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

MARCO AURÉLIO WEHRMEISTER

# Framework Orientado a Objetos para Projeto de Hardware e Software Embarcados para Sistemas Tempo-Real

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

Prof. Dr. Carlos Eduardo Pereira
Orientador

Porto Alegre, março de 2005

**CIP – CATALOGAÇÃO NA PUBLICAÇÃO**

Wehrmeister, Marco Aurélio

Framework Orientado a Objetos para Projeto de Hardware e Software Embarcados para Sistemas Tempo-Real / Marco Aurélio Wehrmeister. – Porto Alegre: PPGC da UFRGS, 2005.

104f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, BR-RS, 2005. Orientador: Carlos Eduardo Pereira.

1. Sistemas tempo-real embarcados. 2. RT-UML. 3. Real-Time Specification for Java. 4. Escalonadores de tarefas. 5. Síntese de Objetos. I. Pereira, Carlos Eduardo. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Profª Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Philipe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Flávio Rech Wagner
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

# AGRADECIMENTOS

Agradeço aos meus pais, Nelson e Berta Gnewuch Wehrmeister, aos meus irmãos Fernando César e Leonardo Rafael pelo apoio, compreensão e, pela fonte inesgotável de carinho e afeto. Agradeço também pelo estímulo e motivação para a conclusão de mais esta etapa.

Aos meus tios, Vendelino e Janete Gnewuch, a meus primos Rudolf, Priscila e Bárbara por me acolher e apoiar durante todo o tempo amenizando a saudade da família que ficou em Santa Catarina.

Agradeço também à minha namorada Josiane de Oliveira pelo seu amor e principalmente pela paciência e compreensão nos momentos em que a distância faz aumentar a saudade e a vontade de estarmos juntos.

Agradeço especialmente ao meu orientador Carlos Eduardo Pereira que, com muita sabedoria e amizade, desempenhou um papel fundamental para a realização desta dissertação, assim como em muitos outros trabalhos desenvolvidos ao longo dos últimos dois anos.

Aos amigos Leandro Buss Becker, Julio Carlos Balzano de Mattos, Antonio Carlos Beck Filho pela valiosa discussão e pela grande ajuda com algumas das ferramentas utilizadas neste trabalho.

Agradeço também a todos os amigos e companheiros de trabalho do laboratório de sistemas embarcados, pelas discussões, dicas, assim como pela descontração das atividades “extra classe”.

E a Deus por ter me dado a oportunidade, saúde e capacidade para conseguir alcançar mais este objetivo, de muitos outros que ainda estão por ser perseguidos e alcançados.

# SUMÁRIO

**LISTA DE ABREVIATURAS E SIGLAS** ........................................ 6
**LISTA DE FIGURAS** ........................................ 8
**LISTA DE TABELAS** ........................................ 10
**RESUMO** ........................................ 11
**ABSTRACT** ........................................ 12
**1 INTRODUÇÃO** ........................................ 13
**1.1 Motivações** ........................................ 14
**1.2 Objetivos** ........................................ 15
**1.3 Organização do Texto** ........................................ 16
**2 REVISÃO DE CONCEITOS** ........................................ 17
**2.1 Sistemas Tempo-Real** ........................................ 17
**2.2 Sistemas Embarcados** ........................................ 18
**2.3 Orientação a Objetos** ........................................ 19
**2.4 Desenvolvimento de Sistemas Tempo-Real Embarcados** ........................................ 20
2.4.1 Modelagem de Sistemas Tempo-Real Embarcados ........................................ 21
2.4.2 Implementação de Sistemas Tempo-Real Embarcados ........................................ 22
**2.5 Real-Time UML** ........................................ 23
**2.6 Java** ........................................ 26
2.6.1 Maquinas Virtuais Java ........................................ 27
2.6.2 Java Tempo-Real ........................................ 28
**3 CONTEXTUALIZAÇÃO DO TRABALHO** ........................................ 31
**3.1 SEEP** ........................................ 31
**3.2 SIMOO-RT** ........................................ 33
**3.3 SASHIMI** ........................................ 34
**4 ANÁLISE DO ESTADO DA ARTE** ........................................ 37
**4.1 Projeto de Sistemas Tempo-Real Embarcados Usando Orientação a Objetos** 37
4.1.1 ROPES ........................................ 37
4.1.2 Metropolis ........................................ 38
**4.2 Implementação de Sistemas Tempo-Real Embarcados Utilizando Java** ........ 40
4.2.1 Máquinas Virtuais Tempo-Real ........................................ 40
4.2.2 Processadores Java ........................................ 45
**5 PROPOSTA DE INTEGRAÇÃO DE FASES NO PROJETO SEEP** ........... 52
**5.1 Descrição do Framework Tempo-Real Orientado a Objetos** ........................ 53

5.1.1 Classes para Modelagem de Tempo .......... 53
5.1.2 Classes para Modelagem das Tarefas .......... 56
5.1.3 Classes para Modelagem de Escalonadores de Tarefas .......... 59
5.1.4 Classes para Modelagem de Temporizadores .......... 61
**5.2 Adaptações na ferramenta SASHIMI .......... 63**
5.2.1 Adaptação da Análise de Bytecodes .......... 64
5.2.2 Adaptação da Síntese do Processador e da Aplicação .......... 66
**5.3 Adaptações no FemtoJava .......... 71**
**5.4 Proposta de Mapeamento .......... 73**
5.4.1 Visão geral da proposta .......... 74
5.4.2 Mapeamento do Modelo RT-UML para Framework RT-OO .......... 74
**5.5 Geração Automática de Código Fonte .......... 79**
**6 ESTUDOS DE CASO .......... 81**
**6.1 Cadeira de Rodas .......... 81**
**6.2 Sistema de Guindastes .......... 87**
**7 CONCLUSÕES E TRABALHOS FUTUROS .......... 94**
**REFERÊNCIAS .......... 96**
**ANEXO A NOVAS INSTRUÇÕES SUPORTADAS .......... 101**

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| A/D | Conversor analógico/digital |
| API | *Application Programming Interface* |
| AO/C++ | *Active Object C++* |
| ASIC | *Application Specific Integrated Circuit* |
| ATC | *Asynchronous Transfer of Control* |
| CACO-PS | *Cycle Accurate Configurable Power Simulator* |
| CAD | *Computer Aided Design* |
| CDC | *Connected Device Configuration* |
| CLDC | *Connected Limited Device Configuration* |
| D/A | Conversor digital/analógico |
| EDF | *Earliest Deadline First* |
| FIFO | *First In First Out* |
| FPGA | *Field Programmable Gate Array* |
| FRT | *Firm Real-Time* |
| GCJ | *GNU Compiler for Java* |
| HIRTS | *High-Integrity Real-Time Systems* |
| HRT | *Hard Real-Time* |
| I/O | *Input/Output* |
| IP | *Intellectual Property* |
| J2EE | *Java 2 Platform Enterprise Edition* |
| J2ME | *Java 2 Platform Micro Edition* |
| J2SE | *Java 2 Platform Standard Edition* |
| JamaicaVM | *Jamaica Virtual Machine* |
| JC | *J Consortium* |
| JDK | *Java Development Kit* |
| JEG | *Java Expert Group* |
| JIT | *Just-In-Time* |
| JOP | *Java Optimized Processor* |
| JNI | *Java Native Interface* |
| JVM | *Java Virtual Machine* |
| LSE | Laboratório de Sistemas Embarcados |
| MAC | *Macintosh* |
| MIDP | *Mobile Information Device Profile* |
| MoC | *Model of Computation* |
| NIST | *U.S. National Institute of Standards and Technology* |
| OMG | *Object Management Group* |
| OO | Orientado à Objetos |
| PC | *Personal Computer* |
| QoS | *Quality of Service* |
| RAM | *Random Access Memory* |
| RFP | *Request for Proposals* |
| RISC | *Reduced Instruction Set Computer* |

| | |
|---|---|
| RM | *Rate Monotonic* |
| ROM | *Read Only Memory* |
| ROPES | *Rapid Object-Oriented Process for Embedded Systems* |
| RTCE | *Real-Time Core Extension* |
| RT-JAVA | *Real-Time Java* |
| RTOS | *Real-Time Operation System* |
| RTSJ | *Real-Time Specification for Java* |
| RT-UML | *Real-Time Unified Modeling Language* |
| SASHIMI | *System As Software and Hardware In MIcrocontrolers* |
| SEDC | *Small Embedded Devices Configuration* |
| SEEP | Sistemas Eletrônicos Embarcados baseados em Plataformas |
| SIMOO-RT | Plataforma Orientada a Objetos para a Simulação Discreta Multi-Paradigma Tempo-Real |
| SoC | *Systems-on-Chip* |
| SRT | *Soft Real-Time* |
| TAFT | *Time Aware Fault Tolerant* |
| UML | *Unified Modeling Language* |
| VHDL | *VHSIC Hardware Description Language* |
| VHSIC | *Very-High-Speed Integrated Circuit* |
| WCET | *Worst Case Execution Time* |

# LISTA DE FIGURAS


Figura 2.1: Estrutura geral do perfil RT-UML (BECKER, 2003)....................................... 24
Figura 2.2: *Framework* para modelagem de concorrência (OMG, 2002)........................ 25
Figura 2.3: *Framework* para modelagem de escalonabilidade (OMG, 2002).................. 25
Figura 2.4: Conjunto de plataformas especializadas Java (SUN, 2004b) ........................ 27
Figura 2.5: Pilha de operandos da JVM ............................................................................ 28
Figura 2.6: Exemplo de código utilizando a biblioteca da RTSJ .................................... 30
Figura 3.1: Diagrama do ciclo de desenvolvimento para sistemas embarcados ............... 31
Figura 3.2: Diagrama de classes e de instâncias do SIMOO-RT ..................................... 33
Figura 3.3: Diagrama de transição de estados do SIMOO-RT......................................... 34
Figura 3.4: Fluxo de projeto suportado pela ferramenta SASHIMI (MATTOS; CARRO, 2003)............................................................................................................ 35
Figura 4.1: Fluxo de projeto proposto no Metropolis (Chen *et al.*, 2003)........................ 39
Figura 4.2: Arquitetura do jRate........................................................................................ 43
Figura 4.3: Arquitetura da JamaicaVM (SIEBERT; WALTER, 2004) ........................... 44
Figura 4.4: Arquitetura do processador picoJava-I (O'CONNOR, 1997)........................ 46
Figura 4.5: *Datapath* do processador JOP (SCHOEBERL, 2004b)................................. 48
Figura 4.6: Arquitetura do FemtoJava (MATTOS; CARRO, 2003)................................ 49
Figura 4.7: Organização da memória do FemtoJava (ITO, 2000).................................... 50
Figura 5.1: Contextualização do presente trabalho no ciclo de projeto SEEP ................. 52
Figura 5.2: Diagrama de classes do framework tempo-real orientado a objetos............... 53
Figura 5.3: Diagrama de classes detalhado para modelagem de tempo ........................... 54
Figura 5.4: Diagrama de classes detalhado para a modelagem das tarefas ....................... 57
Figura 5.5: Diagrama de classes detalhado para a modelagem dos escalonadores de tarefas ........................................................................................................... 60
Figura 5.6: Diagrama de classes detalhado para a modelagem de temporizadores........... 62
Figura 5.7: Fluxo de projeto da ferramenta SASHIMI..................................................... 63
Figura 5.8: Diagrama de classes da nova estrutura de representação das classes no SASHIMI.................................................................................................. 65
Figura 5.9: Trecho de código fonte em Java para possibilitar a síntese de objetos através do SASHIMI.......................................................................................... 66
Figura 5.10: Trecho de código do arquivo RAM.MIF indicando a alocação de um objeto ........................................................................................................... 67
Figura 5.11: Trecho de código binário para o método abstrato mainTask() da classe RealtimeThread....................................................................................... 68
Figura 5.12: Especificação das *threads* concorrentes na versão original do SASHIMI.... 69
Figura 5.13: Esquema do empilhamento dos *frames* da pilha de cada *thread* ................. 70
Figura 5.14: Real-Time Clock........................................................................................... 71
Figura 5.15: Código VHDL do Real-Time Clock ............................................................ 72
Figura 5.16: Modelagem de um recurso escalonável com ativação periódica ................. 75

Figura 5.17: Mapeamento do modelo RT-UML para os elementos do *framework* (*RealtimeThread*) .... 76
Figura 5.18: Modelagem de um temporizador com disparos periódicos .... 77
Figura 5.19: Mapeamento do modelo RT-UML para os elementos do *framework* (*PeriodicTimer*) .... 78
Figura 6.1: Diagrama de casos de uso indicando todas as funcionalidade da cadeira da rodas .... 81
Figura 6.2: Diagrama de casos de uso do controle de movimentação da cadeira de rodas 82
Figura 6.3: Diagrama de colaboração do sensoramento do movimento da cadeira de rodas .... 83
Figura 6.4: Diagramas de colaboração do controle de movimento da cadeira de rodas ... 83
Figura 6.5: Código Java da classe principal do controle de movimento da cadeira de rodas .... 84
Figura 6.6: Código Java da classe MovementController .... 85
Figura 6.7: Sistema de controle de guindastes movendo uma carga (MOSER; NEBEL, 1999) .... 87
Figura 6.8: Diagrama de casos de uso do sistema de controle de guindastes .... 88
Figura 6.9: Diagramas de colaboração da operação normal do controle de guindastes .... 88
Figura 6.10: Classe principal do sistema de controle de guindastes .... 89
Figura 6.11: Código da classe *Controller* .... 90
Figura 6.12: Código da classe *ConsoleInterface* .... 91
Figura 6.13: Código da classe *UserInputTimer* .... 92

# LISTA DE TABELAS

Tabela 4.1: Lista de instruções suportadas pelo FemtoJava ............................................ 51
Tabela 5.1: Métodos da classe *HighResolutionTime* ....................................................... 55
Tabela 5.2: Métodos da classe *RealtimeThread* ............................................................... 58
Tabela 5.3: Métodos da classe *Scheduler* .......................................................................... 60
Tabela 5.4: Descrição dos métodos da classe *Timer* ........................................................ 62
Tabela 5.5: Elementos do perfil RT-UML e seus respectivos mapeamentos para o *framework* ........................................................................................................ 78
Tabela 6.1: Dados da simulação do controle de movimento ............................................ 86
Tabela 6.2: Dados da simulação do sistema de guindastes .............................................. 92

# RESUMO

A crescente complexidade dos sistemas tempo-real embarcados demanda novas metodologias e ferramentas para gerenciar os problemas de projeto, análise, integração e validação de sistemas complexos. Este trabalho aborda o tema co-projeto de sistemas tempo-real embarcados, propondo estratégias para a integração das fases iniciais de modelagem de um sistema tempo-real embarcado com as fases subseqüentes do projeto, como a implementação do software e do hardware. É proposto um framework orientado a objetos que permite a criação de modelos orientados a objetos de sistemas tempo-real embarcados, utilizando conceitos temporais similares aos propostos em UML-RT (ou mais especificamente no *UML Profile for Schedulability, Performance and Time*). É proposta uma estratégia de mapeamento dos requisitos temporais dos diagramas UML-RT para uma interface de programação (API) baseada na "Especificação Tempo-Real para Java" (*Real-Time Specification for Java* ou RTSJ), a qual pode ser executada tanto em software – em programas RTSJ executando em máquinas virtuais Java (JVM) tempo-real – ou em hardware – em processadores Java Tempo-Real. Para permitir o mapeamento para hardware são propostas extensões tempo-real ao processador Java FemtoJava, desenvolvido no âmbito de dissertações de mestrado e projetos de pesquisa no PPGC, criando-se um novo processador tempo-real denominado de RT-FemtoJava. Dentre as extensões propostas ao processador FemtoJava destaca-se a inclusão de um relógio de tempo-real e o suporte a instruções para alocação e manipulação de objetos. Os conceitos propostos foram validados no âmbito de estudos de caso, sendo os resultados obtidos descritos na presente dissertação.

**Palavras-chave**: Sistemas Tempo-Real Embarcados, RT-UML, Real-Time Specification for Java, Escalonadores de tarefa, Síntese de Objetos

**An Object Oriented Framework to Embedded Hardware and Software Design of Real-Time Systems**

# ABSTRACT

The growing complexity of embedded real-time systems demands new methodologies and tools to manage the problems of design, analysis, integration and validation of complex systems. In this work an object-oriented framework to the design of embedded real-time systems is presented. Strategies to map high-level object-oriented models – which include temporal concepts such as those proposed by RT-UML (more specifically in the UML profile for Schedulability, Performance and Time) - to an application programming interface (API) based on the Real-Time Specification for Java (RTSJ) are discussed. Applications written using the proposed API can be executed both as software – in RTSJ programs running on a real-time Java Virtual Machine (JVM) – or as hardware – in real-time Java processors (such as the FemtoJava processor, which was developed within the scope of research projects at PPGC/UFRGS). In order to allow the mapping to hardware some real-time extensions to the FemtoJava processor were proposed, creating a new real-time processor named RT-FemtoJava. Examples of implemented extensions are the inclusion of a real-time clock and the support to bytecodes which allow objects allocation and memory handling. The proposed concepts were validated with cases studies, whose results are described in this work.

**Keywords:** Embedded Real-Time Systems, RT-UML, Real-Time Specification for Java, Task Scheduler, Object Synthesis

# 1 INTRODUÇÃO

O mercado de sistemas embarcados tem crescido de maneira expressiva ao longo dos últimos anos. Uma grande quantidade de atividades realizadas pelo homem é auxiliada por dispositivos com alguma “inteligência” e sua presença no cotidiano tende a aumentar ainda mais devido a avanços tecnológicos que disponibilizam componentes com crescente capacidade de processamento, menor consumo de energia, dimensões reduzidas e menor custo. Esse avanço está provocando uma crescente demanda da utilização de sistemas computacionais em aplicações críticas. Como exemplo, pode-se citar sistemas de automação industrial e residencial, eletrônica embarcada em automóveis, controle aeroespacial, equipamentos médicos entre outras aplicações que envolvem o controle de sistemas físicos.

Os exemplos citados no parágrafo anterior têm como objetivo aumentar a eficiência e a segurança na realização das tarefas para as quais o sistema embarcado foi proposto. Porém, ainda existe outra característica muito importante: todos os exemplos são classificados como sistemas tempo-real. Os sistemas tempo-real são sistemas computacionais onde o correto funcionamento não depende somente do processamento correto das informações, depende também da satisfação dos requisitos temporais da aplicação para a qual o sistema tempo-real embarcado foi projetado (STANKOVIC, 1988; HALANG; STOYENKO, 1991).

O projeto de sistemas tempo-real embarcados é complexo e envolve conceitos pouco explorados e analisados pela computação de propósitos gerais. O projeto destes sistemas deve levar em consideração o hardware e o software necessários à realização da tarefa para a qual o sistema foi concebido. Geralmente existem fatores limitantes no projeto como, por exemplo, a disponibilidade limitada de recursos computacionais (quantidade de memória, poder de processamento, etc.), o limite do consumo de energia ou ainda o curto espaço de tempo de projeto (CARRO; WAGNER, 2003).

Junto com a crescente complexidade do projeto de sistemas tempo-real embarcados, no qual cada vez mais funcionalidades vão sendo incorporadas a um único sistema, está a crescente necessidade de dispor-se de metodologias e técnicas de projeto que visam gerenciar essa complexidade. Além disso, existe a multi-disciplinaridade da equipe de projeto, onde estão envolvidas equipes de engenheiros de hardware, de engenheiros de software, de programadores e equipes responsáveis pelos testes e validações do sistema tempo-real embarcado. A metodologia de projeto utilizada deve permitir uma comunicação clara e uniforme entre os projetistas, de modo a minimizar os erros causados pela interpretação incorreta dos requisitos e funcionalidades do sistema.

O uso de conceitos de orientação a objetos, que é uma tecnologia consagrada no desenvolvimento de sistema computacionais de grande porte e complexidade, no projeto de sistemas tempo-real embarcados apresenta-se como uma interessante alternativa para tentar amenizar os problemas de projeto desta classe de sistemas. Essa tendência pode ser confirmada por estudos recentemente publicados a respeito (IEEE, 2004; WEHRMEISTER, 2004).

## 1.1 Motivações

Haberman *et al.* (1976) defende que hierarquia e abstração são termos chave para o gerenciamento da complexidade dos projetos e para o gerenciamento do aumento do esforço de projeto. Uma decomposição hierárquica reduz a complexidade subdividindo sucessivamente o problema principal em subproblemas de menor complexidade até o ponto que o desenvolvedor possa tratá-los (NEBEL e SCHUMACHER, 1996). A utilização da abstração permite que o desenvolvedor ignore os detalhes da implementação de um elemento do sistema como, por exemplo, a implementação de um escalonador de processos ou o funcionamento exato de um sensor de velocidade. Ao invés disso o desenvolvedor precisa apenas saber qual o comportamento do elemento em questão, e como esse elemento interage com os outros elementos do sistema, diminuindo a complexidade na descrição do sistema tempo-real embarcado. Apesar do conceito de abstração não ser intrinsecamente ligado ao conceito de reuso, a abstração promove a reutilização de componentes de hardware e software pertencentes a bibliotecas de componentes criadas pela própria equipe de desenvolvimento ou compradas de fornecedores externos ao projeto. Para reutilizar um componente é preciso apenas conhecer o seu comportamento e a sua interface.

Visando aumentar a produtividade no projeto de sistemas embarcados, diversos autores têm proposto a utilização de técnicas orientadas a objetos no projeto do hardware e software de sistemas tempo-real embarcados, como exemplo, pode-se citar (AXELSSON, 2000), (LAVAZZA *et al.*, 2001) (STOEL; KARRFALT, 1995) (SINHA *et al.* 2000), (BJORKLUND; LILIUS, 2002) e (MCUMBER; CHEN, 1999).

A utilização da orientação a objetos pela comunidade de desenvolvedores de software foi amplamente discutida na literatura. Como não havia um padrão para a modelagem do software orientado a objetos foi proposta a *Unified Modeling Language* (UML) (BOOCH *et al.*, 1999). A UML é uma linguagem para modelagem orientada a objeto de sistemas que tem sido amplamente utilizada e aceita na comunidade dos desenvolvedores de software por propiciar diferentes visualizações para os problemas que estão sendo modelados.

Em (WEHRMEISTER, 2004) foi realizado um estudo do estado da arte da utilização de orientação a objetos no projeto de sistema embarcados tempo-real. Neste estudo podem-se classificar as propostas existentes nos seguintes grupos:

- Propostas de utilização da UML para modelar sistemas embarcados;
- Extensões da UML, com base no mecanismo de perfis da UML, para expressar de forma mais precisa os requisitos dos sistemas embarcados;
- Metodologias para projeto de sistemas tempo-real embarcados que permitem a geração de especificações executáveis;
- Utilização do conceito de plataformas para o projeto de hardware e software orientados a objeto;
- Propostas de formalização de modelos descritos em UML;
- Ferramentas que permitem o mapeamento de modelos UML para linguagens de descrição de hardware e software;
- Extensões orientadas a objeto para linguagens de descrição de hardware.

Entretanto, considera-se que nenhuma das propostas permite a geração de hardware e software de maneira automática a partir de uma especificação orientada a objetos. Em especial, percebe-se que aqueles métodos que se baseiam em UML não fazem uso das restrições tempo-real propostas no perfil de Escalonabilidade, Performance e Tempo (*UML Profile for Schedulability, Performance and Time*) recentemente aprovado pela

OMG (OMG, 2002). O uso da UML em conjunto com o perfil de Escalonabilidade, Performance e Tempo também é conhecido como RT-UML.

Como conclusões retiradas deste estudo pode-se citar que, como a UML foi inicialmente proposta para a modelagem de artefatos de software, ela carece de características para modelar requisitos dos sistemas tempo-real embarcados, como as restrições temporais e restrições de consumo de potência. Pode-se salientar ainda que não existe uma semântica formal para os diagramas da UML nem para a interação entre eles, o que pode provocar ambigüidades na interpretação dos mesmos, levando a um código gerado de forma incorreta e/ou um código ineficiente.

O presente trabalho encontra-se inserido dentro do projeto de pesquisa Sistemas Eletrônicos Embarcados baseados em Plataformas (SEEP) (LSE, 2003) do Laboratório de Sistemas Embarcados (LSE) desta universidade. A integração das fases do projeto de um sistema tempo-real embarcado é o principal fator motivante para este trabalho.

## 1.2 Objetivos

O objetivo deste trabalho é promover a integração das fases iniciais de modelagem de um sistema tempo-real embarcado com as fases subseqüentes, como a implementação do software e do hardware, promovendo o co-projeto do sistema.

Este trabalho baseia-se em dois trabalhos principais:

- SIMOO-RT (BECKER, 1999), uma extensão da ferramenta SIMOO (COPSTEIN, 1997) a qual permite a modelagem, simulação e implementação de sistemas tempo-real distribuídos, visando evitar as descontinuidades no processo de desenvolvimento de aplicações tempo-real orientadas a objetos;
- SASHIMI (ITO, 2000) que permite a geração de hardware a partir de uma descrição Java (SUN, 1995).

Com o trabalho proposto pretende-se combinar as duas propostas citadas anteriormente, a modelagem em alto-nível com a geração de hardware e software. Permitindo assim que, a partir de uma especificação orientada a objetos de uma aplicação tempo-real para um sistema embarcado, utilizando a RT-UML (OMG, 2002), sejam gerados o software e o hardware do sistema tempo-real embarcado.

Para cumprir tal objetivo, foi proposta uma extensão tempo-real para a ferramenta SASHIMI na forma de um framework orientado a objetos que permita que os requisitos temporais do sistema tempo-real sejam expressos de forma mais precisa e clara. O framework proposto é baseado na *Real-Time Specification for Java* ou simplesmente RT-Java (BOLELLA, 2001).

Assim, utilizando o framework tempo-real proposto neste trabalho, é possível mapear um modelo orientado a objetos de alto-nível descrito utilizando a RT-UML para código fonte Java, compatível com a ferramenta SASHIMI, promovendo o co-projeto do sistema tempo-real embarcado. O mapeamento do modelo em alto-nível descrito em RT-UML é baseado no mapeamento apresentado em (BECKER *et al.*, 2002). A ferramenta de modelagem utilizada para gerar o modelo RT-UML é Artisan RT-Studio (ARTISAN, 2004).

## 1.3 Organização do Texto

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

No próximo capítulo é feita uma breve revisão de conceitos, onde são apresentados conceitos de sistema tempo-real, sistemas embarcados e alguns conceitos do paradigma orientado a objetos.

No capitulo 3 é apresentado o estado da arte no projeto de sistemas tempo-real embarcados onde são apresentadas algumas metodologias de projeto propostas na literatura, linguagens utilizadas na modelagem em alto-nível de sistema, assim como opções para a implementação de sistemas tempo-real embarcados dando ênfase na implementação de aplicações tempo-real utilizando Java.

Já no capitulo 4 é apresentada a descrição detalhada do *framework* orientado a objetos proposto nesta dissertação, onde são detalhadas todas as classes que compõem o *framework* tempo-real orientado a objetos. Neste capitulo também são descritas as adaptações que foram feitas na ferramenta SASHIMI e no processador FemtoJava.

O capitulo 5 apresenta o mapeamento da modelagem alto-nível para os elementos do framework. Além disto, este capítulo apresenta o estudo realizado na tentativa de conseguir realizar a geração de automática de código, partindo do mapeamento proposto no capítulo 4.

No capítulo seguinte, o capítulo 6, são apresentados dois estudos de caso a fim de validar o framework, as modificações realizadas na ferramenta SASHIMI e no processador FemtoJava, e o mapeamento de modelos RT-UML para os elementos do *framework* tempo-real orientado a objetos.

Finalmente, no capitulo 7 são apresentadas as conclusões juntamente com os trabalhos futuros que podem ser realizados para dar continuidade a este trabalho.

# 2 REVISÃO DE CONCEITOS

Neste capítulo será feita uma revisão dos conceitos básicos de sistemas de tempo-real, sistemas embarcados, orientação a objetos, projetos orientado a objetos de sistemas tempo-real embarcados, *Real-Time UML*, assim como a tecnologia Java, nos quais baseia-se a proposta desta dissertação. Maiores detalhes sobre sistemas tempo-real podem ser obtidos em (BURNS; WELLING, 1996). Detalhes sobre sistemas embarcados podem ser encontrados em (CARRO; WAGNER, 2003). Informações sobre orientação a objetos podem ser encontradas em (BOOCH, 1994) e sobre o projeto orientado a objetos em (SOMMERVILLE, 2003).

## 2.1 Sistemas Tempo-Real

Sistemas de tempo-real são uma classe sistemas computacionais onde o correto processamento dos algoritmos implementados não é suficiente para garantir o correto funcionamento do sistema, ou seja, o resultado do processamento dos algoritmos do sistema deve estar pronto nos tempos definidos pelos requisitos temporais do sistema tempo-real. Por essa razão os sistemas de tempo-real são considerados sistemas determinísticos (LAPLANTE, 1997). Processar dados em microssegundos não torna um sistema tempo-real, o que importa são os tempos de resposta serem limitados e previsíveis. Uma série de outros conceitos errados, comumente atribuídos a sistemas de tempo-real, podem ser vistos no artigo publicado por Stankovic (1988).

Existe uma diferença que pode ser percebida nos sistemas de tempo-real, com relação ao não cumprimento dos requisitos temporais. Os sistemas que podem sofrer uma falha crítica se as restrições temporais forem violadas são chamados *Hard Real-Time* (HRT). Já os sistemas que podem continuar operando, ainda que com um desempenho deteriorado, são chamados de *Soft Real-Time* (SRT). Ainda existe uma terceira classificação, que pode ser considerada intermediária entre sistemas HRT e SRT, que são os chamados sistemas *Firm Real-Time* (FRT) onde uma baixa probabilidade de perda de requisitos temporais pode ser admitida (LAPLANTE, 1997).

Sistemas HRT são tipicamente encontrados interagindo com o meio físico, como no caso dos sistemas tempo-real embarcados. Por exemplo, um sistema de supervisão do motor de um carro é um sistema HRT uma vez que o processamento atrasado de um sinal pode causar a falhar do motor, o que pode colocar em risco a vida de seres humanos. Outros exemplos de sistemas embarcados HRT são os equipamentos médicos como um marca-passo cardíaco ou um controlador de processos industriais.

Já os sistemas SRT são usados tipicamente onde existe algum fluxo concorrente de acesso a dados e existe a necessidade de se manter um número conectado de sistemas atualizados, com relação às modificações ocorridas nos dados. Um exemplo destes sistemas pode ser o sistema de transmissão de áudio e vídeo, onde as violações das

restrições temporais resultam na degradação da qualidade, mas o sistema normalmente pode continuar operando.

Os sistemas tempo-real impõem alguns requisitos que merecem comentários:

- O instante de tempo em que uma tarefa deve fornecer uma resposta, também conhecido como *deadline*, que é o instante de tempo máximo que o sistemas tem para concluir o processamento de um determinado algoritmo. Geralmente este é um fator de extrema importância nos sistemas HRT;
- Pior caso de tempo de execução ou *Worst Case Execution Time* (WCET), que indica a máxima duração que um algoritmo pode consumir durante a sua execução;
- A periodicidade indica que uma determinada tarefa do sistema tempo-real deve ser executada ciclicamente com período *t*, onde *t* indica a quantidade de tempo. Geralmente esta característica está presente em tarefas de sistemas de sensoramento e controle;
- A previsibilidade é uma característica fundamental, pois o comportamento do sistema ou o tempo de reação aos estímulos precisa ser conhecido. O tempo de latência e *jitter* devem ser garantidos dentro do tempo limite conhecido. A latência indica o tempo decorrido desde a percepção do estímulo até a efetiva execução do código de processamento para o estímulo. O *jitter* representa a diferença do tempo entre duas medidas de tempo de um mesmo item como, por exemplo, a diferença entre o instante de ativação de duas ocorrências de uma mesma tarefa periódica ou a diferença de latência de uma tarefa;
- O tratamento de exceção, quando o *deadline* ou o período de ativação de uma tarefa cíclica não é respeitado, deve propiciar a realização de ações corretiva de modo a tentar amenizar ou até mesmo eliminar os efeitos da falha temporal.

## 2.2 Sistemas Embarcados

Existem muitas definições de sistemas embarcados, mas todas estas definições podem ser combinadas em um conceito simples. Um sistema embarcado é um sistema computacional de propósito especial que é utilizado em uma tarefa específica. Este sistema computacional de propósito específico é normalmente menos poderoso que os sistemas computacionais de propósito geral, embora em alguns casos existam sistemas embarcados que sejam complexos, desempenhando várias funções diferentes (WOLF, 2000).

Segundo Ganssle e Barr (2003) dos 6,2 bilhões de processadores produzidos em 2002, menos de 2% tornaram-se o cérebro de novos PCs, MACs e *workstations* Unix. Os outros 6,1 bilhões foram usados em sistemas embarcados. Quase todos dispositivos eletrônicos modernos desde brinquedos, eletrônica embarcada de aviões à controladores de máquinas industriais, utilizam processadores para ajudar a controlar fábricas, gerenciar sistemas de armamento e habilitar a comunicação entre pessoas e produtos.

Normalmente em sistemas embarcados é utilizado um processador de baixo consumo de potência com uma quantidade limitada de memória. Alguns destes sistemas embarcados utilizam sistemas operacionais muito pequenos e otimizados, que possuem uma capacidade limitada de funções de operação (CARRO; WAGNER, 2003). Contudo, a escolha dos componentes do sistema sempre deve ser baseada nas características da aplicação para a qual o sistema embarcado está sendo proposto.

Muitos sistemas embarcados são concebidos partindo do pressuposto que devem ser utilizados por um longo período de tempo sem necessitarem manutenção. Na realidade, a intenção de se produzir um sistema embarcado para uma determinada aplicação é

construir um sistema e deixá-lo operando independentemente por toda a sua vida útil (WOLF, 2000). Por essa razão a maioria dos sistemas embarcados não possui componentes mecânicos como ventiladores[1], discos[2], etc. Estes tipos de componentes sofrem desgaste natural com o tempo e periodicamente necessitam ser trocados e/ou reparados. Ao invés disso são utilizados componentes alternativos que provêem a mesma funcionalidade como, por exemplo, componentes de memória ROM e *flash* onde são armazenados o sistema operacional e o software da aplicação.

## 2.3 Orientação a Objetos

Segundo Rumbaugh *et at.* (1994), modelagem e projetos baseados em objetos é um modo de estudar problemas com utilização de modelos fundamentados em conceitos do mundo real. A estrutura básica é o objeto, que combina a estrutura e o comportamento dos dados em uma única entidade.

A análise de sistemas no mundo orientado a objetos é feita analisando-se os objetos e os eventos que interagem com esses objetos. O projeto orientado a objetos é feito reusando-se classes de objetos existentes e, quando necessário, construindo-se novas classes. Ao modelar uma sistema, os projetistas devem identificar seus tipos de objetos e as operações que fazem com que esses objetos se comportem de certas maneiras (MARTIN; ODELL, 1995).

Ainda segundo Rumbaugh *et al.* (1994), um objeto é simplesmente alguma coisa que faz sentido no contexto de uma aplicação. Todos os objetos possuem identidade e são distinguíveis, por exemplo, duas maçãs da mesma cor, formato e textura continuam sendo maçãs distintas. Na análise orientada a objetos, segundo Martin e Odell (1995), o tipo de objeto é uma noção conceitual, que especifica uma família de objetos sem estipular como o tipo e o objeto são implementados.

Os atributos representam as características do objeto, por exemplo, o objeto caneta, possui como atributos: tamanho, cor, fabricante e modelo. Segundo Rumbaugh *et al.* (1994), diferentes instâncias podem ter valores iguais ou diferentes para um dado atributo. Um atributo deve ser um puro valor de dado, e não um objeto.

Uma operação é uma função ou transformação que pode ser aplicada a objetos ou por estes a uma classe. *AmostraSensores* e *ExecutaControle* são exemplos de operações da classe *ControladorVelocidade*. A mesma operação pode ser aplicada a classes diferentes. Uma operação assim é dita polimórfica, isto é, a mesma operação pode tomar formas diferentes em classes diferentes. Um método é a implementação de uma operação em uma classe.

O ato de unir ao mesmo tempo dados e métodos em uma única entidade é denominado encapsulamento. O objeto oculta seus dados de outros objetos e permite que os dados sejam acessados por intermédio de seus próprios métodos. O encapsulamento também oculta os detalhes de sua implementação interna aos usuários do objeto, aumentando assim o nível de abstração no desenvolvimento dos sistemas tempo-real embarcados. Os usuários entendem quais operações do objeto podem ser solicitadas, mas não precisam conhecer os detalhes de como a operação é executada. O encapsulamento permite que as implementações do objeto sejam modificadas sem exigir que os aplicativos que as usam sejam também modificados (MARTIN; ODELL, 1995).

---

[1] Responsável pelo resfriamento da fonte de energia e/ou do processador.
[2] Responsável pelo armazenamento de dados.

Ligações e associações são os meios para estabelecer relacionamentos entre objetos e classes. Uma ligação é uma conexão física ou conceitual entre instâncias de objetos. Matematicamente, uma ligação é definida como uma tupla, isto é, uma lista ordenada de instâncias de objetos.

A agregação é o relacionamento "parte-todo" ou "uma-parte-de" no qual os objetos que representam os componentes são associados a um objeto que representa a estrutura inteira. A propriedade mais significativa da agregação é a transitividade, isto é, se ***A*** faz parte de ***B*** e ***B*** faz parte de ***C***, então ***A*** faz parte de ***C***. A agregação é também, anti-simétrica, isto é, se ***A*** faz parte de ***B*** então ***B*** não faz parte de ***A*** (RUMBAUGH *et al.*, 1994).

Generalização e herança são abstrações para o compartilhamento de semelhanças entre classes, ao mesmo tempo em que suas diferenças são preservadas. Generalização é o relacionamento de uma classe e uma ou mais versões refinadas dela. A classe que estiver em processo de refinamento é chamada de superclasse e cada versão refinada é denominada subclasse. Por exemplo, *Equipamento* é uma superclasse de *Bomba* e *Tanque*. Os atributos e operações comuns a um grupo de subclasses são incluídos na superclasse e compartilhados por todas as subclasses. Diz-se que a subclasse herda as características da superclasse (RUMBAUGH *et al.*, 1994).

