuiria bastante o número de conectores necessários para a comunicação e, conseqüentemente, melhoraria a legibilidade da arquitetura especificada.

Uma outra extensão seria a verificação automática das interfaces lógicas dos sistemas embutidos projetados, permitindo a validação das interfaces e integridade física dos sinais. Para isso, a descrição da arquitetura deve estar um nível de abstração mais baixo.

Uma outra extensão possível seria a ampliação da biblioteca de projeto, que até o momento possui a descrição de alguns componentes e conectores, o que permite a geração quase “automática” de descrições de arquiteturas. A descrição de outras arquiteturas de microcontroladores, como ATMEL AVR e os da Motorola (HC08 e HC12), aumentaria

significativamente o poder de expressão de LACCES.

Finalmente, outros tipos de análises podem ser incorporados pela ferramenta. Até o momento, somente um subconjunto de análises possível foi explorado: mapeamento de memória para microcontroladores da família MSP430 e verificação de consistências da arquitetura. Uma análise bastante interessante seria a geração de estimativas do consumo de energia dos processos de software executados na plataforma definida. Essas estimativas poderiam ser geradas durante a simulação da arquitetura, de acordo com os parâmetros previamente definidos na sua descrição, tais como a potência dissipada e o tempo gasto para a realização do seu comportamento.

Referências Bibliográficas

- [1] ADAMS, J. K., e THOMAS, D. E. *The Design of Mixed Hardware/Software Systems*. In Proceedings of the 33rd annual conference on Design Automation, 1996, pages 515-520, Las Vegas, United States, 1996.
- [2] AHO, V. A., SETHI, R., e ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
- [3] ALEXANDER, P., KAMATH, R., e BARTON, D. *System Specification in Rosetta*. IEEE Engineering of Computer Base Systems Symposium. Nashville, Edinburgh, UK, Abril 2000.
- [4] ALEXANDER, P., KONG, C., e BARTON, D. *The Rosetta Usage Guide*. University of Kansas, Technical Report, disponível em <http://www.ittc.ukans.edu/Projects/rosetta/>
- [5] ALLEN, R., e GARLAN, D. *Formalizing Architectural Connection*. In Proceedings of the 16th International Conference on Software Engineering, pages 71-80, Sorrento, Italy, Maio 1994.
- [6] ALLEN, R., e GARLAN, D. *The Wright Architectural Specification Language*, CMU 1996.
- [7] BHASKER, A. *A SystemC Primer*. Star Galaxy Publishing, 2002.
- [8] BIRMINGHAM, W. P., GUPTA, A. P., e SIEWIOREK, D. P. *The MICON System for Computer Design*. IEEE Micro, volume 9, n^o.5, 61-67, Outubro 1989.
- [9] BOOCH, G. RUMBAUGH, e J., JACOBSON, I. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [10] BOUSSINOT, F., e SIMONE, R. *The ESTEREL language*. Proceedings of the IEEE, 79 (9):1293-1304, Setembro, 1991.
- [11] BROOKS, C., LEE, E. A., LIU, X., NEUENDORFFER, S., ZHAO, Y., e ZHENG, H. *Ptolemy II Heterogeneous Concurrent Modeling and Design in Java*. Technical memorandum UCB/ERL M04/27, 2004. <http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm>.
- [12] CHEN, C.T. *System-level Design Techniques and Tools for Synthesis of Application-Specific Digital Systems*. Ph.D. Thesis, University of Southern California, Los Angeles, CA 1994
- [13] CHIDO, M., ENGELS, D., HSIEH, H., GIUSTO, P., JURECSKA, A., LAVAGNO, L., SUZUKI, K., e SANGIOVANNI-VINCENTELLI, A. *A case study in computer-aided co-design of embedded controllers*. Design Automation for Embedded Systems, 1(1-2):51-67, Janeiro 1996.
- [14] CHIDO, M., GIUSTO, H.H., JURECSKA, A., LAVAGNO, L., e VINCENTELLI, A. S. *A formal specification model for hardware/software codesign*. Technical Report UCB/ERL M93/48, U.C. Berkeley, Junho 1993.
- [15] DEREMER, F., e KRON, H. H. *Programming-in-the-large versus programming-in-the-small*. IEEE Trans. Software Eng., vol. SE-2, pp. 80-86, Junho 1976.

