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

Um Método para Abordar todo o Ciclo de Desenvolvimento de Aplicações Tempo Real

por

LEANDRO BUSS BECKER

Tese submetida à avaliação, como
requisito parcial para a obtenção do grau de
Doutor em Ciência da Computação

Prof. Dr. Carlos Eduardo Pereira
Orientador

Porto Alegre, maio de 2003.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Becker, Leandro Buss
Um Método para Abordar para todo o Ciclo de Desenvolvimento de Aplicações Tempo Real / por Leandro Buss Becker. – Porto Alegre: PPGC da UFRGS, 2003.
151 p.:il.

Tese (Doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós Graduação em Computação, Porto Alegre, BR, 2003. Orientador: Pereira, Carlos Eduardo.

1. Método de desenvolvimento. 2. Sistemas Tempo Real. 3. Orientação a Objetos. 4. Modelagem. 5. Requisitos Temporais. 5. Escalonamento. 6. Validação. I. Pereira, Carlos Eduardo. II. Título.

Este trabalho não teria sido feito sem o amor, carinho e apoio de algumas pessoas especiais na minha vida: meus pais, Carlos Celeste e Maria Teresa; minhas irmãs, Anne e Cristiane; meus avós, Adão e Dila; e minha madrinha, Tia Cláudia.

Agradecimentos

Agradeço aos meus dois “pais”, ou *Doktor Father*, que muito me ensinaram e apoiaram nestes longos anos: Carlos Eduardo Pereira e Edgar Nett. Também ressalto a fundamental importância de dois colegas e amigos, que compartilharam comigo discussões, implementações, artigos e principalmente palavras de amizade e apoio: Martin Gergeleit e Rudy Höltz. Aos integrantes do grupo de pesquisa EUK da Universidade de Magdeburg, Petra, Manuela, Spiro, Manfred, Thomas e Stefan, meu agredecimento especial por terem me propiciado um segundo lar e preenchido os espaços da saudade e da solidão. Agradeço ainda os amigos do Laboratório de Automação da UFRGS Carlos Mitidieri, Ronaldo Hüsemann e Cláudio Villela, que também contribuíram com sua competência técnica e amizade para o desenvolvimento deste trabalho.

Sumário

Lista de Abreviaturas

AAC	*Autonomous Activation Condition*
ACE	*Adaptive Communication Environment;*
ADM	*Adaptive Deadline Monotonic*;
ADOORATA	*A Distributed Object-Oriented Architecture for Real-Time Automation;*
AIE	*Asynchronously Interrupted Exception;*
AMI	*Asynchronous Method Invocation*
AO/C++	*Active Object C++;*
API	*Application Programming Interface;*
ATC	*Asynchronous Transfer of Control;*
BMS	*Bi-Modal Scheduler;*
CASE	*Computer Aided Software Engineering;*
COMET	*Concurrent Object Modeling and Architectural Method*;
CORBA	*Common Object Request Broker Architecture;*
COTS	*Commercial Off-The-Shelf;*
CP	*Critical Period;*
CPU	*Central Processing Unit*;
DL	*Deadline*;
ECET	*Expected Case Execution Time;*
EDF	*Earliest Deadline First;*
EDL	*Earliest Deadline as Late as possible;*
EP	*Exception Part;*
FIFO	*First-In First-Out;*
GC	*Garbage Collector;*
HDF	*Highest Density First;*
IEEE	*Institute of Electrical and Electronics Engineers;*
IDL	*Interface Definition Language;*
I/O	*Input/Output;*
IOR	*CORBA's Interoperable Object Reference;*
IP	*Internet Protocol;*
IPC	*Interprocess Communication;*
ISO	*International Standard Organization;*
JC	*J Consortium;*
JDK	*Java Development Kit;*
JEG	*Java Experts Group;*
JVM	*Java Virtual Machine;*
J2ME	*Java 2 Micro Edition;*
LRT	*Latest Release Time;*
MMC	Mínimo Múltipo Comum;
MP	*Main Part;*
MUF	*Maximum Utilization Factor;*
MVD	*Maximum Validity Duration;*
NIST	*U.S. National Institute of Standards and Technology;*
OCL	*Object Constraint Language;*
OCTOPUS	*Object-Oriented Technology for Real-Time Systems*;
ODS	*Object Data Space*
OMG	*Object Management Group*;
OORTAC	*Object-Oriented Real-Time and Control*;

ORB	*Object Request Broker;*
ORC	*Object-Oriented Real-Time Distributed Computing*;
OS	*Operating System*
PC	*Personal Computer;*
PERC	*Portable Executive for Reliable Control;*
PERL	*Practical Extraction and Report Language;*
QDL	*Quality Description Language;*
PDF	*Probability Distribution Function*;
POA	*CORBA's Portable Object Adapter;*
POSIX	*Portable Operating System Interface;*
QoS	*Quality of Service;*
QuO	*Quality Objects;*
RAM	*Random Access Memory;*
RM	*Rate Monotonic;*
RMI	*Remote Method Invocation;*
ROOM	*ObjecTime's Real-Time Object-Oriented Modeling;*
ROPES	*Rapid Object-Oriented Process for Embedded Systems;*
RPC	*Remote Procedure Call;*
RT	*Real-Time;*
RT-CORBA	*Real-Time CORBA*;
RTC++	*Real-Time Extension of C++;*
RTJVM	*Real-Time Java Virtual Machine;*
RTL	*Real-Time Logic;*
RTLinux	*Real-Time Linux;*
RTORB	*Real-Time Object Request Broker;*
RTOS	*Real-Time Operating System;*
RTPOA	*Real-Time Portable Object Adapter;*
RTSJ	*The Real-Time Specification for Java;*
SART	*Structured Analysis Real-Time;*
SIMOO	Plataforma Orientada a Objetos para Simulação Discreta Multi-Paradigma*;*
SIMOO-RT	Extensão tempo real da plataforma SIMOO;
SpM	*Spontaneous Methods;*
SvM	*Service Methods;*
SO	Sistema Operacional;
TAO	*The ACE ORB;*
TAFT	*Time-Aware Fault-Tolerant*;
TCP	*Transmission Control Protocol;*
TMO	*Time-triggered Message-triggered Object;*
TP	*Task Pair;*
TVL	*Tag Value Language;*
UML	*Unified Modeling Language;*
UML-TR	Referência para o *Profile for Schedulability, Performance, and Time;*
UNIX-TR	UNIX Tempo real;
VCAT	*Visual Communication Analysis Tool*;
WCET	*Worst Case Execution Time;*

Lista de Figuras

Lista de Tabelas

Resumo

Neste trabalho apresenta-se um método de desenvolvimento integrado baseado no paradigma de orientação a objetos, que visa abordar todo o ciclo de desenvolvimento de uma aplicação tempo real. Na fase de especificação o método proposto baseia-se no uso de restrições temporais padronizadas pelo perfil da UML-TR, sendo que uma alternativa de mapeamento destas restrições para o nível de programação é apresentada. Este mapeamento serve para guiar a fase de projeto, onde utilizou-se como alvo a interface de programação orientada a objetos denominada TAFT-API, a qual foi projetada para atuar junto ao ambiente de execução desenvolvido no âmbito desta tese. Esta API é baseada na especificação padronizada para o Java-TR. Este trabalho também discute o ambiente de execução para aplicações tempo real desenvolvido. Este ambiente faz uso da política de escalonamento tolerante a falhas denominada TAFT (*Time-Aware Fault-Tolerant*). O presente trabalho apresenta uma estratégia eficiente para a implementação dos conceitos presentes no escalonador TAFT, que garante o atendimento a todos os *deadlines* mesmo em situações de sobrecarga transiente. A estratégia elaborada combina algoritmos baseados no *Earliest Deadline*, sendo que um escalonador de dois níveis é utilizado para suportar o escalonamento combinado das entidades envolvidas. Adicionalmente, também se apresenta uma alternativa de validação dos requisitos temporais especificados. Esta alternativa sugere o uso de uma ferramenta que permite uma análise qualitativa dos dados a partir de informações obtidas através de monitoração da aplicação. Um estudo de caso baseado em uma aplicação real é usado para demonstrar o uso da metodologia proposta.

Palavras-Chave: método de desenvolvimento; sistemas tempo real; orientação a objetos; modelagem; requisitos temporais; escalonamento; validação.

TITLE: “A METHOD FOR APPROACHING THE COMPLETE DEVELOPMENT CYCLE OF REAL-TIME APPLICATIONS”.

Abstract

This thesis presents an integrated design method based in the object-oriented paradigm, which covers the whole development cycle of real-time applications. During the specification phase the proposed method suggests making use of timing annotations defined in the RT-UML standard. This work proposes an approach for bridging the gap between specification and design, aiming to provide a clear link between the modeled real-time constraints and the programming entities that provide their implementation. The main idea is to enhance the traceability as well as readability of timing constraints from a model-based requirements model to implementation. Relationships between the stereotypes and tags used to decorate the UML diagrams and their code representation are explained. The timing annotations are used as input for the design phase, which proposes a strategy for mapping these annotations to the programming level. As target language the work uses the TAFT-API, which is an object-oriented programming interface based in the RT-Java standard designed within the context of the TAFT execution environment. This environment makes use of the Time-Aware Fault-Tolerant (TAFT) scheduling policy. The present work discusses an efficient implementation strategy for those concepts proposed by the TAFT scheduler, which provides execution guarantees to ensure that deadlines are always met, even during transient overload situations. The proposed strategy is composed of a two-level scheduler based in Earliest Deadline algorithms. For the validation of the timing requirements from the designed applications it is suggested the use of an analysis tool, which allows a qualitative analysis from the data obtained from application-monitoring during runtime.

Keywords: development method; real-time systems; object orientation; modeling; timing requirements; scheduling, validation.

1 Introdução

Este documento apresenta a tese do autor, desenvolvida no Instituto de Informática da Universidade Federal do Rio Grande do Sul em colaboração com o Departamento de Sistemas Distribuídos da Universidade de Magdeburg, Alemanha. Ao longo do capítulo, esta tese é contextualizada na área de pesquisa, são ressaltadas as suas motivações e objetivos, bem como as suas principais contribuições científicas. No final, apresenta-se a maneira como se encontra organizado o restante deste documento.

1.1 Contextualização

Aplicações tempo real devem satisfazer não somente requisitos funcionais, mas também devem possuir um comportamento temporal determinístico. Assim, não basta que sejam computados algoritmos cujos resultados estejam logicamente corretos, se os mesmo não forem fornecidos dentro do limite de tempo pré-estabelecido. Stankovic, no seu clássico trabalho [STA 88], analisa as diversas interpretações erradas associadas com sistemas de tempo real e discute os desafios a serem vencidos nesta área de pesquisa. Outros trabalhos, como [HAL 91; MOT 93; SHI 94], discutem a terminologia associada com os requisitos presentes nesta classe de aplicações.

Pesquisas mostram que o desenvolvimento de aplicações tempo real tem recebido uma maior atenção nos últimos anos. Isto se deve ao uso disseminado de componentes de *hardware*, que devido aos avanços tecnológicos tiveram um aumento significativo em sua capacidade de processamento, ao mesmo tempo em que diminuíram o custo, consumo de energia e tamanho. No ano de 1999, foi estimado que 98% das CPUs eram utilizadas para uso em sistemas embutídos (grande parte dos quais com requisitos temporais), enquanto que somente 2% estavam sendo usadas pelo mercado de computadores de uso pessoal [TUR 99]. Este dado justifica a grande quantidade de investimentos atualmente aplicada a esta classe de sistemas.

Analisando o desenvolvimento de sistemas tempo real, percebe-se que até bem pouco tempo este ocorria de maneira *ad-hoc*. Segundo Gomaa [GOM 93], tal desenvolvimento era centrado na codificação e fazia pouco uso das técnicas de estruturação existentes. Com o avanço crescente da tecnologia e o conseqüente aumento na complexidade destes sistemas, os projetistas precisaram se familiarizar com o uso de metodologias de projeto. Estas metodologias oferecem conceitos úteis para aumentar o nível de abstração das especificações e também estabelecem regras para guiar o ciclo de desenvolvimento dos projetos.

As primeiras metodologias de projeto voltadas para sistemas tempo real surgiram como extensão às técnicas de análise estruturada. As mais importantes destas extensões foram as metodologias SART (*Structured Analysis Real-Time*) [WAR 85; HAT 87]. Fatores como o baixo grau de encapsulamento, baixa reusabilidade e difícil manutenção acabaram por caracterizar negativamente o emprego desta metodologia.

Nos últimos anos, as metodologias de projeto baseadas no paradigma de orientação a objetos têm sido apontadas como uma alternativa interessante para combater as deficiências apresentadas pelas técnicas de análise estruturada. Este paradigma apresenta diversas características que facilitam o entendimento do modelo, bem como permite um maior encapsulamento para os dados e facilita o re-uso. Conseqüentemente, o

paradigma de orientação a objetos também acabou sendo aplicado com sucesso no desenvolvimento de sistemas tempo real (vide [AWA 96], [DOG 98], [GOM 2000], [KIM 99], [KIM 2000], [PER 94a], [PER 97], [SEL 94]).

Hoje em dia a chamada computação tempo real orientada a objetos (*Object Real-Time Computing – ORC*) têm recebido um grande impulso [SHO 2000], sendo inclusive realidade em diversas aplicações industriais. As tecnologias ORC se fazem presentes ao longo das diferentes etapas que compõem o ciclo de vida de um sistema computacional tempo real.

1.2 Motivações

Esta seção aborda os principais fatores que motivaram o desenvolvimento deste trabalho, a constar: (i) falta de padronização na especificação e no projeto de requisitos temporais; (ii) falta de um ambiente de execução que maximize os uso dos recursos de processamento e ainda assim ofereça garantias ao atendimento dos deadlines; (iii) lacunas na transição entre as fases de desenvolvimento. A seguir descreve-se cada um destes fatores individualmente.

1.2.1 Especificação e Programação de Requisitos Temporais

Por bastante tempo a especificação de requisitos temporais era tida como uma carência por parte das metodologias de desenvolvimento, conforme destacado nos trabalhos de Pereira [PER 96] e Selic [SEL 99]. Justamente para abordar este problema, foi adotado no começo de 2002 o perfil para o UML-TR [OMG 2002], que utiliza os mecanismos de extensão da UML para padronizar a descrição dos diversos requisitos associados com o desenvolvimento de aplicações tempo real. O perfil UML-TR permite aos projetistas decorar os diagramas orientados a objetos com anotações temporais padronizadas, aumentando o entendimento do modelo e aumentando também o seu formalismo, de forma a facilitar o intercâmbio de informações entre a ferramenta de modelagem e ferramentas de análise.

Entretanto, este perfil não trata sobre como estes requisitos devam ser programados (talvez devido ao caráter genérico do perfil UML-TR). Analisando historicamente, verifica-se que a grande maioria das metodologias de desenvolvimento tem relegado a programação das restrições temporais a um segundo plano, considerado-as meros detalhes de implementação. Entretanto, verifica-se que algumas abordagens são exceção em relação a esta prática. O trabalho de Burns e Wellings com a metodologia HRT-HOOD [BUR 95] foi um dos pioneiros nesta prática, abordando o mapeamento de restrições temporais para a linguagem Ada. A metodologia OCTOPUS [AWA 96] também discute o projeto de modelos orientado a objetos de maneira a atender os seus requisitos temporais. Também destaca-se o trabalho desenvolvido pelo presente autor com o ambiente SIMOO-RT [BEC 99]. Este trabalho propõe extensões que permitem a associação de requisitos temporais com os métodos das classes ativas. Com isto, o ambiente suporta a definição de métodos periódicos e também métodos com *deadlines*. Nesse trabalho, os requisitos temporais definidos durante a modelagem são diretamente mapeados para uma extensão tempo real da linguagem C++.

Mesmo com a popularização das metodologias orientadas a objetos e a expansão das tecnologias ORC, alguns problemas acabam por prejudicar o uso destas técnicas no

processo de desenvolvimento como um todo. Além da falta de familiarização de muitos programadores e projetistas com os conceitos da orientação a objetos, existe um sério problema no que diz respeito à interação entre as diversas etapas de desenvolvimento que compõem o ciclo de vida de um sistema computacional. Geralmente, as etapas de desenvolvimento possuem um baixo índice de acoplamento, onde os desenvolvedores, ao iniciar uma nova etapa, acabam por causar profundas mudanças naquilo que fora concebido anteriormente.

Por exemplo, muitas vezes o resultado de um trabalho de análise e projeto orientado a objetos não é codificado em uma linguagem de programação orientada a objetos. Este simples fato tende a provocar uma quebra de estrutura entre a especificação e a implementação. Outro exemplo pertinente sobre a falta de sistematização no processo de transição diz respeito à programação das especificações temporais. Mesmo provedores de ferramentas consideram a sua representação em código como "detalhes de implementação". Como conseqüência, o código que expressa os requisitos temporais acaba por se misturar ao código que trata da funcionalidade da aplicação, dificultando a sua manutenção e validação[1].

Este fato vai em desencontro aos avanços obtidos pelas tecnologias ORC voltadas para programação. Existem várias opções de interfaces de programação (*Application Programming Interface* – API) e extensões de linguagens disponíveis, como por exemplo AO/C++ [PER 94], QuO [ROD 2000], RT-CORBA [OMG 99], RTC++ [ISH 90], RTSJ [BOL 2001] e TMO [KIM 99]. Estas tecnologias oferecem opções estruturadas e claras para a programação de sistemas tempo real orientados a objetos. Apesar de adequadas para a programação, quando se trata de cumprir os requisitos temporais a responsabilidade é deixada com o ambiente de execução ou sistema operacional subjacente, que nem sempre conseguem oferecer as garantias desejadas. Pereira et al [PER 99] demonstram que sistemas convencionais baseados em plataformas Ms-Windows não possuem o comportamento determinístico desejado. Bacellar [BAC 2000] apresenta um estudo semelhante, onde o foco é o sistema operacional embutido Windows-CE.

1.2.2 Suporte dos Ambientes de Execução

O suporte oferecido pelo ambiente de execução é essencial para que uma aplicação tempo real venha a executar de forma correta. As características que um sistema operacional deve possuir para suportar a execução de aplicações tempo real são definidas pela norma POSIX 1003.1b, a qual encontra-se subcontida em [IEEE 2001]. Dentre o conjunto de características definidas pela norma POSIX, encontra-se a política de escalonamento para o sistema. O escalonador é o componente mais importante em um sistema tempo real, uma vez que o mesmo é encarregado de alocar, distribuir e controlar recursos às tarefas, de modo a garantir que os *deadlines* são sempre cumpridos. Tradicionalmente, a garantia do cumprimento dos *deadlines* em sistemas tempo real críticos é baseada no uso de tempos de execução de pior caso, denominados *Worst Case Execution Time* (WCET) , caracterizados pelo tempo mais longo de execução da tarefa. Estes tempos denotam uma perspectiva pessimista, uma vez que em muitos casos práticos, o tempo mais longo é bastante superior ao tempo de execução

[1] A norma IEEE 610.12-1990 [IEEE 90] define validação como "o processo de avaliar um sistema ou componente durante ou no final do processo de desenvolvimento, a fim de determinar se ele satisfaz os requisitos especificados".

médio. Todavia, considerando que um sistema tempo real deve sempre atender seus requisitos temporais, o WCET é considerado uma métrica importante e é usado tanto em algoritmos estáticos (vide[LIU 2000]), como o *Rate Monotonic* [LIU 73] e o *Deadline Monotonic* [LEU 82], quanto em algoritmos dinâmicos (vide [STA 98]), como o *Earliest Deadline First* [LIU 73] e o *Highest Density First* [BUT 95].

O uso de WCET leva a situações de "sobrecarga artificial", i.e. podem existir situações em que apesar de existir CPU disponível para executar novas tarefas, as mesmas não são aceitas pelo escalonador. Por outro lado, nas políticas tradicionais de escalonamento mencionadas as garantias são perdidas caso os WCET forem subestimados. Adicionalmente, a crescente complexidade dos programas de computador, incluindo aspectos de concorrência, recursividade, laços de decisão que dependem de valores dinâmicos de variáveis, etc, acabam tornando muito difícil estimar o WCET através de métodos analíticos [EDG 2001]. Assim, os resultados computados acabam sendo um limite superior muito pessimista dos valores de tempo de execução observados. Já em muitas aplicações compostas por sistemas heterogêneos, torna-se virtualmente impossível de se computar os WCETs.

Embora o uso de WCETs superestimados possa ser considerado como uma solução elegante do ponto de visto de garantias de atendimento de requisitos temporais, o elevado custo dos sistemas computacionais obtidos pode se tornar proibitivo.

Abordagens alternativas, tais como apresentado por Gergeleit em [GER 2001] usam tempos de execução estimados com base em estatísticas e na monitoração on-line do sistema. Já na proposta de Edgar e Burns [EDG 2001] é apresentado um esquema estatístico para estimar WCETs com uma certa confidência, sendo o resultado usado como entrada para um escalonador estático. Enquanto esta proposta elimina a necessidade de se fazer a geralmente ineficiente análise de código, a mesma não pode ser usada em escalonamento dinâmico devido às possíveis mudanças no sistema e das suas conseqüências nas estimativas de tempo e também na carga total do sistema.

Um efeito colateral das mudanças dinâmicas no sistema são as chamadas situações de sobrecarga transiente. Isto pode ocorrer em condições especiais, como por exemplo durante a chegada de tarefas esporádicas ou distúrbios em algum conjunto de dados sendo analisado, que levam as tarefas a ultrapassar o seu tempo de execução estimado, causando uma demanda maior do que a capacidade de processamento da CPU. Aplicações que se deparam com estas situações, longe de serem irreais, necessitam de um componente de escalonamento dinâmico, capaz de trabalhar com valores incertos de tempo de execução (por serem provenientes de estimativas), e com mudanças dinâmicas de comportamento.

Uma maneira de se alcançar os requisitos desejados, quais sejam as garantias a priori de que deadlines serão atendidos mesmo em presença de sobrecargas transientes, é negociando aspectos de funcionalidade por precisão temporal, uma estratégia bastante conhecida em tolerância a falhas. Entende-se por "falha temporal" o fato de uma tarefa ultrapassar o seu *deadline*. A política de escalonamento conhecida por TAFT (*Time-Aware Fault-Tolerant*) [STR 94; NET 97; NET 2001] aborda justamente estes aspectos de negociação, sendo capaz de tratar falhas temporais iminentes, de modo a garantir que os *deadlines* sejam sempre cumpridos mesmo em situações de sobrecarga transiente. Diferentemente de outras abordagens para tratar com situações de sobrecarga transiente, como o *Robust EDF* (RED) [BUT 93], o *Adaptative DM* (ADM) [RIC 2001] e o *Bi-Modal Scheduler* [CAS 2001], as quais ainda dependem do uso de WCET, o escalonador TAFT se baseia no uso de tempos de execução estimados – *Expected Case*

Execution Times (ECET). A estratégia em questão permite otimizar o uso de CPU sem perder *deadlines*.