## 2.4 Desenvolvimento de Sistemas Tempo-Real Embarcados

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

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

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

Entretanto, os sistemas embarcados também possuem componentes implementados em software que necessitam ser desenvolvidos concorrentemente com os componentes em hardware de modo a atender os requisitos do mercado (*time-to-market*). Em muitos casos a parte em software domina o tempo e o custo do projeto de um sistema tempo-

real embarcado devido a razões como a sua complexidade e a sua flexibilidade que permite alterar uma funcionalidade nas fases finais do projeto (Nebel *et al.*, 2001).

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

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

### 2.4.1 Modelagem de Sistemas Tempo-Real Embarcados

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

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

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

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

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

### 2.4.2 Implementação de Sistemas Tempo-Real Embarcados

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

Segundo Balarin *et al.* (1999), no passado as configurações de hardware dominavam a maioria das implementações, todavia na abordagem atual de implementação, a maioria das aplicações são implementadas em configurações mistas onde o software constitui a principal parte do sistema.

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

Por essa razão a utilização da técnica de orientação a objetos na implementação tanto do software como do hardware de sistemas tempo-real embarcados está ganhando força (WEHRMEISTER, 2004), pois permite o aumento do nível de abstração durante a codificação e facilita o reuso de componentes desenvolvidos em outros projetos. O aumento no nível de abstração traz consigo um overhead no código final gerado pelos compiladores. Contudo este problema tem diminuído de importância graças ao avanço da tecnologia, pois o poder de processamento e a capacidade de armazenamento tendem a aumentar nos dispositivos embarcados.

Uma linguagem orientada a objetos que, cada vez mais, vem ganhando adeptos para o desenvolvimento de software embarcado é a linguagem Java principalmente devido ao fato de Java ser multi-plataforma, ou seja, todo o sistema pode ser executado e validado previamente em outra plataforma de desenvolvimento e depois portado para a plataforma alvo.

## 2.5 Real-Time UML

A UML é uma linguagem que surgiu com o intuito de padronizar a modelagem de sistemas utilizando a orientação a objetos, permitindo várias visualizações diferentes do problema que está sendo modelado. A utilização da UML como linguagem de modelagem de sistema é amplamente aceita pela comunidade de desenvolvedores de software. Já existem algumas propostas para a sua utilização no domínio dos sistemas tempo-real embarcados de forma a modelar não somente o software, mas também os componentes de hardware que serão utilizados no projeto como pode ser visto em (JONG, 2002), (AXELSSON, 2000) e (BJORKLUND e LILIUS, 2002).

Contudo, como a UML foi inicialmente concebida para a modelagem de elementos de software, ela possui deficiências para modelar requisitos necessários para a caracterização de elementos em domínios específicos de aplicação. Para remediar esta deficiência, a UML possui um mecanismo de extensão baseado em perfis (*profiles*), que permite um detalhamento maior para modelar um sistema em um determinado domínio de aplicação. Sendo assim, para a modelagem de sistemas tempo-real embarcados utilizando a UML necessita-se de elementos para modelar características importantes como os requisitos temporais.

Para preencher essa lacuna a OMG, em março de 1999, fez uma requisição para propostas (*Request for Proposals* ou RFP) para a comunidade de tempo-real, de modo que essa apresentasse propostas sobre como a UML poderia ser utilizada para a modelagem de sistemas de tempo-real. Como resultado desta RFP surgiu o perfil UML para Escalonabilidade, Performance e Tempo (*UML profile for Schedulability Performance and Time*) (OMG, 2002) que permite a modelagem necessária para expressar as características de um sistema tempo-real. A utilização da UML juntamente com o perfil da Escalonabilidade, Performance e Tempo também é conhecida como RT-UML[3].

O perfil RT-UML é dividido, de forma hierárquica, em um conjunto de *frameworks* voltados para a modelagem genérica de elementos em um sistema tempo-real e alguns outros *frameworks* especializados, como pode ser observado na Figura 2.1. O conjunto de *frameworks* genéricos é denominado Modelo Geral de Recursos, onde a sua base é o *framework* para modelagem de recursos. Existem ainda dois outros *frameworks* que derivam características do *framework* de recursos, são eles: o *framework* para modelagem de tempo e o *framework* para modelagem de concorrência. As outras partes do perfil RT-UML estão divididos em um módulo para análise de escalonabilidade e um módulo para definição de infra-estrutura de execução.

A idéia da utilização *framework* para modelagem de recursos (*RTresourceModeling*) é permitir que o projetista possa expressar os requisitos de qualidade de serviço (QoS) dos elementos presentes na aplicação que está sendo modelada. Com os requisitos de QoS expresso no modelo, é possível que seja feita uma análise de contexto estática ou dinâmica dependendo da necessidade do projetista. A análise de contexto envolve a representação dos elementos presentes em uma determinada situação de uso. Um exemplo de análise de contexto estática é comparar o tempo máximo de resposta oferecido por um recurso com o tempo máximo de retorno de um método que utiliza tal recurso.

---

[3] Denominação utilizada em (Selic, 2002)

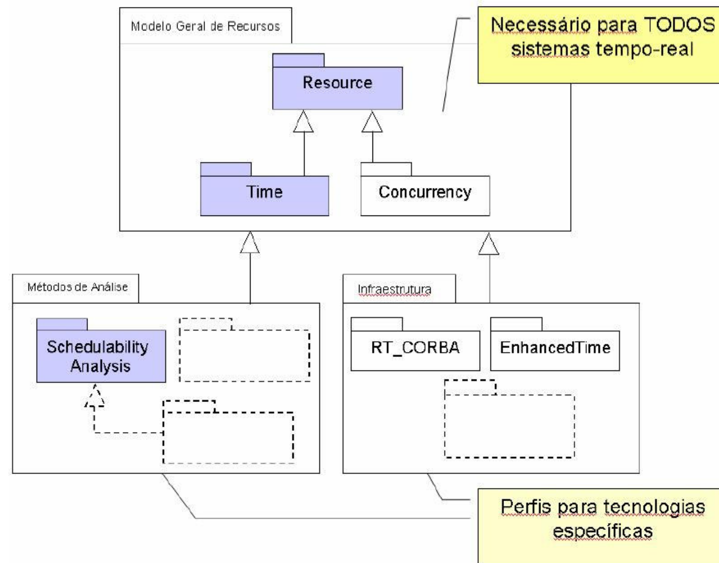


Figura 2.1: Estrutura geral do perfil RT-UML (BECKER, 2003)

O *framework* para modelagem de tempo (*RTtimeModeling*) é dividido em quatro módulos distintos:

- ***TimeModel***: módulo para modelagem de tempo e valores de tempo, onde são definidos conceitos referentes às medidas de tempo. Um valor de tempo (*time value*) representa um instante no tempo. O conceito de duração (*duration*) é o tempo decorrido entre dois instantes de tempo e é representado por um intervalo de tempo (*time interval*);
- ***TimingMechanisms***: módulo para modelagem de mecanismos de tempo. Este módulo oferece duas formas de mecanismos para modelar a infra-estrutura de mecanismos temporais: *timers* e *clocks*. Os timers podem ser programados para gerarem um evento quando determinado ponto no tempo for atingido enquanto os *clocks* são programados para gerarem eventos periodicamente;
- ***TimedEvents***: módulo para modelagem de eventos no tempo. Este módulo permite a modelagem de um tipo especial de evento chamado *timed event* que indica que o mesmo irá ocorrer em um tempo pré-determinado, característica essa que o difere de um evento normal;
- ***TimingServices***: módulo para modelagem de serviços temporais onde estão presentes os mecanismos essenciais para a programação de requisitos temporais como operações de leitura e atualização de relógios, assim como a criação e manutenção de timers, entre outros.

O *framework* para modelagem de concorrência (*RTconcurrencyModeling*) tem o intuito de prover mecanismos para representar a concorrência em sistemas tempo-real embarcados, pois é um aspecto fundamental desta classe de sistemas devido ao fato da grande interação com o meio físico, que é concorrente por natureza. Na **Figura 2.2** é apresentado um diagrama de classes contendo a composição do *framework* para modelagem de concorrência. Um dos elementos principais deste *framework* são os recursos concorrentes (*ConcurrentUnit*) que representam mecanismos para execução concorrente em um sistema, sendo representados por processos ou tarefas de um RTOS. Outro elemento importante são os cenários concorrentes (*Scenario*) que representam seqüências de ações efetuadas pelos recursos concorrentes, sendo estas ações tomadas em resposta a estímulos (*Stimulus*). Os serviços dos recursos concorrentes (*ResourceServiceInstance*) representam como os cenários concorrentes são ativados,

isto é, se os cenários necessitam ser imediatamente executados (*ImmediateService*) ou não (*DeferredService*).

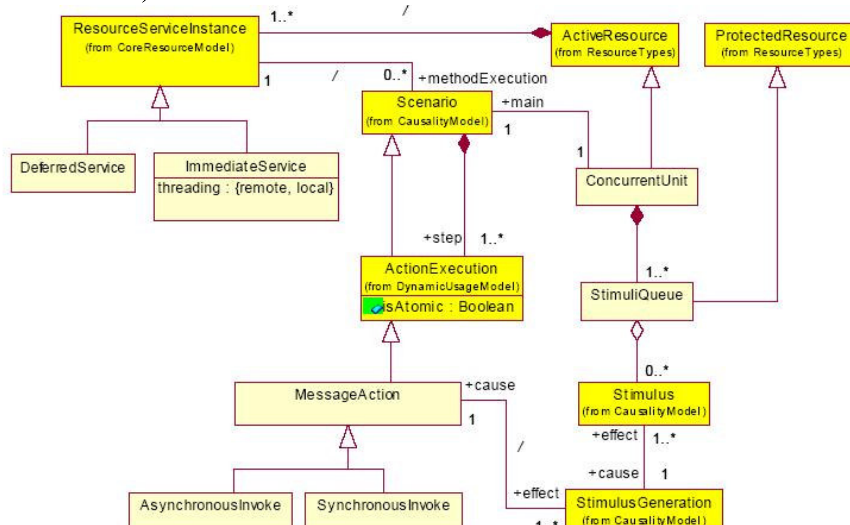

Figura 2.2: *Framework* para modelagem de concorrência (OMG, 2002)

Outro módulo que merece destaque é o módulo de análise de escalonabilidade onde se encontra o *framework* para modelagem de escalonabilidade que permite determinar se as características de QoS solicitadas pelos clientes podem ser atendidas pelos recursos que estão sendo utilizados. A composição deste *framework* pode ser observada na Figura 2.3, onde se observa a existência de um elemento que representa a análise de uma situação onde as características temporais são avaliadas (*RealTimeSituation*). Esta análise envolve dois tipos de recursos: o ambiente de execução (*ExecutionEngine*) que representa um elemento que pode executar os cenários e o recurso protegido (*SResource*) que representa um recurso que pode ser compartilhado por um ou mais cenários. A execução das seqüências de ações de um cenário é ativada quando ocorrer um evento ou estímulo (*Trigger*) e esta é representada através de uma resposta (*Response*) onde podem ser modeladas características de QoS relevantes ao escalonamento dos cenários, como o pior caso de execução (*Worst Case Execution Time* - WCET). A combinação de um estímulo com uma resposta é caracterizada como sendo um trabalho agendado (*SchedulingJob*) e é representado como uma unidade concorrente em um RTOS como sendo um recurso escalonável (*SchedulableResource*).

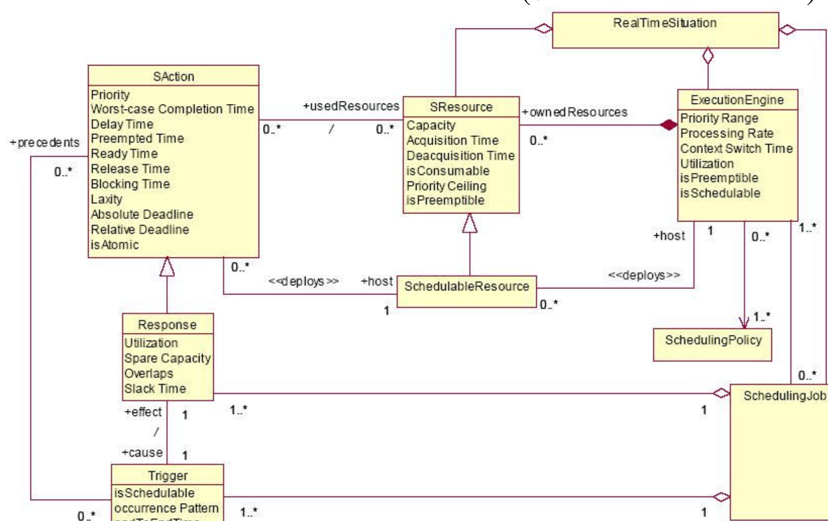

Figura 2.3: *Framework* para modelagem de escalonabilidade (OMG, 2002)

O detalhamento maior do perfil de escalonabilidade, performance e tempo pode ser encontrado em (OMG, 2002) assim como exemplos de utilização dos estereótipos na modelagem UML podem ser vistos nos estudos de caso apresentados no capítulo 6.

## 2.6 Java

Segundo a visão da *Sun Microsystems* (SUN, 2004b), Java não é apenas uma linguagem de programação e sim uma tecnologia que envolve a linguagem de programação Java e um conjunto de plataformas especializadas como pode ser observado na Figura 2.4. A plataforma *Enterprise Edition* (J2EE) define padrões para o desenvolvimento de aplicações empresariais multicamada baseadas em componentes. A plataforma *Standard Edition* (J2SE) é o "coração" do Java, pois propicia o ambiente para o desenvolvimento de aplicações Java para o *desktop* e serve como base para o J2EE. A terceira plataforma é *Micro Edition* (J2ME) que possui um conjunto de tecnologias e especificações para dispositivos embarcados. A quarta plataforma tem o nome de *Java Card* e tem como objetivo adaptar a plataforma Java para que possa ser executada em dispositivos com memória e processamento extremamente limitados (SUN, 2004b).

A linguagem disponível na tecnologia Java é orientada a objetos, permite distribuição de objetos de forma transparente, é interpretada, fortemente tipada, além de ser independente de plataforma, o que a torna portável. É uma linguagem dinâmica, ou seja, suporta alocação dinâmica de memória e possui liberação automática de memória através de um mecanismo de coletor de lixo (*garbage collector*), sendo assim não possui manipulação explícita de ponteiros, diminuindo a possibilidade de erros de programação. Além disso, Java suporta a execução concorrente de múltiplas *threads*.

Segundo *Sun Microsystems* (1995), a tecnologia Java segue a filosofia "*write once, run everywhere*", o que significa que um programa escrito na linguagem Java pode ser executado em qualquer plataforma sem necessidade de alterações, desde que este programa respeite as restrições impostas pela plataforma mais simples. O compilador Java gera código binário (*bytecode*) para ser executado em uma máquina virtual Java (*Java Virtual Machine* ou simplesmente JVM). A JVM prove uma interface com a máquina nativa, ou seja, um programa Java pode executar em um *browser* Web assim como em um computador com diferentes sistemas operacionais como Windows, Linux, ou ainda em um computador *iMac* da *Apple*. Existe ainda outra possibilidade de execução de aplicações Java, a utilização de processadores que executam seus *bytecodes* nativamente como o processador PicoJava (SUN, 1999b), JOP (SCHOEBERL, 2004b) e FemtoJava (ITO, 2001).

A implementação do software para os dispositivos embarcados, utilizando Java, é facilitada com a utilização da plataforma J2ME, pois a mesma proporciona um ambiente robusto e flexível para a execução nos mais variados dispositivos embarcados como celulares, PDAs, e set-top boxes para TVs digitais. Isso se deve a sua arquitetura que compreende uma variedade de configurações, perfis e pacotes opcionais que os desenvolvedores podem escolher para construir o ambiente de execução Java de modo que os requisitos da aplicação sejam respeitados. A plataforma J2ME possui duas configurações: *Connected Limited Device Configuration* (CLDC) e *Connected Device Configuration* (CDC). Essas configurações compreendem uma JVM e um conjunto mínimo de bibliotecas de classe. Para prover um ambiente de execução completo para uma categoria específica de dispositivos uma configuração deve ser combinada com um perfil que é um conjunto de APIs de alto-nível que define um modelo de ciclo de vida

para as aplicações, a interface com o usuário, e o acesso a funcionalidades especificas dos dispositivos. Uma combinação largamente utilizada é a CLDC com o perfil *Mobile Information Device Profile* (MIDP) que caracteriza aplicações para celulares e outros dispositivos com capacidades similares.

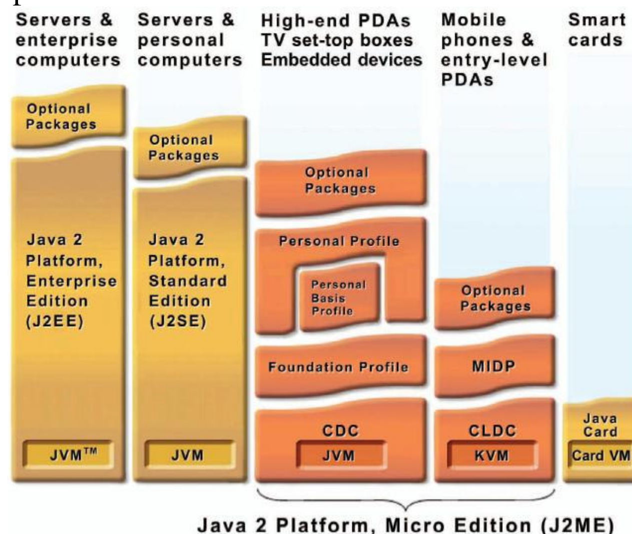


Figura 2.4**:** Conjunto de plataformas especializadas Java (SUN, 2004b)

Entretanto, apesar destas facilidades proporcionadas pela tecnologia Java a sua utilização na implementação em sistemas tempo-real embarcados ainda é pequena, principalmente devido à necessidade de existir uma camada intermediária entre a aplicação e a arquitetura alvo, a JVM. Essa opção causa um grande *overhead* de execução diminuindo o desempenho do aplicativo em uma arquitetura já limitada. A segunda opção, a utilização de um processador Java resolve este problema de desempenho, contudo ainda existe o problema da previsibilidade, característica principal nesta classe de sistemas.

Com o intuito de prover a previsibilidade temporal, dois grupos propuseram extensões para a linguagem Java que seguem as recomendações de um projeto (NIST, 1999) financiado pelo NIST (*U.S. National Institute of Standards and Technology*). O *J Consortium* (JC) propõem a *Real-Time Core Extension* (RTCE) (JC, 2000) que é uma especificação que necessita modificações sintáticas na linguagem Java. Enquanto o *Java Experts Group* (JEG) propôs a especificação denominada *Real-Time Specification for Java* (RTSJ) (BOLLELLA *et al.*, 2001) que define uma interface de programação (API) para a linguagem Java que permite a criação, verificação, análise, execução e gerenciamento de *threads* tempo-real, procurando satisfazer os requisitos temporais.

### 2.6.1 Maquinas Virtuais Java

O componente principal da tecnologia Java é a Máquina Virtual Java. É a JVM que permite a tecnologia Java ser multi-plataforma, além proteger os usuários de programas maliciosos. Ainda, permite que o tamanho do código compilado de um programa Java seja pequeno, devido a JVM possuir as bibliotecas da plataforma Java (J2SE, J2ME, etc.) que ela implementa. Como dito anteriormente o código Java pode ser executado tanto em um processador Java como em uma máquina virtual, que é um software que serve como camada intermediária entre a arquitetura da máquina real e a aplicação.

A JVM é uma máquina de computação abstrata, que possui um conjunto de instruções e usa várias áreas de memória como qualquer máquina real. Ela é uma máquina de pilha que possui um paradigma de execução diferente das máquinas convencionais que possibilitam a transferência de valores entre registradores. A pilha da

JVM recebe o nome de pilha de operandos (*operand stack*), mas esta pilha além de guardar os valores dos operandos de uma instrução, guarda também os valores das variáveis locais assim como os dados de retorno de métodos que foram chamados.

O mecanismo de invocação de métodos é mostrado na Figura 2.5. Esse mecanismo é baseado em *frames* que são alocados na pilha da JVM. Cada método que foi chamado possui um *frame* que pode ser seguido de outros *frames* de outros métodos que são chamados pelo método corrente, formando assim a pilha de chamadas da aplicação. Quando há a chamada de um método, os seus parâmetros são empilhados primeiro, seguidos pelas informações de retorno então o método é chamado. Então este método que está sendo chamado configura as suas primeiras variáveis locais como sendo os seus parâmetros, eliminando a necessidade de cópias dos parâmetros entre os métodos, economizando memória.

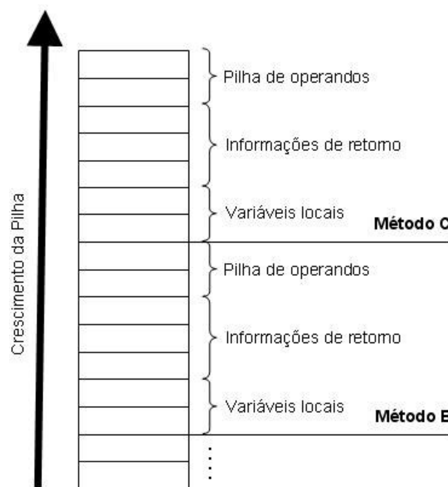


Figura 2.5**:** Pilha de operandos da JVM

O conjunto de instruções da JVM possui 201 instruções diferentes (*opcodes*) sendo divididas como segue:

- **Leitura/Gravação**: são, respectivamente, as instruções de *load* e *store* que permitem a transferência de dados entre a área de variáveis locais e a pilha de operandos, totalizando 70 instruções;
- **Operações Aritméticas e de Conversão de Tipo**: totalizam 53 instruções que permitem, por exemplo, realizar operações de adição, subtração, multiplicação e divisão;
- **Controle do Fluxo de Execução**: são instruções responsáveis pelos desvios condicionais e incondicionais no fluxo de execução de um programa, totalizando 28 instruções;
- **Manipulação de Pilha**: um conjunto de 9 instruções que permitem operações na pilha como, por exemplo, empilhar (*push*) e desempilhar (*pop*) valores na pilha;
- **Manipulação de Objetos**: são um total de 27 instruções que são utilizadas para criação e manipulação de objetos e vetores;
- **Chamada e Retorno de Métodos**: são 10 instruções responsáveis pela invocação de métodos, bem como responsáveis pelo retorno destas chamadas;
- **Demais Instruções:** o restante das instruções que são responsáveis pelas mais diversas operações como controle de *thread*, sincronização e controle de exceções.

### 2.6.2 Java Tempo-Real

Como dito anteriormente a *Real-Time Specification for Java* (RTSJ) é uma API que permite a utilização de *threads* tempo-real e possui alguns sistemas comerciais com

suporte à sua especificação, diferentemente da proposta desenvolvida pelo JC. A RTSJ introduz conceitos de compatibilidade entre aplicações RT-Java e aplicações Java tradicionais, isto é, as aplicações tradicionais não devem sofrer restrições quando executadas em JVMs compatíveis com a RTSJ e estas JVMs não devem incluir especificações que restrinjam o uso da RTSJ em ambientes particulares como no domínio de sistemas tempo-real embarcados. Na seção 4.2.1 serão apresentadas alguma JVMs que implementam a RTSJ.

A RTSJ introduz o conceito de objetos escalonáveis que são quaisquer instâncias de uma classe que implemente a interface *Schedulable* como a classe *RealtimeThread*, *NoHeapRealtimeThread* e *AsyncEventHandler*, sendo que instâncias da duas últimas classes possuem prioridade de execução maior que a execução do *garbage collector* do sistema.

Os objetos escalonáveis estão associados com outros objetos que indicam a necessidade de demanda por recursos. A classe *SchedulingParameters* e suas classes derivadas *PriorityParameters* e *ImportanceParameters* fornecem informações úteis ao escalonador do sistema. Enquanto a classe *ReleaseParameters* e suas derivadas *AperiodicParameters*, *PeriodicParameters* e *SporadicParameters* fornecem informações relativas à ativação do objeto escalonável.

Por definição, o escalonamento padrão do sistema deve ser baseado em prioridades e preemptivo com pelo menos 28 prioridades para a execução de threads com requisitos tempo-real e mais 10 prioridades para as threads sem requisitos tempo-real. As threads que possuem a mesma prioridade são colocadas em uma fila com a política *First In First Out* (FIFO). Também é possível adicionar dinamicamente novos escalonadores possuindo políticas de escalonamento diferentes como *Earliest Deadline First* (EDF), *Rate Monotonic Scheduling* (RMS), definidas na RTSJ.

Os efeitos colaterais introduzidos pelo garbage collector (e.g. imprevisibilidade durante o processo de coleta de lixo) são problemas não desejados em aplicações tempo-real, sendo assim a RTSJ define seis áreas de memória que podem ser utilizadas: *Scoped Memory*, *Physical Memory*, *Raw Memory*, *Immortal Memory*, *Heap Memory*. A *Scoped Memory* é uma memória com tempo de vida limitado, pois quando a última *thread* que referência esta memória é finalizada, todos os destrutores de objetos são invocados e a área de memória é liberada. A *Physical Memory* é usada para controlar a alocação em regiões específicas de memória. A *Raw Memory* permite o acesso em baixo nível à memória e à dispositivos de I/O mapeados na memória, com granularidade de bytes. A *Immortal Memory* é uma memória compartilhada por todas as threads e não possui um garbage collector, sendo que os objetos criados nesta memória são liberados apenas no fim da aplicação. A *Heap Memory* é a memória tradicional dotada do garbage collector.

A implementação da sincronização dos recursos no sistema também feita utilizando-se o mecanismo *synchronized*, como na implementação padrão da linguagem Java, porém a RTSJ exige um algoritmo para prevenir a inversão de prioridade. O protocolo padrão para tratamento da inversão de prioridade é o *priority inheritance protocol*, mas o *priority ceiling protocol* também é suportado e pode ser utilizado.

O tempo é representado, basicamente, por duas classes: *RelativeTime* e *Absolute Time*. A primeira classe representa um tempo relativo a um instante no tempo (e.g. 100 milisegundos) e a segunda classe representa um instante absoluto no tempo (e.g. 01/01/2004 10:52:30). Ambas as classes possuem um objeto da classe *Clock* associado, que permite representar referências de tempo com resoluções diferentes. Ainda existe uma terceira classe a *RationalTime* que representa um intervalo de tempo que é dividido em subintervalos com base em alguma freqüência.

Para representar os eventos do mundo externo como o tratamento de uma interrupção ou um sinal POSIX, ou ainda para representar um evento interno (através da chamada do método *fire()*) a RTSJ define a classe *AsyncEvent*. Um objeto *AsyncEvent* está associado a um tratador de evento (objeto da classe *AsyncEventHandler*) e este pode estar limitado a apenas uma *thread* tempo-real. Os tratadores de evento são executados conforme a política de escalonamento do sistema e a sua liberação pode ser restringida por um intervalo mínimo de tempo entre a ativação de dois eventos.

Os temporizadores são representados pela classe *Timer* que é uma especialização da classe *AsyncEvent* e representa um evento cuja ocorrência é determinada por um relógio tempo-real, em outras palavras, representa a ocorrência de um evento após a expiração de um tempo pré-determinado (*timeout*). Existem dois tipos de temporizadores: *OneShotTimer* e *PeriodicTimer*. A primeira classe representa os temporizadores que disparam um evento uma única vez em um ponto determinado no tempo. Enquanto a segunda classe representa um temporizador periódico, o qual ciclicamente dispara um evento em um determinado tempo e, após isso, dispara o evento continuamente de acordo com o tempo especificado.

Uma característica importante da RTSJ é a inclusão do mecanismo que estende o tratador de exceções do Java, permitindo que o programador possa mudar o ponto de controle de uma *thread* para outra *thread* de uma aplicação. Essa técnica é conhecida como Transferência Assíncrona de Controle (*Asynchronous Transfer of Control*, ATC) e é implementada através da classe *AsynchronouslyInterruptedException*, da sua subclasse *Timed*, da interface Interruptible, assim como com a semântica de métodos interrompidos das classes *Thread* e *RealtimeThread*. A classe *Timed* serve para limitar o tempo de execução de um determinado método desde que este suporte a exceção *AsynchronouslyInterruptedException*, terminando a execução da *thread* corrente e passado o controle para a outra *thread*.

A Figura 2.6 apresenta um exemplo de utilização da biblioteca da RTSJ, através de uma classe de ativação periódica, representado uma tarefa periódica em um sistema tempo-real, que possui um código para tratar a perda de um *deadline*. Neste exemplo também é apresentado o mecanismo de sincronização que utiliza o protocolo de herança de prioridade.

```
public class RTSJ_Sample extends RealtimeThread {
  private AsyncEventHandler m_asyncDeadline = new AsyncEventHandler() {
    public void handleAsyncEvent() {
      // Codigo de exceção de perda de Deadline
    }
  }
  public RTSJ_Sample() {
    super(new PriorityParameters(Thread.NORM_PRIORITY),
            new PeriodicParameters(null, null,
                                   new RelativeTime(500,0),
                                   new RelativeTime(500,0),
                                   null, null));

    getReleaseParameters().setDeadlineMissHandler(m_asyncDeadline);
    MonitorControl.setMonitorControl(this, PriorityInheritance.instance());
  }
  public void run() {
    while (m_isRunning) {
      operation1();
      waitForNextPeriod();
    }
  }
  public synchronized void operation1() {
    // codigo protegido
  }
}
```

Figura 2.6: Exemplo de código utilizando a biblioteca da RTSJ

# 3 CONTEXTUALIZAÇÃO DO TRABALHO

Neste capítulo é feita a contextualização deste trabalho. Primeiro é apresentado o projeto SEEP, no qual este trabalho se encontra inserido. Depois são apresentadas duas ferramentas que foram desenvolvidas como trabalhos no Instituto de Informática desta universidade e que serviram de base para o presente trabalho. Estas ferramentas são: o SIMOO-RT, uma ferramenta de modelagem em alto nível de sistemas tempo-real; e o SASHIMI, uma ferramenta que permite fazer a síntese de hardware e software de um sistema embarcado baseada numa descrição feita em Java.

## 3.1 SEEP

O projeto Sistemas Eletrônicos Embarcados baseados em Plataformas (SEEP) (LSE, 2003) propõe uma metodologia completa e integrada para o projeto e teste de sistemas tempo-real embarcados considerando seus mais variados requisitos. A metodologia propõe um ciclo de projeto que parte desde as especificações de mais alto nível, como a modelagem do sistema em UML, até a geração do hardware e do software embarcado. O diagrama do ciclo de desenvolvimento proposto pode ser observado na Figura 3.1.

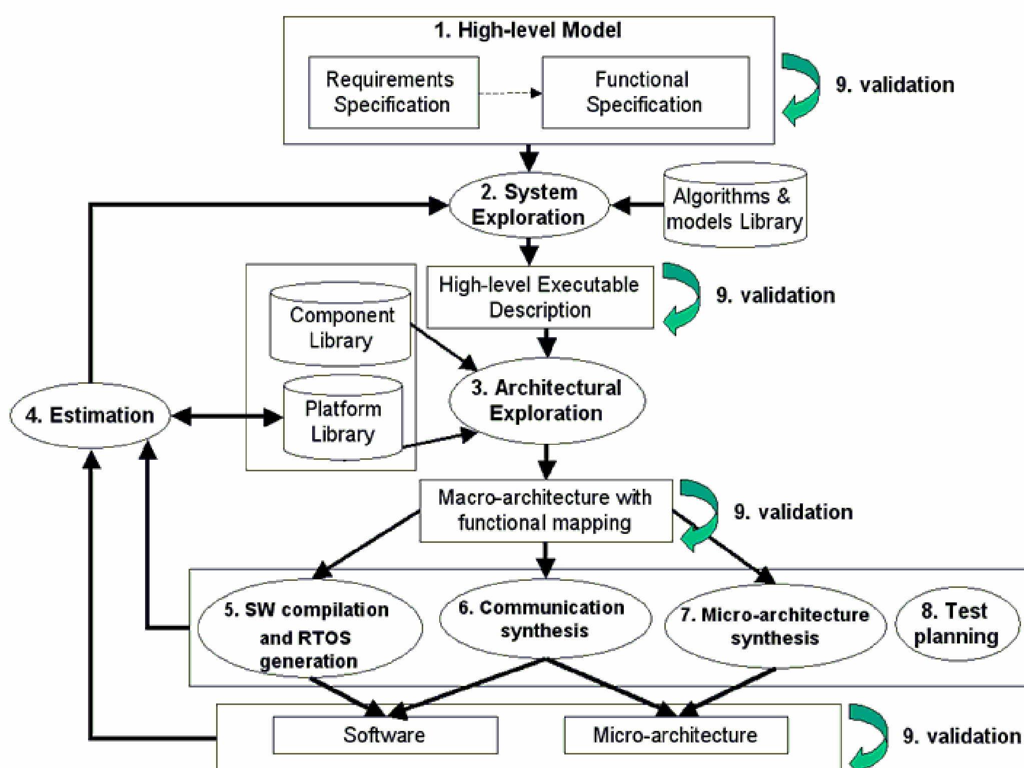


Figura 3.1: Diagrama do ciclo de desenvolvimento para sistemas embarcados

Na Figura 3.1 pode ser observado que o ciclo de projeto inicia com a modelagem em alto-nível do sistema tempo-real embarcado, onde a validação funcional do sistema já pode ser feita. Nesta fase do projeto é feita a especificação dos requisitos e a especificação das funcionalidades do sistema, em alto-nível, sem a preocupação com os detalhes da implementação do sistema. Para expressar os elementos modelados no alto-nível utiliza-se a RT-UML, ou seja, modelam-se os requisitos e funcionalidades utilizando a UML anotando os elementos do modelo com os estereótipos provenientes do perfil de Escalonabilidade, Performance e Tempo (OMG, 2002).

A próxima fase no ciclo de projeto é a exploração do sistema, onde o projetista pode escolher entre diferentes algoritmos que atendam às funcionalidades e requisitos do sistema. Em outras palavras, escolher os algoritmos que estão na biblioteca e implementam a função desejada. Como resultado desta fase, obtém-se uma descrição executável em alto-nível do sistema tempo-real embarcado. Essa descrição é feita utilizando linguagens de programação de alto-nível como Java.

Na fase seguinte o projetista faz a exploração do espaço arquitetural do sistema tempo-real embarcado, onde é feita a associação de funções do sistema com os componentes de hardware. Estes componentes podem ser plataformas de hardware ou componentes de propriedade intelectual (componentes IP). O projeto SEEP possui basicamente duas alternativas de plataformas em hardware que podem ser utilizadas. Uma das plataformas é baseada em processadores Java implementados em FPGAs e a outra é baseada em uma plataforma comercial que possui dois processadores PowerPC e uma área em FPGA. No final desta fase tem-se uma *macro-arquitetura* onde as funções do sistema tempo-real embarcado estão particionadas ou divididas entre os componentes de software e de hardware do sistema. Contudo nesta fase, a marco-arquitetura ainda abstrai os detalhes de mais baixo nível de implementação.

Um ponto importante a ser ressaltado é que, durante a exploração arquitetural e do sistema, são necessárias estimativas em alto-nível para poder avaliar o impacto que as decisões de projeto tomadas têm sobre o sistema tempo-real embarcado como um todo. Isso é necessário para permitir que eventuais erros de projeto possam ser detectados o mais cedo possível durante o desenvolvimento do sistema.

O passo seguinte na metodologia é a síntese automática do hardware e do software do sistema tempo-real embarcado baseado no particionamento das funcionalidades efetuado em fases anteriores do projeto. Na síntese do software além de ser gerado o código binário da aplicação, também é gerado o sistema operacional tempo-real otimizado de modo a atender os requisitos da aplicação como, por exemplo, a comunicação entre processos ou ainda a política de escalonamento das tarefas concorrentes do sistema. Na síntese do hardware, os componentes IP selecionados na fase de exploração arquitetural são reunidos e interligados, fazendo com que se comuniquem a fim de prover a operacionalidade do hardware do sistema tempo-real embarcado.

No final do ciclo de desenvolvimento tem-se o sistema tempo-real embarcado executando a aplicação para a qual ele foi projetado, atendendo os requisitos que foram definidos. Mas ainda cabe salientar que durante todas as etapas do ciclo de projeto, estão previstos testes e validações do sistema guiando o desenvolvimento de forma a minimizar os problemas advindos de decisões erradas tomadas durante o desenvolvimento do sistema tempo-real embarcado.

## 3.2 SIMOO-RT

O SIMOO-RT (BECKER, 1999) é uma ferramenta que permite a modelagem orientada a objetos de sistemas tempo-real distribuídos, onde é possível fazer a simulação do modelo e a geração do código fonte na linguagem AO/C++[4] (PEREIRA, 1994) baseado nas entidades do sistema tempo-real que foram modeladas.

A abordagem hierárquica é adotada pelo SIMOO-RT permitindo que o modelo seja detalhado por níveis conforme a necessidade, isto é, os objetos dos níveis mais baixos indicam que estes fazem parte da construção de objetos nos níveis mais altos. Entre os diagramas que podem ser utilizados no SIMOO-RT estão o diagrama de classes, o diagrama de instâncias e o diagrama de transição de estados. Na Figura 3.2 é apresentada a ferramenta de modelagem do SIMOO-RT onde são ilustrados o diagrama de classes e o diagrama de instâncias.

