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

Desenvolvimento de Sistemas Tempo-Real usando Orientação a Objetos: Estudo sobre o Mapeamento de Especificações para Linguagens de Programação

por

RUDY HAMILTON HÖLTZ

Trabalho de conclusão submetido à avaliação,
como requisito parcial para obtenção
do grau de Mestre em Informática

Prof. Dr. Carlos Eduardo Pereira
Orientador

Prof. Dr. João César Netto
Co-Orientador

Porto Alegre, março de 2002.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Höltz, Rudy Hamilton
Desenvolvimento de Sistemas Tempo-Real usando Orientação a Objetos: Estudo sobre Mapeamento de Especificações para Linguagens de Programação / por Rudy Hamilton Höltz. – Porto Alegre: PPGC da UFRGS, 2002.
75p.:il.

Trabalho de Conclusão (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós Graduação em Computação, Porto Alegre, BR, 2002. Orientador: Pereira, Carlos Eduardo; Co-orientador: Netto, João César.

1. Objetos Tempo-Real. 2. RT-UML. 3. RT-Java 4. Geração de Código. I. Pereira, Carlos Eduardo. II. Título.

Dedico este trabalho para
minha mulher Jussara Elisa Tolosa Höltz
e para meu pai Olnei Milton Höltz

Agradecimentos

Consta numa parábola religiosa, que um homem ao chegar ao céu e analisar seus passos ao longo da vida percebeu nas marcas deixadas no chão que foi acompanhado por várias pessoas em sua existência. Mesmo nos momentos de solidão, as marcas dos passos de Deus estavam presentes ao seu lado, mas ele percebeu que nos piores momentos de sua vida havia caminhado só e questionou o Senhor por tê-lo abandonado. Deus respondeu que nunca o abandonara e que as marcas solitárias eram Suas carregando este homem nos ombros. A confecção de um trabalho de conclusão de mestrado envolve um trabalho significativo, mas certamente não seria possível sem a contribuição maior ou menor de diversas pessoas, entre orientadores, professores, funcionários, colegas e amigos.

Gostaria de agradecer em particular ao amigo Leandro Buss Becker por sua inestimável contribuição através de sugestões e revisões.

Sumário

Lista de Abreviaturas e Siglas

ACE	*Washington University's The Adaptive Communication Environment;*
AIE	*AsynchronouslyInterruptedException*;
AO/C++	*Active Object C++;*
API	*Application Programming Interface;*
ASP	*Microsoft's Active Server Pages;*
ATC	*Asynchronous Transfer of Control;*
ATM	*Asynchronous Transfer Mode;*
CAN	*Controller Area Network;*
CASE	*Computer Aided Software Engineering;*
COMET	*Concurrent Object Modeling and Architectural Method*;
CORBA	*Common Object Request Broker Architecture;*
COTS	*Commercial Off-The-Shelf;*
DCOM	*Distributed Component Object Model;*
EDF	*Earliest Deadline First Scheduling Algorithm;*
GC	*Garbage Collector;*
GIOP	*General Inter-ORB Protocol;*
HTTP	*Hypertext Transfer Protocol;*
IBM	*International Business Machine;*
IDL	*Interface Definition Language;*
IEEE	*Institute of Electrical and Electronics Engineers;*
IIOP	*CORBA's Internet Inter-ORB Protocol;*
I/O	*Input/Output;*
ISBN	*International Standard Book Number;*
ISORC	*IEEE International Symposium on Object-Oriented Real-Time Distributed Computing;*
J2EE	*Java 2 Platform, Enterprise Edition;*
J2ME	*Java 2 Platform, Micro Edition;*
JAR	*Java Archiver;*
JCP	*Java Community Process;*
JDK	*Java Development Kit;*
JRMI	*Java Remote Method Invocation;*
JSP	*Java Server Pages;*

JSR	*Java Specification Requests;*
JVM	*Java Virtual Machine;*
NIST	*U.S. National Institute of Standards and Technologies;*
NT	*Microsoft's New Technology;*
OCL	*Object Constraint Language;*
OMG	*Object Management Group;*
ORB	*Object Request Broker;*
OS	*Operating System;*
PERC	*Portable Executive for Reliable Control;*
PPGC	Programa de Pós-Graduação em Computação;
QoS	*Quality of Service;*
RFP	*Request For Proposal;*
RMA	*Rate Monotonic Analysis;*
RMI	*Remote Method Invocation;*
ROOM	*ObjecTime's Real-Time Object-Oriented Modeling;*
ROPES	*Rapid Object-Oriented Process for Embedded Systems;*
RT	*Real-Time;*
RTSJ	*The Real-Time Specification for Java;*
RT-UML	*Real-Time Unified Modeling Language;*
SCADA	*Supervisory Control and Data Acquisition;*
SIMOO	Plataforma Orientada a Objetos para Simulação Discreta Multi-Paradigma;
TAO	*Washington University's The ACE ORB;*
TMO	*Time-triggered Message-triggered Object;*
UCP	Unidade Central de Processamento;
UFRGS	Universidade Federal do Rio Grande do Sul;
UML	*Unified Modeling Language;*
UNC	*Universal Naming Convention;*
UPS	*Uninterruptible Power System;*
UMLSPT	*UML Profile for Scheduling, Performance, and Time;*
WWW	*World Wide Web;*

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho realiza um estudo sobre a criação de sistemas tempo-real usando orientação a objetos, com enfoque no mapeamento de especificações para linguagens de programação. O paradigma de orientação a objetos tem sido usado nas diferentes fases relacionadas com o desenvolvimento de sistemas tempo-real, variando desde a modelagem até o ambiente de programação e execução, mas atualmente estas iniciativas ainda focam etapas isoladas do ciclo de desenvolvimento. O objetivo deste trabalho é o de preencher esta lacuna, propondo um mapeamento entre uma metodologia ou ferramenta de análise e projeto de sistemas tempo-real orientados a objetos e uma linguagem ou ambiente de desenvolvimento baseado no paradigma de orientação a objetos que possua suporte para atender às restrições temporais especificadas.

O mapeamento proposto foi desenvolvido utilizando estudos de caso clássicos em aplicações tempo-real que foram baseados em dois recentes padrões. O primeiro é o emergente padrão Real-Time UML, que visa realizar a especificação de requisitos temporais utilizando diagramas UML com extensões que os representem. O outro padrão é o *Real-Time Specification for Java*, que consiste de uma interface de programação (API) para desenvolvimento de aplicações tempo-real com a linguagem Java. O relacionamento entre *stereotypes* e *tags* usados para representar restrições temporais em diagramas UML e o código Java correspondente é explicado e um sumário da estratégia de mapeamento é discutido.

Este trabalho foi financiado pela empresa Altus Sistemas de Informática.

Palavras-Chave: Objetos Tempo-Real, RT-UML, RT-Java e Geração de Código.

TITLE: “REAL-TIME OBJECT-ORIENTED SYSTEMS DEVELOPMENT: STUDY ABOUT TIME CONSTRAINTS TO PROGRAMMING LANGUAGE MAPPING”.

Abstract

This work deals with object-oriented methodologies for developing real-time systems, focusing on strategies for mapping specifications to programming constructs. The object-oriented paradigm has been extensively applied to the development of real-time systems, however most of existing approaches focus only on a single development phase. The present work proposes an approach for fill the existing gap between recent proposals for real-time object-oriented specifications and the target object-oriented real-time programming or run time environment which should meet the specified timing requirements. A clear link connecting the modeled real-time constraints and the programming entities that realize these specifications is presented.

Some classical real-time case studies were used to develop the proposed mapping approach that is based on two recent standards of real-time object-oriented computing. The first is the emerging RT-UML standard, which addresses the specification of real-time requirements and timing aspects using extended UML diagrams. The second is the Real-Time Specification for Java, consisting in an Application Programming Interface (API) for real-time object-oriented programming. Relationships between the stereotypes and tags used to decorate the UML diagrams and their code representation are explained and summarize mapping strategies are discussed.

The company Altus Sistemas de Informática sponsored this work.

Keywords: Object-Oriented Real-Time Computing, RT-UML, RT-Java and Code Generation.

1 Introdução

1.1 Sistemas Tempo-Real

Influenciada pelos avanços tecnológicos nas áreas de instrumentação, microeletrônica e informática, verifica-se a crescente demanda por sistemas computacionais em aplicações criticas. Como exemplo, é possível citar sistemas de automação industrial, usinas elétricas, eletrônica embarcada, controle aeroespacial, telecomunicações, equipamentos médicos e outras aplicações que envolvem controle físico. Além de visarem aumento de produtividade, eficiência, segurança e redução de custos, uma característica comum a todos estes sistemas computacionais é sua classificação como sistemas tempo-real: são sistemas computacionais cujo correto funcionamento não depende apenas de um processamento das informações de forma a atender requisitos funcionais, mas também dos instantes de tempo em que os valores de entrada são adquiridos, o tempo de execução dos algoritmos envolvidos e o instante de tempo em que os valores de saída são gerados [PER 2000].

A orientação a objetos é uma tecnologia consagrada no desenvolvimento sistemas computacionais de grande porte e complexidade e diversos estudos recentes [IEEE 99, 2000 e 2001] confirmam a tendência de sua utilização no desenvolvimento de sistemas tempo-real e sistemas distribuídos. No âmbito de metodologias de análise e projeto, pode-se citar extensões da linguagem *Unified Modeling Language* (UML) para tempo-real. Em tecnologia de computação distribuída, também surgiram alguns produtos com extensões para o *Distributed Component Object Model* (DCOM), mas o maior destaque cabe à versão tempo-real do *Common Object Request Broker Architecture* (CORBA). No caso de linguagens, temos Ada 95, PEARL e Real-Time Java como exemplos nos quais é possível representar diretamente conceitos e mecanismos necessários ao desenvolvimento de aplicações tempo-real.

1.2 Objetivos

Embora seja significativo tanto o emprego como o estudo do paradigma de orientação a objetos na criação de sistemas tempo-real, atualmente estas iniciativas ainda focam etapas isoladas do ciclo de desenvolvimento. Os esforços para criação de extensões tempo-real para ambientes de programação orientados a objetos não estão necessariamente alinhados com propostas de padronização de especificações para sistemas tempo-real baseados no paradigma de orientação a objetos. Por outro lado, trabalhos que abordam o ciclo completo de desenvolvimento são freqüentemente específicos para um ambiente em particular, com baixa chance de popularização.

O objetivo deste trabalho é preencher esta lacuna, propondo um mapeamento entre uma metodologia ou ferramenta de análise e projeto sistemas tempo-real orientados a objetos e uma linguagem ou ambiente de desenvolvimento baseado no paradigma de orientação a objetos que possua suporte para atender as restrições temporais especificadas. Esta abordagem assume que serão selecionadas tecnologias largamente utilizadas no desenvolvimento de sistemas tempo-real ou com reais perspectivas de aceitação pelo mercado.

Para atingir este objetivo, este trabalho envolve as seguintes etapas:

- Seleção de uma metodologia ou ferramenta de modelagem e levantamento dos requisitos temporais disponíveis; definição de um subconjunto destes requisitos que serão considerados no restante deste texto;
- Seleção de um ambiente de desenvolvimento que permita a especificação de requisitos temporais, tais como as extensões tempo-real de tecnologias *middleware* e linguagens de programação e levantamento dos requisitos temporais disponíveis;
- Proposta de mapeamento das especificações temporais selecionadas da metodologia de modelagem para representações no ambiente de programação; ou seja, proposta de tradução de cada tipo de restrição temporal no modelo para solicitação de um serviço de *middleware* ou primitiva de uma linguagem;
- Desenvolvimento do mapeamento baseado em estudos de caso clássicos em aplicações tempo-real e análise dos resultados obtidos.

1.3 Organização do Texto

O restante deste texto está organizado da seguinte forma: o capítulo 2 realiza uma breve revisão de conceitos; o capítulo 3 apresenta o estado da arte no desenvolvimento de sistemas tempo-real; o capítulo 4 apresenta uma proposta de mapeamento entre especificações e linguagens de programação num ambiente tempo-real; o capítulo 5 aborda estudos de caso onde esta proposta é testada e finalmente no capítulo 6 são resumidas as conclusões e possibilidades de trabalhos futuros.

2 Revisão de Conceitos

Neste capítulo serão caracterizados os sistemas tempo-real e suas diferentes restrições temporais, e será feita uma apresentação dos motivos que tornam a orientação a objetos interessante na implementação de sistemas tempo-real. Maiores informações sobre sistemas tempo-real podem ser encontradas no livro de Burns e Welling [BUR 96]. Uma descrição do modelo de orientação a objetos* pode ser encontrada no livro de Booch [BOO 94] e maiores informações sobre o estado da arte em orientação a objetos podem ser obtidas em [SCM 2000].

2.1 Sistemas Tempo-Real

Sistemas tempo-real são sistemas nos quais o resultado do processamento é válido apenas quando emitido dentro de um limite determinado de tempo. Numa máquina operatriz, por exemplo, as velocidades dos eixos calculadas para uma determinada posição da ferramenta de corte são válidas por milissegundos ou menos - qualquer atraso no processamento pode levar a um desvio na trajetória da ferramenta de corte, levando à usinagem de uma peça fora das especificações dimensionais. Em oposição temos os sistemas convencionais, nos quais não existem requisitos temporais - os cálculos das fundações de um edifício podem estar igualmente corretos sendo realizados em minutos ou horas. Processar dados em microssegundos não torna um sistema tempo-real, o que importa são tempos de resposta limitados e previsíveis [LAP 97].

Conforme Selic et al [SEL 94], as aplicações tempo-real podem ser classificadas em dois tipos: as aplicações *Hard Real-Time* são aquelas nas quais o não cumprimento de um único requisito temporal é considerado inaceitável, pois envolve vidas humanas. Por exemplo: estações nucleares, equipamentos médicos e controle aéreo. Nas aplicações *Soft Real-Time* a não observância de um limite de tempo é aceitável, pois as conseqüências são mais restritas ou recuperáveis através de repetição como a perda de uma ligação telefônica, a perda de algum *frame* numa videoconferência ou o atraso eventual de posicionamento de uma peça numa linha de produção. De certa forma esta definição também poderia englobar um editor de texto que precisa responder aos comandos num tempo razoável ou torna impraticável sua utilização. Assim temos também a expressão *Firm Real-Time* que corresponde a um meio-termo entre *Hard* e *Soft*, aplicações onde uma baixa probabilidade de perda de requisitos temporais pode ser admitida [LAP 97].

As aplicações de automação industrial, usinas elétricas, controle aeroespacial, eletrônica embarcada, telecomunicações e equipamentos médicos *et cetera* são exemplos de sistemas computacionais que precisam atender a requisitos temporais. A correção temporal (*timeliness*) é a capacidade de um sistema de responder a estímulos em um tempo especificado, ou seja, precisa respeitar as seguintes imposições:

- Tempo máximo de execução (*deadline*) – requisito que exige que o processamento seja concluído dentro de um limite de tempo, devido ao impacto físico da atividade do sistema controlado sobre o ambiente [STA 88]. Por exemplo, se o comando de parada de uma máquina operatriz não é executado, pode colocar em risco a vida do

* Abstração, encapsulamento, modularidade e hierarquia são os elementos que caracterizam um modelo orientado a objetos. Conceitos como tipo, concorrência e persistência também são úteis mas não essenciais [BOO 94].

metalúrgico que a está operando. Também pode ser visto como a limitação rígida do intervalo de tempo fim-a-fim entre um estímulo e a conclusão da atuação correspondente;

- Periodicidade – é outro tipo de requisito que exige que a computação seja realizada uma vez a cada período de tempo ou tenha um ciclo de execução limitado ao intervalo especificado;
- Previsibilidade – o comportamento do sistema ou o tempo de reação aos estímulos precisa estar com garantia dentro de limites conhecidos. Também conhecido como comportamento determinístico[†], ou seja, o tempo total para execução da computação precisa ser determinado matematicamente, sem incluir fatores que dependam de probabilidades. Esta necessidade de previsibilidade do atraso é conseqüência do ambiente físico que o sistema está controlando;
- Tratamento de exceção – quando o tempo máximo de execução ou o período de uma atividade cíclica não é respeitado, existe necessidade da realização de alguma ação corretiva para eliminar os efeitos da falha temporal ocorrida ou para assumir um estado seguro (*fail-safe*). Por exemplo, se o cálculo do posicionamento vertical de uma lança de oxigênio numa aciaria não puder ser executado no prazo, deve-se recolher a lança, porque uma resposta inexata porém correta é preferível a uma resposta atrasada.

Um fator de complexidade nestas aplicações é a necessidade de atender individualmente aos requisitos temporais das diversas tarefas, atividades com requisitos de tempo, ao mesmo tempo em que executa as demais atividades com um reduzido tempo médio de resposta [STA 88]. O escalonamento de processos ou *threads* precisa ser orientado aos tempos máximos de execução e não apenas às prioridades. Todas estas imposições precisam ser submetidas à rígida inspeção numa avaliação do sistema tempo-real, mas além destas existem outras condições de contorno que podem ser exigidas no contexto da aplicação. A lista de requisitos que segue foi baseada em diversas fontes, tais como [STA 93, HÖL 96, FLO 98, KAI 99 e KOP 99]:

- Variação do tempo máximo de execução – o limite de tempo pode variar conforme o estado do sistema ou é admitida uma certa taxa de execuções que não atende às restrições temporais;
- Dinamismo – as propriedades do sistema podem variar com o tempo, o que ocorre com a inclusão e exclusão de atividades periódicas. Por exemplo, num sistema de controle de tráfego aéreo, uma aeronave precisa ser verificada periodicamente tão logo entre na região de determinado aeroporto. Como conseqüência deste atributo, o sistema também deve ser adaptável para novas especificações ou expansões – não é desejável que o atendimento a todos os requisitos temporais tenha sido garantido apenas através de um projeto estático;
- Baixa latência[‡] – está relacionada com a capacidade de reação ao estímulo e equivale ao requisito que o intervalo de tempo entre um evento e o início do processamento correspondente obedeça algum limite;

[†] Um sistema é dito determinístico quando para cada possível estado e cada conjunto de entradas, um único conjunto de saídas e o próximo estado do sistema puder ser determinado. Se o tempo de resposta para cada conjunto de entradas for conhecido, o sistema também possui determinismo temporal [LAP 97].

[‡] *Latency* ou Latência no contexto de comunicação é o tempo necessário para que um sinal passe de um ponto para outro em uma rede [MIC 98].

- Dependência de recursos – uma tarefa pode ter necessidade de acesso a certos recursos além da UCP, como dispositivos de entrada e saída, estruturas de dados, arquivos e bancos de dados. Quando a alocação de recursos não considera a prioridade do requisitante ou a possibilidade de preempção[§], pode ocorrer uma inversão na ordem de execução[**] mais adequada para atender aos *deadlines*. Maiores informações sobre problemas e propostas de solução para escalonamento de sistemas tempo-real pode ser encontrado no artigo de Rodríguez [ROD 97] ou em outros artigos na mesma seção dos anais;
- Requisitos de concorrência – múltiplas tarefas precisam ser executadas, tendo acesso a recursos compartilhados garantindo que suas consistências não serão violadas. As *threads* de controle que executam as tarefas podem precisar de mecanismos de sincronismo para compartilhar estes recursos;
- Requisitos de precedência – é preciso reagir a eventos gerados de forma assíncrona ou imprevisível, mantendo controle sobre a ordem relativa de ocorrência destes eventos, provendo uma resposta para todos os estímulos externos dentro de um certo limite de tempo que considere as alterações no estado do sistema provocadas pelos acontecimentos anteriores. Este conceito está relacionado com a necessidade de ordenação na execução das tarefas;
- Restrição de *jitter*[††] ou flutuação de fase – algumas aplicações também impõem restrições em relação à variação entre atuações consecutivas ou variação do início ou fim da execução em relação à base de tempo;
- Validade dos dados – seria uma restrição para garantir a consistência de dados ao longo do tempo que torna um valor inválido após expirar o tempo máximo de atualização. É um mecanismo encontrado em alguns bancos de dados tempo-real de sistemas *Supervisory Control and Data Acquisition* (SCADA);
- Nível de importância – o atendimento ao *deadline* para uma situação de emergência é normalmente mais crítico que atender ao *deadline* de uma tarefa que executa sob condições normais de operação. Numa planta petroquímica, o comando de desligamento de uma caldeira frente a uma ameaça de superaquecimento é mais crítico que garantir o percentual correto de mistura de matérias-primas.

2.2 Orientação a Objetos

O paradigma de orientação a objetos é uma técnica de desenvolvimento de sistemas que iniciou como uma técnica de programação, evoluindo para definição de requisitos, projeto de software e indo até o *co-design* de sistemas. No nível mais baixo de granularidade objetos podem ser vistos como tipos abstratos de dados, mas conforme [SEL 94] na modelagem de grandes sistemas, é mais adequada uma visão de máquinas

---

[§] Preempção é a ação ou evento que causa mudança do processamento de uma aplicação para outra. [LEX 99].

[**] Inversão de prioridade é a situação na qual uma tarefa pode ficar esperando por outra de menor prioridade por um tempo indeterminado. Por exemplo, se duas tarefas compartilham um recurso e a tarefa menos prioritária entrar na região crítica, a tarefa de maior prioridade ficará bloqueada. Enquanto isto, outras tarefas de prioridade intermediária podem assumir o processamento indefinidamente [ROD 97].

[††] *Jitter* em comunicação de dados é a distorção causada pela falta de sincronismo dos sinais [MIC 98] ou uma instabilidade do sinal em um pequeno espaço de tempo.

lógicas ou componentes ativos, que podem ser implementados em hardware ou software. A programação baseada em objetos pode ser vista como a construção de diferentes máquinas que são interligadas para executar a tarefa apropriada. Como os componentes ativos podem tanto ser processos em execução no sistema operacional quanto processadores executando uma função, esta abordagem é compatível com a evolução para arquiteturas distribuídas.

O paradigma de orientação a objetos no desenvolvimento de grandes sistemas já é uma técnica consagrada e madura, e conforme estudos citados por Becker [BEC 98] também se mostra bastante adequado para sistemas distribuídos tempo-real, como por exemplo as atuais arquiteturas de automação industrial. Uma das vantagens mais instigantes é a possibilidade de mapeamento direto entre as máquinas lógicas ou componentes ativos do modelo e as unidades autônomas de processamento que existem no mundo real, tais como os componentes de uma planta industrial. Como os objetos são unidades naturais para execução concorrente, a modelagem de sistemas distribuídos é feita de uma forma muito intuitiva.

Assim como a automação industrial tem sido usada no atendimento das crescentes exigências de produtividade e qualidade do mercado internacional, a complexidade de sistemas distribuídos tempo-real utilizados nestas aplicações tem atingido níveis sem precedentes que exigem metodologias de engenharia de sistemas [PER 97]. Os poderosos conceitos da orientação a objetos tais como modularidade, encapsulamento, abstração, generalização, especialização e agregação podem ser utilizados no gerenciamento desta complexidade, permitindo uma quebra lógica do domínio do problema em objetos independentes com acréscimo incremental dos detalhes [FLO 98]. Relações cliente-servidor e serviços hierárquicos podem ser implementados através de mensagens trocadas entre objetos.

Conforme Kim [KIM 2000], a nova geração de programação distribuída precisa ser baseada num estilo de programação geral e de alto nível que possa acomodar com mínimo esforço as atuais gerações de linguagens comerciais, definindo a interação entre componentes distribuídos e os requisitos de tempo de uma forma intuitiva. A aplicação da orientação a objetos na implementação de sistemas distribuídos levou ao surgimento de tecnologias como *Microsoft's DCOM* [MIC 2000], *Sun's Java RMI* [SUN 2000] e *Object Management Group's CORBA* [OMG 2000 e SCH 2000]. Conhecidas como *middleware* estas tecnologias oferecem um nível de abstração mais elevado para os projetistas. Por exemplo, o CORBA permite aplicações se comunicarem através de um barramento de objetos (ORB) não importando sua localização ou quem as projetou, facilitando a programação distribuída em ambientes heterogêneos.

Conforme Loyall et al [LOY 2000], em teoria é possível desenvolver um sistema complexo partindo do zero, isto é, sem utilizar nenhum *middleware* de mercado ou *Commercial Off-The-Shelf* (COTS). No entanto, as pressões e restrições competitivas tornam isto implausível na prática, assim os COTS *middleware* têm desempenhado papéis crescentes no desenvolvimento de sistemas distribuídos tempo-real e nas pesquisas para adaptá-los. Tecnologias de objetos convencionais como DCOM, Java RMI e CORBA falham em atender os requisitos fim-a-fim, porque escondem os detalhes necessários para especificar, medir e controlar a qualidade de serviço. Neste contexto, diferentes extensões tempo-real têm sido desenvolvidas para estes modelos.