1.2.3 Transição entre as Etapas de Desenvolvimento

Defende-se nesta tese que as transições entre as diversas etapas de desenvolvimento que compõem o ciclo de vida de um sistema computacional devem ocorrer de maneira suave, isto é, sem a ocorrência de descontinuidades. Dentro deste modelo, são feitas as seguintes considerações:

1. O resultado de um processo de análise e projeto orientado a objetos deve ser codificado em uma linguagem de programação também orientada a objetos;
2. A linguagem de programação utilizada deve possuir uma API ou biblioteca que permita a especificação de restrições temporais de maneira clara e separada da funcionalidade da aplicação;
3. O ambiente de execução utilizado deve ser flexível e principalmente garantir a execução das restrições temporais presentes nas aplicações.

Diante do contexto exposto, esta tese apresenta um conjunto de elementos que permitem superar as limitações discutidas. Na próxima seção são apresentados em detalhes os objetivos e as contribuições desta tese.

1.3 Objetivos e Contribuições da Tese

O objetivo geral desta tese é abordar a criação de um método de desenvolvimento integrado, capaz de cobrir todo o ciclo de desenvolvimento de uma aplicação tempo real orientada a objetos, abrangendo as etapas de modelagem, projeto, execução e validação temporal. De modo mais específico, objetiva-se preencher as lacunas verificadas na transição do modelo de análise – decorado com restrições temporais – para uma aplicação, i.e. mapeamento do modelo OO para código e do código para o ambiente de execução. Outro objetivo é oferecer um ambiente de execução com uma política de escalonamento flexível, o qual garante o cumprimento dos requisitos temporais da aplicação sem basear-se no uso de WCETs, aumentando assim os índices de utilização do mesmo. Além disso, objetiva-se também oferecer uma infra-estrutura para permitir a validação dos requisitos temporais modelados.

Em resumo, pretende-se cobrir as etapas de modelagem e projeto do sistema, oferecer uma infra-estrutura de execução adequada e suportar a validação dos requisitos temporais especificados. Assim, contribui-se para minimizar os problemas apontados na seção anterior durante a transição do modelo para a aplicação. A FIGURA 1.1 resume os elementos que constituem o método de desenvolvimento proposto, juntamente com os seus relacionamentos.

Analisando o processo evolutivo desta tese, percebe-se que a mesma promove o avanço da dissertação de mestrado do autor, que resultou na criação do ambiente SIMOO-RT [BEC 99]. O SIMOO-RT define o mapeamento de um modelo orientado a objetos decorado com restrições temporais para a linguagem de programação AO/C++ [PER 94], sem fazer considerações a respeito do ambiente de execução da mesma. Nesta tese a proposta original foi adaptada para as novas tecnologias surgidas ao longo dos últimos 4 anos, dentre as quais destacam-se a padronização para descrição de requisitos

temporais [OMG 2002] e novas APIs de programação [KIM 2000; BOL 2001]. Na FIGURA 1.2 utiliza-se de um diagrama de classes para situar o escopo e a contribuição da presente tese (delimitado pelo pacote Tese) no escopo dos trabalhos relacionados e também dos elementos adicionais incluídos.

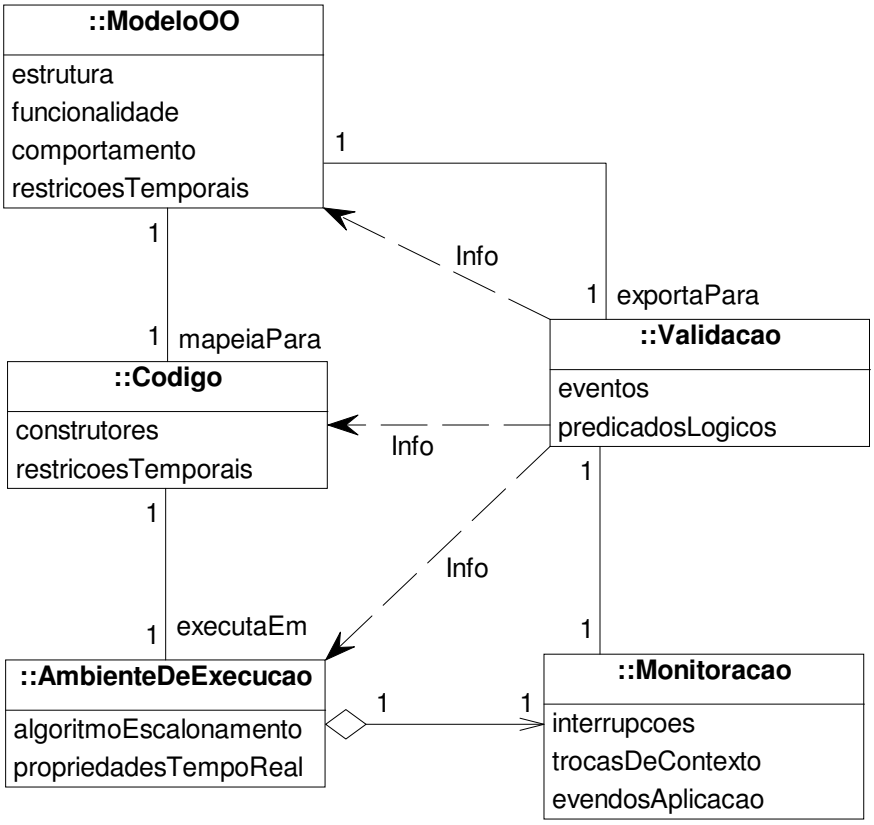


FIGURA 1.1 - Elementos envolvidos na metodologia proposta

Analisando em detalhes o pacote Tese, encontra-se a proposta do chamado ambiente de desenvolvimento integrado (*Integrated Environment*) [BEC 2001], que é fruto da extensão do SIMOO-RT [BEC 99] para permitir a monitoração das aplicações geradas com o auxílio da ferramenta Jewel [GER 97]. Este trabalho foi desenvolvido no âmbito da cooperação Brasil-Alemanha, junto com os pesquisadores da Universidade de Magdeburg, onde o autor realizou o seu estágio de doutorado sanduíche. Posteriormente se acrescentou um outro componente no ambiente integrado, a ferramenta VCAT (*Visual Communication Analysis Tool*) [WIL 2000], para permitir uma análise mais detalhada das informações monitoradas. Para que a mesma pudesse trabalhar com os arquivos de dados utilizados pelo Jewel, projetou-se uma nova interface no VCAT para leitura de dados.

Ainda dentro do escopo da cooperação com a Universidade de Magdeburg, foi definido um novo esquema de escalonamento para o TAFT, o qual baseia-se no uso de algoritmos do tipo *Earlieste Deadline*. O esquema proposto é utilizado para implementar um ambiente de execução baseado na proposta do TAFT (vide [BEC 2001b]). O objetivo do ambiente projetado é evitar situações de sobrecarga artificial, mantendo um índice elevado de utilização de CPU enquanto garante que deadlines são cumpridos. Além disso, a estratégia adotada permite tratar situações de sobrecarga transiente provendo códigos de tratamento de exceção, negociando funcionalidade por

garantias de execução. Com isto, consegue-se suprir as carências apresentadas pelo ambiente de execução originalmente utilizado com o SIMOO-RT. Este ambiente é representado na FIGURA 1.2 através da classe FT-Scheduler. Ressalta-se que este ambiente de execução mantém os recursos para monitoração do ambiente integrado [BEC 2001].

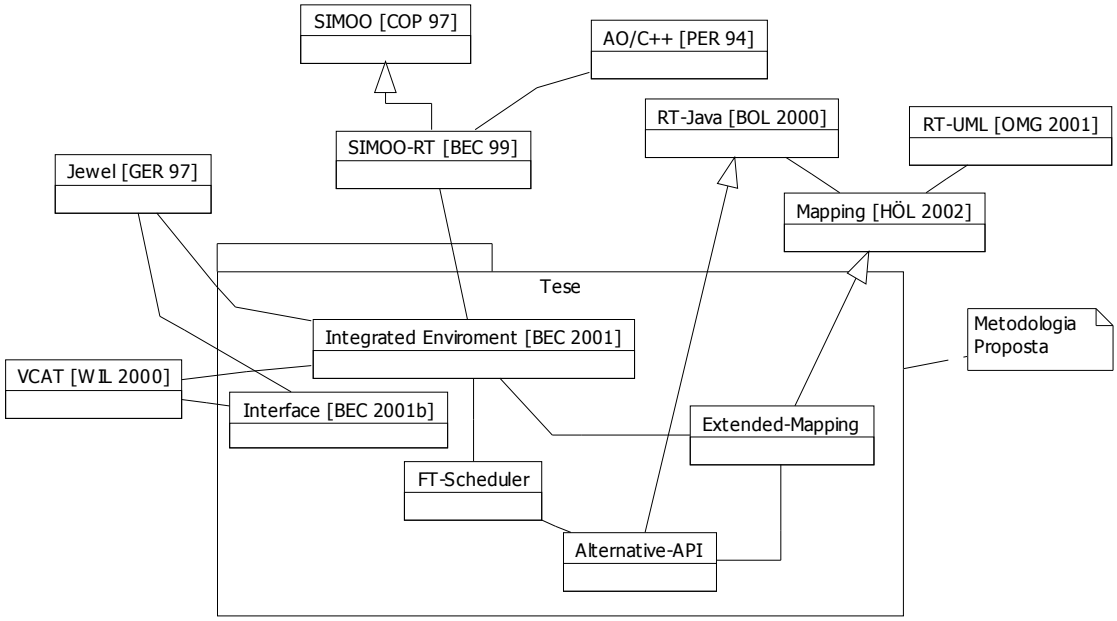


FIGURA 1.2 - Elementos envolvidos na tese proposta

Outro componente importante no escopo da tese é a proposta de mapeamento de restrições temporais padronizadas em modelos orientados a objetos para o nível de programação. O enfoque desta tese está voltado para a UML, uma vez que esta fora adotada pela OMG como linguagem padrão para a modelagem de sistemas orientados a objetos. Através de estudos de casos desenvolvidos no âmbito do trabalho de conclusão de mestrado profissional de Höltz [HÖL 2002], verificou-se que é possível representar os elementos presentes no *framework* para análise de escalonabilidade do perfil UML-TR através de construtores de programação padronizados, como a API do RTSJ. Nesta tese apresenta-se uma extensão do trabalho de Höltz para cobrir também os *frameworks* para modelagem de tempo e de concorrência. Na FIGURA 1.2 ela é representada pela classe *Extended-Mapping*.

Para relacionar o mapeamento citado com a API do ambiente de execução desenvolvido, modificou-se a implementação das bibliotecas utilizadas para trabalhar com as primitivas que interagem com o sistema operacional utilizado. Denominou-se o resultado desta API de TAFT-API, conforme também ilustrado na FIGURA 1.2.

Com base nos objetivos e elementos expostos, destacam-se a seguir as principais contribuições da tese:

- Na área de modelagem e implementação:
  - Oferece uma proposta de mapeamento de requisitos temporais (descritos usando a notação padronizada pelo perfil UML-TR) para uma linguagem de programação orientada a objetos, utilizando para uma API para a programação dos requisitos temporais

- Na área de programação:
  - Define a implementação da chamada TAFT-API, a qual é baseada na especificação do Java tempo real e permite a programação dos concentos propostos pelo TAFT;
- Na área de escalonamento:
  - Propõe uma nova estratégia de implementação para o mecanismo de escalonamento TAFT, que tolera eventuais falhas temporais no sistema, a qual é mais flexível e eficiente do que as implementações anteriores;
  - Implementa um ambiente de execução que adota o mecanismo de escalonamento proposto, oferecendo garantias de execução para aplicações tempo real;
- Na área de monitoração:
  - Apresenta uma arquitetura capaz de monitorar e também promover uma análise qualitativa das restrições temporais presentes nas aplicações executadas no ambiente proposto.

1.4 Organização do Volume

O restante desta proposta encontra-se organizado da seguinte forma. No próximo capítulo é discutida a construção de modelos orientados a objetos decorados com restrições temporais. Posteriormente, no capítulo 3, apresenta-se um conjunto de tecnologias de programação baseadas no paradigma de orientação a objetos e que suportam a especificação de restrições temporais. No capítulo 4 é feita uma análise crítica do perfil UML-TR, comparando as alternativas de modelagem oferecidas com o suporte existente para a sua programação. Já no capítulo 5 é detalhada a proposta do ambiente de execução para escalonar dinamicamente tarefas tempo real, com a garantia de cumprir os *deadlines* especificados para aquelas tarefas admitidas para execução. Apresenta-se no capítulo 6 o projeto da TAFT-API, usada na implementação das restrições do UML-TR para o ambiente TAFT. Logo após é discutido no capítulo 7 o estudo de caso que demonstra o mapeamento de modelos UML-TR para a TAFT-API. Já no capítulo 8 apresenta-se a arquitetura proposta para permitir a validação dos requisitos temporais especificados durante a modelagem do sistema. Por fim, no capítulo 9 apresentam-se as conclusões obtidas e os trabalhos futuros que devem dar continuidade a esta tese.

2 Restrições Temporais em Modelos Orientados a Objetos

Até março de 2002 não existia um consenso sobre como as restrições tempo real eram expressas em modelos orientados a objetos, onde neste cenário cada autor e/ou metodologia propunha o seu próprio conjunto de restrições. O próprio ambiente SIMOO-RT [BEC 99] é um exemplo desta falta de consenso, pois definia o seu próprio conjunto de restrições e também uma notação própria.

Uma dificuldade em se padronizar o uso de restrições temporais em modelos orientados a objetos se deve ao fato dos próprios diagramas não serem padronizados. Estes problemas passaram a ser superados a partir de 1997, momento em que o *Object Management Group* (OMG) adotou a *Unified Modeling Language* (UML) [BOO 99] como notação padronizada para a modelagem de sistemas orientados a objetos. Com isto, a UML se popularizou e acabou sendo abordada por diversos autores como alternativa para a modelagem de sistemas tempo real orientados a objetos.

Todavia, a falta de padronização para a especificação de restrições temporais (agora em diagramas UML) prevaleceu até o surgimento do perfil para o UML tempo real [OMG 2002], ou simplesmente perfil UML-TR, recentemente padronizado pela OMG. Este último utiliza os mecanismos de extensão da UML para padronizar os diversos tipos de restrições existentes em aplicações tempo real, como por exemplo restrições de tempo, de performance, de concorrência, entre outras.

Neste capítulo faz-se uma perspectiva em relação aos avanços na descrição de requisitos temporais em modelos orientados a objetos até se chegar no perfil UML-TR.

2.1 Propostas Iniciais

Em [BEC 97] analisaram-se algumas ferramentas CASE para modelagem de sistemas OO, com o objetivo de levantar o suporte à especificação de requisitos temporais. Nesta análise se destacaram as ferramentas *Rational Rose*, *ObjecTime* e SIMOO.

O *Rational Rose* é uma ferramenta voltada para análise e projeto de sistemas OO, com suporte às notações do OMT [RUM 91], Booch [BOO 91] e também da versão inicial da UML. Talvez por ser uma ferramenta voltada para a modelagem de sistemas de propósitos gerais, onde o fator "determinismo temporal" não é preponderante, encontrou-se pouco suporte à especificação de requisitos temporais. As poucas exceções dizem respeito a anotações não padronizadas que poderiam ser feitas junto aos diagramas modelados e também como comentários nos métodos de classe.

Já o *ObjecTime* é uma ferramenta especificamente voltada para a construção de sistemas tempo real. Baseada na metodologia ROOM de Selic et al [SEL 94], tem como base a definição de *timers* para especificar restrições temporais. Os *timers* são criados para servir como cão de guarda (*watchdog*) àquelas operações consideradas essenciais, bem como para indicar ocorrências periódicas.

Por fim, analisou-se também a ferramenta SIMOO [COP 97], voltada para a construção de modelos OO de simulação. Por não possuir suporte à definição de requisitos temporais, mas devido às suas características de modelagem e simulação, o

SIMOO acabou sendo estendido para incorporar a definição de requisitos temporais. O resultado desta extensão foi a criação do ambiente SIMOO-RT [BEC 99], detalhado na próxima subseção.

2.1.1 Ambiente SIMOO-RT

Por decisões de projeto e também por questões históricas[2], nem todos os diagramas do SIMOO-RT seguem a notação proposta pela UML. Apesar disso, é possível afirmar que os diagramas do SIMOO-RT possuem nível de abstração similar aos diagramas propostos em UML, sendo inclusive possível ter-se um mapeamento entre os mesmos.

A principal alteração do SIMOO-RT em relação ao seu antecessor, o SIMOO, é a capacidade de permitir a associação de restrições temporais aos métodos das classes ativas[3]. Estas restrições permitem ao usuário especificar métodos cíclicos, métodos com *deadlines* e, quando conveniente, operações de tratamento especial para aqueles métodos que não cumprem o seu *deadline*. As restrições das operações são especificadas através de uma janela gráfica especial, que contém as propriedades dos métodos. Além das restrições associadas aos métodos, também é possível acrescentar ao modelo restrições mais genéricas, como a prioridade em que o objeto irá executar.

Mesmo após a conclusão da dissertação de mestrado que resultou no ambiente SIMOO-RT, continuou-se o estudo de novas alternativas para inserção de requisitos temporais no modelo OO. Com a popularização da UML, acrescentou-se uma ferramenta no ambiente, com suporte para a edição de diagramas de seqüência. Neste novo diagrama, que descreve a interação entre um grupo de objetos ao longo do tempo, é possível também anotar restrições temporais. De acordo com a proposta elaborada, existem três tipos de restrições que podem ser associadas ao diagrama, conforme definição a seguir:

1. **Restrição de ativação**: a mensagem que dá inicio ao diagrama de seqüência pode ser classificada como periódica ou episódica. A primeira é implicitamente associada a um *timer* e possui, à sua esquerda, o valor do período. Já a segunda está associada com mensagem vinda de outro objeto, além de ser inerentemente imprevisível. Opcionalmente, pode-se definir o valor do menor intervalo entre duas ocorrências. A notação gráfica que diferencia ocorrências periódicas de ocorrências episódicas é mostrada na FIGURA 2.1.
2. **Tempo máximo de execução**: as operações ativadas em resposta às mensagens recebidas podem ter um *deadline* (tempo máximo de execução) associado, definido à direita do instante em que o objeto recebe a mensagem (vide por exemplo a FIGURA 2.1).
3. **Timeout**: as mensagens trocadas entre os objetos de forma síncrona podem possuir um *timeout* (tempo máximo de espera) associado, bastando para tanto uma anotação à esquerda da mensagem (vide *extCall()* na FIGURA 2.2) Esta restrição se aplica às mensagens do tipo *call* e *waiting*, definidas segundo o padrão da UML [BOO 99].

[2] É importante notar que o desenvolvimento das primeiras versões do ambiente SIMOO-RT antecede inclusive ao da promulgação pela OMG de UML como um padrão para sistemas orientados a objeto

[3] Entidade cujas instâncias têm execução autônoma, i.e. sua própria *thread* de controle.

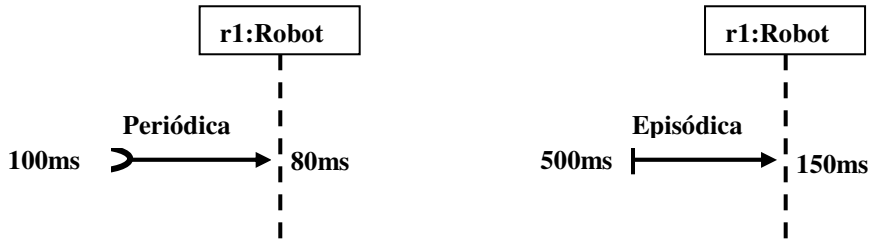


FIGURA 2.1 - Notação associada com a mensagem inicial do diagrama de seqüência

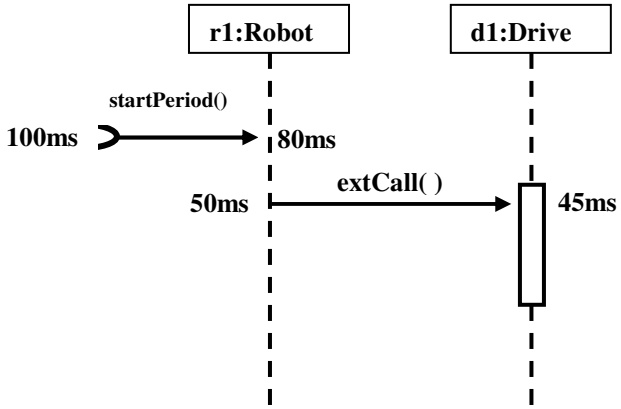


FIGURA 2.2 - Exemplo de restrição em comunicação assíncrona

Nas próximas seções são descritas as principais propostas de extensão da UML para tempo real e também o recém adotado perfil UML-TR.

2.2 Extensões da UML para Tempo Real

A proposta de extensão da UML para tempo real apresentada por Selic em [SEL 99] utiliza os principais conceitos da metodologia ROOM [SEL 94], com o argumento de aumentar a precisão das especificações. Conceitos como cápsulas, protocolos, papéis nos protocolos e portas são acrescentados na UML através de estereótipos (*stereotypes*). Conforme citado no início do capítulo, no ROOM os requisitos temporais são controlados por instâncias da classe *Timer*.

Por outro lado, a proposta descrita por Douglass [DOU 98] mantém a sintaxe da UML praticamente inalterada. Nesta proposta, denominada ROPES (*Rapid Object-Oriented Process for Embedded Systems*), tem-se como maior contribuição a definição de uma seqüência de passos de análise e projeto voltados para a construção de sistemas tempo real embutídos. Além disso, esta proposta sugere a associação de marcas (*tags*) temporais no diagrama de seqüência. No entanto, tais anotações não possuem qualquer significado semântico para a UML, podendo ser considerados como comentários associados aos diagramas. Também é proposto um novo diagrama, denominado *Timing Diagrams* usado para representar a evolução dos estados de objetos ativos ao longo do tempo.

Já a proposta de Gomaa [GOM 2000] apresenta o método COMET, *Concurrent Object Modeling and Architectural Method*. Neste método, os requisitos funcionais do

sistema são definidos em termos de atores e cenários, através de diagramas *Use Case*[4]. 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 estereótipos, 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*», entre outros.

Fazendo uma análise crítica, observa-se que as extensões da UML para tempo real citadas nesta seção ainda necessitam utilizar uma semântica mais específica, deficiência reconhecida até mesmo pelos proponentes das extensões. Um dos problemas enfrentados é a tentativa de minimizar o número de conceitos a serem introduzidos na UML, a fim de manter a compatibilidade com o padrão original. Atualmente, procura-se manter este compromisso através do uso do conceito de perfis (*profiles*), propostos por UML como alternativa comum para se modelar características específicas, voltadas para um determinado tipo de problema. Por exemplo, recentemente foi adotado o perfil UML-TR, desenvolvido pelo comitê técnico da OMG nas áreas de análise e projeto de sistemas tempo real. Seu objetivo é a definição de padrões para a modelagem de características relacionadas com escalonabilidade, desempenho e tempo em sistemas tempo real, visando especialmente:

- Permitir a construção de modelos que possam ser usados em análises quantitativas a respeito das características mencionadas;
- Facilitar a comunicação entre profissionais da área sobre aspectos de projeto de maneira padronizada;
- Permitir a interoperabilidade entre ferramentas de análise e projeto.