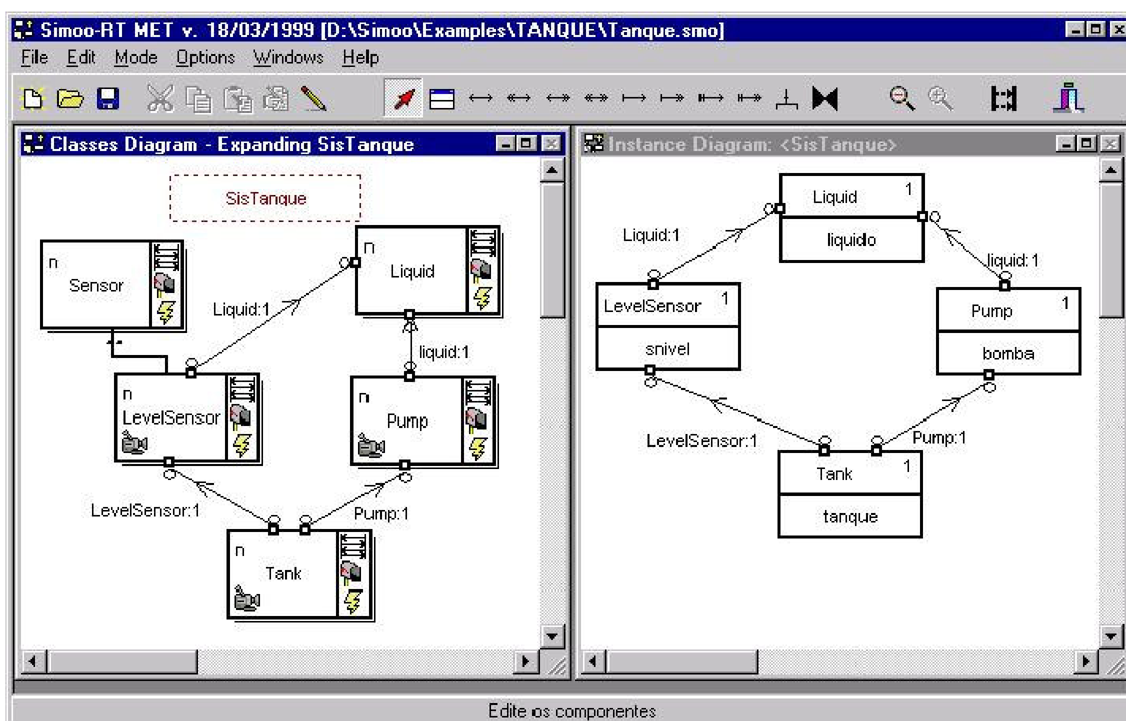


Figura 3.2: Diagrama de classes e de instâncias do SIMOO-RT

O diagrama de classes é um modelo estático utilizado nas etapas iniciais de projeto e representa a estrutura genérica para os elementos que farão parte do sistema. Neste diagrama são apresentadas as relações de herança e associação entre as classes, enquanto as relações de agregação entre as classes são representadas através da hierarquia dos modelos. O diagrama de instâncias também é utilizado nas etapas iniciais de projeto e representa o comportamento dinâmico do sistema tempo-real modelado através da interação entre os objetos modelados. O diagrama de transição de estados é apresentado na Figura 3.3. Neste diagrama é apresentado o comportamento dinâmico de uma classe sendo que uma transição é disparada pela chegada de alguma mensagem, ou seja, quando for feita um chamada de método do objeto.

[4] O *Active Objects* C++ é linguagem que permite o mapeamento do modelo de objetos logicamente distribuídos da linguagem C++ com o modelo físico distribuído de um sistema operacional UNIX de tempo-real chamado QNX.

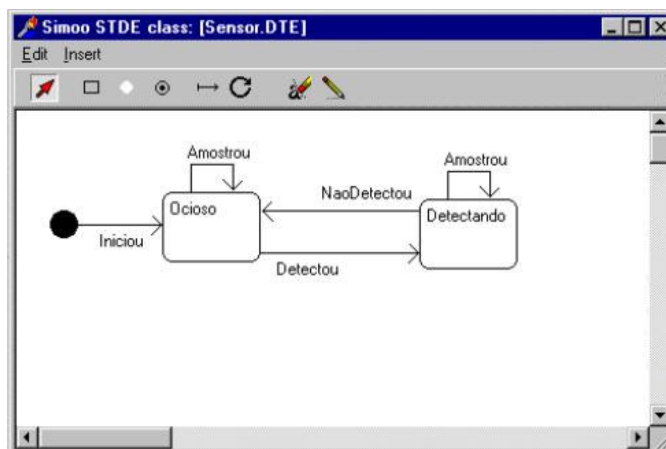


Figura 3.3**:** Diagrama de transição de estados do SIMOO-RT

As restrições temporais que podem ser modeladas no SIMOO-RT são relativas aos métodos das classes e representam, basicamente, a execução periódica e a execução com limite de tempo.

O gerador de código permite a geração de código fonte para a linguagem AO/C++ que é uma extensão da linguagem C++ que inclui primitivas para a definição de objetos ativos, métodos agendados, requisitos temporais e a possibilidade de distribuição física dos objetos ativos em uma rede, de forma transparente, através de suas primitivas de comunicação. O mapeamento do modelo nas construções do AO/C++ é direto, pois esta linguagem possui construções nativas para a associação das características temporais e os métodos das classes.

## 3.3 SASHIMI

A metodologia de projeto *System As Software and Hardware In MIcrocontrolers* (SASHIMI) (ITO, 2000) difere da prática convencional de desenvolvimento de sistemas embarcados, pois tais dispositivos têm sido programados utilizando-se a linguagem *assembly* dos processadores utilizados no projeto, resultando em programas eficientes, mas de alto custo de desenvolvimento e manutenção. A abordagem utilizada pelo SASHIMI permite a utilização da linguagem de alto-nível Java (SUN, 1995) não somente para programação, mas para a especificação completa do sistema embarcado. A utilização de linguagens de alto-nível para a implementação de sistemas embarcados se justifica por elevar o nível de abstração da implementação, diminuindo assim a sua complexidade.

O SASHIMI possui um conjunto de bibliotecas que representam o comportamento dos componentes mais utilizados no projeto de sistemas embarcados, como displays, botões, conversores e filtros, entre outros, que proporcionam o suporte para a modelagem do sistema embarcado. A simulação funcional do sistema modelado pode ser feita utilizando-se da Máquina Virtual Java (*Java Virtual Machine* ou JVM) (LINDHOLM; YELLIN, 1997) do ambiente *desktop* onde foi feito o modelo, facilitando os testes do sistema embarcado.

O fluxo de projeto definido pelo método SASHIMI, apresentado na Figura 3.4, é suportado por uma ferramenta CAD que também recebeu o nome do método, ou seja, Sashimi. Esta ferramenta permite fazer a síntese de uma JVM na forma um processador que executar *bytecodes* Java nativamente chamado FemtoJava (ITO, 2000). Apesar da ferramenta Sashimi permitir a especificação do sistema embarcado utilizando a linguagem Java, a sua utilização é restrita devido ao fato da ferramenta suportar

somente a síntese de métodos e atributos estáticos para as classes, não permitindo assim a utilização plena de conceitos de orientação a objetos diminuindo os benefícios de utilização da linguagem Java.

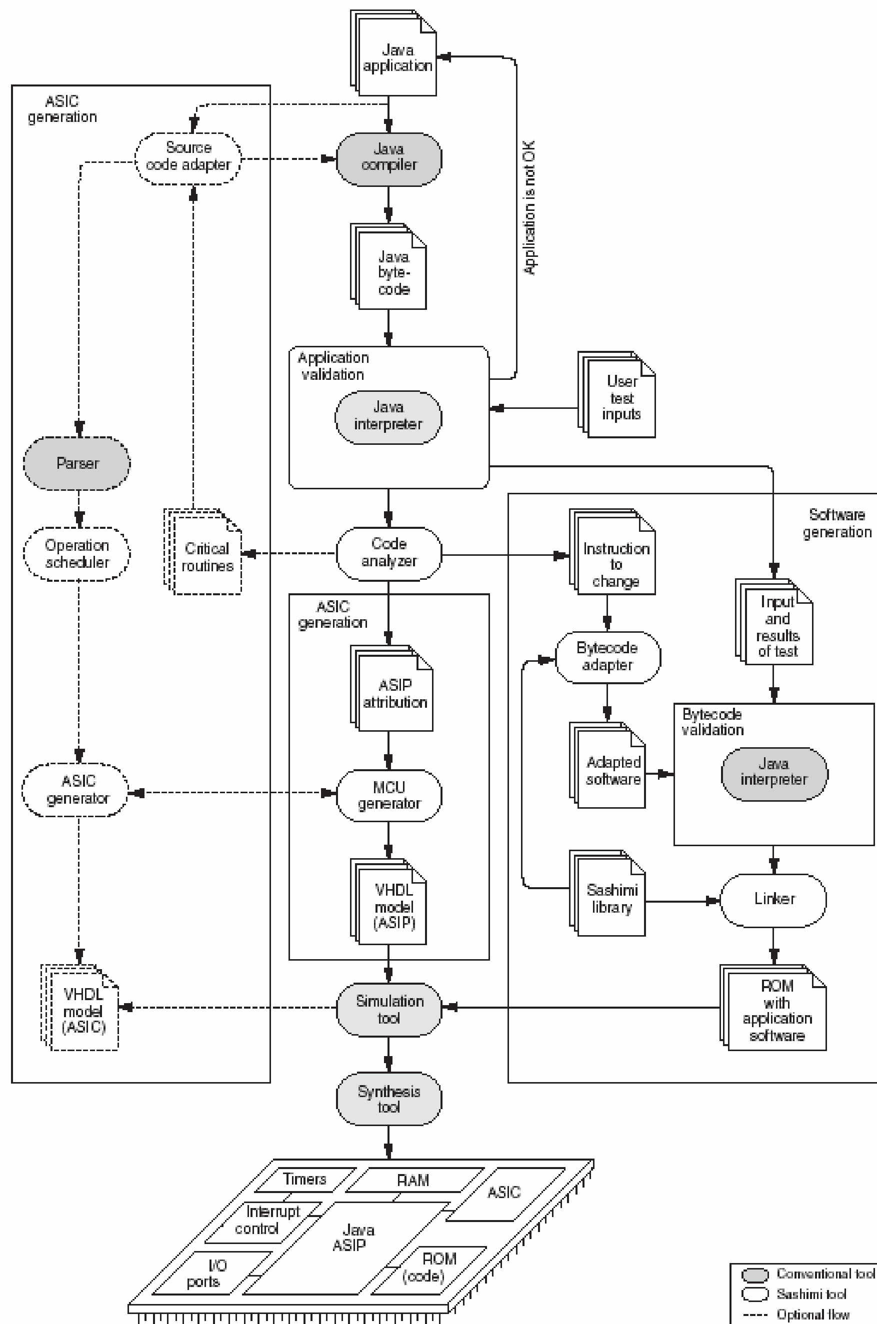


Figura 3.4: Fluxo de projeto suportado pela ferramenta SASHIMI (MATTOS; CARRO, 2003)

O projeto é iniciado através da modelagem da aplicação utilizando a linguagem Java, conforme mostrado na Figura 3.4. Esta fase do projeto segue as regras de um sistema convencional, em outras palavras, implementação do código, compilação execução e validação utilizando-se o ambiente *desktop* com seu kit de desenvolvimento Java (*Java Development Kit* ou JDK) (SUN, 2004a). A execução da aplicação no ambiente *desktop* é equivalente à simulação da aplicação no hardware que ainda não está disponível nesta fase do projeto. Essa simulação pode fazer uso das classes disponíveis no JDK como *threads*, *strings* e entre outros recursos que não são

permitidos para realizar a síntese do sistema embarcado. Posteriormente na fase de síntese estes recursos "proibidos" serão substituídos por código equivalente suportado pelo hardware do sistema. Após as simulações obterem o resultado desejado do funcionamento do sistema embarcado, os vetores de teste utilizados são armazenados para posterior utilização durante na fase de validação final do sistema.

Seguindo o fluxo de projeto tem-se a geração do processador FemtoJava. A ferramenta Sashimi recebe como entrada os arquivos binários (*Java classfiles*) gerados pelo compilador Java padrão da JDK que são analisados afim de obter informações de desempenho e área ocupada pelo hardware. Em outra palavras, os arquivos binários são analisados e os *opcodes* utilizados são identificados de forma a permitir que a máquina de controle do processador FemtoJava seja gerada dando suporte apenas a estes *opcodes*, reduzindo o tamanho do processador.

Entretanto, o processador FemtoJava não suporta todos os *opcodes* disponíveis na JVM padrão (LINDHOLM; YELLIN, 1997), fazendo que algumas funcionalidades não possam ser utilizadas na aplicação pois não podem ser sintetizadas nem executadas pelo processador FemtoJava. É neste ponto que entra a ferramenta de adaptação de código, disponível na ferramenta Sashimi, que permite fazer a substituição de instruções complexas (e.g. *tableswitch, lookupswitch*) por seqüências de instruções suportadas pelo FemtoJava mantendo a semântica da aplicação. Essa adaptação de código ainda pode remover *bytecodes* desnecessários que são usados somente na simulação (como chamadas ao método *System.out.println()*) de acordo com as informações provenientes da análise de código.

Na última fase do fluxo de projeto os arquivos binários (*Java classfiles*) adaptados, compatíveis com o processador FemtoJava que será gerado, são otimizados de forma a eliminar informações desnecessárias como as estruturas de informação para depuração. Em seguida é feita a resolução da hierarquia de classes, a ligação do código e a conversão do código da aplicação e do sistema (bibliotecas de acesso aos recursos do hardware, entre outras) em uma única descrição de ROM. Por fim é feita a geração dos arquivos VHDL correspondentes do processador FemtoJava que podem ser sintetizados por uma ferramenta de síntese e gravados em um FPGA.

# 4 ANÁLISE DO ESTADO DA ARTE

Neste capítulo será apresentada uma análise do estado da arte no que diz respeito ao projeto orientado a objetos de sistemas tempo-real embarcados. Serão apresentadas algumas metodologias de projeto, assim como algumas formas de implementação de sistemas embarcados de tempo-real utilizando Java.

## 4.1 Projeto de Sistemas Tempo-Real Embarcados Usando Orientação a Objetos

Com base no exposto na revisão de conceitos, é possível concluir que para atender aos mais diversos requisitos impostos no projeto dos sistemas tempo-real embarcados, existe a necessidade de metodologias que possibilitem a integração dos projetos de hardware e de software. Nesta seção são apresentadas duas metodologias utilizadas para o projeto de sistemas tempo-real embarcados. Outras metodologias orientadas a objetos utilizadas no projeto de sistemas embarcados podem ser encontradas no estudo apresentado em (WEHRMEISTER, 2004).

### 4.1.1 ROPES

Segundo Douglass (2000) a metodologia *Rapid Object-Oriented Process for Embedded Systems* (ROPES) tem como missão produzir sistemas com menos esforço, minimizando os defeitos e focando na previsibilidade do sistema. A metodologia ROPES é um processo geral para o desenvolvimento de sistemas que, enquanto enfatiza os aspectos relativos ao desenvolvimento de software, inclui a engenharia do sistema, a integração dos seus elementos bem como todos os testes e validações deste sistema.

Existe um pequeno número de tecnologias que auxiliam o processo do ROPES, entre elas pode-se citar:

- **Modelagem Visual**: que traz consigo dois pontos, a capacidade de visualizar diferentes aspectos do sistema e a capacidade de visualizar um aspecto em diferentes níveis de abstração;
- **Modelos de Execução**: uso de modelos executáveis permitem executar e testar modelos, inclusive parte de um modelo, tão logo as características sejam capturadas na ferramenta de modelagem. Isso permite identificar erros ou falhas na modelagem permitindo que sejam tomadas ações corretivas o mais cedo possível, visando diminuir o esforço de correção desses mesmos erros nos níveis de abstração mais baixos;
- **Associação Código-Modelo**: é uma capacidade de manter a semântica do código e do modelo coerentes uma com a outra. Douglass (2003) defende que uma associação código-modelo dinâmica deve ser suportada pelas ferramentas utilizadas no projeto, assim uma modificação feita em uma visão pode ser visualizada na outra visão, em outras palavras, uma alteração em algum

elemento no código do sistema pode ser visualizada nos diagramas que representam a funcionalidade alterada;

- **Testes Automatizados baseados nos Requisitos**: a metodologia ROPES propõe a realização de testes freqüentes nos artefatos do sistema, já nas fases iniciais do projeto. Assim que algum artefato do sistema tenha alguma descrição ele já é testado. A versão testada desse artefato recebe o nome de protótipo. Um protótipo é construído, com base no protótipo anteriormente testado, adicionando-se mais funcionalidades. Assim os primeiros protótipos não possuem toda a funcionalidade definida para o artefato do sistema;
- **Desenvolvimento Interativo**: o desenvolvimento interativo e incremental visa melhorar na produtividade e na previsibilidade do projeto. Um ciclo de vida de projeto em espiral permite o desenvolvimento incremental do projeto. Assim os sistemas vão sendo construídos com base em pedaços pequenos que são testados, que por sua vez vão sendo agrupados e testados, sucedendo assim até que o sistema como um todo seja testado.

O ROPES utiliza o ciclo de vida em espiral, o que pode ser considerado como sendo quebrar o desenvolvimento do projeto em vários subprojetos menores. Estes subprojetos menores são agendados um após o outro de forma a possibilitar que um subprojeto utilize os artefatos construídos nos subprojetos anteriores, adicionando novas funcionalidades e provendo artefatos para os subprojetos que estão por vir. Segundo Douglass (2003) pode-se visualizar o ciclo proposto no ROPES em três escalas diferentes de tempo o macrociclo, o microciclo e o nanociclo

O macrociclo ocorre no período de meses até anos e guia todo o desenvolvimento desde a concepção até a entrega final do produto. As macro fases são uma forma de mostrar os objetivos dos propósitos ao longo do tempo. Os primeiros protótipos são estão focados nos conceitos chave como os requisitos, a arquitetura e/ou a tecnologia do sistema. O próximo conjunto de protótipos foca os conceitos secundários dos requisitos, da arquitetura e da tecnologia. Em seguida o foco dos protótipos muda para os detalhes do projeto e da implementação do sistema. Por fim o último conjunto de protótipos foca nas otimizações e na distribuição (no hardware alvo e no ambiente onde o sistema irá executar).

O microciclo tem o escopo mais limitado que o macrociclo, pois seu foco é no desenvolvimento e na liberação de um único protótipo com a funcionalidade limitada, porém com alto grau de qualidade na sua funcionalidade.

A última escala de tempo em um projeto que segue o ROPES é o nanociclo. A premissa do nanociclo é: assim que o projetista constrói o modelo de algum artefato do sistema, este é constantemente testado para garantir o seu funcionamento correto. A intenção é que este ciclo seja rápido, ou seja, que demore apenas alguns minutos, por essa razão possui poucas atividades.

Esta metodologia é suportada pela ferramenta Rhapsody (I-LOGIX, 2005) e foca principalmente no desenvolvimento do software do sistema tempo-real embarcado. A principal deficiência desta metodologia é a inexistência de regras para customização do hardware que irá executar o software da aplicação tempo-real.

### 4.1.2 Metropolis

O objetivo do projeto Metropolis (BALARIN *et al.*, 2003) é proporcionar uma infra-estrutura baseada em modelos com uma semântica precisa, de modo a suportar os modelos de computação (MoCs) existentes no seu *framework* e ainda acomodar novos modelos que não foram previamente definidos, mas que necessitam ser incorporados na infra-estrutura. Este metamodelo proposto suporta a análise e captura das

funcionalidades, assim como a descrição arquitetural e o mapeamento das funcionalidades nos elementos da arquitetura. O Metropolis utiliza a linguagem lógica para capturar as restrições declarativas e não-funcionais, isso faz com que o modelo tenha uma semântica precisa, que pode ser suportada por várias ferramentas de análise formal e síntese além de possibilitar a simulação do modelo.

Um dos objetivos do Metropolis é promover a comunicação das intenções e resultados do projeto, entre as equipes de desenvolvimento que trabalham em diferentes níveis de abstração. Sendo assim o metamodelo inclui um mecanismo que possibilita a representação, em uma forma abstrata, dos requisitos que devem ser satisfeitos pelo resto do sistema ou que ainda não foram implementados. Em Chen (2003), é proposto um perfil chamado *UML Platform*, o qual deve ser utilizado em conjunto com os diagramas de caso e uso, de colaboração, de seqüências, de classes e diagramas de estado, de forma a prover uma semântica mais clara dos requisitos modelados. Este perfil define estereótipos representando padrões de MoCs como redes de processos Kahn, *dataflows* síncronos, entre outros modelos de computação. Para representar os MoCs são definidos blocos elementares para construção de sistemas como *buffers*, protocolos, etc. que podem ser utilizados para definir um MoC específico.

A metodologia de projeto proposta pelo projeto Metropolis pode ser vista na Figura 4.1. A primeira atividade a ser realizada durante o projeto é a formulação do problema, especificando os requisitos da aplicação.

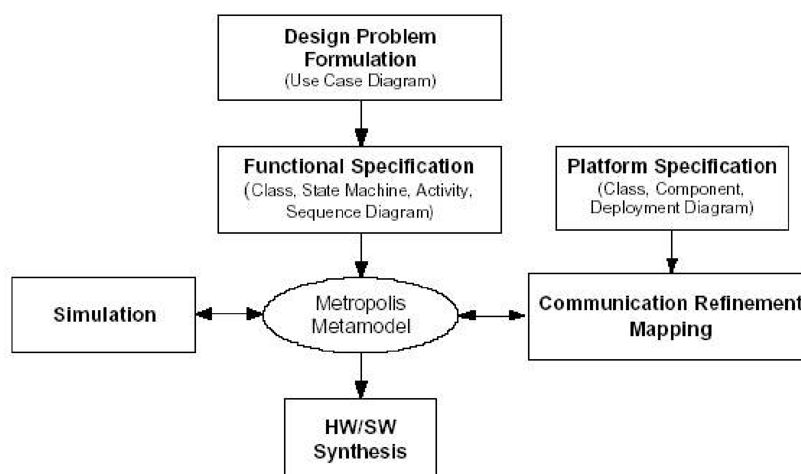


Figura 4.1: Fluxo de projeto proposto no Metropolis (Chen *et al.*, 2003)

A segunda etapa do projeto é a especificação funcional do sistema. Em Chen *et al.* (2003) os elementos da especificação funcional são modelados utilizando os diagramas de classe, de estados, de atividades e de seqüências, todos devidamente anotados com os estereótipos disponível no perfil *UML Platform*.

Em paralelo à especificação funcional acontece a especificação arquitetural, onde é feita a especificação das plataformas que serão utilizadas para implementar as funcionalidades. Dois aspectos distinguem arquiteturas diferentes: a funcionalidade que ela implementa e a eficiência desta implementação. A funcionalidade de um componente arquitetural é representada como sendo um conjunto de serviços que uma arquitetura fornece aos elementos do modelo funcional. Cada serviço é decomposto em uma seqüência de eventos, onde cada evento é anotado com um valor representando o seu custo. O somatório dos custos de todos os serviços representa a eficiência da arquitetura. Um aspecto importante a se destacar é que o metamodelo não possui a

noção de tempo embutido em si. Para modelar o tempo deve-se representá-lo como sendo mais um custo dos elementos do modelo arquitetural.

Depois de ter o modelo funcional e o modelo arquitetural, o próximo passo é fazer o mapeamento dos elementos no modelo funcional para o modelo arquitetural. Após esse mapeamento, pode-se fazer a simulação do sistema. Uma vez validada a simulação, a síntese dos projetos de hardware e software pode ser feita e o sistema é gerado.

Assim como no ROPES, a principal deficiência da metodologia do Metropolis é a falta de uma etapa explícita, durante o projeto do sistema embarcado, que permite a geração de hardware customizado para a aplicação embarcada.

## 4.2 Implementação de Sistemas Tempo-Real Embarcados Utilizando Java

A implementação de sistemas tempo-real embarcados pode ser feita de várias maneiras, sendo assim, esta seção irá focar na implementação destes sistemas utilizando a tecnologia Java e sua especificação tempo-real, a RTSJ. Como visto na revisão de conceitos, a tecnologia Java necessita de uma máquina virtual (JVM) para rodar as aplicações escritas usando Java. Essa JVM pode ser implementada tanto em software como em hardware. As subseções a seguir mostram o estado da arte dessas implementações da JVM com suporte a RTSJ.

### 4.2.1 Máquinas Virtuais Tempo-Real

Originalmente a JVM não tem a execução previsível, uma vez que ela foi idealizada como uma máquina abstrata para que fosse implementada como sendo uma camada de abstração entre a máquina nativa e a aplicação Java. Sendo assim não poderia ser utilizada no domínio de aplicações em tempo-real. A aprovação da RTSJ trouxe consigo uma nova especificação de requisitos que uma JVM deve preencher para estar apta a executar aplicações em tempo-real que utilizam a API da RTSJ. Inicialmente a utilização desta API era fraca pois havia poucas opções de JVMs que suportavam a RTSJ. Entretanto atualmente já existam algumas JVMs compatíveis com o J2SE que podem trabalhar em tempo-real suportando a RTSJ.

Porém a RTSJ foi inicialmente planejada para executar na plataforma J2SE que possui uma configuração muito "custosa" em termos de tamanho de JVM e de bibliotecas suportadas. Como no domínio dos sistemas tempo-real embarcados tem-se restrições quanto à quantidade de memória, poder de processamento e consumo de potência, segundo a *Sun Microsystems*, deve-se utilizar a plataforma J2ME. Dessa forma, a possibilidade de utilização da RTSJ com a plataforma J2ME é uma opção que começa a ser estudada. Schoeberl (2004a) apresenta uma configuração chamada *Small Embedded Devices Configuration* (SEDC) para a plataforma J2ME, juntamente com o perfil *High-Integrity Real-Time Systems* (HIRTS) para sistema embarcados tempo-real. Contudo este perfil não é compatível com a RTSJ, porém segundo Schoeberl, com a adição de uma camada de emulação, é possível executar uma aplicação RTSJ compatível com as configurações de um dispositivo SEDC.

Nas subseções a seguir serão apresentadas três opções de JVMs compatíveis com a RTSJ.

#### 4.2.1.1 Projeto Mackinac

O projeto Mackinac (SUN, 2004d) é a primeira implementação comercial da *Sun Microsystems* de uma JVM compatível com a RTSJ que combina as funcionalidades em

tempo-real com a usabilidade, a abstração e as vantagens do padrão da tecnologia Java. O projeto Mackinac suporta tanto funcionalidades com requisitos tempo-real rígidos (*hard real-time*) como funcionalidades não tempo-real (*non-real-time*) executando na mesma JVM. A JVM do projeto Mackinac é baseada na JVM original da plataforma J2SE e, segundo a *Sun Microsystems*, permite uma performance competitiva com soluções utilizando software compilado como as que utilizam a linguagem C++ com extensões tempo-real, e ainda permite o uso de abstrações de alto-nível que torna Java tão popular entre os desenvolvedores e arquitetos de sistema.

A RTSJ permite que os desenvolvedores decidam para quais áreas as soluções podem ser propostas, que vão desde o nível de sistemas tempo-real embarcados até sistemas tempo-real de médio e grande porte. No projeto Mackinac a *Sun Microsystems* escolheu endereçar as necessidades de sistemas tempo-real que fazem parte de sistemas maiores, que focam no uso de computadores SPARC/x86 que rodam o sistema operacional Solaris e realizam tarefas como monitoramento, controle e gerenciamento de elementos no mundo externo.

A primeira versão da JVM do projeto MAckinc vai executar em computadores desktop ou *workstations*, contudo esta versão ainda não está disponível. Em versões futuras do projeto Mackinac, a *Sun Microsystems* pretende incluir suporte a pequenos dispositivos embarcados assim como aplicações tempo-real em nível de corporação.

A JVM do projeto Mackinac não faz uso do compilador *Just-In-Time* (JIT) como ele é normalmente utilizado. Ao invés disso, na inicialização, o compilador lê uma lista de todos os métodos que necessitam ser pré-compilados e realiza a compilação. Esta abordagem é chamada de *Initialization-Time Compilation* (ITC). Assim os métodos não tempo-real continuam sendo compilados usando o JIT e os métodos HRT e SRT usam o ITC evitando o comportamento imprevisível introduzido pelo compilador.

A implementação da JVM do projeto Mackinac permite os três tipos de tarefas suportados pela RTSJ: tarefas periódicas, esporádicas e aperiódicas. As tarefas periódicas possuem um deadline e um período, sendo liberadas para execução pelo escalonador em intervalos regulares de acordo com a freqüência indicada no período. As tarefas esporádicas possuem um deadline, porém não tem um período de ocorrência fixo. São liberadas por eventos que não ocorrem em intervalos regulares, porém possuem um tempo mínimo entre duas ocorrências. Para tratar este tipo de evento o escalonador o trata como uma tarefa periódica sendo que o seu período é igual ao tempo mínimo de ocorrência entre dois eventos. Já as tarefas aperiódicas não possuem *deadline* e são disparadas em resposta a eventos que podem acontecer a qualquer momento. A sua ocorrência é tratada pela JVM do Mackinac com o mecanismo de um servidor esporádico (*sporadic server*). Para evitar a quebra do comportamento temporal da JVM, limita a execução de todas as tarefas aperiódicas ao tempo de execução do servidor esporádico e, no final deste, se houver uma tarefa aperiódica executando, esta será suspensa até a próxima ativação do servidor esporádico, onde o processamento é retomado do momento em que foi suspenso. A RTSJ ainda prevê mecanismos para executar o tratador de eventos para o caso de uma tarefa exceder o tempo de execução indicado em seu custo (*Cost Overrun Handler*, COH) e para o caso de uma tarefa perder o seu *deadline* (*Missed Deadline Handler*, MDH), porém em (SUN, 2004d) apenas é dito que a JVM suportará este mecanismo mas não detalha como será este mecanismo.

Com relação ao escalonamento das tarefas do sistema, a RTSJ não obriga que o teste de aceitação de tarefas seja implementado. Assim, qualquer tarefa nova pode ser aceita pelo escalonador, independente da possibilidade do aumento da carga de processamento do sistema, o que pode levar à perdas de *deadlines*. O projeto Mackinac incluirá um algoritmo para o teste de aceitação de acordo com as necessidades dos clientes iniciais

da *Sun Microsystems* para este projeto, porém com o passar do tempo novas versões da JVM podem incluir algoritmos diferentes dependendo das necessidades especificas da comunidade de desenvolvedores de sistemas tempo-real.

A RTSJ suporta vários tipos de memória (*Scoped Memory*, *Physical Memory*, *Raw Memory*, *Immortal Memory*, *Heap Memory*) sendo que na versão inicial da JVM do projeto Mackinac será suportado a *Heap Memory*, que é o tipo de memória utilizado nas *threads* com características SRT. Na *Heap Memory* existe *garbage collector* SRT, que interage com as *threads* SRT de forma a permitir a previsibilidade de aplicações SRT. Em (SUN, 2004d) não são dados maiores detalhes sobre a implementação do *garbage collector* SRT nem sobre o seu algoritmo. Em versões futuras a *Sun Microsystems* pretende dar suporte às memórias *Scoped Memory* e *Immortal Memory* as quais são reservadas para *threads* HRT. Não foi mencionada a intenção de implementação dos tipos *Physical Memory* e *Raw Memory* no projeto Mackinac.

No que diz respeito aos eventos assíncronos, a JVM do Mackinac dá suporte ao *AsyncEvent* e ao *AsyncEventHandler*. Assim que um evento ocorre, o código tratador do evento é escalonado e executado conforme a ordem do escalonador, evitando a quebra da integridade temporal do resto das aplicações tempo-real. Já a transferência assíncrona de controle (ATC) é implementada de forma a evitar os *deadlocks*, ou seja, uma *thread* não pode ser terminada se ela possui *locks* em recursos compartilhados. Para conseguir este objetivo a implementação de ATC da JVM do Mackinac utiliza o conceito de *deferred sections* que são trechos de código onde as exceções de interrupção não são imediatamente lançadas, isto é, o ATC é desligado enquanto o controle da *thread* estiver dentro de uma *deferred section*. Existem dois tipos de *deferred sections*: os blocos sincronizados (que utilizam a palavra reservada *synchronized*) e os métodos que não lançam a exceção *AsynchronouslyInterruptedException*. Sendo assim todos as bibliotecas existentes em Java não podem ter a sua execução interrompida.

Com relação à performance da execução da JVM do Mackinac, em comparação a JVM não tempo-real, o *throughput* é 10% menor em média. Segundo (SUN, 2004d), estes valores estão entre os limites aceitos pela comunidade tempo-real que diz que a média de redução fica entre 10% a 20% no *throughput* do sistema. Estes dados apresentados pela *Sun Microsystems* são preliminares, uma vez que a JVM Mackinac ainda não está disponível.

#### 4.2.1.2 JRate

O jRate (CORSARO; SCHMIDT, 2002) é uma implementação código aberto para Java tempo-real baseada na RTSJ. O jRate estende o ambiente de execução do compilador Java código aberto *GNU Compiler for Java* (GCJ) para proporcionar uma plataforma compilada para o desenvolvimento de aplicações compatíveis com a RTSJ.

A arquitetura do jRate é apresentada na Figura 4.2 onde pode ser notada a principal diferença da arquitetura do jRate (Figura 4.2a) para a arquitetura Java tradicional (Figura 4.2b): a ausência da JVM. Sendo assim, o *bytecode* Java não é interpretado, em vez disso, o código da aplicação compatível com o RTSJ é compilado antes da execução e código nativo de máquina pelo jRate.

Um lado negativo desta abordagem é que implementações compiladas da RTSJ como o jRate podem dificultar ou até mesmo impedir a portabilidade da aplicação, uma vez que esta precisa ser recompilada cada vez que for portada para uma nova arquitetura. Segundo Corsaro e Schmidt (2002), para os desenvolvedores de software para sistemas tempo-real embarcados este é um pequeno preço a ser pago pelos benefícios substanciais em performance se comparados com implementações baseadas em interpretadores ou compiladores JIT.

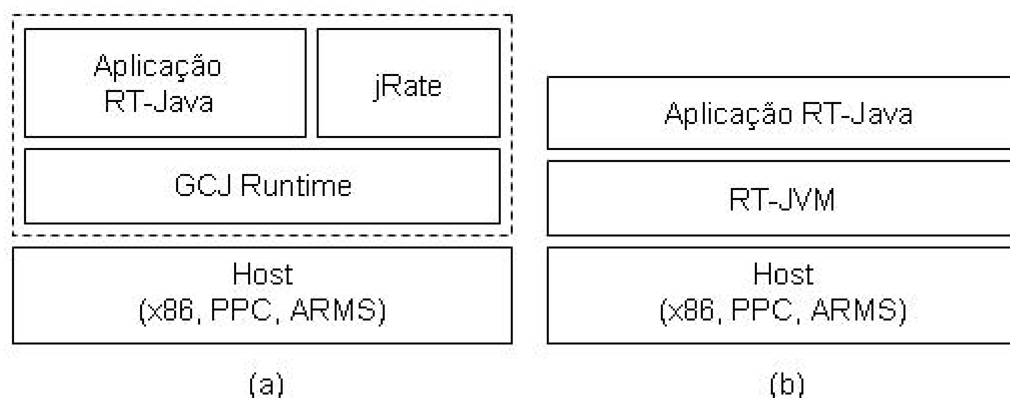


Figura 4.2: Arquitetura do jRate

O jRate suporta dois tipos de memória disponíveis na RTSJ: *Scoped Memory* e *Immortal Memory*. No jRate existem dois tipos diferentes de implementação da *Immortal Memory*, memória de tempo linear e memória de tempo variável, que podem ser definidos durante o início da execução da aplicação. Quanto à *Scoped Memory*, o jRate implementa um novo tipo de abordagem, que faz uma troca entre tempo de alocação e quantidade alocada de memória durante a criação. Esta área de memória é "zerada" na inicialização e também a cada vez que o contador de referências desta memória chegar a zero. Isto proporciona tempo de alocação constante para a criação de objetos nesta área de memória.

Com relação ao ambiente de execução, o jRate suporta *threads* de tempo-real do tipo *RealtimeThread* e *NoHeapRealtimeThead*, usando um escalonador preemptivo baseado em prioridades. Esta implementação simplesmente conta com suporte do escalonador preemptivo baseado em prioridades do sistema operacional tempo-real. O jRate implementa a classe *HighResolutionTime* e suas subclasses sendo que, dependendo da plataforma de hardware e do sistema operacional, a resolução obtida pode ser de nanosegundos até microssegundos.

Segundo Corsaro e Schmidt (2002), o jRate fornece uma implementação robusta e eficiente para o tratamento de eventos assíncronos. Esta implementação impede a inversão de prioridade e fornece um mecanismo de *dispatch* livre de bloqueios. Para isto são usadas filas de prioridades ordenadas pela elegibilidade da execução dos tratadores dos eventos. Esta abordagem é a aplicação dos formalismos apresentados em (CORSARO *et.al.*, 2001).

#### 4.2.1.3 Jamaica

A Jamaica Virtual Machine (JamaicaVM) é uma implementação da JVM com suporte a RTSJ voltada para o domínio de sistema tempo-real e de sistemas tempo-real embarcados. A JamaicaVM suporta grande parte da RTSJ, sendo que não são suportadas as classes *PhysicalMemory* e suas subclasses, além de não fornecer suporte total para a classe *PriorityCeilingEmulation*. Existem duas formas de execução de uma aplicação Java que utiliza a RTSJ: utilizando uma máquina virtual ou então utilizando um executável único em código nativo da plataforma alvo. Estas arquiteturas podem ser vista na Figura 4.3, juntamente com o conjunto de ferramentas disponível no ambiente Jamaica.

Segundo Siebert e Walter (2001) a JamaicaVM é a única implementação que fornece garantias HRT para todas as características da linguagem juntamente com uma alta eficiência de desempenho durante a execução, incluindo o gerenciamento de

memória dinâmica que é executado pelo *garbage collector*. Isso se deve ao fato da JamaicaVM garantir um tempo de pior caso para o atraso de execução para qualquer código (SIEBERT; WALTER, 2004).

Todas as *threads* executadas na JamaicaVM são *threads* tempo-real não existindo necessidade de distinguir *threads* tempo-real de *threads* não tempo-real. Qualquer *thread* de prioridade alta pode interromper uma *thread* de prioridade baixa com um atraso fixo (no pior caso) para a troca de contexto, sendo assim uma *thread* com prioridade alta pode inclusive interromper o *garbage collector*. Como a JamaicaVM executa todo o código Java com garantias HRT, as tarefas do sistema podem ser implementadas usando completamente a linguagem Java i.e. alocação de objetos, chamadas da funções em bibliotecas, etc.