- [16] DOMER, R. *The SpecC System-Level Design Language and Methodology*. Embedded Systems Conference, São Francisco, 2002.
- [17] EDWARDS, S. A. *Languages for Digital Embedded Systems*, Kluwer Academic Press, Boston, 2000.
- [18] GAJSKI, D., DUTT, N. D., WU, A. C-H, e LIN, S. Y-L. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.
- [19] GARLAN, D., ALLEN, R., e OCKERBLOOM, J. *Exploiting Style in Architectural Design Environments*. In Proceedings of SIGSOFT'94. The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 179-185. ACM Press, Dezembro, 1994.
- [20] GARLAN, D., MONROE, R., e WILE, D. *Acme: An Architecture Description Interchange Language for Software Architecture*. Technical Report CMU-CS-95-219, Carnegie Mellon University, 1997. <http://www.cs.cmu.edu/~acme>.
- [21] GARLAN, D., e SHAW, M. *An Introduction to Software Architecture*. 1-39. Advances in Software Engineering and Knowledge Engineering Volume 2. New York, NY: World Scientific Press, 1993.
- [22] GERSTLAUER, A. *SpecC Modeling Guidelines*. Technical Report CECS-02-16, Abril, 2002.
- [23] GROTKER, T., LIAO, S., MARTIN, G., e SWAN, S. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
- [24] GUPTA, R.K., e DE MICHELI, G. *Hardware-Software Cosynthesis for Digital Systems*. IEEE Design and Test of Computers, Setembro 1993.
- [25] HAYES, J.P. *Computer Architecture and Organization*, McGraw Hill, 1998.
- [26] HOARE, C. A. R. *Communication Sequential Processes*. Communications of the ACM, 21(8), 666-677. 1978
- [27] HUBERT, H. *A Survey of Hw/Sw Cosimulation Techniques and Tools*. Thesis Work, Stockholm, Junho 1998.
- [28] IEEE. *IEEE Standard VHDL Language Reference Manual*, IEEE 2000.
- [29] IOTECH, Inc. *Signal Conditioning and PC-based Data Acquisition Handbook*, 1997.
- [30] ITU-T Recommendation Z.100: *Specification and Description Language(SDL)* – Technical report ITU, Geneva, Novembro 2000.
- [31] LEE, E. A. *Overview of the Ptolemy Project*. Technical memorandum UCB/ERL M01/11, University of California, Berkeley. Março 6, 2001. <http://ptolemy.eecs.berkeley.edu/>
- [32] LEVINE, J.R., MASON, T., e BROWN, D. *Lex&Yacc*. O'Reilly, 2ª edição, 1992.
- [33] LIU, Z., HANSEN, M.R., MADSEN, J., e BRAGE, J.P. *Real-Time Semantics for Data Flow Graphs*. Technical Report, Department of Computer Science, the Technical University of Denmark, 1995.
- [34] LUCKHAN, D. C., AUGUSTIN, L. M., KENNEY, J. J., VERA, J., BRYAN, D., e MANN, W. *Specification and Analysis of System Architecture using Rapide*. IEEE Transactions on Software Engineering, Special Issue on Software Architecture, 21(4): 336-355, Abril, 1995.