3 Estado da Arte em Sistemas Tempo-Real

Este capítulo apresenta uma visão do estado da arte no desenvolvimento de sistemas tempo-real, com enfoque em modelagem e implementação utilizando técnicas de orientação a objetos. Além dos tópicos apresentados, diversos outros estudos recentes podem ser encontrados nos anais dos últimos *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (ISORC) [IEEE 99, 2000 e 2001], o que confirma os atuais esforços de utilização do paradigma de orientação a objetos no desenvolvimento de sistemas distribuídos tempo-real.

3.1 Modelagem Tempo-Real

Esta seção apresenta algumas tecnologias para análise e projeto de sistemas tempo-real que se baseiam no paradigma de orientação a objetos. São abordados modelos acadêmicos como o SIMOO-RT e modelos criados por fabricantes de ferramentas de modelagem como o ROOM, mas a maior ênfase é dada ao estudo das extensões tempo-real da *Unified Modeling Language* (UML).

3.1.1 SIMOO-RT

O SIMOO-RT, *Plataforma Orientada a Objetos para Simulação Discreta Multi-Paradigma Tempo-Real*, apresentado por Becker [BEC 99] é um ambiente para modelagem orientada a objetos, simulação e geração de código para sistemas distribuídos tempo-real.

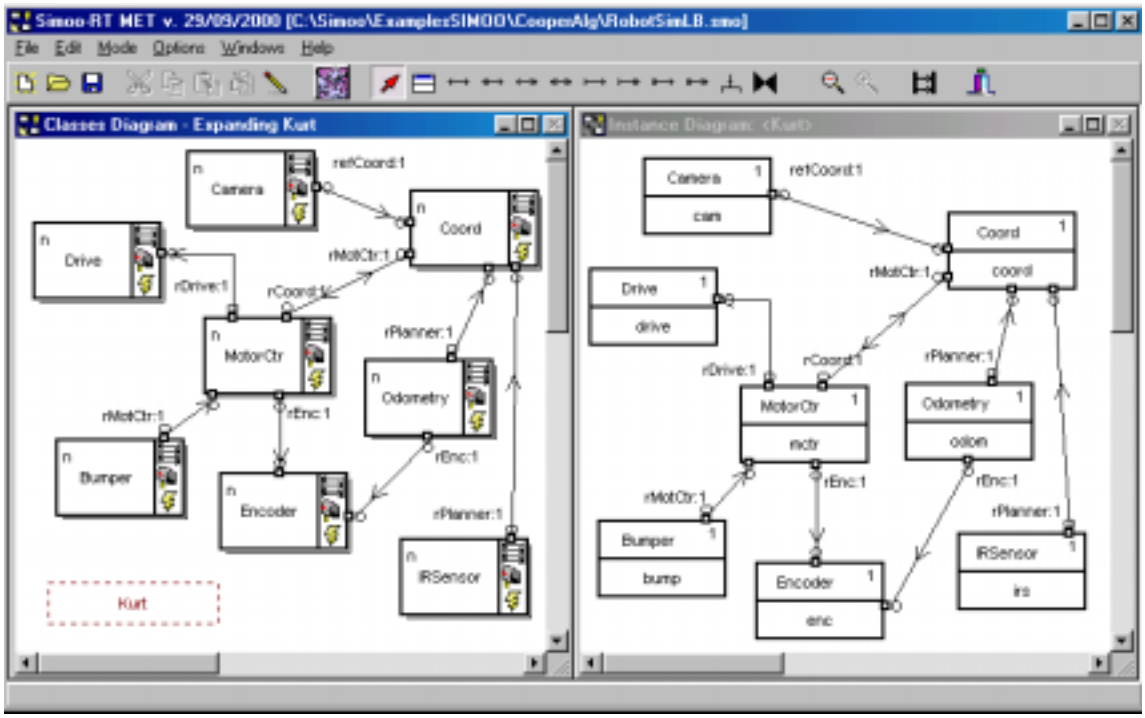


FIGURA 3.1 – Diagrama de Classes e Instâncias do SIMOO-RT [BEC 2001]

Dois diagramas fundamentais no ambiente SIMOO-RT são o diagrama de classes e o de instâncias. Estes diagramas são utilizados nas etapas iniciais de projeto, permitindo a definição dos elementos constituintes da aplicação e as formas com as quais estes elementos se relacionam. Enquanto o primeiro representa a estrutura genérica para os elementos que irão fazer parte do sistema, o segundo sugere a realização desta estrutura através das instâncias de classes, ou simplesmente objetos. No SIMOO-RT, o conceito de agregação é modelado de forma hierárquica através do detalhamento de uma classe, conforme pode ser observado na FIGURA 3.1 que representa as classes que integram a classe *Kurt* [BEC 2001].

Como o SIMOO-RT permite associar restrições temporais aos métodos das classes, basicamente execução periódica e execução com limite de tempo de execução, os requisitos temporais levantados nas fases de análise podem ser adequadamente representados nos diagramas. O gerador de código atualmente disponível é para linguagem *Active Object C++* (AO/C++), uma extensão de C++ que inclui primitivas para definição de objetos ativos, métodos agendados, requisitos de tempo, e a possibilidade que seja feito um mapeamento da distribuição lógica dos objetos modelados para distribuição física dos processos numa rede, utilizando as primitivas de comunicação entre processos do sistema operacional QNX. A geração de código é relativamente direta, porque a linguagem AO/C++ possui suporte nativo a estas associações entre características temporais e métodos de classes.

3.1.2 Real-Time Object-Oriented Modeling

O *Real-Time Object-Oriented Modeling* (ROOM) [SEL 94] é uma linguagem de modelagem visual orientada a objetos com semânticas para a especificação de sistemas tempo-real, sendo suportada pela ferramenta CASE que era comercializada pela empresa canadense *ObjecTime*[‡‡]. É uma linguagem otimizada para especificação, visualização, documentação e construção automatizada de complexos sistemas orientados a eventos, potencialmente distribuídos e com características tempo-real.

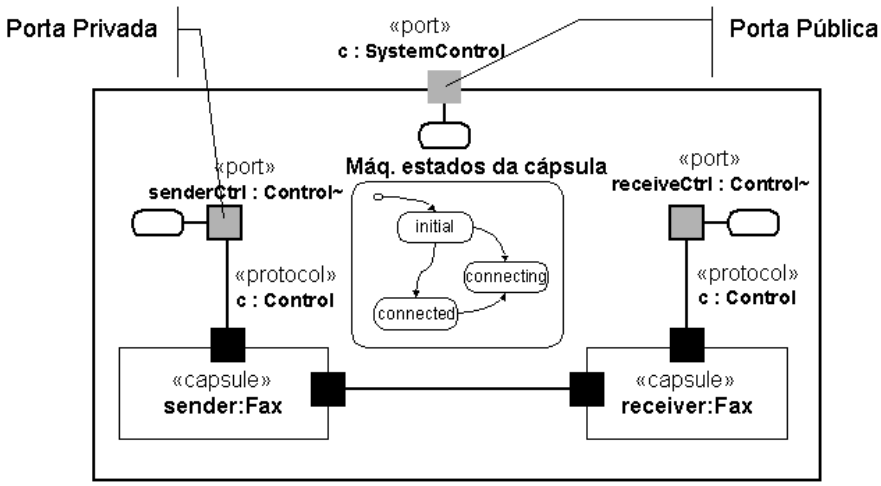


FIGURA 3.2 – Elementos da Metodologia ROOM [BEC 2001]

[‡‡] A metodologia ROOM foi absorvida pela empresa *Rational Software*.

A metodologia ROOM propõe dois níveis de modelagem. Na modelagem da arquitetura é feita uma especificação abstrata que delimita os padrões estruturais e comportamentais de todos os componentes do sistema. Na modelagem detalhada são incorporadas construções C++ na forma de pseudocódigo. As mensagens e o detalhamento de dados são representados por objetos passivos enquanto os objetos que possuem sua própria *thread* de execução concorrente são representados por objetos ativos, denominados cápsulas[§§]. As cápsulas podem receber mensagens através de portas, cada qual com seu próprio protocolo. Conforme pode ser visto na FIGURA 3.2, o diagrama de estrutura apresenta os relacionamentos de comunicação entre as cápsulas e a decomposição das cápsulas mais complexas em elementos mais simples. Através de diagramas de máquinas de estado denominados *ROOMCharts* é possível descrever o comportamento interno das cápsulas.

3.1.3 Real-Time Unified Modeling Language

Conforme Booch et al [BOO 99], a popularização do paradigma de orientação a objetos levou à publicação de inúmeras metodologias de análise e projeto desta natureza, crescendo de 10 para mais de 50 no período de 1989 a 1994. Muitos usuários destes métodos tiveram problemas em encontrar uma linguagem de modelagem que atendesse completamente suas necessidades. Introduzida em 1997 pela *Object Management Group* (OMG), a linguagem UML teve rápida aceitação através da indústria de software como uma linguagem gráfica para especificação, construção, visualização e documentação de sistemas, sendo considerada atualmente um padrão de referência para a representação de sistemas orientados a objetos. Os sistemas distribuídos tempo-real envolvem uma série de requisitos temporais, assim, foi natural o surgimento de extensões na linguagem UML para melhorar o tratamento desta classe de sistemas.

3.1.3.1 Rapid Object-Oriented Process for Embedded Systems

Em seu livro *Real-Time UML*, Douglass [DOU 99] apresenta um método de modelagem orientado a objetos voltado para sistemas tempo-real embarcados utilizando UML, denominado *Rapid Object-Oriented Process for Embedded Systems* (ROPES). Neste trabalho, Douglass define um conjunto de atividades baseadas em UML para análise e projeto de sistemas tempo-real e o utiliza em alguns exemplos típicos de sistemas tempo-real. Nas fases de análise iniciais, são levantados os requisitos sem revelar a estrutura interna e é definida a arquitetura do sistema determinando a melhor divisão entre hardware e software. Na análise de objetos são identificados classes e objetos, sua estrutura estática e seu comportamento em resposta a estímulos internos ou do ambiente. O projeto define a solução que otimiza a aplicação conforme os objetivos do projeto, sendo detalhados a arquitetura, os processos concorrentes e a dinâmica de colaboração entre os objetos. Utilizando os mecanismos de extensão padrão da própria UML, Douglass associa anotações temporais nos diagramas de interação (*tagged values* e *constraints*), no entanto estas anotações não possuem qualquer significado semântico na linguagem, podendo ser considerados como comentários associados aos diagramas. Também é proposto um novo diagrama, denominado *Timing Diagrams* usado para

[§§] Os objetos ativos eram originalmente denominados como atores, mas foram rebatizados como cápsulas em trabalhos posteriores de Selic [SEL 99] para evitar conflito de conceitos com UML.

representar a evolução dos estados de objetos ativos ao longo do tempo.

3.1.3.2 Concurrent Object Modeling and Architectural Method - COMET

Outro método de modelagem bastante interessante é o COMET, *Concurrent Object Modeling and Architectural Method*, apresentado por Gomaa [GOM 2000] e baseado no conceito de *Use Case* da UML. Os requisitos funcionais do sistema são definidos em termos de atores e cenários[***], onde são definidas as seqüências de interações entre um ou mais atores e o sistema. Os *Use Cases* podem ser vistos em diagramas com vários níveis de detalhamento. No modelo de requisitos os requisitos funcionais do sistema são definidos em termos de atores e *Use Cases*. No modelo de análise, os diagramas são refinados para descrever os objetos participantes e suas interações. No modelo de projeto, a arquitetura de software é definida, endereçando aspectos de distribuição, concorrência e orientação a objetos (modularidade, encapsulamento de informação, abstração, generalização, especialização e agregação). O COMET utiliza apenas mecanismos de extensão padrão da própria UML para representar aspectos temporais em seus diagramas. Para caracterizar atores e objetos em critérios como ativação, concorrência, sincronismo, comportamento e estruturação são definidos diversos tipos de *stereotypes* como por exemplo: «*asynchronous device*», «*connector*», «*coordinator*», «*external timer*», «*mutually exclusive clustering*», «*non-time critical*», «*periodic input device interface*», «*resource monitor*», «*state dependent control*», «*temporal clustering*» *et cetera*.

3.1.3.3 UML Profile for Scheduling, Performance, and Time

A definição de perfis (*profiles*) foi uma alternativa criada pela OMG para evitar a proliferação de novos conceitos na UML, representando uma maneira comum de modelar características específicas, voltadas para um determinado tipo de problema. Para a área de modelagem tempo-real foi proposto o *UML Profile for Scheduling, Performance, and Time* (UMLSPT) criado por especialistas da OMG [OMG 2001] para análise e projeto de sistemas tempo-real.

Um sistema de tempo-real precisa apresentar um comportamento temporal previsível, ou seja, os escalonamentos e reações a eventos precisam ser quantificados e conhecidos com antecedência à execução e construção do sistema. Para isto, deve ser padronizada a forma de utilizar os mecanismos de extensão da UML na inclusão de informações temporais nos diagramas de forma que estas possam ser compartilhadas entre ferramentas de análise e ferramentas de modelagem.

O perfil UMLSPT define um esqueleto básico que é subdividido nos modelos de referência, modelo de tempo e seus mecanismos e modelo de concorrência e baseado no esqueleto básico existe também o modelo de escalonabilidade. Nestes modelos são padronizados conceitos como instante e padrão de ativação, duração, pior caso, tempo limite de execução, prioridade, controle de acesso, representação de arquiteturas físicas, modelagem de diagramas de concorrência *et cetera*.

Neste trabalho de conclusão, o modelo de escalonabilidade foi considerado um dos mais interessantes do perfil UMLSPT porque define conceitos que podem corresponder aos diagramas de interação UML acrescidos de *stereotypes* para caracterizar os aspectos

[***] Cenário é uma seqüência de ações que ilustra um comportamento. Um cenário pode ser usado para demonstrar uma interação ou a execução de uma instância de *Use Case* [BOO 99a].

temporais. Esta prática corresponde exatamente à abordagem adotada por Douglass [DOU 99] e Gomaa [GOM 2000] em seus respectivos métodos de modelagem.

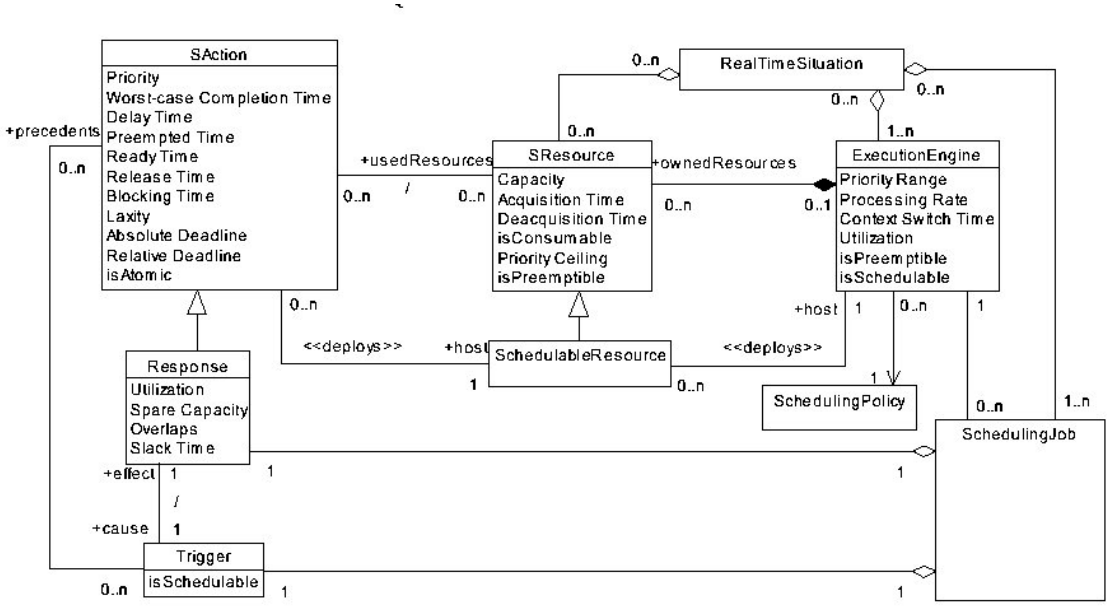


FIGURA 3.3 – The Schedulability Analysis Model [OMG 2001]

Conforme representado na FIGURA 3.3, *RealTimeSituation* é um tipo de especial de contexto de análise no qual existem dois tipos de recursos: *execution engines* que executam cenários e recursos protegidos (*SResource*) que podem ser compartilhados por um ou mais cenários. Um *trigger* representa a carga de uso que especifica o padrão de ativação de um cenário. Um cenário é uma seqüência de ações, sendo representado por um *response* que possui características de escalonamento como recursos usados, pior caso de execução, prioridade *et cetera*. Estas características representam os valores de qualidade de serviço (QoS) desejados que precisam ser compatíveis com os valores oferecidos pelos *execution engines* e recursos protegidos. Um recurso escalonável é um tipo especial de recurso, similar a uma *thread*, que pode executar concorrentemente as ações de um cenário conforme uma política de escalonamento[†††].

Um exemplo de *RealTimeSituation* representado por um diagrama de colaboração com seus *triggers*, *responses*, recursos compartilhados e características correspondentes pode ser visto na FIGURA 5.20.

3.2 Programação Tempo-Real

Nesta seção são abordadas algumas soluções orientadas a objetos para implementação de sistemas tempo-real. São apresentadas as extensões tempo-real para *Common Object Request Broker Architecture* (CORBA), o que permite a utilização deste *middleware* em sistemas distribuídos tempo-real. Também são comentadas as atuais propostas de extensão tempo-real para linguagem Java.

[†††] A política de escalonamento pode ser implementada por algoritmos de escalonamento como *Rate Monotonic*, *Earliest Deadline First* entre outros, descritos no livro de Burns e Welling [BUR 96].

3.2.1 Real-Time CORBA

A especificação Real-Time CORBA define uma série de características para suportar previsibilidade fim-a-fim de operações com prioridade fixa, estendendo o padrão CORBA existente, conforme apresentado no artigo de Schmidt & Khuns [SCH 2000b]. Esta especificação realiza melhorias na versão 1.1 do protocolo GIOP/IIOP (*General Inter-ORB Protocol/ Internet Inter-ORB Protocol*) e na política de QoS da especificação de mensagens da OMG.

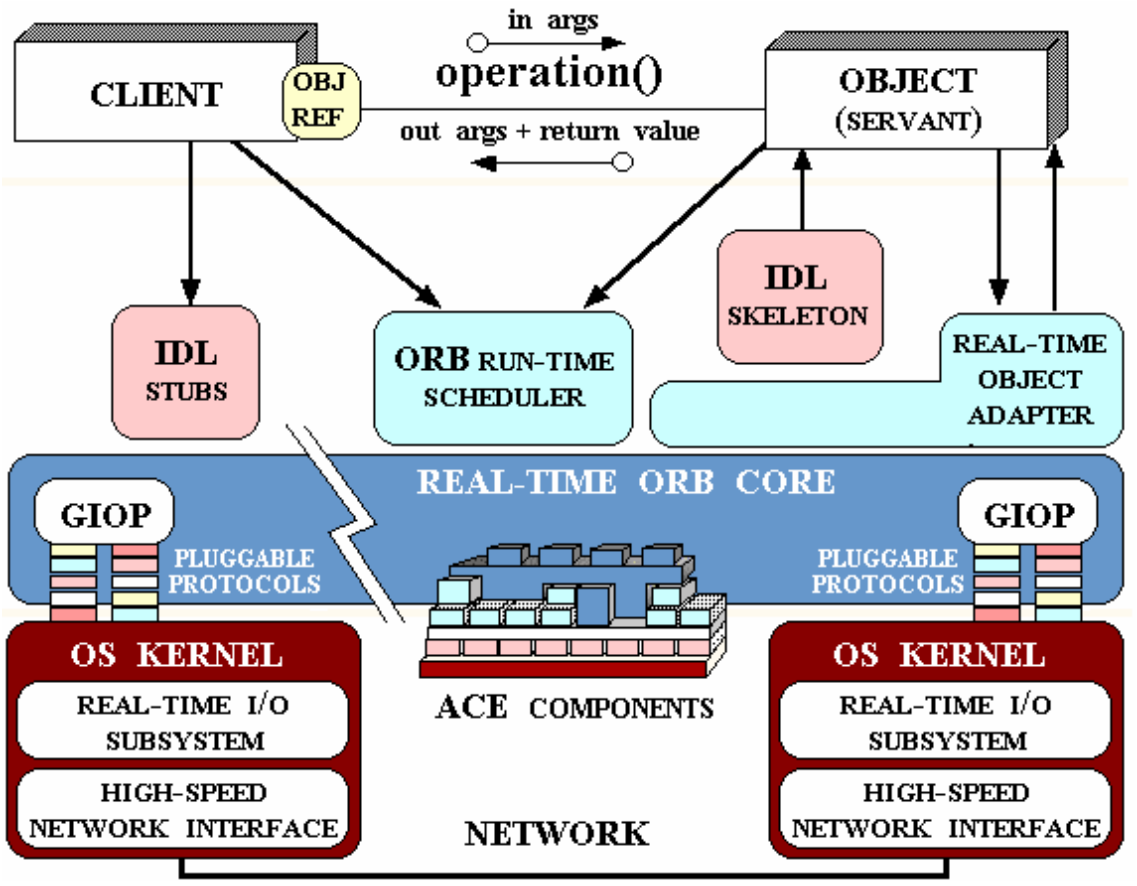


FIGURA 3.4 – TAO: The ACE ORB [SCH 2000a]

Para assegurar um comportamento previsível de atividades fim-a-fim que envolvem múltiplos objetos distribuídos, o RT-CORBA inclui definições em diferentes níveis de abstração:

- Gerenciamento de recursos da infra-estrutura de comunicação – um sistema RT-CORBA precisa possuir mecanismos que garantam os recursos necessários a uma comunicação previsível. Por exemplo, a alocação prévia de recursos de rede para atender a uma vazão e um atraso definidos no momento de uma conexão do tipo *Asynchronous Transfer Mode* (ATM);
- Mecanismos de escalonamento – como o RT-CORBA se destina a sistemas tempo-real com prioridade fixa, o barramento de objetos (ORB) precisa definir a prioridade das *threads* do sistema operacional;

- ORB tempo-real – existem interfaces padronizadas para as aplicações definirem suas necessidades de recursos. É possível configurar prioridades das *threads*, *buffers* de mensagens, conexões no nível de transporte e sinalizações de rede de forma a permitir o controle do comportamento do ORB;
- Serviços tempo-real – para garantir um comportamento previsível de atividades fim-a-fim que envolvem clientes e servidores, existem serviços como um escalonamento global que pode ser usado para gerenciar recursos distribuídos.

O RT-CORBA define um tipo de prioridade global que pode ser levada pela mensagem de invocação ao longo de diferentes ORB's. Dependendo do tipo de escalonamento, esta prioridade pode ser usada pelos servidores para ordenar as requisições recebidas. Estas prioridades podem tanto ser propagadas pelo cliente quanto definidas pelo servidor.

Assim como qualquer outra padronização definida pela OMG, a versão inicial do RT-CORBA emitido em outubro de 1998, foi baseada na união de diferentes *Request For Proposal* (RFP). Novos estudos continuam sendo realizados e suas contribuições devem ser gradativamente incluídas no padrão da OMG. Por exemplo, Steven Wohlever et al [WOH 99] apresentam extensões tempo-real para o *Trader Service*, que permitem que os objetos que publicam seus serviços para aguardar por requisições de atividades, possam também divulgar seu comportamento temporal.

Da mesma forma, o TAO representado na FIGURA 3.4 é um ORB compatível com o padrão RT-CORBA de fevereiro de 1999, que possui suas próprias extensões como: escalonamento dinâmico, gerenciamento de eventos de entrada e saída (que evitam problemas de inversão de prioridade) usadas na interface com redes de alta velocidade, uma versão tempo-real para o *Event Service et cetera*.

3.2.2 Java Expert Group versus J Consortium

Existem dois grupos concorrentes trabalhando na especificação de extensões tempo-real para a linguagem Java: o *Java Expert Group* e o *J Consortium*. Embora discordem em alguns pontos, ambas especificações seguem as recomendações no *National Institute of Standards and Technologies* (NIST) [NIS 99]. Ao comparar estas especificações, a proposta do *Java Expert Group* foi considerada mais sólida e estava num estágio mais avançado durante a fase de pesquisa deste trabalho de conclusão e será abordada com maior profundidade nas próximas seções.

3.2.2.1 J Consortium

Uma das primeiras iniciativas de utilização da linguagem Java no desenvolvimento de aplicações com restrições temporais foi de Nielsen [NIL 98] que especificou uma API Java denominada *Portable Executive for Reliable Control* (PERC) Real-Time API. Esta especificação previa algumas construções Java para execução de código atômico e execução de código com limite de tempo, mas que nunca foram implementadas e não fazem parte do produto PERC da empresa *Newmonics* [NEW 2000]. Como a especificação do *J Consortium* [JC 2000] é baseada no PERC, um produto vinculado ao antigo JDK 1.1, e este grupo não conta com a participação da *Sun Microsystems*, existe um risco de incompatibilidade com futuras versões da linguagem Java e conseqüentemente menores chances de popularização desta proposta.