Na próxima seção discute-se em detalhes o perfil UML-TR.

2.3 Perfil para o UML Tempo Real

O Perfil UML para Escalonabilidade, Performance e Tempo (*UML Profile for Schedulability, Performance and Time*) [OMG 2002], ou simplesmente perfil UML-TR[5] (UML-TR), foi desenvolvido pelo grupo de especialistas da OMG para análise e projeto de sistemas tempo real. O principal requisito desta especificação é permitir a modelagem de sistemas determinísticos e previsíveis temporalmente com o uso da UML. Um sistema é caracterizado como determinístico quando as reações aos eventos podem ser quantificadas e conhecidas com antecedência. O mesmo é definido como previsível temporalmente quando suas características temporais são conhecidas e limitadas. Exemplos destas características são o instante de ativação (*release time*), tempo de execução, *deadlines*, prioridade, entre outros. Para tanto, são necessárias informações quantitativas (relacionadas com o "fator tempo") referentes à implementação de uma determinada funcionalidade.

[4] 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].
[5] Denominação perfil do UML-TR proveniente do artigo de Selic em [SEL 2002].

A estrutura do perfil UML-TR é dividida de maneira hierárquica, contendo uma parte genérica, comum a todos os elementos do modelo, e também partes especializadas, que abordam requisitos e funcionalidades específicas. A parte genérica é denominada Modelo Geral de Recursos, cujo núcleo é um *framework* para a modelagem de recursos. Dois outros *frameworks* derivam-se do primeiro: um para modelagem de tempo e outro para modelagem de concorrência. As subpartes especializadas são divididas em um módulo para definição de métodos de análise e outro para definição da infra-estrutura de execução. O módulo contendo os métodos de análise sugere um *framework* para análise de escalonabilidade, enquanto o segundo sugere o mapeamento para um ambiente de execução específico (e.g. RT-CORBA). A FIGURA 2.3 ilustra a composição desta estrutura, onde as características dos elementos destacados são ressaltadas nas próximas subseções.

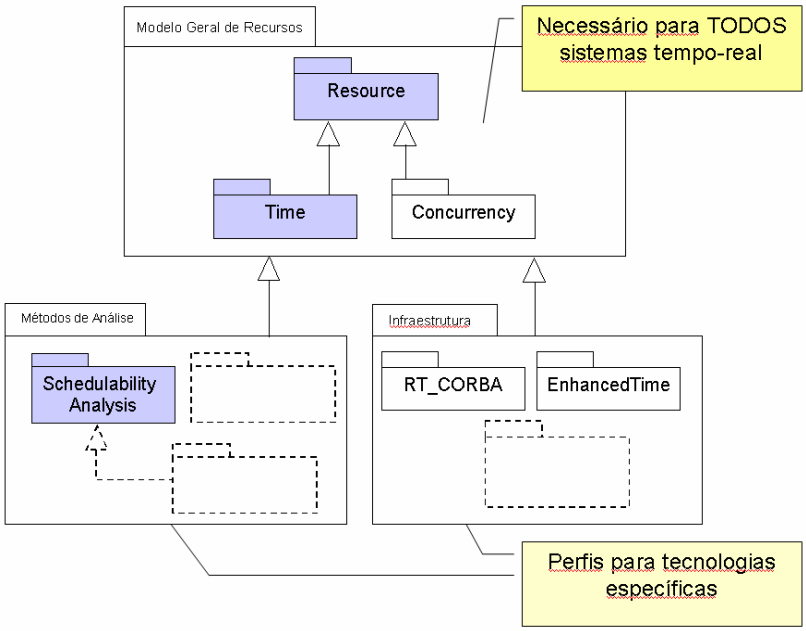


FIGURA 2.3 - Estrutura geral do perfil UML-TR

2.3.1 Framework para Modelagem de Recursos

O *framework* para modelagem de recursos (*RTresourceModeling*) serve de base para que o projetista possa expressar os requisitos de qualidade de serviço (QoS) provenientes da aplicação. O modelo conceitual defendido neste perfil, constituído por recursos, serviços e requisitos de QoS, é formado pelos elementos exibidos no diagrama UML da FIGURA 2.4. Este modelo faz uma distinção clara entre os descritores (especificação) e as entidades em tempo de execução que implementam os requisitos. Os elementos no lado direito do diagrama representam os requisitos, sendo que cada um possui um tipo de instância correspondente. É importante ressaltar que os elementos mostrados neste diagrama não precisam necessariamente representar um estereótipo no perfil UML-TR, visto que neste caso a maioria deles representa entidades abstratas.

Também é importante o conceito denominado Análise de Contexto, que envolve a representação dos elementos presentes em uma determinada situação de uso. Tal análise representa uma demanda por recursos que pode ser determinada de maneira dinâmica ou

estática, variando de acordo com as necessidades do projetista. Um exemplo simples de análise estática é a invocação de um método, onde o objeto cliente indica o seu requisito de QoS para a chamada (e.g. tempo máximo de retorno) e o objeto servidor contém requisito de QoS oferecido (e.g. tempo máximo de resposta) para o método em questão. Por outro lado, uma análise dinâmica permite representar situações mais complexas, quase sempre presentes em aplicações reais.

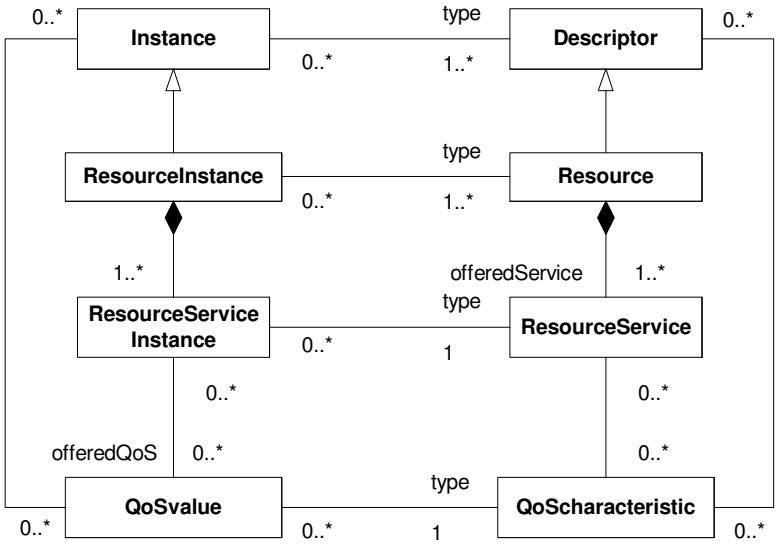


FIGURA 2.4 - Componentes básicos do modelo geral de recursos

2.3.2 Framework para Modelagem de Tempo

Outro elemento pertencente à base comum do perfil UML-TR é o *framework* para modelagem de tempo, derivado do *framework* para modelagem de recursos, onde são introduzidos conceitos para a especificação de restrições temporais. Os conceitos aqui definidos são associados com estereótipos iniciados pelo prefixo *RT*. Este *framework* é particionado em quatro módulos distintos, porém relacionados entre si, conforme segue:

1. Módulo para modelagem de tempo e valores de tempo (*TimeModel*);
2. Módulo para modelagem de mecanismos de tempo (*TimingMechanisms*);
3. Módulo para modelagem de eventos no tempo (*TimedEvents*);
4. Módulo para modelagem de serviços temporais (*TimingServices*).

O módulo para modelagem de tempo discute os detalhes relacionados com a conceituação de tempo. É definido neste módulo o conceito de valores de tempo (*time value*), que se referem a uma medida de tempo. Também é definido o conceito de duração (*duration*), que representa o tempo passado entre dois instantes, sendo representada neste módulo por um intervalo de tempo (*time interval*).

Os mecanismos de tempo oferecem duas facilidades para a modelagem da infra-estrutura e mecanismos temporais: *timers* e *clocks*. Os *timers* podem ser programados para gerarem um evento quando um determinado ponto no tempo for alcançado. O instante de tempo para geração do evento pode ser tanto definido de forma absoluta (tal como, "às 10 horas") ou relativa ("após 15 minutos"). Já os *clocks* são mecanismos que geram periodicamente uma interrupção. O valor anotado pelo *clock* em um determinado

instante de tempo é denominado *timestamp*. Sua representação se dá através de uma instância de um *time value*. Este último pode ser representado por valores inteiros ou reais, e até mesmo por tipos de dados estruturados mais sofisticados, tal como datas ou contagens do *clock*.

Já os eventos, definidos no módulo para modelagem de eventos no tempo, são caracterizados por não possuírem duração, sendo associados apenas com um instante no tempo. Um tipo especial de evento é denominado *timed event*, que indica a chegada em algum instante de tempo pré-determinado. Isto pode ser conseqüência da expiração de um certo intervalo de tempo (e.g. *timeout*) ou da leitura de um *clock*.

Por fim, o modelo para modelagem de serviços temporais trata daqueles mecanismos essenciais para a programação de requisitos temporais, geralmente presentes nos sistemas operacionais tempo real. Como exemplos destes serviços tem-se as operações de leitura e acerto de relógios e também as operações de criação e manutenção de temporizadores (*timers*).

A especificação dos limites de tempo pode ser feita neste *framework* de duas formas exclusivas entre si: ou através da definição das marcas *RTstart* e *RTend*, que denotam os tempos de início e de fim de um estímulo ou ação, ou diretamente através da marca *RTduration*. As duas primeiras marcas podem ser associadas ou com um valor de tempo, representado pelo estereótipo *«RTtime»*, ou também com um intervalo, representado pelo estereótipo *«RTinterval»*. Já a duração está sempre associada com um intervalo. Ressalta-se que na prática não existe diferença entre *«RTtime»* e *«RTinterval»*, visto que a segunda nada mais é do que um caso especial da primeira. Também se chama a atenção pelo fato de que as marcas *RTstart*, *RTend* e *RTduration* podem ser representadas pela notação TVL (*Tag Value Language*), a qual é definida pelo perfil UML-TR (vide apêndice A de [OMG 2002]). Esta notação suporta diversos tipos de construções, entre elas algumas funções de distribuição de probabilidade.

A FIGURA 2.5 exemplifica o uso deste *framework*, onde a notação TVL é empregada para se definir limites de tempo usando as marcas *RTstart* e *RTend* para as mensagens relacionadas com a chamada de um método, estereotipadas com *«RTstimulus»,* e também para a ação tomada em resposta ao atendimento da mensagem, i.e. a próprioa execução do método, estereotipada com *«RTaction»*. Uma versão alternativa para este diagrama é apresentada na FIGURA 2.6, onde ao invés de limites de tempo fixo usa-se intervalos baseados em funções de distribuição de probabilidade.

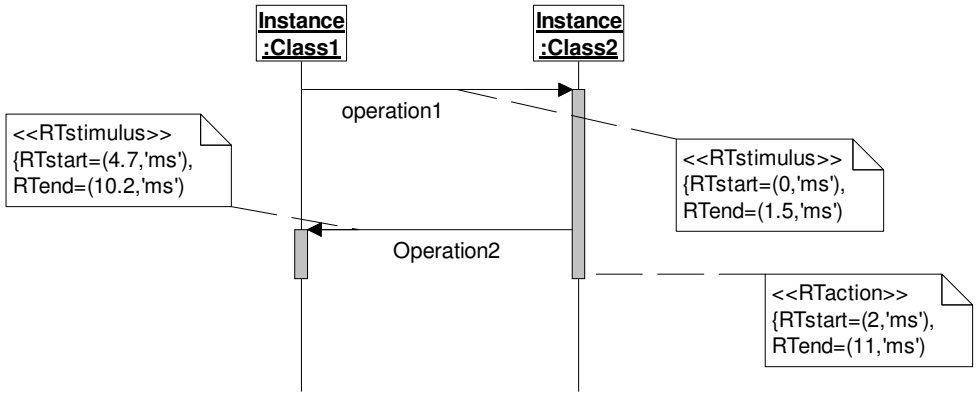


FIGURA 2.5 - Modelagem de limites de tempo usando restrições fixas

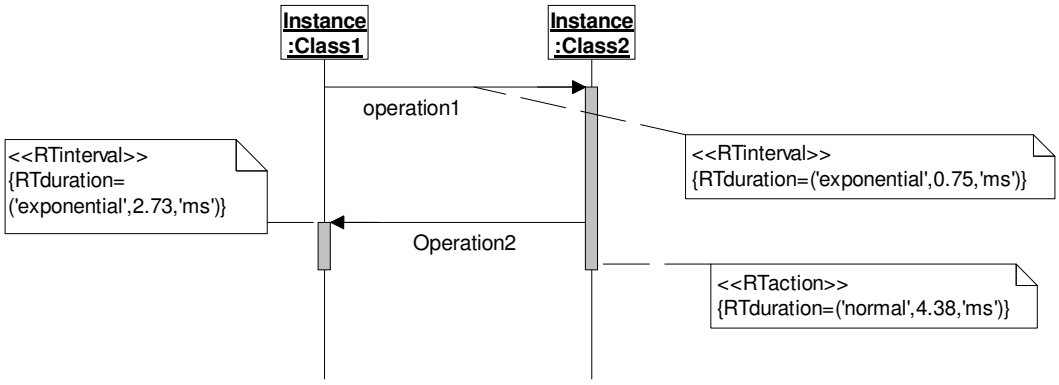


FIGURA 2.6 - Modelagem de limites de tempo usando intervalos

Para concluir esta seção, lista-se na TABELA 2.1 os componentes do *framework* para modelagem de tempo.

TABELA 2.1 – Elementos do *framework* para modelagem de tempo

Elemento do perfil UML-TR	Descrição
«RTaction»	Estereótipo que delimita ação realizada por um bloco de código ou um método.
~.RTstart	Marca para definir o tempo de início da ação.
~.RTend	Marca para definir o tempo de término da ação.
~.RTduration	Marca para definir a duração da ação.
«RTclkInterrupt»	Estereótipo que modela uma interrupção de *clock*.
«RTclock»	Estereótipo que modela um *clock*.
~.RTclockid	Marca para caracterizar o *clock*.
«RTdelay»	Estereótipo que caracteriza uma ação para bloquear a entidade que a chama por um tempo determinado.
«RTevent»	Estereótipo que caracteriza a ocorrência de um evento.
~.RTat	Marca que contém informações sobre o instante de ocorrência e o padrão de ativação do evento.
«RTinterval»	Estereótipo para modelar um intervalo de tempo.
~.RTintStart	Marca para definir o tempo de início do intervalo.
~.RTintEnd	Marca para definir um tempo de término do intervalo.
~.RTintDuration	Marca para definir a duração do intervalo.
«RTnewClock»	Estereótipo que caracteriza a criação de um *clock*.
«RTnewTimer»	Estereótipo que caracteriza a criação de um *timer*.
~.RTtimePar	Marca para definir os parâmetros do *timer*.
«RTpause»	Estereótipo que pára um *clock* ou *timer*.
«RTreset»	Estereótipo que reinicializa um *clock* ou *timer*.
«RTset»	Estereótipo que define o valor de um *clock* ou *timer*.
~.RTtimePar	Marca utilizada para definir os parâmetros do *clock* ou *timer*.
«RTstart»	Estereótipo que inicia um *clock* ou *timer*.
«RTstimulus»	Estereótipo que modela um estímulo temporizado.
~.RTstart	Marca para definir o tempo de início do estímulo.
~.RTend	Marca para definir o tempo de término do estímulo.
«RTtime»	Estereótipo que modela um instante de tempo.
~.RTkind	Marca para definir o tipo de tempo (denso ou contínuo).
~.RTrefClk	Marca para definir o *clock* de referência.
«RTtimeout»	Estereótipo que modela um sinal ou ação de *timeout*.

«RTtimer»	Estereótipo que define um objeto do tipo *timer*.
~.RTduration	Marca para definir a duração do *timer*.
~.RTperiodic	Marca para definir se o *timer* é periódico.
«RTtimeService»	Estereótipo que define um "serviço de tempo"
«RTtimingMechanism»	Estereótipo abstrato que serve como base para *clocks* e *timers*, que são chamados de "mecanismos de tempo".
~.RTstability	Marca para definir a habilidade do mecanismo de tempo para fornecer intervalos consistentes.
~.RTdrift	Marca para definir a diferença absoluta máxima entre a freqüência do mecanismo de tempo em relação à freqüência do seu *clock* de referência.
~.RTskew	Marca para definir a taxa de mudança do *offset* entre o mecanismo de tempo e o seu *clock* de referência.
~.RTmaxValue	Marca para definir o valor máximo que o mecanismo de tempo pode assumir.
~.RTorigin	Marca para definir um evento cuja ocorrência representa o início da contagem.
~.RTresolution	Marca para definir a resolução do mecanismo.
~.RToffset	Marca para definir o desvio do mecanismo.
~.RTacurracy	Marca para definir a precisão do mecanismo.
~.RTcurrentVal	Marca para definir o valor corrente do mecanismo.
~.RTrefClk	Marca para definir o *clock* de referência.

2.3.3 Framework para Modelagem de Concorrência

Este *framework* foi elaborado tendo em vista que concorrência é um aspecto fundamental na maioria dos sistemas tempo real distribuídos, visto que existe uma forte interação destes com o meio físico, que é inerentemente concorrente. Os estereótipos definidos neste *framework* iniciam com o prefixo *CR*. O mesmo encontra-se dividido segundo o diagrama de classes apresentado na FIGURA 2.7, sendo que os seguintes pontos específicos são abordado em detalhes:

1. Recursos concorrentes: são os recursos que representam os mecanismos para um comportamento concorrente (ou pseudo-concorrente) no sistema, sendo representados no sistema operacional por processos ou tarefas;
2. Cenários concorrentes: representam seqüências de ações geralmente interligadas efetuadas por recursos concorrentes;
3. Serviços de recursos concorrentes: representam serviços que possuem algum tipo de política de controle de acesso que os protegem contra os efeitos indesejados da própria concorrência.

Recursos concorrentes (representado por *ConcurrentUnit* no modelo conceitual) são modelados como recursos ativos que, a partir da criação, iniciam a executar um cenário principal (denominado '*main*') e continuam até o cenário terminar ou até que seja explicitamente abortado. Este elemento pode possuir uma ou mais filas para armazenar mensagens ou estímulos vindas de outros recursos, denominada *StimuliQueue*, as quais não necessitam ser imediatamente processadas, i.e. são deferidas. Isto se aplica aos serviços denominados *deferred*. Por outro lado, serviços *immediate* implicam na criação de um novo cenário de execução, o qual necessita de uma unidade concorrente auxiliar para poder executar. Esta unidade pode ser criada por demanda ou pré-alocada em filas (*pools*) dela mesma.

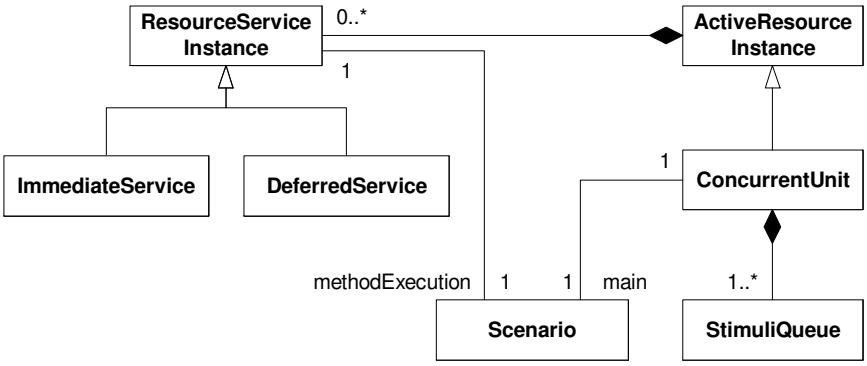


FIGURA 2.7 – Modelo geral de concorrência

Esta questão representa um ponto chave neste modelo, i.e. determina que o próprio recurso concorrente é quem decide quando a resposta a uma solicitação de serviço deverá ser executada. Esta decisão é tomada ao longo do curso da operação '*main*', que pode ficar bloqueada aguardando a chegada de um novo elemento na fila de estímulos, ou pode realizar verificações de tempos em tempos sem causar bloqueios, i.e. *poolings*. Esta capacidade diferencia o recurso concorrente de recursos passivos, onde a decisão sobre o momento da execução das ações solicitadas é tomada pelo cliente.

O modelo de concorrência faz ainda a distinção entre as ações de comunicação, i.e. solicitações de serviços, que podem ser síncronas ou assíncronas. A primeira bloqueia o cliente até que o serviço invocado seja completado, enquanto na segunda o cliente permanece com o controle do fluxo de execução.

Na FIGURA 2.8 exemplifica-se o uso deste modelo. Neste exemplo, tem-se que um *clock* ativa a operação *dataDisplay()* periodicamente a cada 10 ms. Esta operação pertence a uma unidade concorrente, o que indica o início de um novo cenário. O processamento da operação invocada é deferido, i.e. não é processado imediatamente, portanto sua execução é feita de maneira assíncrona em relação a unidade concorrente que a invocou (*clock*). Neste cenário são invocadas duas operações síncronas, as quais executam no contexto (i.e. *thread*) da unidade que fez a invocação. Outro detalhe é que a segunda operação invocada, i.e. *display()*, ativa uma ação atômica, i.e. que não pode sofrer preempção.

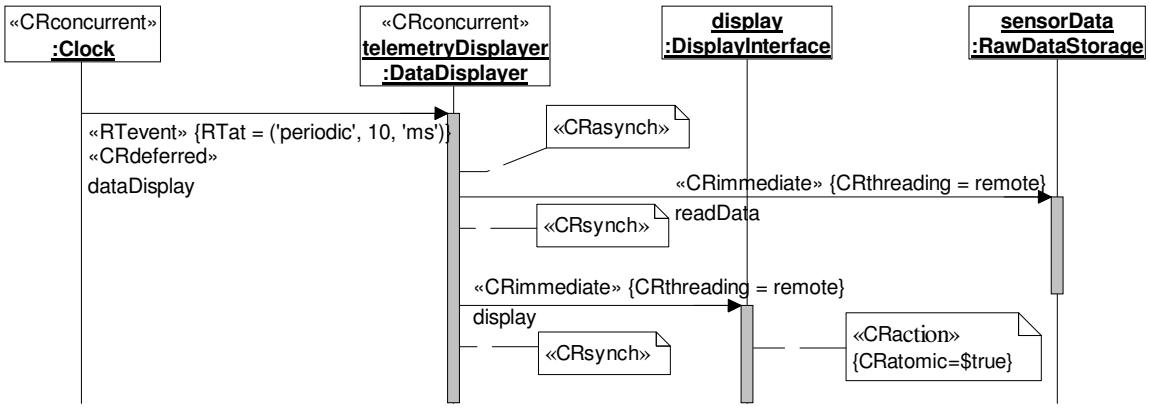


FIGURA 2.8 – Exemplo de utilização do modelo geral de concorrência

Para concluir esta seção, lista-se na TABELA 2.2 os estereótipos pertencentes ao *framework* para modelagem de concorrência.