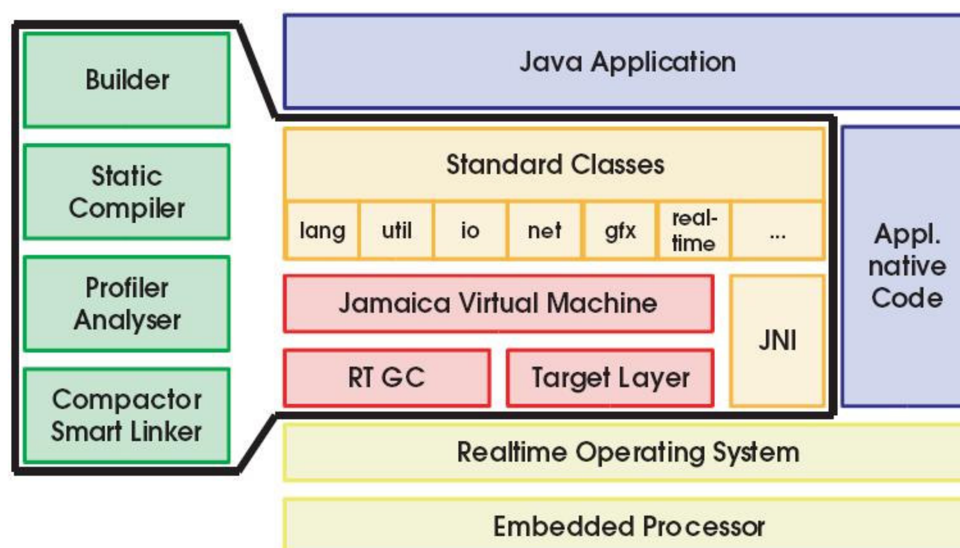


Figura 4.3: Arquitetura da JamaicaVM (SIEBERT; WALTER, 2004)

Com relação ao suporte à liberação automática de memória alocada, o *garbage collector* da JamaicaVM faz o seu trabalho de forma previsível dentro das *threads* da aplicação e é ativado quando a memória necessita ser alocada, podendo ser interrompido caso uma *thread* de prioridade alta se tornar ativa e necessitar ser executada (SIEBERT; WALTER, 2004). A implementação do *garbage collector* executa a liberação da memória executando uma série de pequenas liberações de 32 *bytes* de memória, isto é, a cada alocação de memória o *garbage collector* libera no máximo 32 *bytes* de memória alocada e não utilizada.

Para garantir um *footprint* mínimo, dependendo da plataforma alvo, o tamanho da JamaicaVM é de 128 KB de memória. A maior parte da memória necessária para guardar uma aplicação Java é tipicamente a área ocupada pelos arquivos de classe (*classfiles*) gerados pelo compilador Java, assim o Jamaica provê ferramentas para a análise dos *classfiles* que permite detectar e remover qualquer código que não pode ser acessado durante a execução, provocando a diminuição do tamanho dos *classfiles*. Segundo Siebert e Walter (2004), essa diminuição dos *classfiles* pode chegar a 90% do tamanho original mesmo para o código de aplicações não convencionais. Essa abordagem permite a utilização de bibliotecas complexas sem a preocupação com o *overhead* adicional de memória.

A JamaicaVM implementa a *Java Native Interface* (JNI) versão 1.2, que permite incluir código nativo legado em aplicações Java, ou ainda incluir rotinas de acesso ao hardware escritas em C ou em código de máquina. A previsibilidade temporal do código

Java não é afetada com a presença de código nativo na aplicação, isto inclui a execução do *garbage collector*.

Uma outra característica da JamaicaVM é a possibilidade de ligar o código da aplicação disponível nos *classfiles* juntamente com o código da máquina virtual dentro de um único executável. Isto possibilita que este executável possa ser armazenado em uma memória *Flash* ou uma memória ROM desde que todos os arquivos necessários pela aplicação Java estejam empacotados dentro deste executável, permitindo assim a não utilização de um sistema de arquivos para o sistema tempo-real embarcado.

Segundo Siebert e Walter (2004), a JamaicaVM é implementada em C usando o compilador GNU C, sendo que as *threads* são baseadas nas *threads* nativas do sistema operacional. A versão atual do Jamaica está disponível para o desenvolvimento nas plataformas Linux, SunOS/Solaris e Windows sendo que as aplicações são desenvolvidas para os seguintes sistemas operacionais tempo-real: VxWorks, QNX, EUROS, embOS, Linux/RT, NetOS, ThreadX, Linux. As aplicações também pode rodar nos sistemas operacionais não tempo-real Linux, SunOS/Solaris e Windows, contudo como estes não garantem determinismo, a JamaicaVM pode ser interrompida inesperadamente. Para finalizar, os processadores suportados pela JamaicaVM são: x86, PowerPC, Sparc, ERC32, StrongARM, NEC v850, Net+ARM (NS7520, NetSilicon) e C16x.

### 4.2.2 Processadores Java

Uma alternativa para o ambiente de execução de aplicações Java é a implementação da JVM em hardware, em outra palavras, implementar um processador que tem a capacidade de executar os *bytecodes* Java nativamente. Geralmente as implementações em hardware da JVM constituem em uma boa opção para o domínio de sistemas tempo-real embarcados devido ao seu melhor desempenho e menor consumo de energia se comparados com a implementação da JVM em software. Existem várias implementações de processadores Java tanto na área acadêmica como na área comercial que podem ser utilizados no desenvolvimento da plataforma que será utilizada no sistema tempo-real embarcado. Contudo neste trabalho serão apresentados apenas três processadores Java: PicoJava, JOP e FemtoJava.

#### 4.2.2.1 PicoJava

A primeira implementação de uma JVM em hardware foi um processador feito pela *Sun Microsystems* em 1997 que recebeu o nome de picoJava-I (O'CONNOR, 1997). O mercado alvo para este processador é o dos sistemas embarcados, sendo assim o *core* do picoJava-I tem um tamanho reduzido e pode ser configurado de acordo com as necessidades da aplicação. Dependendo dos objetivos de custo e desempenho, o projetista pode optar pela forma de I/O adequada, assim como os blocos funcionais para o ambiente de execução alvo da aplicação Java. Além disso o picoJava-I permite *cache* de dados e instruções de tamanho variável, e a opção da inclusão ou não de uma unidade de ponto flutuante. A arquitetura do picoJava-I é apresentada na Figura 4.4, onde pode ser observado que os blocos que são opcionais ao picoJava-I estão em destaque.

Com relação às instruções implementadas pelo picoJava-I, o seu *core* implementa um *pipeline* para melhorar o desempenho de processador como em um processador RISC normal. O picoJava-I não implementa o conjunto completo de instruções de uma JVM em hardware, ao invés disso, apenas algumas instruções que melhoram a execução de programas Java como uma adição de inteiros ou uma instrução para acessar o valor de um atributo de um objeto. Estas instruções geralmente levam de um a três ciclos para

serem executadas. As demais instruções que não são implementadas diretamente em hardware, mas que são consideradas críticas para o desempenho do sistema, são implementadas através de microcódigo ou máquinas de estado. Como exemplo dessa classe de instruções tem-se as instruções que fazem chamadas aos métodos, que são complexas e são operações críticas para o desempenho do sistema. As instruções restantes são capturadas e emuladas em software pelo *core* do picoJava-I. O operador *new* é um bom exemplo dessa classe de instruções, pois é um operador pouco comum de figurar nos *bytecodes* da aplicação e é muito mais complexo que uma instrução de chamada de método. O picojava-I implementa uma série de instruções adicionais, as quais não estão previstas na JVM, como instruções para gerenciamento da *cache*, instruções para comunicação com dispositivos de I/O que são mapeadas na memória, instruções para desligar (*power-down*) o sistema, entre outras.

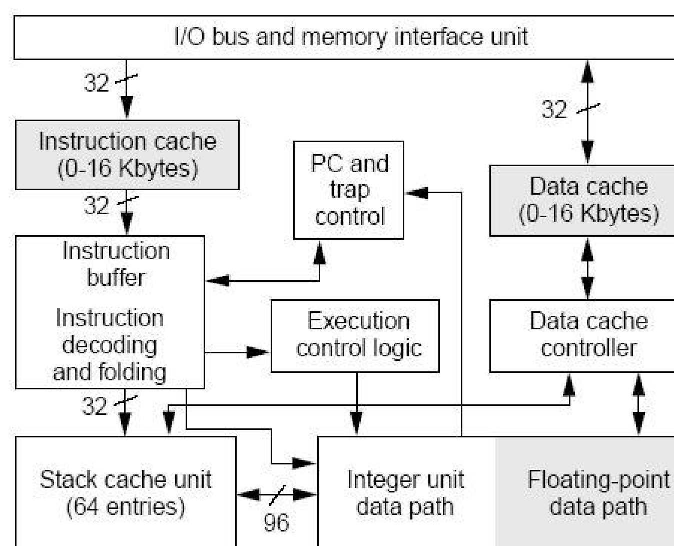


Figura 4.4: Arquitetura do processador picoJava-I (O'CONNOR, 1997)

O *pipeline* do picoJava-I possui quatro estágios para executar as instruções. O *core* traz as instruções do *cache* de instruções e as coloca em um *buffer* de instruções no estágio de busca. No estágio de decodificação, o *core* decodifica a instrução no início do *buffer* de instruções, dividindo-a nas suas respectivas partes e acessa a *cache* da pilha. A instrução pode demorar um ou mais ciclos para ser executada (acessando os dados na *cache*) no estágio de execução. Finalmente o *core* escreve o resultado novamente na pilha no estágio de *write-back*. Existe uma forma de passar diretamente o resultado da computação da instrução anterior para a instrução atual que está no estágio de execução, permitindo assim que a espera pelo término do estágio de *write-back* seja eliminada, aumentado o desempenho do processador. O picoJava-I não possui uma lógica de predição de desvios, ele simplesmente assume que todos os desvios do código (*branches*) não serão pegos, ou seja, o código irá executar sem saltos. Isso não é um problema grave, pois o *pipeline* é curto (apenas quatro estágios) e a penalidade é apenas de dois ciclos caso o desvio realmente for pego.

A unidade de ponto flutuante disponível no picoJava-I suporta números de ponto flutuante com precisão simples e dupla, compatíveis com o padrão IEEE 754. Se a unidade de ponto flutuante não tiver sido incluída durante a geração do *core* do picoJava-I, cada instrução de ponto flutuante é capturada e desviada para uma rotina de software que pode emular a instrução usando instruções de números inteiros que estão disponíveis no processador.

A *Sun Microsystems* criou uma outra versão para este processador, chamada de picoJava-II (SUN, 1999a; SUN, 1999b). O picoJava-II é baseado no seu antecessor herdando a maioria de suas características. As principais melhorias no processador dizem respeito ao mecanismo de *folding* e ao aumento do tamanho do *pipeline*, que permite que a freqüência do processador seja aumentada.

O *pipeline* do picoJava-II possui seis estágios: busca, decodificação, registro, execução, *cache* e *write-back*. No primeiro estágio as instruções são buscadas ou da *cache* de instruções ou da memória externa. No segundo estágio é feita a decodificação e o agrupamento das instruções na *Instruction Folding Unit*. No estágio de registro é feita a busca dos operandos na pilha e determinadas as condições de uso e carga, as condições de *bypass* e as condições de perda de dados na *cache*, sendo que esta lógica de operação fica localizada na Unidade de Controle de Registro. O próximo estágio é o de execução onde são feitas as operações aritméticas ou os cálculos de endereço para a leitura ou gravação em memória. Seguindo o *pipeline* vem o estágio de *cache*, onde os dados da *cache* são acessados e capturados no final do ciclo. Por fim o resultado da instrução é escrito novamente na pilha no estágio de *write-back*. Maiores detalhes sobre a arquitetura do picoJava-II pode ser encontros em (SUN, 1999a) e (SUN, 1999b).

#### 4.2.2.2 Java Optimized Processor

O *Java Optimized Processor* (JOP) é uma implementação em hardware da JVM com um tempo de execução curto e previsível para os *bytecodes*. O JOP é implementado como um *soft core* em um FPGA e foi desenvolvido para aplicações em sistemas tempo-real embarcados. Sendo assim seu projeto procurou respeitar algumas restrições: todos os aspectos da arquitetura obrigatoriamente necessitavam ter previsibilidade temporal; o processador necessitava ser pequeno o suficiente para se ajustar em um dispositivo de FPGA de baixo custo. O JOP implementa todos os *bytecodes* da especificação CLDC da JVM na plataforma J2ME.

Segundo Schoeberl (2004c), o processador JOP possui uma abordagem para o mapeamento dos *bytecodes* Java em micro instruções nativas do JOP, assim como um *pipeline* de execução em sua arquitetura, com um único ciclo de execução para as micro instruções do micro código. Basicamente o JOP difere do picoJava, com relação ao mapeamento dos *bytecodes* em micro instruções, pelo fato de que cada *bytecode* é traduzido em um endereço de início de um micro código que implementa a funcionalidade, enquanto no picoJava apenas um subconjunto de *bytecodes* é implementado em micro código e os demais *bytecodes* são capturados e executados via software. O *datapath* do processador JOP pode ser observado na Figura 4.5.

No primeiro estágio do *pipeline* é feita a busca pelos *bytecodes* Java, onde são traduzidos em endereços no microcódigo, assim como os desvios são decodificados e executados neste estágio. Um *bytecode* que foi buscado no primeiro estágio resulta em um salto absoluto no microcódigo para que, no segundo estágio do *pipeline*, seja feita a busca da instrução na memória de microcódigo do JOP e seu respectivo salto no microcódigo. No terceiro estágio acontece a decodificação da microinstrução assim como a geração do endereço da pilha na RAM. No estágio de execução, as operações são realizadas em dois registradores (TOS e TOS-1) que representam os dois elementos mais ao topo da pilha, assim não existe a necessidade de um estágio de *write-back* ou a utilização de *forwarding* (SCHOEBERL, 2004b).

Como o *pipeline* do JOP é curto, o preditor de desvios não foi implementado provocando um atraso de quatro ciclos na execução das instruções quando houver a necessidade de executar uma instrução de desvio. Estes espaços do *pipeline* podem ser preenchidos com instruções *nop*.

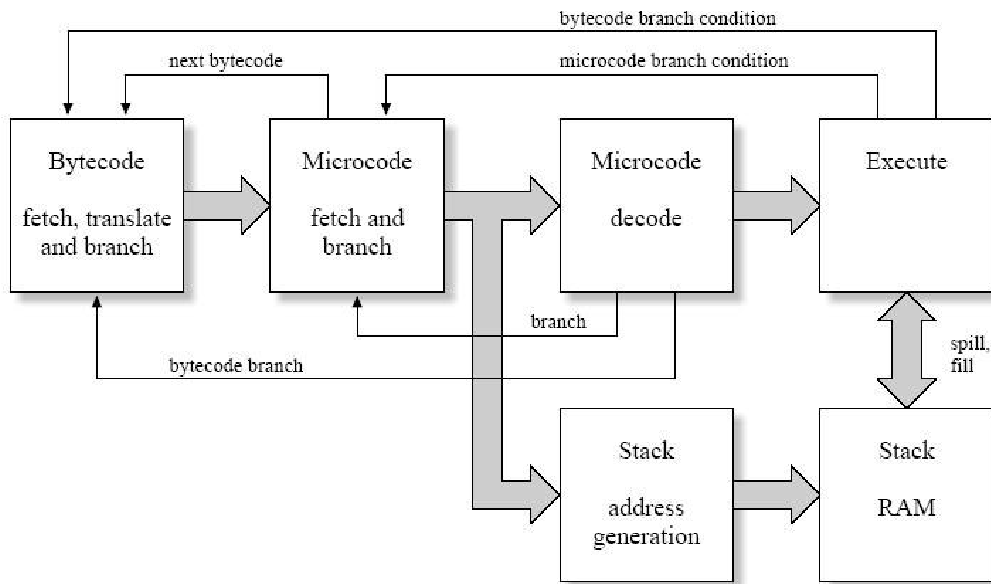


Figura 4.5: *Datapath* do processador JOP (SCHOEBERL, 2004b)

A pilha do JOP é implementada em uma pequena memória interna no FPGA, com os dois elementos mais ao topo da pilha implementados na forma de registradores. Segundo Schoberl (2004b), essa memória pequena que permite o acesso em um único ciclo pode ser vista como uma *cache* de dados. Todavia a troca de dados entre a pilha da memória interna e a pilha em memória externa não é feita automaticamente como no picoJava, ela é controla explicitamente pelo programa nas chamadas e retorno de métodos ou na troca de contexto das *threads*.

Com relação às características necessárias para suportar um sistemas de tempo-real, segundo (SCHOEBERL, 2004b), o JOP implementa:

- **Interrupção de relógio:** o propósito principal da interrupção de relógio é representar o tempo e a liberação de tarefas periódicas ou disparadas em determinados instantes de tempo. A abordagem adotada é de fazer a programação deste temporizador na liberação de uma tarefa para execução, sendo de responsabilidade do escalonador a reprogramação deste temporizador após cada ocorrência desta interrupção;
- **Interrupções Externas:** são as interrupções de hardware que representam os eventos assíncronos associados com uma *thread*. Isto significa que o código associado a estes eventos são controlados pelo escalonador;
- **Interrupções de Software**: são interrupções geradas por software como, por exemplo, acesso ilegal a uma área de memória ou uma divisão por zero e são representadas por exceções em Java. Estes eventos assíncronos são suportados da mesma maneira que as interrupções externas;
- **Mecanismo de Despacho**: para realizar uma troca de contexto rápida, o JOP permite salvar apenas o topo da pilha, assim o contexto pode ser rapidamente restaurado assim que a tarefa for novamente ativada;
- ***Garbage Collector***: não é implementado no JOP porque, segundo (SCHOEBERL, 2004b), o uso da *heap* é evitado em sistemas HRT. Sem um *garbage collector* a representação de um objeto em memória é mais simples economizando em memória, um item importante em se tratando de sistemas tempo-real embarcados.

#### 4.2.2.3 FemtoJava

O FemtoJava (ITO, 2000) é um processador desenvolvido tendo como foco o domínio de aplicação que necessita apenas de um simples microcontrolador mas que deve permitir que um programa Java execute com o desempenho desejado. O seu funcionamento é consistente com a especificação da arquitetura da JVM (LINDHOLM; YELLIN, 1997).

A arquitetura do FemtoJava é uma arquitetura *Harvard* com um conjunto reduzido de instruções que utiliza blocos simples como pode ser visto na Figura 4.6. O FemtoJava é composto por uma unidade de processamento baseada em um arquitetura de pilha, possui memórias RAM e ROM integradas, portas de I/O mapeadas em memória, um mecanismo de interrupções baseado em dois níveis de prioridade e dois temporizadores. As portas de I/O, os temporizadores e o mecanismo de interrupções, segundo ITO *et al.* (2001), são estrutura importantes para sistema embarcados embora estas características estejam ausentes na especificação da JVM.

O FemtoJava faz a emulação da pilha em memória RAM, sendo que as duas posições mais ao topo da pilha são mapeadas em registradores, assim como é feito no picoJava e no JOP. Essa abordagem possibilita um ganho de desempenho visto que uma grande parte das instruções executadas em um programa Java opera com topo e/ou o topo-1 elementos da pilha.

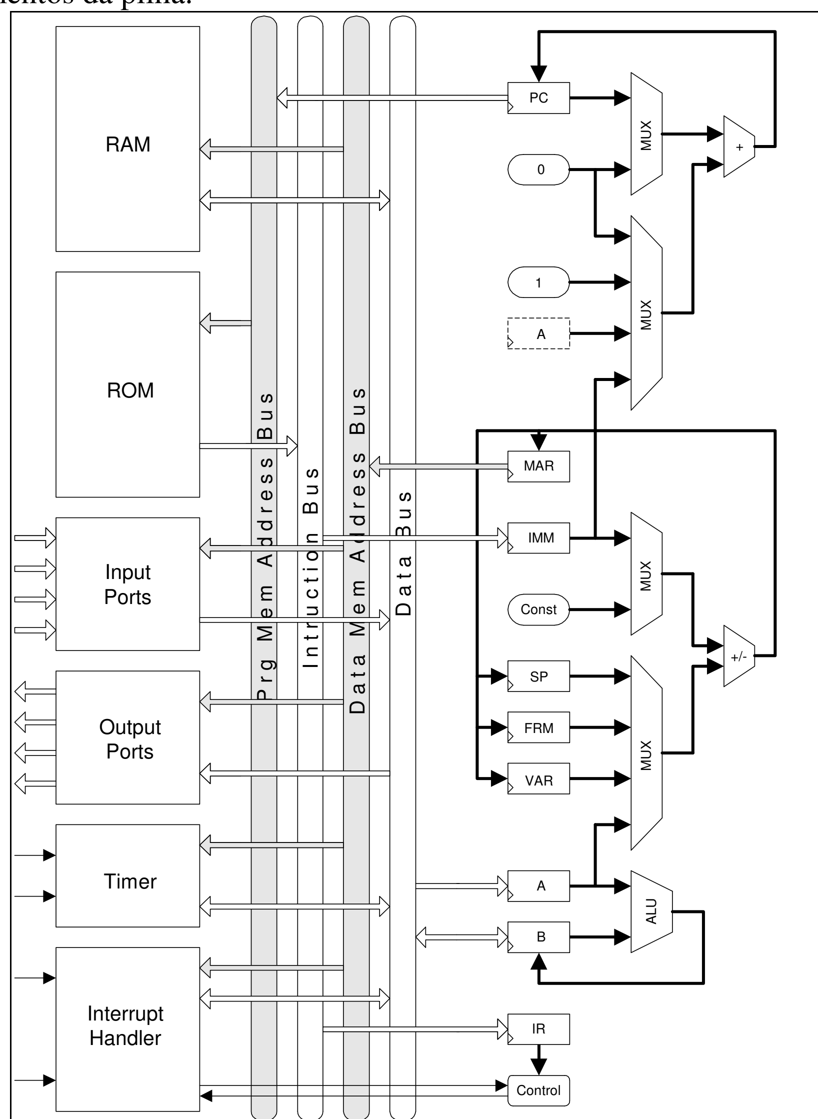


Figura 4.6: Arquitetura do FemtoJava (MATTOS; CARRO, 2003)

A memória do FemtoJava está dividida em duas partes, a memória de dados, que pode ser vista na Figura 4.7a, e a memória de programa que pode ser vista na Figura 4.7b. Na memória de dados os endereços iniciais, da posição 00H até a posição 10H, contém alguns registradores mapeados em memória, que tem a finalidade de possibilitar a programação das interrupções e dos temporizadores assim como as permitir as operações de I/O. A partir do endereço 10H a memória é utilizada para guardar as constantes e os campos dos objetos, para a pilha do FemtoJava, e como conseqüência para a alocação dos *frames* dos métodos. Já a memória de instruções, a primeira parte é reservada para o código necessário para a chamada dos métodos de manipulação das interrupções e dos temporizadores, a memória restante guarda o código de todos os métodos da aplicação. Para otimizar a chamada dos métodos e a alocação dos seus respectivos *frames*, no cabeçalho dos métodos são colocadas informações sobre o total de parâmetros e o total de variáveis locais, que permitem que o cálculo do ponteiro da pilha seja feito sem a necessidade de acesso à pilha, restaurando o *frame* original corretamente.

Com relação às instruções, do total de 226 instruções suportadas pela JVM, o FemtoJava suporta apenas 68 instruções que estão elencadas na Tabela 4.1. Este subconjunto inclui instruções necessárias para executar operações básicas sobre números inteiros e sobre a pilha, manipulação de vetores, saltos condicionais e incondicionais, execução de métodos estáticos e acesso a campos estáticos das classes. O FemtoJava pode executar apenas o código de métodos de classe porque seu conjunto de instruções possui apenas as instruções *invokestatic*, *return* e *ireturn* como instruções para manipulação de métodos. Segundo ITO *et al.* (2001), esta limitação não é séria porque a maioria do software para sistemas embarcados não necessita de alocação de objetos complexos durante a sua execução.

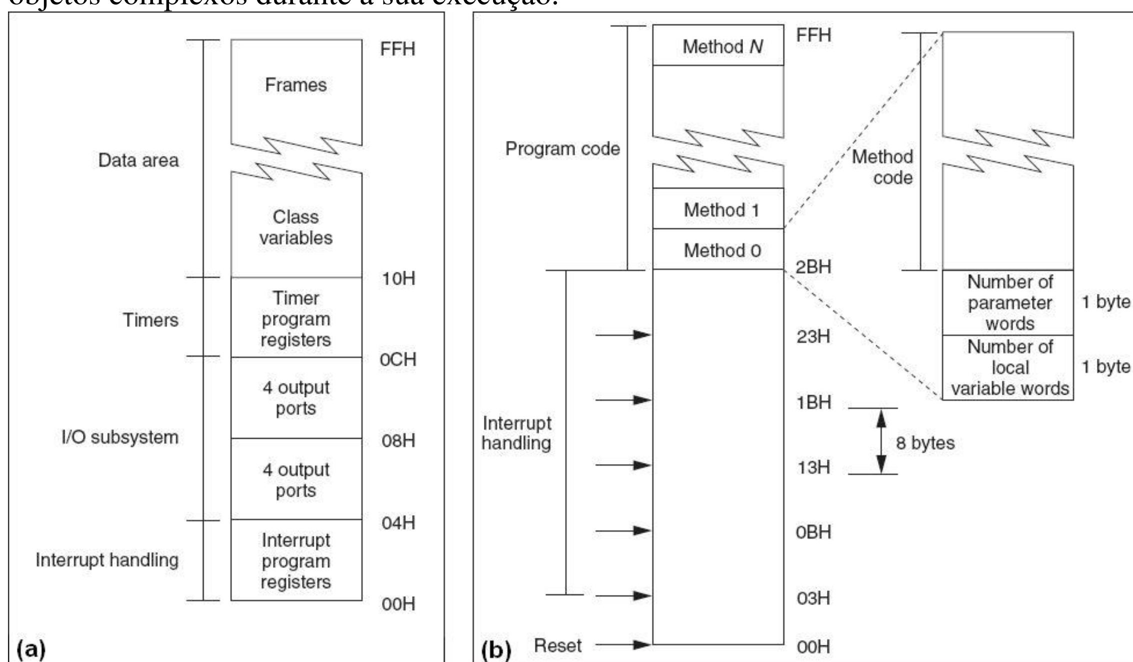

Figura 4.7: Organização da memória do FemtoJava (ITO, 2000)

Todas as instruções suportadas pelo *core* FemtoJava podem executar em, no máximo, 14 ciclos. Isso se deve ao fato de algumas instruções acessarem a memória, que é mais lenta que um processador, em termos de freqüência de utilização. O FemtoJava é gerado pela ferramenta de CAD SASHIMI (ITO, 2000) descrita em detalhes na seção 3.3, sendo assim o tamanho da sua máquina de controle é diretamente

proporcional às instruções utilizadas na aplicação que está servindo de entrada para o SASHIMI. Em outras palavras, apenas as instruções utilizadas na aplicação receberão suporte de hardware, fazendo com que o tamanho do FemtoJava gerado diminua, se comparado com a versão que suporta todas as instruções. Como conseqüência desta diminuição de complexidade da máquina de controle, a freqüência máxima de utilização do FemtoJava também varia, pois com menos instruções, menor o tamanho da máquina de controle e maior pode ser a freqüência de utilização do processador.

Tabela 4.1: Lista de instruções suportadas pelo FemtoJava

| **Tipo de instrução** | |
|---|---|
| Aritméticas e lógicas | iadd, isub, imul, ineg, ishr, ishl, iushr, iand, ior, and ixor |
| Controle de fluxo | goto, ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, return, ireturn, invokestatic |
| Pilha | iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, bipush, pop, pop2, dup, dup_x1, dup_x2, dup2, dup2_x1, swap |
| Load/Store | iload, iload_0, iload_1, iload_2,iload_3, istore, istore_0, istore_1, istore_2, istore_3 |
| Vetor | iaload, baload, caload, daload, iastore, bastore, castore, sastore, arraylength |
| Estendidas | load_idx, store_idx, sleep |
| Outras | nop, iinc, getstatic, putstatic |

# 5 PROPOSTA DE INTEGRAÇÃO DE FASES NO PROJETO SEEP

Neste capítulo será apresentada a proposta de integração das fases no projeto SEEP. A Figura 5.1 apresenta as etapas do ciclo de projeto proposto no SEEP, sendo que as etapas destacadas representam as fases que são integradas com a proposta apresentada neste trabalho.

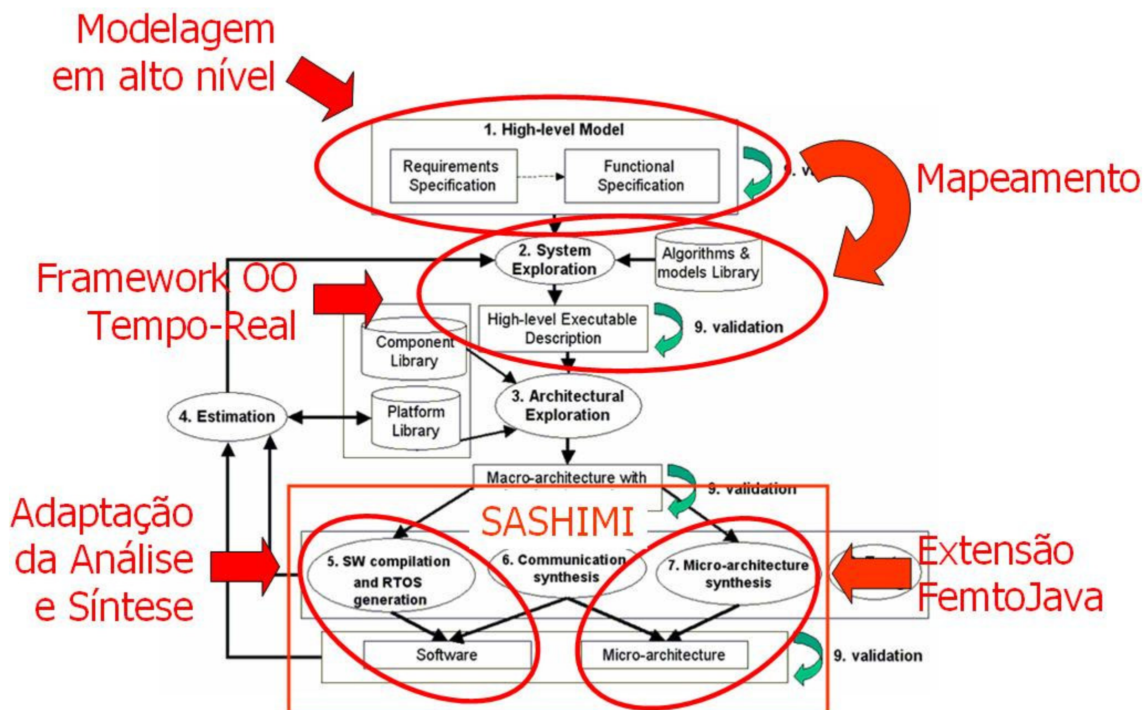


Figura 5.1: Contextualização do presente trabalho no ciclo de projeto SEEP

Nas etapas de modelagem de alto nível a especificação dos requisitos e funcionalidades do sistema tempo-real embarcado são feitas utilizando a RT-UML através da ferramenta Artisan Real-Time Studio (ARTISAN, 204). Durante a exploração do sistema é feita a seleção dos componentes que irão ser utilizados na implementação do sistema, onde é selecionado o *framework* tempo-real orientado a objetos proposto, o qual está contido na biblioteca de componentes disponíveis para utilização. A descrição executável de alto-nível é feita utilizando o *framework* tempo-real orientado a objetos baseado na RTSJ. Então é feito o mapeamento dos modelos RT-UML para os elementos do *framework*. As etapas de geração do software da aplicação e do RTOS, assim como a síntese da micro-arquitetura é feita utilizando a ferramenta SASHIMI que necessitou ser adaptada de forma a suportar o *framework* proposto. Também é importante salientar que a micro-arquitetura também teve que ser adaptada,

pois a plataforma FemtoJava não possuía suporte a aplicações tempo-real, requisito que foi implementado através da inclusão de um *Real-Time Clock* (RTC).

Nas sessões a seguir são detalhadas cada uma destas etapas realizadas durante o desenvolvimento desta dissertação.

## 5.1 Descrição do Framework Tempo-Real Orientado a Objetos

O *framework* proposto neste trabalho tem como objetivo facilitar o mapeamento de modelos orientado a objetos expressos através da RT-UML em código fonte escrito na linguagem Java. Este *framework* permite a expressão de elementos comuns em sistemas de tempo-real como tarefas, escalonadores, temporizadores, entre outros elementos. A sua concepção foi baseada na *Real-Time Specification for Java* (RTSJ), pois considera-se que a mesma apresenta uma forma clara de expressar os requisitos temporais na linguagem de programação Java. A Figura 5.2 mostra o diagrama de classe do *framework* proposto, onde podem ser observados os relacionamentos entre todas as classes que compõem o *framework*. Tendo em vista a complexidade do *framework* os elementos foram classificados em quatro grupos, que serão detalhados nas subseções a seguir.

Figura 5.2: Diagrama de classes do framework tempo-real orientado a objetos

A API proposta neste trabalho se baseia em um sub-conjunto de classes da RTSJ que foram escolhidas por serem consideradas de uso comum no desenvolvimento de sistemas tempo-real. Contudo, a RTSJ é uma especificação mais ampla que cobre uma série de aspectos igualmente importantes em sistemas de tempo-real, como definições de diferentes tipos de memórias com seus requisitos para alocação e liberação de memória, assim como a definição de eventos assíncronos juntamente com o código executado quando da ocorrência de um evento deste tipo.

### 5.1.1 Classes para Modelagem de Tempo

Neste grupo encontram-se as classes para a modelagem de valores de tempo e do relógio de tempo-real do sistema, como pode ser observado no diagrama de classes

detalhado da Figura 5.3. Neste diagrama podemos notar que existem basicamente duas formas de modelar um valor de tempo com a resolução de nanosegundos, assim como existe uma classe que representa o relógio de tempo-real do sistema.

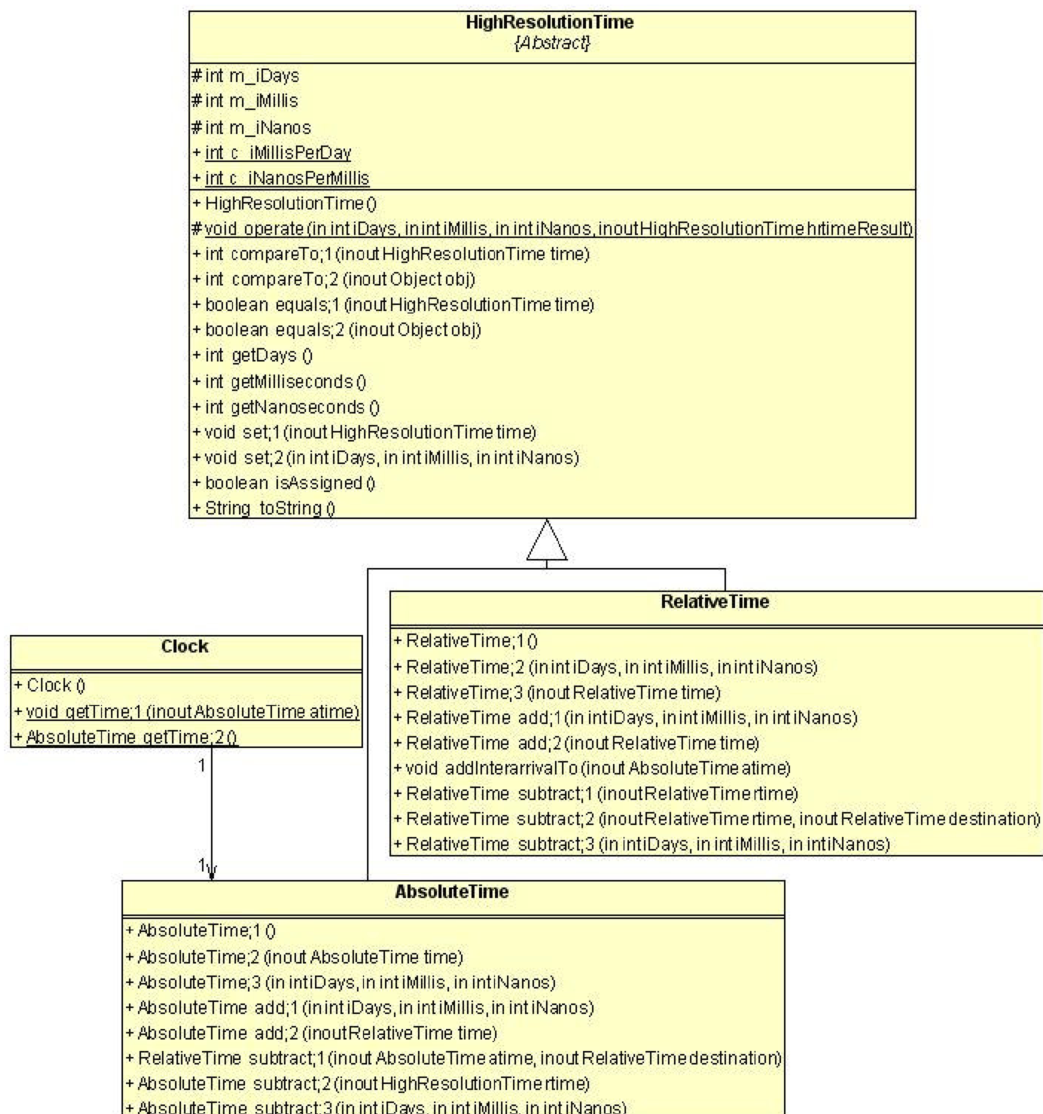

Figura 5.3: Diagrama de classes detalhado para modelagem de tempo

A classe *HighResolutionTime* representa a classe básica para todos os valores de tempo (com resolução de nanosegundos) que são modelados nesta versão inicial do *framework*, devendo servir como base para as futuras classes que representam os valores de tempo que por ventura vierem a ser incorporadas no *framework*. Nesta classe temos duas constantes – c_iMillisPerDay e c_iNanosPerMillis – que representam respectivamente a quantidade de milisegundos em um dia e a quantidade de nanosegundos em um milisegundo. Os outros atributos da classe *HighResolutionTime* são 3 inteiros – m_iDays, m_iMillis, m_iNanos – que representam respectivamente valores de dia decorridos, milisegundos decorridos desde a zero hora dia e nanosegundos decorridos desde o início do milisegundo. As operações suportadas sobre os valores de tempo são implementadas na forma de métodos que são descritos na Tabela 5.1.