3.2.2.2 Java Expert Group

O *Java Expert Group* definiu a primeira especificação oficial para o desenvolvimento de aplicações tempo-real com a plataforma Java, sendo apresentada no livro *The Real-Time Specification for Java* (RTSJ) de Bollella et al [BOL 2000]. Desenvolvida no âmbito do *Java Community Process*, mecanismo criado pela *Sun Microsystems* para gerenciar melhorias e alterações na linguagem Java, esta proposta tem como vantagens não estar atrelada a nenhuma implementação específica e a garantia de compatibilidade com futuras versões do Java. Além disso, atualmente já existem sistemas comerciais que a implementam, tais como o processador da empresa *aJile* [AJI 2002] e as placas de desenvolvimento *JStamp* da empresa *Systronix* [SYS 2002] ou ainda o sistema J9 da empresa IBM [IBM 2002], este último uma combinação do ambiente *Visual Age Micro Edition* rodando sobre o sistema operacional *Neutrino* da empresa QNX.

3.2.3 The Real-Time Specification for Java

Apresentada em Bollella et al [BOL 2000], esta especificação define uma interface de programação (API) para a linguagem Java que permite a criação, verificação, análise, execução e gerenciamento de *threads* tempo-real, cuja correção depende também da satisfação de requisitos temporais. O *Java Expert Group* adotou os seguintes princípios gerais a serem seguidos pela RTSJ:

- Aplicabilidade para ambientes Java particulares: o RTJS não deve incluir especificações que restrinjam seu uso para ambientes Java particulares, tais como versões especiais do JDK ou J2ME;
- Compatibilidade reversa: aplicações sem restrições temporais não devem sofrer restrições de execução no RTJS;
- Portabilidade: o RTSJ reconhece a importância do princípio "escreva uma vez e rode em qualquer lugar" (*write once run anywhere*), mas reconhecendo a dificuldade de alcançar este objetivo em aplicações tempo-real, não irá exigir a compatibilidade binária ao custo da previsibilidade do código;
- Prática atual *versus* técnicas avançadas: a especificação deve abordar a prática atual de programação para sistemas tempo-real, mas permitir que futuras implementações utilizem técnicas avançadas;
- Previsibilidade de execução: esta deve ser a prioridade em todas as escolhas, com a conseqüência de necessitar a realização de medidas de desempenho;
- Ausência de extensões sintáticas: de modo a facilitar o trabalho dos desenvolvedores de ferramentas e aumentar a probabilidade de implementações dependentes de tempo, o RTSJ não deverá introduzir novas palavras-chave ou fazer alterações sintáticas na linguagem Java;
- Permitir variações em decisões de implementação: é um fato reconhecido que a implementação do RTSJ pode ter variações em inúmeras decisões de implementação, tais como o uso de algoritmos mais eficientes ou menos eficientes, escolha entre eficiência temporal ou de tamanho, implementação de algoritmos de escalonamento não requeridos na especificação mínima *et cetera*. A especificação é flexível, permitindo a criação de versões da API que sejam mais adequadas para atender os requisitos de cada clientes.

No trabalho de especificação da RTSJ API desenvolvido pelo *Java Expert Group*, as alterações e melhorias introduzidas na linguagem Java podem ser agrupadas em sete áreas conforme descrição que segue:

- Escalonamento e despacho de *threads*: reconhecendo uma significativa diversidade de modelos de escalonamento e despacho e a larga aplicação destes em sistemas tempo-real, a RTSJ permite que as *threads* Java de tempo-real utilizem mecanismos de escalonamento legados do sistema operacional, e que as implementações da RTSJ forneçam algoritmos de escalonamento não previstos inicialmente. O algoritmo básico de escalonamento a ser fornecido em todas as versões da API é o escalonamento preemptivo baseado em prioridade, com no mínimo 28 diferentes níveis de prioridade.
- Gerenciamento de memória: reconhecendo a importância do gerenciamento automático de memória na linguagem Java, também conhecido como *garbage collector* (GC), será seguida a filosofia de permitir, na medida do possível, que este trabalho continue sendo realizado da mesma forma, sem perturbar a atividade de programação. Como já existem diversos algoritmos GC disponíveis em certos sistemas tempo-real, o RTSJ procura acomodá-los definindo as seguintes características em seu mecanismo de gerenciamento de memória: independência de qualquer algoritmo GC em particular; permitir ao programa caracterizar precisamente os efeitos do algoritmo GC no tempo de execução, preempção e despacho de *threads* Java de tempo-real; permitir a alocação e liberação de memória fora de qualquer interferência de qualquer algoritmo GC;
- Sincronização e compartilhamento de recursos: a semântica da palavra-chave Java *synchronized* é mantida, mas são previstos um ou mais algoritmos de sincronização para solucionar o problema de inversão de prioridade que exige maior atenção em sistemas tempo-real. Para as situações nas quais é necessária uma prioridade superior a dos algoritmos GC, o compartilhamento de recursos pode ser feito através de classes que implementam filas livres de espera;
- Tratamento de eventos assíncronos: sistemas tempo-real interagem fortemente com o mundo real onde eventos ocorrem de forma assíncrona, assim são necessários mecanismos de programação eficientes para acomodar-se a este ambiente. O RTSJ prevê classes que representam coisas que podem acontecer e lógicas que executam quando estas coisas acontecem, sendo esta execução escalonada e despachada da mesma forma que as *threads* Java de tempo-real;
- Transferência assíncrona de controle (ATC): algumas vezes, alterações drásticas e assíncronas no mundo real exigem que a lógica em execução seja transferida imediatamente e de forma eficiente para outro código. O RTSJ inclui um mecanismo que permite a programação de uma troca assíncrona de controle de uma *thread* para outra;
- Encerramento assíncrono de *thread*: novamente, devido a alterações drásticas e assíncronas no mundo real pode ser necessário encerrar a execução de uma *thread*, mas ao contrário do antigo método *stop* que está em processo de eliminação do JDK (*deprecated*), este mecanismo do RTSJ é seguro, pois gera uma exceção que ao ser tratada permite a execução de finalizações;

- Acesso físico à memória: apesar de não ser diretamente uma questão tempo-real, o acesso físico à memória é desejável por muitas aplicações que poderiam fazer uso da implementação da RTSJ. Foi definida uma classe que permite este tipo de acesso, bem como classes que permitem a construção de objetos num endereço de memória física.

Uma descrição resumida de todas as classes especificadas para *Real-Time Specification for Java* API pode ser encontrada em [HÖL 2001].

3.2.4 Outras Opções de Programação Tempo-Real

No estudo para este trabalho de conclusão também foram analisadas algumas outras tecnologias que têm sido apresentadas em seminários recentes [IEEE 99, 2000 e 2001] que abordam ambientes de programação que permitem expressar restrições temporais. Seguem alguns exemplos:

- O esquema denominado *Time-triggered Message-triggered Object* (TMO) procura remover as limitações das técnicas convencionais de orientação a objetos para aplicações tempo-real. Em [KIM 99] é proposta sua inclusão como um *middleware* em sistemas operacionais de mercado sem realizar alterações de *kernel*. Seu protótipo foi implementado no Windows NT, um dos sistemas que atende aos requisitos do TMO;
- No método *Time-Triggered Architecture* é proposto por Kopetz et al [KOP 99] um tipo especial de interface denominada *temporal firewall*, a ser utilizada entre subsistemas de aplicações distribuídas tempo-real do tipo *safety critical*. O método define três interfaces deste tipo, para comunicação com a rede, para controle dos objetos e para função de *gateway* entre os subsistemas;
- No artigo de Rufino et al [RUF 99] são analisados os desafios na utilização do paradigma de orientação a objetos no barramento de campo *Controller Area Network* (CAN), uma solução largamente utilizada no chão de fábrica devido a suas características de tolerância a falhas. As maiores dificuldades são os limites para largura de banda (1 Mbps) e o tamanho máximo da mensagem (8 bytes), que são insuficientes para suportar a interação entre objetos distribuídos. Na abordagem de Kaiser & Mock [KAI 99] para esta mesma rede, é proposto o modelo de publicador e assinante utilizando comunicações anônimas e assíncronas entre os objetos através dos mecanismos de *broadcast e hardware filtering* do CAN;
- O produto *OAenterprise* é um sistema *Supervisory Control and Data Acquisition* (SCADA) que utiliza a tecnologia DCOM com algumas melhorias para atender requisitos de automação industrial. Por exemplo, no ciclo de vida do modelo DCOM, os objetos são automaticamente eliminados quando sua contagem de referência chega a zero. No ambiente de manufatura isto é inaceitável porque o objeto pode conter o código de controle de um elemento crucial da lógica de controle. Assim, este sistema possui seu próprio gerenciamento de ciclo de vida e serviço de nomes que utiliza operações explícitas como *startup* e *shutdown*. Outra importante característica é um mecanismo para gerenciamento de objetos redundantes executando em diferentes estações, no qual o objeto redundante assume o comando de forma transparente em caso de falha no objeto primário [WEL 2000].

4 Proposta de Mapeamento

4.1 Visão Geral da Proposta

Existem atualmente várias iniciativas de padronização de uso de orientação a objetos no desenvolvimento de sistemas tempo-real, porém estas estão focadas em etapas isoladas do ciclo de vida de desenvolvimento de sistemas. Existem propostas relacionadas à utilização da *Unified Modeling Language* [DOU 99, GOM 2000 e OMG 2001] na modelagem de sistemas tempo-real, estudos relacionados à extensão do *middleware* CORBA [SCH 2000] para sua adequação a sistemas distribuídos tempo-real [LOY 2000, OMG 2001a, SCH 2000a, SCH 2000b e WOH 99] e propostas de extensões de linguagens de programação como Real-Time Java [BOL 2000, ITO 99, JC 2000, NIL 98 e NIS 99]. Apesar de já existirem trabalhos que cobrem todo o ciclo de desenvolvimento de sistemas tempo-real, como o SIMOO-RT desenvolvido por Becker [BEC 99], ainda não existem trabalhos ligando tecnologias de modelagem e programação de larga aceitação como Real-Time UML com Real-Time Java.

No desenvolvimento deste trabalho, optou-se por realizar o mapeamento de uma linguagem de modelagem para uma linguagem de programação com características que facilitam a especificação de requisitos temporais, em detrimento do mapeamento para um *middleware* ou sistema operacional tempo-real. Ao utilizar conceitos mais abstratos, uma linguagem de programação para sistemas tempo-real está muito mais próxima de uma linguagem de modelagem, o que reduziu o esforço despendido na confecção desta dissertação. Portanto, será proposto o mapeamento entre diagramas UML com restrições temporais e a API tempo-real para Java, *The Real-Time Specification for Java* (RTSJ) [BOL 2000]. Como a RTSJ é deliberadamente focada em sistemas centralizados, aspectos relacionados a sistemas distribuídos tempo-real não serão abordados neste trabalho.

Neste capítulo inicialmente serão apresentados alguns requisitos temporais que devem fazer parte da especificação de sistemas tempo-real e na seqüência serão abordados os conceitos disponíveis em UML e RTSJ. Esta lista de requisitos restringe-se aos conceitos mais comuns em sistemas centralizados tempo-real que foram selecionados para o escopo deste trabalho, sendo possível sua utilização simultânea na caracterização de um determinado cenário:

- Ativações assíncronas – além da execução síncrona convencional, deve ser possível especificar que a execução de uma determinada atividade irá executar em paralelo com o solicitante que não precisará aguardar pelo término da atividade solicitada[‡‡‡];
- Padrão de ativação – deve ser possível definir se determinada computação será realizada a cada período de tempo, será ativada com um intervalo mínimo entre execuções ou algum outro padrão qualquer de ativação aperiódica;
- Limite de tempo de execução (*deadline*) – requisito que exige que o processamento seja concluído dentro de um limite de tempo, devido ao impacto físico da atividade do sistema controlado sobre o ambiente [STA 88]. Também

[‡‡‡] Uma ativação assíncrona implica na delegação da execução da atividade solicitada para uma unidade de processamento concorrente tipo uma *thread*, processo ou similar.

pode ser visto como a limitação rígida do intervalo de tempo fim-a-fim entre um estímulo e a conclusão da atuação correspondente ou, ainda, como o tempo máximo de espera pela chegada de uma mensagem;

- Exceções – quando o tempo máximo de execução ou o período de uma atividade cíclica não é respeitado, existe necessidade da realização de alguma ação corretiva para eliminar os efeitos da falha temporal ocorrida ou para assumir um estado seguro (*fail-safe*);
- Sincronização – ao compartilhar recursos com outras tarefas, existe necessidade de controle de acesso que garanta que as consistências não serão violadas, ao mesmo tempo em que são evitados problemas de inversão de prioridade;
- Prioridade de execução – atributo que define a preferência na execução de determinada tarefa concorrente, sendo definido em função do algoritmo de escalonamento utilizado. No caso do algoritmo *Earliest Deadline First* (EDF) a definição da prioridade é baseada nos *deadlines*, enquanto no *Rate Monotonic Analysis* (RMA) a prioridade é determinada com base no período das tarefas [BUR 96];
- Restrição de *jitter* ou flutuação de fase – algumas aplicações também impõem restrições em relação à variação entre atuações consecutivas ou variação do início ou fim da execução em relação à base de tempo.

4.2 Conceitos UML a Serem Considerados

O desenvolvimento de um sistema tempo-real envolve as etapas tradicionais de Engenharia de Software, como análise, projeto, codificação e testes, além de uma etapa adicional para garantir sua correção temporal. Devido à sua natureza genérica, UML oferece uma vasta gama de conceitos para suportar as etapas de análise e projeto de um sistema, mas neste trabalho serão considerados apenas os elementos utilizados na representação dos aspectos temporais.

4.2.1 Diagramas UML

Para o desenvolvimento de um sistema utilizando o paradigma de orientação a objetos, UML define diagramas que permitem a modelagem de diferentes visões do sistema sendo criado, podendo ser utilizada em diferentes etapas do desenvolvimento, em especial nas fases de análise de requisitos e projeto de software. Estes diagramas incluem conceitos que permitem expressar os requisitos do sistema, a estruturação estática do sistema em classes e objetos, o relacionamento dinâmico entre estes objetos, a seqüência de estados internos assumidos por estes objetos, a distribuição destes objetos na arquitetura física do sistema *et cetera*.

- Diagrama *Use Case*: são diagramas utilizados na fase de análise para capturar os requisitos ou funcionalidades providas por um sistema através da troca de mensagens entre o sistema e atores externos, conforme exemplificado na FIGURA 5.5. Cada um destes diagramas define parte do comportamento do sistema sem revelar sua estrutura interna. Alguns aspectos de tempo e

desempenho já podem ser identificados em sua elaboração;

- Diagrama de classes: é uma representação gráfica de uma visão estática de um conjunto de classes, como na FIGURA 5.18. Este diagrama pode conter elementos que definem características de comportamento (métodos das classes), inclusive contendo restrições temporais (como período e *deadline*), embora sua dinâmica seja expressa em outros diagramas;
- Diagrama de seqüência: é um diagrama para modelagem dinâmica que mostra a interação entre objetos arranjados numa seqüência de tempo. Na modelagem de sistemas tempo-real, é possível associar limites de tempo à seqüência de mensagens[§§§] trocadas de forma intuitiva conforme visto na FIGURA 5.12;
- Diagrama de colaboração: é outro tipo de diagrama de interação que permite modelar o relacionamento dinâmico entre papéis exercidos pelos objetos dentro da colaboração. Como o tempo não corresponde a uma das dimensões do diagrama, a seqüência de mensagens e as *threads* concorrentes são representadas utilizando números seqüenciais conforme observado na FIGURA 5.20;
- Máquina de estado (*statechart*): é um diagrama que especifica o comportamento de um objeto, através da definição da seqüência de estados assumidos por este objeto em resposta a eventos recebidos em conjunto com as ações correspondentes. As transições entre os estados também podem ser disparadas por eventos periódicos ou ao atingir limites de tempo de permanência num determinado estado;
- Diagrama de instalação (*deployment*): é um diagrama que mostra a configuração de execução dos nodos de processamento, sua interligação e os componentes e objetos que executam nestes nodos. Este diagrama oferece bons recursos para a inserção de características de qualidade de serviço (QoS) oferecidas pelos recursos, como por exemplo vazão máxima disponível num canal de comunicação.

4.2.2 Mecanismos Padrões de Extensão

No desenvolvimento de sistemas tempo-real, existe a necessidade de registrar as restrições temporais identificadas em cada uma das fases de projeto. Estes requisitos de tempo são conceitos não previstos por Booch et al [BOO 99 e BOO 99a] na especificação da linguagem UML e assim precisam ser expressos nos diagramas através dos seus mecanismos padrões de extensão:

- *Constraints*: são restrições representadas por uma expressão textual entre chaves que podem ser associados a diferentes elementos de um diagrama. Várias linguagens podem ser utilizadas tais como notação matemática, linguagem de programação, linguagem específica para restrições como *Object Constraint Language* (OCL) [OMG 97], pseudocódigo ou uma linguagem natural informal. Um limite de tempo poderia ser representado como: {*duration* < 500 ms};

[§§§] Uma mensagem corresponde ao envio de uma informação de um objeto para outro, podendo ser uma sinalização ou a chamada de uma operação [BOO 99a].

- *Tagged values*: são pares de *strings* associados com elementos de um diagrama UML que representam uma propriedade ou atributo deste elemento e seu valor. A prioridade de uma ação poderia ser representada como: {*priority* = 2};
- *Stereotypes*: representam um tipo especial de elemento de modelagem associado ao domínio particular da aplicação que possui restrições especiais em relação à sua utilização ou relacionamento com outros elementos. São representados nos diagramas como textos cercados pelos caracteres « » e colocados próximos ao símbolo do elemento básico de modelagem ou estes elementos podem possuir o seu próprio ícone. Representam um tipo especial de diagrama, de classe, de associação, de mensagem, de pacote *et cetera*. Um objeto ativo que executa uma ação periódica poderia ser representado como «*periodic*», conforme pode ser visto na FIGURA 5.1.

Através dos mecanismos padrões de extensão, é possível incluir restrições temporais nos diagramas UML especificando conceitos como execução periódica, ativações síncronas e assíncronas, limites de tempo de execução, exclusão mútua, prioridade de execução *et cetera*.

4.3 Construções Disponíveis em RTSJ

Conforme apresentado no capítulo anterior, a especificação *The Real-Time Specification for Java* desenvolvida pelo *Java Expert Group* cobre uma série de áreas de interesse para o desenvolvimento de aplicações com restrições temporais. Segue um breve resumo das classes que serão consideradas nesta proposta de mapeamento – maiores informações podem ser encontradas em Bollella et al [BOL 2000].

4.3.1 AsyncEvent

Um objeto *AsyncEvent* representa algo que pode acontecer, como uma sinalização POSIX, uma interrupção de hardware ou um evento de software como a entrada de um avião numa determinada região e quando um destes eventos ocorre, os *handlers* associados são executados.

4.3.2 AsyncEventHandler

Uma instância desta classe é similar a uma *thread*, quando um evento é disparado, os métodos *handleAsyncEvent* dos *handlers* associados são escalonados. A diferença é que um *AsyncEventHandler* está associado a instâncias de *ReleaseParameters*, *SchedulingParameters* e *MemoryParameters* que ajustam a efetiva execução do *handler* associado ao objeto *AsyncEvent* disparado.

4.3.3 AsynchronouslyInterruptedException

Esta exceção é gerada quando uma *thread* é interrompida de forma assíncrona. Um método indica para máquina virtual Java sua concordância em ser interrompido a qualquer momento quando inclui esta exceção em sua cláusula *throws*.

4.3.4 Interruptible

A implementação desta interface permite a uma classe a execução de um método nesta interface que execute dentro de um limite de tempo, ou esta execução será interrompida por uma *AsynchronouslyInterruptedException* (AIE).

4.3.5 MonitorControl

É uma classe abstrata da qual derivam classes que implementam o comportamento de monitores usados pelos comandos e métodos *synchronized* no sistema. Conforme a implementação, precisam suportar a política de herança de prioridade ou emulação da prioridade definida como teto (*priority ceiling emulation*).

4.3.6 PeriodicParameters

É uma classe derivada de *ReleaseParameters* que condiciona a execução periódica de objetos escalonáveis associados (instâncias da classe *Schedulable*), definindo que o método *waitForNextPeriod* será desbloqueado no início de cada período.

4.3.7 PeriodicTimer

Esta classe representa *Timers* que disparam um *AsyncEvent* com um intervalo constante levando à execução do *AsyncEventHandler* associado. Este intervalo de tempo pode ser representado em termos de *RelativeTime* ou *RationalTime*.

4.3.8 PriorityCeilingEmulation

Esta classe é um *MonitorControl* que especifica a política de emulação da prioridade definida como teto (*priority ceiling emulation*) para os monitores usados pelos comandos e métodos *synchronized* no sistema. Nesta política, quando uma *thread* entra no monitor, sua prioridade efetiva é elevada para prioridade teto, sendo restaurada para a prioridade prévia ao sair do monitor.

4.3.9 PriorityInheritance

Esta classe é um *MonitorControl* que especifica a política de herança de prioridade para os monitores usados pelos comandos e métodos *synchronized* no sistema. Nesta política, quando uma *thread* deseja entrar no monitor, a prioridade da *thread* que está correntemente dentro do monitor é elevada para a prioridade da *thread* que está tentando entrar, sendo restaurada para a prioridade prévia ao sair do monitor.

4.3.10 PriorityParameters

Instâncias desta classe devem ser atribuídas a *threads* que são manipuladas por escalonadores que usam um simples inteiro para determinar a ordem de execução.

4.3.11 RealtimeThread

A classe *RealtimeThread* estende a classe *Thread* incluindo em seu construtor parâmetros que permitem informar ao sistema suas demandas de processamento e temporais. Por exemplo: *ReleaseParameters* definem as condições de liberação da *thread* para execução; *SchedulingParameters* contém valores para definição da elegibilidade de execução, como prioridade da *thread*; *MemoryParameters* que podem ser utilizados para controle de admissão da *thread* no escalonador.

4.3.12 RelativeTime

Uma instância de *RelativeTime* representa um ponto no tempo que é relativo a outro instante de tempo. Representa um intervalo de tempo com resolução de nanossegundos.

4.3.13 Timed

Esta classe cria um escopo durante a execução de uma *RealtimeThread* que pode ser interrompido por uma *AsynchronouslyInterruptedException* ao ser atingido o limite de tempo.

4.3.14 WaitFreeReadQueue

Esta classe implementa uma fila que não bloqueia quando está vazia, simplesmente retorna *null*. As classes que implementam filas livres de espera (*wait-free queue*) permitem a comunicação e sincronização entre instâncias de *RealtimeThread* e *Thread*, evitando alguma eventual influência do *GarbageCollector* sobre a *RealtimeThread*.

4.4 Mapeamento

Na TABELA 4.1 é apresentada uma relação entre os conceitos tempo-real que serão considerados neste trabalho, sua representação em UML e as construções ou classes correspondentes na RTSJ API. A geração da proposta de mapeamento propriamente dita será realizada através de Estudos de Caso que serão desenvolvidos no próximo capítulo abordando os conceitos resumidos na tabela citada conforme o seguinte planejamento:

- O estudo de caso do Piloto Automático Automotivo deve explorar o conceito de execução periódica, sua representação em UML, o desenvolvimento do código Java correspondente utilizando o suporte da RTSJ API e os testes de validação deste mapeamento;
- A implementação de um Controle de Tráfego Aéreo (seção 5.1.2) explorará execuções assíncronas, limite de tempo na recepção de mensagens periódicas e limite de tempo na execução de um bloco de código no âmbito de UML e RTSJ;
- O estudo de caso de um Marca Passo Cardíaco (seção 5.1.3) analisará o limite de tempo numa máquina de estados e execuções assíncronas com passagem de parâmetros através de filas;

- Finalmente o estudo de caso de um Sistema de Telemetria (seção 5.2.1), que aborda o mapeamento parcial de padrões definidos no *UML Profile for Scheduling, Performance, and Time* (UMLSPT) para código Java com suporte da RTSJ API.

TABELA 4.1 – Mapeamento de Conceitos entre UML e RTSJ

Conceito Tempo-Real	Exemplos de representação em UML	Classes relacionadas na RTSJ API
Ativação assíncrona	Mensagens assíncronas ou eventos apresentados em diagramas de interação.	▪ AsyncEvent; ▪ AsyncEventHandler; ▪ AsynchronouslyInterruptedException; ▪ Interruptible; ▪ RealtimeThread; ▪ WaitFreeReadQueue.
Execução periódica****	Mensagens periódicas apresentadas em diagramas de interação.	▪ AsyncEvent; ▪ AsyncEventHandler; ▪ PeriodicParameters; ▪ PeriodicTimer; ▪ RealtimeThread; ▪ RelativeTime.
Execução com limite de tempo	Limites incluídos como *constraints* ou *tagged values* em diagramas de interação.	▪ AsynchronouslyInterruptedException; ▪ Interruptible; ▪ RelativeTime; ▪ Timed.
Exceções	Representadas como eventos em diagramas de máquinas de estado.	▪ AsynchronouslyInterruptedException.
Sincronização	Características apresentadas como *stereotypes* em diagramas de interação.	▪ MonitorControl; ▪ PriorityCeilingEmulation; ▪ PriorityInheritance; ▪ RealtimeThread.
Prioridade de execução	Características apresentadas como *tagged values* em diagramas de interação.	▪ PriorityParameters; ▪ RealtimeThread.
Restrição de *jitter*	Incluídos como *constraints* num diagrama de interação, definindo uma tolerância no período de uma mensagem.	▪ PeriodicParameters; ▪ PeriodicTimer.