TABELA 2.2 – Elementos do *framework* para modelagem de concorrência

Elemento do perfil UML-TR	Descrição
«CRaction»	Estereótipo que representa a execução de uma ação.
~.CRatomic	Marca para definir se a ação pode ou não ser preemptada.
«CRasynch»	Estereótipo que representa a invocação assíncrona de uma ação.
«CRconcurrent»	Estereótipo que representa o conceito de unidade concorrente.
~.CRmain	Marca para definir o método executado ao se criar a unidade concorrente.
«CRcontains»	Estereótipo usado para modelar diversos relacionamentos.
«CRdeferred»	Estereótipo que representa o conceito de um serviço cujo atendimento à solicitação é postergada.
«CRimmediate»	Estereótipo que representa o conceito de um serviço cujo atendimento à solicitação é imediato.
~.CRthreading	Marca para definir se um novo cenário (i.e. uma nova unidade concorrente) é ou não criado para atender à solicitação.
«CRmsgq»	Estereótipo que representa uma fila de mensagens.
«CRsynch»	Estereótipo que representa a invocação síncrona de uma ação.

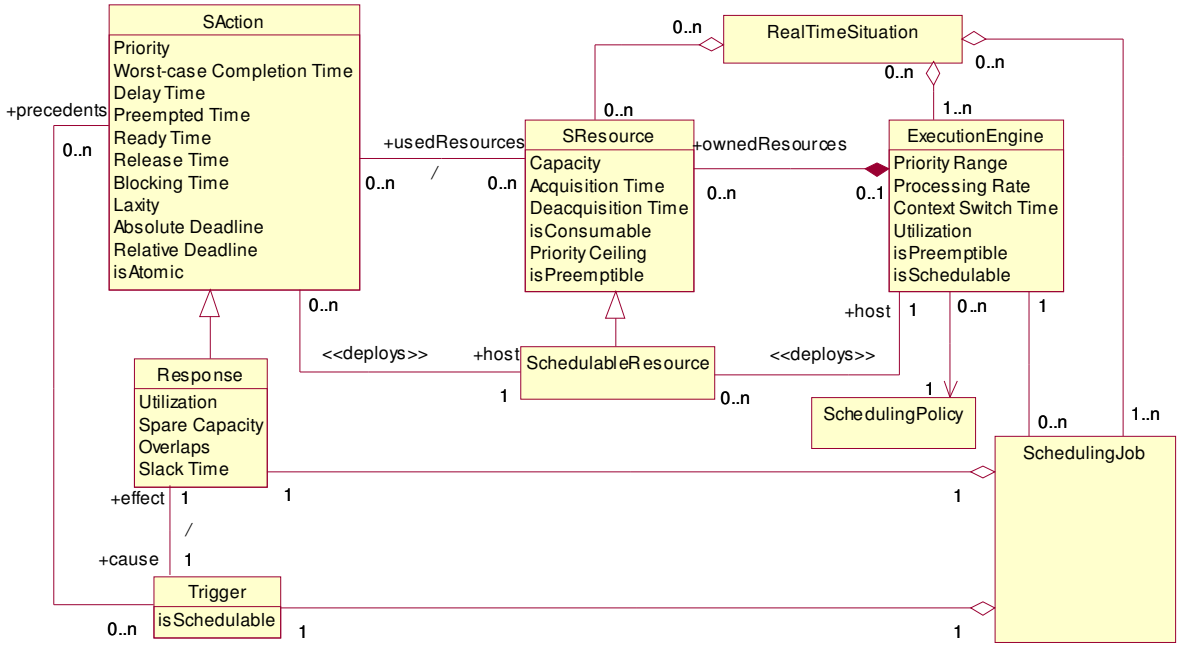


FIGURA 2.9 - Composição do *framework* para Análise de Escalonabilidade [OMG 2002]

2.3.4 Framework para Modelagem de Escalonabilidade

Para completar a descrição do perfil UML-TR, apresenta-se o *framework* para suportar a análise de escalonabilidade do modelo. Esta funcionalidade permite

determinar se as características de QoS solicitadas pelos clientes podem ser atendidas pelos recursos. Os estereótipos definidos neste *framework* iniciam com o prefixo *SA*.
Analisando a composição deste *framework*, mostrado na FIGURA 2.9, tem-se que *RealTimeSituation* representa um tipo especial de análise envolvendo dois recursos: ambiente de execução (*ExecutionEngine*), o qual executa cenários, e recurso protegido (*SResource*), que pode ser compartilhado por um ou mais cenários. Um evento ou estímulo (*Trigger*) representa

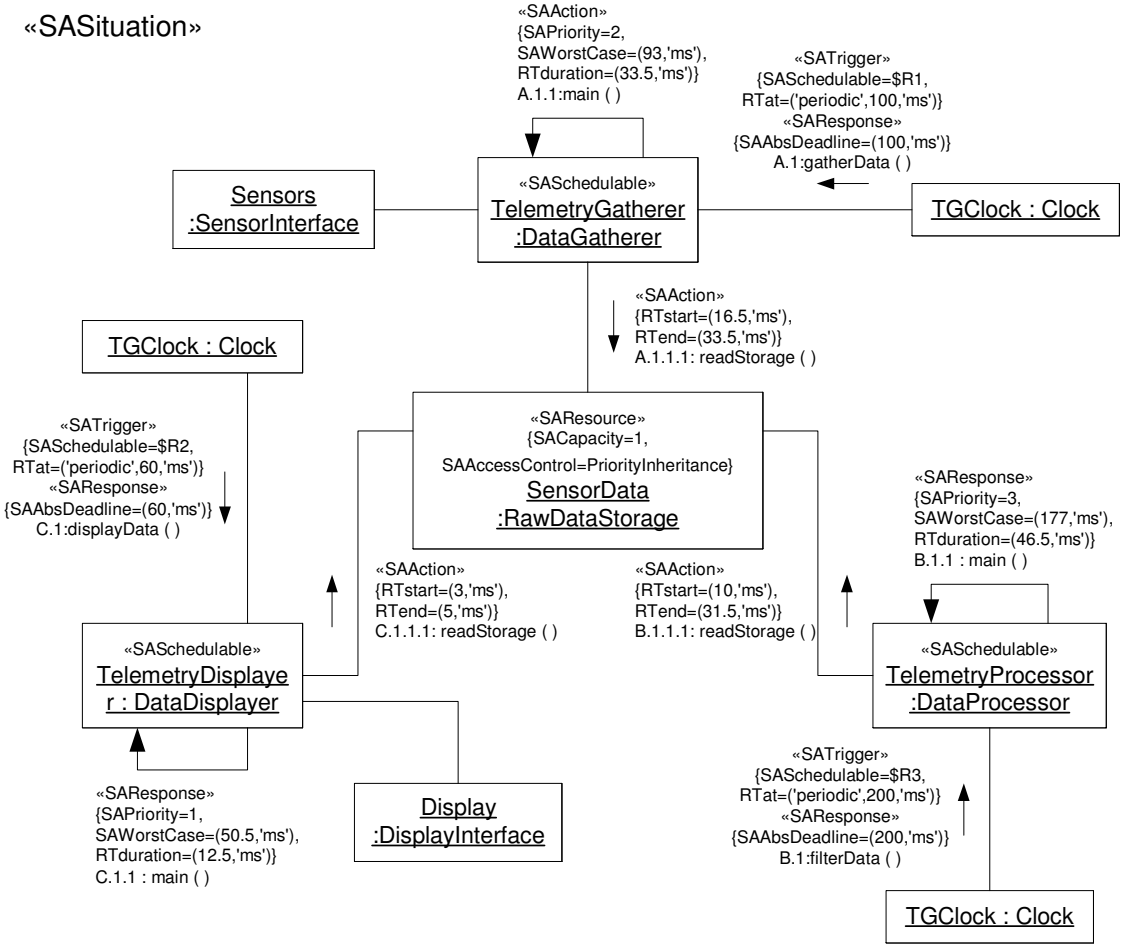


FIGURA 2.10 - Exemplo de situação tempo real [OMG 2002]

o acontecimento que insere carga de processamento no sistema. Mais especificamente, este evento representa o padrão de ocorrência de um cenário dinâmico, o qual é representado por uma resposta (*Response*). Este último representa um conjunto de ações, cujas características de escalonamento relevantes, tal como recursos utilizados, pior caso de tempo de execução (*Worst Case Execution Time* – WCET), entre outros, são definidos explicitamente. Estas características representam os valores de QoS solicitados e que precisam ser comparados com os valores oferecidos nos ambientes de execução e nos recursos utilizados. O elemento *SchedulingJob* indica a combinação de um estímulo com uma resposta. Já o chamado recurso escalonável (*SchedulableResource*) representa uma unidade concorrente no SO que é usada para executar os *SchedulingJob*.

Uma "situação tempo real" (*RealTimeSituation*) consiste tipicamente em múltiplos eventos e respostas utilizando um conjunto de recursos e ambientes de execução. A

análise é feita em modelos com instâncias, tipicamente constituída por uma colaboração que descreve eventos, respostas e recursos (juntamente com suas características). Um exemplo de situação tempo real é mostrado na FIGURA 2.10, onde se encontram três *SchedulingJobs* (*A*, *B* e *C*). Cada um deles é caracterizado por um evento e uma resposta, qualificados pelos devidos requisitos de QoS. Nota-se que os estereótipos do *framework* para análise de escalonabilidade iniciam pelo prefixo *SA*. Detalhando o *SchedulingJob A*, verifica-se que o mesmo tem início pela resposta denominada *gatherData()* (A.1), que possui ativação periódica a cada 100 ms e tem um *deadline* de mesmo valor. Esta resposta dispara a ação *main()* (A.1.1 - caracteriza a primeira ação a ser executada ao se processar uma resposta), que possui uma duração estimada de 33,5 ms e um WCET de 93 ms. A *main()* por sua vez executa outra ação, chamada *readStorage()* (A.1.1.1), cuja duração de 17 ms pode ser calculada através da diferença entre os tempos de início da ação (16,5 ms) e de final da mesma (33,5 ms). Esta última sofre controle de exclusão mútua por acessar o recurso compartilhado *RawDataStorage*, sendo coordenada de acordo com a política de controle de acesso estabelecida pelo recurso, que neste caso é a herança de prioridade (*Priority Inheritance*).

Para concluir esta seção, lista-se na TABELA 2.3 os estereótipos pertencentes ao *framework* para modelagem de concorrência.

TABELA 2.3 – Elementos do *framework* para análise de escalonabilidade

Elemento do perfil UML-TR	**Descrição**
«SAaction»	Estereótipo que delimita uma ação, a qual pode estar contida em um bloco de código ou método.
~.*SApriority*	Marca utilizada para definir a prioridade da ação sob o ponto de vista do algoritmo de escalonamento.
~.*SAblocking*	Marca utilizada para indicar o tempo que a ação ficou bloqueada aguardando por algum recurso compartilhado (*«SAresource»*).
~.*SAdelay*	Marca que indica o tempo que uma ação selecionada para execução despende enquanto solicitando e liberando recursos compartilhados.
~.*SApreempted*	Marca que indica a quantidade de tempo que uma tarefa em execução é preemptada por tarefas de maior prioridade.
~.*SAready*	Marca que indica o atraso entre o instante de tempo em que a ação fica pronta pra execução e o tempo de início do processamento.
~.*SArelease*	Marca que caracteriza o instante em que a tarefa fica pronta para execução.
~.*SAworstCase*	Marca que indica o tempo de execução de pior caso, ou seja, o mais longo tempo de execução possível para a ação.
~.*SAabsDeadline*	Marca que especifica o instante de tempo máximo para que a ação seja concluída (usada com deadlines rígidos – *hard*)
~.*SAlaxity*	Marca que especifica o "tipo" do deadline, i.e. *hard* ou *soft*.
~.*SArelDeadline*	Marca que especifica o instante de tempo desejado para que a ação seja concluída (usada com deadlines não-rígidos – *soft*).
~.*SAusedResource*	Marca utilizada para indicar todos os recursos (descritos por *«SAresource»*) utilizados pela ação.
~.*SAhost*	Marca que referência para o elemento do modelo (e.g. processador, computador - estereotipados como *«SAschedulable»*) que executa a ação.

«SAengine»	Estereótipo que representa um recurso do tipo proces-sador o qual é alocado para executar recursos de escalonamento.
~.SAaccessPolicy	Marca que caracteriza o conjunto de regras que governam as requisições a este recurso.
~.SAcontextSwitch	Marca que caracteriza o tempo necessário (sobrecarga) para preemptar entre uma tarefa e outra.
~.SAschedulable	Marca que caracteriza o resultado da análise, indicando se recurso de processamento é ou não escalonável.
~.SApreemptible	Marca que indica se o recurso pode ou não ser preemptado ao iniciar a execução de uma ação.
~.SApriorityRange	Marca que contém o conjunto de prioridades suportadas por este recurso.
~.SArate	Marca que indica uma velocidade de processamento relativa para o recurso.
~.SAschedulingPolicy	Marca que indica o algoritmo de escalonamento utilizado.
~.SAutilization	Marca que caracteriza o resultado da análise, indicando a taxa de utilização do recurso.
~.SAresources	Marca que indica os recursos de escalonamento executa-dos neste recurso de escalonamento.
«SAowns»	Estereótipo que caracteriza relações de implementações, i.e. quais recursos alocados a cada processador.
«SApreceds»	Estereótipo que caracteriza relações de precedência entre estereótipos e gatilhos.
«SAresource»	Estereótipo que caracteriza um recurso cujo acesso deve ser compartilhado por diversos recursos compartilhados, incluindo mecanismos de exclusão mútua.
~SAacquisition	Marca que indica o tempo passado entre solicitação e alocação do recurso.
~.SACapacity	Marca que define o número máximo de *threads* que podem acessar simultaneamente um recurso compartilhado.
~.SAdeacquisition	Marca que indica o tempo passado entre liberação e desalocação do recurso.
~.SAconsumable	Marca que indica se o recurso é "consumido" pelo uso.
~.SAaccessControl	Marca que 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.
~.SAptyCeiling	Marca que indica um valor calculado de acordo com a política de controle de acesso (e a ser usado pela mesma).
~.SApreemptible	Marca que indica se o recurso pode ou não ser preemptado durante o uso.
«SAresponse»	Estereótipo que caracteriza a "resposta" propriamente dita. Em conjunto com *«SAtrigger»* define o padrão de ativação e as propriedades de escalonamento de uma entidade de escalonamento.
~.SAutilization	Marca que indica o percentual do período da ação no qual o recurso de escalonamento é ocupado.
~.SAspare	Marca que indica a quantidade de tempo de execução que pode ser adicionada a uma tarefa de escalonamento sem afetar a execução das tarefas de menor prioridade.
~.SAslack	Indica a diferença entre a quantidade de trabalho restante e a quantia de tempo remanescente no período.
~.SAoverlaps	Indica o número de instâncias que podem ter o seu *deadline* ultrapassado.

«SAschedulable»	Estereótipo que caracteriza um objeto que constitui uma unidade de escalonamento.
«SAscheduler»	Estereótipo que representa o escalonador do sistema.
~.SAexecutionEngine	Marca que indica o conjunto de recursos do tipo processador gerenciados pelo escalonador.
~.SAschedulingPolicy	Marca que caracteriza o algoritmo de escalonamento utilizado.
«SAsituation»	Estereótipo que uma seqüência de interações, a qual é usada para fins de análise.
«SAtrigger»	Estereótipo que caracteriza um gatilho, i.e. o padrão de ativação de uma "resposta", executada por uma unidade de escalonamento.
~.SAschedulable	Marca que contém o resultado da análise, indicando se o gatilho pode ou não ser escalonado.
~.SAprecedents	Marca que indica o conjunto de ações que devem executar antes deste gatilho.
~.SAoccurrence	Marca que caracteriza o padrão de ocorrência do gatilho.
«SAusedHost»	Estereótipo que caracteriza uma dependência a qual explicita qual recurso escalonável uma ação necessita para executar.
«SAuses»	Estereótipo que caracteriza um tipo de dependência de uso que indicas os recursos compartilhados utilizados por uma ação.

2.4 Considerações Finais

Discutiu-se neste capítulo alternativas de modelagem de restrições presentes em aplicações tempo real em modelos orientados a objetos, sendo que um maior destaque foi dado ao perfil do UML-TR. Ressalta-se aqui o alto grau de generalização proposto pelo perfil UML-TR, cuja intenção é de não se concentrar em uma dada metodologia ou tecnologia. Assim, verifica-se que o maior objetivo do perfil não é voltado para o usuário final, mas sim para os desenvolvedores de ferramentas de apoio (CASE tools). Isto porque as ferramentas podem basear os seus diagramas e anotações em um *framework* padronizado, facilitando assim a troca de informações e modelos entre diversos desenvolvedores.

Também fica evidente o fato que o perfil UML-TR possui uma grande preocupação em permitir a modelagem dos mais variados mecanismos de suporte à programação de aplicações tempo real, os quais são geralmente encontrados em sistemas operacionais tempo real. Nota-se contudo que, na maioria dos casos, as alternativas de modelagem para alguns requisitos requerem modelos com um certo grau de complexidade. Identifica-se portando uma contradição: o modelo de alto nível torna-se tão complexo quanto os detalhes de programação. Contudo, vislumbra-se neste trabalho a especificação de restrições temporais, e não um detalhamento de como as mesmas deverem ser implementadas.

Verifica-se um problema semelhante em relação ao *framework* para análise de escalonabilidade. Neste, constata-se que muitos dos seus elementos estão diretamente voltados para uma análise de escalonabilidade, utilizada tipicamente antes da execução do sistema (denominada *off-line analysis*), cujo objetivo é de verificar se o sistema é ou não escalonável. Caso o projetista desejar se concentrar somente na especificação dos requisitos do sistema, percebe-se que muitos dos elementos definidos neste *framework* não seriam utilizados.

3 Tecnologias de Programação e Execução para Sistemas Tempo Real Orientados a Objetos

Neste capítulo retrata-se o estado da arte em relação aos mecanismos voltados para a programação de sistemas tempo real orientado a objetos. Estas propostas visam suprir as carências das arquiteturas convencionais para este tipo de desenvolvimento, ao passo que também discutem modelos de organização dos objetos da aplicação. Neste capítulo são descritas as propostas RT-CORBA (seção 3.1), RT-Java (seção 3.2), o modelo TMO (seção 3.3), Active Objects/C++ (seção 3.4) e a proposta *Quality Objects* – QuO (seção 3.5). Além de identificar as principais características das tecnologias apresentadas, neste capítulo também se procura determinar o grau de adequação das mesmas à programação de restrições temporais e de outros requisitos comumente encontrados em aplicações tempo real. Os pontos a serem verificados são listados abaixo:

1. Ações de ocorrência periódica;
2. Ações com controle de *deadline*;
3. Comunidação tipo síncrona e assíncrona;
4. Definição de *timeout* em chamadas síncronas;
5. Recursos de acesso protegido;

No final do capítulo faz-se um resumo das principais características das tecnologias apresentadas. Além disso, também é feito uma classificação de cada tecnologia estudada em relação ao suporte oferecido para a programação dos pontos verificados.

3.1 Real-Time CORBA

O Real-Time CORBA [OMG 99], ou simplesmente RT-CORBA, é uma extensão do CORBA [OMG 98] para sistemas tempo real a qual acrescenta mecanismos para prover controle de qualidade de serviço – QoS, priorizando a comunicação e a execução das mensagens e processos mais prioritários a fim de atender os requisitos temporais dos mesmos. Adicionalmente RT-CORBA oferece mecanismos que permitem um controle sobre as inversões de prioridade da aplicação e também o gerenciamento de requisitos do tipo fim-a-fim. As políticas de controle e gerenciamento sobre inversões de prioridade abrangem recursos de processamento, comunicação e memória. Para o controle dos recursos de processamento são oferecidos filas de *threads*, além de um novo modelo de prioridades e escalonamento. Já o controle sobre os recursos de comunicação é feito através de políticas de configuração dos protocolos e de conexões dedicadas. Por fim, o controle sobre os recursos de memória é feito através de configurações de armazenamento de mensagens. O restante desta seção aborda a maneira como estes mecanismos podem ser configurados.

O RT-CORBA adota uma política de pré-alocação de recursos para garantir o atendimento dos requisitos temporais da aplicação. Desta forma, é permitida a pré-alocação de filas de *threads* para gerenciar as *threads* que estão executando no lado servidor do *Object Resource Broker* (ORB), que é o componente principal do CORBA.

As características destas filas ajudam a reduzir inversões de prioridades, permitindo ao programador garantir recursos para satisfazer um certo número de invocações concorrentes. Além disso, as mesmas também ajudam a reduzir a latência e aumentar a previsibilidade, evitando a destruição e recriação de *threads* entre invocações. Outra vantagem é que as mesmas oferecem uma divisão de *threads* de forma automática, isolando partes do sistema de influências causadas por *threads* de outras aplicações. Novamente, esta característica também ajuda na redução de inversões de prioridades.

Esta arquitetura também disponibiliza um serviço de escalonamento global, baseado em prioridades fixas, que permite ao programador especificar os requisitos de escalonamento de modo intuitivo. Desta forma, uma camada de abstração esconde a coordenação dos parâmetros de escalonamento do RT-CORBA, como por exemplo, as prioridades CORBA e as políticas do adaptador de objetos tempo real (*Real-Time Portable Object Adapter* – RTPOA). O projetista faz uso do serviço de escalonamento atuando sobre objetos e atividades[6] nomeadas explicitamente pelo programador. O mesmo é responsável por determinar as características de ativação destas entidades, cuja repercussão será na ordem em que as mesmas serão escalonadas para execução (sempre visando o cumprimento dos requisitos temporais). A definição destas características é feita através de interfaces definidas pelo RTORB, que permitem a especificação de parâmetros como WCET, períodos, prioridade e importância.