- [35] LUCKHAN, D. C., e VERA, J. *An Event-Based Architecture Definition Language*. IEEE Transactions on Software Engineering, 1995.
- [36] MADSEN, J., Grode, J., KNUDSEN, P. V., PETERSEN, M. E., e HAXTHAUSEN, A. E. *LYCOS: the Lyngby Co-Synthesis System*. Journal for Design Automation of Embedded Systems, 2(2):195–235, Março 1997.
- [37] MONROE, R. T. *Rapid Development of Custom Software Architecture Design Environments*. Ph.D. Thesis, University of Washington, 1999.
- [38] MSPGCC. *A port of the GNU tools to the Texas Instruments MSP430 microcontrollers*. Disponível em <http://mspgcc.sourceforge.net/>
- [39] OMMERING, R.V., LINDEN, F.V., KRAMER, J., e MAGEE, J. *The Koala component model for consumer electronics software*. IEEE Computer vol. 33, pages 78-85, Março, 2000.
- [40] OPEN SYSTEMC INITIATIVE. *Functional Specification for SystemC 2.0*, Outubro 2001. Disponível em <http://www.systemc.org>
- [41] OPEN SYSTEMC INITIATIVE. *SystemC 2.0- User's Guide*. Outubro 2001. Disponível em <http://www.systemc.org>
- [42] PAULA FILHO, W. P. *Engenharia de software; fundamentos, métodos e padrões*. 2.ed. Rio de Janeiro: Editora LTC, 2003. 602 p.
- [43] PAXSON, V. *Flex: A fast scanner generator*. Versão 2.5, 1995. Disponível em <http://www.gnu.org/software/flex/>
- [44] PECOSPROJECT. *Composition environment*. <http://www.pecos-project.org/software.html>. Janeiro, 2003.
- [45] PEIXOTO, Daniela C. C., e SILVA JR., Diógenes Cecílio. *A Framework for Architectural Description of Embedded System*. SCOPES 8th International Workshop on Software and Compilers for Embedded Systems. Amsterdam, The Netherlands, Setembro 2 - 3, 2004. LNCS 3199.
- [46] PRIETO-DIAZ, R., e NEIGHBORS, J. M. *Module Interconnection Languages*. Journal of Systems and Software, 6, 307-334, Novembro 1986.
- [47] SCHILDT, H. *C Completo e Total*. McGraw-Hill, 1991.
- [48] SCHLESINGER, J. D., e KUEHN, J. T. *ADL: An Architecture Description Language Version 3.0*. Supercomputing Research Center, SRC-TR-93-104, Junho 1995.
- [49] SILVA JR, Diógenes C. *A Comprehensive Framework for the Specification of Hardware/Software Systems*. PhD Thesis, University of Southern California, Los Angeles, CA. 2001.
- [50] SHAW, M., DELINE, R., KLEIN, D. V., ROSS, T. L., YOUNG, D. M., e ZELESNIK, G. *Abstractions for Software Architecture and Tools to Support Them*. IEEE Transactions on Software Engineering, Special Issue on Software Architecture, 21(4):314-335, Abril, 1995.
- [51] SHAW, M., e GARLAN, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

- [52] STAUNSTRUP, J. e WOLF, W. *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers.1997.
- [53] STEVENS, A. *Teach yourself... C++*. 5th Edition. Mis:Press, 1997.
- [54] STROUSTRUP, B. *The C++ Programming Language*. Addison-Wesley, 1997, Third Edition.
- [55] THOMAS, D.E., ADAMS, J. K., e SCHMIT, H. *A Model and Methodology for Hardware-Software Codesign*. IEEE Design and Test of Computers, vol. 10, pages 6-15. Julho, 1993.
- [56] THOMAS, D. E., e MOORBY, P. *The VERILOG Hardware Description Language*, Kluwer Academic Press. 1991.
- [57] WILMSHURST, T. *An Introduction to the design of small-scale embedded systems*. 2001, Palgrave.
- [58] WINTER, M., GENBLER, T., CHRISTOP, A. ET. ALI. *Components for Embedded Software- The PECOS Approach*. In Proc. Second International Workshop on Composition Languages, 2002. In conjunction with 16th European Conference on Object-Oriented Programming (ECOOP) Malaga, Spain, Junho 11, 2002.

Apêndices

A.1 Gramática

Este apêndice apresenta a descrição da gramática de LACCES.

As **palavras chaves** estão em negrito.