Tabela 5.1: Métodos da classe *HighResolutionTime*

| **Método** | **Descrição** |
|---|---|
| *operate()* | Método estático que realiza uma soma algébrica sobre o valor de tempo, armazenando o resultado em um objeto *HighResolutionTime* passado como parâmetro. |
| *compareTo()* | Permite comparar um objeto *HighResolutionTime* com outro objeto retornando: <br>▪ 0 caso os dois tempos representados nos objetos sejam iguais; <br>▪ 1 caso o objeto represente um tempo maior que o objeto passado como parâmetro; <br>▪ -1 caso o objeto represente um tempo menor que o objeto passado como parâmetro. |
| *equals()* | Retorna um valor booleano indicando que o objeto apresenta o mesmo valor de tempo que o objeto passado como parâmetro. |
| *getDays()* | Retorna um inteiro representando a quantidade de dias pertencentes ao tempo representado no objeto. |
| *getMilliseconds()* | Retorna um inteiro representando a quantidade de milisegundos pertencentes ao tempo representado no objeto. |
| *getNanoseconds()* | Retorna um inteiro representando a quantidade de nanosegundos pertencentes ao tempo representado no objeto. |
| *set()* | Permite atribuir um valor de tempo a um objeto. |
| *isAssigned()* | Retorna um valor booleano indicando se o objeto já recebeu ou não um valor de tempo. |
| *toString()* | Utilizado apenas para a simulação funcional, este método retorna o valor de tempo representado através de uma cadeia de caracteres. |

A classe *AbsoluteTime* é utilizada para representar um valor de tempo absoluto como, por exemplo, 18/06/2004 15:49:06.589. O atributo m_iDays conterá o valor que represente a quantidade de dias decorridos desde 01/01/1970, o atributo m_iMillis conterá a quantidade de milisegundos decorridos desde a 00:00:00 do respectivo dia, assim como o atributo m_iNanos irá conter a quantidade de nanosegundos que transcorreram desde o início do milisegundo. Esta classe basicamente possui dois métodos especializados para realizar operações sobre o valor de tempo representado pelo objeto, são eles: *add()* e *subtract()*. O método *add()* permite adicionar um valor de tempo ao tempo absoluto representado no objeto corrente, sendo que este possui duas variações com diferentes tipos de parâmetros. De forma análoga o método *subtract()* permite subtrair um valor tempo do tempo absoluto, assim como possuir duas variações com parâmetros distintos. Se o subtraendo for maior que o minuendo, então resultado da operação é o valor zero.

A classe *RelativeTime* define objetos que representam valores de tempo relativos a um determinado instante. Assim como na RTSJ, geralmente os valores de tempo representado por objetos desta classe são relativos ao instante de tempo corrente do sistema. Da mesma forma que a sua classe irmã *AbsoluteTime*, a classe *RelativeTime* possui métodos para as operações de adição e subtração. Contudo existe uma diferença semântica no método *subtract()*, pois como a classe *RelativeTime* representa um valor relativo de tempo, o resultado da subtração pode ser menor que zero. Além destes métodos ainda possui o método *addInterarrivalTo()* que foi incluído para manter a compatibilidade com a classe especificada na RTSJ e permite adicionar o tempo relativo a um objeto *AbsoluteTime*.

A última classe deste grupo é a classe *Clock* que representa o relógio de tempo-real do sistema. Esta é uma classe que permite a interface com o relógio de tempo-real adicionado à arquitetura do FemtoJava e descrita na seção 5.3. Basicamente a classe *Clock* não necessita ser instanciada pois possui um único método estático, que retorna o tempo absoluto corrente do sistema desde o momento em que o mesmo foi ligado.

### 5.1.2 Classes para Modelagem das Tarefas

Este grupo possui as classes utilizadas na modelagem das tarefas em um sistema tempo-real. A Figura 5.4 mostra o diagrama de classes detalhado para o conjunto de classes para a representação das tarefas. A classe principal deste grupo é a classe *RealtimeThread*, que representa uma *thread* no sistema tempo-real embarcado. Assim como em um RTOS comum, uma *thread* possui parâmetros que definem o seu comportamento e.g. se a *thread* é periódica ou não, o seu tempo máximo de resposta (*deadline*), a sua prioridade, entre outros parâmetros.

Um objeto da classe *RealtimeThread* representa um objeto ativo no sistema, que é um objeto que tem a sua própria *thread* de controle. A classe *RealtimeThread* estende a classe padrão *Thread* da linguagem Java. Os dois métodos principais desta classe são *mainTask()* e *exceptionTask()*, que representam respectivamente o código da tarefa e o tratador de exceção de perda de *deadline* de uma *thread*. Como estes métodos são abstratos na classe *RealtimeThread*, eles necessitam ser implementados nas subclasses que a estendem, em outras palavras, a classe *RealtimeThread* não pode ser instanciada diretamente. Ao invés disso, quando o programador desejar representar uma *thread* no sistema ele deve criar uma classe que estenda a classe *RealtimeThread*, sobrescrevendo os métodos *mainTask()* e *exceptionTask()* provendo a funcionalidade desejada para a tarefa. Associados a um objeto *RealtimeThread* estão alguns objetos que representam os parâmetros de liberação e de escalonamento de uma *thread*, que são, respectivamente, objetos das classes *ReleaseParameters* e *SchedulingParameters*.

A classe *ReleaseParameters* indica os parâmetros utilizados pelo escalonador para liberar a execução de uma *thread* e indicam se é uma *thread* periódica ou aperiódica. A *thread* periódica é aquela que possui um objeto *PeriodicParameters* associado, sendo que neste objeto estão especificados características como:

- Tempo de início da execução *thread* (*hrtStart*);
- Tempo final de execução da *thread* (*hrtEnd*);
- Período de ativação da *thread* (*hrtPeriod*), ou seja, a sua freqüência de execução;
- O custo computacional da *thread* (*hrtCost*), em outras palavras, o WCET da *thread*;
- O tempo máximo de resposta (*hrtDeadline*).

Quando a *thread* for do tipo aperiódica ou do tipo esporádica, ou seja, não exista um tempo predeterminado para a sua ocorrência, o objeto *RealtimeThread* deve possuir um objeto do tipo *AperiodicParameters* ou *SporadicParameters* associado como parâmetro de liberação. Ambos os parâmetros possuem o custo computacional (*hrtimeCost*) e o tempo máximo de resposta *(hrtimeDeadline)* como características. Assim, basicamente, a diferença entre um objeto *AperiodicParameters* e *SporadicParameters* é a existência de mais uma característica no objeto *SporadicParameters* indicando o tempo mínimo de ocorrência entre duas ativações da *thread* (*hrtimeMinInterarrival*).

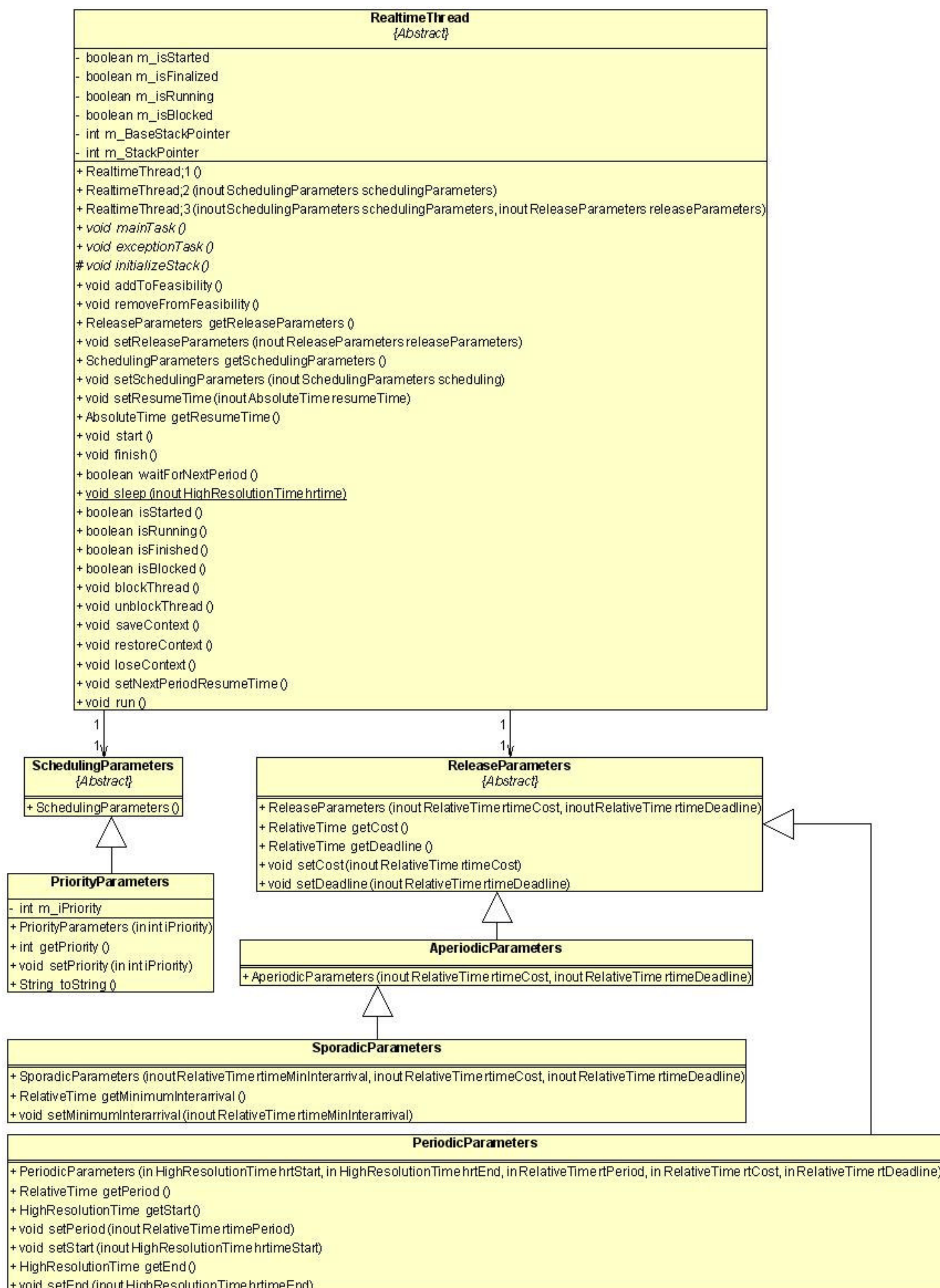

Figura 5.4: Diagrama de classes detalhado para a modelagem das tarefas

A classe *SchedulingParameters* é utilizada como base para os objetos que representam os parâmetros da tarefa que são usados pelo escalonador. Na versão inicial do *framework* existe apenas um tipo de parâmetro de escalonamento (*PriorityParameter*) que é utilizado pelo escalonador de prioridade fixa (*PriorityScheduler*) e representa apenas um número inteiro para a prioridade.

Os estados de uma *thread* são representados pelos seguintes atributos da classe *RealtimeThread*:

- *m_isStarted* : indica se a *thread* já foi iniciada ou não;
- *m_isFinalized* : indica que a *thread* foi finalizada;
- *m_isRunning*: indica se a *thread* está executando;
- *m_isBlocked*: indica se a *thread* está bloqueada aguardando a liberação de um recurso compartilhado.

Com relação ao contexto de uma *thread*, os atributos *m_BaseStackPointer* e *m_StackPointer* indicam respectivamente, a base e o topo da pilha da *thread* do objeto ativo. Quando existe uma troca de contexto, o topo da pilha atual é salvo no atributo *m_StackPointer* do objeto *RealtimeThread* da *thread* corrente, e então o novo topo da pilha para a próxima *thread* é restaurado carregando o atributo *m_StackPointer* do respectivo objeto *RealtimeThread.*

Dando continuidade na descrição da classe *RealtimeThread*, a Tabela 5.2 apresenta a descrição da funcionalidade dos métodos desta classe.

Tabela 5.2: Métodos da classe *RealtimeThread*

| **Método** | **Descrição** |
|---|---|
| *addToFeasibility()* | Adiciona a *thread* na lista de *threads* do escalonador. A tarefa pode ser rejeitada caso a análise de escalonabilidade do escalonador não permita que uma nova *thread* seja adicionada no sistema. |
| *removeFromFeasibility()* | Remove a *thread* na lista de *threads* do escalonador. |
| *getReleaseParameters()* | Retorna o objeto contendo os parâmetros de liberação de uma *thread.* |
| *setReleaseParameters()* | Atribui um objeto contendo os parâmetros de liberação para uma instância da classe *RealtimeThread.* |
| *getSchedulingParameters()* | Retorna o objeto contendo os parâmetros de escalonamento de uma *thread.* |
| *setSchedulingParameters()* | Atribui um objeto contendo os parâmetros de escalonamento para uma *thread.* |
| *setResumeTime()* | Atribui o tempo de liberação para a execução de uma *thread*. |
| *getResumeTime()* | Retorna o tempo de liberação para a execução de uma *thread.* |
| *start()* | Altera o estado de uma *thread* para iniciada, contudo é o escalonador quem indica em que momento o código da *thread* começa a ser executado. |
| *finish()* | Altera o estado de uma *thread* para finalizada, fazendo com que a mesma não seja mais escalonada para execução. |
| *waitForNextPeriod()* | Faz com que a *thread* corrente espere o próximo período de ativação, incrementando o valor do tempo de liberação com o período da tarefa e, em seguida, invocando a execução do escalonador. É utilizado em tarefas periódicas. |
| *sleep()* | Faz com que a *thread* entre durma durante o tempo especificado |
| *isStarted()* | Retorna um valor booleano indicando se a *thread* está iniciada ou não. |

| | |
|---|---|
| *isRunning()* | Retorna um valor booleano indicando se a *thread* está executando ou não. |
| *isFinished()* | Retorna um valor booleano indicando se a *thread* está finalizada ou não. |
| *isBlocked()* | Retorna um valor booleano indicando se a *thread* está bloqueada ou não. |
| *block()* | Altera o estado de uma *thread* para bloqueada. |
| *unblock()* | Altera o estado de uma *thread* para desbloqueada. |
| *saveContext()* | Salva o contexto da *thread*. |
| *restoreContext()* | Restaura o contexto da *thread*. |
| *loseContext()* | Altera o estado de uma *thread* para não-executando. |
| *Run()* | Método que deve obrigatoriamente ser implementado em classes que estendem a classe *Thread*. |

Embora o *framework* proposto seja fortemente baseado na RTSJ, algumas classes possuem diferenças na sua implementação em relação ao que foi proposto na RTSJ. A classe *RealtimeThread* é uma delas pois propõe a utilização dos métodos *mainTask()* e *exceptionTask()* para a implementação de uma *thread* contendo um código para a parte principal e outro para o tratamento de exceções de perda de *deadline* da *thread*. O método *exceptionTask()* substitui o uso de um objeto *AsyncEventHandler* que, segundo a RTSJ, deveria ser passado para o objeto *ReleaseParameters*.

Segundo a semântica da RTSJ, se um *deadline* é perdido, o objeto *AsyncEventHandler* é colocado na lista de escalonamento e a sua execução é controlada pelo escalonador. A semântica para a *exceptionTask()* é parecida, contudo é o escalonador quem verifica se a *thread* perdeu o *deadline* e executa o método *exceptionTask()*. Esta abordagem permite a execução do código do tratamento da exceção dentro do contexto do objeto, ou seja, qualquer um dos atributos do objeto (privados, protegidos e/ou públicos) podem ser acessados pelo código, a fim de deixar o objeto *RealtimeThread* em um estado consistente. Isso não acontece com a abordagem adotada pela RTSJ pois o tratador de eventos é um outro objeto, sendo assim, ele tem acesso apenas aos atributo públicos do objeto *RealtimeThread*.

Esta modificação foi proposta para dar suporte a algoritmos de escalonamento que utilizam o conceito de pares de tarefas como o algoritmo de escalonamento *Time-Aware Fault-Tolerant* (TAFT) (NETT *et al.*, 2002). Outro objetivo destas modificações é permitir uma maior legibilidade no código fonte de uma *thread* do sistema tempo-real embarcado, pois os códigos da parte principal e do tratador de exceção estão contidos na mesma classe. Esta proposta de modificação ainda permite uma otimização na utilização da memória do sistema tempo-real embarcado pois não é necessário memória para armazenar um segundo objeto responsável apenas pela execução do método de tratamento de exceção.

### 5.1.3 Classes para Modelagem de Escalonadores de Tarefas

Este grupo de classes é responsável pelo escalonamento dos objetos ativos do sistema tempo-real embarcado, ou seja, o conjunto de classes que representam a política de escalonamento do sistema. O diagrama de classes detalhado para as classes dos escalonadores pode ser visto na Figura 5.5.

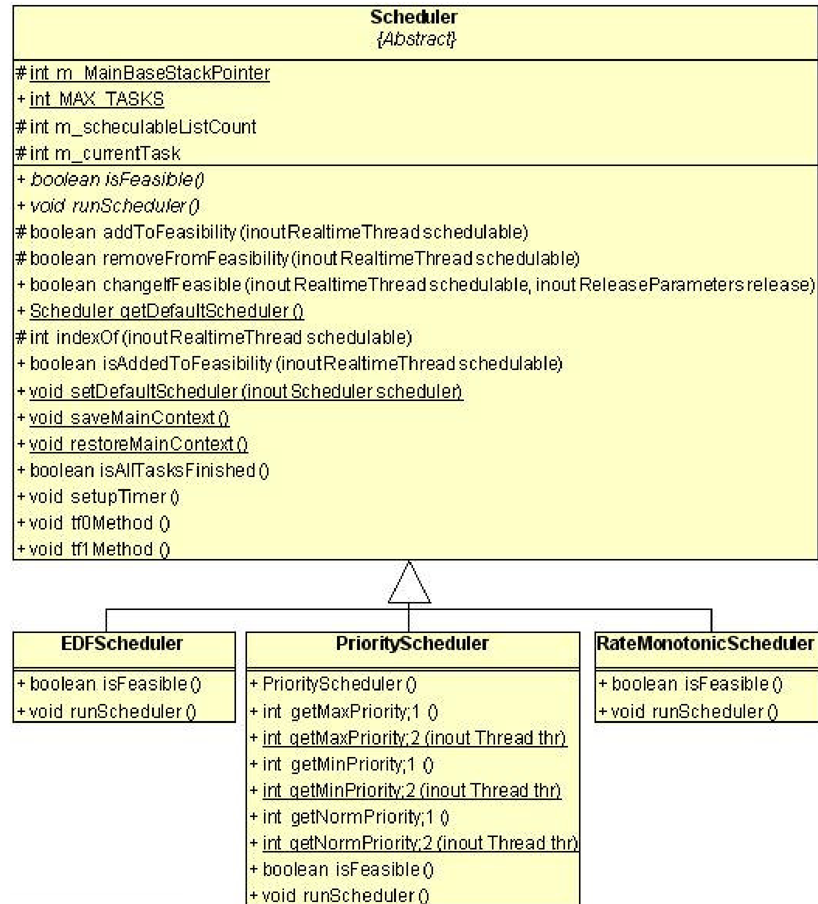


Figura 5.5: Diagrama de classes detalhado para a modelagem dos escalonadores de tarefas

A classe base para os escalonadores presentes no *framework* é a classe *Scheduler*. O escalonador proposto neste *framework* pode suportar no máximo 8 *threads* rodando simultaneamente. Esta restrição foi definida com base na quantidade de memória RAM disponível para o FemtoJava e com base no crescimento da pilha (ROSA JUNIOR., 2004). Basicamente os atributos desta classe são relativos ao conjunto de tarefas a serem escalonados: o atributo *m_schedulableListCount* é o total de *threads* que foram adicionadas na lista de escalonamento; *m_currentTask* indica o índice da *thread* que está sendo executada no momento; m_MAX_TASKS é uma constante indicando o máximo de *threads* que podem ser adicionadas na lista. Apenas o atributo m_MainBaseStackPointer não tem relação com a lista das *threads*, pois este atributo representa o topo da pilha da *thread* principal do sistema, ou seja, o contexto do laço infinito que representa o sistema operacional do sistema tempo-real embarcado. A descrição dos métodos da classe *Scheduler* pode ser vista na Tabela 5.3.

Tabela 5.3: Métodos da classe *Scheduler*

| **Método** | **Descrição** |
|---|---|
| *addToFeasibility()* | Adiciona um objeto ativo na lista de *threads* para serem escalonadas. |
| *removeToFeasibility()* | Remove um objeto ativo na lista de *threads* a serem escalonadas. |
| *isAddedToFeasibility()* | Retorna um valor booleano indicando se uma *thread* está na lista de *threads* a escalonar ou não. |
| *getDefaultScheduler()* | Retorna o objeto contendo a política de escalonamento atual do sistema. |

| | |
|---|---|
| *setDefaultScheduler()* | Atribui um objeto que define a política de escalonamento do sistema. |
| *saveMainContext()* | Salva o topo da pilha para a *thread* principal do sistema. |
| *restoreMainContext()* | Restaura o topo da pilha da *thread* principal do sistema. |
| *isAllTasksFinished()* | Retorna um valor booleano indicando se todas as *threads* estão terminadas ou não. |
| *setupTimer()* | Configura o temporizador para a ativação do escalonador. |
| *tf0Method()* | Compatibilidade com versão anterior do Sashimi. |
| *tf1Method()* | Compatibilidade com versão anterior do Sashimi. |

As políticas de escalonamento disponíveis na proposta inicial deste *framework* estão implementadas nas classes *EDFScheduler*, *RateMonotonicScheduler* e *PriorityScheduler*, representando respectivamente os algoritmos *Earliest Deadline First* (EDF) (LIU; LAYAND, 1973), *Rate Monotonic Scheduler* (RMS) (LIU; LAYAND, 1973) e o escalonador de prioridades fixas (BURNS; WELLING, 1996). Todas estas classes que estendem a classe *Scheduler* devem, obrigatoriamente, sobrescrever os métodos abstratos *isFeasible()* e *runScheduler()* que implementam, respectivamente, o teste de escalonabilidade e o algoritmo de escalonamento.

### 5.1.4 Classes para Modelagem de Temporizadores

Este conjunto de classes define os mecanismos de temporização (*timers*) que podem ser utilizados na implementação de um sistema tempo-real embarcado. Na Figura 5.6 podem ser observadas as três classes que compõem o mecanismo dos *timers*. A classe base para este mecanismo de temporização é a classe *Timer*, que propicia a infra-estrutura básica para os temporizadores.

A versão do processador FemtoJava utilizada durante o desenvolvimento deste trabalho suporta apenas dois temporizadores, sendo que ambos podem estar ativos no mesmo instante. Como um dos temporizadores é utilizado pelo escalonador de processos, apenas um temporizador pode ser usado pelos objetos *Timer*.

O tempo de expiração do temporizador (*timeout*) é representado por um objeto *HighResolutionTime*, ou seja, o *timeout* pode ser um tempo absoluto ou um tempo relativo. Cada objeto *Timer* possui um número identificador (*m_ID*) , sendo assim, o atributo de classe *m_activeTimer* representa qual dos temporizadores do sistema está ativo em dado instante. O estado do temporizador pode ser verificado nos atributos *m_isEnable* e *m_isRunning* que indicam, respectivamente, se o temporizador está habilitado e se ele está ativo. A descrição da funcionalidade dos métodos da classe *Timer* pode ser vista na Tabela 5.4.

Existem dois tipos de temporizadores, sendo que um tipo sinaliza o *timeout* apenas uma única vez, representado na classe *OneShotTimer*, e o outro tipo de temporizador que sinaliza o *timeout* continuamente, representado na classe *PeriodicTimer*. A semântica de temporizadores *PeriodicTimer* é da seguinte forma: após o primeiro *timeout*, o temporizador é reconfigurado com o mesmo intervalo de tempo de modo que o tempo comece a ser medido novamente até que ocorra o novo *timeout* e, após esta reconfiguração, o código associado ao temporizador é executado. O sinal de *timeout* ocorrerá ciclicamente até que o método *stop()* seja executado.

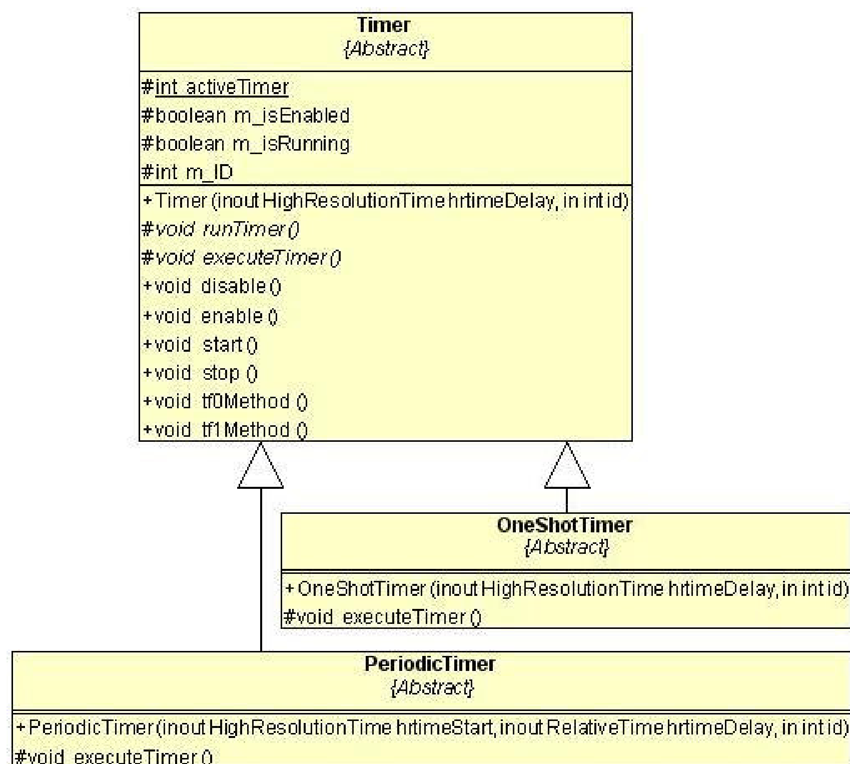


Figura 5.6: Diagrama de classes detalhado para a modelagem de temporizadores

Basicamente, para criar um temporizador utilizando o *framework* deve-se escolher qual o tipo do temporizador (*timeout* único ou contínuo) identificando a sua respectiva classe de acesso. Uma vez definido qual será a superclasse, basta apenas sobrescrever o método *runTimer()*, escrevendo o código referente ao tratador do *timeout*. Contudo como o processador FemtoJava possui apenas dois temporizadores concorrentes, as aplicações só podem ter, no máximo, dois temporizadores ativos. Se a aplicação que está sendo desenvolvida fizer uso do mecanismo de escalonamento disponibilizado por este *framework*, então a aplicação poderá utilizar apenas um temporizador. Isso se deve ao fato do mecanismo de escalonamento utilizar um dos temporizadores para que o mesmo seja avisado quando uma tarefa deve ser interrompida por outra ou quando o *deadline* desta tarefa foi perdido.

Para minimizar esta limitação poderiam ser adicionados ao FemtoJava mais temporizadores. Todavia, com a estrutura atual dos temporizadores, a adição de novos temporizadores implicaria no aumento grande de área ocupada pelo FemtoJava no FPGA. Isso se deve ao fato de que cada temporizador possui seu próprio contador de tempo. Com a inclusão de um RTC no Femtojava, o mecanismo dos temporizadores poderia ser alterado de forma a usar esta referência de tempo para todos os temporizadores, diminuindo a área gasta na adição de novos temporizadores.

Tabela 5.4: Descrição dos métodos da classe *Timer*

| Método | Descrição |
|---|---|
| *enable()* | Habilita o temporizador para a execução. |
| *disable()* | Desabilita a execução do temporizador. |
| *start()* | Inicia a contagem de tempo para o *timeout*. |
| *stop()* | Para a contagem de tempo para o *timeout*. |
| *tf0Method()* | Compatibilidade com versão anterior do Sashimi. |
| *tf1Method()* | Compatibilidade com versão anterior do Sashimi. |

## 5.2 Adaptações na ferramenta SASHIMI

O SASHIMI é a ferramenta CAD utilizada para realizar a síntese do processador FemtoJava e da aplicação que irá executar neste processador. Na versão disponível para utilização no início dos trabalhos desta dissertação, não era possível fazer a síntese do *framework* proposto pelo fato da mesma não suportar orientação a objetos. Assim sendo esta ferramenta necessitou sofrer modificações no que diz respeito à análise dos *bytecodes* Java passados como entrada para a ferramenta. Além disso, esta versão também não suportava a síntese de objetos, um ítem que limitava o reuso dos componentes previamente desenvolvidos, pois havia a necessidade de adaptações do código fonte dos algoritmos implementados em Java já validados em projetos anteriores e que existiam na biblioteca para serem reutilizados. A Figura **5.7** apresenta as modificações realizadas na ferramenta SASHIMI. À esquerda pode ser observado o fluxo de projeto original da ferramenta, enquanto à direita é apresentado o fluxo modificado, onde as partes destacadas indicam as modificações propostas neste trabalho.

Na realidade esta limitação advinha da não possibilidade de utilização da orientação a objetos disponível na linguagem Java, pois a análise dos *bytecodes* e a síntese do código só estavam habilitadas a identificar atributos e métodos estáticos de classe. A possibilidade de utilização da orientação a objetos facilita muito o reuso de classes desenvolvidas e testadas anteriormente através do mecanismo de herança de classes, onde todas as características são passadas adiante na hierarquia das classes do sistema tempo-real embarcado. Assim, por exemplo, se um método precisar incluir um simples teste antes da sua execução basta incluir o teste no código da subclasse e chamar o método na superclasse, o que é bem mais simples do que tentar compreender a implementação do algoritmo para saber onde incluir o teste. Esta uma tarefa que pode ser muito complexa se o desenvolvedor que está adaptando a classe não for o mesmo que criou a superclasse.

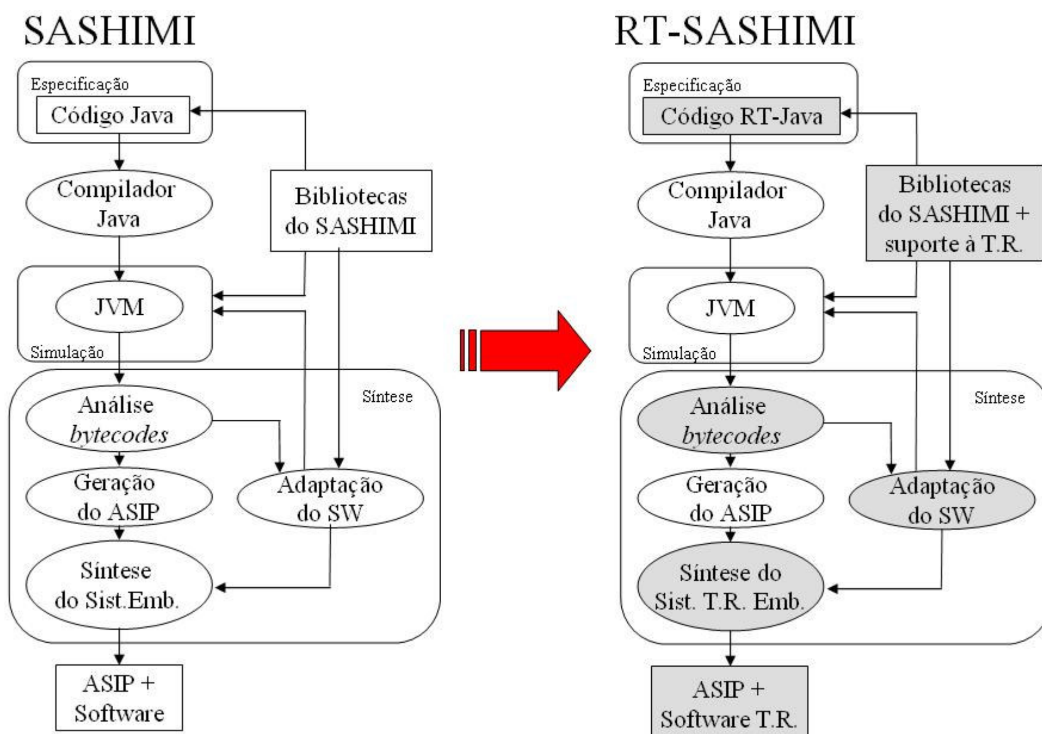


Figura 5.7: Fluxo de projeto da ferramenta SASHIMI

### 5.2.1 Adaptação da Análise de Bytecodes

Após a análise do código fonte da ferramenta SASHIMI constatou-se que a estrutura que representava os *bytecodes* utilizados como entrada da ferramenta atendia apenas alguns requisitos para a geração do software para o sistema tempo-real embarcado. Sendo assim, a primeira adaptação da ferramenta SASHIMI foi a criação de uma estrutura que possibilitasse a representação das classes encontradas nos *bytecodes* Java, assim como permitisse identificar e armazenar quais objetos foram criados na aplicação. O diagrama de classes contendo a nova estrutura criada para o SASHIMI é apresentado na Figura 5.8.

A classe *RTClassInfo* representa uma classe encontrada nos *bytecodes* Java, sendo que esta possui uma série de atributos como seu nome (*m_ClassName*), se esta classe estende outra classe, ou seja, se possui uma classe pai (*m_SuperClass*), a lista de atributos e métodos (*m_Fields* e *m_Methods*), o identificador único da classe (*m_ClassID*), uma lista de objetos (*m_InstanceObjects* e *m_StaticObjects*) e a lista de subclasses (*m_ChildClasses*). Os métodos da classe *RTClassInfo* são basicamente para a manipulação de atributos, métodos, objetos e subclasses com algumas exceções como os métodos *adaptPolymorphicMethod()*, *allocateObject()*, *recalculateFieldsOffset()* e *getSizeOfObject()* que são métodos utilizados na síntese do código da classe.

A classe *RTFieldInfo* por sua vez representa um atributo de uma classe sendo que esta representa características como o nome do atributo (*m_FieldName*), a sua assinatura (*m_FieldSignature*), o seu deslocamento com base no endereço inicial da memória onde a classe foi alocada (*m_Offset*), o seu índice na *ConstantPool* do *bytecode* da classe (*m_CnstPoolIdx*), se o atributo é acessado no código da classe ou não (*m_isUsed*) ou se é um atributo estático ou de instância (*m_isStatic*). Os métodos da classe *RTField* permitem o acesso aos atributos da classe.

A classe *RTMethodInfo* representa os métodos de uma classe. As informações dos métodos representados por esta classe são: nome (*m_MethodName*); assinatura (*m_Signature*); se o método é abstrato (*m_isAbstract*); se o método é estático (*m_isStatic*); o endereço do método na memória ROM (*m_Address*); uma lista contendo todos os métodos das subclasses que sobrescrevem o método (*m_OverrideMethods*); o código binário (*m_Code*); a quantidade de variáveis locais e parâmetros (*m_LocalVarCount* e *m_ParamsCount*); e um flag indicando se o método sobrescreve outro método (*m_isOverrideMethod*). Assim como na classe *RTClassInfo*, a maioria dos métodos da classe *RTMethodInfo* servem para acessar as informações sobre características de um método, contudo os métodos *adaptPolymorphicMethod()*, *addPolymorphisCtrlParam()* e *fixLocalVariableReferences()* são utilizados durante síntese do código da aplicação. Estes métodos modificam o código binário do método que está sendo representado pelo objeto *RTMethodInfo*, inserindo o código necessário para o funcionamento do mecanismo de polimorfismo.

Figura 5.8: Diagrama de classes da nova estrutura de representação das classes no SASHIMI

Por fim a classe *RTObject* foi criada para representar um objeto criado pela aplicação, sendo que as informações que ela pode prover dizem respeito à classe do objeto (*m_ObjectClass*), ao local onde este objeto foi criado (*m_RefClass*, *m_RefMethod* e *m_RefLine*), ao endereço de memória onde ele foi alocado (*m_Address*), se o objeto foi usado no código ou não (*m_isObjectUsed*) e à instrução *new* que o criou (*m_instruction*). Este último atributo (*m_instruction*) foi incluído apenas para fins de otimização do processo de síntese do código da aplicação, pois permite uma facil localização da instrução que deve ser substituída pelo endereço de memória que representa o início do objeto alocado estaticamente. Os métodos dessa classe seguem o mesmo principio dos métodos das classe *RTClassInfo* e *RTMethodInfo*, ou seja, a maioria serve apenas para acessar as características representadas pela classes, com exceção dos métodos *allocateObject()* e *replaceNewOpcode()*.

Um ponto muito importante a ser ressaltado sobre a análise de *bytecodes* diz respeito à identificação do instante de criação dos objetos. O escopo deste trabalho foi a criação

de um *framework* para a expressão dos requisitos tempo-real de um sistema embarcado e o mapeamento de modelos orientados a objeto que utilizam a RT-UML nos elementos do *framework*, sendo assim o suporte da orientação a objetos na ferramenta SASHIMI foi desenvolvido apenas permitindo a criação de objetos estaticamente, em outras palavras, a análise dos *bytecodes* consegue identificar e sintetizar objetos criados em atributos estáticos das classes.

Sendo assim, a solução proposta para a alocação de objetos não utiliza tabelas para a representação da hierarquia de classes, nem para a representação dos métodos que são polimórficos[5] como é proposto na especificação da JVM (LINDHOLM; YELLIN, 1997). A escolha desta abordagem permite a economia de memória RAM na representação dos objetos, aumentando o código binário[6] e eliminando a indeterminação temporal introduzida pelo uso de tabelas para percorrer a hierarquia das classes. Maiores detalhes da síntese dos objetos serão dados na subseção a seguir.

### 5.2.2 Adaptação da Síntese do Processador e da Aplicação

Como dito anteriormente, a ferramenta SASHIMI suportava apenas a síntese de atributos e métodos estáticos de classe. Com as modificações feitas no SASHIMI agora é possível fazer a síntese de objetos. A Figura 5.9 mostra o código Java parcial de uma classe onde se mostra a criação de um objeto de uma classe que estende a classe *RealtimeThread*, e representa uma *thread* na aplicação do sistema tempo-real embarcado.

```
public class MultiTaskSystem {

   public static Task1 t1 = new Task1(null, pp_300_ms);
   ... // other class elements

}
```

Figura 5.9: Trecho de código fonte em Java para possibilitar a síntese de objetos através do SASHIMI

Após a análise do *bytecode* Java da classe apresentada parcialmente na Figura 5.9, que foi gerado pelo compilador Java padrão[7], a ferramenta SASHIMI gera o arquivo RAM.MIF para o sistema tempo-real embarcado onde o objeto é alocado estaticamente. O trecho respectivo mostrando o objeto alocado pode ser verificado na Figura 5.10. Na posição 63h da memória encontra-se o identificador único da classe, que é usado pelo mecanismo de polimorfismo para saber qual método deve ser executado além de ser utilizado nas instruções de checagem de tipo de classe como a instrução *instanceof*. Os atributos herdados da classe *RealtimeThread* estão localizados nos endereços 64h a 6Ch. Por fim nos endereços 6Dh à 6Fh estão os atributos específicos da classe *Task1*. Com isso, pode-se perceber que a organização em memória de um objeto alocado estaticamente inicia com o identificador único da classe do objeto, em seguida vem os atributos de instância herdados das classes ancestrais sempre iniciando pela classe mais ao topo da hierarquia, seguido dos atributos de instância da própria classe.