A API do RT-CORBA oferece poucas facilidades para a programação de restrições temporais. Dentre as facilidades oferecidas, encontra-se a possibilidade de se definir operações com ativação periódica, conforme destacado no código da FIG. Outras restrições também importantes, como por exemplo a programação de operações com *deadline*, precisa ser programada com o uso dos mecanismos oferecidos pelo SO subjacente. Na FIGURA 3.2 exemplifica-se o código necessário para o controle de *deadline* em uma ação através da criação de *timers* no SO-TR QNX.

```
void Class2_i::anyMethod( CORBA::Environment &ACE_TRY_ENV ){
  ACE_Time_Value timevalue (0, 500);
  TimeBase::TimeT timet;
  ORBSVCS_Time::Time_Value_to_TimeT (timet, timevalue);
  ACE_ConsumerQOS_Factory consumer_qos_factory;
  consumer_qos_factory.start_disjunction_group();
  consumer_qos_factory.insert_type(this->_rt_info->handle+HANDLE_BASE,0);
  consumer_qos_factory.insert_time(ACE_ES_EVENT_INTERVAL_TIMEOUT,timet,
                                   this->_rt_info->handle);

  this->_proxy_supplier->connect_push_consumer(this->_consumer_ref.in(),
                        consumer_qos_factory.get_ConsumerQOS(),ACE_TRY_ENV);
}

void Class2::push(const RtecEventComm::EventSet& events,
                  CORBA::Environment& ACE_TRY_ENV){
    //Código "normal" da ação
}
```

FIGURA 3.1 – Programação de ação com ativação periódica no RT-CORBA

Por sua vez, o problema de mapeamento de prioridades é típico de aplicações que executam em ambientes heterogêneos. Isto porque, por exemplo, é difícil definir qual a prioridade em que irá executar uma chamada de operação no sistema operacional B,

[6] Atividade pode ser definida como um conjunto de ações agrupadas, i.e. delimitadas explicitamente por um início e um fim.

feito por um objeto rodando no sistema operacional A. Para solucionar este problema, o RT-CORBA define um mecanismo que permite às aplicações utilizarem prioridades de maneira consistente. Assim, dois modelos de prioridades distintos podem ser selecionados e configurados através de uma interface especial. O primeiro destes modelos, denominado "propagado pelo cliente", consiste na propagação da prioridade da atividade do cliente para o servidor. Como requisito, o lado servidor do ORB deve executar as chamadas subseqüentes nesta prioridade (sujeitando-se a qualquer protocolo de herança de prioridade). No segundo modelo proposto a prioridade de um método é determinada pelo servidor onde o método é executado sendo portanto conhecida como "declarada pelo servidor".

```
void controledAction(){
  timer_t timer_id1;
  struct  itimerspec my_timer;

  //Funcao para tratar a interrupcao do Timer
  signal(SIGALRM, trataTimeout);

  //Criando o timer
  timer_id1 = timer_create(CLOCK_REALTIME, NULL);
  if (timer_id1 == -1) {
    cerr << "Timer nao pode ser criado.\n";
    exit(1);
  }

  //Setando valores e inicializando timer
  long msec = long((100)*1000000);
  my_timer.it_value.tv_sec     = 0;
  my_timer.it_value.tv_nsec    = msec;
  timer_settime(timer_id1,0, &my_timer,NULL);

  //Código "normal" da ação

  timer_delete(timer_id1);
}

void trataTimeout(int sig_number){
  //Código de exceção....
}
```

FIGURA 3.2 – Controle de deadline no RT-CORBA sobre o SO-TR QNX

Para suportar a especificação de QoS sobre os recursos de comunicação, são oferecidas interfaces que permitem a seleção e configuração de protocolos nos lados do servidor e do cliente do ORB. Apesar de ser unanimidade que o TCP não oferece garantias de tempo de resposta determinístico, ironicamente o RT-CORBA define um conjunto de propriedades somente para este protocolo. Tais propriedades permitem configurar o tamanho das mensagens e dos *buffers* de armazenamento, e indicações de roteamento, *delay* e tipo de conexão. Todavia, somente a configuração do protocolo de comunicação não é suficiente para garantir o atendimento aos requisitos de QoS. Para tanto, são adotadas políticas para oferecer controle sobre os recursos de comunicação, usando mecanismos explícitos de conexão. Assim, o RT-CORBA oferece facilidades para um cliente se comunicar com um servidor via múltiplas conexões, com cada uma delas tratando invocações feitas em prioridades, ou faixa de prioridades distintas. A seleção da conexão apropriada é transparente para a aplicação, que normalmente usa uma simples referência para objeto. Também é permitido a um cliente obter uma conexão de transporte privada com um servidor, que não será multiplexada (compartilhada) com outras conexões de objetos do tipo cliente-servidor.

A comunicação entre objetos no RT-CORBA pode ser feita de maneira síncrona ou assíncrona. A segunda é realizada através das invocações assíncronas de métodos (*asynchronous method invocation* – AMI), conforme ilustrado no código da FIG. Ainda neste contexto, aplicações RT-CORBA podem estabelecer um *timeout* associado a uma invocação, de modo a limitar o tempo que o cliente da aplicação permanece bloqueado esperando pela resposta. O código apresentado na FIGURA 3.4 exemplifica a programação deste requisito. Esta propriedade é importante para aumentar a previsibilidade do sistema.

```
void Class1_i::anyMethod( CORBA::Environment &ACE_TRY_ENV ){
  ACE_TRY {
    this->_Instance->operation2(ACE_TRY_ENV); //1: Chamada assíncrona
  }
  ACE_CATCH ( CORBA::Exception,exc ) {
    //Código de exceção....
  }
  ACE_ENDTRY;
}

void Class2_i::operation2( CORBA::Environment &ACE_TRY_ENV ){
  ACE_TRY {
    int value = this->_Instance->operation3(ACE_TRY_ENV); //2: Chamada síncr
  }
  ACE_CATCH ( CORBA::Exception,exc ) {
    //Código de exceção....
  }
  ACE_ENDTRY;
}

int Class3_i::operation3( CORBA::Environment &ACE_TRY_ENV )
                          ACE_THROW_SPEC (( CORBA::SystemException )){
  ACE_TRY
  {
    return this->_value;
  }
  ACE_CATCH (CORBA::Exception,exc)
  {
    //Código de exceção....
  }
  ACE_ENDTRY;
}
```

FIGURA 3.3 – Programação de chamadas síncronas e assíncronas no RT-CORBA

```
// 50 milisegundos (unidades básicas são 100 nanosegs)
CORBA::Any val; val <<= TimeBase::TimeT (500000UL);

// Criação da política de timeout
CORBA::PolicyList policies (1); policies.length (1);
policies[0] = orb->create_policy
   (Messaging::RELATIVE_RT_TIMEOUT_POLICY_TYPE, val);

// Sobrescreve a política de acesso da Class2
CORBA::Object_var obj = class2->_set_policy_overrides
   (policies, CORBA::ADD_OVERRIDE);

Class2_var class2_with_timeout = Class2::_narrow (obj);
try {  class2_with_timeout->operation1(); }
catch (const CORBA::TIMEOUT &e) { // Código de exceção }
```

FIGURA 3.4 – Programação de *timeout* na invocação de uma operação síncrona no RT-CORBA

Adicionalmente, o RT-CORBA também oferece uma maneira segura de manter a sincronização entre os objetos da aplicação. Isto porque, no CORBA tradicional, os mecanismos de exclusão mútua oferecidos estão sujeitos a problemas de inversão de prioridades. O RT-CORBA corrige este problema através da interface *Mutex*, a qual garante que a consistência do mecanismo de exclusão mútua seja mantida além dos limites da aplicação e do próprio ORB. O uso da interface *Mutex* é exemplificado na FIGURA 3.5.

```
void Data_i::getData() ( CORBA::Environment &ACE_TRY_ENV )
  ACE_THROW_SPEC (( CORBA::SystemException ))
  {
    CORBA::Boolean result;
    this->_mutex_arrival = this->_rt_orb->create_mutex(ACE_TRY_ENV);
    result = this->_mutex_arrival->try_lock(tm_mutex,ACE_TRY_ENV);
    if(result){
      _data = this->data;
      this->_mutex_arrival->unlock(ACE_TRY_ENV);
      return this->_data;
    }
}
```

FIGURA 3.5 – Programação de área com exclusão mútua no RT-CORBA

3.2 RT-Java

A grande popularidade da linguagem Java motivou a criação de extensões tempo real para a mesma. Estas extensões foram desenvolvidas por dois grupos distintos, o *Java Experts Group* (JEG) e o *J Consortium* (JC). Apesar de discordarem em alguns aspectos, ambas as especificações seguem as recomendações do documento fruto de um projeto financiado pelo NIST (*U.S. National Institute of Standards and Technology*) [NIS 99]. Este trabalho concentra-se na descrição da proposta do JEG, uma vez que se considera esta proposta mais sólida e acessível. Além disso, sistemas comerciais para RT-Java segundo a especificação do JEG já se encontram disponíveis, tais como o processador da empresa *aJile* e as placas de desenvolvimento *JStamp* da empresa *Systronix* ou ainda o sistema J9 da empresa IBM, este último uma combinação do ambiente *Visual Age Micro Edition* rodando sobre o sistema operacional *Neutrino* da empresa QNX.

A especificação do JEG [BOL 2001], denominada *The Real-Time Specification for Java* (RTSJ), define uma API para a linguagem Java que permite a criação, verificação, análise, execução e gerenciamento de *threads* tempo real, procurando satisfazer seus requisitos temporais. Ao contrário da proposta do JC, não são feitas alterações sintáticas na linguagem Java. Os princípios de programações definidos constituem-se de requisitos de alto nível, que delimitam o escopo do trabalho feito pelo JEG e introduz princípios de compatibilidade entre a especificação do RT-Java e aplicações Java tradicionais. Tais aplicações não devem sofrer restrições de execução em implementações do RTSJ. Este último não deve incluir especificações que restrinjam o uso do RT-Java à ambientes particulares. Estes princípios encontram-se listados a seguir:

Esta especificação introduz o conceito de "objeto escalonável", constituído por qualquer instância de classe que implemente a interface *Schedulable*. São especificadas três classes contendo objetos escalonáveis: *RealtimeThread*, *NoHeapRealtimeThread* e *AsyncEventHandler*. Também é especificado um conjunto de classes para armazenar parâmetros que representam uma demanda particular por recursos de um ou mais objetos escalonáveis. A classe *SchedulingParameters* e suas classes derivadas *PriorityParameters* e *ImportanceParameters* fornecem informações úteis para o escalonador do sistema. Já a classe *ReleaseParameters*, superclasse de *AperiodicParameters*, *PeriodicParameters e Sporadic-Parameters*, contém uma série de parâmetros úteis na especificação das características de ativação do objeto escalonável. Na FIGURA 3.6 exemplifica-se criação de um objeto escalonável com ativação periódica, com período e deadline de 500ms. Este código também inclui a definição de uma operação de tratamento de exceção associada ao objeto m_asyncDeadline. Esta operação é acionada em caso do objeto escalonável não terminar a execução do seu código antes do *deadline* especificado.

```
private AsyncEventHandler m_asyncDeadline =
  new AsyncEventHandler() {
      public void handleAsyncEvent() {
        //Código de exceção da ação
      }
  };

public Class2(){
  super(new PriorityParameters(Thread.NORM_PRIORITY),
        new PeriodicParameters(null, null, new RelativeTime(500, 0),
                               new RelativeTime(500, 0), null, null) );
        getReleaseParameters().setDeadlineMissHandler( m_asyncDeadline );
}

public void run(){
    while( m_isRunning ) {
        operation1();
        waitForNextPeriod();
    }
}

public void operation1(){
      //Código "normal" da ação
}
```

FIGURA 3.6 – Programação de operação periódica no RT-Java

Dado que o gerenciamento de memória é reconhecido como uma propriedade importante do ambiente de programação Java, a proposta pretende manter este gerenciamento de maneira automática no ambiente de execução e não passá-lo para uma tarefa de programação. Para que este objetivo seja alcançado é necessário que os programas de gerenciamento automático de memória, conhecidos como *garbage collector* (GC) sejam modificados, a fim de permitir uma caracterização precisa dos efeitos da execução do GC no tempo de execução, preempção e despacho de *threads* tempo real. De modo alternativo, também deve ser permitida a alocação e a liberação de objetos fora de qualquer espaço de memória gerenciado pelo GC.

Outro conceito proposto por esta especificação é a definição da superclasse *MonitorControl*, que oferece controle de elegibilidade para recursos compartilhados, como o processador ou outros blocos sincronizados. Para manter o controle sobre

inversões de prioridade, foi decidido preservar a palavra-chave *synchronized*. Desta forma, deverá ser implementado um protocolo de herança de prioridades (classe *PriorityInheritance*). A especificação também oferece um mecanismo para permitir ao programador alterar a política adotada no sistema, ou em algum monitor particular, desde que seja oferecida uma implementação para a mesma. Além disso, também é especificada uma política do tipo *priority ceiling* (classe *PriorityCeilingEmulation*). Porém, esta restrição não é suficiente para garantir que as *threads* não sofram inversão de prioridade e que elas não terão prioridade de execução menor que a do GC. A FIGURA 3.7 exibe a associação de um monitor à classe Data, juntamente com a especificação de um método chamado *writeData()* com exclusão mútua de acesso.

```
public Data() {
  MonitorControl.setMonitorControl( this, PriorityInheritance.instance() );

  public syncronized void writeData(Data data) {
    this.data = data;
  }
}
```

FIGURA 3.7 – Programação de objeto com área de exclusão mútua no RT-Java

Considerando-se que sistemas tempo real típicos possuem elevada interação com o mundo físico, e que este se comporta de maneira assíncrona, foram inclusos mecanismos de programação para tratar a ocorrência de eventos assíncronos. Este recurso engloba duas classes: *AsyncEvent* e *AsyncEventHandler*. Um objeto *AsyncEvent* representa um acontecimento assíncrono, como por exemplo um sinal POSIX, uma interrupção de *hardware*, ou até mesmo um evento do próprio sistema. A ocorrência destes eventos é indicada pela chamada do método *fire()*. Este dispara o métodos *handleAsyncEvent()* correspondente, presente em instâncias da classe *AsyncEventHandler*. Os atendimentos aos eventos são escalonados de acordo com a política adotada pelo sistema. A FIGURA 3.8 exibe o uso destas classes para implementar uma chamada assíncrona de método, i.e. objetos da classe *Class1* não ficam bloqueados enquanto o código do método *operation2()* é processado.

```
public Class1 extends RealtimeThread{
  public anyMethod(){
    Instance.operation2(); //1: Chamada assíncrona
  } };

public Class2 extends RealtimeThread{
  private Operation2EventHandler m_operation2 = new Operation2EventHandler() {
    public void handleOperation2Event() {
      int valor = Instance.operation3(); //2: Chamada síncrona
    }
  };

public void operation2()
{
    m_operation2.fire();
} };

public Class3 extends RealtimeThread{
  public operation3(){
    return this.valor;
  } };
```

FIGURA 3.8 – Programação de chamada assíncrona de método no RT-Java

Uma rotina de tratamento de eventos assíncronos (*AsyncEventHandler)* possui associados parâmetros de ativação (*ReleaseParameters*), escalonamento (*SchedulingParameters*) e memória (*MemoryParameters*), os quais servem para controlar a execução da rotina quando o evento assíncrono (*AsyncEvent*) associado for disparado. Está previsto que o ambiente de execução possa tratar situações em que houver um elevado número de eventos assíncronos e suas respectivas rotinas de atendimento. Espera-se porém que o número de eventos disparados (durante execução) seja menor do que os limites definidos para o sistema.

Uma forma especializada de um *AsyncEvent* é a classe temporizador (*timer*), que representa um evento cuja ocorrência é determinada por um relógio tempo real. São especificados dois tipos de temporizadores: *OneShotTimer* e *PeriodicTimer*. Instâncias do primeiro são disparadas somente uma vez, em algum ponto determinado no tempo. Os temporizadores periódicos disparam em um momento específico e voltam a disparar ciclicamente de acordo com o período especificado. Os temporizadores são controlados por objetos da classe *Clock*, que pode representar um relógio tempo real. Esta também pode ser estendida para representar qualquer outro tipo de relógio que for disponibilizado pelo sistema subjacente. O código exibido na FIGURA 3.9 exemplifica a definição de um *timer*.

```
private AsyncEventHandler m_asyncRadarTimeout = new AsyncEventHandler() {
  public void handleAsyncEvent(){
    //Código do timeout
  }
};

void anyOperation(){
  RelativeTime rtimePeriod = new RelativeTime( 1000, 0 );
  m_ptimer = new PeriodicTimer( null, rtimePeriod, m_asyncRadarTimeout );
  m_ptimer.start();
}
```

FIGURA 3.9 - Exemplo de programação de *timer* no RT-Java

Outra propriedade importante desta especificação é a inclusão de um mecanismo para estender o tratador de exceções Java, permitindo que as aplicações mudem programaticamente o ponto de controle para outra *thread*. Esta técnica é denominada Transferência Assíncrona de Controle (*Asynchronous Transfer of Control* - ATC). Os elementos principais do ATC encontram-se agregados na classe *AsynchronouslyInterruptedException* (AIE), na sua subclasse *Timed*, na interface *Interruptible* e na semântica dos métodos interrompidos de *Thread* ou *RealtimeThread*. Um método indica que está pressuposto a ser interrompido referenciando a AIE na sua cláusula *throws*. Diversas soluções para tratamento do AIE estão disponíveis. Uma delas oferece ao programador a opção de utilizar cláusulas *catch* e um mecanismo de baixo nível que especifica controle sobre propagação. Em outra pode ser usado um recurso de alto nível que permite a especificação de códigos para geração e tratamento de interrupção em métodos separados, conforme o código mostrado na FIGURA 3.10. Este trecho de programa serve para limitar o tempo de execução de uma ação. Com características semelhantes ao ATC, também é oferecido um mecanismo chamado

término assíncrono de *threads*, sendo uma maneira segura para finalizar processos concorrentes.

```
private void controledAction() throws AsynchronouslyInterruptedException {
  RelativeTime rtimeRelativeDeadline = new RelativeTime( 100MS, 0 );

  Interruptible interruptible = new Interruptible() {
    public void run( AsynchronouslyInterruptedException aie )
        throws AsynchronouslyInterruptedException {
          //Código "normal" da ação
    }

    public void interruptAction( AsynchronouslyInterruptedException aie ) {
        //Código de exceção....
    }
  };

  new Timed( rtimeRelativeDeadline ).doInterruptible( interruptible );
}
```

FIGURA 3.10 – Uso da classe *Timed* para limitar o tempo de execução de uma ação

Por fim, apesar deste não ser necessariamente um requisito tempo real, também foi definido uma classe que permite aos programadores acesso direto à memória física. A classe *RawMemoryAccess* define métodos que permitem ao programador construir um objeto que representa uma faixa de endereços da memória física e acessar a memória com uma granularidade de vários tipos de dados. Esta classe permite aos programadores de sistemas tempo real implementar rotinas de acesso aos dispositivos (*device drivers*), entradas e saídas mapeadas em memória (incluindo memórias *flash* e RAM alimentada por baterias) e diversas rotinas equivalentes de software baixo nível.

3.3 TMO

O modelo de objetos TMO (*Time-triggered Message-triggered Objects*) [KIM 99] foi desenvolvido para auxiliar a construção dos sistemas onde os aspectos temporais são de vital importância. A principal característica na qual se baseia este modelo é a facilidade para projetar aspectos temporais, garantindo aos objetos a capacidade de oferecer serviços com ativações dependentes de restrições temporais. O desenvolvimento do modelo de objetos TMO adota como metas principais o princípio da estruturação uniforme dos programas, tanto para sistemas tempo real distribuídos quanto para suas aplicações em ambientes de simulação, e também a garantia de oferecer aos objetos serviços com tempo de duração determinado durante o projeto. A FIGURA 3.11 apresenta a estrutura básica de um objeto TMO.

Quatro motivos essenciais tornam TMO uma extensão do modelo básico de objetos. Primeiramente, para alguns métodos de um objeto TMO, um relógio tempo real oferece um mecanismo de sincronização e gatilho para a execução destes métodos, cujos valores são especificados em nível de projeto. Estas constantes de tempo aparecem no primeiro parágrafo da especificação de um SpM, sendo conhecidas por condições de ativação autônoma (*autonomous activation condition -AAC*). Esta condição é exemplificada na FIGURA 3.12 para caracterizar um SpM que ocorre das 10hs as 10hs e 50min, com ativação a cada 30 min (t_1 = 10hs e t_2 = 10hs e 30min) e início entre t_i e t_i + 5 min e final até t_i + 10 min.

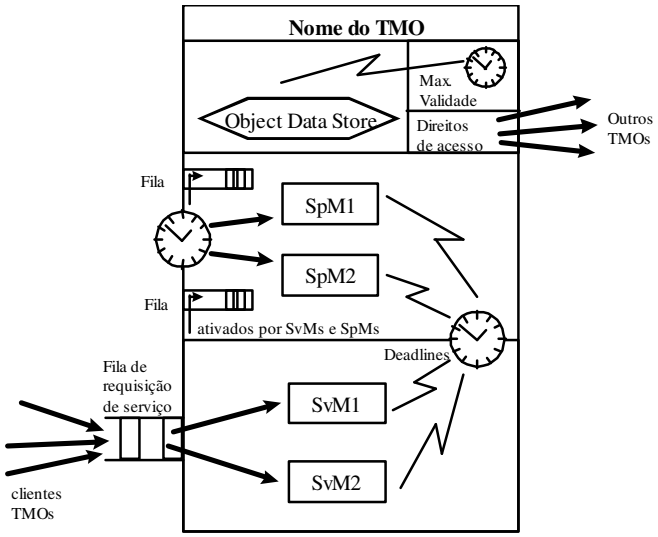


FIGURA 3.11 - Estrutura de um objeto TMO

```
"for t = from 10am to 10:50am
     every 30min
 start-during (t, t+5min) finish-by t+10min", i.e.:

{"start-during (10am, 10:05am)
  finish-by 10:10am",
 "start-during (10:30am, 10:35am)
  finish-by 10:40am"}.
```

FIGURA 3.12 - Exemplo de condição AAC

Os métodos de um objeto TMO são divididos em duas categorias. Os primeiros são chamados de espontâneos (*spontaneous methods - SpM*), sendo utilizados para representar operações periódicas. Os SvMs são claramente diferenciados dos métodos convencionais de serviço (*service methods - SvM*), os quais são ativados através das mensagens esporádicas ou aperiódicas, provenientes dos objetos clientes. Ações "tempo-dependentes" só podem ser executadas pelos SpMs. Outro motivo do TMO ser considerado uma extensão do modelo básico de objetos é imposição de um limite temporal de execução (*deadline*) a todos os métodos dos objetos TMO. A FIGURA 3.13 exemplifica a criação de um SpM, incluindo a especificação da sua AAC e também de um *deadline* no valor de 30 ms. Outro detalhe é que os dados contidos em um objeto TMO possuem um prazo máximo de validade (*maximum validity duration -MVD*) sendo que todo valor atribuído a um dado torna-se inválido após expirar o prazo fixado.