4.4.1 Ambiente de Validação do Mapeamento

O livro *The Real-Time Specification for Java* de Bollella et al [BOL 2000], publicado em junho de 2000, corresponde a uma versão preliminar da especificação e conforme sua própria introdução somente estará disponível uma versão final com a liberação de uma implementação de referência. Neste momento, que foi exatamente contemporâneo ao início deste trabalho, não havia nenhuma implementação disponível para a *Application Programming Interface* (API) especificada.

**** Utilizando a RTSJ é possível implementar outros padrões de ativação, mas a execução periódica é diretamente suportada pela API.

Isto motivou o desenvolvimento de uma versão de simulação da API com o objetivo de testar o mapeamento proposto. Esta simulação inclui a totalidade de classes e métodos especificados, o que permite a compilação de qualquer programa de exemplo criado para demonstrar as características de implementações reais. Embora sem a pretensão de atender, mas sim demonstrar as restrições temporais previstas na especificação, esta versão permite a execução simulada de exemplos reais, incluindo todos os estudos de caso. O desenvolvimento desta versão de simulação demandou um volume de trabalho de programação e teste bastante significativo[††††], mas que foi fundamental para um melhor conhecimento da API e conseqüente maturidade da proposta de mapeamento. As falhas que foram introduzidas intencionalmente no código dos estudos de caso permitem, através das impressões geradas, demonstrar a correta atuação dos mecanismos tempo-real desta versão da RTSJ API.

Embora atualmente já existam implementações comerciais da RTSJ API [AJI 2002, IBM 2002 e SYS 2002], esta simulação tem como vantagem sua independência de plataforma, permitindo testes e estudos acadêmicos sem ônus. A biblioteca em formato JAR, os códigos fonte, documentações e exemplos podem ser obtidos em [HÖL 2001].

[††††] A API de simulação da RTSJ possui 61 classes, 7615 linhas de código fonte e 105 kbytes de código compilado (*.class), sem contar as rotinas de testes e os estudos de caso que totalizam mais ou menos o mesmo volume de código.

5 Estudos de Caso

Neste capítulo serão estudadas algumas situações típicas que ocorrem em aplicações tempo-real através de diagramas UML com restrições temporais associadas. Cada um destes diagramas foi total ou parcialmente implementado em Java, sendo o atendimento dos requisitos temporais delegado ao suporte fornecido pela RTSJ API definida em [BOL 2000]. Estes exemplos de mapeamento de UML para Java Real-Time foram testados através da utilização da versão de simulação da RTSJ API desenvolvida no âmbito deste mestrado, com o objetivo de validar o mapeamento proposto. No final deste capítulo, são resumidas as soluções de mapeamento nos diferentes tipos de diagramas e as construções Java utilizadas para representar as restrições temporais.

5.1 UML para RTSJ

Os três estudos de caso discutidos nesta seção 5.1 foram selecionados na literatura [DOU 99 e GOM 2000] de forma que as construções Java utilizadas pudessem explorar diferentes características disponíveis no suporte tempo-real da RTSJ API, sendo fundamentados em diagramas UML que incluem restrições temporais conforme extensões propostas por seus autores. Já os estudos de caso da próxima seção são baseados no *UML Profile for Scheduling, Performance, and Time*, perfil proposto pela OMG [OMG 2001], cuja abrangência e generalidade o credenciam como candidato a padrão *"Real-Time UML"*.

5.1.1 Piloto Automático Automotivo

A principal função de um piloto automático automotivo é manter uma determinada velocidade constante num veículo. Este sistema pode também incluir funções auxiliares como acompanhar a distância percorrida após cada tipo de manutenção e alertar o motorista ao atingir os limites previstos ou apresentar a velocidade e consumo médio do veículo ao longo de uma viagem. Para atender estes requisitos, o piloto automático precisa dispor tanto da velocidade instantânea quanto da distância total percorrida, que podem ser calculados pela contagem do número de voltas do eixo de saída da caixa de câmbio e pela variação da contagem no intervalo de tempo entre duas amostragens.

Ao analisar esta aplicação, Gomaa [GOM 2000] projetou a arquitetura reproduzida na FIGURA 5.1. Nesta figura são apresentados elementos que compõem o subsistema *Distance & Speed*, no qual uma tarefa periódica é responsável pela execução dos cálculos de velocidade instantânea e distância total percorrida. O ciclo de 500 ms desta tarefa periódica foi incluído na figura através de anotações entre chaves.

Na implementação deste subsistema procurou-se analisar a representação de tarefas periódicas em Java com suporte da RTSJ API. Neste desenvolvimento foram assumidas as seguintes decisões de projeto:

- O objeto *Distance & Speed* é um objeto ativo da classe *DistanceAndSpeed*, uma subclasse de *RealtimeThread* da RTSJ; o ator *Clock* está incluso na API, e os eventos de tempo gerados pelo mesmo estão implícitos no suporte à execução de *RealtimeThread* periódicas conforme será detalhado a seguir;

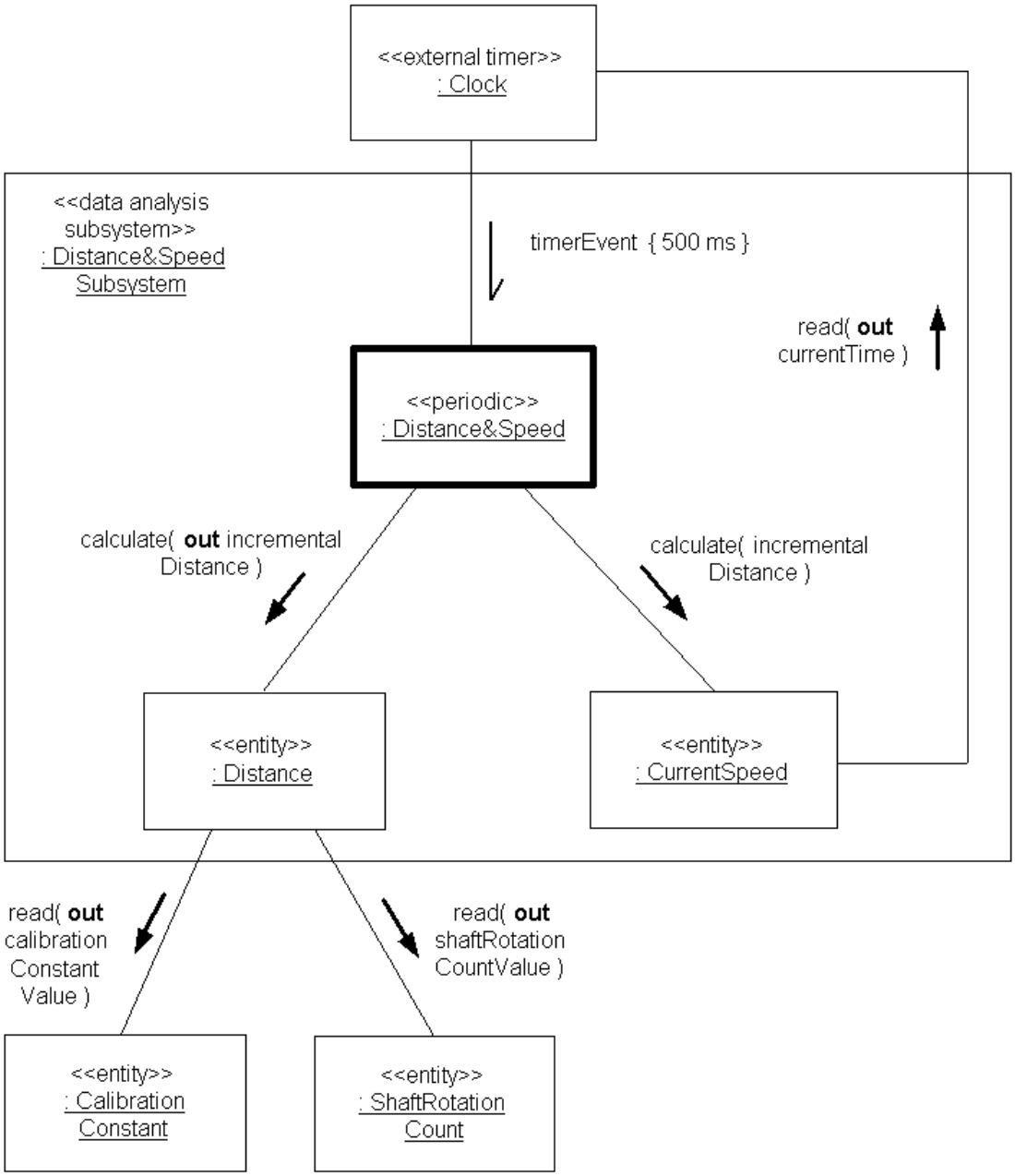


FIGURA 5.1 – *Distance & Speed Subsystem: task architecture* [GOM 2000][‡‡‡‡]

- As entidades[§§§§] *CalibrationConstant*, *ShaftRotationCount* e *Distance* foram implementadas como classes Java com todos membros estáticos (existe apenas uma instância pré-criada de cada classe) sem qualquer dependência da RTSJ; a entidade *CurrentSpeed* foi implementada da mesma forma, mas utiliza a API para obter o horário corrente (*Clock.getRealtimeClock().getTime()*);

[‡‡‡‡] Todos os estudos de caso deste trabalho são casos clássicos em aplicações tempo-real e as respectivas figuras, obtidas na literatura, foram reproduzidas sem alterações.
[§§§§] Entidades são classes cujo papel principal consiste em armazenar dados [GOM 2000].

- A atualização da contagem de voltas do eixo não faz parte do subsistema *Distance & Speed*, assim para simular o movimento do veículo foi criado *Shaft Interface*, um objeto ativo derivado de *RealtimeThread* que atualiza ciclicamente a entidade *ShaftRotationCount*;
- O controle geral da simulação é realizado pela classe *Main* responsável pelo ajuste do valor inicial da constante de calibração e pelo acompanhamento dos valores de distância e velocidade instantânea calculados;
- Procurou-se minimizar os atrasos gerados pelas mensagens de acompanhamento. Ao contrário de enviá-las diretamente para tela, as mesmas são apenas registradas com horário e descarregadas apenas ao final da simulação. Para atender esta necessidade e outras funções auxiliares foi desenvolvida a classe *Util*, incluída também nos demais exemplos Java desta dissertação.

Ao contrário de uma *thread* Java convencional, os objetos escalonáveis (aqueles que implementam a interface *Schedulable*: *RealtimeThread*, *NoHeapRealtimeThread* e *AsyncEventHandler*) podem informar ao escalonador suas condições de liberação através de parâmetros: ou seja, se a liberação para execução será periódica, aperiódica ou esporádica[*****]. O objeto ativo periódico da figura poderia ser mapeado para uma *RealtimeThread* periódica, sendo esta característica definida através da definição dos *ReleaseParameters* da mesma como periódicos.

No construtor da classe *DistanceAndSpeed*, é criado um objeto *PeriodicParameters* com o período de 500 ms recebido como parâmetro da classe *Main* e o mesmo é atribuído ao objeto *RealtimeThread* que está sendo construído, conforme pode ser visto em destaque na FIGURA 5.2.

```
public DistanceAndSpeed( RelativeTime rtimePeriod )
{
    PeriodicParameters periodicParameters =
        new PeriodicParameters( null, rtimePeriod, null, null, null, null††††† );
    setReleaseParameters( periodicParameters );

    start();
}
```

FIGURA 5.2 – Construtor da classe *DistanceAndSpeed*

Após definição dos *PeriodicParameters*, um objeto escalonável deve informar o final de cada período de processamento para o escalonador. Para isto, existe o método *waitForNextPeriod* que quando chamado bloqueia a execução até o início do próximo período, retornando o controle no mesmo ponto do código. Assim, a implementação do método *run* da classe *DistanceAndSpeed* corresponde a um laço no qual são executados os cálculos de distância e velocidade corrente seguidos da espera pelo novo período, conforme pode ser visto em destaque na FIGURA 5.3.

[*****] Um objeto escalonável com liberação esporádica é similar ao aperiódico, mas respeita um intervalo mínimo entre liberações para execução [BOL 2000].
[†††††] O tratamento de *deadline* foi omitido neste primeiro estudo de caso.

```
public void run()
{
    while( m_isRunning ) {
        CurrentSpeed.calculate( Distance.calculate() );
        waitForNextPeriod();
    }
}
```

FIGURA 5.3 – Método *run* da classe *DistanceAndSpeed*

Na saída gerada pelo projeto *Cruise Control* apresentado na FIGURA 5.4, podem ser vistos valores de velocidade e distância monitorados a cada 500 ms, mas deve-se ressaltar que este intervalo entre mensagens depende apenas da freqüência de monitoração da classe *Main*, pois a tarefa *DistanceAndSpeed* realiza apenas os cálculos e não emite qualquer tipo de registro histórico.

```
Main  Dom, 28 Out 2001 20:30:12+582,000000ms Start of simulation
Main  Dom, 28 Out 2001 20:30:42+665,000000ms Calibration Constant 2992
Main  Dom, 28 Out 2001 20:30:43+676,000000ms Speed 120.32km/h Distance 0.033km
Main  Dom, 28 Out 2001 20:30:44+177,000000ms Speed 120.08km/h Distance 0.049km
Main  Dom, 28 Out 2001 20:30:44+678,000000ms Speed 120.08km/h Distance 0.066km
Main  Dom, 28 Out 2001 20:30:45+179,000000ms Speed 117.679km/h Distance 0.082km
Main  Dom, 28 Out 2001 20:30:45+679,000000ms Speed 122.776km/h Distance 0.099km
Main  Dom, 28 Out 2001 20:30:46+180,000000ms Speed 117.73km/h Distance 0.116km
Main  Dom, 28 Out 2001 20:30:46+681,000000ms Speed 122.526km/h Distance 0.133km
Main  Dom, 28 Out 2001 20:30:47+181,000000ms Speed 117.961km/h Distance 0.149km
Main  Dom, 28 Out 2001 20:30:47+692,000000ms Speed 115.174km/h Distance 0.165km
Main  Dom, 28 Out 2001 20:30:48+193,000000ms Speed 117.73km/h Distance 0.182km
Main  Dom, 28 Out 2001 20:30:49+174,000000ms End of simulation
```

FIGURA 5.4 – Resultado da Execução do Projeto *Cruise Control*

5.1.2 Controle de Tráfego Aéreo

Um controle de tráfego aéreo é responsável pela supervisão de aviões deslocando-se nas imediações de um aeroporto através de sinais recebidos de um ou mais radares. Os sinais recebidos dos radares indicam a posição instantânea de cada avião, e através do processamento destes sinais é possível determinar suas velocidades e trajetórias. Muitas vezes as restrições temporais de um sistema são identificadas durante a fase de análise na elaboração de *Use Cases*[‡‡‡‡‡], através da inclusão de anotações entre chaves (UML *Constraints*).

Por exemplo, no *Use Case* que modela este controle de tráfego aéreo, através desta notação poderia ser definido que o sistema precisa receber mensagens periódicas do ator que representa o radar primário num ciclo máximo de 50 ms e o processamento dos sinais dos radares precisa ocorrer num intervalo máximo de 30 ms. As mensagens enviadas pelo ator que representa o radar secundário ou *transponder* também seriam utilizadas no processamento, mas sem qualquer restrição temporal em relação a seu

[‡‡‡‡‡] Um diagrama UML *Use Case* especifica uma seqüência de ações, incluindo variações de seqüência e erros, no qual um sistema, subsistema ou classe realiza uma interação com atores externos [BOO 99a].

recebimento. Na FIGURA 5.5[§§§§§] adaptada de Douglass [DOU 99], é apresentado um diagrama de seqüência no nível de *Use Cases* onde foram incluídas as restrições temporais desta forma.

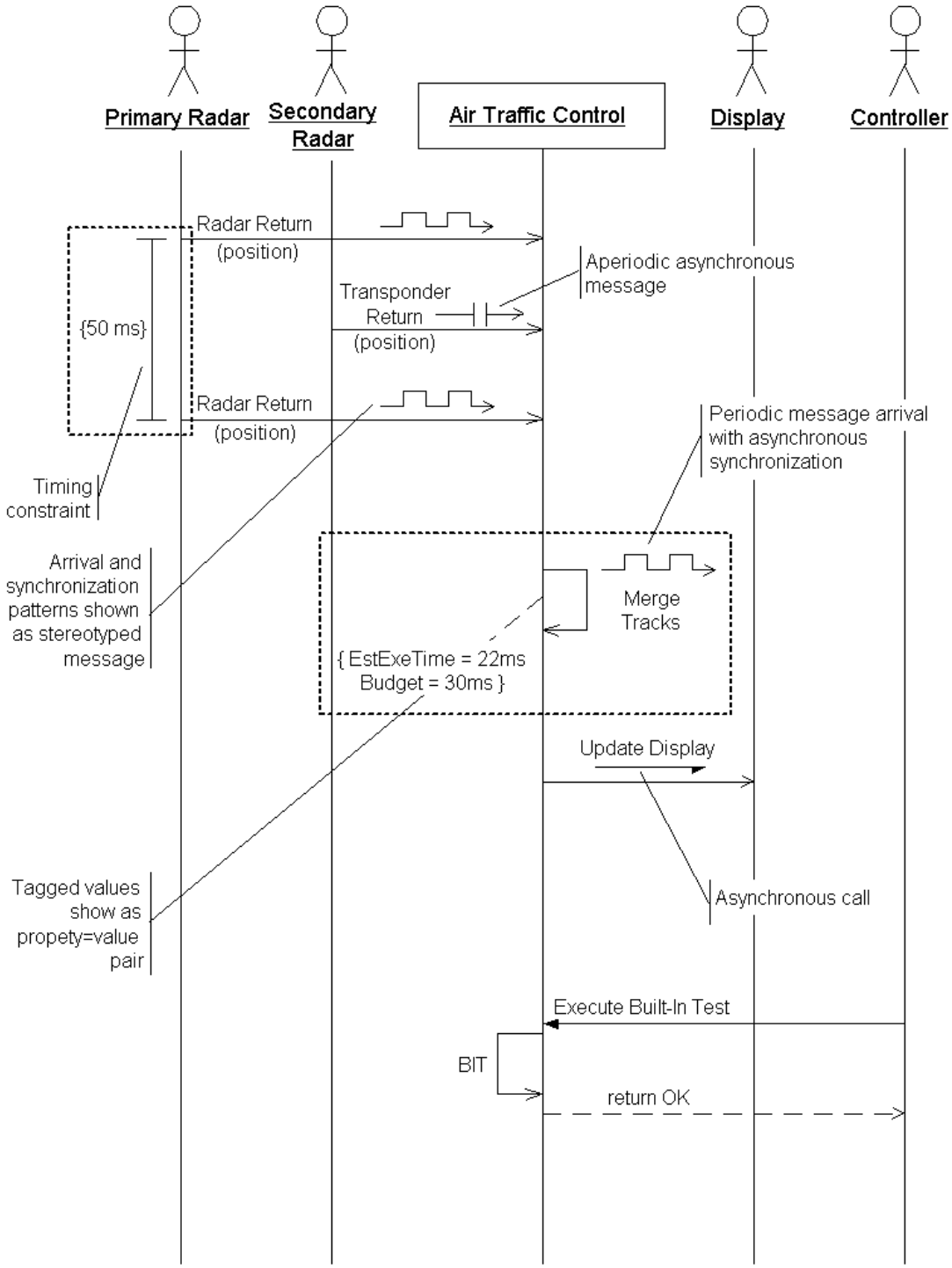


FIGURA 5.5 – *Air Traffic Control Sequence Diagram with Constraints* [DOU 99]

[§§§§§] Os retângulos tracejados apresentadas nesta figura não correspondem à notação UML, tendo sido incluídas apenas para destacar trechos do diagrama que serão relevantes na geração de código.

A situação na qual um objeto aguarda por uma mensagem periódica é bastante comum, sendo necessário prever uma solução baseada na RTSJ API para a mesma, sendo este o principal objetivo desta implementação. Além disso, este exemplo de controle de tráfego aéreo também envolveu o mapeamento para API de outro tipo de problema: o caso de um objeto ativo que precisa realizar um processamento dentro de um limite de tempo. Neste desenvolvimento foram assumidas as seguintes decisões de projeto:

- O objeto *Air Traffic Control* é um objeto ativo da classe *AirTrafficModel*, uma subclasse de *RealtimeThread* da RTSJ; o controle geral da simulação, as ações realizadas pelos atores *Primary Radar*, *Secondary Radar*, *Display* e *Controller* ficaram a cargo de uma classe denominada *Main*;
- As mensagens assíncronas para o objeto *Air Traffic Control* foram realizadas através do disparo de eventos da classe *AirTrafficEventHandler* enquanto as chamadas e mensagens assíncronas enviadas pelo sistema para os atores foram simplesmente simuladas;
- A mensagem síncrona *Execute Built-In Test* enviada pelo ator *Controller* para o objeto *Air Traffic Control* foi implementada como uma chamada direta ao método *executeBuiltInTest* da classe *AirTrafficModel*;
- Da mesma forma que nos demais exemplos Java deste trabalho, o projeto *Air Traffic Control* utiliza o suporte da classe *Util* no registro das mensagens de acompanhamento da simulação para postergar para o final da execução o tempo despendido na escrita na tela.

Para enviar mensagens assíncronas para o sistema, o ator *Secondary Radar* dispara eventos da classe *AirTrafficEventHandler* que escalonam a execução do *handler* correspondente conforme código mostrado na FIGURA 5.6. Esta classe é subclasse de *AsyncEventHandler* da RTSJ e inclui como membro um evento da classe *AsyncEvent* da API, tendo sido desenvolvida para permitir a passagem assíncrona de um parâmetro, a posição ou distância do avião detectada pelos radares primário ou secundário.

```
private AirTrafficEventHandler
    m_ateTransponder = new AirTrafficEventHandler() {
        public void handleAirTrafficEvent( long lnPosition ) {
            Util.saveMsg( "Transponder return " +
                                          lnPosition + " m" );
            doTimedMergeTracks( lnPosition );
        }
    };

public void sendTransponderReturn( long lnPosition )
{
    m_ateTransponder.fire( lnPosition );
}
```

FIGURA 5.6 – Mensagens Assíncronas do Radar Secundário

A solução adotada para acompanhar o limite no intervalo entre as mensagens assíncronas enviadas pelo ator *Primary Radar* para o sistema está apresentada em destaque na FIGURA 5.7. Foi criada uma instância da classe *PeriodicTimer* RTSJ

construída com intervalo igual ao limite a ser controlado (1 segundo / 20 = 50 ms), e o *handler m_asyncRadarTimeout* associado a este *timer* é responsável pelo tratamento de falhas no radar primário.

```
private AsyncEventHandler
    m_asyncRadarTimeout = new AsyncEventHandler() {
        public void handleAsyncEvent() {
            handleRadarFailure();
        }
    };
}

private void setRate( int iReturnForSecond ) {
    RelativeTime rtimePeriod = new RelativeTime( 1000 / iReturnForSecond, 0 );
    m_ptimer = new PeriodicTimer(null,rtimePeriod,m_asyncRadarTimeout);
    m_ptimer.start();
}
```

FIGURA 5.7 – Verificação de Falha na Sinalização do Radar Primário

Para enviar mensagens assíncronas para o sistema, o ator *Primary Radar* também dispara eventos da classe *AirTrafficEventHandler*, mas a cada disparo é preciso realizar uma sincronização e reescalonamento do *timer* para o instante corrente mais o intervalo de controle. Desta forma a rotina de tratamento de falha do radar primário será executada apenas se for atingido o limite de tempo entre sinalizações deste radar sem uma nova sincronização e reescalonamento. Na FIGURA 5.8 é apresentado em destaque o trecho de código correspondente.

```
private AirTrafficEventHandler
    m_ateRadar = new AirTrafficEventHandler() {
        public void handleAirTrafficEvent( long lnPosition ) {
            doTimedMergeTracks( lnPosition );
        }
    };

public void sendRadarReturn( long lnPosition )
{
    AbsoluteTime now = Util.now();
    m_ptimer.reschedule( now.add( m_ptimer.getInterval() ) );
    m_ateRadar.fire( lnPosition );
}
```

FIGURA 5.8 – Mensagens Assíncronas Periódicas do Radar Primário

Conforme previsto na FIGURA 5.5, o processamento dos sinais dos radares implementado pelo método *mergeTracks* precisa ser executado num tempo máximo de 30 ms, assim os *handlers* de *AirTrafficEventHandler* não chamam este método diretamente. A solução adotada utiliza o método *doTimedMergeTracks* sendo apresentada em destaque na FIGURA 5.9 e foi baseada na classe *Timed* da RTSJ que consegue controlar a execução limitada em tempo de um objeto ativo que implementa a interface *Interruptible*. No projeto *Air Traffic Control* quando o processamento é abortado, o método *interruptAction* simplesmente apresenta uma mensagem informativa.

```
public void doTimedMergeTracks( final long lnPosition )
{
    Interruptible interruptibly = new Interruptible() {
        public void run( AsynchronouslyInterruptedException aie )
                throws AsynchronouslyInterruptedException {
            mergeTracks( lnPosition );
        }

        public void interruptAction(
                AsynchronouslyInterruptedException aie ) {
            Util.saveMsg( "Execution timeout in mergeTracks" );
        }
    };
    new Timed( new RelativeTime( 30/*ms*/, 0 ) ).
                          doInterruptible( interruptibly );
}
```

FIGURA 5.9 – Método *doTimedMergeTracks* da classe *AirTrafficModel*

Conforme definição da RTSJ, apenas instâncias da *RealtimeThread* podem utilizar esta solução, ou seja, o método *doTimedMergeTracks* não pode ser executado diretamente por uma instância de *AsyncEventHandler*. Na FIGURA 5.10, é ressaltado em destaque a construção utilizada na classe *AirTrafficEventHandler* para atender esta definição, que cria uma *RealtimeThread* auxiliar para executar o método *handleAirTrafficEvent* que é sobrecarregado pelos usuários desta classe (FIGURA 5.8).

```
public void handleAsyncEvent()
{
    final long lnPosition = ((Long) m_vector.remove(0)).longValue();

    RealtimeThread thr = new RealtimeThread() {
         public void run() {
             handleAirTrafficEvent( lnPosition );
         }
     };

    // Atribui a RealtimeThread o nome do handler e dispara a mesma
    thr.setName( getName() );
    thr.start();
}
```

FIGURA 5.10 – Método *handleAsyncEvent* da classe *AirTrafficEventHandler*

A FIGURA 5.11 apresenta a saída gerada pelo projeto *Air Traffic Control* ao ser executado com o suporte da versão de simulação da RTSJ API. Pode ser observado que a cada vez que o limite de 50 ms entre sinalizações do radar primário não é respeitado, o *handler* correspondente (*Air.A)* apresenta uma mensagem (*Radar signal is failure*) informando esta situação[******].