---

[5] São método que sobrescrevem métodos das classes ancestrais

[6] Pois o algoritmo que implementa o mecanismo de polimorfismo dos métodos é inserido no código da aplicação durante a síntese da aplicação tempo-real embarcada

[7] Compilador *javac* que é disponibilizado com o JDK disponível para ser copiado no site da *Sun Microsystems*

```
...
63   :   0b; -- ----------------------------> ID for Task1
64   :   00; -- m_releaseParameters[Lsashimi/realtime/ReleaseParameters]
65   :   00; -- m_schedulingParameters[Lsashimi/realtime/SchedulingParameters]
66   :   00; -- m_isStarted [Z.1]
67   :   00; -- m_isFinalized [Z.1]
68   :   00; -- m_isRunning [Z.1]
69   :   00; -- m_isBlocked [Z.1]
6a   :   00; -- m_BaseStackPointer [I.1]
6b   :   00; -- m_StackPointer [I.1]
6c   :   00; -- m_ResumeTime [Lsaito/sashimi/realtime/AbsoluteTime;.1]
6d   :   00; -- m_aux [I.1]
6e   :   00; -- m_Loop [I.1]
6f   :   00; -- m_SchedCount [I.1]
...
```

Figura 5.10: Trecho de código do arquivo RAM.MIF indicando a alocação de um objeto

Como a alocação dos objetos é feita estaticamente, o SASHIMI substitui a instrução *new* encontrada nos métodos de inicialização das classes (*<clinit>*, encontrado no *bytecode* de cada classe) por uma instrução que empilha o endereço de memória onde o objeto foi alocado. No caso do exemplo citado nas figuras 5.8 e 5.9 a instrução *new* é substituída pelo a instrução "*sipush 63*" que coloca o valor 63h no topo da pilha.

Para possibilitar o acesso aos atributos de instância incluiu-se suporte ao uso das instruções que fazem acesso aos atributos de instância de um objeto: *getfield* e *putfield*. Estas instruções, segundo a especificação da JVM (LINDHOLM; YELLIN, 1997), fazem acesso às tabelas de representação de hierarquia o que introduz indeterminismo temporal, algo inaceitável em aplicações tempo-real. Como a abordagem utilizada não faz uso deste mecanismo, a semântica destas instruções foi alterada: ao invés destas instruções conterem o índice para as tabelas, esse inteiro foi alterado para representar o deslocamento do atributo (*offset*) com relação ao endereço inicial do objeto na memória. Por exemplo, para acessar o campo *m_Loop* o índice da tabela é substituído pelo valor 11, quando uma instrução *getfield* for executada, o endereço 63h estará no topo da pilha e será incrementado com o valor 11, resultando no endereço 6Eh, então o conteúdo desde endereço será colocado no topo da pilha. A instrução *putfield* executa o cálculo de maneira idêntica, porém ao invés de ler a posição de memória resultante é feita a gravação de um valor nesta posição. Novamente, esta abordagem garante o determinismo temporal pois o tempo pode ser medido através da quantidade de ciclos necessários para a execução das instruções *getfield* e *putfield*.

A síntese do mecanismo de polimorfismo é implementada nos *bytecodes* que serão sintetizados, ou seja, os *bytecodes* serão alterados de forma automática a fim de permitir que o WCET de um método que sobrescreve outro possa ser determinado com base no tempo total de execução das instruções incluídas nos *bytecodes*. Na Figura 5.11 é apresentado o código binário sintetizado que implementa o polimorfismo para o método *mainTask()* da classe *RealtimeThread*. Como este método é abstrato na definição da classe no *framework*, ele necessita ser implementado nas classes que representam as *threads* do sistema. Nesta aplicação exemplo existem duas *threads* no sistema tempo-real embarcado, sendo assim o método *mainTask()* foi sobrescrito por duas classes diferentes e deve estar apto a executar o código de ambos os métodos quando este for chamado.

Assim que a ferramenta SASHIMI identifica quais métodos sobrescrevem outros métodos, durante a fase de análise de *bytecodes*, o mecanismo do polimorfismo começa a ser incluído nos *bytecodes*. O código inserido nos *bytecodes* da aplicação segue os seguintes passos:

- Primeiramente, nos métodos que são sobrescritos é feita a inclusão de um parâmetro booleano, no final da lista de parâmetros, indicando que este método é sobrescrito por métodos das subclasses da classe corrente;
- Em seguida, um valor booleano é empilhado imediatamente antes de todas as chamadas ao método (*invokevirtual* ou *invokespecial*);
- O passo seguinte é incluir o código de suporte aos outros métodos dentro do código do método sobrescrito. As instruções que são incluídas podem ser vistas nos trechos 9Ch-ABh e ACh-BBh da Figura 5.11.

```
...
95   :   00; -- --------- saito/sashimi/realtime/RealtimeThread.mainTask.()V.0
96   :   02; --
     -- Verif param que controla execucao do mecanismo de polimorfismo
97   :   15; -- iload
98   :   01; --
99   :   99; -- ifeq
9a   :   00; --
9b   :   21; --
     -- Compara identificador da primeira classe filha
9c   :   2a; -- aload_0
9d   :   b4; -- getfield
9e   :   00; --
9f   :   00; --
a0   :   10; -- bipush
a1   :   0b; --
a2   :   a0; -- if_icmpne
a3   :   00; --
a4   :   08; --
     -- Se o objeto for da classe, executa o método sobreescrevente
a5   :   2a; -- aload_0
a6   :   b6; -- invokevirtual
a7   :   08; --
a8   :   98; --
a9   :   a7; -- goto
aa   :   00; --
ab   :   11; --
     -- Compara identificador da proxima classe filha
ac   :   2a; -- aload_0
ad   :   b4; -- getfield
ae   :   00; --
af   :   00; --
b0   :   10; -- bipush
b1   :   0c; --
b2   :   a0; -- if_icmpne
b3   :   00; --
b4   :   08; --
     -- Se o objeto for da proxima classe, executa o método sobreescrevente
b5   :   2a; -- aload_0
b6   :   b6; -- invokevirtual
b7   :   09; --
b8   :   c4; --
b9   :   a7; -- goto
ba   :   00; --
bb   :   01; --
bc   :   b1; -- return
...
```

Figura 5.11: Trecho de código binário para o método abstrato mainTask() da classe RealtimeThread

As novas instruções que foram incluídas têm a seguinte semântica: primeiro é feito um teste para verificar se o parâmetro booleano (que indica se o método foi sobrescrito) tem o valor *true*; caso verdadeiro, o identificador único da classe do objeto (*m_ClassID*) é comparado com o identificador de uma das subclasses que implementa o método que sobrescreve o método da superclasse; se os identificadores forem iguais então o método

é chamado, caso contrário essa seqüência de instruções é executada novamente para a próxima subclasse. Dessa forma cada método sobrescrito recebe as novas instruções (e.g. trecho 9Ch a ABh) para cada subclasse que o sobrescreve. Essa abordagem permite que o WCET possa ser calculado com base na quantidade de ciclos necessário para a execução das instruções desde o início do código binário até a instrução *invokevirtual* do método.

A forma de especificação do ambiente de execução para múltiplas *threads*, original do SASHIMI, foi modificado devido à sua complexidade, baixo nível de especificação e facilidade de gerar código de aplicação com *bugs*. Originalmente para especificar a concorrência do sistema o desenvolvedor deveria criar a pilha do processo de forma manual, *frame* a *frame* de cada método, conforme ilustrado na Figura 5.12. Como pode ser observado, os endereços da pilha deveriam ser informados manualmente, assim como os endereços da rotina do escalonador e da *thread*. Como estes endereços de rotinas não são conhecidos *a priori*, deve-se primeiro sintetizar a aplicação para então verificar o endereço das rotinas no arquivo ROM.MIF, e então modificar o código fonte da inicialização para poder sintetizar a versão final da aplicação. Esse procedimento deve ser feito para cada *thread* concorrente no sistema.

```
public static void initTasks(){

  /* Begin: incializa pilhas dos processos
   **/
  // processo 1 (em ordem decrescente da pilha)
  FemtoJavaSO.initProcess(                     0,  -((0xE67F ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess(                     0,  -((0xE67E ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess(                     3,  -((0xE67D ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess(-((0xE67F ^ 0xFFFF) + 1), -((0xE67C ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess(-((0xE67F ^ 0xFFFF) + 1), -((0xE67B ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess(-((0xE67F ^ 0xFFFF) + 1), -((0xE67A ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess(-((0xE67F ^ 0xFFFF) + 1), -((0xE679 ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess( 0x4FC, -((0xE677 ^ 0xFFFF) + 1)); // ESCALONADOR
  FemtoJavaSO.initProcess(-((0xE67D ^ 0xFFFF) + 1), -((0xE676 ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess(-((0xE67D ^ 0xFFFF) + 1), -((0xE675 ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess(-((0xE67D ^ 0xFFFF) + 1), -((0xE674 ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess(-((0xE67D ^ 0xFFFF) + 1), -((0xE673 ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess( 0x6D5, -((0xE672 ^ 0xFFFF) + 1)); // THREAD 1
  FemtoJavaSO.initProcess(-((0xE677 ^ 0xFFFF) + 1), -((0xE671 ^ 0xFFFF) + 1));
  FemtoJavaSO.initProcess(-((0xE678 ^ 0xFFFF) + 1), -((0xE670 ^ 0xFFFF) + 1));

  stack[1] = -((0xE66F ^ 0xFFFF) + 1);

  ... // continues
```

Figura 5.12: Especificação das *threads* concorrentes na versão original do SASHIMI

Com a adaptação da ferramenta SASHIMI, esse código de inicialização da pilha para as *threads* é gerado automaticamente no método *initializeStack()* das classes que estendem a classe *RealtimeThread*. Os *frames* empilhados na pilha da tarefa seguem o esquema apresentado na Figura 5.13, sendo que a ordem dos campos do *frame* do método segue o padrão definido para os *frames* de métodos no FemtoJava (ITO, 2000). Inicialmente é empilhado o *frame* da execução do escalonador, em seguida o *frame* do método *mainTask()* do objeto que estende a *RealtimeThread* e por fim o *frame* que será desempilhado quando o escalonador colocar a *thread* para ser executada pela primeira vez. A quantidade de *threads* que podem existir simultaneamente em um sistema tempo-real embarcado que utiliza o *framework* proposto neste trabalho foi limitada para

8 *threads*, devido ao valor estipulado para o crescimento da pilha[8] da *thread*. Este valor pode ser alterado conforme a quantidade de memória disponível no FemtoJava, de modo que o crescimento da pilha de uma *thread* não sobrescreva os dados do sistema, nem a pilha de execução de outras *threads*.

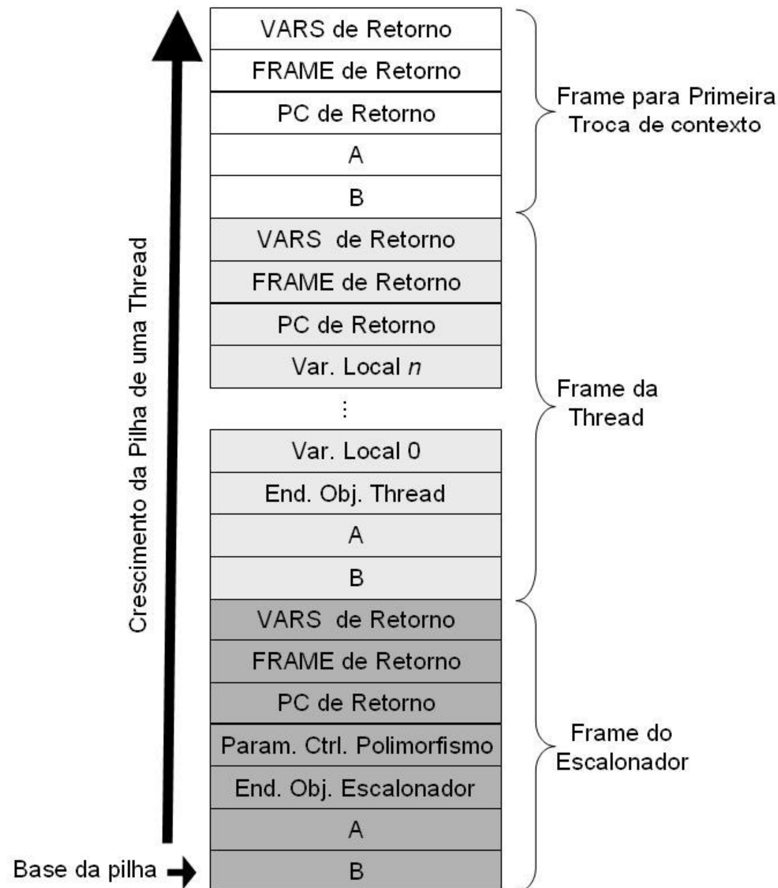

Figura 5.13: Esquema do empilhamento dos *frames* da pilha de cada *thread*

Outro ponto modificado na síntese da aplicação é a inclusão do código de inicialização das classes (<*clinit*>) após a área de código dos tratadores de interrupção na memória de instruções do processador FemtoJava. É justamente neste código que é feita a substituição das instruções *new* pelo empilhamento do endereço onde os objetos foram alocados estaticamente, em tempo de síntese.

Finalizando o processo de síntese da aplicação, o código orientado a objetos da aplicação é gerado no arquivo ROM.MIF. Então os arquivos VHDL gerados pelo SASHIMI servem como entrada para uma ferramenta de síntese de FPGAs a qual sintetiza o processador FemtoJava customizado para a aplicação tempo-real. Cabe aqui ressaltar que apesar do código da aplicação ter sofrido algumas alterações no processo de síntese, este código ainda é otimizado pois inclui somente o código dos métodos utilizados na aplicação do sistema tempo-real embarcado. Além de otimizar a utilização de memória necessária para armazenar os objetos da aplicação.

[8] A pilha cresce conforme a quantidade de chamadas de métodos aninhados, ou seja, quanto mais métodos forem chamados, uns dentro dos outros, maior será o crescimento da pilha

## 5.3 Adaptações no FemtoJava

Além das alterações na ferramenta SASHIMI, foi necessária a adaptação do processador FemtoJava de modo a incluir o suporte de execução para as novas instruções suportadas na ferramenta SASHIMI e também possibilitar que o sistema tempo-real embarcado possua uma referência global de tempo.

O primeiro ítem a ser incluído no processador FemtoJava foi um relógio de tempo-real (*Real-Time Clock* ou RTC). O RTC foi implementado como sendo um módulo dentro do SoC do FemtoJava e segue o modelo mostrado na Figura 5.14. O RTC tem o formato da data/tempo é composto por três inteiros de 32 bits que representam os dias, milisegundos e nanosegundo decorridos desde a data 01/01/1970 00:00:00.000[9]. O acesso aos registradores que compõem o RTC é feito através de endereços na memória RAM, assim como é feito com os outros registradores disponíveis no FemtoJava. Os endereços utilizados pelo RTC são: o endereço 10h que é associado ao registrador que representa os dia; o endereço 11h é associado ao registrador dos milisegundos; e o 12h associado com o registrador dos nanosegundos. Como os sinais de entrada e saída do RTC estão ligados no árbitro do barramento de dados, os valores nos registradores do RTC podem ser lidos e escritos pelo programador da aplicação, bastando apenas a utilização da classe *Clock* disponível no *framework*. O código VHDL correspondente ao RTC pode ser visto na **Figura 5.15**.

Figura 5.14: Real-Time Clock

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
  use work.all;

entity fj_rtc is
  generic(DAYS_WIDTH: INTEGER := 32;
          MILLIS_WIDTH: INTEGER := 27;
          NANOS_WIDTH: INTEGER := 20);
  port(-- input
       clock     : in std_logic; -- System clock
       reset     : in std_logic; -- System clock
       days_rw   : in std_logic; -- Days write signal
       milis_rw  : in std_logic; -- Miliseconds write signal
       nanos_rw  : in std_logic; -- Nanoseconds write signal
       -- Data (days, milis or nanos) signal
       data_in   : in std_logic_vector(31 downto 0);
       -- output
       -- Days past from 01/01/1970
```

[9] Esta data de referência é a mesma utilizada na classe java.util.Date da especificação J2SE (SUN, 1995).

```
       days_out  : out std_logic_vector(DAYS_WIDTH - 1 downto 0);
       -- Milliseconds past from 00:00:00
       milis_out: out std_logic_vector(MILLIS_WIDTH - 1 downto 0);
       -- Nanoseconds past from begin of millisecond
       nanos_out : out std_logic_vector(NANOS_WIDTH - 1 downto 0));
end fj_rtc;

architecture behaviour of fj_rtc is
  constant MILLIS_PER_DAY: INTEGER := 86400000;
  constant NANOS_PER_MILLI: INTEGER := 1000000;
  constant MICROS_PER_MILLI: INTEGER := 1000;

  signal var_days_out  : unsigned(DAYS_WIDTH - 1 downto 0);
  signal var_milis_out: unsigned(MILLIS_WIDTH - 1 downto 0);
  signal var_nanos_out : unsigned(NANOS_WIDTH - 1 downto 0);

begin
  process(clock, reset)
  variable var_clock_div  : Integer;
  begin
    if (reset = '1') then
      var_clock_div := 5;
    else
      if (clock'EVENT) AND (clock = '1') then
        if (days_rw = '1') then
          var_days_out <= unsigned(data_in);
        else
          if (milis_rw = '1') then
            var_milis_out<=unsigned(data_in(MILLIS_WIDTH-1 downto 0));
          else
            if (nanos_rw = '1') then
             var_nanos_out<=unsigned(data_in(NANOS_WIDTH-1 downto 0));
            end if; -- (nanos_rw = '1')
          end if; -- (milis_rw = '1')
        end if; -- (days_rw = '1')

        var_clock_div := var_clock_div - 1;
        if (var_clock_div <= 0) then
          var_nanos_out <= var_nanos_out + 1;
          if (var_nanos_out = MICROS_PER_MILLI) then
            var_nanos_out <= B"00000000000000000000";
            var_milis_out <= var_milis_out + 1;
            if (var_milis_out = MILLIS_PER_DAY) then
              var_milis_out <= B"000000000000000000000000000";
              var_days_out <= var_days_out + 1;
            end if; -- (var_millis_out = MILLIS_PER_DAY)
          end if; -- (var_nanos_out = NANOS_PER_MILLI)
        end if; -- (var_clock_div <= 0)
      end if; -- (clock'EVENT AND clock = '1')
    end if; -- (reset = '1')
  end process;

  days_out <= std_logic_vector(var_days_out);
  milis_out <= std_logic_vector(var_milis_out);
  nanos_out <= std_logic_vector(var_nanos_out);
end behaviour;
```

Figura 5.15: Código VHDL do Real-Time Clock

Para suportar a execução dos objetos as seguintes instruções foram incluídas na máquina de controle do processador FemtoJava:

- **Manipulação de variáveis do tipo referência de objeto**: aconst_null, aload, aload_0, aload_1, aload_2, aload_3, aaload, astore, astore_0, astore_1, astore_2, astore_3, aastore e areturn;
- **Desvio condicional:** if_acmpne, ifnull e ifnonnull;
- **Acesso aos atributos de instância das classes:** getfield e putfield;
- **Chamada de métodos de instância:** invokevirtual e invokespecial;
- **Teste e *typecast* de classes:** instanceof e checkcast;

As instruções para manipulação de variáveis que contêm a referência dos objetos tem a mesma semântica das instruções para leitura e escrita de variáveis de números inteiros, pois as referências de objetos nada mais são que números inteiros que representam o endereço inicial dos objetos na memória RAM. Os desvios condicionais incluídos na máquina de controle do FemtoJava também seguem a semântica dos desvios condicionais que testam números inteiros.

Já as instruções que fazem acesso aos atributos de instância dos objetos tiveram que ter sua seqüência de palavras de controle criada, pois não havia instrução semelhante disponível no FemtoJava. Estas instruções possuem um número inteiro associado, que indica o deslocamento de memória para que o endereço onde se localiza o valor do atributo possa ser calculado, com base no endereço inicial do objeto.

A seqüência de palavras de controle para as instruções referentes à chamada de métodos de instância foram baseadas nas instruções para chamada de métodos estáticos, ou seja, as chamadas de métodos estáticos e de instância possuem a mesma seqüência de palavras de controle.

Por fim as instruções que trabalham com a identificação do tipo de classe de um objeto também foram implementadas, pois não havia instrução semelhante disponível no FemtoJava. Durante a implementação do *framework*, houve necessidade da implementação das instruções *instanceof* e *checkcast*. A instrução *intanceof* que permite testar se um objeto é uma instância de uma classes especificada tem uma limitação com relação à semântica que foi especificada na JVM padrão. Esta instrução suporta apenas o teste direto de uma classe específica, em outras palavras, a instrução *intanceof* apenas permite testar se um objeto é da classe “X”, ao invés de verificar em toda a estrutura hierárquica da classe. Assim, a instrução *instanceof* apenas verifica se o objeto é do tipo da classe “X” e não se é do tipo de alguma classe que descende da classe “X”.

Esta limitação foi imposta pela forma que o mecanismo de herança/polimorfismo foi implementado. O nível de complexidade necessário para implementar a semântica para esta instrução, conforme a especificação da JVM (LINDHOLM; YELLIN, 1997), foi considerado alto. Assim, como o foco principal deste trabalho não era implementar todo o mecanismo de orientação a objetos disponível no Java, optou-se por esta implementação da instrução *instanceof*. Resumindo, o programador do sistema tempo-real embarcado deve estar ciente que, quando utilizar a operação “*objeto instanceof ClasseX”* o resultado retornará verdadeiro apenas se o objeto realmente for da “ClasseX”.

Maiores detalhes sobre a implementação e a seqüência das palavras de controle de cada instrução podem ser encontrados no anexo A.

## 5.4 Proposta de Mapeamento

Este capítulo trata do mapeamento do modelo orientado a objetos em alto-nível, utilizando a RT-UML, para o código fonte do software de um sistema tempo-real embarcado. O objetivo deste mapeamento é permitir que o código fonte possa ser gerado por ferramentas de geração automática de código fonte.

O restante deste capítulo abordará os conceitos de tempo-real que podem ser representados por elementos UML utilizando os estereótipos disponíveis no perfil de escalonabilidade, performance e tempo, bem como o mapeamento destes elementos nas construções disponíveis no *framework* proposto neste trabalho. Por fim será analisada a possibilidade de geração automática de código a partir do repositório da ferramenta *Artisan Real-Time Studio*.

### 5.4.1 Visão geral da proposta

A utilização de uma linguagem de modelagem facilita a especificação dos requisitos e funcionalidade de sistemas tempo-real embarcados, além de auxiliar na compreensão do problema, servir como documentação de projeto e unificar a comunicação entre os integrantes da equipe de desenvolvimento podendo até mesmo servir como meio de comunicação com os usuários finais do sistema. Nesse ambiente a utilização de linguagens de modelagem largamente aceitas pelas comunidades de projetistas é de fundamental importância, assim a utilização da RT-UML vem ganhando força no domínio dos sistemas tempo-real embarcados.

Contudo, a RT-UML não possui uma semântica formal definida o que permite ambigüidades na sua interpretação. Em razão disso o mapeamento dos elementos modelados para linguagens de programação com uma semântica formal se faz necessária. Como o modelo RT-UML permite a expressão dos requisitos temporais e mecanismos para a implementação de funcionalidades que envolvam requisitos tempo-real, se faz necessário a utilização de linguagens de programação que também permitem expressar esses requisitos e mecanismos de tempo-real.

Baseado nas afirmações anteriores, o presente trabalho propõe um mapeamento de alguns elementos disponíveis na RT-UML para as construções disponíveis no *framework* orientado a objetos tempo-real proposto. O objetivo principal neste mapeamento é possibilitar que o desenvolvedor possa fazer uma transição suave do modelo de um sistema tempo-real embarcado para a sua implementação, promovendo a integração da fase de modelagem com a fase de implementação no ciclo de vida do projeto.

Basicamente, os elementos disponíveis na RT-UML que serão considerados para o mapeamento estão presentes no *framework* para modelagem de tempo e no *framework* para modelagem da análise de escalonabilidade presentes no perfil de escalonabilidade, performance e tempo (OMG, 2002). Serão considerados apenas elementos destes dois *frameworks* do perfil citado pelo fato do *framework* orientado a objetos tempo-real proposto neste trabalho possibilitar a expressão apenas de valores de tempo, *threads* concorrentes, escalonadores e temporizadores (*timers*), não fazendo sentido analisar os outros elementos disponíveis no perfil, pois os mesmos não possuem suporte no *framework* proposto.

### 5.4.2 Mapeamento do Modelo RT-UML para Framework RT-OO

A descrição do mapeamento do modelo RT-UML para os elementos disponíveis no *framework* orientado a objetos tempo-real será iniciada com a análise do estereótipo «*SAschedulableResource*» presente no *framework* modelagem análise de escalonabilidade. Este estereótipo estende o estereótipo «*SAresource*» e representa um recurso que pode ser escalonado sendo que, segundo (OMG, 2002), é um recurso concorrente que possui sua própria *thread* pode executar ações. Este elemento pode ser associado ao estereótipo «*CRconcurrent*» definido no *framework* de modelagem de concorrência embora a especificação da RT-UML não defina nenhum tipo de relacionamento entre estes elementos. O recurso escalonável é usado para representar a

execução de uma ou mais ações representadas pelo estereótipo «*SAaction*». Resumindo, ambos os estereótipos, «*SAschedulableResource*» e «*CRconcurrent*» são especificados como elementos para identificar uma unidade de execução concorrente e.g. uma tarefa, um processo ou um *thread*. A Figura 5.16 apresenta um exemplo de modelagem de um recurso de escalonamento com ativação periódica, que pode ser considerado como uma tarefa periódica do sistema tempo-real embarcado.

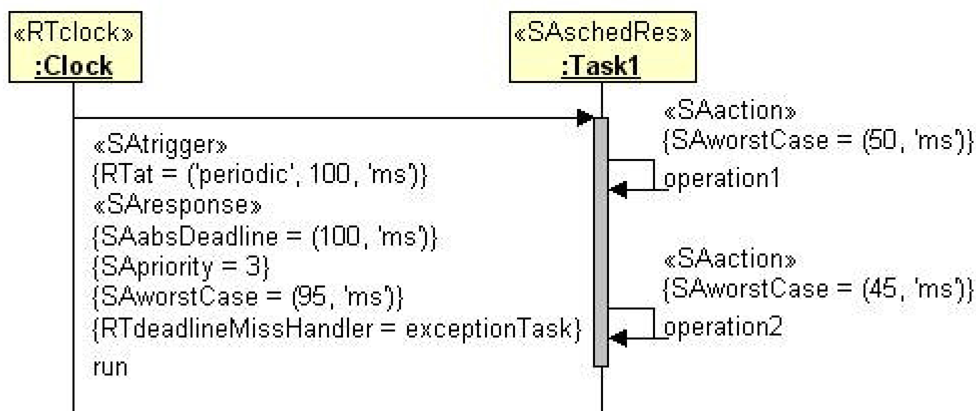


Figura 5.16: Modelagem de um recurso escalonável com ativação periódica

A idéia de ação definida pelo estereótipo «*SAaction*» é estendida pelo estereótipo «*SAresponse*» que caracteriza uma situação de uso («*SAsituation*»), ou seja, uma atividade a ser executada. O estereótipo «*SAtrigger*» representa um evento que ativa a execução de uma atividade. Sendo assim um estereótipo «*SAresponse*» em conjunto com um «*SAtrigger*» representam um conjunto de interações que representam um *job* que deve ser executado por um recurso escalonável.

Analisando a Figura 5.16 pode-se associar alguns dos estereótipos e *tagged values* com diversos elementos do *framework* orientado a objetos tempo-real proposto neste trabalho. O primeiro estereótipo analisado é o «*SAschedulableResource*» pois ele representa uma unidade concorrente do sistema, sendo assim, claramente se vê a sua associação com a classe *RealtimeThread*. Logo a classe *Task1* representa uma classe que estende a classe *RealtimeThread* como o apresentado na linha 01 da Figura 5.17.

O próximo passo é analisar os valores de tempo expressos no diagrama da Figura 5.16, onde se pode notar que todos são valores relativos de tempo que podem ser mapeados diretamente para objetos da classe *RelativeTime*, como pode ser observado nas linhas 03, 04 e 05 da Figura 5.17. O estereótipo «*SAresponse*» através dos *tagged values SAabsDeadline* e *SAworstCase* representam respectivamente os valores do deadline (linha 04) e o WCET (linha 03) da *thread*. Além destes valores de tempo o estereótipo «*SAresponse*» indica a prioridade da *thread* que é mapeada em um objeto *PriorityParameters* (linha 12) e pode ser usado pelo escalonador *PriorityScheduler* definido no *framework*. Já «*SAtrigger*» através do *tagged value RTat* indica que a *thread* é periódica, sendo que o período de ativação é de 100 ms. Assim o seu mapeamento direto para um objeto *RelativeTime* representando o período da *thread* (linha 05) e um objeto *PeriodicParameters* representando os parâmetros de liberação da *thread* (linha 06) fica evidenciado.

```
01  public class Task1 extends RealtimeThread
02  {
03    protected static RelativeTime m_Cost = new RelativeTime(0,95,0);
04    protected static RelativeTime m_Deadline = new RelativeTime(0,100,0);
05    protected static RelativeTime m_Period = new RelativeTime(0,100,0);
06    protected static PeriodicParameters m_RelParams = new PeriodicParameters(
07                                                           null,//start time
08                                                           null,//end time
09                                                           m_Period,
10                                                           m_Cost,
11                                                           m_Deadline);
12    protected static PriorityParameters m_schPar = new PriorityParameters(3);
13    protected static AbsoluteTime m_Task1RelTime = new AbsoluteTime(0,0,0);
14
15    // other attribute declarations
16
17    public Task1()
18    {
19      super(m_schPar,
20            m_RelParams);
21      m_ReleaseTime = m_Task1RelTime;
22    }
23
24    public void mainTask()
25    {
26      while (m_isRunning)
27      {
28        operation1();
29        operation2();
30        waitForNextPeriod();
31      }
32    }
33
34    public void exceptionTask()
35    {
36    }
37
38    protected void initializeStack()
39    {
40      // do nothing
41    }
42  }
```

Figura 5.17: Mapeamento do modelo RT-UML para os elementos do *framework* (*RealtimeThread*)

Outro mapeamento importante é o realizado entre o método *run()* da Figura 5.16, que representa a atividade que deve ser executada, e o método *mainTask()* da classe Task1 (linhas 24-32), que representa o código principal de uma *thread*. Optou-se por utilizar o mapeamento do método *run()* para indicar o código de uma *thread* devido a este ser o método padrão para a execução de uma *thread* na linguagem Java e da especificação RTSJ. Como este método possui o estereótipo «*SAtrigger*» indicando em seu *tagged value RTat* que a sua ativação é periódica, o mapeamento para o método *mainTask()* deve incluir o laço das linhas 26 à 36, fazendo com que a *thread* repita ciclicamente a execução das ações (linhas 28 e 29) esperando o próximo período de ativação para a *thread* (linha 30).

Caso a *thread* que está sendo modelada for aperiódica, o estereótipo «*SAtrigger*» não deve possuir o *tagged value RTat*, sendo que para o seu mapeamento no *framework* deve ser utilizada a classe *AperiodicParameters* no lugar de *PeriodicParameters* definida no exemplo da Figura 5.17 (linha 06). Caso a *thread* seja esporádica, o valor de *RTat* deve ser, por exemplo "*RTat*=('*sporadic*', 20, 'ms')", sendo que 20 é o tempo mínimo de intervalo entre dois eventos consecutivos. O seu mapeamento direto é com

um objeto da classe *SporadicParameters* e o tempo mínimo entre duas ocorrências é mapeado em um objeto da classe *RelativeTime*.

O último mapeamento da Figura 5.16 é o do valor *RTdeadlineMissHandler* do estereótipo «*SAresponse*» o qual representa o tratamento de uma exceção da perda de um *deadline* da atividade. O valor *RTdeadlineMissHandler* foi proposto por Becker (BECKER, 2003) e não existe na proposta inicial do perfil de escalonabilidade, performance e tempo. Entretanto, considerou-se que este valor poderia ser utilizado, tendo em vista a sua correspondência com o método *exceptionTask()* proposto na classe *RealtimeThread* do *framework*.

Outro mecanismo importante utilizado com freqüência em aplicações tempo-real é o uso de temporizadores (*timers*). A modelagem de um temporizador utilizando RT-UML pode ser vista na Figura 5.18.

O estereótipo «*RTtimer*» tem mapeamento direto para as subclasses da classe *Timer*, dependendo da existência ou não do *tagged value RTperiodic*. Na Figura 5.19 é mostrado o mapeamento da classe *Task1Timer* como sendo uma classe que estende a classe *OneShotTimer* (linha 01) pelo fato de não existir o *tagged value RTperiodic* no modelo apresentado na Figura 5.18. Caso este *tagged value* tivesse sido utilizado, a classe *Task1Timer* seria mapeada para estender a classe *PeriodicTimer*.

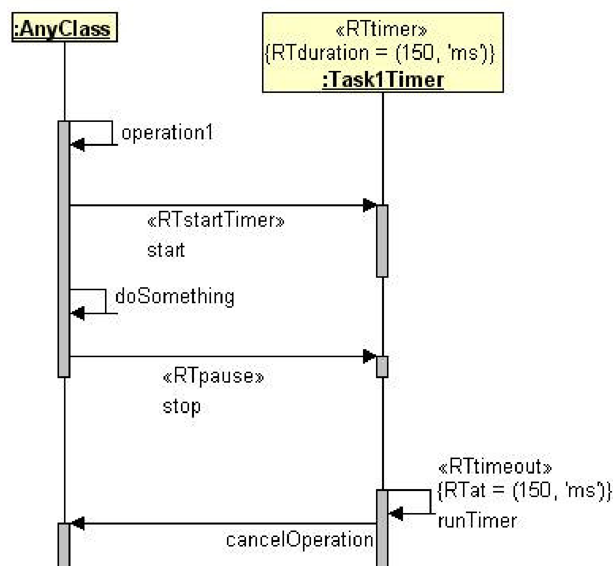


Figura 5.18: Modelagem de um temporizador com disparos periódicos

O *tagged value RTduration* representa o valor de tempo que será aguardado até que o sinal de *timeout* seja gerado para a aplicação tempo-real. O mapeamento deste *tagged value* é feito criando uma instância de um objeto *RelativeTime* o qual é passado para o construtor da classe *OneShotTimer*.