O modelo de objetos TMO também possui um inibidor de concorrência (*basic concurrency constraint - BCC*), que previne conflitos entre os SpMs e os SvMs ao acessarem dados compartilhados. Basicamente, a execução de um serviço ativado por um cliente somente é permitida quando o potencial SpM conflitante não estiver sendo executado. Esta informação é determinada durante a programação do método, conforme exibido no trecho que define os direitos de acesso à região de dados (*Object Data Space - ODS*) na FIGURA 3.13. Para ser exato, quando um método de serviço (SvM) acessar o ODS de um método espontâneo (SpM), a execução do primeiro não deve ser permitida no intervalo de tempo predestinado à execução do segundo. Portanto, os SpMs recebem maior prioridade de execução do que os SvMs. A programação do ODS deve conter

métodos que delimitam o acesso aos dados compartilhados, conforme exibido na FIGURA 3.14.

```
operation1SpM::operation1SpM(ODSClass2 *odss){
       odsT = odss;
       TMOSLstrcpy(SpMParm.TMO_Name, "Class2");
       TMOSLstrcpy(SpMParm.SpM_Name, "operation1SpM");

       // Define direitos de acesso à região de dados (ODS)
       SpMParm.NumUsedODSS = 0;
       SpMParm.ODSAccessRight[0].ODSS_ID = odss->ODSSID;
       SpMParm.ODSAccessRight[0].AccessMode = READ_WRITE;

       // Define expressão AAC
       SpMParm.NumUsedAACExp = 1;
       SpMParm.AACExp[0].AACExpID = 3;
       SpMParm.AACExp[0].Type = AUTOMATIC;

       SpMParm.AACExp[0].EST.Sec = 10;
       SpMParm.AACExp[0].StopTime.Hour = 2400;
       ::ZeroMemory( &SpMParm.AACExp[0].StopTime, sizeof ( TimeOfDay ) );

       SpMParm.AACExp[0].StopTime.Sec = 20;
       SpMParm.AACExp[0].Interval.Msec = 2000;
       SpMParm.AACExp[0].Deadline.Msec = 30;

       SpMParm.AACExp[0].LST.Sec = SpMParm.AACExp[0].EST.Sec;
       SpMParm.AACExp[0].LST.Msec = SpMParm.AACExp[0].EST.Msec + 15;

       if (RegisterSpM() == FAIL)
       {
              TMOSLprintf("Fail to register ControlarSpM\n");
       }
}

void operation1SpM::SpMBody(){
     //Código "normal" da ação
}
```

FIGURA 3.13 – Programação de operação periódica no TMO

```
class ODSData:public ODSSBaseClass{
  Data data;
public:
  ODSData(){};

  void getData(Data *data){
    EnterODSS_RO();
    *data = this->data;
    ExitODSS_RO();
  }

  void Setestado(Data data){
    EnterODSS_RW();
    this->data = data;
    ExitODSS_RW();
  }
};
```

FIGURA 3.14 – Código que define uma estrutura de dados com acesso compartilhado

No que diz respeito à interação entre os objetos TMO, existem basicamente dois modos de comunicação: chamadas bloqueantes e chamadas não-bloqueantes para um SvM. O uso destas chamadas é mostrado na FIGURA 3.15. Outra característica do TMO é permitir a definição do tempo máximo de espera durante chamadas síncronas. O TMO também adota uma variação importante destas chamadas, com o objetivo de reduzir a sobrecarga de comunicação em algumas situações. A essência desta extensão é permitir um certo arranjo, onde um cliente faz uma chamada a um SvM e depois recebe a resposta de outro SvM. Tal extensão consiste de passar o cliente de um SvM para um outro SvM, envolvendo mensagens intra-objetos, sendo denominada *client-transfer call*. As mensagem intra-objetos são bem mais simples que mensagens interobjetos. O mesmo sistema pode ser usado entre objetos diferentes, ou seja, um SvM pode transferir a requisição de um serviço para um outro SvM e outro objeto TMO. Neste caso serão usadas mensagens interobjetos.

```
void anyMethodSpM::SpMBody(){
  //1: Chamada assíncrona
  NonBlockingSR("Class2","operation2", &SpMreturnParam,
sizeof(SpMreturnParam), &timestamp);

void operation2SpM::SvMBody(){
  ReceiveSR(&Client_RRQID, &SvMinputParam, sizeof(SvMinputParam),
&timestamp);

  //2: Chamada síncrona
  BlockingSR("Class3","operation3", &SpMreturnParam, sizeof(SpMreturnParam),
500 /*timeout*/);

  if ( SpMreturnParam == ERROR ) {
    //Código de exceção
  }
}

void operation3SpM::SpMBody(){
  ReceiveSR(&Client_RRQID, &SvMinputParam, sizeof(SvMinputParam),
&timestamp);
}
```

FIGURA 3.15 – Modelos de comunicação entre objetos TMO

Resumindo as diferenças desta abordagem, verifica-se que uma das principais características do modelo de objetos TMO é a clara distinção entre os métodos espontâneos e os métodos de serviço. Em nenhuma outra abordagem é feita esta distinção entre os métodos que são ativados quando alguma constante de tempo atingir um determinado valor e os que são ativados por mensagens comuns. A vantagem é que um SvM no modelo TMO não é iniciado se existir a possibilidade de ocorrer conflito com outro SpM que estiver em execução. Adicionalmente, o ambiente de execução dos objetos TMO faz uma análise tanto do melhor quanto do pior caso para executar um determinado serviço. Isto garante que seja verificado a possibilidade, ou não, do serviço ser realizado no tempo projetado. O TMO é um dos poucos modelos a adotar os *client-transfer calls*, que permitem interações com troca de mensagens intra-objetos (entre métodos do mesmo objeto) e também interobjetos (entre métodos de objetos diferentes). Na sua versão mais recente, o modelo TMO está implementado sobre o CORBA.

3.4 Active Objects/C++

O modelo *Active Objects*/C++ (AO/C++) [PER 94] foi idealizado durante o desenvolvimento de alguns estudos de caso, cujos resultados mostraram que a união dos conceitos de orientação a objetos com as características de sistemas operacionais tempo real conduz à produção de sistemas mais compactos e legíveis, e de manutenção facilitada. A idéia principal do AO/C++ é mapear o modelo logicamente distribuído das linguagens orientadas a objetos, caso de C++, com o modelo fisicamente distribuído de um sistema operacional UNIX tempo real (UNIX-TR) orientado a processos. Para tanto é utilizado o conceito de objetos ativos, isto é, que possuem a capacidade de operar concorrentemente, assumindo assim que cada objeto será mapeado para um processo UNIX-TR.

A implementação de um aplicativo em AO/C++ se dá através da codificação com C++ acrescido por algumas palavras chave, como *active* por exemplo, a qual indica que a classe declarada é ativa. Posteriormente, um pré-processador converte o código AO/C++ em um código C++ usual, acrescido de chamadas ao sistema operacional ou *middleware* subjacente. Para fins de estruturação, as classes ativas devem ser codificadas em dois arquivos separados (exclusivos para cada classe): o cabeçalho da classe (*header*) é declarado em um arquivo de extensão *.ph*, enquanto a definição dos seus métodos é feita em um arquivo *.pC*. A definição de uma classe ativa possui um significado semelhante à definição de uma interface no CORBA, enquanto que o pré-processador tem papel semelhante ao do compilador IDL.

Todas as classes ativas no AO/C++ herdam as propriedades de uma classe básica, a qual possui as seguintes características:

- possui uma referência para um processo UNIX-TR, onde a computação (métodos da classe) é executada;
- é responsável por mapear as chamadas de funções feitas pelas classes C++ para mensagens.

Um construtor especial é definido para as classes ativas. Ele gera, em tempo de execução, um processo UNIX-TR bem como uma instância C++ passiva, isto é, uma instância de classe usual do C++. Enquanto o primeiro executa as instruções da instância, a segunda é responsável por mapear as chamadas de funções C++ para mensagens destinadas ao processo relacionado e também por empacotar/desempacotar os argumentos passados e os resultados retornados (possuem, respectivamente, o mesmo papel do *Stub* e do *Skeleton* no CORBA). Este construtor é uma característica particular do modelo, que permite combinar o poder dos compiladores C++ existentes com o modelo concorrente de características tempo real dos processos UNIX-RT.

Conforme já mencionado, instâncias de classes ativas podem executar de forma concorrente e/ou distribuída com outras instâncias ativas. Assim, faz-se necessário definir alguns parâmetros individuais de cada objeto ativo que serão utilizados pelo ambiente de execução. Estes parâmetros dizem respeito à prioridade de execução do objeto e ao nó da rede em que o mesmo será alocado. A FIGURA 3.18 contém o código responsável a operação de criação de um objeto ativo denominado obj1. Observa-se que a instância sendo criada recebe dois parâmetros, que são passados para o construtor da classe, indicando respectivamente a prioridade do objeto e o nó da rede em que ele vai executar.

```
void main(void){
   Class1 obj1(8, 2);  //(prioridade, nó_execução)
   char *nome = "/celula";
   obj.create(nome);  //ativa a aplicação
   cout << "Objeto " << nome << " criado!" << endl;
   ...
};
```

FIGURA 3.16 – Criação de objetos no AO/++

Em um objeto ativo, as instruções e funções definidas por *public* são consideradas a sua interface. Cabe mencionar que as funções de interface destes objetos ativos, apesar de manterem a sintaxe original do C++, podem ser chamadas a partir de outros processos ou mesmo de outros processadores, funcionando como uma chamada remota de método (RMI). O pré-processador do AO/C++ torna estas chamadas remotas transparentes, modificando o corpo das funções de interface. Devido a esta característica, quando um objeto ativo chama um método de outro objeto ativo, a chamada é convertida para uma mensagem do tipo *send()*. No receptor, a mensagem é então decodificada e a função correspondente é ativada.

Para este mapeamento automático da interface é decisivo se a operação na classe ativa espera um valor de retorno definido ou não. Se a função não retornar nenhum valor, a troca de mensagens é mapeada de forma assíncrona. Se houver tipo definido, a comunicação é mapeada de forma síncrona. No caso de comunicação síncrona, o objeto cliente fica bloqueado até que o objeto servidor processe a mensagem e retorne os dados resultantes. O código mostrado na FIGURA 3.17 exibe a definição de uma operação síncrona e outra assíncrona.

```
active class Class1{
  void anyMethod(){
    Instance->operation2();  //1: Chamada assíncrona
  }
}

active class Class2{
  void operation2(){
    int valor = Instance->operation3();  //2: Chamada síncrona
  }
}

active class Class3{
  int operation3(){
    return (this->valor);
  }
}
```

FIGURA 3.17 – Modelos de comunicação entre objetos AO/++

Construtores especiais, semelhantes aos utilizados em linguagens tempo real de alto nível (como PEARL [WER 89]), também foram adicionados ao AO/C++ para dar mais poder ao modelo. Estes construtores permitem a definição de processos com ativação periódica, garantindo a capacidade de definir métodos de objetos com ativação cíclica e também definir *deadlines* associados à execução dos métodos (tanto no lado do cliente como no lado do servidor). Também é oferecido suporte para a definição do código de tratamento de exceção a ser executado em caso de violação de deadline. A FIGURA

3.18 exibe o código que representa a definição de uma classe ativa em AO/C++, juntamente com os construtores para definição de restrições temporais.

```
active class Sensor {
private:
       // reference to other active objects
       Pump REF thePump;
       DrvSensor REF theDrvSensor;

public:
       void Read(cycle_t){ //Método cíclico
       //Código de Inicialização....
       begin_cycle
              //Código cíclico....
       end_cycle
       }
       void Read(dead_l){ Método com deadline e timeout
       begin_operation
              //Código de Operação Normal....
       end_ operation
       begin_exception
              //Código de Exceção....
       end_ exception
       }
} // end of sensor class
```

FIGURA 3.18 - Definição de uma classe ativa em AO/C++ com restrições temporais

3.5 A Proposta *Quality Objects*

Conforme relatado ao longo do capítulo, problemas de alocação e controle de recursos têm sido abordados através da adoção de *middlewares* para computação com objetos distribuídos. Estes, porém, não oferecem alternativas para tratar com a questão da adaptabilidade das aplicações. Aplicações adaptativas devem suportar operações cujo comportamento varia de acordo com a quantidade de recursos disponíveis, estes normalmente avaliados segundo as condições de qualidade de serviço. Contudo, a definição de requisitos de QoS por parte da aplicação é feita através de interfaces complicadas e que tem se modificado constantemente.

Motivados por estas dificuldades, foi proposta uma nova camada de abstração para tratar com a adaptabilidade das aplicações tempo real, chamada *Quality Objects* (QuO)[7] [ROD 2000]. Aplicações QuO possuem capacidade de especificar, controlar, monitorar e se adaptar às condições de qualidades de serviço do sistema. Para tanto, é necessário especificar regiões de operação, implementações alternativas e estratégias de adaptação. Por exemplo, uma região de operação indica a faixa de valores de um determinado parâmetro, como um *deadline*. As implementações alternativas indicam trechos de código distintos para as faixas de valores especificadas. Por sua vez, as estratégias de adaptação são semelhantes a um código de exceção, indicando uma ação para tentar corrigir situações indesejáveis.

Usando-se a chamada "Linguagem de Descrição de Qualidade" (*Quality Description Language - QDL*), especificações de qualidades de serviço desejadas na comunicação entre objetos clientes e servidores podem ser descritas, dando origem aos

[7] http://www.dist-systems.bbn.com/tech/QuO

chamados "contratos" (*contracts*). Durante a execução as condições de QoS são monitoradas pelos "objetos de condições do sistema" (*system conditions objects*) os quais permitem uma modificação dinâmica dos contratos nos casos de degradação dos tempos de transmissão e recepção de mensagens. Os contratos, por sua vez, avisam os programas clientes, usuários, gerentes e outros objetos de condições sobre as mudanças ocorridas, através de transições de comportamento. Neste ponto são afetadas múltiplas camadas de adaptação, tais como dos gerentes e mecanismos[8], dos contratos e finalmente dos clientes e usuários. A camada responsável pelo controle da QoS no sistema é a dos gerentes e mecanismos, que oferecem serviços dependentes do ORB, tal como comunicação controlada, replicação ou segurança. Os contratos e os delegados é que interagem com estes serviços através dos objetos de condições do sistema. A FIGURA 3.19 ilustra os elementos do QuO.

O projetista dos QuO é quem define contratos, objetos de condições do sistema e delegados da aplicação. Os contratos resumem os possíveis estados de QoS no sistema e o comportamento nas transições executadas quando as condições de QoS se alterarem. Regiões representam diferentes períodos no tempo nos quais as informações de QoS se tornam disponíveis. Como exemplo cita-se as regiões negociadas (*negotiated regions*), que representam o nível de serviços que o cliente espera receber e que o servidor consegue fornecer, e as regiões reais (*reality regions*), que representam os níveis de serviço observados. As regiões são definidas por meio de predicados sobre os objetos de condições do sistema. O comportamento a ser disparado quando for alterada a região ativa é definido através de transições.

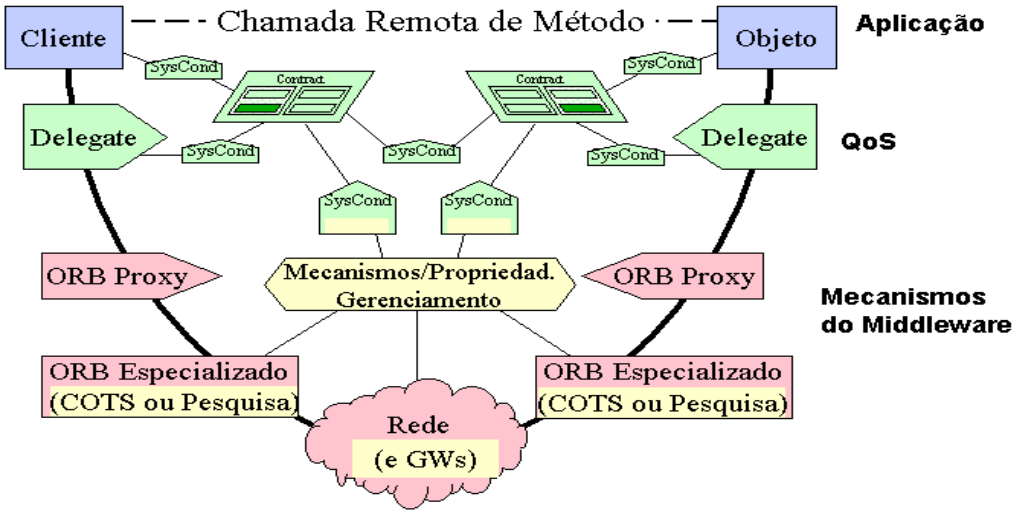


FIGURA 3.19 - Elementos da proposta QuO

Objetos de condições do sistema são usados para medir e controlar QoS. Para tanto, eles oferecem interfaces para mensurar os recursos do sistema, as expectativas dos clientes e servidores, os mecanismos, os gerentes e as funções especializadas do ORB. Mudanças nos objetos de condições do sistema observadas pelos contratos podem causar transição de regiões. Também são oferecidos métodos para prover acesso ao controle de QoS feito pelos recursos, mecanismos, gerentes e ORBs.

[8] Servem para indicar as propriedades de gerenciamento.

Os delegados definem o estado local para objetos remotos. Assim, quando for executado uma chamada ou um retorno de método, os delegados checam o estado atual do contrato e escolhem o comportamento com base do estado corrente dos aspectos de QoS. Por exemplo, um delegado pode escolher entre métodos alternativos, conexões (*binding*) alternativas entre objetos remotos, executar processamento de dados local ou simplesmente repassar a chamada ou retorno de método. Este mecanismo adiciona controle e medição de parâmetros de QoS na chamada remota.

Conforme citado anteriormente, outro elemento importante do *framework* QuO é a QDL. Esta linguagem é análoga a IDL do CORBA, com a diferença de suportar a especificação de contratos de qualidades de serviços, delegados (com o seu comportamento adaptativo) e conexão, criação e inicialização de componentes da aplicação QuO. Tais características constituem um conceito denominado programação orientada a aspectos (*aspected-oriented programming*). Os aspectos são programados em separado da aplicação, através de linguagens de propósito especiais como a QDL. Após a descrição dos aspectos, são feitos mapeamentos automáticos para transformar as especificações em elementos do QuO. No caso do QuO, as linguagens de aspecto são compostas por outras três linguagens:

- *Contract Definition Language*: define as regiões de QoS esperadas, as regiões de QoS reais e as transições de adaptação para mudanças nos níveis de serviço;
- *Structure Description Language*: contém alternativas de comportamento para objetos remotos e seus delegados e também estratégias alternativas de ligações e conexões;
- *Resource Description Language*: determina os recursos de sistema disponíveis e os seus estados.

3.6 Análise Comparativa

Para comparar as tecnologias discutidas ao longo do capítulo, selecionou-se quatro aspectos principais: descrição dos requisitos temporais, tratamento de eventos assíncronos, controle de prioridades e controle de concorrência. A TABELA 3.1 resume a comparação efetuada.

O primeiro aspecto comparado foi o suporte para a descrição de requisitos temporais. Para tanto, o RT-CORBA oferece uma interface *RTCosScheduling*, que permite a descrição dos parâmetros custo, tempo de execução, período, prioridade e importância. Entretanto, verifica-se que estes parâmetros são usados somente como informação ao escalonador off-line, e não como requisitos a serem usados na programação dos mesmos. Por sua vez, o RT-Java sugere o uso de classes como *SchedulingParameters* e *ReleaseParameters* (e suas classes derivadas), podendo representar respectivamente características de escalonamento (como prioridade) e também características de ativação (instante de ativação, período, tempo de execução – ou custo e *deadline*). Já o TMO permite associar condições de ativação autônoma (AAC) aos métodos, definindo assim instantes de ativação, período e *deadline*. Esta mesma tecnologia permite também associar tempos máximos de atualização para as variáveis dos objetos. O modelo AO/C++ permite a definição de operações cíclicas e também operações com *deadlines* e operações de tratamento de exceção por violação dos mesmos. Por fim, a proposta QuO sugere a definição de contratos para estabelecer limites aos requisitos de qualidade de serviço de um modo geral (como as restrições temporais).

Voltando-se para o segundo aspecto comparado, que é o tratamento de eventos assíncronos, verifica-se que o RT-Java oferece um suporte especial para esta questão, através da classe *AsyncEventHandler*. RT-CORBA propõe o conceito de canais de eventos (*event channels*), enquanto o TMO e o AO/C++ tratam simplesmente como uma chamada de operação. O TMO possui a vantagem de verificar possíveis conflitos com os métodos ativados por tempo. Na abordagem QuO são utilizadas as características do *middleware* subjacente para tratar deste requisito.

No que tange a definição de prioridades, somente o RT-CORBA considera diretamente questões relativas à distribuição da aplicação, através dos seus modelos de prioridades. O QuO faz uso implícito destas considerações, visto que é implementado sobre o RT-CORBA. As outras tecnologias tratam do escalonamento de métodos ou objetos apenas num escopo local, onde tanto o RT-Java quanto o TMO escalonam métodos de objetos, enquanto o AO/C++ trata apenas do escalonamento dos objetos ativos através das suas prioridades.

Para finalizar esta comparação, considera-se os mecanismos para controle de concorrência entre os objetos. Tais mecanismos são necessários para promover controle sobre o acesso concorrente aos atributos do objeto, sendo úteis quando os objetos possuírem mais de uma *thread* de execução. Assim, o RT-CORBA e o RT-Java são bastante semelhantes, pois sugerem o uso de monitores. O TMO oferece uma política mais conservativa, impedindo a ativação de métodos em conflito. Novamente, a abordagem QuO faz uso das definições do RT-CORBA. O AO/C++ não necessita oferecer mecanismos para garantir exclusão mútua, visto que cada objeto ativo possui uma única *thread* de controle, evitando assim o acesso concorrente à sua área de dados (incluindo os objetos passivos referenciados).

TABELA 3.1 - Comparação entre as tecnologias para programação e execução de sistemas tempo real

Abordagem/ Conceitos	**RT-CORBA**	**RT-Java**	**TMO**	**AO/C++**	**QuO**
Descrição de requisitos temporais	Interface *RTCosSchedulin g*	Classes derivadas de *Parameters*	MVD var. AAC SpM	Métodos cíclicos e temporizados.	*Contracts (CDL)*
Tratamento de eventos assíncronos	Canais de eventos	*AsyncEventHan dler*	SvM (concorren te com os SpM)	Métodos de obj. ativos (atendimento serializado) FIFO	*System Condition Objects*
Controle de prioridades	Controle. Distribuído: *server-declared*; client-propagated. Prioridades globais e mapeamento	Controle nodo local (*threads* RT)	SpMs têm prioridade sobre SvMs	Controle nodo local (obj. ativos) baseado em prioridades	*Delegates*
Controle de concorrência	Mutex	*Blocos synchronized* com os protocolos *priority ceiling* e *priority inhrtc*	BCC	Objetos são mono-*thread,* em caso de agregação de objetos ativos, estes atuam como monitores	*Delegates*

4 Análise do Perfil UML-TR e do seu Mapeamento para o Nível de Programação

O objetivo deste capítulo é analisar qualitativamente as construções propostas no perfil UML-TR, em especial verificando as facilidades e dificuldades relacionadas com o uso destas construções no desenvolvimento de aplicações tempo real. Adicionalmente, promove-se uma análise no sentido reverso (*bottom-up*), aonde se verifica a adequabilidade das construções do perfil UML-TR para representar os requisitos provenientes das tecnologias de programação apresentadas no capítulo 3 - com exceção do QuO, por ser este estritamente voltado para aspectos de comunicação. Através desta análise, objetiva-se especificar alternativas de implementação para os requisitos encontrados, além de identificar eventuais carências nas construções padronizadas.