[******] Estas falhas foram intencionalmente introduzidas no código da classe *Main* que implementa a simulação dos atores *Primary Radar* e *Secondary Radar* para demonstrar a correta atuação dos mecanismos tempo-real da RTSJ API. Esta prática também foi adotada nos demais exemplos Java desenvolvidos neste mestrado.

```
Main  Sex, 12 Out 2001 14:51:32+915,000000ms Start of simulation
Air.R Sex, 12 Out 2001 14:51:33+095,000000ms First signal 6117 m
Air   Sex, 12 Out 2001 14:51:33+115,000000ms Velocity 13000 m/s
Air   Sex, 12 Out 2001 14:51:33+146,000000ms Velocity 4612 m/s
Air   Sex, 12 Out 2001 14:51:33+176,000000ms Velocity 5200 m/s
Air.A Sex, 12 Out 2001 14:51:33+216,000000ms Radar signal is failure
Main  Sex, 12 Out 2001 14:51:33+226,000000ms Radar failure
Air   Sex, 12 Out 2001 14:51:33+236,000000ms Velocity 5533 m/s
Air   Sex, 12 Out 2001 14:51:33+266,000000ms Velocity 6066 m/s
Air   Sex, 12 Out 2001 14:51:33+296,000000ms Velocity 6500 m/s
Air   Sex, 12 Out 2001 14:51:33+326,000000ms Velocity 6933 m/s
Air.T Sex, 12 Out 2001 14:51:33+326,000000ms Transponder return 7532 m
Air   Sex, 12 Out 2001 14:51:33+336,000000ms Velocity 6900 m/s
Air   Sex, 12 Out 2001 14:51:33+356,000000ms Velocity 7600 m/s
Air   Sex, 12 Out 2001 14:51:33+386,000000ms Velocity 7800 m/s
Air   Sex, 12 Out 2001 14:51:33+416,000000ms Velocity 8233 m/s
Air   Sex, 12 Out 2001 14:51:33+446,000000ms Velocity 8666 m/s
Air   Sex, 12 Out 2001 14:51:33+476,000000ms Velocity 9100 m/s
Air   Sex, 12 Out 2001 14:51:33+506,000000ms Velocity 9533 m/s
Air   Sex, 12 Out 2001 14:51:33+536,000000ms Velocity 9966 m/s
Air.A Sex, 12 Out 2001 14:51:33+576,000000ms Radar signal is failure
Air   Sex, 12 Out 2001 14:51:33+596,000000ms Velocity 10300 m/s
Air   Sex, 12 Out 2001 14:51:33+626,000000ms Velocity 10833 m/s
Air   Sex, 12 Out 2001 14:51:33+656,000000ms Velocity 11266 m/s
Main  Sex, 12 Out 2001 14:51:33+676,000000ms Radar ok
Air   Sex, 12 Out 2001 14:51:33+686,000000ms Velocity 11700 m/s
Air.T Sex, 12 Out 2001 14:51:33+706,000000ms Transponder return 11028 m
Air   Sex, 12 Out 2001 14:51:33+716,000000ms Velocity 3766 m/s
Main  Sex, 12 Out 2001 14:51:33+736,000000ms End of simulation
```

FIGURA 5.11 – Resultado da Execução do Projeto *Air Traffic Control*

5.1.3 Marca Passo Cardíaco

A função básica de um marca passo cardíaco é acompanhar o ritmo de batimentos do coração atuando no caso de atraso nas contrações atrioventriculares, ou seja, após um tempo limite entre contrações consecutivas, o equipamento gera um pulso elétrico para ajustar o ritmo cardíaco. No *Use Case* que modela um marca passo cardíaco, através da inclusão de anotações entre chaves pode ser definido um tempo limite de 800 ms na espera de uma mensagem do ator Coração a ser recebida pelo sistema. Ao realizar o refinamento deste cenário, o sistema é substituído por objetos, são incluídas as mensagens internas e detalhados os estados destes objetos, conforme diagrama de seqüência da FIGURA 5.12[††††††] reproduzida de Douglass [DOU 99].

Das restrições temporais representadas no diagrama, a duração de 5 ms do pulso elétrico foi mapeada diretamente para uma chamada de *Thread.sleep*, assim o objetivo da implementação deste cenário em Java foi analisar como representar com a RTSJ API a outra restrição temporal, a execução de um estado com limite de tempo (800 ms). Dentro deste objetivo foram assumidas as seguintes decisões de projeto:

- Os objetos *Atrial Model* e *Ventricular Model* são objetos ativos da classe *HeartModel*, uma subclasse de *RealtimeThread* da RTSJ; o controle geral da simulação, as ações realizadas pelo objeto *Comm. Gnome* e pelos atores *Programmer* e *Heart* ficaram a cargo da classe principal, denominada de *Main*;

[††††††] Os retângulos tracejados apresentadas nesta figura não correspondem à notação UML, tendo sido incluídas apenas para destacar trechos do diagrama que serão relevantes na geração de código.

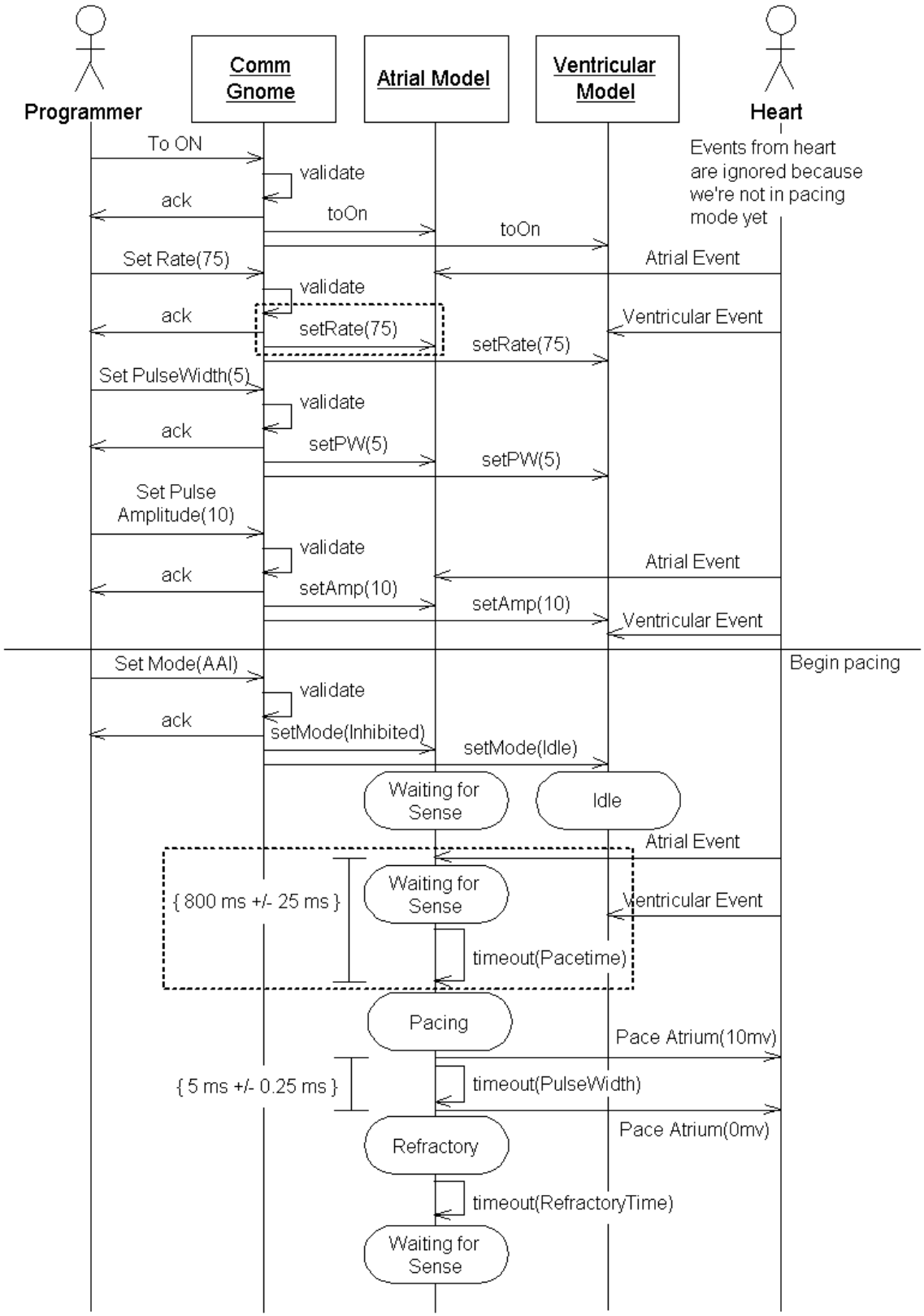


FIGURA 5.12 – *Pace the Heart in AAI*[‡‡‡‡‡‡] *Mode (Object Level)* [DOU 99]

[‡‡‡‡‡‡] Num marca passo em modo AAI, o ajuste de ritmo e a monitoração são referentes à aurícula. Quando é sentida uma contração não estimulada na aurícula, o pulso elétrico é desabilitado.

- Todas as sinalizações e chamadas assíncronas para os objetos *Atrial Model* e *Ventricular Model* foram realizadas através de filas[§§§§§§] da classe *HeartQueue*, enquanto que as chamadas assíncronas para o ator *Heart* foram simplesmente simuladas; estas filas para envio de mensagens para instâncias de *RealtimeThread* utilizam o suporte da RTSJ API fornecido através da classe *WaitFreeReadQueue*;
- O projeto *Pace Heart* também utiliza o suporte da classe *Util* no registro das mensagens de acompanhamento da simulação, postergando a apresentação destas para o final da execução, evitando desta forma a influência do tempo de escrita na tela.

Umas das formas previstas na RTSJ para executar uma determinada ação limitada no tempo é baseada na utilização da classe *Timed*. Para que um determinado método que não concluiu seu processamento no tempo limite previsto possa ser abortado, o mesmo precisa sinalizar sua concordância para a JVM ao incluir em sua implementação a cláusula *throws AsynchronouslyInterruptedException*[*******].

Portanto, para adotar esta solução neste exemplo Java em particular, a leitura da fila de mensagens recebidas pela classe *HeartQueue* precisa ser implementada por um método compatível. Como as classes da RTSJ não fornecem diretamente um método de leitura de filas com esta característica, foi necessário realizar um teste cíclico do método *read* da classe *WaitFreeReadQueue* que fornece uma leitura livre de bloqueio, conforme pode ser visto em destaque na FIGURA 5.13[†††††††].

```
public Message read() throws AsynchronouslyInterruptedException
{
    Object obj;

    while( (obj = m_queue.read()) == null )
        RealtimeThread.interruptibleSleep(
            LOOP_TIME_WAITING_EVENT );

    return (Message) obj;
}
```

FIGURA 5.13 – Método *read* da classe *Heart Queue*

Na FIGURA 5.12, foi ressaltada através de retângulos tracejados a restrição temporal de 800 ms e a chamada ao método *setRate* com o parâmetro correspondente de 75 pulsos por minuto (60 s / 75 = 800 ms). Assim, na solução adotada neste exemplo, este limite de tempo de execução é um parâmetro do construtor da classe *Timed*.

[§§§§§§] Esta implementação também poderia utilizar chamadas diretas de métodos (na parametrização dos modelos cardíacos) e sinalizações assíncronas sem argumentos (como eventos do coração), mas esta solução mais geral no permite estudar mensagens parametrizadas com limite de tempo.

[*******] No projeto *Air Traffic Control* por exemplo, o método *mergeTracks* sinaliza sua concordância com interrupções assíncronas ao incluir a cláusula *throws* necessária.

[†††††††] Para efeitos de análise, deve-se considerar que o método *interruptibleSleep* da classe *RealtimeThread* é exatamente o mesmo que o método *sleep* de sua classe básica *Thread*. Este método não previsto é necessário apenas com a versão de simulação da RTSJ API.

Conforme pode ser visto em destaque na FIGURA 5.14, o método *setRate* da classe *HeartModel* define o tempo máximo de espera pela contração cardíaca criando um objeto da classe *Timed*.

```
private void setRate( int iHitForMinute )
{
    if( iHitForMinute < 30 || iHitForMinute > 120 )
       throw new IllegalArgumentException( "Illegal heart rate" );

    m_lnTimeoutRefractoryTime = 60000 / iHitForMinute / 2;

    m_timedWaitingForSense =
         new Timed( new RelativeTime(60000/iHitForMinute, 0) );
}
```

FIGURA 5.14 – Método *setRate* da classe *HeartModel*

A execução do método *doWaitingForSense* da classe *HeartModel* corresponde ao estado *Waiting for Sense* de sua máquina de estados. No diagrama da FIGURA 5.12 este estado encontra-se explícito no objeto *Atrial Model*, indicando o momento no qual está aguardando pelas mensagens de contrações cardíacas. Conforme destaques na FIGURA 5.15, este método e todos os demais métodos intermediários chamados pelo mesmo até alcançar o método *read* da classe *HeartQueue* incluem a cláusula *throws AsynchronouslyInterruptedException*, podendo ser interrompidos de forma assíncrona.

```
private boolean handleQueue() throws
    AsynchronouslyInterruptedException
{
    HeartQueue.Message msg = m_queue.read();

    // omitido restante do código deste método ...
}

public void doWaitingForSense() throws
    AsynchronouslyInterruptedException
{
    Util.saveMsg( "WaitingForSense" );

    if( handleQueue() )
        Util.saveMsg( "Atrial event received" );
}
```

FIGURA 5.15 – Método *doWaitingForSense* da classe *HeartModel*

Um objeto da classe *Timed* consegue controlar a execução limitada em tempo de objetos que implementam a interface *Interruptible*. Assim, para que o objeto criado pelo método *setRate* possa controlar a execução do método *doWaitingForSense*, foi adotada uma estrutura que é apresentada na FIGURA 5.16. Quando a leitura da fila de mensagens é abortada, o tratamento da interrupção assíncrona (implementada pelo método *interruptAction* na mesma figura) já realiza a troca da máquina de estados da classe *HeartModel* para o estado PACING, conforme previsto na FIGURA 5.12.

```
public void doTimedWaitingForSense()
{
    Interruptible interruptibly = new Interruptible() {
        public void run(AsynchronouslyInterruptedException aie)
        throws AsynchronouslyInterruptedException {
            doWaitingForSense();
        }

        public void interruptAction(
            AsynchronouslyInterruptedException aie ) {
                m_iState = PACING;
        }
    };

    m_timedWaitingForSense.doInterruptible( interruptibly );
}
```

FIGURA 5.16 – Método *doTimedWaitingForSense* da classe *HeartModel*

O limite máximo de 800 ms entre contrações cardíacas (correspondente à freqüência cardíaca selecionada de 75 batimentos por minuto) pode ser observado na FIGURA 5.17 que representa a saída gerada pelo projeto *Pace Heart*, ao ser executado com o suporte da versão de simulação da RTSJ API.

```
Gnome  Seg, 10 Set 2001 17:50:14+839,000000ms Start of simulation
Atrial Seg, 10 Set 2001 17:50:14+979,000000ms toOn
Ventri Seg, 10 Set 2001 17:50:14+979,000000ms toOn
Ventri Seg, 10 Set 2001 17:50:18+975,000000ms toMode(Idle)
Atrial Seg, 10 Set 2001 17:50:18+975,000000ms toMode(Inhibited)
Atrial Seg, 10 Set 2001 17:50:19+005,000000ms WaitingForSense
Atrial Seg, 10 Set 2001 17:50:19+385,000000ms Atrial event received
Atrial Seg, 10 Set 2001 17:50:19+385,000000ms WaitingForSense
Atrial Seg, 10 Set 2001 17:50:19+976,000000ms Atrial event received
Atrial Seg, 10 Set 2001 17:50:19+986,000000ms WaitingForSense
Atrial Seg, 10 Set 2001 17:50:20+657,000000ms Atrial event received
Atrial Seg, 10 Set 2001 17:50:20+657,000000ms WaitingForSense
Atrial Seg, 10 Set 2001 17:50:21+458,000000ms Pacing Atrial(10 mv)
Atrial Seg, 10 Set 2001 17:50:21+468,000000ms Pacing Atrial(0 mv)
Atrial Seg, 10 Set 2001 17:50:21+468,000000ms Start refractory
Atrial Seg, 10 Set 2001 17:50:21+869,000000ms Finish refractory
Atrial Seg, 10 Set 2001 17:50:21+869,000000ms WaitingForSense
Atrial Seg, 10 Set 2001 17:50:22+349,000000ms Atrial event received
Atrial Seg, 10 Set 2001 17:50:22+349,000000ms WaitingForSense
Atrial Seg, 10 Set 2001 17:50:22+960,000000ms Atrial event received
Atrial Seg, 10 Set 2001 17:50:22+960,000000ms WaitingForSense
Gnome  Seg, 10 Set 2001 17:50:24+152,000000ms End of simulation
```

FIGURA 5.17 – Resultado da Execução do Projeto *Pace Heart*

5.2 UMLSPT para RTSJ

Alguns autores defendiam a inclusão de novos conceitos na UML para permitir expressar restrições temporais (como *Timing Diagrams* [DOU 99]), enquanto outros empregavam os próprios mecanismos padrões de extensão da UML (*stereotypes*, *tagged values* e *constraints*) para modelar aplicações tempo-real [GOM 2000]. Mesmo neste último caso, os modelos gerados estariam vinculados às convenções particulares do método proposto por um autor específico. Para preencher esta lacuna, foi proposto o documento *UML Profile for Scheduling, Performance, and Time* (UMLSPT) criado por especialistas da OMG [OMG 2001] para análise e projeto de sistemas tempo-real.

Um sistema de tempo-real precisa apresentar um comportamento temporal previsível, isto é, a reação aos eventos precisa ser quantificada e conhecida com antecedência. O objetivo do perfil UMLSPT é utilizar os mecanismos de extensão já existentes na UML para caracterizar o determinismo temporal de um sistema que está sendo modelado. O perfil está estruturado em diversos modelos, nos quais são padronizados conceitos como: instante e padrão de ativação; duração, pior caso e tempo limite de execução; prioridade, controle de acesso *et cetera*.

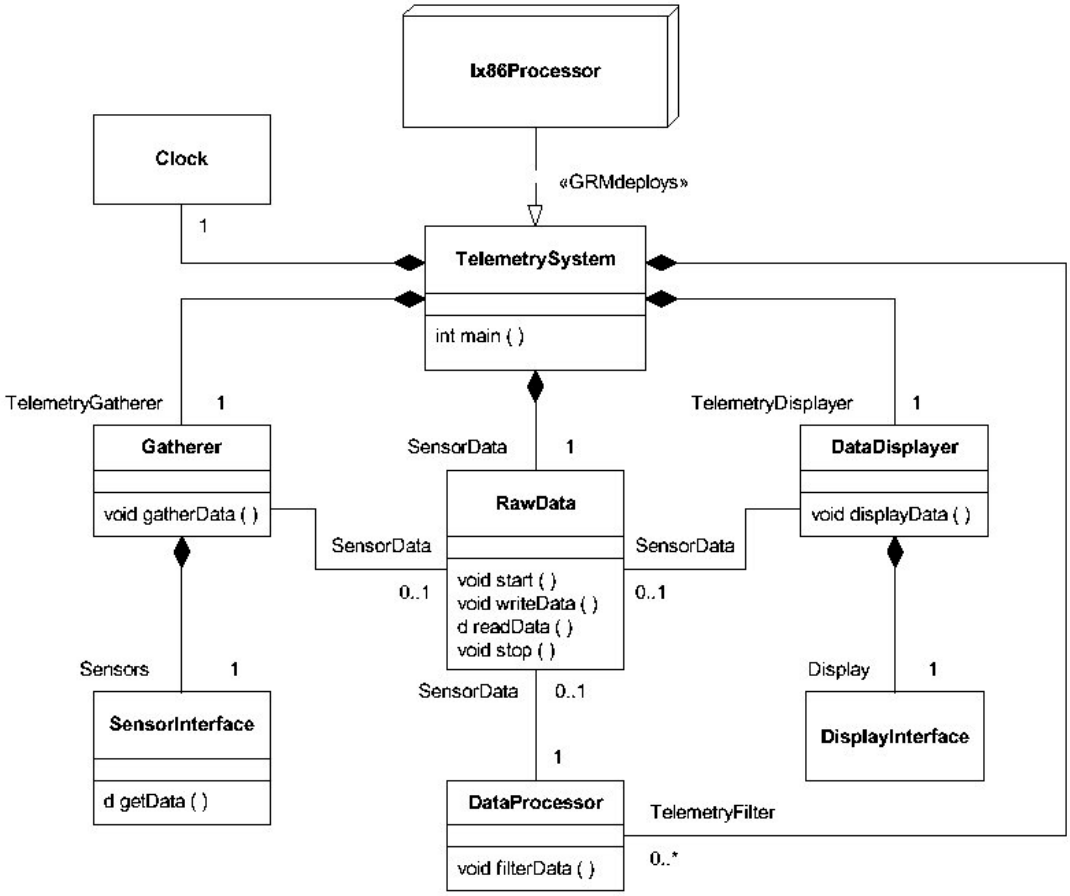


FIGURA 5.18 – Diagrama de Classes de um Sistema de Telemetria [OMG 2001]

5.2.1 Sistema de Telemetria

Na FIGURA 5.18 é apresentado o diagrama de classes do sistema de telemetria, empregado pelo modelo de escalonabilidade do perfil UMLSPT para exemplificar os conceitos padronizados no próprio, sendo utilizado neste texto para demonstrar um mapeamento para código Java com suporte da RTSJ API.

Uma possível aplicação para este sistema de telemetria poderia ser a monitoração do nível de água no lago de uma represa. As informações de diferentes sensores de nível distribuídos ao longo das margens do lago poderiam ser periodicamente consultadas através de ondas de rádio e armazenadas numa base de dados. Através de critérios estatísticos, os níveis de água coletados de sensores não calibrados poderiam ser eliminados do conjunto de dados. O sistema também incluiria funções para apresentação cíclica dos resultados válidos.