```
01  public class Task1Timer extends OneShotTimer
02  {
03    protected static RelativeTime m_Timeout = new RelativeTime(0,150,0);
04
05    public Timer1()
06    {
07      super(_100_ms, // timeout
08            1);      // Timer ID
09    }
10
```

```
11     protected void runTimer()
12     {
13       anyClass.cancelOperation();
14     }
15   }
```

Figura 5.19: Mapeamento do modelo RT-UML para os elementos do *framework* (*PeriodicTimer*)

Na RT-UML ainda existem outros estereótipos para modelar operações que podem ser efetuadas nos temporizadores como: definir um valor de *timeout* («*RTset*»); iniciar contagem («*RTstartTimer*»); parar contagem («*RTpause*»); e reiniciar contagem («*RTreset*»). Estes estereótipos têm relação direta com os métodos presentes na classe *Timer*. Sendo assim, os métodos que são anotados com o estereótipo «*RTstart*» são mapeados para o método *start()* e os métodos que contém os estereótipo «*RTpause*» são mapeados para o método *stop()*.

O estereótipo «*RTtimeout*» indica o método que deve ser executado quando acontece um *timeout*, no caso do exemplo da Figura 5.18 em 150 ms. Um método que possui este estereótipo é mapeado para o método *runTimer()* que implementa o código que será executado quando acontecer o *timeout*.

É importante deixar claro que, a sugestão de modelagem de temporizadores proposta pela OMG (OMG, 2002) é um pouco diferente do sugerido para a modelagem neste trabalho. Segundo (OMG, 2002) para se modelar a criação de um temporizador deve-se utilizar o estereótipo «*RTnewTimer*» em um método, com o *tagged value RTtimerPar* indicando o período de contagem para o *timeout*. Contudo como a criação de objetos está limitada pela ferramenta SASHIMI para permitir apenas a alocação estática de objetos, optou-se por definir a modelagem de um temporizador como o apresentado na Figura 5.18 com o período de contagem para o *timeout* definido no *tagged value RTduration*.

Para concluir esta seção, a Tabela 5.5 apresenta os elementos do perfil de escalonabilidade, performance e tempo que devem ser aplicados nos elementos dos modelos UML, de modo que estes modelos possam ser mapeados para código fonte Java que faz uso do *framework* orientado a objetos tempo-real proposto neste trabalho. O perfil UML para Escalonabilidade, Performance e Tempo define uma série de outros esteriótipos que não foram utilizados no mapeamento para a API proposta neste trabalho. Isto se deve principalmente pela grande quantidade de esteriótipos disponíveis no diversos *frameworks* que fazem parte deste perfil.

Tabela 5.5: Elementos do perfil RT-UML e seus respectivos mapeamentos para o *framework*

| **Elemento RT-UML** | **Significado** | **Elemento *framework*** |
|---|---|---|
| «*SAschedulableResource*» | Recurso concorrente que pode executar ações | *RealtimeThread* |
| «*SAresponse*» | Atividade a ser executada | |
| *~.SAabsDeadline* | Tempo máximo de execução (*deadline*) | *RelativeTime* |
| *~.SAworstCase* | Tempo do WCET de execução da atividade | *RelativeTime* |
| *~.PriorityParameters* | Indica a prioridade de uma *thread*, usado apenas em algoritmos de escalonamento estáticos | *PriorityParameters* |
| *~.RTdeadlineMissHandler* | Define o método para o tratamento de exceção de perda de *deadline* | *RealtimeThread. exceptionTask()* |

| | | |
|---|---|---|
| «*SAtrigger*» | Evento que ativa a execução de uma atividade | |
| ~.*RTat*=(‘*periodic*’, período_ativação, unidade_tempo) | Indica que a ativação da execução de uma atividade é periódica, com um período de *período_ativação* unidades de tempo | *PeriodicParameters, RelativeTime* |
| ~.*RTat*=(‘*sporadic*’, período_mínimo, unidade_tempo) | Indica que a ativação da execução de uma atividade é esporádica, com um período mínimo de *período_mínimo* unidades de tempo entre duas ocorrências | *SporadicParameters, RelativeTime* |
| ~. | Sem o *tagged value RTat*, indica uma ativação aperiódica | *AperiodicParameters* |
| «*RTtimer*» | Temporizador | *OneShotTimer* |
| ~.*RTperiodic* | Indica que o temporizador é cíclico | *PeriodicTimer* |
| ~.*RTduration* | Indica a quantidade de tempo que será aguardado até que o sinal de *timeout* seja disparado | *RelativeTime* |
| «*RTtimeout*» | Indica o método que deve ser executado quando ocorrer um sinal de *timeout* | *runTimer()* da classe *OneShotTimer* ou *PeriodicTimer* |
| «*RTstartTimer*» | Iniciar contagem de tempo do temporizador | *Timer.start()* |
| «*RTpause*» | Para contagem de tempo do temporizador | *Timer.stop()* |

## 5.5 Geração Automática de Código Fonte

Na seção anterior foi discutido o mapeamento de modelos UML anotados com o perfil de escalonabilidade, performance e tempo para construções da linguagem de programação Java utilizando o *framework* orientado a objetos proposto nesta dissertação. Sendo assim foi demonstrado que a geração de código fonte Java utilizando a biblioteca é possível de ser realizada.

Para fazer a geração automática de código fonte Java, a partir de um modelo RT-UML, existem duas possibilidades: adaptar ferramentas comerciais existentes no mercado ou criar uma ferramenta para realizar essa geração automática de código fonte. Para criar uma ferramenta de geração automática de código fonte a partir do inicio, necessita de uma série de etapas, entre elas:

- Ler o modelo RT-UML do repositório da ferramenta CASE que foi utilizada. Normalmente cada ferramenta possui um formato de armazenamento diferente para o modelo fazendo com que, para cada ferramenta, tenha que ser feito um estudo da organização e de forma de armazenamento do modelo;
- Para que a ferramenta de geração de código não seja dependente de uma única ferramenta CASE, é necessário criar uma estrutura intermediária para a representação para o modelo RT-UML que possibilite representar todos os elementos do modelo (e.g. classes, atributos, métodos, objetos, relacionamento entre classes e objetos, estereótipo, tagged values, entre muitos outros elementos pertencentes ao modelo);

- A fim de flexibilizar a geração automática de código fonte seria necessário criar também uma linguagem de script que seria responsável por conduzir o processo de geração de código, associando elementos da estrutura intermediária com elementos da linguagem alvo. Isso possibilitaria que a geração automática do código fosse realizada para outras linguagens que possuíssem uma versão compatível do *framework* proposto nesta dissertação.

Analisando todos os requisitos para a criação de uma ferramenta de geração automática de código fonte, optou-se por tentar fazer a geração automática de código adaptando-se uma das diversas ferramentas CASE disponível no mercado. Escolheu-se adaptar a ferramenta *Artisan Real-Time Studio* 4.3 (ARTISAN, 2004) visto que esta é uma das poucas ferramentas disponíveis no mercado que suporta a utilização do perfil para escalonabilidade, performance e tempo. Outro motivo desta escolha foi o fato desta ferramenta estar disponível para utilização no Instituto de Informática através de uma cooperação que a universidade possui com a empresa Artisan Software.

Iniciou-se então o estudo da ferramenta de geração de código fonte disponível juntamente com a ferramenta de modelagem *Artisan Real-Time Studio* 4.3 (ARTISAN, 2004). Verificou-se que encontravam-se disponíveis as seguintes ferramentas de geração de código:

- *C Synchronizer*, *C++ Synchronizer*: estas duas ferramentas permitem a geração de código C e C++ respectivamente, assim como permitem a engenharia reversa de códigos existentes. Elas possuem um mecanismo de *templates* para fazer a geração de código e podem ser usadas em conjunto com a ferramenta *State Machine Generator* que permite gerar o comportamento dinâmico das classes;
- *State Machine Generator*: esta ferramenta interpreta os diagramas de estado modelados para cada classe permitindo a geração de código contendo o comportamento dinâmico das classes. O código gerado é compatível com C ou C++ dependendo da opção selecionada no assistente de geração de código fonte;
- *Java Synchronizer*: assim como as demais ferramentas *Synchronizer*, esta ferramenta permite fazer a geração da estrutura de classes na linguagem Java. Porém esta ferramenta não utiliza *templates* para a geração de código, sendo assim esse procedimento não pode ser adaptado.

Apesar de todas estas ferramentas disponíveis para se fazer a geração de código fonte a partir do repositório da ferramenta *Artisan Real-Time Studio* não foi possível a implementação da geração automática de código de acordo com o mapeamento proposto na seção anterior.

Inicialmente tentou-se utilizar a ferramenta *Java Synchronizer* pois o *framework* foi proposto para ser utilizado com a linguagem Java permitindo o uso da ferramenta SASHIMI. Depois de alguns experimentos, concluiu-se que a utilização dessa ferramenta não era possível devido ao processo de geração de código fonte disponível na ferramenta não poder ser adaptado. Assim o código fonte gerado representava apenas a estrutura de classes que tinha sido modelada no *Artisan Real-Time Studio*, não sendo possível identificar quais elementos possuíam estereótipos impossibilitando fazer a geração de código fonte de acordo com o mapeamento proposto na seção anterior.

A segunda tentativa foi a utilização da ferramenta *C++ Synchronizer* que possuía um mecanismo de *templates* para adaptação da geração automática de código fonte. Novamente não foi possível a geração de código fonte baseado no mapeamento proposto na seção anterior devido ao fato de não poder acessar os estereótipos que decoravam nos elementos UML, através da linguagem de scripts.

# 6 ESTUDOS DE CASO

Neste capítulo serão apresentados dois estudos de caso que exemplificam a utilização dos elementos disponíveis no *framework* orientado a objetos tempo-real proposto neste trabalho. Também são demonstrados os mapeamentos dos elementos no modelo RT-UML para os elementos do *framework.* O primeiro estudo de caso é a automação de uma cadeira de rodas de modo a embutir "inteligência" em uma cadeira de rodas elétrica. O segundo estudo de caso ilustra uma aplicação industrial através de um guindaste.

## 6.1 Cadeira de Rodas

Este estudo de caso apresenta o projeto de automação de uma cadeira de rodas que está sendo desenvolvido no âmbito do projeto SEEP (LSE, 2004), como validação da metodologia proposta no projeto, em parceria com a empresa Freedom. A motivação principal deste estudo de caso é auxiliar no transporte de pessoas com necessidades especiais de forma a maximizar o seu bem estar, controlando os seus sinais vitais i.e. respiração, freqüência cardiovascular, entre outros. Na Figura 6.1 é apresentado o levantamento da funcionalidades para esta aplicação.

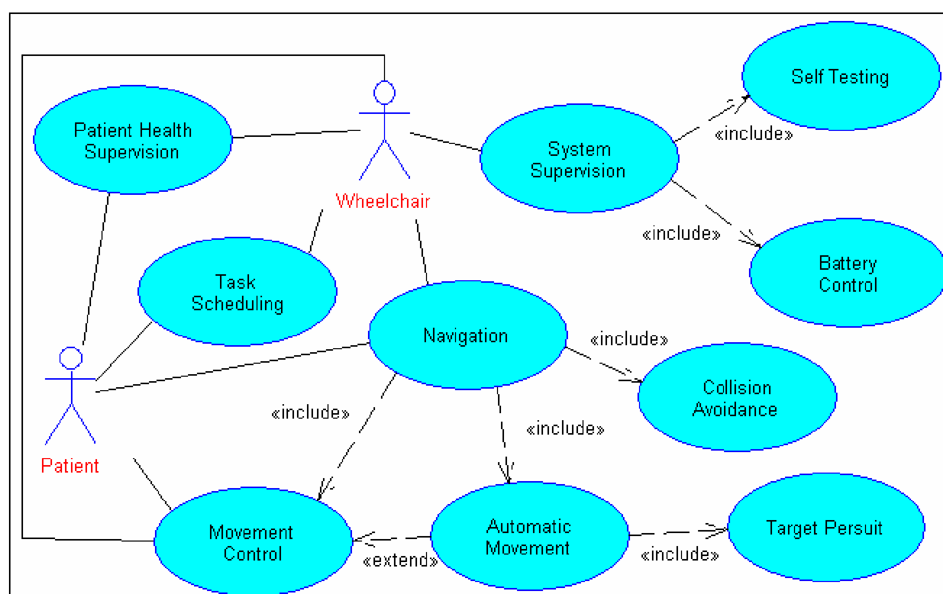

Figura 6.1: Diagrama de casos de uso indicando todas as funcionalidade da cadeira da rodas

Esta seção abordará a funcionalidade do controle da movimentação da cadeira de rodas, cujo diagrama de casos de uso é apresentado na Figura 6.2. Leitores interessados em maiores detalhes sobre este estudo de caso devem consultar (GREFF *et al.*, 2004). O

controle de movimentação funciona da seguinte maneira: a direção e a velocidade de locomoção da cadeira de rodas são fornecidos através da utilização de um joystick, que se encontra em um dos braços da cadeira. A direção do movimento é dada através do movimento do joystick para frente ou para trás. De maneira análoga, quando se move o joystick para a direita ou para a esquerda, é definido qual o ângulo de rotação para o movimento da cadeira de rodas. Já a velocidade é fornecida através da posição do joystick em relação ao seu centro, quando mais distante do centro, maior a velocidade aplicada no movimento.

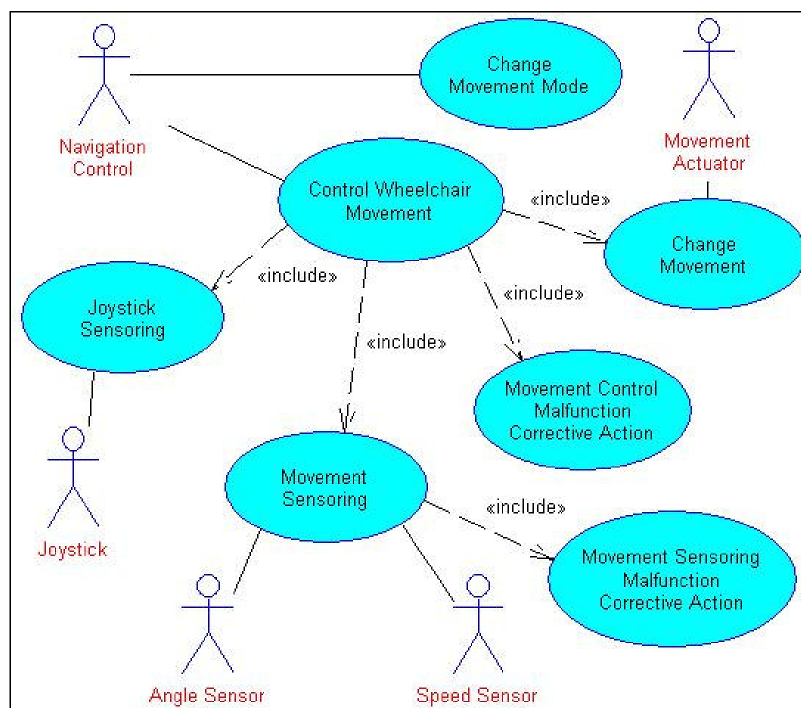


Figura 6.2: Diagrama de casos de uso do controle de movimentação da cadeira de rodas

Segundo a especificação do sistema de controle do movimento da cadeira de rodas, o sistema é composto por 4 tarefas periódicas com as seguintes características:

- Uma tarefa para monitorar a troca do modo de operação do controle de movimento. O modo de operação indica qual é a ação que deve ser tomada quando ocorrer uma perda de *deadline* em alguma das tarefas do sistema de controle de movimento. Existem quatro modos de operação: silencioso, sinalizar a ocorrência de uma perda de *deadline*, sinalizar a perda do *deadline* mas continuar a operação com o último valor válido, e sinalizar o *deadline* e imediatamente frear a cadeira de rodas. Os dois primeiros modos não realizam nenhuma ação corretiva, sendo que o segundo indica, através de um sinal, que houve uma perda de *deadline*;
- Uma tarefa que amostra os sensores de movimento (sensor de velocidade e sensor de ângulo) que é ativada a cada 4 ms, com *deadline* de 4 ms. Caso ocorra uma perda de *deadline* o tratamento deste evento segue o que foi convencionado no modo de operação. O diagrama de colaboração desta tarefa pode ser observado na Figura 6.3a;
- Uma tarefa para amostrar a posição do joystick que também é ativada a cada 4 ms e também possui *deadline* de 4 ms, sendo que o respectivo diagrama de colaboração pode ser visto em Figura 6.3b;

- Uma tarefa responsável pelo controle do movimento baseado nos valores amostrados nos sensores e no modo de operação ativo no momento. O período de ativação desta tarefa é de 15 ms possuindo o *deadline* de 15 ms. O diagrama da colaboração da Figura 6.4a ilustra esta tarefa. As ações que são tomadas caso ocorra a perda de um *deadline* são apresentadas na Figura 6.4b, sendo que estas seguem o que foi convencionado nas ações corretivas para o modo de operação ativo.

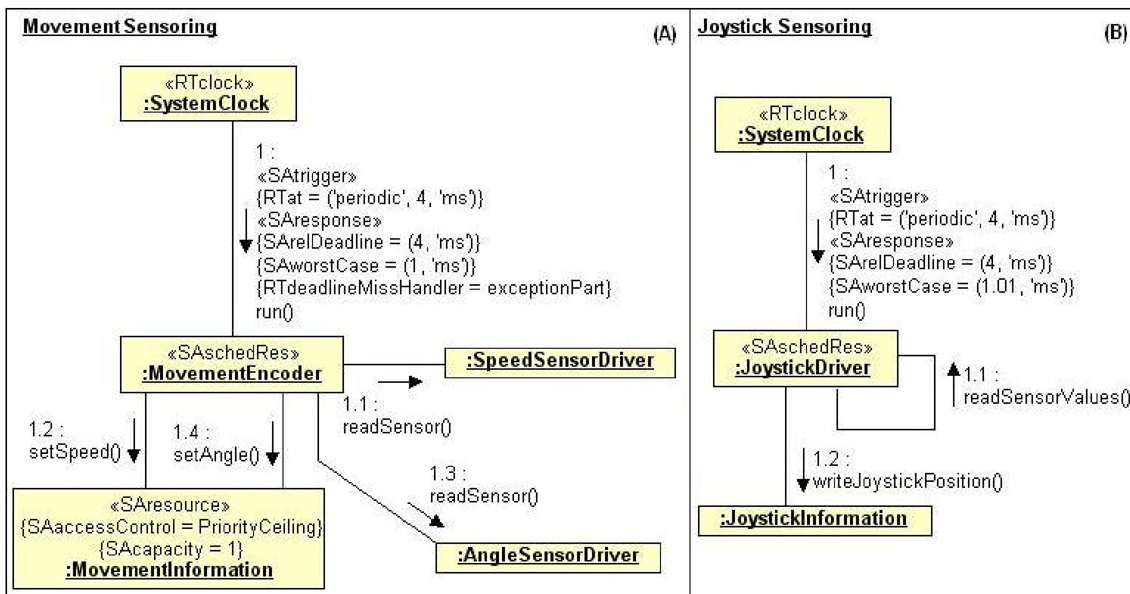


Figura 6.3: Diagrama de colaboração do sensoramento do movimento da cadeira de rodas

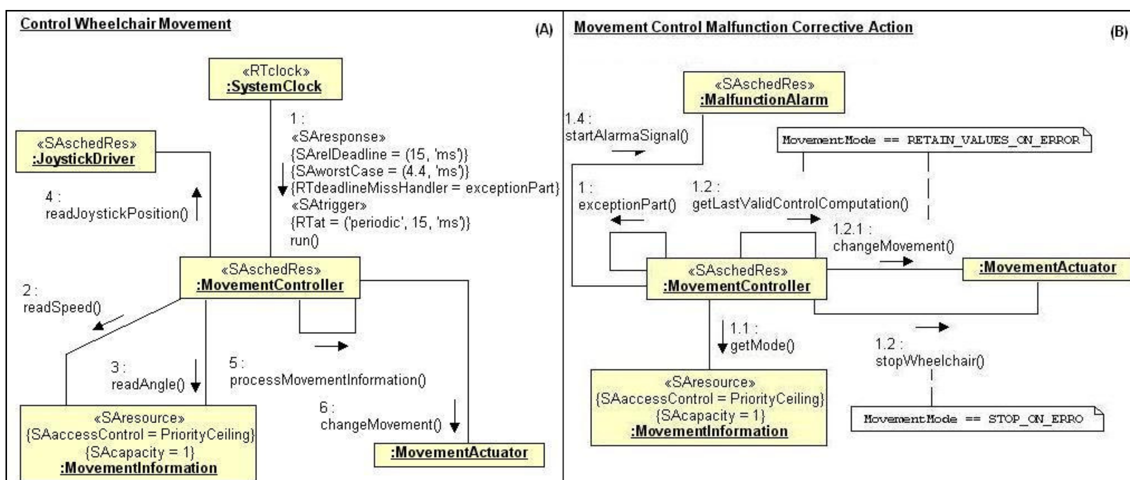


Figura 6.4: Diagramas de colaboração do controle de movimento da cadeira de rodas

A seguir serão mostrados apenas os códigos fonte das classes principais do controlador de movimento da cadeira de rodas. Na Figura 6.5 é apresentada a classe principal do controle de movimentação onde está a alocação do objeto de controle de movimento (linha 09), objeto das informações do movimento (linha 08) e do codificador dos dados dos sensores (linha 07). A política de escalonamento escolhida para esta aplicação foi o algoritmo EDF (linha 13). O método *initSystem()* representa o código que será sintetizado pela ferramenta SASHIMI e nele são feitas as inicializações dos sistema antes de entrar no laço que representa o tempo ocioso do sistema tempo-real embarcado.

```
01  public class WheelchairMovementControl
02  {
03    public static Alarm alarm = new Alarm();
04    public static JoystickDriver joystick = new JoystickDriver();
05    public static JoystickInformation
06                            joystickInfo = new JoystickInformation();
07    public static MovementEncoder movEncoder = new MovementEncoder();
08    public static MovementInformation movInfo = new MovementInformation();
09    public static MovementController
10                            movController = new MovementController();
11    public static MovementModeController
12                            movModeController = new MovementModeController();
13    public static EDFScheduler scheduler = new EDFScheduler();
14
15    public static void initSystem() {
16      Scheduler.setDefaultScheduler(scheduler);
17      joystick.addToFeasibility();
18      movEncoder.addToFeasibility();
19      movController.addToFeasibility();
20      movModeController.addToFeasibility();
21      joystick.start();
22      movEncoder.start();
23      movController.start();
24      movModeController.start();
25      scheduler.setupTimer();
26      idleTask();
27    }
28
29    public static void idleTask() {
30      while (true)
31        FemtoJava.sleep();
32    }
33
34    public static void main(String[] args) {
35      WheelchairMovementControl.initSystem();
36    }
37  };
```

Figura 6.5: Código Java da classe principal do controle de movimento da cadeira de rodas

O próximo trecho de código que será analisado é o código da classe *MovementController*, que é apresentado na Figura 6.6. Comparando o código com o diagrama de colaboração apresentado na Figura 6.4a pode-se notar que os parâmetros de liberação para a *thread* indicam que ela é uma *thread* periódica (linhas 09-15) com pior caso de execução (linha 14) de 4,4 ms, período (linha 13) e *deadline* (linha 15) de 15 ms.

O código principal da *thread* (linhas 23-34) segue o que foi modelado no diagrama de colaboração da Figura 6.4a, ou seja, primeiro é feita a leitura das informações do joystick, em seguida são lidos os valores dos sensores de velocidade e ângulo, para então realizar os cálculos de controle do movimento. No final do código é chamado o método *waitForNextPeriod()* que faz com que a *thread* aguarde o seu próximo período de ativação. Já o código do tratamento da exceção de perda do *deadline* (linhas 36-50) corresponde ao diagrama de colaboração da Figura 6.4b, executando a seqüência de ações corretivas de acordo com o modo de operação do controle de movimento.

```
01  public class MovementController extends RealtimeThread
02  {
03    private static AbsoluteTime m_taskResumeTime = new AbsoluteTime(0,0,0);
04    private static AbsoluteTime m_taskActiveTime = new AbsoluteTime(0,0,0);
05    private static RelativeTime
06                            m_taskPrevProcTime = new RelativeTime(0,0,0);
07    public static MovementActuator
08                            movementActuator = new MovementActuator();
```

```
09    private static PeriodicParameters
10                              releaseParams = new PeriodicParameters(
11                                              null, // start time
12                                              null, // end time
13                                              TimeObjects._15_ms, // period
14                                              TimeObjects._4_400_ms, // cost
15                                              TimeObjects._15_ms); // deadline
16    public MovementController() {
17      super(null, releaseParams);
18      m_ResumeTime = m_taskResumeTime;
19      m_ActiveTime = m_taskActiveTime;
20      m_PreviousProcessTime = m_taskPrevProcTime;
21    }
22
23    public void mainTask() {
24      while (isRunning()) {
25        int xpos = WheelchairMovementControl.joystickInfo.getXPos();
26        int ypos = WheelchairMovementControl.joystickInfo.getYPos();
27        int newSpeed = WheelchairMovementControl.movInfo.readSpeed();
28        int newAngle = WheelchairMovementControl.movInfo.readAngle();
29        // ...
30        // movement control calculations
31        // ...
32        waitForNextPeriod();
33      }
34    }
35
36    public void exceptionTask() {
37      int mode = WheelchairMovementControl.movInfo.getMode();
38      if (mode == MovementInformation.ALARM_ONLY)
39        WheelchairMovementControl.alarm.startAlarmSignal();
40      else if (mode == MovementInformation.RETAIN_VALUES_ON_ERROR) {
41        getLastValidControlComputation();
42        movementActuator.changeMovement(m_LastValidSpeedValue,
43                                        m_LastValidAngleValue);
44        WheelchairMovementControl.alarm.startAlarmSignal();
45      }
46      else if (mode == MovementInformation.STOP_ON_ERROR) {
47        movementActuator.stopWheelchair();
48        WheelchairMovementControl.alarm.startAlarmSignal();
49      }
50    }
51    protected void initializeStack() {} // SASHIMI will fill it with code
52    // other class methods
53  };
```

Figura 6.6: Código Java da classe MovementController

Este experimento foi sintetizado pela ferramenta SASHIMI, a qual sintetizou os objetos ocupando um total de 588 bytes de memória RAM para os atributos dos objetos e 5284 bytes da memória ROM para armazenar o código dos métodos. A execução deste estudo de caso foi feita utilizando a ferramenta *Cycle Accurate Configurable Power Simulator* (CACO-PS) (BECK FILHO *et al.*, 2003) que é um simulador ciclo a ciclo que pode medir o desempenho da aplicação (em ciclos) e o consumo de potência (em termos de chaveamentos de capacitâncias das portas).

A versão do processador FemtoJava utilizado na simulação com o CACO-PS foi a versão multiciclo de 32-bits com o RTC proposto nesta dissertação incluído, funcionando a uma freqüência de 20 MHz. Esta freqüência foi escolhida pelo fato de ser a freqüência mais baixa do FemtoJava que permitia que todas as tarefas executassem sem perda de *deadline*, em outras palavras, com esta freqüência o tempo de execução das tarefas diminui e o teste de escalonabilidade fica com carga de utilização menor que 1. A simulação do controle de movimento foi mensurada durante 200 ms. As características temporais que foram apuradas através dos dados fornecidos pelo

simulador CACO-PS são apresentadas na Tabela 6.1, onde as informações temporais do experimento são apresentadas em milisegundos.

Analisando as informações apresentadas na Tabela 6.1 pode-se perceber que os tempos de execução apresentados para as quatro tarefas periódicas representam o tempo de processamento da tarefa mais o tempo que o escalonador necessitou para rodar o seu algoritmo de escalonamento. Também pode-se notar a ausência da tarefa aperiódica que dispara e faz o controle do alarme, o qual indicaria a perda de um *deadline*. Essa ausência é justificada pela não ocorrência de perdas de *deadline*.

Tabela 6.1: Dados da simulação do controle de movimento

| | Nr. Exec. | Mínimo | Médio | Máximo |
|---|---|---|---|---|
| **JoystickDriver** | | | | |
| Tempo Execução | 50 | 0,65445 | 0,7228204 | 0,82225 |
| Latência de ativação | | 0,61045 | 1,0963684 | 2,4302 |
| Jitter de ativação | | -1,7211 | -0,0170327 | 1,0993 |
| **MovementEncoder** | | | | |
| Tempo Execução | 50 | 0,7293 | 0,7463204 | 0,8247 |
| Latência de ativação | | 1,34285 | 1,8364714 | 3,2088 |
| Jitter de ativação | | -1,77165 | -0,0185224 | 1,14655 |
| **MovementController** | | | | |
| Tempo Execução | 19 | 2,9678 | 3,0379975 | 3,13385 |
| Latência de ativação | | 0,59045 | 1,3717842 | 2,3194 |
| Jitter de ativação | | -2,6597 | -0,1398395 | 1,72635 |
| **MovementModeController** | | | | |
| Tempo Execução | 4 | 0,71535 | 0,7181625 | 0,72055 |
| Latência de ativação | | 5,32695 | 5,8879333 | 7,0099 |
| Jitter de ativação | | -2,61265 | -0,8708833 | 1,68295 |
| **EDFScheduler** | | | | |
| Tempo Execução | 193 | 0,38335 | 0,5937674 | 0,76595 |

Com relação à latência de ativação das tarefas, o valor apresentado na Tabela 6.1 representa a diferença entre o instante de tempo no qual a tarefa deveria começar a executar e o instante real do início da execução da tarefa. O valor apresentado como *jitter* de ativação representa a diferença entre dois instantes de ativação consecutivos de uma tarefa, em outras palavras, representa a diferença da latência de ativação entre duas execuções consecutivas das tarefas.

A política de escalonamento de tarefas definida para este experimento foi a EDF, um algoritmo de escalonamento dinâmico onde a tarefa que tiver o *deadline* mais próximo é a que possui prioridade maior de execução. Este é um algoritmo de escalonamento mais complexo e requer uma quantidade de processamento maior que outros algoritmos disponíveis no *framework*. Analisando os tempos de execução do objeto *EDFScheduler* pode-se notar que o tempo máximo e mínimo de execução varia bastante e, neste experimento, o WCET do algoritmo foi de 0,76595 milisegundos. As medidas de latência e *jitter* da ativação das tarefas também variam. Isto se deve ao critério adotado pelo EDF para a definição das prioridades das tarefas que, como é baseado no *deadline*, faz com que os tempos de ativação tenham uma grande variação, diferentemente do algoritmo *Rate Monotonic* (também disponível no *framework*), onde a prioridade se baseia no período de ativação das tarefas.

## 6.2 Sistema de Guindastes

Neste estudo de caso é apresentado um sistema embarcado para controle de guindastes, o qual é considerado um *benchmark* para sistemas tempo-real (MOSER; NEBEL, 1999). A Figura 6.7 ilustra o sistema de controle do guindaste que move uma carga ao longo de seu trilho para uma posição desejada. O sistema é composto por um carro que se move ao longo de um trilho e um cabo de tamanho fixo onde a carga é conectada.

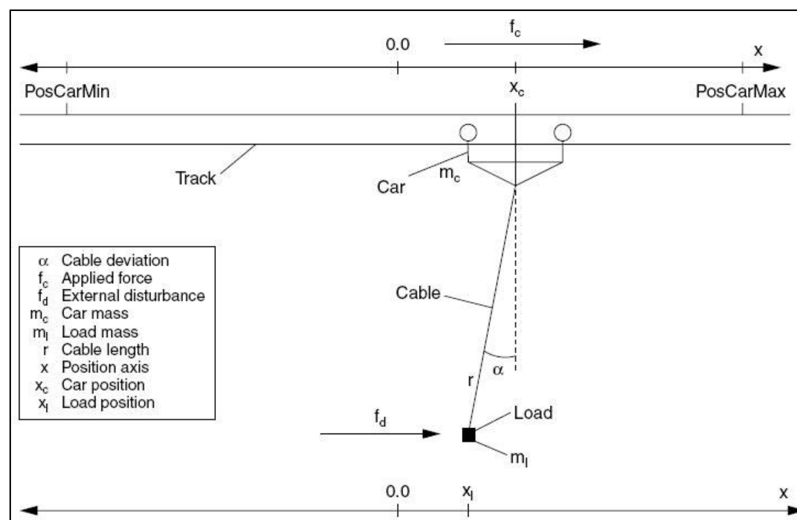

Figura 6.7: Sistema de controle de guindastes movendo uma carga (MOSER; NEBEL, 1999)

O sistema embarcado inclui sensores, atuadores e processos de controle. Os sensores disparam eventos que servem como entrada para o sistema e controlam a posição atual do carro, a posição mínima e máxima do carro no trilho, bem como o ângulo do cabo onde a carga está conectada. Entre os atuadores estão aqueles que controlam o motor e o freio, que freia imediatamente o carro. Os processos de controle são compostos por: um processo de inicialização que faz a checagem dos sensores para definir os valores máximo e mínimo da posição do carro no trilho; um processo de diagnóstico que fica observando a posição do carro e o ângulo do cabo; um processo que espera os comandos de movimento do carro e aplica o algoritmo de controle. As funcionalidades do sistema de controle de guindastes podem ser visto no diagrama de casos de uso da Figura 6.8.

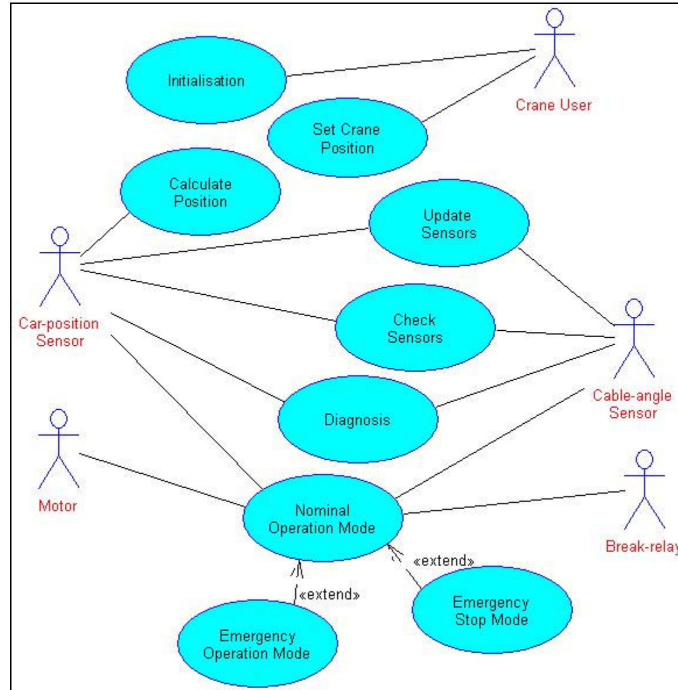


Figura 6.8: Diagrama de casos de uso do sistema de controle de guindastes

Nesta seção será apresentado o mapeamento do modelo RT-UML para código fonte Java utilizando o *framework* proposto neste trabalho, através dos diagramas de colaboração para o caso de uso da operação normal do sistema (Figura 6.9a) e para caso de uso do deslocamento do guindaste (Figura 6.9b).

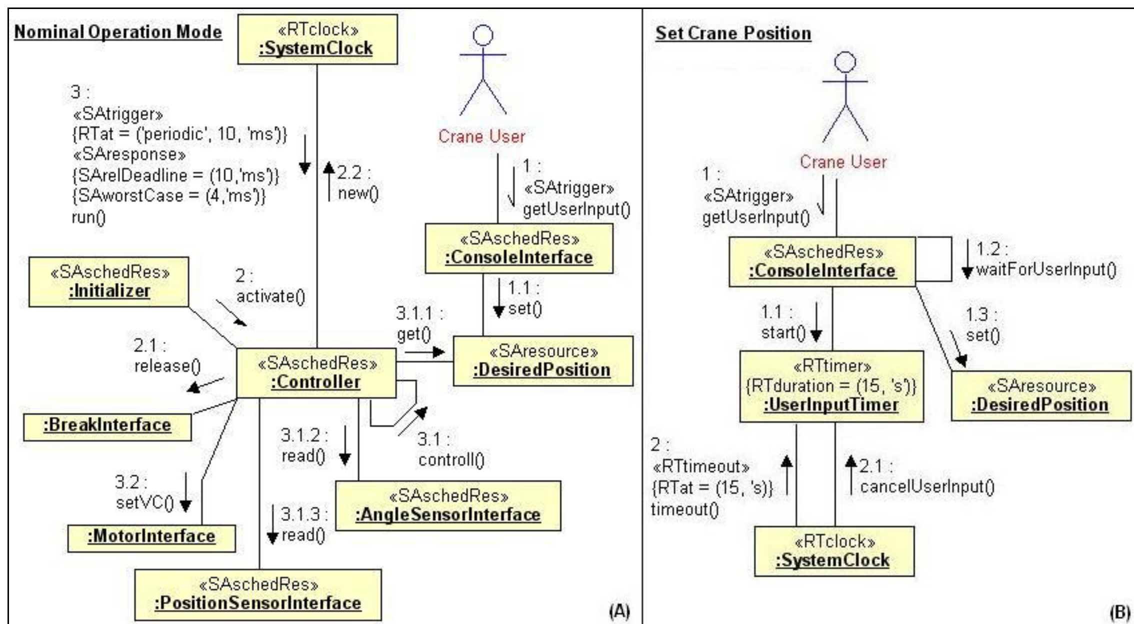


Figura 6.9: Diagramas de colaboração da operação normal do controle de guindastes

A operação normal do guindaste é representada no diagrama de colaboração da Figura 6.9a onde é possível perceber que existe um processo periódico que é ativado a cada 10 ms. O *deadline* e o WCET do tempo de execução desta *thread* são respectivamente 10 e 4 ms. Pode se observar que este processo é ativado após a chamada do método *activate()*.

A implementação da classe principal do sistema de controle de guindastes pode ser vista na Figura 6.10. Nesta classe são feitas as alocações das *threads* do sistema, os objetos que representam os tempos são alocados dentro das classes que representam as *threads*. Os demais objetos que representam os recursos do sistema (posição máxima e mínima, ângulo do cabo, entre outros) também são alocados nesta classe e acessados pelas demais classes do sistema. A política de escalonamento definida para este experimento foi a EDF (linha 27).

```
01  public class Crane
02  {
03    // Tasks in the Crane embedded system
04    public static CraneInitializer initializer = new CraneInitializer();
05    public static Controller nominalCtrl = new Controller();
06    public static ConsoleInterface
07                        consoleInterface = new ConsoleInterface();
08    public static AngleSensorInterface
09                        angleSensorInterface = new AngleSensorInterface();
10    public static Diagnoser diagnoser = new Diagnoser();
11    public static PositionSensorInterface
12                    positionSensorInterface = new PositionSensorInterface();
13
14    // Resources of the Crane embedded system
15    public static DesiredPosition desiredPosition = new DesiredPosition();
16    public static PosCarMin posCarMin = new PosCarMin();
17    public static PosCarMax posCarMax = new PosCarMax();
18    public static DeltaPosCar deltaPosCar = new DeltaPosCar();
19    public static VcCheck vcCheck = new VcCheck();
20
21    // Other system objects
22    public static BreakInterface breakInterface = new BreakInterface();
23    public static MotorInterface motorInterface = new MotorInterface();
24    public static SwPosCarMin swPosCarMin = new SwPosCarMin();
25    public static SwPosCarMax swPosCarMax = new SwPosCarMax();
26
27    public static EDFScheduler scheduler = new EDFScheduler();
28
29    public static void initSystem() {
30      Scheduler.setDefaultScheduler(scheduler);
31      initializer.powerOn();
32      scheduler.setupTimer();
33      idleTask();
34    }
35
36    public static void idleTask() {
37      while (!scheduler.isAllTasksFinished())
38        FemtoJava.sleep();
39    }
40
41    public static void main(String[] args) {
42      Crane.initSystem();
43    }
44  };
```

Figura 6.10: Classe principal do sistema de controle de guindastes

O mapeamento do diagrama de colaboração da Figura 6.9a para a classe *Controller* é mostrado na Figura 6.11. A classe *Controller* representa uma *thread* periódica que é ativada a cada 10 ms, possui *deadline* de 10 ms e o WCET de 4 ms conforme especificado no diagrama da Figura 6.9a. O método *mainTask()* representa o controle do guindaste (linhas 35-42) e, como esta classe representa uma *thread* periódica, o código para este método possui um laço infinito (linhas 36-40) sendo assim o código efetivo da tarefa (linhas 37-38) é executado ciclicamente. No final no laço (linha 39) é executado o método que faz com que a *thread* aguarde o próximo período de ativação.

Não foi definido o código de tratamento de exceção para a perda de *deadline* por essa razão o método *exceptionTask()* aparece em "branco". O código desta classe não foi apresentado em sua totalidade pelo fato de ser muito extenso contendo uma série de cálculos de controle do guindaste, sendo assim apenas o código relacionado com o uso do *framework* foi ilustrado.