(...) Parênteses são utilizados para agrupar as produções

[...] Produção opcional

(...)? Seqüência de zero ou um elemento

(....)* Seqüência de zero ou mais elementos

(...)⁺ Seqüência de um ou mais elementos

| Separa escolhas alternativas

```
Init_system ::= Definition_list System_decl
Definition_list ::= (UnitsDefinition|PacketDecl)*
System_decl ::= System ID '{'(System_Body)*}'
System_Body ::= ComponentDecl
                |ConnectorDecl
                |CompInterfaceDecl
                |ConnInterfaceDecl
                |RepresentationDecl
                |ConnectorsBlock
                |ComponentsBlock
                |AttachmentDecl
                |BindingsDecl
ConnBody ::= (ConnPropertyDecl)? (RepresentationDecl)?
CompBody ::= (CompPropertyDecl)? (RepresentationDecl)?
ComponentDecl ::= Component ID ':' ComponentType
                '{'(CompInterfaceDecl)+ [CompBody]}'
ConnectorDecl ::= Connector ID ':' ConnectorType
                '{' ConnInterfaceDecl [ConnBody] '}'
```

```

ComponentsBlock      ::= Components '{' (ComponentDecl)+ '}'
ConnectorsBlock      ::= Connectors '{' (ConnectorDecl)+ '}'
CompInterfaceDecl    ::= Interface ID{'ParsePortDecl IntBehaviorDecl'}
ConnInterfaceDecl    ::= ParseRoleDecl (RoleBindDecl)?
RoleBindDecl         ::= RoleBindings '{'(Name to Name ';'')+ '}'
RoleDecl             ::= Role Direction Name (',' Name)* ':' Type ';'
ParsePortDecl        ::= (PortDecl)+ | PortsBlock
ParseRoleDecl        ::= (RoleDecl)+ | RolesBlock
PortsBlock           ::= Ports '{' (PortDecl)+ '}'
RolesBlock           ::= Roles '{'(RoleDecl)+ '}'
PortDecl             ::= Port Direction [PortBind] Name (',' Name)* ':'
                        Type ';'
PortBind             ::= '['Num ']'
ConnectorType        ::= HW_PARALLEL | SW_PARALLEL | SPI | I2C | HW_FIFO |
                        SW_FIFO | UART | ANALOGUE | SEMAPHORE | IPC
                        | VIRTUAL
ComponentType        ::= MEMORY | MCU | LCD_MODULE | RTC | ADC | DAC |
                        CPU | MULTIPLIER | LDC_DRIVER | TIMER | EVENT |
                        PWM | TIMER_COUNTER | MODULE | KEYPAD | GENERIC
Direction            ::= in | out | inout
ID                   ::= [A-Za-z][A-Za-z0-9_\-]*
Name                 ::= ID | Name '.' ID | Name '[' (ValueList)? ']'
Type                 ::= INT | BIT | INT '['(Num)?']' | BIT '['(Num)?']'
                        | BYTE | BYTE '['(Num)?']' | ANALOGUE | ID
Num                  ::= [0-9]+
RepresentationDecl   ::= Representation {'System_decl [BindingsDecl]'}
AttachmentDecl       ::= Attachments '{'(Name to Name ';'')+ '}'
BindingsDecl         ::= Bindings '{'(Name to Name ';'')+ '}'
IntBehaviorDecl      ::= Provides '{' (ProcDecl)+ '}' Requires '{'
                        (Proc ';'')+ '}' | Provides '{' (ProcDecl)+ '}'
                        | Requires '{'(Proc ';'')+ '}'
PropertyDecl         ::= Name Operator UnitsConst
Operator             ::= '=' | '=>' | '<=' | '!='
Exp                  ::= primary | set | range | list
primary              ::= ID | Num
set                  ::= '{' (ValueList)? '}'
range                ::= '[' (ValueList)? ']'
list                 ::= '(' (ValueList)? ')'
ValueList            ::= primary (',' primary)* | primary '...' primary

```

```

ProcDecl      ::= Proc (SensDecl)? (ImplDecl)? (HasDecl)? ';'
Proc          ::= Name '(' ')'
SensDecl      ::= sensitive (Name) ( ',' Name)*
ImplDecl      ::= Implementation '{' in Name ';' FileType ID ';'
               [ ImplType ID ';' ] [ With ID ';' ]
               [ Language ID ';' ]}'
HasDecl       ::= has {(PropertyDecl)+}
FuncDecl      ::= [Type] Func (AssocDecl)? (ImplDecl)?
               (HasDecl)? ';'
AssocDecl     ::= assoc (Name) ( ',' Name)*
Func          ::= ID '(' ParameterList ')'
ParameterList ::= Type Name ( ',' Type Name)*
PacketDecl    ::= Packet ID ':' ProtocolType '{' (PacketElem)+'}'
PacketElem    ::= ID ( ',' ID)* : Type ';'
ProtocolType  ::= SPI | I2C | UART | ID
UnitsDefinition ::= Units ID '(' BaseUnit (DerivedUnit)+ ')'
BaseUnit      ::= ID
DerivedUnit    ::= ID '=' UnitsDef
ConnPropertyDecl ::= (FundDecl| PropertyDecl)+
CompPropertyDecl ::= (ProcDecl| PropertyDecl)+
UnitsDef       ::= Exp [ID] ';'
UnitsConst     ::= UnitsDef | ProcDecl