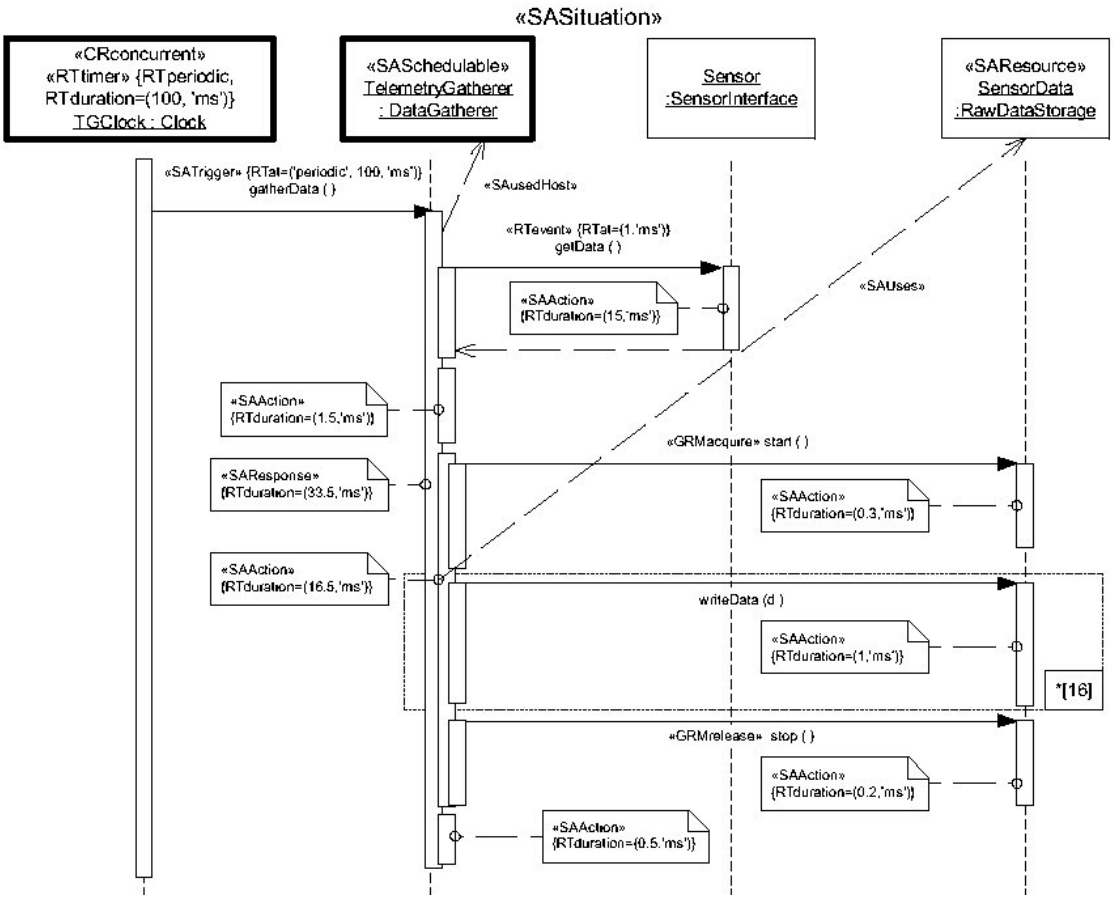


FIGURA 5.19 – Diagrama de Seqüência de um Sistema de Telemetria [OMG 2001]

Nas FIGURAS 5.19 e 5.20 também obtidas de [OMG 2001] são apresentados diagramas de seqüência e colaboração, nos quais o comportamento temporal do sistema de telemetria é expresso através de *stereotypes*, *tagged values* e *constraints* correspondentes aos conceitos propostos no modelo de escalonabilidade do perfil UMLSPT. Devido ao largo espectro dos requisitos temporais abordados neste perfil, este trabalho foi restrito à discussão de algumas idéias preliminares que viabilizem a

tradução para RTSJ, acompanhadas do estudo de caso do exemplo do sistema de telemetria onde estas idéias são validadas:

- O mapeamento do perfil UMLSPT para RTSJ deve utilizar o suporte fornecido pelo pacote "*javax.rtuml*", que já implementa em Java alguns dos conceitos mais comuns necessários neste mapeamento;
- Os *stereotypes* do UMLSPT, quando aplicados a classes que estão sendo modeladas, correspondem a classes básicas do pacote "*javax.rtuml*" das quais as primeiras devem ser derivadas. As classes básicas deste pacote já incluem como atributos todos os *tags* previstos no perfil para os respectivos *stereotypes*. Deste conjunto de *tags*, os referenciados no modelo correspondem a parâmetros de um construtor adequado de uma classe do pacote "*javax.rtuml*";
- Os *stereotypes* aplicados a métodos de classes de um modelo correspondem a métodos implementados em classes do pacote "*javax.rtuml*" que as classes modeladas devem estender, utilizando o suporte fornecido na implementação dos respectivos métodos.

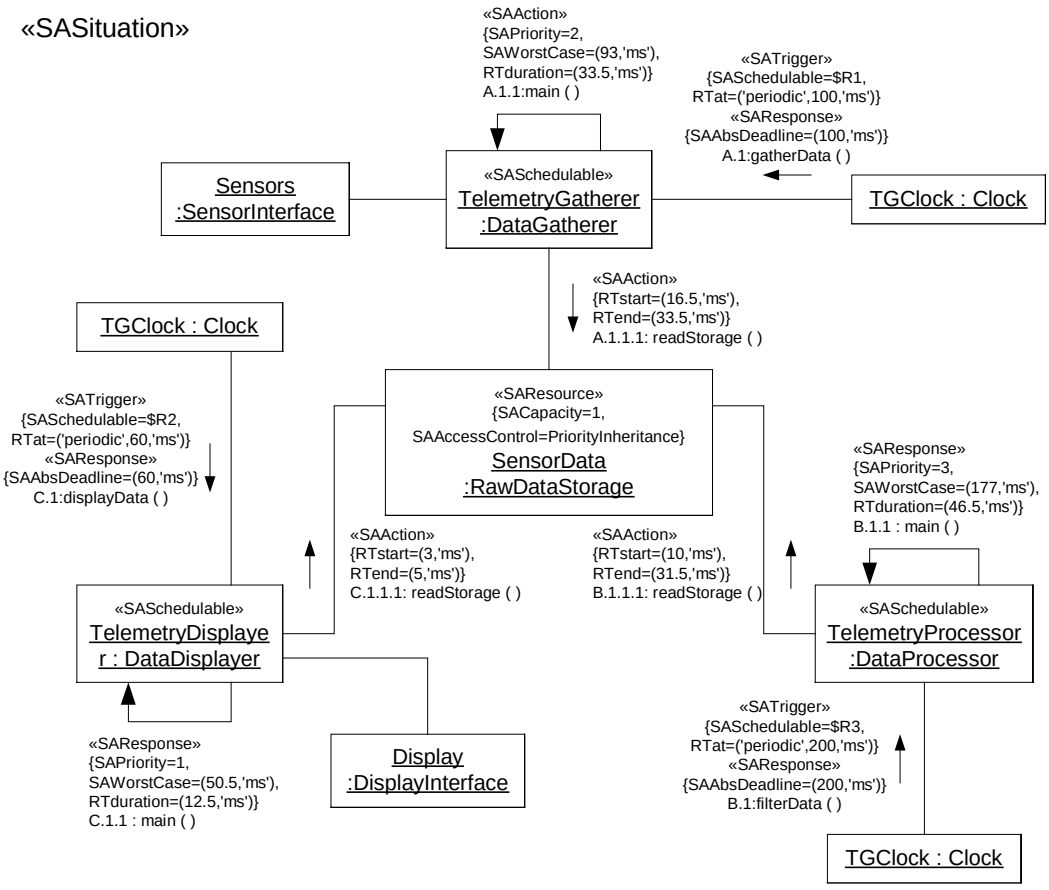


FIGURA 5.20 – Diagrama de Colaboração de um Sistema de Telemetria [OMG 2001]

Nos diagramas do sistema de telemetria apresentados nas FIGURA 5.19 e 5.20, o objeto *SensorData* da classe *RawDataStorage* representa a entidade responsável por manter os

dados de telemetria, sendo utilizado pelos demais objetos para armazenamento e consulta. Este objeto é modelado como um tipo de recurso protegido, no qual os acessos concorrentes seguem uma disciplina, delimitando seu acesso com chamadas aos métodos *start* e *stop*. A aplicação do *stereotype* «SAResource» neste objeto com os *tags SACapacity=1* e *SAAccessControl=PriorityInheritance* corresponde ao trecho de código apresentado na FIGURA 5.21, no qual a classe *RawDataStorage* é derivada de uma classe do pacote "*javax.rtuml*" que implementa o conceito de recurso protegido.

```
public class RawDataStorage extends javax.rtuml.SAResource {

    public RawDataStorage() {
        super( 1/*SAResource.SACapacity*/,
           SAAccessControl.PriorityInheritance/*SAResource.SAAccessControl*/ );
    }

    // ...
}
```

FIGURA 5.21 – Construtor da classe *RawDataStorage*

Os *stereotypes* «GRMacquire» e «GRMrelease» representam a execução das operações de aquisição e liberação de um recurso protegido. Quando aplicados aos métodos *start* e *stop* correspondem ao código da FIGURA 5.22 que utiliza o suporte fornecido pela mesma classe básica para implementar um acesso exclusivo com herança de prioridade.

```
public void start() throws AsynchronouslyInterruptedException {
    super.GRMacquire( true /*isBlocking*/ );
}

public void stop() {
    super.GRMrelease();
}
```

FIGURA 5.22 – Métodos *start* e *stop* da classe *RawDataStorage*

Nos diagramas das FIGURAS 5.19 e 5.20, o objeto *TelemetryGatherer* da classe *DataGatherer* representa um objeto ativo que executa a leitura dos sensores para aquisição dos dados de telemetria. Este objeto é modelado como um recurso escalonável que é disparado periodicamente pelo relógio do sistema para executar uma seqüência de ações com um limitado tempo de execução fim-a-fim. O comportamento temporal deste cenário é caracterizado através dos seguintes elementos:

- O *stereotype* «SASchedulable» aplicado à classe *DataGatherer*, indica que a mesma é um recurso escalonável, com sua própria *thread* de controle;
- O *stereotype* «SATrigger» aplicado à mensagem *gatherData* com o *tag RTat=('periodic', 100, 'ms')* representa o disparo a cada 100 ms do método *gatherData* da classe *DataGatherer*; nesta mesma mensagem também se utiliza o *stereotype* «SAResponse» com o *tag SAAbsDeadline=(100,' ms')* indicando que o tempo limite para execução completa desta seqüência de ações é idêntico ao respectivo período de disparo;

- A mensagem interna *main* representa a execução da *thread* de controle da classe *DataGatherer*, com prioridade definida através da aplicação do *stereotype* «SAAction» com o *tag SAPriority=2*; associados ao mesmo *stereotype* existem os *tags RTduration=(33.5,'ms')* e *SAWorstCase=(93, 'ms')* que informam a duração e o pior tempo de execução da *thread* a cada disparo periódico.

Com exceção das informações sobre duração e pior caso de execução que seriam utilizados apenas para uma análise de viabilidade do escalonamento deste cenário e não necessariamente para geração de código, os demais elementos correspondem ao trecho de código apresentado na FIGURA 5.23, no qual a classe *DataGatherer* é derivada de uma classe do pacote "*javax.rtuml*" que implementa o conceito de recurso escalonável.

```
public class DataGatherer extends javax.rtuml.SASchedulable {

    private static final RelativeTime _100ms = new RelativeTime( 100, 0 );

    public DataGatherer() {
        super(
            new SATrigger( RTarrivalPattern.periodic, _100ms/*RTat*/ ),

            new SAResponse( Thread.NORM_PRIORITY+2 /*SAPriority*/,
                            _100ms /*SAAbsDeadline*/ )
        );
    }

    //...
}
```

FIGURA 5.23 – Construtor da classe *DataGatherer*

O construtor da classe *SASchedulable* cria internamente uma *RealtimeThread* com *SchedulingParameters* baseados no *tag SAPriority* e com *PeriodicParameters* baseados no padrão de ativação definido pela classe *SATrigger* e no tempo limite de execução definido pelo *tag SAAbsDeadline*. Conforme já discutido no capítulo anterior, quando uma *RealtimeThread* é instanciada com *PeriodicParameters*, não existe necessidade de programar explicitamente *timers* externos para manipular a ativação periódica: basta à *RealtimeThread* sinalizar a conclusão da execução do período através do método *waitForNextPeriod* para implicitamente iniciar a espera pela próxima ocorrência do evento periódico.

O método *main* da classe *DataGatherer* é executado pela *RealtimeThread* da classe *SASchedulable* e sua implementação corresponde a um laço no qual a chamada ao método *waitForNextPeriod*[‡‡‡‡‡‡‡], apresentada em destaque na FIGURA 5.24, faz com que o método *gatherData* seja disparado a cada 100 ms, conforme padrão de ativação definido nos diagramas do sistema de telemetria (FIGURAS 5.19 e 5.20).

[‡‡‡‡‡‡‡] Como o método *main* não faz parte de uma classe derivada de *RealtimeThread*, é necessário obter uma referência a *thread* interna da classe *SASchedulable* para executar o método *waitForNextPeriod*.

```
public void main() {
    RealtimeThread thr = getThread();

    while( isRunning() ) {
        gatherData();
        thr.waitForNextPeriod();
    }
}
```

FIGURA 5.24 – Método *main* da classe *DataGatherer*

A classe *SASchedulable* associa um *AsyncEventHandler* à sua *RealtimeThread* interna que será disparado em qualquer período no qual o processamento não seja concluído e executa o método *handleDeadline* que pode ser sobrecarregado pelas classes derivadas. Se em algum período *gatherData* não concluir sua execução no limite de 100 ms, será ativado o método *handleDeadline* para realizar o tratamento desta situação. Na FIGURA 5.25 observa-se que o tratamento adotado foi simplesmente interromper a execução da *RealtimeThread*, cuja execução será finalizada de forma assíncrona através de uma exceção *AsynchronouslyInterruptedException*. Na seqüência o método *main* irá simplesmente aguardar pela ocorrência do próximo evento periódico.

```
protected void handleDeadline() {
    interruptThread(); // javax.rtuml.SASchedulable.interruptThread()
}

private void gatherData()
{
    try {

        // ...

    } catch( AsynchronouslyInterruptedException ignore ) { }
}
```

FIGURA 5.25 – Tratamento de *deadline* na classe *DataGatherer*

Para completar o sistema de telemetria e permitir sua execução também foram implementadas as demais classes previstas nos diagramas:

- *DataDisplayer* é baseada em *SASchedulable* da mesma forma que *DataGatherer* sendo responsável pela apresentação periódica dos valores máximos e mínimos de nível de água coletados pelo sistema de telemetria utilizando o suporte da classe *DisplayInterface*;
- A classe *DataProcessor*, também derivada de *SASchedulable*, é responsável por eliminar periodicamente da base de dados os valores inferiores a 200 cm e superiores a 800 cm;
- *SensorInterface* é a classe que gera valores randômicos para os níveis de água coletados dos sensores distribuídos ao longo das margens do lago da represa;
- O sistema é representado pela classe *TelemetrySystem* que agrega os objetos *SensorData*, *TelemetryGatherer*, *TelemetryDisplayer* e *TelemetryProcessor*;

- Assim como nos demais estudos de caso, este projeto também inclui uma classe *Main* responsável pelo controle geral da simulação e a classe *Util* para registro das mensagens de acompanhamento da simulação.

A FIGURA 5.26 apresenta a saída gerada pelo projeto *Telemetry System* ao ser executado com o suporte da versão de simulação da RTSJ API. Observa-se que devido às freqüências de execução definidas nos diagramas, o objeto *TelemetryProcessor* não consegue eliminar todos os valores inconsistentes antes de sua apresentação pelo objeto *TelemetryDisplayer* ou antes de seu descarte realizado pelo objeto *TelemetryGatherer*, visto que a base de dados mantém apenas os valores mais recentes de nível de água coletada dos sensores.

```
Main               Qui, 15 Nov 2001 18:22:41+301,000000ms Start of simulation
TelemetryDisplayer Qui, 15 Nov 2001 18:22:41+371,000000ms Min: 276 Max: 730
TelemetryDisplayer Qui, 15 Nov 2001 18:22:41+431,000000ms Min: 286 Max: 703
TelemetryDisplayer Qui, 15 Nov 2001 18:22:41+481,000000ms Min: 286 Max: 703
TelemetryDisplayer Qui, 15 Nov 2001 18:22:41+551,000000ms Min: 275 Max: 747
TelemetryDisplayer Qui, 15 Nov 2001 18:22:41+612,000000ms Min: 276 Max: 784
TelemetryDisplayer Qui, 15 Nov 2001 18:22:41+662,000000ms Min: 276 Max: 784
TelemetryDisplayer Qui, 15 Nov 2001 18:22:41+732,000000ms Min: 296 Max: 826
TelemetryDisplayer Qui, 15 Nov 2001 18:22:41+782,000000ms Min: 274 Max: 847
TelemetryDisplayer Qui, 15 Nov 2001 18:22:41+842,000000ms Min: 274 Max: 847
TelemetryProcessor Qui, 15 Nov 2001 18:22:41+882,000000ms Delete value  847
TelemetryDisplayer Qui, 15 Nov 2001 18:22:41+912,000000ms Min: 283 Max: 826
TelemetryDisplayer Qui, 15 Nov 2001 18:22:41+962,000000ms Min: 283 Max: 826
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+022,000000ms Min: 256 Max: 832
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+092,000000ms Min: 256 Max: 832
TelemetryProcessor Qui, 15 Nov 2001 18:22:42+092,000000ms Delete value  832
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+142,000000ms Min: 247 Max: 847
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+212,000000ms Min: 289 Max: 864
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+272,000000ms Min: 289 Max: 864
TelemetryProcessor Qui, 15 Nov 2001 18:22:42+283,000000ms Delete value  864
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+323,000000ms Min: 265 Max: 835
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+393,000000ms Min: 216 Max: 848
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+443,000000ms Min: 216 Max: 848
TelemetryProcessor Qui, 15 Nov 2001 18:22:42+493,000000ms Delete value  848
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+513,000000ms Min: 261 Max: 807
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+563,000000ms Min: 261 Max: 807
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+633,000000ms Min: 296 Max: 814
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+683,000000ms Min: 271 Max: 771
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+753,000000ms Min: 271 Max: 771
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+803,000000ms Min: 253 Max: 793
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+873,000000ms Min: 253 Max: 793
TelemetryDisplayer Qui, 15 Nov 2001 18:22:42+923,000000ms Min: 284 Max: 777
Main               Qui, 15 Nov 2001 18:22:42+994,000000ms End of simulation
```

FIGURA 5.26 – Resultado da Execução do Projeto *Telemetry System*

5.2.2 Versão Refinada do Sistema de Telemetria

O pacote "*javax.rtuml*" foi proposto para concentrar uma série de soluções de mapeamento dos conceitos previstos no UMLSPT para classes Java utilizando o suporte da RTSJ API. Como os nomes das classes deste pacote correspondem às denominações definidas pelo perfil, é bastante direta a identificação no código Java dos elementos que caracterizam seu comportamento temporal nos diagramas.

No entanto, como qualquer mapeamento realizado corresponde à utilização de uma das soluções disponíveis nesta biblioteca, nada impede que este nível intermediário seja eliminado numa etapa posterior de refinamento, gerando como resultado uma aplicação Java que chama diretamente as primitivas da RTSJ API.

```
public class RawDataStorage {

    private Semaphore m_semaphore;

    public RawDataStorage() {
        m_semaphore = new Semaphore( 1/*SAResource.SACapacity*/ );

        MonitorControl.setMonitorControl( m_semaphore,
            PriorityInheritance.instance()/*SAResource.SAAccessControl*/ );
    }

    // ...
}
```

FIGURA 5.27 – Construtor da classe *RawDataStorage*

O código do construtor da classe *RawDataStorage* anteriormente apresentado poderia ser convertido para o código apresentado na FIGURA 5.27 no qual o tipo de controle de acesso ao recurso protegido é definido através da classe *MonitorControl* da RTSJ aplicado a um objeto da classe *Semaphore*[§§§§§§§].

Da mesma forma, os métodos *start* e *stop* apresentados originalmente na FIGURA 5.22 passariam a utilizar diretamente o suporte fornecido pela classe *Semaphore*, para realizar o controle de acesso com herança de prioridade ao recurso protegido, conforme trecho de código apresentado na FIGURA 5.28.

```
public void start() throws AsynchronouslyInterruptedException {
    m_semaphore.acquire( true/*isBlocking*/ );
}

public void stop() {
    m_semaphore.release();
}
```

FIGURA 5.28 – Métodos *start* e *stop* da classe *RawDataStorage*

O código do construtor da classe *DataGatherer* anteriormente apresentado poderia ser convertido para o código apresentado na FIGURA 5.29, no qual sua característica de recurso escalonável é diretamente mapeada através da derivação de *RealtimeThread*. O padrão de ativação, limite de tempo de execução e prioridade são todos representados através de parâmetros do construtor da classe básica.

[§§§§§§§] A classe *Semaphore* não está incluída no JDK ou na RTSJ, mas pode ser facilmente implementada utilizando recursos básicos da linguagem Java (synchronized, wait & notify), conforme apresentado por Doug Lea [LEA 2000].

```
public class DataGatherer extends RealtimeThread {

    private static final RelativeTime _100ms = new RelativeTime( 100, 0 );

    public DataGatherer()
    {
        super(

        new PriorityParameters( Thread.NORM_PRIORITY+2/*SAAction.SAPriority*/ ),

        new PeriodicParameters( null, _100ms/*SATrigger.RTat*/, null,
                                _100ms/*SAResponse.SAAbsDeadline*/, null, null )

        );

        getReleaseParameters().setDeadlineMissHandler( m_asyncDeadline );
    }

    // ...
}
```

FIGURA 5.29 – Construtor da classe *DataGatherer*

Em destaque na FIGURA 5.29, observa-se um *AsyncEventHandler* sendo associado à *RealtimeThread* logo após a construção da própria. Se a mesma não concluir sua execução no limite de 100 ms em algum período, será abortada por uma exceção *AsynchronouslyInterruptedException* disparada pelo *AsyncEventHandler* conforme pode ser visto na FIGURA 5.30.

```
private final RealtimeThread m_thrDataGatherer = this;

private AsyncEventHandler m_asyncDeadline =
    new AsyncEventHandler() {
        public void handleAsyncEvent() {
            if( m_thrDataGatherer.isAlive() ) {
                m_thrDataGatherer.interrupt();
            }
        }
    };
```

FIGURA 5.30 – Tratamento de *deadline* na classe *DataGatherer*

Da mesma forma, o método *main* apresentado originalmente na FIGURA 5.24 seria substituído pelo método *run* e poderia chamar diretamente o método *waitForNextPeriod* conforme destacado na FIGURA 5.31. Esta construção leva à execução periódica do método *gatherData*, disparado a cada 100 ms[********].

[********] O controle para finalização da execução da *thread* passa a ser baseado na variável booleana *m_isRunning* em substituição a chamada ao método SASchedulable.isRunning().

```
public void run()
{
    while( m_isRunning ) {
        gatherData();
        waitForNextPeriod();
    }
}
```

FIGURA 5.31 – Método *run* da classe *DataGatherer*

5.3 Conclusões sobre os Estudos de Caso

Os métodos de modelagem baseados na UML tais como COMET, *Concurrent Object Modeling and Architectural Method*, de Gomaa [GOM 2000] e ROPES, *Rapid Object-Oriented Process for Embedded Systems*, de Douglass [DOU 99] utilizam basicamente os diagramas *Use Case* nas fases iniciais de análise do sistema, diagramas de classes na modelagem estática, diagramas de interação na modelagem dinâmica do sistema e diagramas *Statechart* para modelar máquinas de estados dos objetos quando necessário. Os elementos encontrados nos diagramas de classes tais como objetos, classes, operações, atributos, agregações e generalizações tipicamente não especificam aspectos temporais. As restrições temporais identificadas nas fases de análise e modelagem dinâmica do sistema podem ser incluídas nos diagramas de seqüência e colaboração, através dos mecanismos de extensão da UML: *stereotypes*, *tagged values* e *constraints* conforme estudos de caso apresentados neste trabalho. Este capítulo resume o mapeamento proposto entre as restrições temporais incluídas em diagramas UML e construções Java correspondentes baseadas no suporte tempo-real da RTSJ API.

5.3.1 Diagramas de Classes

Os diagramas de classes da UML representam o relacionamento estático (permanente), existente entre classes, tais como associações, agregações e generalizações conforme mostrado na FIGURA 5.18. Os elementos representados nestes diagramas como classes e objetos e seus métodos e atributos podem ser diretamente mapeados para o código correspondente em Java de forma trivial. Basta que as restrições da linguagem sejam obedecidas pelos diagramas: por exemplo, ao contrário de C++, uma classe Java não pode ser uma especialização de duas classes.