O restante do capítulo trata dos *frameworks* introduzidos pelo perfil UML-TR, os quais encontram-se divididos conforme o tipo de requisito abordado, conforme citado no capítulo 2. De maneira mais específica, nas próximas seções identificam-se os conceitos que compõem cada *framework*, procura-se definir as carências na semântica destes elementos e também definem-se alternativas de programação para os mesmos. Por fim são apresentadas as conclusões obtidas a partir da análise efetuada.

4.1 Mapeamento do *Framework* para Modelagem de Tempo

Devido ao caráter genérico do *framework* para modelagem de recursos, o qual contém os elementos básicos do perfil UML-TR, decidiu-se por não fazer uma análise do mesmo. Isto porque não poderiam ser feitas relações diretas entre os seus elementos e o nível de programação. Portanto, inicia-se esta análise pelo *framework* para modelagem de tempo, o qual contém um conjunto de estereótipos para suportar a descrição de requisitos temporais em um modelo UML, conforme descrito na seção 2.3.2. Um aspecto interessante deste *framework* é o fato dele mesclar recursos para a especificação de requisitos temporais propriamente ditos com recursos para especificar mecanismos tipicamente presentes em sistemas operacionais tempo real (SO-TR), usados na programação destes requisitos. Analisando as tecnologias de programação apresentadas no capítulo 3, observa-se que elas oferecem um nível de abstração elevado, de modo que não fazem uso explícito destes mecanismos de programação “de baixo nível”, considerados “de baixo nível”. Entretanto, verifica-se que para aumentar a legibilidade do código, tornando aspectos funcionais separados de requisitos de projeto, a programação de tais requisitos deve ficar localizada em um nível de abstração intermediário, como por exemplo nas funções das APIs (as quais não são visíveis para os usuários da mesma).

O mapeamento dos estereótipos é explicado nesta seção de maneira individual. Procurou-se comentar o mapeamento dos mesmos conforme a ordem em que eles são listados na TABELA 2.1. Partindo de uma explanação sucinta do estereótipo, define-se o mapeamento proposto. Um maior detalhamento dos estereótipos do perfil UML-TR pode ser obtido na referência mencionada.

Inicia-se esta discussão pelo estereótipo *«RTaction»*, o qual modela uma ação que consome tempo para executar. Este requisito é de grande valia pois permite especificar a duração (i.e. tempo de execução) de uma ação. A especificação pode ser feita de duas formas exclusivas entre si: ou através da definição das marcas *RTstart* e *RTend*, que denotam os tempos de início e de fim da ação, ou diretamente através da marca *RTduration*. As duas primeiras marcas podem ser associadas ou com um valor de tempo, representado pelo estereótipo *«RTtime»*, ou também com um intervalo, representado pelo estereótipo *«RTinterval»*. Já a duração está sempre associada com um intervalo. Ressalta-se que na prática não existe diferença entre *«RTtime»* e *«RTinterval»*, visto que a segunda nada mais é do que um caso especial da primeira. Também se chama a atenção pelo fato de que ambas podem ser representadas pela notação TVL do perfil UML-TR. Conforme descrito na seção 2.3.2, esta notação suporta diversos tipos de construções, entre elas algumas funções de distribuição de probabilidade.

O tipo de informação presente no estereótipo *«RTaction»* é útil para o escalonador de tarefas computacionais, o qual necessita conhecer a duração total do conjunto de ações que constituem uma unidade de escalonamento, que é definida no perfil UML-TR como um *scheduling job*. Logo, nota-se que o estereótipo *«RTaction»* não reflete necessariamente uma restrição no código. Uma ressalva é feita quando o projetista deseja especificar um *deadline* como requisito para a ação (o *deadline* pode ser maior ou igual à duração especificada), pois neste caso a implementação deve ser capaz de perceber tal situação. Entretanto, pelas suas características atuais, o estereótipo *«RTaction»* não permite a especificação de *deadline*.

A proposta defendida pelos autores do perfil UML-TR sugere a modelagem explícita de mecanismos para detecção de *deadlines*, exatamente como é feito durante a programação. Na FIGURA 4.1 ilustra-se um diagrama de seqüência que representa a especificação de uma ação com duração de 60 ms, juntamente com um mecanismo capaz de detectar a violação de um *deadline* definido em 100 ms. Deve-se atentar o fato de que neste diagrama o *deadline* está associado ao tempo de disparo do *timer* criado, aonde a duração da ação é acrescida do tempo de criação do *timer*.

Todavia, apesar do perfil UML-TR oferecer os mecanismos apresentados na FIGURA 4.1, acaba-se deparando com uma situação indesejável: enquanto as tecnologias de programação estudadas, na sua maioria, oferecem construções de "alto nível" para controlar a duração das suas ações, o perfil UML-TR sugere o uso dos mecanismos de "baixo nível" presentes nos SO-TR para representar esta restrição. Portanto, constata-se que o estereótipo *«RTaction»* ainda carece de aperfeiçoamentos, a fim de que o mesmo possa oferecer um nível de abstração mais elevado para os projetistas.

Analisando a questão sob o ponto de vista das tecnologias estudadas, verifica-se que as mesmas apresentam diferentes formas de se programar a situação exibida na FIGURA 4.1. No AO/C++, por exemplo, o método *controledAction()* seria do tipo *timed*, sendo que a "ação" fica delimitada na área reservada para uma execução normal (entre as macros BEGIN_OPERATION e END_OPERATION), e o tratamento da violação do tempo de execução fica na área reservada para tal condição (entre as macros BEGIN_EXCEPTION e END_EXCEPTION). Já no RTSJ é necessário executar a ação no escopo de uma classe *Timed*, que recebe como parâmetro uma instância da classe *Interruptible*. Com isto, a ação é definida através da sobrecarga do método *run()* da

classe *Interruptible*. No caso de *timeout*, o método *run()* é interrompido e um código de exceção é chamado.

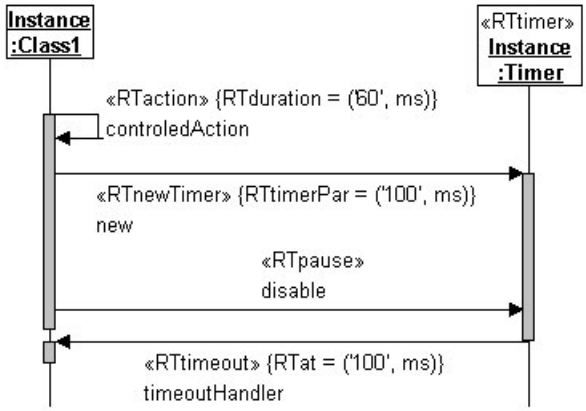


FIGURA 4.1 - Representação UML para execução de ação com controle de *deadline*

Situações diferentes são verificadas no RT-CORBA e no TMO. O primeiro não implementa nenhum mecanismo para controlar a situação em questão, assumindo que as facilidades do SO subjacente podem ser utilizadas para tal finalidade. Já o TMO pressupõe que uma ação nunca terá o seu tempo de execução violado, pois o mesmo é garantido pela GCT - *guaranteed completition time*. Portanto, não se verifica a necessidade de código para tratamento de exceção.

Nota-se portanto que seguindo as definições do perfil UML-TR a modelagem de uma ação com *deadline* (situação apresentada na FIGURA 4.1) pode ser bem mais complexa do que a sua implementação. Isto indica que existem formas mais simples de se especificar os conceitos que se deseja expressar. Assim sendo, propõe-se a criação de um novo estereótipo, denominado *«RTtimedAction»*. Este estereótipo herda de *«RTaction»* e acrescenta um campo que permite a especificação de um *deadline* para a ação (representado pela marca *RTdeadline*).

Seguindo o modelo adotado pelas tecnologias estudadas, a semântica do estereótipo *«RTtimedAction»* indica que uma exceção por *timeout* deve ser gerada caso o *deadline* não seja cumprido. Conseqüentemente, esta exceção deverá ser capturada e posteriormente tratada, sendo que o código de tratamento da mesma deve constar no modelo, a exemplo da especificação permitida pelo ambiente SIMOO-RT. A maneira mais intuitiva de acrescentar este código no modelo, mantendo-se de acordo com a semântica do UML, é justamente através da criação de uma operação de exceção, que estará associada ao método estereotipado com *«RTtimedAction»*.

Para tornar esta associação mais intuitiva, i.e. código de tratamento de exceção e mensagem de exceção gerada pela perda do *deadline*, propõe-se o acréscimo da marca *RTdeadMissException*, cujo intuito é de funcionar como "referência" para uma das exceções definida no modelo. A FIGURA 4.2 mostra como ficaria o mesmo diagrama apresentado na FIGURA 4.1 com o uso do estereótipo *«RTtimedAction»*. Note que o método *timeoutHandler()* apresentado na FIGURA 4.1 seria equivalente a exceção *DeadMissException* da FIGURA 4.2. Por fim, acrescenta-se que a operação de tratamento de exceção associada com ações também poderia ter um estereótipo próprio (e.g. *RTactionDeadMissHandler*), a fim de diferenciá-la de outras possíveis exceções sendo tratadas no sistema.

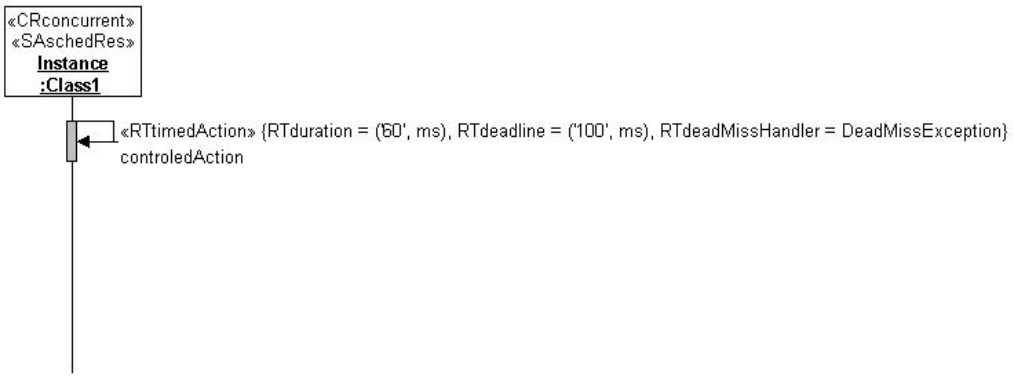


FIGURA 4.2 - Sugestão de estereótipo para ação controlada no tempo

Neste ponto cabe enfatizar a abrangência do conceito de ação, visto que a proposta aqui apresentada entra em conflito com a especificação da OMG no perfil UML-TR. Enquanto esta última propõe relacionar "ação" à chamada de um método, e não à sua execução propriamente dita, aqui se propõe a associação de ação com a execução do método. Além disso, a definição da OMG sugere o uso do estereótipo *«CRcontains»* (do *framework* para modelagem de concorrência) para representar a chamada "dependência de uso[9]" entre os métodos de classe e as ações que os chamam. Diante do contexto exposto, utilizou-se dos mecanismos oferecidos pela OMG para solicitar que o conceito ação englobe também a execução de um método. Portanto, considera-se ao longo deste trabalho a proposta recém apresentada para o conceito de ação em detrimento à especificação da OMG.

Feita essa ressalva, retoma-se a análise do *framework* para modelagem de tempo com o estudo do estereótipo *«RTdelay»*, que define uma simples ação de atraso (ou espera) por um tempo determinado. O mapeamento do mesmo para as tecnologias estudadas é direto, sendo representado por uma simples chamada da função.

Na seqüência, tem-se os estereótipos *«RTclkInterrupt»* e *«RTclock»*, que servem respectivamente para representar uma interrupção de *clock* e o *clock* propriamente dito. Visto que estes estereótipos recaem nos denominados "mecanismos de baixo nível", não se define uma representação explícita dos mesmos nas APIs das tecnologias estudadas. Uma situação típica de modelagem na qual se usa o estereótipo *«RTclock»* é para representar a geração de um evento periódico, conforme exibido na FIGURA 4.3. Como pode ser visto na figura, o evento gerado é estereotipado com *«RTevent»*. Apesar desta situação ser comumente encontrada em aplicações tempo real, um possível mapeamento da mesma para o nível de programação depende das características da ação de resposta ao evento. Por exemplo, é importante saber se a ação constitui ou não uma nova unidade de escalonamento. Como este tipo de informação está fora do escopo do *framework* para modelagem de tempo, pois é tratada pelo *framework* para análise de escalonabilidade (vide seção 4.3), o mapeamento da mesma é tratado posteriormente neste trabalho.

[9] Representa uma relação onde um elemento requer a existência de outro para completar a sua implementação.

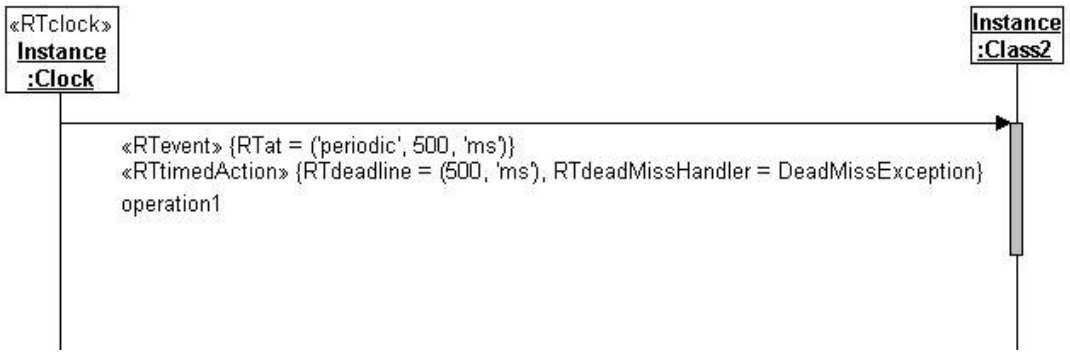


FIGURA 4.3 - Definição de um evento periódico

Ainda sobre o estereótipo *«RTevent»*, verifica-se que o mesmo tem um significado semântico muito parecido com o *«RTaction»*, sendo usado nos casos onde a duração ou o tempo de término da ação não é significante. A marca *RTat* é usada para armazenar o instante de tempo relacionado com a ocorrência (i.e. início) do evento. Esta marca possui as mesmas características da marca *RTstart*, que quando descrita pela notação TVL pode incluir um padrão de ocorrência para o evento (i.e. periódico, esporádico, etc.). Apesar de abrangente, nota-se que uma expressão TVL não consegue representar um intervalo no qual um evento periódico é gerado. Por exemplo, em TVL não é possível representar a expressão AAC suportada pela API do TMO, mostrada na FIGURA 3.12.

Na seqüência analisa-se o estereótipo *«RTinterval»* que, conforme discutido anteriormente, representa um valor de tempo correspondente a um intervalo. Este estereótipo contém as marcas *RTintStart*, *RTintEnd* e *RTintDuration* (seus significados são os mesmos das marcas correspondentes em *«RTaction»*). Apesar deste estereótipo ser bastante sugestivo, praticamente inexiste suporte para a sua programação por parte das tecnologias pesquisadas. Tomando o TMO como exemplo, nota-se que intervalos de tempo se encontram implícitos nas AACs. Nas outras tecnologias é possível calcular um intervalo de tempo através da diferença entre dois instantes, valores estes que podem ser obtidos pela leitura do *clock* do sistema através das facilidades oferecidas nas APIs. No RTSJ, esta diferença pode ser calculada através do método *subtract()*, pertencente às classes que representam instantes de tempo (*RelativeTime* e *AbsoluteTime*).

Voltando para o *framework* em análise, o mesmo também define outros estereótipos para representar mecanismos de SO-TR, como a criação propriamente dita (i.e. instanciação) de *clocks* e *timers*. Estes mecanismos são respectivamente representados pelos estereótipos *«RTnewClock»* e *«RTnewTimer»* (vide exemplo de *«RTnewTimer»* na FIGURA 4.1). Discutiu-se anteriormente que *clocks* não precisam ser explicitamente criados nas tecnologias estudadas, logo o primeiro estereótipo deverá ser pouco utilizado. Já a criação de *timers* é um procedimento utilizado com freqüência na programação, tendo suporte em algumas APIs, como o RTSJ, e principalmente nos SO-TR. No RTSJ, tal criação decorre da alocação de um objeto ou da classe *OneShotTimer* ou da classe *PeriodicTimer*, conforme mostrado na FIGURA 3.9.

Além destes dois mecanismos, ainda são definidas outras operações que podem ser efetuadas sobre os *clocks* e os *timers*, tal como "definir valor", "iniciar contagem", "parar contagem" e "reiniciar contagem". Os estereótipos que representam estas operações são, respectivamente, *«RTset»*, *«RTstart»*, *«RTpause»* e *«RTreset»*. Embora

algumas tecnologias estudadas ofereçam alternativas para a manipulação destas operações, nem sempre elas possuem a mesma semântica. Um exemplo é a operação "parar contagem", que no RTSJ permite somente desabilitar o recurso (*timer*) para o disparo de eventos, isto sem interromper a contagem enquanto desativado.

Continuando a análise dos estereótipos se chega no *«RTstimulus»*, utilizado para representar um estímulo[10] que possui uma marca de tempo associada (vide FIGURA 2.5). Junto com este estereótipo são definidas as marcas *RTstart* e *RTend*. Este elemento é interessante de ser utilizado naqueles sistemas onde o tempo de propagação de mensagens ou sinais são significantes, como nos sistemas distribuídos. Com isto, esta informação pode ser usada, por exemplo, como um requisito de QoS para a comunicação.

O próximo estereótipo analisado é o *«RTtime»*, usado para modelar um valor de tempo. Esta representação serve como alternativa ao uso da notação TVL para modelar elementos genéricos, porém com alguma semântica de tempo (e.g. a representação de uma data). As marcas associadas com este estereótipo são *RTkind*, que indica o modo como o tempo avança (e.g. de modo discreto ou contínuo), e *RTrefClk*, que sinaliza algum elemento do modelo usado como *clock* de referência.

Na seqüência, tem-se os estereótipos *«RTtimer»* e *«RTtimeout»*, exemplificados na FIGURA 4.1. Assim como o *clock*, este mecanismo está mais relacionado com os recursos do SO do que com um requisito de modelagem. Como o próprio nome sugere, o primeiro é usado para modelar um *timer*, sendo que possui duas marcas associadas. A marca *RTperiodic* define se o *timer* é aperiódico (*RTperiodic* = *false*) ou periódico (*RTperiodic* = *true*) (no exemplo da FIGURA 4.1 o *timer* é aperiódico, por isso a marca não é exibida). Por sua vez, a marca *RTduration* é o parâmetro que indica a sua duração. Já o estereótipo *«RTtimeout»*, como o nome sugere, representa o estímulo gerado como conseqüência do término da contagem realizada pelo *timer*. No RTSJ, uma das poucas APIs que aborda a criação de *timers*, o *timeout* é tratado internamente, cabendo ao programador associá-lo a uma operação de tratamento (*handler*) para este estímulo. As outras tecnologias possuem suporte do SO subjacente.

Outros dois estereótipos deste modelo, *«RTtimeService»* e *«RTtimingMechanism»*, não são usados explicitamente na modelagem, mas servem de base a outros elementos do *framework* relacionados com mecanismos dos SO-TR. Enquanto o primeiro se aplica aos serviços de criação de *clocks* e *timers*, o segundo se aplica à caracterização destes elementos. No total, estes elementos contêm dez marcas associadas, cujo detalhamento foge ao escopo deste trabalho. Ainda assim, ressalta-se que nem todas as marcas de *«RTtimingMechanism»* podem ser configuradas nos ambientes de execução das tecnologias levantadas, como por exemplo as marcas listadas abaixo (principalmente por serem características intrínsecas destes ambientes):

- *RTorigin*: evento cuja ocorrência representa o início da contagem;
- *RTmaxValue*: valor máximo que o mecanismo de tempo pode assumir;
- *RTstability*: habilidade do mecanismo de tempo para fornecer intervalos de tempo consistentes;
- *RTdrift*: diferença absoluta máxima entre a freqüência do mecanismo de tempo em relação à freqüência do seu *clock* de referência;

[10] Definido pela UML 1.4 como uma instância de uma comunicação em trânsito entre um objeto chamador e o objeto chamado.

- *RTskew*: taxa de mudança do *offset* entre o mecanismo de tempo e o seu *clock* de referência.

4.2 Mapeamento do *Framework* para Modelagem de Concorrência

Nesta seção é analisado o *framework* para modelagem de concorrência, que como o próprio nome sugere serve para permitir a descrição de objetos concorrentes e das suas características de comunicação. Tal descrição permite uma associação direta com a infra-estrutura oferecida pelo ambiente de execução subjacente. No contexto do mapeamento proposto, este *framework* é importante para explicitar os objetos concorrentes (ativos) da aplicação e a maneira como eles se comunicam. A TABELA 2.2 contém os elementos deste *framework* que são descritos nesta seção.

O primeiro estereótipo abordado é o *«CRaction»*, que representa a execução de uma ação. Este estereótipo possui apenas uma marca, *CRatomic*, usada para caracterizar se a ação pode ou não sofrer preempção. As tecnologias estudadas não oferecem alternativas para representar esta restrição. Analisando a documentação do perfil UML-TR, chega-se à conclusão que este estereótipo estende o conceito definido para o estereótipo *«RTaction»*. Todavia, a especificação não define nenhum tipo de relação entre eles (seria natural que *«CRaction»* herdasse de *«RTaction»*). Logo, entra-se novamente em conflito com a definição do perfil UML-TR. Além disso, a definição da OMG sugere o uso do estereótipo *«CRcontains»* para representar a chamada "dependência de uso[11]" entre os métodos de classe e as ações que as chamam. Semelhante ao ocorrido com o estereótipo *«RTaction»*, aqui também se defende que o conceito de ação deva englobar também a execução de um método, além, é claro, de representar um bloco de código. Ressalta-se que toda a discussão feita na seção anterior sobre o mapeamento de *«RTaction»* para as tecnologias analisadas também é válida para o presente estereótipo.

Outros dois estereótipos relacionados são o *«CRasynch»*, que representa a (ação) invocação de uma operação assíncrona, e o *«CRsynch»*, que representa a (também ação) invocação de uma operação síncrona, conforme ilustrado na FIGURA 4.4[12]. As tecnologias estudadas no capítulo 3 oferecem facilidades para programar ambos os tipos de invocação. No AO/C++ por exemplo, existe o conceito de invocações remotas síncronas e assíncronas. Já no TMO existem as denominadas invocações bloqueantes (síncronas) e não-bloqueantes (assíncronas). Por sua vez, o RT-CORBA define chamadas do tipo *one-way* (assíncronas) e chamadas *two-way* (síncronas). Em relação ao RTSJ, apesar do mesmo não abordar mecanismos de distribuição, também é possível realizar invocações síncronas e assíncronas. Um exemplo de invocação assíncrona no RTSJ seria a ativação de um evento assíncrono no receptor, definido pela classe *AsyncEventHandler* (semelhante ao mecanismo de tratamento de *timeout* em *timers* explicado na seção anterior). Já as invocações síncronas são feitas de maneira intuitiva, através de uma simples chamada de método.