```

A.2 Unidades de Medida

A descrição da arquitetura requer predicados com diferentes unidades de medida. As unidades pré-definidas na linguagem estão contidas no arquivo `units.adl`. Esse arquivo deve ser incluído em todos os projetos cujos valores dos predicados sejam descritos em unidades de medidas.

```

units space(
    bit
    bits = bit;
    byte = 8 bits;
    bytes = byte;
    KB = 1024 bytes;
    MB = 1024 KB;
    GB = 1024 MB;

)
units power (
    W
    KW = 1000 W;
    mW = 0.001 W;
    uW = 0.001 mW;
    nW = 0.001 uW;
    pW = 0.001 nW;
)
units length (
    mm
    cm = 10 mm;
    dm = 10 cm;
    m = 10 dm;
)

units time(
    ns
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
)

units bandwidth(
    bps
    Kbps = 1024 bps;
    Mbps = 1024 Kbps;
    Gbps = 1024 Mbps;
    Bps = 8 bps;
    KBps = 1024 Bps;
    MBps = 1024 KBps;
    GBps = 1024 MBps;
)

units frequency(
    Hz
    KHz = 1000 Hz;
    MHz = 1000 KHz;
    GHz = 1000 MHz;
)

```

A.3 Semântica das Propriedades

Este apêndice apresenta um sumário dos predicados que são conhecidos pela linguagem LACCES. Ele está organizado como um conjunto de tabelas. Cada tabela corresponde a um domínio de representação.

A primeira coluna das tabelas contém os tipos das construções da linguagem, a segunda coluna contém os nomes que são semanticamente conhecidos por cada tipo, e a terceira coluna contém uma breve descrição do valor associado a cada nome, de acordo com as unidades definidas no **Apêndice A.2**. A descrição completa da semântica e sintaxe da linguagem LACCES encontra-se nos **Capítulos 3, 4 e 5**.

Tipo	Predicado	Descrição
Memory	<i>read</i>	Operação de leitura de uma posição de memória. Em memórias seriais pode ser lido mais de uma posição em um único acesso.
	<i>write</i>	Operação de escrita em uma posição de memória. Em memórias seriais, essa operação pode escrever mais de um dado em um único acesso.
	<i>erase</i>	Operação que apaga o conteúdo da memória.
	<i>reset</i>	Operação que estabelece o estado inicial da memória.
	<i>read_write</i>	Operação que lê uma posição da memória e, em seguida, escreve um dado no endereço lido.
	<i>refresh</i>	Operação que consiste no refrescamento dos dados da memória.
	<i>lock</i>	Predicado que define o tamanho da área de memória que não pode ser apagada e nem escrita. Seu valor é uma tupla que contém a posição inicial e a final da área protegida.
CPU	<i>reset</i>	Operação que atribui ao PC o endereço inicial do programa.
	<i>interrupt</i>	Operação que interrompe a execução da CPU para interação com periféricos e com o mundo exterior.
	<i>code</i>	Operação que representa a execução de um conjunto de instruções da CPU.
ADC / DAC	<i>conversion</i>	Operação de conversão de um sinal analógico para um digital, no caso de um ADC; ou de um sinal digital para um analógico, no caso de um DAC.
Multiplier	<i>multiplication</i>	Operação de multiplicação.
Timer_Counter	<i>interrupt</i>	Operação que gera uma interrupção quando ocorre um <i>overflow</i> do temporizador/contador.
	<i>read</i>	Operação de leitura do valor do armazenado no temporizador/contador.
	<i>write</i>	Operação de escrita de um valor no temporizador/contador.
	<i>clear</i>	Operação que apaga o valor armazenado no temporizador/contador.
PWM	<i>waveform_generation</i>	Operação que gera um sinal PWM de acordo com o período e o ciclo de trabalho pré-definidos.