5.3.2 Diagramas de Interação

Os diagramas de seqüência ou colaboração da UML, também conhecidos como diagramas de interação, permitem realizar a modelagem dinâmica, ou seja, especificar como os objetos interagem através da troca de mensagens, cada um enfatizando diferentes aspectos. Na FIGURA 5.19 pode ser visto um diagrama de seqüência, no qual é possível representar a troca de mensagens ao longo de uma linha de tempo, mas o papel dos objetos nos relacionamentos é implícito. Enquanto que nos diagramas de colaboração como na FIGURA 5.20 é possível representar os papéis de cada objeto nos

relacionamentos de forma geométrica[††††††††] [BOO 99a], embora a seqüência de tempo seja menos clara.

Uma mensagem enviada de um objeto A para um objeto B, corresponde à chamada de um método do B sendo realizada pelo objeto A, e os eventuais parâmetros da mensagem tornam-se parâmetros da chamada. A tradução para código Java varia em função do tipo de associação entre os objetos especificada nos diagramas de classes e o tipo de mensagem representada nos diagramas de interação. No caso de mensagens síncronas, o processamento do método chamado seria realizado pela *thread* de controle do próprio objeto emissor da mensagem, conforme os diferentes formas de acesso apresentados na TABELA 5.1. Todos as soluções apresentadas nesta tabela são independentes da RTSJ API, portanto, compatíveis com máquinas virtuais Java convencionais.

TABELA 5.1 – Mapeamento de Mensagens Síncronas para Código Java

Tipo de Associação	Acesso ao Método	Exemplo de Código Java

Generalização ou especialização	Chamada de método da super classe	`public class DataProcessor extends SASchedulable {`
		`  protected void handleDeadline() {`
		`  **super.interruptThread();**`
		`  }`
		`  //...`
		`}`

Composição ou agregação	Chamada de método de um atributo da própria classe	`public class DataGatherer extends RealtimeThread {`
		` private SensorInterface m_Sensors=new SensorInterface();`
		` private void gatherData() {`
		`  ArrayList list = **m_Sensors.getData();**`
		`  //...`
		` }`
		` //...`
		`}`

Associação	Chamada de método estático de outra classe	`public class Distance {`
		`  public static double calculate() {`
		`  int iShaftRotationCount = **ShaftRotationCount.read();**`
		`  //...`
		`  }`
		`  //...`
		`}`

Associação	Chamada de método de um atributo estático de outra classe	`public class DataProcessor extends RealtimeThread {`
		`  private void filterData() {`
		`  try {`
		`  **TelemetrySystem.SensorData.start();**`
		`  //...`
		`  }`
		`  }`
		`  //...`
		`}`

Associação	Chamada de método através de uma referência ou cópia obtida através de outro método	`public class DataDisplayer extends SASchedulable {`
		`  public void main() {`
		`  RealtimeThread thr = getThread();`
		`  while( isRunning() ) {`
		`  displayData();`
		`  **thr.waitForNextPeriod();**`
		`  }`
		`  }`
		`}`

[††††††††] Os relacionamentos são enfatizados através da disposição relativa entre os objetos no diagrama.

As mensagens assíncronas correspondem em Java às interações entre duas *threads*. A *thread* de controle do objeto emissor da mensagem realiza a execução de um método do objeto receptor da mensagem que não realiza o processamento propriamente dito, mas apenas a inclusão dos parâmetros da chamada numa fila implícita ou explícita. A *thread* de controle do objeto receptor verifica periodicamente sua fila de mensagens ou é disparada pelo próprio envio da mensagem e realiza o processamento solicitado de forma concorrente com a *thread* emissora da mensagem. Qualquer uma das formas síncronas apresentadas na TABELA 5.1 pode ser utilizada pelo objeto emissor na chamada ao método que dispara a execução assíncrona. As soluções para disparar execuções assíncronas apresentadas na TABELA 5.2 utilizam o suporte da RTSJ API.

TABELA 5.2 – Mapeamento de Mensagens Assíncronas para Código Java

Fila	Tipo de Solução	Exemplo de Código Java

Implícita:
 	
a própria RTSJ API possui uma fila para múltiplos disparos de um evento
	Disparo de um evento que promove o escalonamento de um *handler* que executa um método do objeto receptor em paralelo com o objeto emissor	`public class AirTrafficModel extends RealtimeThread {`
		`  private AirTrafficEventHandler`
		`  m_ateTransponder = new AirTrafficEventHandler() {`
		`  **public void handleAirTrafficEvent(long lnPos)**`
		`  {`
		`  doTimedMergeTracks( lnPos );`
		`  }`
		`  };`
		`  //...`
		`  public void sendTransponderReturn( long lnPosition ) {`
		`  **m_ateTransponder.fire( lnPosition );**`
		`  }`
		`  //...`
		`}`

Explícita:
 	
o objeto receptor possui uma fila de mensagens como atributo
	Inclusão de uma mensagem numa fila verificada periodicamente pelo objeto receptor em paralelo com o objeto emissor	`public class HeartQueue {`
		`  public class Message {`
		`  //...`
		`  }`
		`  private WaitFreeReadQueue m_queue;`
		
		`  private void write( Message msg ) {`
		`  try {`
		`  **m_queue.write( msg );**`
		`  }`
		`  catch( MemoryScopeException mse ) {`
		`  }`
		`  }`
		`  public Message read()`
		`  throws AsynchronouslyInterruptedException {`
		`  Object obj;`
		`  **while( (obj = m_queue.read()) == null )**`
		`  RealtimeThread.`
		`  interruptibleSleep( LOOP_TIME_WAITING_EVENT );`
		`  return (Message) obj;`
		`  }`
		`  //...`
		`}`

5.3.3 Máquinas de Estados

Para modelar o comportamento de objetos, UML dispõe dos diagramas de máquina de estado que especificam uma seqüência de estados seguida pelos objetos em resposta a

eventos e suas respectivas ações. Neste trabalho não foi desenvolvido nenhum estudo de caso baseado em diagramas deste tipo, mas no projeto do marca passo cardíaco foi implementada uma máquina de estados baseada nas informações disponíveis na FIGURA 5.12.

Assumindo que uma instância da classe *HeartModel* é um objeto ativo, uma solução possível de implementação em Java é apresentada na FIGURA 5.32. O método *Thread.run* deste objeto executa ciclicamente uma instrução *switch* utilizada para selecionar o método correspondente ao código numérico do estado atual[‡‡‡‡‡‡‡]. O estado pode ser alterado como resultado do processamento realizado por estes métodos (FIGURA 5.16), ou em função de eventos ou mensagens recebidas durante a execução destes mesmos métodos. Neste caso específico as mensagens são recebidas através da leitura de uma fila de mensagens explícita conforme já apresentado na TABELA 5.2.

```
public void run() {

    while( true ) {
        switch( m_iState ) {
            case ON_STATE:
                // fall
            case IDLE_STATE:        doIdle();                 break;
            case WAITING_FOR_SENSE: doTimedWaitingForSense(); break;
            case PACING:            doPacing();               break;
            case REFRACTORY:        doRefractory();           break;
            case STOP_SIMULATION:
            default:                                          return;
        }
    }
}
```

FIGURA 5.32 – Método *run* da classe *HeartModel*

5.3.4 Restrições Temporais

Os estudos de caso apresentados nesta dissertação abordam algumas situações típicas de aplicações tempo-real, que ao serem modeladas com UML especificam suas restrições temporais através dos mecanismos padrões de extensão desta linguagem. Os limites de tempo foram definidos com UML *Constraints* incluídos em diagramas de interação e mapeados para código Java com suporte da RTSJ API. A TABELA 5.3 resume as soluções de mapeamento adotadas para estes casos.

Este texto também abordou a modelagem de aplicações tempo-real utilizando os padrões definidos pelo perfil UMLSPT e o mapeamento destes diagramas para código Java. Apesar deste estudo ter sido focado apenas no modelo de escalonabilidade do perfil UMLSPT, a comparação com os diagramas UML anteriores demonstra que a utilização dos padrões deste perfil permite uma melhor caracterização das restrições temporais. O resultado deste estudo está resumido na TABELA 5.4 que relaciona elementos do perfil da OMG com classes ou atributos da RTSJ.

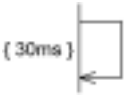
[‡‡‡‡‡‡‡] A construção sugerida assume que cada objeto ativo possui uma única *thread*.

TABELA 5.3 – Mapeamento das Restrições Temporais

Restrição Temporal	Tipo de Solução	Exemplo de Código Java

Execução periódica
 
	Utiliza suporte a execução periódica de *RealtimeThreads* associadas a *Periodic Parameters*.	`public class DistanceAndSpeed extends RealtimeThread {`
		`  RelativeTime period = new RelativeTime( 500, 0 );`
		
		`  PeriodicParameters periodicParameters = new`
		`  PeriodicParameters(null,period,null,null,null,null);`
		
		`  public void run() {`
		`  **setReleaseParameters( periodicParameters );**`
		`  while( m_isRunning ) {`
		`  CurrentSpeed.calculate( Distance.calculate() );`
		`  **waitForNextPeriod();**`
		`  }`
		`  }`
		`  //...`
		`}`

Recepção periódica de mensagens com limite de tempo
 
	Timer periódico é re-escalonado na chegada de cada mensagem periódica e dispara *handler* assíncrono no caso de atingir o limite de tempo nesta recepção.	`public class AirTrafficModel extends RealtimeThread {`
		`  private PeriodicTimer m_ptimer;`
		
		`  private AsyncEventHandler`
		`  **m_async = new AsyncEventHandler()** {`
		`  public void handleAsyncEvent() {`
		`  //...`
		`  }`
		`  };`
		
		`  public AirTrafficModel() {`
		`  RelativeTime period = new RelativeTime( 50, 0 );`
		`  **m_ptimer =**`
		`  **new PeriodicTimer(null,period,m_async);**`
		`  m_ptimer.start();`
		`  }`
		
		`  public void sendRadarReturn( long lnPosition ) {`
		`  AbsoluteTime now = Util.now();`
		`  **m_ptimer.**`
		`  **reschedule(now.add(m_ptimer.getInterval()));**`
		`  //...`
		`  }`
		
		`  //...`
		`}`

Execução de bloco de código com limite de tempo
 
	A versão *timed* de um método realiza a execução do método no escopo de um bloco que pode ser interrompido de forma assíncrona se o limite de tempo for atingido.	`public void doTimedMergeTracks( final long lnPosition ) {`
		`  **Interruptible interruptibly=new Interruptible()** {`
		`  public void run(`
		`  AsynchronouslyInterruptedException aie )`
		`  throws AsynchronouslyInterruptedException {`
		`  mergeTracks( lnPosition );`
		`  }`
		
		`  public void interruptAction(`
		`  AsynchronouslyInterruptedException aie ) {`
		`  //...`
		`  }`
		`  };`
		
		`  RelativeTime timeout = new RelativeTime( 30, 0 );`
		`  **new Timed( timeout ).**`
		`  **doInterruptible( interruptibly );**`
		`}`

Máquina de estados com limite de tempo
 
	Da mesma forma que no caso anterior, a execução do estado *timed* corresponde a um bloco que pode ser interrompido de forma assíncrona.	`public class HeartModel extends RealtimeThread {`
		`  private Timed m_timedWaitingSense = new`
		`  Timed( new RelativeTime(800, 0) );`
		
		`  public void doTimedWaitingForSense() {`
		`  **Interruptible`**
		`  **interruptibly = new Interruptible()** {`
		`  public void run(`
		`  AsynchronouslyInterruptedException aie )`
		`  throws AsynchronouslyInterruptedException {`
		`  doWaitingForSense();`
		`  }`
		
		`  public void interruptAction(`
		`  AsynchronouslyInterruptedException aie ) {`
		`  m_iState = PACING;`
		`  }`
		`  };`
		
		`  **m_timedWaitingSense.`**
		`  **doInterruptible( interruptibly );`**
		`  }`
		
		`  //...`
		`}`

TABELA 5.4 – Relação entre Perfil UMLSPT e RTSJ API

Elemento do Perfil UMLSPT	Classe ou Atributo da RTSJ API
«SAResource»	A aplicação deste *stereotype* caracteriza uma classe que corresponde a um recurso cujo acesso deve ser compartilhado por diversas *threads*, incluindo mecanismos de exclusão mútua como, por exemplo, os providos por um semáforo.
SAResource.SACapacity	Este atributo define o número máximo de *threads* que podem acessar simultaneamente um recurso compartilhado.
SAResource.SAAccessControl	Este atributo caracteriza a política de controle de acesso ao recurso compartilhado, ou seja, o algoritmo utilizado na sincronização das *threads* que disputam o recurso, por exemplo, herança de prioridade.
«GRMacquire»	A classe correspondente ao recurso compartilhado sempre possui um método que implementa a requisição de acesso exclusivo a este recurso, sendo este método definido pela aplicação deste *stereotype*.
GRMacquire.isBlocking	Parâmetro booleano que define se a requisição de acesso exclusivo a um recurso compartilhado deverá bloquear até o recurso ser liberado ou não.
«GRMrelease»	Este *stereotype* define o método que implementa a liberação de um recurso compartilhado.

«SASchedulable»	*Stereotype* que caracteriza qualquer objeto ativo a ser mapeado para uma *RealtimeThread* ou qualquer classe derivada.
«SATrigger»	Caracteriza o padrão de ativação de uma *RealtimeThread*, mas não possui um mapeamento direto para uma classe ou atributo da RTSJ.
SATrigger.SAoccurrencePattern	Atributo que define o padrão de ativação de uma *thread*: periódico, aperiódico ou esporádico o que pode ser mapeado para a classe correspondente derivada de *ReleaseParameters*.
«SAResponse»	Este *stereotype* em conjunto com «SATrigger» caracteriza o padrão de ativação e as propriedades de escalonamento de uma *RealtimeThread*, mas também não existe um mapeamento direto para elementos da RTSJ.
SAResponse.SAAbsDeadline	Atributo utilizado para caracterizar o tempo limite de execução de uma *RealtimeThread*. Corresponde ao atributo *deadline* da classe *ReleaseParameters*.
SAResponse.SAPriority	Atributo utilizado para definir a prioridade de execução de uma *RealtimeThread*, corresponde à classe *PriorityParameters*.
SAResponse.RTduration	Atributo utilizado para caracterizar a duração de uma *RealtimeThread*, ou seja, seu tempo efetivo de execução – corresponde ao conceito *cost* da classe *PeriodicParameters*.
«SAAction»	Delimita um bloco de código, sendo mapeado tipicamente para um método.
SAAction.SARelDeadline	Quando este valor é diferente de NULL, a ação é mapeada para um método ou bloco de código que executa no escopo de um objeto da classe *Timed*, sendo utilizado para definir o tempo limite de execução desta ação.

6 Conclusões e Trabalhos Futuros

O desenvolvimento de um sistema tempo-real é uma tarefa complexa, uma vez que adicionalmente ao atendimento de requisitos funcionais e lógicos, uma correção temporal é exigida. A utilização do paradigma de orientação a objetos como forma de combater esta complexidade, algo já sugerido por diversos autores da área de objetos tempo-real [IEEE 99, 2000 e 2001], pode ser comprovada neste trabalho através da experiência adquirida na implementação da simulação da API *The Real-Time Specification for Java* (RTSJ) de Bollella et al [BOL 2000] e sua posterior utilização na implementação dos estudos de caso.

O desenvolvimento deste trabalho demonstrou que ao utilizar diagramas UML na modelagem de sistemas tempo-real é possível realizar o mapeamento das restrições temporais incluídas nos diagramas para código Java com suporte da RTSJ API. Dependendo do elemento ou restrição temporal em questão, podem existir diversas possibilidades de geração de código RT-Java e não existe um consenso sobre qual a melhor solução. A melhor métrica a ser usada na determinação da melhor solução seria baseada nos resultados obtidos do ponto de vista temporal: determinismo na resposta, eqüidistância de ativação cíclica *et cetera*. No entanto, como a responsabilidade por este tipo de resultados foi transferida para a qualidade da implementação da RTSJ API, esta métrica poderia ser utilizada apenas para confirmar ou descartar uma opção de solução. O mapeamento proposto foi focado na simplicidade com o objetivo de aumentar a legibilidade e pode ser validado de forma intuitiva através de testes com o código gerado, comparando o resultado da execução com as definições especificadas nos diagramas UML. Entretanto, esta tradução não é verificável – a comprovação formal da correção deste mapeamento não estava incluída nos objetivos desta dissertação.

Já existe praticamente um consenso nos meios acadêmicos e profissionais quanto ao emprego da linguagem UML na modelagem de sistemas orientados a objetos, e estudos recentes [DOU 99, GOM 2000 e OMG 2001] comprovam os esforços para permitir e padronizar sua utilização no desenvolvimento de sistemas tempo-real. O interesse por RT-Java e mais especificamente pela RTSJ API cresceu de forma significativa ao longo do desenvolvimento desta dissertação. Isto pode ser comprovado acompanhando o aumento no volume e destaque a estes temas entre os simpósios ISORC 2000 e 2001 [IEEE 2000 e 2001]. Porém, até o primeiro semestre de 2001, durante as pesquisas realizadas para fundamentar este trabalho não foi encontrada nenhuma publicação ligando os dois tópicos, o que indica que o tema desta dissertação ainda é bastante recente. Outros trabalhos ainda não publicados devem estar sendo desenvolvidos em paralelo com este, mas provavelmente não devem invalidar os estudos de caso realizados. Assim, acredita-se que mapeamento proposto possa contribuir para aproximação inevitável entre os tópicos RT-UML e RT-Java.

O alinhamento deste trabalho com tecnologias promissoras e de grande aceitação na área de computação tempo-real orientada a objetos pode ser considerado como um avanço em relação a trabalhos prévios desenvolvidos na UFRGS, como a linguagem AO/C++ definida por Pereira ou o SIMOO-RT desenvolvido por Becker [BEC 99] – esta foi a maior justificativa para não aproveitar a base já desenvolvida. No entanto, o resultado deste trabalho é apenas o passo inicial para gerar uma ferramenta funcional no mesmo nível que o SIMOO-RT, que cobre todo o ciclo de desenvolvimento até a geração automática do código.

Contando com a credencial da entidade OMG, o documento *UML Profile for Scheduling, Performance, and Time* (UMLSPT) [OMG 2001] tem boas chances de tornar-se o padrão na modelagem de sistemas tempo-real utilizando a linguagem UML. Devido ao dinamismo desta área, a última versão do UMLSPT sofreu uma grande evolução em relação à versão do ano anterior, mas o presente trabalho já reflete algumas das atualizações recentemente publicadas e consideradas mais relevantes para o foco do mesmo. Contudo, não foi possível abordar todos os modelos previstos naquela publicação e o mapeamento proposto restringe-se ao modelo de escalonabilidade.

Existe uma boa oportunidade para trabalhos futuros que poderiam analisar todos os conceitos apresentados no perfil UMLSPT, propondo o código RT-Java equivalente. Se a tendência de aceitação deste padrão se confirmar, a continuidade sugerida poderá corresponder a uma contribuição acadêmica bastante significativa.

Atualmente existem disponíveis no mercado ferramentas de modelagem UML com capacidade de geração automática de parte do código Java. Ao modelar sistemas tempo-real já é possível seguir o perfil UMLSPT, se a ferramenta em questão já incluir os mecanismos de extensão da UML: *stereotypes*, *tagged values* e *constraints*. A continuidade deste trabalho poderia envolver a alteração de ferramentas para interpretar as extensões do perfil UMLSPT incluídas nos diagramas, permitindo mapeá-las para código Java com suporte da RTSJ.

Outras possibilidades de trabalhos futuros envolvem questões como: mapeamento de diagramas UML para ambientes RT-Java distribuídos [SUN 2002]; integração do RTSJ com o *middleware* RT-CORBA [OMG 2001a]; análise de desempenho do código mapeado nas diferentes implementações da RTSJ já existentes [AJI 2002, IBM 2002 e SYS 2002] ou a serem lançadas no mercado; assumindo prováveis particularidades entre estas implementações, seria necessária também uma análise de portabilidade do código mapeado. Evidentemente qualquer novo estudo deverá ficar sintonizado com atualizações da RTSJ e do perfil UMLSPT.

Bibliografia

[AJI 2002] AJILE SYSTEMS. **aJile Products**. Disponível em: <http://www.ajile.com/products.htm>. Acesso em: 13 jan. 2002.

[ALT 98] ALTUS SISTEMAS DE INFORMÁTICA. **Manual de Características Técnicas**. [S.l.: s.n.], 1998. CD-ROM.

[AUT 2000] AUTOMACAO.NET. **OPC**: OLE for Process Control. Disponível em: <http://www.automacao.net/opc.htm>. Acesso em: 8 mar. 2000.

[BEC 98] BECKER, Leandro Buss; PEREIRA, Carlos Eduardo. Proposta de Ferramenta Orientada a Objetos para Modelagem, Simulação e Implementação de Sistemas Tempo Real Distribuídos. In: SIMPÓSIO DE SISTEMAS DE SIMULAÇÃO E DE CONTROLE, 1., 1998, Rio de Janeiro. **Anais...** Rio de Janeiro: IPQM, 1998. 175p. p.13-18.

[BEC 99] BECKER, Leandro Buss. **Ambiente de Modelagem e Implementação de Sistemas Tempo Real usando Paradigma de Orientação a Objetos**. 1999. 80p. p.25-55. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BEC 2000] BECKER, Leandro Buss. **Aplicação de Tecnologias de Computação com Objetos Distribuídos no Desenvolvimento de Sistemas Tempo-Real**. 2000. 77p. Exame de Qualificação (Doutorado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BEC 2001] BECKER, Leandro Buss, PEREIRA, Carlos Eduardo. Aplicação de Tecnologias Orientadas a Objetos no Desenvolvimento de Sistemas Computacionais Tempo-Real Distribuídos. In: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, 2001, Fortaleza. **Anais...** [S.l.]: SBC, 2001.

[BOL 2000] BOLLELLA, Greg; GOSLING, James; BROSGOL, Benjamin. **The Real-Time Specification for Java**. Reading, Massachusetts: Addison-Wesley, 2000. 195p. ISBN 0-201-70323-8.

[BOO 94] BOOCH, Grandy. **Object-Oriented Analysis and Design with Applications**. Reading, Massachusetts: Addison-Wesley, 1994. 589p. p.27-80. ISBN 0-8053-5340-2.

[BOO 99] BOOCH, Grandy; RUMBAUGH, James; JACOBSON, Ivar. **The Unified Modeling Language User Guide**. Reading, Massachusetts: Addison-Wesley, 1999. 482p. p.XVIII. ISBN 0-201-57168-4.

[BOO 99a] BOOCH, Grandy; RUMBAUGH, James; JACOBSON, Ivar. **The Unified Modeling Language Reference Manual**. Reading, Massachusetts: Addison-Wesley, 1999. 550p. ISBN 0-201-30998-X.

[BOX 99] BOX, Don. Windows 2000 Brings Significant Refinements to the COM+ Programming Model. **Microsoft System Journal**, [S.l.], May 1999. Disponível em: <http://www.microsoft.com/msj/0599/complusprog/complusprog.htm>. Acesso em: 12 mar. 2000.

[BUR 96] BURNS, Alan; WELLING, Andy. **Real-Time Systems and Programming Languages**. 2nd.ed. Reading, Massachusetts: Addison-Wesley, 1996. 611p. ISBN 020140365X.

[CAH 98] CAHNERS BUSINESS INFORMATION. OPC Solves the I/O Driver Problem. **Control Engineering**, [S.l.], May 1998. Disponível em: <http://www.manufacturing.net/magazine/ce/archives/1998/ctl0501.98/05g506.htm>. Acesso em: 27 jan. 2000.

[CAW 98] CAWLFIELD, David. Achieving Fault-Tolerance with PC-Based Control. In: ISA TECH/EXPO, 1998, Houston, USA. **Technology Update**. [S.l.]: Fry Communications Inc., 1998. CD-ROM.

[CHU 2000] CHUNG, P. Emerald et al. **DCOM and CORBA Side by Side, Step by Step, and Layer by Layer**. Disponível em: <http://www1.bell-labs.com/user/emerald/dcom_corba/Paper.html>. Acesso em: 8 jun. 2000.

[CLE 99] CLEVERLEY, James. COM Plus Watch: COMDeveloper's Eagle Eye on the future of COM and DCOM. **COMDeveloper**, [S.l.], 1999. Disponível em: <http://www.comdeveloper.com/complus/>. Acesso em: 12 mar. 2000.

[CMS 98] CONTROL MAGAZINE. OPC for Process Control: Specification Update / OPC Makes Headway as Foundation Maps the Future / The Automation Supplier Perspective / Development and Support Costs Cut. Itasca, Illinois, 1998. Supplement. Disponível em: <http://www.controlmagazine.com/opc/opc008.html>. Acesso em: 10 fev. 2000[ϕ]

[CMS 99] CONTROL MAGAZINE. OPC for Process Control: Toolkits and Compliance Testing / OPC Nuts and Bolts / Expanding the Spec / No Rest for the Dedicated. Itasca, Illinois, Sept. 1999. Supplement. Disponível em: <http://www.controlmagazine.com/supplement/tct.htm>. Acesso em: 31 mar. 2000[ϕ].