```
01  public class Controller extends RealtimeThread
02  {
03    private static AbsoluteTime m_taskResumeTime = new AbsoluteTime(0,0,0);
04    private static AbsoluteTime m_taskActiveTime = new AbsoluteTime(0,0,0);
05    private static RelativeTime
06                           m_taskPrevProcTime = new RelativeTime(0,0,0);
07    public static AbsoluteTime
08                           m_ControllerResumeTime = new AbsoluteTime(0,0,0);
09    private static RelativeTime _4_ms = new RelativeTime(0, 4, 0);
10    private static RelativeTime _10_ms = new RelativeTime(0, 10, 0);
11    private static PeriodicParameters relParams =
12                               new PeriodicParameters(null, // start time
13                                                      null, // end time
14                                                      _10_ms, // period
15                                                      _4_ms, // cost
16                                                      _10_ms); // deadline
17    private boolean isActive;
18    private int poscar, alfa;
19    // other control atributes
20
21    public Controller() {
22      super(null, relParams);
23      m_ResumeTime = m_taskResumeTime;
24      m_ActiveTime = m_taskActiveTime;
25      m_PreviousProcessTime = m_taskPrevProcTime;
26      // do other variable initializations
27    }
28    public void activate() {
29      start();
30    }
31    private int controll() {
32      // contoll actions
33    }
34
35    public void mainTask() {
36      while (true) {
37        controll();
38        Crane.motorInterface.setVC(m_vc);
39        waitForNextPeriod();
40      }
41      this.finish();
42    }
43
44    public void exceptionTask() { // treat deadline missing }
45    protected void initializeStack() {}
46    // other control methods
47  }
```
Figura 6.11: Código da classe *Controller*

A escolha da nova posição para o carro do guindaste feita pelo usuário, processo que ocorre concorrentemente ao processo de controle do guindaste, é detalhada no diagrama de colaboração da Figura 6.9b. Este diagrama de colaboração representa a interação do usuário do guindaste com a interface de operação, que aguarda a nova posição para a qual o guindaste deve se deslocar. Se o usuário iniciar o procedimento de movimentação e não informar um valor para a nova posição em 15 segundos, o procedimento é abortado. Para implementar esta funcionalidade foi utilizado um temporizador (vide Figura 6.13) e uma *thread* aperiódica. O mapeamento deste diagrama para o código

pode ser observado na Figura 6.12 através da classe *ConsoleInterface* e na Figura 6.13 através da classe *UserInputTimer*.

A classe *ConsoleInterface* representa uma *thread* aperiódica (linha 08-10) que é ativada quando o usuário do guindaste opera a interface de comando do guindaste. Assim que o usuário escolhe a opção do console relativa à mudança de posição, ele recebe uma mensagem solicitando que a nova posição seja informada. A partir deste momento o temporizador é iniciado (linhas 20-24). Se o usuário informar a operação antes do *timeout* a nova posição é salva no objeto *desiredPosition* do sistema.

```
01  public class ConsoleInterface extends RealtimeThread
02  {
03    public static AbsoluteTime m_UserInputInterfaceResumeTime =
04                                                   new AbsoluteTime(0,0,0);
05    protected static RelativeTime _15_s = new RelativeTime(0,15000,0);
06    protected static RelativeTime _2_s = new RelativeTime(0,2000,0);
07    private static UserInputTimer
08                              paramTimeOut = new UserInputTimer(_15_s, 1);
09    private static AperiodicParameters relParams = new AperiodicParameters(
10                                                        _2_s, // cost
11                                                        _15_s); // deadline
12    private boolean m_WaitUserInput;
13    private boolean m_AbortUserInput;
14    private int newPosition = 0;
15
16    public ConsoleInterface() {
17      super(null, relParams);
18      m_ResumeTime = m_UserInputInterfaceResumeTime;
19    }
20
21    public void getUserInput() {
22      paramTimeOut.enable();
23      paramTimeOut.start();
24      start();
25    }
26
27    public void waitForUserInput() {
28      m_WaitUserInput = true;
29      m_AbortUserInput = false;
30      while (m_WaitUserInput && !m_AbortUserInput)
31        FemtoJava.sleep();
32      if (!m_AbortUserInput) {
33        paramTimeOut.disable();
34        paramTimeOut.stop();
35      }
36    }
37    public void cancelUserInput() {
38      m_AbortUserInput = true;
39    }
40    public void mainTask() {
41      waitForUserInput();
42      if (!m_AbortUserInput)
43        Crane.desiredPosition.seti(newPosition);
44      this.finish();
45    }
46    public void exceptionTask() {
47    }
48    protected void initializeStack() { // do nothing
48    }
49    // other methods implementation
    }
```

Figura 6.12: Código da classe *ConsoleInterface*

A classe *UserInputTimer* representa o temporizador utilizado no caso de uso da movimentação do guindaste. Como este é um temporizador que tem como característica

controlar o timeout uma única vez, a classe *UserInputTimer* deve estender a classe *OneShotTimer*. Para representar o código que deve ser executado quando ocorrer o *timeout*, o método *executeTimer()* deve ser sobrescrito. Como pode-se observar este método tem apenas a chamada de método ao método *cancelUserInput()*, que faz com que a *thread* representada na classe *ConsoleInterface* aborte a operação de movimentação do guindaste.

```
01  public class UserInputTimer extends OneShotTimer {
02
03    public UserInputTimer(HighResolutionTime delay, int id) {
04      super(delay, id);
05    }
06
07    protected void runTimer() {
08      Crane.consoleInterface.cancelUserInput();
09    }
10  }
```

Figura 6.13: Código da classe *UserInputTimer*

Este experimento também foi sintetizado utilizando a ferramenta SASHIMI e foi simulado usando o simulador CACO-PS. Os objetos do sistema de controle de guindastes foram sintetizados ocupando um total de 1066 bytes para os seus atributos em memória RAM e 9175 bytes para o seu código em memória ROM.

A simulação também foi feita utilizando o processador FemtoJava versão multiciclo de 32-bits com o RTC, rodando com a freqüência de 15 MHz. Novamente, tem-se que a freqüência foi selecionada de modo a permitir que todas as tarefas executem sem haver perda de *deadline*, de forma análoga ao experimento da cadeira de rodas. A simulação do controle de guindastes foi mensurada durante 200ms. As características temporais que foram apuradas, através dos dados fornecidos pelo simulador CACO-PS, podem ser vistos na Tabela 6.2, sendo que as medidas estão apresentadas em milisegundos.

Tabela 6.2: Dados da simulação do sistema de guindastes

| | **Nr. Exec.** | **Mínimo** | **Média** | **Máximo** |
|---|---|---|---|---|
| **AngleSensorInterface** | | | | |
| Tempo Execução | 50 | 0,8671 | 0,9727 | 1,0851 |
| Latência de ativação | | 0,8855 | 1,7355 | 3,5352 |
| Jitter de ativação | | -1,9091 | -0,0282 | 1,8009 |
| **PositionSensorInterface** | | | | |
| Tempo Execução | 50 | 0,9557 | 0,9834 | 1,0785 |
| Latência de ativação | | 1,9021 | 2,7189 | 4,5468 |
| Jitter de ativação | | -1,9099 | -0,0306 | 1,7714 |
| **Controller** | | | | |
| Tempo Execução | 20 | 3,3359 | 3,5846 | 3,6815 |
| Latência de ativação | | 0,8718 | 1,9498 | 3,0859 |
| Jitter de ativação | | -2,9573 | -0,1944 | 2,0797 |
| **Diagnoser** | | | | |
| Tempo Execução | 4 | 0,9488 | 0,9515 | 0,9563 |
| Latência de ativação | | 8,8709 | 13,3898 | 19,1852 |
| Jitter de ativação | | -10,3006 | -3,4381 | 7,7337 |
| **EDFScheduler** | | | | |
| Tempo Execução | 148 | 0,6563 | 0,8729 | 1,0183 |

Os tempos para as quatro tarefas periódicas apresentados na Tabela 6.2, seguem a mesma semântica apresentada no estudo de caso do controle de movimentação da cadeira de rodas. O tempo de execução das tarefas representa o tempo gasto pelo escalonador mais o tempo gasto para executar a tarefa. A latência de ativação representa a diferença entre o instante de tempo esperado e o instante de tempo real de ativação da tarefa. Por fim o *jitter* representa a diferença da latência de ativação entre duas execuções consecutivas das tarefas.

Com relação ao escalonador, este experimento também utilizou a classe *EDFScheduler*. Como o processador FemtoJava está rodando a 15 MHz, o WCET do algoritmo foi de 1,0183 ms. Os valores de latência e *jitter* de ativação das tarefas seguem o mesmo comportamento apresentado no estudo de caso do controle de movimento da cadeira de rodas, ou seja, como a política de escalonamento prioriza a execução de tarefas com base no *deadline* mais próximo, a diferença entre os valores médios, máximos e mínimos possui uma variação relativamente alta.

# 7 CONCLUSÕES E TRABALHOS FUTUROS

O projeto de sistemas tempo-real embarcados vem se tornando cada vez mais complexo devido ao número crescente de funcionalidades incorporadas nestes sistemas. O texto desta dissertação mostra que o uso de orientação a objetos no domínio de sistemas tempo-real embarcados apresenta-se como uma alternativa interessante para o problema do gerenciamento do aumento da complexidade do seu projeto. Outro fator muito importante para no projeto de sistemas embarcados é a possibilidade de reuso permitindo uma redução no tempo de projeto. A reutilização é uma das características chave da orientação a objetos.

Com o *framework* proposto nesta dissertação e as alterações feitas na ferramenta SASHIMI, consegue-se trazer a orientação a objetos para o domínio de sistemas tempo-real embarcados, assim como permite a utilização de uma biblioteca baseada na RTSJ que possibilita a expressão em alto nível das características temporais de uma aplicação tempo-real. Assim as classes definidas em projetos anteriores podem ser facilmente reutilizadas em novos projetos diminuindo o esforço de adaptação destas classes na nova aplicação. O *framework* ainda possibilita a expressão mais clara e precisa dos requisitos temporais dos sistemas tempo-real embarcados. Suportando o determinismo temporal expresso pelas classes do *framework*, este trabalho incluiu um RTC no processador FemtoJava que permite a precisão na amostragem do tempo no sistema tempo-real embarcado.

No domínio dos computadores de propósito geral (e.g. PCs e MACs) a utilização de orientação a objetos é bastante sólida e o projeto orientado a objetos para o software destes sistemas foi bastante discutida por inúmeros autores. Para a modelagem de sistemas a comunidade de desenvolvedores de software tomou como padrão a utilização da UML para expressar os elementos do modelo do sistema. Seguindo esta tendência a comunidade de tempo-real também começa a utilizar a UML como padrão para modelagem de sistemas de tempo-real como visto em (DOUGLASS, 2000), (BALARIN *et al.*, 2003), (WEHRMEISTER, 2004), através da utilização do perfil para escalonabilidade, performance e tempo (OMG, 2002).

A utilização da RT-UML no domínio de sistemas embarcados também é abordada neste trabalho, graças ao mapeamento dos elementos no modelo RT-UML do sistema para os elementos disponíveis no *framework*. Este mapeamento foi validado através dos estudos de caso apresentados no capítulo 6.

Com relação às adaptações feitas na ferramenta CAD SASHIMI, a estrutura de representação das classes da aplicação, proposta neste trabalho, facilita a representação das classes que serão sintetizadas no sistema tempo-real embarcado. Com esta estrutura a inclusão de novas funcionalidades na síntese das aplicações embarcadas é facilitada. Grande parte das informações das classes está armazenada nesta estrutura. Se comparada com a estrutura antiga que representava os *bytecodes* como eles aparecem no *classfile* Java, a busca destas informações durante o processo de síntese se torna mais

simplificada, facilitando a inclusão de novas funcionalidades na síntese do sistema tempo-real embarcado.

A síntese dos objetos que são alocados estaticamente constitui em uma representação menos custosa em termos de memória total necessária para o armazenamento dos objetos em memória RAM, se comparado com a alocação dinâmica proposta na especificação da JVM (LINDHOLM; YELLIN, 1997), uma vez que as tabelas de informações das classes não são utilizadas. Porém isto implica em uma diminuição de flexibilidade para a alocação de objetos dentro de outros objetos, pois como os atributos que armazenam as referências são estáticos na classe, a atribuição de uma nova referência neste atributo estático afetará todos os objetos daquela classe.

Para finalizar as conclusões deste trabalho são feitas algumas considerações a respeito de trabalhos que podem ser feitos a fim de continuar o trabalho desta dissertação:

- Extensão do *framework* para incluir outras construções disponíveis na RTSJ, assim outras funcionalidades tempo-real podem ser utilizadas no domínio dos sistemas tempo-real embarcados;
- Criação de uma ferramenta que permita a geração automática de código utilizando o *framework* proposto, levando em consideração as anotações das classes utilizando o perfil de escalonabilidade, performance e tempo;
- Aumentar a quantidade de *threads* que podem executar simultaneamente no processador FemtoJava;
- Inclusão do suporte para a alocação dinâmica de objetos na síntese da ferramenta SASHIMI, assim os problemas advindos da alocação estática de objetos são resolvidos;
- Inclusão de suporte para a avaliação de toda a estrutura de classes na instrução *instanceof* do processador FemtoJava;
- Aumentar a quantidade de temporizadores disponíveis no FemtoJava. Como sugestão, poderia ser criado um mecanismo que inclua o RTC e compare o tempo do RTC com o tempo de *timeout* em registradores e, caso o tempo do RTC seja maior, este *timeout* seja sinalizado através uma interrupção.

# REFERÊNCIAS


ARTISAN SOFTWARE. **Real-Time Studio.** 2004. Disponível em: <http://www.artisansw.com/products/professional_overview.asp>. Acesso em: 16 nov. 2004.

AXELSSON, J. Real-World Modeling in UML. In: INTERNATIONAL CONFERENCE ON SOFTWARE AND SYSTEMS ENGINEERING AND THEIR APPLICATIONS, 13., Paris, France, 2000. **Proceedings…** [S.l.:s.n.], 2000.

BALARIN, F. *et al*. Synthesis of Software Programs for Embedded Control Applications**. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, New York, p. 834-849, June 1999.

BALARIN, F. *et al.* Metropolis: an Integrated Electronic System Design Environment. **IEEE Computer**, Los Alamitos, v. 36, n.4, p. 45-52, Apr. 2003.

BECK FILHO, A.C.S.; MATTOS, J.C.B.; CARRO, L.; WAGNER, F.R. CACO-PS: a General Purpose Cycle-Accurate Configurable Power Simulator. In: INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2003. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2003. p.349-354

BECKER, L. B. **Ambiente de Modelagem e Implementação de Sistema Tempo-Real Usando o Paradigma Orientado a Objetos.** 1999. 80 f. Dissertação (Mestrado em Ciências da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

BECKER, L. B. **Um Método para Abordar Todo o Ciclo de Desenvolvimento de Aplicações Tempo-Real.** 2003. 151 f. Tese (Doutorado em Ciências da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

BECKER, L. B.; HOLTZ, R.; PEREIRA, C. E. On Mapping RT-UML Specifications to RT-Java API: Bridging the Gap. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 5., 2002, Washington. **Proceedings…** Los Alamitos: IEEE Computer Society, 2002. p. 348-355

BJORKLUND, D.; LILIUS, J. From UML Behavioral Descriptions to Efficient Synthesizable VHDL. In: IEEE NORCHIP CONFERENCE, 20., 2002, Copenhagen, Dinamarca. **Proceedings…** [S.l.:s.n.], 2002.

BOLLELLA, G.; GOSLING, J.; BROSGOL, B. **The Real-Time Specification for Java**. 2001. Disponível em: <http://www.rtj.org/rtsj-V1.0.pdf>. Acesso em: 19 mar. 2004.

BOOCH, G. **Object-Oriented Analysis and Design with Applications.** Massachusetts: Addison-Wesley, 1994.

BOOCH, G; RUMBAUGH, J; JACOBSON, I. **The Unified Modeling Language User Guide**. Reading, Massachusetts: Addison-Wesley, 1999. 482p.

BURNS, A.; WELLING, A. **Real-Time Systems and Programming Languages.** 2nd ed. Massachusetts: Addison-Wesley, 1996.

BUTTAZZO, G. C. Rate Monotonic vs. EDF: Judgement Day. In: **Embedded Software.** [S.l.]: Kluwer Academic Publishers, 2003. p. 67-83. (Lecture Notes in Computer Science, v. 2855).

CARRO, L.; WAGNER, F. R. Sistemas Computacionais Embarcados. In: JORNADAS DE ATUALIZAÇÃO EM INFORMÁTICA, 22., 2003, Campinas. **Livro Texto**. Campinas: SBC, 2003. p. 45-94.

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

CORSARO, A.; SCHMIDT, D. C. The Design and Performance of the jRate Real-Time Java Implementation. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED OBJECTS AND APPLICATIONS, 2002. **Proceedings…** London: Springer-Verlang, 2002.

CORSARO, A. *et al.* Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Disciplines in Open Distributed Real-Time Systems. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED OBJECTS AND APPLICATIONS, 3., 2001. **Proceedings…** [S.l.]: IEEE Computer Society Press, 2001.

COPSTEIN, B. **SIMOO:** Plataforma Orienta a Objetos para Simulação Discreta Multi-Paradigma. Tese (Doutorado em Ciência da Computação) – Instituto de Infromática, UFRGS, Porto Alegre, 1997.

DOUGLASS, B. P. **Real-time UML** second edition: developing efficient objects for embedded systems. Boston, USA: Addison-Wesley, 2000.

GANSSLE, J.; BARR, M. **Embedded Systems Dictionary.** [S.l.]: CMP Books, 2003.

GREFF, A.; RHOD, E.; OLIVEIRA, M.; BECKER, L. B. **Automação de uma Cadeira de Rodas:** Levantamento Preliminar de Funcionalidades. 2004. Disponível em: <http://www.inf.ufrgs.br/~lse/studies/docs/LevantamentoFuncioanlidades.rtf>. Acesso em: 10 nov. 2004.

HABERMANN, A. N.; FLON, Lawrence; COOPRIDE, Lee. Modularization and Hierarchy in a Family of Operation Systems. **Communications of the ACM**, New York, v.19, n.5, p. 266-272, 1976.

HALANG, W.; STOYENKO, A. **Constructing Predictable Real-Time Systems.** Norwell, USA: Kluwer Academic Publishers, 1991.

IEEE INTERNACIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 7., 2004, Vienna, Austria. **Proceedings…** Los Alamitos, Califórnia: IEEE Computer Society, 2004.

I-LOGIX. **Rhapsody for Software Engineering.** 2005. Disponível em: < http://www.ilogix.com/rhapsody/software/software.cfm >. Acesso em: 05 jan. 2005.

ITO, S A. **Projeto de Aplicações Específicas com Microcontroladores Java Dedicados.** 2000. 84 f. Dissertação (Mestrado em Ciências da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

ITO, S. A.; CARRO, L.; JACOBI, R. P. Making Java Work for Microcontroller Applications. **IEEE Design & Test of Computers**, New York, v.18, n.5, p. 100-110, 2001.

JC – J CONSORTIUM. **Real-Time Core Extensions**. 2000. Disponível em: < http://www.opengroup.org/rtforum/rt_java/uploads/40/3417/rtce.1.0.14_May_03.pdf >. Acesso em: 01 dez. 2004.

JONG, G. de. A UML-based Design Methodology for Real-Time and Embedded Systems. In: DESIGN, AUTOMATION, AND TEST IN EUROPE, 2002, Paris. **Proceedings…** Washington: IEEE Computer Society, 2002.

LAPLANTE, P. A. **Real-Time Systems Design and Analysis:** An Engineer's Handbook. 2nd ed. Los Alamitos, Califórnia: IEEE Computer Society, 1997.

LAVAZZA, L.; QUARONI, G.; VENTURELLI, M. Combining UML and formal notations for modeling real-time systems. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE, 8., 2001, Vienna. **Proceedings...** New York: ACM Press, 2001. p. 196-206.

LINDHOLM, T.; YELLIN, F. **The Java Virtual Machine Specification.** [S.l.]: Addison-Wesley, 1997.

LIU, C. L.; LAYAND, J.W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. **Journal of the Association for Computer Machinery**, New York, v.20, n.1, p. 46-61, 1973.

LSE LABORATÓRIO DE SISTEMAS EMBARCADOS. **Projeto de Sistemas Eletrônicos Embarcados Baseados em Plataformas**. 2003. Disponível em: < http://www.inf.ufrgs.br/~lse/pag_projeto.php?cod_projeto=1 >. Acesso em: 10 nov. 2004.

MATTOS, J. C. B.; CARRO, L. **SASHIMI v0.8b:** Manual do Usuário. 2003. Disponível em: < ftp://ftp.inf.ufrgs.br/pub/sashimi/usrman0.8.pdf >. Acesso em: 01 jul. 2004.

MARTIN, J.; ODELL, J. **Análise e projetos orientados a objetos.** São Paulo: Makron Books, 1995.

MCUMBER, W. E.; CHEN, B. H.C. UML-Based Analysis of Embedded Systems Using a Mapping to VHDL. In: IEEE INTERNATIONAL SYMPOSIUM ON HIGH-ASSURANCE SYSTEMS ENGINEERING, 4., 1999, Washington, USA. **Proceedings…** Washington: IEEE Computer Society, 1999. p. 56-53

MOSER, E.; NEBEL, W. Case Study: System Model of Crane and Embedded Control. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 1999, Munique, Alemanha. **Proceedings…** [S.l.]: IEEE Computer Society, 1999.

NEBEL, W.; OPPENHEIMER, F.; SCHUMACHER, G. Object-Oriented Specification and Design of Embedded Hard Real-Time Systems. In: ASHENDEN P. J.; SEEPOLD, R.; MERMET, J. P. (Ed.). **System-on-Chip Methodologies and Design Languages**. [S.l.]: Kluwer Academic Publishers, 2001. p. 285-296.

NEBEL, W.; SCHUMACHER, G. Object-Oriented Hardware Modeling -- Where to Apply and What are the Objects ? In: EURO-DAC; EURO-VHDL, 1996, Geneva, Suíça. **Proceedings…** Los Alamitos: IEEE Computer Society, 1996. p. 428-433.

NETT, E.; GERGELEIT, M.; MOCK, M. Enhancing OO Middleware to become Time-Aware. **Real-Time Systems**, Norwell, v.20, n.2, p. 211-228, mar. 2001.

NIST – NATIONAL INSTITUTE OF STANDADS AND TECHNOLOGY. **Requirements for Real-Time Extension for Java Platform**. 1999. Disponível em: < http://www.nist.gov/itl/div897/ctg/real-time/rtj-final-draft.pdf >. Acesso em: 01 dec. 2004.

O'CONNOR, J. M.; TREMBLAY, M. picoJava-I: The Java virtual machine in hardware. **IEEE Micro**, Los Alamitos, v.17, n.2, p. 45-53, Mar 1997.

OMG – OBJECT MANAGEMENT GROUP. **UML Profile for Schedulability, Performance and Time Specification.** 2002. Disponível em: <http://www.omg.org/cgi-bin/doc?ptc/02-03-03>. Acesso em: 15 mar. 2004.

PEREIRA, C. E. Real-Time Active Objects in C++/Real-Time UNIX. In: ACM SIGPLAN WORKSHOP ON LANGUAGES, COMPILER, AND TOOL SUPPORT FOR REAL-TIME SYSTEMS, 1994, Orlando, EUA. **Proceedings…** [S.l.:s.n.], 1994.

ROSA JUNIOR, Leomar. **Implementação de Multitarefa sobre a Arquitetura Java Embarcada FemtoJava.** 2004. 80 f. Dissertação (Mestrado em Ciências da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

RUMBAUGH, J. *et al*. **Modelagem e projetos baseados em objetos**. Rio de Janeiro: Campus, 1994.

SCHOEBERL, M. Restrictions of Java for Embedded Real-Time Systems. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 7., 2004, Vienna, Austria. **Proceedings…** Los Alamitos, Califórnia: IEEE Computer Society, 2004a. p. 93-100.

SCHOEBERL, M. Design Rationale of a Processor Architecture for Predictable Real-Time Execution of Java Programs. In: INTERNATIONAL CONFERENCE ON REAL-TIME AND EMBEDDED COMPUTING SYTEMS AND APPLICATIONS, 10., 2004, Gothenburg, Suécia. **Proceedings…** [S.l.]: Springer-Verlang, 2004b.

SCHOEBERL, M. Java Technologies in an FPGA. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND ITS APPLICATIONS, 2004, Antwerp, Bélgica. **Proceedings…** [S.l.]: Springer-Verlang, 2004c.

SELIC, B. The emerging real-time UML Standard. **International Journal Of Computer Systems Science Engineering**, Leics, UK, v. 17, n. 2, 2002.

SIEBERT, F.; WALTER, A. Deterministic Execution of Java's Primitive Bytecode Operations. In: JAVA VIRTUAL MACHINE RESEARCH AND TECHNOLOGY SYMPOSIUM, 2001. **Proceedings…** [S.l.:s.n.], 2001.

SIEBERT, F.; WALTER, A. **JamaicaVM User Documentation**: The Virtual Machine for Real-Time and Embedded Systems. 2004. Disponível em: < http://www.aicas.com/jamaica/doc/html/index.html >. Acesso em: 07 dez. 2004.

SINHA, V. *et al*. YAML: A Tool for Hardware Design Visualization and Capture. In: INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, 13., 2000, Madrid, Espanha. **Proceedings…** Los Alamitos: IEEE Computer Society, 2000. p. 9-17.

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

STANKOVIC, J. A. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. **IEEE Computer**, New York, p. 10-19, Oct. 1988.

STOEL, C.; KARRFALT, J. VIOOL for Hardware/Software Codesign. In: INTERNATIONAL SYMPOSIUM AND WORKSHOP ON SYSTEMS ENGINEERING OF COMPUTER BASED SYSTEMS, 1995. **Proceedings...** [S.l.:s.n.], 1995. p. 330-340.

SUN MICROSYSTEMS. **Java 2 Platform Standard Edition Development Kit 5.0.** 2004a. Disponível em: < http://java.sun.com/j2se/1.5.0/README.html >. Acesso em: 02 dez. 2004.

SUN MICROSYSTEMS. **Java 2 Platform, Standard Edition.** 1995. Disponível em: < http://java.sun.com/j2se/index.jsp >. Acesso em: 02 dez. 2004.

SUN MICROSYSTEMS. **picoJava II Programmer's Reference.** [S.l.], 1999a.

SUN MICROSYSTEMS. **picoJava II Microarchitecture Guide.** [S.l.], 1999b.

SUN MICROSYSTEMS. **Java 2 Technology Overview.** 2004b. Disponível em: < http://java.sun.com/overview.html >. Acesso em: 02 dez. 2004.

SUN MICROSYSTEMS. **Java 2 Platform, Micro Edition.** 2004c. Disponível em: < http://java.sun.com/j2me/index.jsp >. Acesso em: 02 dez. 2004.

SUN MICROSYSTEMS. **The Real-Time Java Platform.** White paper. 2004d. Disponível em: <http://research.sun.com/projects/mackinac/mackinac_whitepaper.pdf>. Acesso em: 02 dez. 2004.

WEHRMEISTER, M. A. **Utilização de Orientação a Objetos no Projeto de Sistemas Embarcados Tempo-Real.** 2004. 51p. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

WOLF, W. H. **Computers as Components**: Principles of Embeded Computing System Design. San Francisco: Morgan Kaufmann Publishers, 2000.

# ANEXO A NOVAS INSTRUÇÕES SUPORTADAS

Neste anexo são apresentadas as novas instruções suportadas no processador FemtoJava. Na tabela abaixo se encontram respectivamente o nome da instrução, o número da instrução em binário, e as palavras de controle em cada ciclo da instrução juntamente com um descritivo do comportamento dos sinais ao final do ciclo.

| aload | 00011001b |
|---|---|
| 0010100000000000000000000000001 | immed = ROM[pc]; pc++; RAM[sp] = b; |
| 00000000010001100010000001000100 | b = a; mar = vars - immed; |
| 00000000010000010001001001001 10 | a = RAM[mar]; mar = sp--; |
| 0000000000000000000000000000000 | |

| aload_0 | 00101010b |
|---|---|
| 0010000000000000000000000000000 | RAM[sp] = b; |
| 0000000000000000001000001000100 | b = a; mar = vars - 0; |
| 0000000001000001000100100100110 | a = RAM[mar]; mar = sp--; |
| 0000000000000000000000000000000 | |

| aload_1 | 00101011b |
|---|---|
| 0010000000000000000000000000000 | RAM[sp] = b; |
| 0000000001000001001000001000100 | b = a; mar = vars - 1; |
| 0000000001000001000100100100110 | a = RAM[mar]; mar = sp--; |
| 0000000000000000000000000000000 | |

| aload_2 | 00101100b |
|---|---|
| 0010000000000000000000000000000 | RAM[sp] = b; |
| 0000000001000010001000001000100 | b = a; mar = vars - 2; |
| 0000000001000001000100100100110 | a = RAM[mar]; mar = sp--; |
| 0000000000000000000000000000000 | |

| aload_3 | 00101101b |
|---|---|
| 0010000000000000000000000000000 | RAM[sp] = b; |
| 0000000001000011001000001000100 | b = a; mar = vars - 3; |
| 0000000001000001000100100100110 | a = RAM[mar]; mar = sp--; |
| 0000000000000000000000000000000 | |

| aaload | 00110010b |
|---|---|
| 0000000000000001000101001000110 | b = a + b; mar = sp++; |
| 0000000000000000000010001100000 | a = b; b = MEM[mar]; |
| 0000000000000001001100000000100 | mar = a+1; |
| 0000000000000000000000100100000 | a = MEM[mar]; |
| 0000000000000000001000000000100 | mar = sp; |
| 0000000000000000000000000000000 | |

| astore | 00111010b |
|---|---|
| 0000100000000000000000001100001 | immed = ROM[pc]; pc++; b = a; a = b; |
| 0000000001000110001000000000100 | mar = vars - immed; |
| 0010000000000000000000000000000 | RAM[mar] = b; |
| 0000000000000001000100000000110 | mar = sp++; |
| 0000000000000000000010001000000 | b = RAM[sp]; |
| 0000000000000000000000000000000 | |

| astore_0 | 01001011b |
|---|---|
| 0000000000000000001000001100100 | b = a; a = b; mar = vars - 0; |
| 0010000000000000000000000000000 | RAM[mar] = b; |
| 0000000000000001000100000000110 | mar = sp++; |
| 0000000000000000000010001000000 | b = RAM[sp]; |
| 0000000000000000000000000000000 | |

| astore_1 | 01001100b |
|---|---|
| 0000000001000001001000001100100 | b = a; a = b; mar = vars - 1; |
| 0010000000000000000000000000000 | RAM[mar] = b; |
| 0000000000000001000100000000110 | mar = sp++; |
| 0000000000000000000010001000000 | b = RAM[sp]; |
| 0000000000000000000000000000000 | |

| astore_2 | 01001101b |
|---|---|
| 0000000001000010001000001100100 | b = a; a = b; mar = vars - 2; |
| 0010000000000000000000000000000 | RAM[mar] = b; |
| 0000000000000001000100000000110 | mar = sp++; |
| 0000000000000000000010001000000 | b = RAM[sp]; |
| 0000000000000000000000000000000 | |

| astore_3 | 01001110b |
|---|---|
| 0000000001000011001000001100100 | b = a; a = b; mar = vars - 3; |
| 0010000000000000000000000000000 | RAM[mar] = b; |
| 0000000000000001000100000000110 | mar = sp++; |
| 0000000000000000000010001000000 | b = RAM[sp]; |
| 0000000000000000000000000000000 | |

| aastore | 01010011b |
|---|---|
| 0000000000000000000000001100000 | b = a; a = b; |
| 0010000000000000000000000000000 | RAM[mar] = b; |
| 0000000000000001000100000000110 | mar = sp++; |
| 0000000000000000000010001000000 | b = RAM[mar]; |
| 0000000001000001000101001000110 | b = b + a; mar = sp--; |
| 0000000000000000000010001100000 | a = b; b = RAM[mar]; |
| 0000000000000001001100000000100 | mar = a + 1; |
| 0010000000000000000000000000000 | RAM[mar] = b; |
| 0000000000000010000100000000110 | mar = sp + 2; |
| 0000000000000000000000100100000 | a = RAM[ram]; |
| 0000000000000001000100000000110 | mar = sp++; |
| 0000000000000000000010001000000 | b = RAM[ram]; |
| 0000000000000000000000000000000 | |

| areturn | 10110000b |
|---|---|
| 0000000000000000000000001000100 | b = a; mar = frm; |
| 0000000000000000000000100100000 | a = RAM[frm]; |
| 0000000001110000111000110100101 | pc = a; a = mar = vars; |
| 0010000000000000000000000000000 | RAM[vars] = b; |
| 0000000001000001000000001000110 | b = a; mar = sp = frm - 1; |
| 0000000000000000000000100100000 | a = RAM[frm - 1]; |
| 0000000001000001000100000000110 | mar = sp--; |
| 0000000000000000001100100110000 | frm = a; a = RAM[frm - 2]; |
| 0000000000000000001100000101000 | vars = a; a = b; |
| 0000000000000000001100000000110 | mar = sp = a; |
| 0000000000000000000010001000000 | b = RAM[sp]; |
| 0000000000000001000100000000110 | mar = sp++; |
| 0000000000000000000010001100000 | b = RAM[sp]; a = b; |
| 0000000000000000000000000000000 | |

| if_acmpne | 10100110b |
|---|---|
| 0010000010000001000101001000111 | immed = ROM[pc]; mar = sp++; b = a - b; pc++; |
| 0010000000000000000010001100000 | immed = ROM[pc]; a = b; b = MEM[mar]; |
| 0000000000000001000100000000110 | mar = sp++; |
| 0000000000010000000010001100001 | a = b; b = MEM[mar]; ?stk_top != b : pc++: pc = pc + immed; |
| 0000000000000000000000000000000 | |

| ifnull | 11000110b |
|---|---|
| 0001000000000001000100000000111 | immed = ROM[pc]; mar = sp++; pc++; |
| 0001000000000000000000000000000 | immed = ROM[pc]; |
| 0000000000001000000010001100001 | a = b; b = MEM[mar]; ?stk_top = 0 : pc++: pc = pc + immed; |
| 0000000000000000000000000000000 | |

| ifnonnull | 11000111b |
|---|---|
| 0001000000000001000100000000111 | immed = ROM[pc]; mar = sp++; pc++; |
| 0001000000000000000000000000000 | immed = ROM[pc]; |
| 0000000000010000000010001100001 | a = b; b = MEM[mar]; ?stk_top != 0 : pc++: pc = pc + immed; |
| 0000000000000000000000000000000 | |

| aconst_null | 00000001b |
|---|---|
| 0010000000000000000000000000000 | RAM[sp] = b; |
| 0000000000000000000011001000000 | b = iconst_0; |
| 0000000001000001000100001100110 | a = b; b = a; mar = sp--; |
| 0000000000000000000000000000000 | |

| getfield | 10110100 |
|---|---|
| 0011000000000000000000000000001 | immed = ROM[pc]; pc++; RAM[sp] = b; |
| 0001000000000000000000000000001 | immed = ROM[pc]; pc++; |
| 0000000000000110000011001000000 | b = immed; |
| 0000000000000000000101001000100 | b = a + b; mar = sp; |
| 0000000000000000000010001100000 | a = b; b = MEM[mar]; |
| 0000000000000000001100000000100 | mar = a + 0; |
| 0000000000000000000100100100100 | a = RAM[mar]; mar = sp; |
| 0000000000000000000000000000000 | |

| putfield | 10110101 |
|---|---|
| 0000000000000000000000001100000 | b = a; a = b; |
| 0011000000000000000000000000001 | immed = ROM[pc]; pc++; RAM[sp] = b; |
| 0001000000000000000000000000001 | immed = ROM[pc]; pc++; |
| 0000000000000110000011001000000 | b = immed; |
| 0000000000000000000101001000100 | b = a + b; mar = sp; |
| 0000000000000000000010001100000 | a = b; b = MEM[mar]; |
| 0000000000000000001100000000100 | mar = a + 0; |
| 0010000000000000000000000000000 | RAM[mar] = b; |
| 0000000000000001000100000000110 | mar = sp++; |
| 0000000000000000000000100100000 | a = RAM[mar]; |
| 0000000000000001000100000000110 | mar = sp++; |
| 0000000000000000000010001000000 | b = RAM[mar]; |
| 0000000000000000000000000000000 | |

| invokevirtual | 10110110b |
|---|---|
| 0111000000000000000000000000001 | immed = ROM[pc]; pc++; RAM[sp] = b; |
| 0101000000000000000000000000001 | immed = ROM[pc]; pc++ |
| 0000000001111001010100011100111 | b = a; a = pc; pc = immed; mar = sp--; |
| 0010100001000001000100000000011 | immed = ROM[pc]; pc++; RAM[sp] = b; sp--; |
| 0000000001000110000100001000110 | b = a; mar = sp = sp - immed; |
| 0010000000000000000000110100000 | RAM[sp] = b; a = frm; |
| 0000000001000001000100001000110 | b = a; mar = sp--; |
| 0010000000000000001000110100000 | a = vars; RAM[sp] = b; |
| 0000000001000001000100001000110 | b = a; mar = sp--; |
| 0010000000000001000010000010000 | frm = sp + 2; RAM[sp] = b; |
| 0000000001000001000100000000110 | mar = sp--; |
| 0000100000000011000000000001001 | immed = ROM[pc]; pc++; vars = frm + immed; |
| 0000000000000110001000000001000 | vars = vars + immed; |
| 0000000000000000000000000000000 | |

| invokespecial | 10110111b |
| --- | --- |
| 0111000000000000000000000000001 | immed = ROM[pc]; pc++; RAM[sp] = b; |
| 0101000000000000000000000000001 | immed = ROM[pc]; pc++ |
| 0000000001111001010100011100111 | b = a; a = pc; pc = immed; mar = sp--; |
| 0010100001000001000100000000011 | immed = ROM[pc]; pc++; RAM[sp] = b; sp--; |
| 0000000001000110000100001000110 | b = a; mar = sp = sp - immed; |
| 0010000000000000000000110100000 | RAM[sp] = b; a = frm; |
| 0000000001000001000100001000110 | b = a; mar = sp--; |
| 0010000000000000001000110100000 | a = vars; RAM[sp] = b; |
| 0000000001000001000100001000110 | b = a; mar = sp--; |
| 0010000000000010000100000010000 | frm = sp + 2; RAM[sp] = b; |
| 0000000001000001000100000000110 | mar = sp--; |
| 0000100000000110000000000001001 | immed = ROM[pc]; pc++; vars = frm + immed; |
| 0000000000000110001000000001000 | vars = vars + immed; |
| 0000000000000000000000000000000 | |

| instanceof | 11000001 |
| --- | --- |
| 0011000000000000000000000000001 | immed = ROM[pc]; pc++; RAM[sp] = b; |
| 0001000000000000000000000000001 | immed = ROM[pc]; pc++; |
| 0000000000000110000011001000000 | b = immed; |
| 0000000000000000001100000000100 | mar = a + 0; |
| 0000000000000000000010001100000 | a = b; b = MEM[mar]; |
| 0000011100000001000101001000100 | b = a XOR b; mar = sp |
| 0000000000000000000010001100000 | a = b; b = MEM[mar]; |
| 0000000000000000000000000000000 | |

| checkcast | 11000000 |
| --- | --- |
| 0001000000000000000000000000001 | immed = ROM[pc]; pc++; |
| 0001000000000000000000000000001 | immed = ROM[pc]; pc++; |
| 0000000000000000000000000000000 | |