<i>Event</i>	<i>interrupt</i>	Operação que gera uma interrupção quanto ocorre um evento.
	<i>capture</i>	Operação que captura o momento da ocorrência de um evento.
	<i>compare</i>	Operação que compara o valor armazenado no temporizador/contador e o valor carregado no registrador de comparação.
	<i>read</i>	Operação de leitura dos valores de comparação e do valor capturado.
	<i>write</i>	Operação de escrita do valor a ser comparado.
<i>RTC</i>	<i>interrupt</i>	Operação que gera uma interrupção quando ocorre um evento.
	<i>alarm</i>	Operação que aciona o alarme quando o valor programado no registrador de contagem de tempo é igual ao armazenado no registrador de alarme.
	<i>read</i>	Operação de leitura do contador.
	<i>write</i>	Operação de escrita no contador.
<i>LCD_Driver</i>	<i>pixel_on_off</i>	Operação que atribui valores aos sinais de segmentos e aos sinais de comuns.
<i>LCD_Module</i>	<i>display</i>	Operação que exibe os dados
<i>Keypad</i>	<i>interrupt</i>	Operação de interrupção decorrente do pressionamento de uma tecla.
<i>SPI</i>	<i>send</i>	Operação de envio de um bloco de dados.
	<i>receive</i>	Operação de recebimento de um bloco de dados.
	<i>mode</i>	Número que indica a configuração da fase e polaridade do <i>clock</i> . Esse valor varia de 0 a 3: 0 – CPHA = 0 e CPOL = 0 1 – CPHA = 0 e CPOL = 1 2 – CPHA = 1 e CPOL = 0 3 – CPHA = 1 e CPOL = 1
<i>I2C</i>	<i>send</i>	Operação de envio de um bloco de dados.
	<i>receive</i>	Operação de recebimento de um bloco de dados.
	<i>ack</i>	Operação de confirmação de uma transmissão com sucesso.
<i>UART, Analogue, IPC, Hw_Parallel, Sw_Parallel</i>	<i>send</i>	Operação de envio de dados.
	<i>receive</i>	Operação de recebimento de dados.
<i>Hw_FIFO, Sw_FIFO</i>	<i>read</i>	Operação de leitura de um dado do início da fila, retirando-o da mesma.
	<i>write</i>	Operação de escrita do dado no final da fila.
	<i>blocking</i>	Número que indica se o processo que está executando a operação irá bloquear ou não, o seu valor pode ser: 0 : Não Bloqueia. 1 : Bloqueia.
<i>Semaphore</i>	<i>wait_signal</i>	Operação que tenta utilizar o semáforo: o processo é bloqueado se o semáforo já estiver sendo utilizado.
	<i>trywait</i>	Operação que verifica se o semáforo está sendo utilizado ou não. Essa operação não bloqueia o processo.
	<i>signal</i>	Operação que libera o semáforo para outro processo utilizá-lo.
	<i>getvalue</i>	Operação que recupera o valor do semáforo.

TABELA 27 - Semântica dos predicados comportamentais de LACCES.

Tipo	Predicado	Descrição
Memory	<i>cycle_time</i>	Número na unidade "time".
	<i>access_time</i>	Número na unidade "time".
CPU	<i>instruction_cycle</i>	Número na unidade "time".
ADC, DAC, LCD_Module, Multiplier	<i>delay</i>	Número na unidade "time".
Timer_Counter, LCD_Driver, Sw_FIFO, Hw_Fifo, IPC	<i>cycle_time</i>	Número na unidade "time".
Event, RTC, Semaphore	<i>access_time</i>	Número na unidade "time".
PWM	<i>period</i>	Número na unidade "time".
	<i>duty_cycle</i>	Número que indica a percentagem do período que o valor é '1'.
SPI, I2C, UART, Analogue, Hw_Parallel, Sw_Parallel	<i>word_time</i>	Número na unidade "time".
Module	<i>execution_time</i>	Conjunto de números que representam o tempo de entrada, de computação e de saída, na unidade "time".

TABELA 28 - Semântica dos predicados de tempo de LACCES.