[DIS 98] DISKIN, David; CHUBIN, Sherrie. **Recommendations for USING DCE, DCOM, and CORBA Middleware**, 1998. Disponível em: <http://www.omg.org/library/whitepapers.html>. Acesso em: 11 jun. 2000[ϕ].

[DOU 99] DOUGLASS, Bruce Powel. **Real-Time UML**. 2nd.ed. Reading, Massachusetts: Addison-Wesley, 1999. 328p. ISBN 0-201-65784-8.

[FIS 2000] FISHER-ROSEMOUNT. **Fisher-Rosemount DeltaV System**. Disponível em: <http://www.easydeltav.com/>. Acesso em: 15 fev. 2000. Foto.

[FLO 98] FLORES, Aline Pessano; PEREIRA, Carlos Eduardo. Uma Análise de Metodologias Orientadas a Objetos para o Desenvolvimento de Sistemas Distribuídos e de Tempo Real. In: SIMPÓSIO DE SISTEMAS DE SIMULAÇÃO E DE CONTROLE, 1., 1998, Rio de Janeiro. **Anais...** Rio de Janeiro: IPQM, 1998. 175p. p.7-12.

[FOX 2000] FOXBORO COMPANY. **Foxboro I/A Series System**. Disponível em: <http://www.foxboro.com/iasolutions/>. Acesso em: 15 fev. 2000[ϕ]. Foto.

[GOL 99] GOLDSCHMIDT NETO, Frederico Henrique. **Um Estudo de Ferramentas para Monitoramento de Sistemas e Serviços de Rede**. 1999. 48p. p.28. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[GOM 84] GOMAA, Hassan. A Software Design Method For Real-Time Systems. **Communications of the ACM,** [S.l.], v.27, n.9, p.938-949, Sept. 1984.

[GOM 2000] GOMAA, Hassan. **Designing Concurrent Distributed, and Real-Time Applications with UML**. Boston: Addison - Wesley, 2000. 785p. ISBN 0-201-65793-7.

[GRI 97] GRIMES, Richard. **Professional DCOM Programming**. Olton, Birmingham, United Kingdom: Wrox Press, 1997. 565p. ISBN 1-861000-60-X.

[HEN 99] HENNING, Michi; VINOSKI, Steve. **Advanced CORBA Programming with C++**. Reading, Massachusetts: Addison - Wesley, 1999. 1083p. p.9-35. ISBN 0-201-37927-9.

[HIL 2000] HILCO Technologies. **Java2OPC**: Java Based Software Bridge for Distributing OPC Data Using RMI or CORBA. Disponível em: <http://www.hilco.com/services/appServicesJava.html>. Acesso em: 10 fev. 2000.

[HÖL 96] HÖLTZ, Rudy Hamilton. Estações de Cálculo em Aciarias LD. In: CONGRESSO NACIONAL DE AUTOMAÇÃO, 7., 1996, São Paulo; CONGRESSO MINEIRO DE AUTOMAÇÃO, 1., 1996, Belo Horizonte. **Anais...** [S.l.: s.n.], 1996.

[ϕ] Estas referências não se encontravam mais disponíveis por WWW na conclusão deste trabalho.

[HÖL 2000] HÖLTZ, Rudy Hamilton. **Estudo sobre a Utilização de Padrões de Comunicação para Interoperabilidade de Sistemas Computacionais em Automação Industrial**. 2000. 75p. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[HÖL 2001] HÖLTZ, Rudy Hamilton. **The Real-Time Specification for Java API**: Versão de Simulação. Disponível em http://automation.eletro.ufrgs.br/ooprogramming/. Atualizado em: 24 out. 2001.

[HOR 97] HORSTMANN, M.; KIRTLAND, M. **DCOM Architecture**, 1997. Disponível em: <http://www.microsoft.com/library/backgrnd/html/msdn_dcomarch.htm>. Acesso em: 13 mar. 2000[ϕ]. Figura.

[IBM 2002] IBM. **IBM Visual Age Micro Edition**: Turnkiek Realtime Demonstration. Disponível em: <http://www.embedded.oti.com/learn/rtedemo.phtml>. Acesso em: 13 jan. 2002.

[IEEE 99] IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2., 1999, Saint-Malo, France. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 1999. 352p. ISBN 0-7695-0207-5.

[IEEE 2000] IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 3., 2000, Newport Beach, California, USA. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 2000. 412p. ISBN 0-7695-0607-0.

[IEEE 2001] IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 4., 2001, Magdeburg, Germany. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 2000. 446p. ISBN 0-7695-1089-2.

[INT 99] INTECH BRASIL. Usuários Esclarecem as 30 Dúvidas mais Freqüentes sobre SDCDs e CLPs. **InTech Brasil**, Ribeirão Preto, p.8-40, maio 1999. Entrevista.

[ITO 99] ITO, Sérgio Akira; CARRO, Luigi; JACOBI, Ricardo Pezzuol. Designing a Java Microcontroller to Specific Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 12., 1999, Santander, Spain. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 1999. p.12-15. Disponível em: <http://www.buc.unican.es/IEEE_IEE/Iel_conf_diciembre/conf30.htm>. Acesso em: 5 jul. 2000.

[JC 2000] J CONSORTIUM. **J Consortium Home Page**. Disponível em: <http://www.j-consortium.com/>. Acesso em: 15 set. 2000.

[KAI 99] KAISER, J.; MOCK, M. Implementing the Real-Time Publisher / Subscriber Model on the Controller Area Network (CAN). In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2., 1999, Saint-Malo, France. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 1999. 352p. p.172-181. ISBN 0-7695-0207-5.

[KAS 98] KAISER, Scott. Using PC-based Control for Handling PID Loops. In: ISA TECH/EXPO, 1998, Houston, USA. **Technology Update**. [S.l.]: Fry Communications Inc., 1998. CD-ROM.

[KEN 89] KENNEDY, Paul. **Ascensão e Queda das Grandes Potências**. 4.ed. Rio de Janeiro: Campus, 1989. 675p. p.145. ISBN 85-7001-557-7. Tradução de: The rise and fall of the great powers.

[KIM 99] KIM, K. H.; ISHIDA, Masaki; LIU, Juqiang. An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2., 1999, Saint-Malo, France. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 1999. 352p. p.54-63. ISBN 0-7695-0207-5.

[ϕ] Estas referências não se encontravam mais disponíveis por WWW na conclusão deste trabalho.

[KIM 2000] KIM, K. H. (Kane). API Approximating an Idealistic Real-Time Distributed Object Programming Language. **IEEE Computer**, [S.l.], p.72-80, June 2000. Disponível em: <http://dream.eng.uci.edu/TMO/TMO.htm>. Acesso em: 5 jul. 2000.

[KOP 99] KOPETZ, Hermann; FUCHS, Emmerich; MILLINGER, Dietmar. An Interface as a Design Object. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2., 1999, Saint-Malo, France. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 1999. 352p. p.24-32. ISBN 0-7695-0207-5.

[KRU 97] KRUGLINSKI, David. J. **Inside Visual C++**. 4th.ed. Redmond, Washington: Microsoft Press, 1997. 986p. p.555-597. ISBN 1-57231-565-2.

[LAP 97] LAPLANTE, Phillip A. **Real-Time Systems Design And Analysis**: An Engineer's Handbook. 2nd.ed. Los Alamitos, California: IEEE Computer Society, 1997. 361p. p.7-13. ISBN 0-7803-3400-0.

[LEA 2000] LEA, Doug. **Concurrent Programming in Java**: Design Principles and Patterns. 2nd.ed. Reading, Massachusetts: Addison-Wesley, 2000. 411p. ISBN 0-201-31009-0.

[LEW 95] LEWIS, R.W. **Programming Industrial Control Systems Using IEC 1131-3**. London: The Institution of Electrical Engineers, 1995. 285p. p.16-19. ISBN 0-85296-827-2.

[LEX 99] LEXICON INFORMÁTICA LTDA. **Dicionário Aurélio Eletrônico**: Século XXI, Versão 3.0. [S.l.]: Lexicon Informática, 1999. Correspondente ao Novo Dicionário Aurélio: Século XXI de Aurélio Buarque de Holanda Ferreira. [S.l.]: Nova Fronteira, 1999. CD-ROM.

[LOY 2000] LOYALL, Joseph P. et al. **Flexible and Adaptive Control of Real-Time Distributed Object Computing Middleware**. Submitted to The International Journal of Time-Critical Computing Systems, Kluwer Academic Publishers, 2000. Disponível em: <http://www.cs.wustl.edu/~schmidt/corba-research-realtime.html>. Acesso em: 5 jul. 2000.

[MIC 97] MICROSOFT CORPORATION. **Microsoft Announces COM+**, 1997. Disponível em: <http://www.microsoft.com/presspass/press/1997/Sept97/COMplspr.htm>. Acesso em: 12 mar. 2000[ϕ].

[MIC 98] MICROSOFT CORPORATION. **Dicionário de Informática**. Rio de Janeiro: Campus, 1998. 805p. ISBN 85-3520-255-2. Tradução de: Microsoft Press Computer Dictionary, 3rd.ed., 1997.

[MIC 98a] MICROSOFT CORPORATION. **Microsoft Component Services**: A Technology Overview, 1998. Disponível em: <http://www.microsoft.com/com/wpaper/compsvcs.asp>. Acesso em: 12 mar. 2000.

[MIC 99] MICROSOFT CORPORATION. **MSDN Library Visual Studio 6.0**: Documentação de Referência para Desenvolvedores. [S.l.]: Microsoft, 1999. CD-ROM.

[MIC 2000] MICROSOFT CORPORATION. **Distributed Component Object Model (DCOM)**: Downloads, Specifications, Samples, Papers, and Resources for Microsoft DCOM, 2000. Disponível em: <http://www.microsoft.com/com/tech/dcom.asp>. Acesso em: 12 mar. 2000.

[NEW 2000] NEWMONICS, INC. **PERC 3.01 User Manual**. [S.l.: s.n.], 2000. 259p. CD-ROM.

[NIL 98] NILSEN, Kelvin; LEE, Steve. **PERC Real-Time API** (Draft 1.3), 1998. Disponível em: <http://www.newmonics.com/about/tech/realjava.shtml>. Acesso em: 03 dez. 2000.

[NIS 99] NIST, National Institute of Standards and Technologies. **Requirements for Real-time Extensions for the Java Platform**: Report from the Requirements Group for Real-time Extensions for the Java Platform, 1999. Disponível em: <http://www.nist.gov/itl/div897/ctg/real-time>. Acesso em: 23 ago. 2000.

[ϕ] Estas referências não se encontravam mais disponíveis por WWW na conclusão deste trabalho.

[OMG 95] OBJECT MANAGEMENT GROUP. **CORBA Overview,** 1995. Disponível em: <http://www.infosys.tuwien.ac.at:80/./Research/Corba/OMG/arch2.htm>. Acesso em: 7 mar. 2000.

[OMG 97] OBJECT MANAGEMENT GROUP. **Object Constraint Language Specification**: Version 1.1, 1997. 36p. Disponível em: <http://www-4.ibm.com/software/ad/standards/ocl.html>. Acesso em: 18 dez. 2000.

[OMG 99] OBJECT MANAGEMENT GROUP. **OMA Executive Overview**, 1999. Disponível em: <http://www.omg.org/oma/>. Acesso em: 7 mar. 2000[ϕ].

[OMG 99a] OBJECT MANAGEMENT GROUP. **What is CORBA**, 1999. Disponível em: <http://www.omg.org/corba/whatiscorba.html>. Acesso em: 7 mar. 2000[ϕ].

[OMG 99b] OBJECT MANAGEMENT GROUP. **A Human-Usable Textual Notation for the UML Profile for EDOC**: Request For Proposal, March 1999. Disponível em <http://cgi.omg.org/cgi-bin/doc?ad/99-03-12>. Acesso em: 28 fev. 2001.

[OMG 99c] OBJECT MANAGEMENT GROUP. **UML Profile for Scheduling, Performance, and Time**: Request for Proposal, March 1999. Disponível em: <http://cgi.omg.org/cgi-bin/doc?ad/99-03-13>. Acesso em: 28 fev. 2001.

[OMG 2000] OBJECT MANAGEMENT GROUP. **Object Management Group Home Page**. Disponível em: <http://www.omg.org/>. Acesso em: 7 mar. 2000.

[OMG 2000a] OBJECT MANAGEMENT GROUP. **OMG Unified Modeling Language Specification**: Version 1.3 First Edition, March 2000. Disponível em: <http://www.omg.org/technology/documents/formal/unified_modeling_language.htm>. Acesso em: 28 fev. 2001.

[OMG 2000b] OBJECT MANAGEMENT GROUP. **Response to the OMG RFP for Schedulability, Performance, and Time**: Document Version 1.0, August 2000. Disponível em: <http://cgi.omg.org/cgi-bin/doc?ad/00-08-04>. Acesso em: 28 fev. 2001.

[OMG 2000c] OBJECT MANAGEMENT GROUP. **UML Profile for CORBA Specification, V1.0**: Final Adopted Specification, October 2000. Disponível em: <http://cgi.omg.org/cgi-bin/doc?ptc/00-10-01>. Acesso em: 28 fev. 2001.

[OMG 2001] OBJECT MANAGEMENT GROUP. **Response to the OMG RFP for Schedulability, Performance, and Time**: Revised Submission, June 2001. Disponível em: <http://www.omg.org/cgi-bin/doc?ad/01-06-14>. Acesso em: 05 ago. 2001.

[OMG 2001a] OBJECT MANAGEMENT GROUP. **Real-Time CORBA**, September 2001. Disponível em: <ftp://ftp.omg.org/pub/docs/formal/01-09-28.pdf>. Acesso em: 7 jan. 2002.

[OMG 2001b] OBJECT MANAGEMENT GROUP. **Response to the OMG RFP for Schedulability, Performance, and Time (Erratum Sheet)**, September 2001. Disponível em: <http://cgi.omg.org/cgi-bin/doc?ad/01-09-12 >. Acesso em: 31 out. 2001.

[OPC 98] OPC FOUNDATION. **OPC Common Definitions and Interfaces Version 1.0**, 1998. Disponível em: <http://www.opcfoundation.org/opc_spec_document.htm>. Acesso em: 11 jan. 2000.

[OPC 99] OPC FOUNDATION. **OPC Data Access Custom Interface Specification Version 2.03**, 1999. 181p. p.81-89; 96-105. Disponível em: <http://www.opcfoundation.org/opc_spec_document.htm>. Acesso em: 11 jan. 2000.

[OPC 99a] OPC FOUNDATION. **OPC Alarms and Events Version 1.02**, 1999. Disponível em: <http://www.opcfoundation.org/opc_spec_document.htm>. Acesso em: 11 jan. 2000.

[OPC 2000] OPC FOUNDATION. **OLE for Process Control Foundation Home**. Disponível em: <http://www.opcfoundation.org/>. Acesso em: 01 abr. 2000.

[ORF 97] ORFALI, Robert; HARKEY, Dan; EDWARDS, Jeri. **Instant CORBA**. New York: Wiley & Sons, 1997. 313p. p.3-28. ISBN 0-471-18333-4.

[ϕ] Estas referências não se encontravam mais disponíveis por WWW na conclusão deste trabalho.

[PER 97] PEREIRA, Carlos Eduardo. Desenvolvimento de Sistemas Computacionais para Aplicações em Automação Industrial Usando Paradigma de Orientação a Objetos. In: SEMINÁRIO NACIONAL DE HIDRÁULICA E PNEUMÁTICA, 5., 1997, Florianópolis. **Anais...** [S.l.: s.n.], 1997. p.33-46.

[PER 2000] PEREIRA, Carlos Eduardo. **Curso de Sistemas Tempo-Real**. Disponível em: <http://www.delet.ufrgs.br/~cpereira/temporeal_pos/www/temporeal.htm>. Acesso em: 28 mar. 2000.

[PRA 96] PRADHAN, D. **Fault-Tolerant Computer System Design**. Upper Saddle River, New Jersey: Prentice Hall, 1996. 550p. p.4-8. ISBN 0-13-057887-8.

[RAD 98] RADISYS CORPORATION. **INtime Interrupt Latency Report**: Measured Interrupt Response Times Technical Paper, 1998. Disponível em: <http://www.radisys.com/news_events/pp-main.cfm>. Acesso em: 3 jul. 2000.

[RAD 98a] RADISYS CORPORATION. **Real-Time OPC**: Utilizing INtime to Implement Deterministic OPC Servers White Paper, 1998. Disponível em: <http://www.radisys.com/news_events/pp-main.cfm>. Acesso em: 3 jul. 2000.

[RAD 2000] RADISYS CORPORATION. **RadiSys INtime**: Real-Time For Microsoft Windows NT, 2000. Disponível em: <http://www.radisys.com/oem_products/ds-page.cfm?ms=software&productdatasheetsid=36>. Acesso em: 3 jul. 2000.

[RAJ 98] RAJ, Gopalan Suresh. **A Detailed Comparison of CORBA, DCOM and Java/RMI**. Disponível em: <http://www.execpc.com/~gopalan/misc/compare>. Acesso em: 11 jun. 2000.

[RAJ 99] RAJ, Gopalan Suresh. **A Detailed Comparison of Enterprise JavaBeans (EJB) & The Microsoft Transaction Server (MTS) Models**, 1999. Disponível em: <http://members.tripod.com/gsraj/misc/ejbmts/ejbmtscomp.html>. Acesso em: 15 jun. 2000.

[RAK 99] RAKOW, Bob. Off to a Fast Start With OPC. **Start Magazine**, [S.l.], Sept. 1999. Disponível em: <http://www.startmag.com/v3n7/v3n7p086.asp>. Acesso em: 31 mar. 2000.

[ROD 97] RODRÍGUEZ, Pedro; MOLANO, Anastasio. Characterization of Blocking Time in Real-Time Systems with Dynamic Priority Ceilings. In: EUROMICRO WORKSHOP ON REAL-TIME SYSTEMS, 9., 1997. Toledo, Spain. **Proceedings…** Los Alamitos, California: IEEE Computer Society, 1997. 275p. p.94-101. ISBN 0-8186-8034-2.

[ROF 2000] ROFAIL, Ash; SHOHOUD, Yasser. **Mastering COM and COM+**. San Francisco: Sybex, 2000. 693p. p.1-151. ISBN 0-7821-2384-8.

[ROS 98] ROSEN, Michael; CURTIS, David. **Integrating CORBA and COM Applications**. New York: Wiley & Sons, 1998. 332p. p.1-2. ISBN 0-471-19827-7.

[RGB 2000] ROY-G-BIV CORPORATION. **About OPC**. Disponível em: <http://www.roygbiv.com/tech/AboutOPC.htm>. Acesso em: 31 mar. 2000[ϕ].

[ROY 97] ROY, Mark; EWALD, Alan. **Inside DCOM**. [S.l.]: Miller Freeman, Inc., 1997. Disponível em: <http://www.dbmsmag.com/9704d13.html>. Acesso em: 12 mar. 2000.

[RTM 98] REAL TIME MAGAZINE. Windows & Real-Time. **Real-Time Magazine**, [S.l.], issue 98-3, 1998. Disponível em: <http://www.realtime-info.be/encyc/magazine/98q3/index983.htm>. Acesso em: 31 mar. 2000.

[RUF 99] RUFINO, José; VERÍSSIMO, Paulo; ARROZ; Guilherme. Embedded Platforms for Distributed Real-Time Computing: Challenges and Results. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2., 1999, Saint-Malo, France. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 1999. 352p. p.24-32. ISBN 0-7695-0207-5.

[ϕ] Estas referências não se encontravam mais disponíveis por WWW na conclusão deste trabalho.

[SCH 2000] SCHMIDT, Douglas. **Overview of CORBA**, 2000. Disponível em: <http://www.cs.wustl.edu/~schmidt/corba-overview.html>. Acesso em: 7 mar. 2000.

[SCH 2000a] SCHMIDT, Douglas. **Overview of TAO**, 2000. Disponível em: <http://www.cs.wustl.edu/~schmidt/research.html>. Acesso em: 2 jul. 2000. Figura.

[SCH 2000b] SCHMIDT, Douglas; KUHNS, Fred. **An Overview of the Real-time CORBA Specification**. IEEE Computer special issue on Object-Oriented Real-time Distributed Computing, edited by Eltefaat Shokri and Philip Sheu, June 2000. Disponível em: <http://www.cs.wustl.edu/~schmidt/corba-research-realtime.html>. Acesso em: 5 jul. 2000.

[SCM 2000] SCHNEIDER, Manfred. **Cetus Links**: Links on Objects and Components / Distributed Objects & Components: General Information, Feb. 2000. Disponível em: <http://www.cetus-links.org/top_distributed_objects_components.html>. Acesso em: 8 mar. 2000.

[SEL 94] SELIC, Bran; GULLEKSON, Garth; WARD, Paul. **Real-Time Object-Oriented Modeling**. New York: Wiley & Sons, 1994. 525p. p.1-63. ISBN 0-471-59917-4.

[SEL 99] SELIC, Bran. Protocols and Ports: Reusable Inter-Object Behavior Patterns. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2., 1999, Saint-Malo, France. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 1999. 352p. p.332-339. ISBN 0-7695-0207-5.

[STA 88] STANKOVIC, John A.; RAMAMRITHAM, Krithi. **Hard Real-Time Systems**. Los Alamitos, California: IEEE Computer Society, 1988. 618p. p.1-7. ISBN 0-8186-0819-6.

[STA 93] STANKOVIC, John A.; RAMAMRITHAM, Krithi. **Advances in Real-Time Systems**. Los Alamitos, California: IEEE Computer Society, 1993. 777p. p.1-25; 351-360. ISBN 0-8186-3792-7.

[STW 97] STALLINGS, William. **Data and Computer Communications**. 5th.ed. Upper Saddle River, New Jersey: Prentice-Hall, 1996. 798p. p.328-334. ISBN 0-02-415425-3.

[STU 98] STUDEBAKER, Paul. Object Technology Targets Process Control. **Control Magazine**, Itasca, Illinois, June 1998. Disponível em: <http://www.controlmagazine.com/archives/0698/c0570698.html>. Acesso em: 10 fev. 2000[ϕ].

[SUN 2000] SUN MICROSYSTEMS. **Java Remote Method Invocation (RMI)**. Disponível em: <http://www.javasoft.com/products/jdk/rmi/index.html>. Acesso em: 15 set. 2000.

[SUN 2002] SUN MICROSYSTEMS. **Java Specification Requests 50**: Distributed Real-Time Specification. Disponível em: <http://jcp.org/jsr/detail/50.jsp>. Acesso em: 07 jan. 2002.

[SYS 2002] SYSTRONIX INC. **Systronix Home Page**: JStamp board. Disponível em: <http://www.systronix.com/home.htm>. Acesso em: 13 jan. 2002.

[SZY 97] SZYPERSKI, Clemens. **Component Software**: Beyond Object-Oriented Programming. New York: Addison-Wesley, 1997. 411p. p.132-144; 178-217. ISBN 0-20117888-5.

[TEM 2000] TEMPLEMAN, Julian et al. The Idea of COM. **COMDeveloper**, [S.l.], 2000. Disponível em: <http://www.comdeveloper.com/articles/COMIdea.asp>. Acesso em: 12 mar. 2000.

[TIM 98] TIMMERMAN, Martin. Windows NT Real-Time Extensions: better or worse? **Real-Time Magazine**, [S.l.], issue 98-3, p.11-19, 1998. Disponível em: <http://www.realtime-info.be/encyc/magazine/98q3/index983.htm>. Acesso em: 31 mar. 2000.

[VOG 99] VOGEL, Andreas; RANGARAO, Madhavan. **Programming with Enterprise JavaBeans, JTS and OTS**: Building Distributed Transactions with Java and C++. New York: Wiley Computer Publishing, 1999. 356p. ISBN 0-471-31972-4.

[ϕ] Estas referências não se encontravam mais disponíveis por WWW na conclusão deste trabalho.

[WAS 96] COLLEGE OF ENGINEERING, UNIVERSITY OF WASHINGTON. **Introduction To Programmable Logic Controller**, Dec. 1996. Disponível em: <http://rcs.ee.washington.edu/CR/IA/notes/PLC.html>. Acesso em: 11 fev. 2000.

[WEL 2000] WELLSPRING SOLUTIONS, INC. **OAenterprise 99**: OAenterprise, the OAframework and DCOM in Manufacturing. Disponível em: <http://www.wellspringsolutions.com/solutions/whitepappers/DCOM.htm>. Acesso em: 24 mar. 2000[ϕ].

[WOH 99] WOHLEVER, Steven et al. CORBA-based Real-time Trader Service for Adaptable Command and Control Systems. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2., 1999, Saint-Malo, France. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 1999. 352p. p.64-71. ISBN 0-7695-0207-5.

[ϕ] Estas referências não se encontravam mais disponíveis por WWW na conclusão deste trabalho.