Tipo	Predicado	Descrição
Memory	<i>organization</i>	Organização da memória, na unidade de medida "space". Ex: <i>organization</i> [2.5D] = (8,256,256) bytes.
	<i>device_type</i>	Tipo da memória.
	<i>capacity</i>	Capacidade da memória, na unidade de medida "space". Ex: <i>capacity</i> = 3 MB.
	<i>mem_area</i>	Indica se a memória é de programa e/ou dados. Os valores válidos para esse predicado são: <i>program</i> , <i>data</i> , <i>both</i> .
CPU	<i>architecture_type</i>	Tipo da arquitetura: <i>Harvard</i> ou <i>Von Newmann</i> .
	<i>word_length</i>	Número na unidade "space".
ADC / DAC	<i>architecture_type</i>	Tipo da arquitetura do conversor.
	<i>resolution</i>	Número na unidade "space".
	<i>accuracy</i>	Número na unidade "space".
	<i>sampling_rate</i>	Número na unidade "frequency".
Multiplier	<i>operand_length</i>	Conjunto que contém os tamanhos das duas entradas e da saída, na unidade "space".
	<i>type</i>	Tipo das entradas de dados: <i>singed</i> ou <i>unsigned</i> .
Timer	<i>prescaler</i>	Valor do divisor da frequência do <i>clock</i> .
	<i>clock_source</i>	Nome que indica qual a fonte do <i>clock</i> .
Timer_Counter, PWM, Event, RTC	<i>resolution</i>	Número na unidade de medida "space".
LCD_Driver	<i>display_capacity</i>	Tupla que determina o número total de sinais de segmentos e de comuns. Por exemplo: <i>display_capacity</i> = [16, 25].
	<i>matrix</i>	Tupla que determina o número de sinais de segmentos e de comuns necessários para a exibição do dado. Por exemplo: <i>display_capacity</i> = [16, 25].
LCD_Module	<i>display_capacity</i>	Tupla que determina o número de caracteres ou <i>pixels</i> que o terminal é capaz de exibir. Por exemplo: <i>display_capacity</i> = [2, 16].
	<i>type</i>	Tipo do terminar: <i>character</i> ou <i>pixel</i> .
Keypad	<i>num_keys</i>	Número de teclas.
Module	<i>data_length</i>	Tamanho da área de dados, na unidade de medida "space".
	<i>code_length</i>	Tamanho da área de código, na unidade de medida "space".
IPC	<i>message_size</i>	Número na unidade de medida "space".

<i>Semaphore</i>	<i>type</i>	<i>integer</i> ou <i>binary</i> .
<i>Sw_FIFO, Hw_FIFO</i>	<i>size</i>	Número na unidade de medida “ <i>space</i> ”.
	<i>data_type</i>	Tipo dos dados armazenados na fila.
<i>Sw_Parallel, Hw_Parallel</i>	<i>data_rate</i>	Número na unidade de medida “ <i>bandwidth</i> ”.
	<i>data_length</i>	Número na unidade de medida “ <i>space</i> ”.
<i>Analogue</i>	<i>frequency</i>	Número na unidade de medida “ <i>frequency</i> ”.
	<i>connection_type</i>	<i>single-ended</i> ou <i>differential</i> .
<i>SPI</i>	<i>data_rate</i>	Número na unidade de medida “ <i>bandwidth</i> ”.
	<i>data_format</i>	MSB ou LSB.
<i>I2C</i>	<i>device_address</i>	Número que pode ser 7 ou 10, na unidade de medida “ <i>space</i> ”.
	<i>data_rate</i>	Número na unidade de medida “ <i>bandwidth</i> ”.
<i>UART</i>	<i>baud_rate</i>	Número na unidade de medida “ <i>bandwidth</i> ”.
	<i>data_rate</i>	Número na unidade de medida “ <i>bandwidth</i> ”.
	<i>data_length</i>	Número que varia de 5 a 8, na unidade de medida “ <i>space</i> ”.
	<i>stop_bit</i>	Número que pode ser 1, 1.5, 2, na unidade de medida “ <i>space</i> ”.
	<i>parity</i>	Número que pode ser 0 ou 1: 0- paridade par. 1- paridade ímpar.

TABELA 29 - Semântica dos predicados estruturais de LACCES.

Tipo	Predicado	Descrição
<i>MCU, Memory, CPU, ADC, DAC, Timer, LCD_Module, RTC, UART, Hw_FIFO</i>	<i>catalog</i>	Texto com informações de identificação do dispositivo, como tipo de empacotamento e nome do fabricante.
	<i>chip_name</i>	Texto que contém o nome do <i>chip</i> .
	<i>cost</i>	Valor monetário do dispositivo.
	<i>dimension</i>	Lista de valores das dimensões do dispositivo, na unidade de medida “ <i>length</i> ”.
	<i>power</i>	Valor da potência dissipada, na unidade de medida “ <i>power</i> ”.
<i>Keypad</i>	<i>phy_keys</i>	Número de teclas.
<i>SPI, I2C, UART, Analogue, Hw_Parallel</i>	<i>wires</i>	Número de fios.
<i>Analogue</i>	<i>signal_type</i>	Tipo do sinal: <i>current</i> ou <i>voltage</i>

TABELA 30 - Semântica dos predicados físicos de LACCES.