---

[11] Representa uma relação onde um elemento requer a existência de outro para completar a sua implementação.

[12] Acrescenta-se o estereótipo *«CRconcurrent»*, porque os mecanismos que implementam este tipo de comunicação nas APIs abordadas pressupõem o uso de elementos concorrentes.

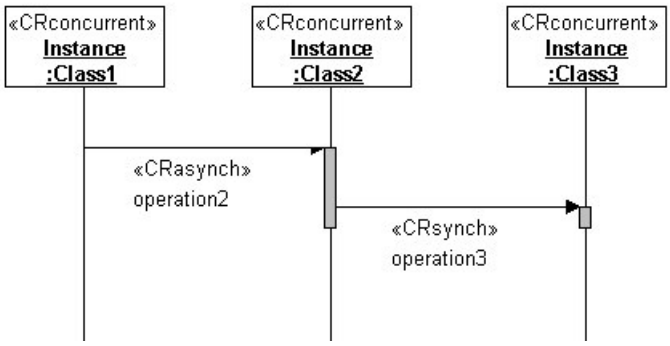


FIGURA 4.4 - Ilustração chamadas síncronas e assíncronas

As tecnologias que implementam um tempo máximo de espera por resposta ao invocar uma operação (logicamente que de maneira síncrona) são o TMO e o RT-CORBA. No RT-CORBA, em caso de não cumprimento do requisito uma exceção é gerada, sendo que o respectivo tratamento pode ser definido em uma cláusula *try-catch*. Já no TMO a chamada retorna um código de erro indicando a ocorrência de *timeout*, de modo que cabe ao cliente testar se a comunicação transcorreu normalmente ou se ocorreu erro. Esta situação, bastante pertinente no contexto de uma aplicação tempo real, pode ser modelada conforme reportado na FIGURA 4.5[13]. Esta figura apresenta uma nova marca denominada *CRcallTimeout*, que foi acrescentada neste exemplo (e sugerida como requisito pendente no perfil UML-TR) para tornar esta informação explícita, i.e. que o projetista a utilize para configurar o *timeout* da chamada através da marca *CRcallTimeout*.

Semelhante à discussão feita na seção 4.1 para o estereótipo *«RTtimedAction»*, uma ação pode ser tomada em detrimento à ocorrência do *timeout*. Assim, optou-se por utilizar uma abordagem semelhante, ou seja, adicionar também uma marca (*CRcallTimeoutMissHandler*) para fazer referência à operação de tratamento de exceção a ser invocada em caso de *timeout*. Cabe aqui enfatizar a diferenciação desta especificação com a restrição exibida na FIGURA 4.2. Esta última prima por especificar um tempo máximo para a realização de uma ação sem que a mesma envolva necessariamente uma invocação de operação. Eventualmente, as duas restrições poderiam fazer parte do mesmo diagrama, conforme exibido na FIGURA 4.6. Nota-se que existem duas rotinas de tratamento de exceção explícitas neste diagrama, uma para o método *controledAction()* (vide marca *RTdeadMissHandler*) e outra para o método *operation1()* (vide marca *CRcallTimeoutMissHandler*). É importante ficar claro que são duas rotinas de tratamento de *timeout* independentes, sendo que podem existir casos onde as duas, apenas uma ou até mesmo nenhuma rotina é executada. Enfatiza-se que o objetivo deste trabalho é discutir uma alternativa para especificar que exceções podem ser geradas e que devem ser tratadas. Entretanto, discussões sobre a forma como as mesmas são tratadas fogem ao escopo desta tese. Para um maior aprofundamento sobre este tema, recomenda-se a leitura das publicações de Randell et al [BUR 98; ROM 99].

[13] A programação desta restrição no RT-CORBA e no TMO é exibida no capítulo 3.

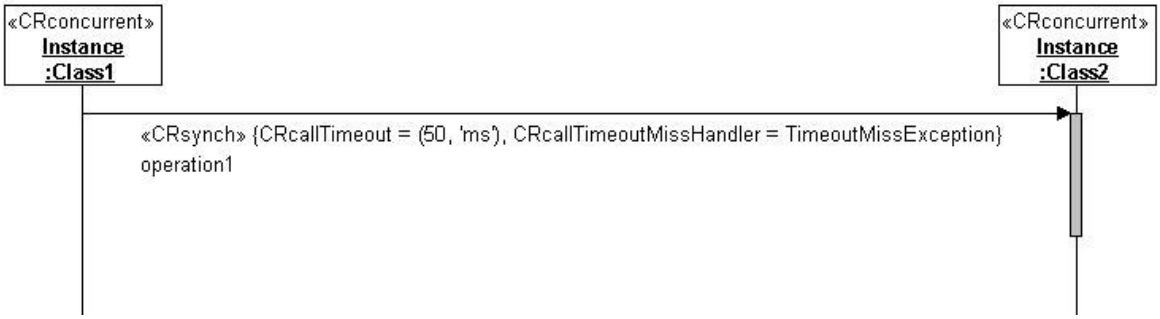


FIGURA 4.5 - *Timeout* associado com a invocação de uma operação síncrona

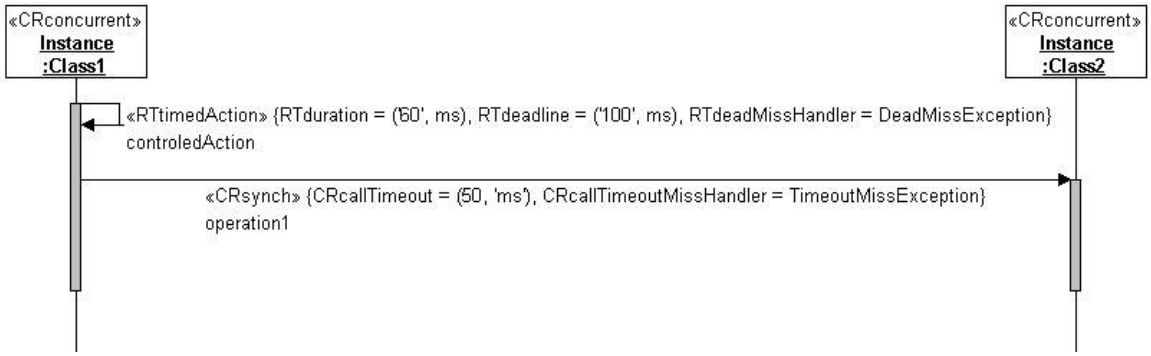


FIGURA 4.6 - Ação com limite de tempo junto com invocação com limite de tempo

O estereótipo *«CRconcurrent»* indica que a classe (ou objeto) com a qual estiver relacionado representa uma entidade concorrente, de modo semelhante ao conceito de classe ativa (cujas instâncias são objetos ativos). Associado ao estereótipo encontra-se a marca *CRmain*, que faz referência ao método invocado ao iniciar a execução da unidade concorrente. Esta definição é muito importante, uma vez que a especificação de entidades concorrentes é fundamental nas tecnologias de programação estudadas. O AO/C++ por exemplo agrega Nesta tecnologia, a marca *CRmain* estaria associada ao método in *start()*. Já no TMO as entidades concorrentes são denominadas *objetos TMO*, sendo que também podem encontrar-se distribuídas em diferentes nodos computacionais. Neste modelo não se consegue definir apenas uma única marca *CRmain*, mas sim múltiplas, visto que elas são representadas pelo próprio corpo dos SpMs e SvMs especificados junto ao *objeto TMO*. A linguagem AO/C++ agrega o conceito de classes ativas, cujas instâncias são objetos independentes e que podem inclusive executar em uma máquina remota. Esta linguagem adota um modelo semelhante ao do TMO, pois os não existe uma única linha de controle. No RTSJ a marca *CRmain* equivale ao método *run()* da *thread* concorrente. Finalmente no RT-CORBA as instâncias dos objetos servidores são utilizadas como unidade de execução concorrente, sendo o mesmo alocado em uma *thread*. A marca *CRmain* seria equivalente ao método *activate()* do objeto servidor.

O *framework* para modelagem de concorrência também faz menção sobre a forma com que um objeto trata as requisições de serviços, que podem ser feitas de modo imediato ou adiadas para um instante posterior. O primeiro, representado pelo estereótipo *«CRimmediate»*, implica no atendimento imediato ao chamado. Este

atendimento pode ser dar na mesma *thread* do invocador (*CRthreading = 'remote'*) ou em uma *thread* local do objeto chamado (*CRthreading = 'local'*). No segundo caso o objeto receptor também deverá ser uma entidade concorrente (*«CRconcurrent»*). Já as requisições representadas pelo estereótipo *«CRdeferred»* indicam que o próprio objeto receptor decide quando processar as solicitações pendentes (logo não são imediatas). Neste caso o objeto receptor também é uma entidade concorrente (*«CRconcurrent»*).

Analisando as possibilidades de se mapear para código as requisições citadas, verifica-se que as tecnologias estudadas oferecem diversas alternativas. Analisam-se primeiramente as chamadas *«CRimmediate»* com *CRthreading = 'remote'*. Estas nada mais são do que chamadas normais de métodos para objetos ditos passivos, i.e. que se encontram no mesmo processo ou *thread* do objeto chamador. Todavia, *CRthreading = 'local'* retrata uma situação típica de comunicação do tipo cliente-servidor, como no caso do TMO e RT-CORBA. Neste último, a execução dos métodos é feita nas *threads* previamente alocadas nas "filas de *threads*". Já no RTSJ, apesar do mecanismo cliente-servidor não estar disponível (uma vez que este não implementa mecanismos para distribuir a aplicação), o mesmo é passível de ser implementado com o auxílio de tratadores de eventos assíncronos (instâncias de *AsyncEventHandler*). Já o AO/C++ opera tratanto as solicitações como sendo do tipo *«CRdeferred»*. Neste último, o estereótipo *«CRmsgQ»*, é utilizado nos seus métodos internos para representar uma fila de espera de chamadas para processamento futuro.

4.3 Mapeamento do *Framework* para Análise de Escalonabilidade

O último elemento do perfil UML-TR a ser analisado neste trabalho é o *framework* para análise de escalonabilidade, o qual é listado da TABELA 2.3. Inicia-se esta análise pelo estereótipo *«SAresource»*, que é usado para caracterizar um recurso protegido, acessado durante a execução de um *scheduling job* - que no contexto do perfil UML-TR indica a combinação de um estímulo com uma resposta. O acesso ao recurso poderá ser feito por múltiplos *jobs* concorrentes, por isso precisa ser protegido por um mecanismo de controle de acesso. Este estereótipo possui um conjunto de marcas associadas, as quais indicam as propriedades do recurso e dos seus serviços (referenciados no perfil por "características de QoS do recurso") utilizados para fins de análise de escalonabilidade. Portanto, muitas destas propriedades não aparecem diretamente no código da aplicação, como por exemplo:

- *SAacquisition*: tempo de espera entre a autorização para obter o recurso e a disponibilidade do mesmo;
- *SAdeacquisition*: tempo de espera de uma ação entre o início da liberação do recurso e o instante em que a mesma se torna apta à execução;
- *SAconsumable*: valor booleano que indica se o recurso é ou não consumido durante o seu uso;
- *SApreemptible*: valor booleano para informar se o recurso pode ou não sofrer preempção durante o seu uso.

Todavia, existem marcas que podem ser usadas para configurar o código da aplicação, conforme listado a seguir:

- *SAaccessControl*: define uma dentre as políticas de controle de acesso suportadas pelo ambiente de execução (e.g. *priority inheritance* ou *priority ceiling*);
- *SAcapacity*: indica o número máximo de usuários simultâneos;
- *SAptyCeiling*: representa a prioridade máxima (*ceiling*) passível de ser utilizada pelo protocolo de controle de acesso *priority ceiling*.

A FIGURA 4.7 exemplifica o uso deste recurso. Um detalhe importante é o uso de dois estereótipos que não foram introduzidos até então por pertencerem ao modelo geral de recursos, que forma a base do perfil UML-TR, conforme discutido na seção 2.3.1. Estes estereótipos são chamados *«GRMacquire»* e *«GRMrelease»*, sendo que servem respectivamente para solicitar e para liberar um recurso compartilhado. As tecnologias estudadas não provêem elementos que permitam um mapeamento direto destes estereótipos. Contudo, verifica-se que o estereótipo *«GRMacquire»* precisa de um parâmetro adicional para controlar um tempo máximo de espera pelo recurso, uma vez que o mesmo pode bloquear ao solicitar um recurso em uso (isto ocorre quando a marca *GRMbloquing* for verdadeira). Logo, sugere-se a criação de duas novas marcas para sanar esta carência: *SAbloqTimeout*, que caracteriza o tempo máximo de espera pelo recurso, e *SAbloqtTimeoutMissHandler*, que contém uma referência para o código de tratamento de exceção invocado em caso de violação do *timeout*.

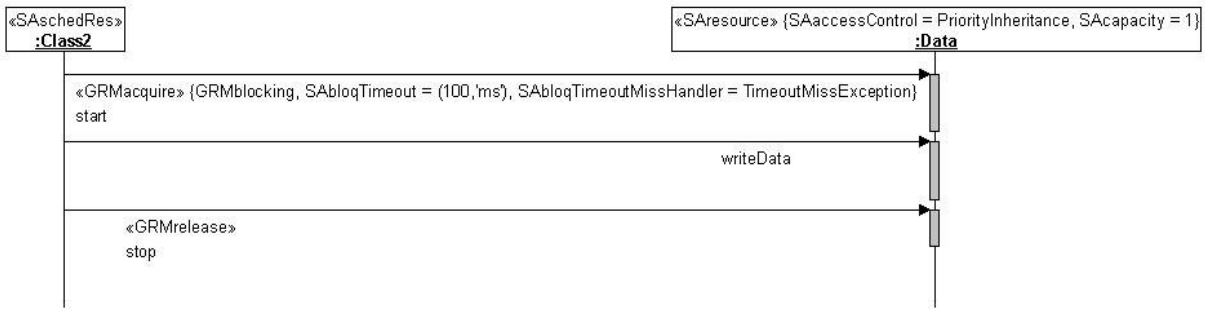


FIGURA 4.7 - Representação de um recurso protegido

Analisando o mapeamento do estereótipo *«SAresource»* para as tecnologias estudadas, chega-se à conclusão que na maioria delas existe um mecanismo correspondente. No RTSJ a representação é dada por qualquer classe que implementa um controle de acesso através da classe *MonitorControl*, onde *SAaccessControl* seria usada para caracterizar o protocolo de controle de acesso utilizado. Já no TMO, este tipo de recurso poderia caracterizar um ODS, entretanto as marcas seriam desprezadas, visto que as características deste elemento são todas pré-definidas. O RT-CORBA oferece elementos do tipo Mutex, sendo possível configurar o ORB para determinar qual a política de controle de acesso a ser implementada. Por fim, verifica-se que o AO/C++ não implementa mecanismos para garantir exclusão mútua, uma vez que cada objeto ativo possui uma única *thread* de controle, evitando assim o acesso concorrente à sua área de dados (incluindo os objetos passivos referenciados). Ressalta-se entretanto que monitores poderiam ser adicionados em caso de alteração no mecanismo de controle dos objetos ativos.

Um *«SAresource»* também serve como base para um recurso de escalonamento, representado pelo estereótipo *«SAschedulable»*, que herda as propriedades do primeiro.

O perfil UML-TR define um recurso de escalonamento como "um recurso concorrente utilizado para executar *jobs*". Com isto, faz-se uma associação natural deste elemento com o estereótipo *«CRconcurrent»* definido no *framework* para modelagem de concorrência. Todavia, a especificação não define nenhum tipo de relacionamento entre estes elementos. Logo, discorda-se novamente com a organização do perfil UML-TR, porque neste perfil ambos os elementos são especificados como "uma unidade de execução concorrente, tal como uma tarefa, um processo, ou uma *thread*". Nos aspectos que tangem a implementação, ambos estereótipos, *«CRconcurrent»* e *«SAresource»*, são programados da mesma forma, valendo assim a discussão feita na seção anterior.

Utiliza-se o recurso de escalonamento descrito anteriormente para a execução de uma ou mais ações, representadas pelo estereótipo *«SAaction»*. Este por sua vez é derivado de *«RTaction»*, descrito na seção 4.1. Com isto, toda discussão de mapeamento feita anteriormente continua válida. Contudo, verifica-se também que este estereótipo deveria herdar as características de *«RTtimedAction»* (estereótipo proposto nesta tese), herdando as marcas *RTdeadline* e *RTdeadMissHandler*. Ao se analisar as marcas definidas pelo perfil UML-TR para o estereótipo *«SAaction»*, percebe-se que a marca referente ao deadline se encontra especificada, faltando apenas a referência ao método de tratamento de exceção.

Analisando este estereótipo (vide TABELA 2.3) se verifica que elas são totalmente voltadas para um processo de análise de escalonabilidade. A principal utilidade deste processo é argumentar se o sistema (representado através de um modelo UML com os estereótipos aqui apresentados) é ou não escalonável. Além disso, muitas das políticas de escalonamento conhecidas geram as prioridades que vêm a ser utilizada para caracterizar as ações. Em resumo, apenas duas das marcas listadas são utilizadas na fase de programação do modelo: *SApriority* (usada em algoritmos de escalonamento com prioridades fixa, como *Rate Monotonic* e *Deadline Monotonic*), e também *SAabsDeadline* e/ou *SArelDeadline* (usadas em algoritmos de escalonamento dinâmicos, como *Earliest Deadline First*). Alguns ambientes de execução, como o RT-CORBA TAO [SCH 98], possuem um módulo para prover análise de escalonabilidade. Uma das informações relevantes para este módulo são as dependências entre os componentes do sistema, causadas pela invocação de operações de outros recursos de escalonamento (*«SAschedulable»*). Estas dependências podem ser expressas pela marca *SAusedResource*, visto que o estereótipo *«SAschedulable»* é uma especialização de *«SAresource»*.

O estereótipo *«SAresponse»* estende a idéia de ação definida por um *«SAaction»* para caracterizar aquilo que o perfil UML-TR chama de "situação de uso"[14] (*«SAsituation»*). Um *«SAResponse»,* em conjunto com um *«SAtrigger»,* representa um conjunto de interações que representam um *job*. Da mesma forma como fora definido anteriormente para as ações, uma resposta também deverá estar associada com um recurso de escalonamento.

Já o estereótipo *«SAtrigger»* deriva de *«RTevent»* e define duas novas marcas: *SAschedulable* e *SAprecedents*. A primeira resulta do processo de análise de escalonabilidade, e indica se o evento é ou não escalonável. Já a segunda informa o conjunto de ações que devem ser executadas antes da ocorrência do evento. Nota-se portanto, que ambas as marcas são de uso exclusivo da análise, e não possuem representação no código. Todavia, apesar do *«SAtrigger»* não adicionar novas marcas

[14] Na literatura, situações de uso são muitas vezes reportadas como sendo uma atividade.

que vão se transformar em código, ele se diferencia conceitualmente de um *«RTevent»* por associar-se com a ativação de um *«SAResponse»* (que constitui um *scheduling job*). Como conseqüência, é possível programar com facilidade uma situação típica em aplicações tempo real: unidades de escalonamento com ativação periódica (vide diagrama da FIGURA 4.8[15]). Portanto, apesar das semelhanças entre os diagramas presentes na FIGURA 4.3 e na FIGURA 4.8, fica ressaltada a diferenciação semântica entre os mesmos.

Chama-se a atenção para a semelhança na semântica do diagrama na FIGURA 4.8 em relação à da construção sugerida para representar este mesmo tipo de ocorrência no ambiente SIMOO-RT, mostrada na FIGURA 2.1. Ao analisar ambos diagramas, verifica-se que a proposta apresentada no ambiente SIMOO-RT é mais simples, apesar dela possui a desvantagem de não usar elementos padronizados. Além de inserir um símbolo adicional à notação da UML para representar o padrão de ativação, a notação do SIMOO-RT ainda define uma construção fora dos padrões UML, uma vez que a origem da mensagem periódica, ou seja, um *Clock*, não é mostrada no diagrama, pois a sua procedência é implícita.

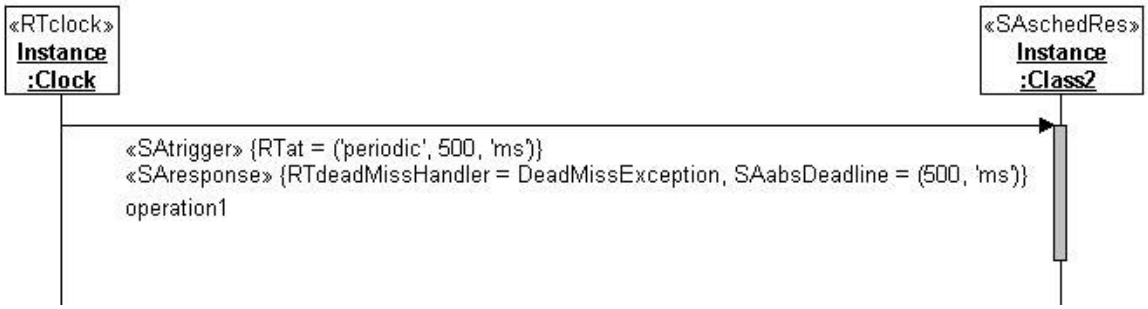


FIGURA 4.8 - Modelagem de uma unidade de escalonamento com ativação periódica

Todas as tecnologias estudadas oferecem recursos de programação que possibilitam o mapeamento dos conceitos discutidos nesta seção. No caso do AO/C++, as respostas com ativação periódica são mapeadas para métodos cíclicos das classes ativas. Assim, o método *operation1()* da FIGURA 4.8 seria do tipo *cyclic* (vide FIGURA 3.18). Respostas com outros tipos de ativação são mapeadas para métodos temporizados ou para métodos normais, porém de classes ativas. Contudo, na implementação atual não é possível associar prioridades às respostas no AO/C++, visto que elas são escalonadas por ordem de chegada (algoritmo do tipo FIFO). Situação semelhante ocorre no TMO, onde as respostas periódicas são representadas por um *SpM*, e as outras ou são um *SvM* ou são um método normal da classe. Uma diferença entre estes dois mecanismos reside no fato de que o algoritmo de escalonamento do TMO dá sempre maior prioridade às *SpMs*.

No RTSJ, por sua vez, quando ocorre ativação periódica é necessário criar uma instância de *RealtimeThread*, cujo padrão de ativação é definido pela classe *PeriodicParameters* (vide FIGURA 3.6). Com isto, as "respostas" correspondem à sobrecarga do método *run()* da classe *RealtimeThread*, sendo que para implementar as ocorrências periódicas este método deve incluir a chamada do método *waitForNextPeriod()* dento de um laço. Se as "respostas" possuírem outro padrão de ocorrência, então elas podem ser mapeadas para outros métodos normais ou até mesmo

[15] Esta figura usa o estereótipo *«SAschedRes»* para referenciar o estereótipo *«SAschedulable»*.

para instâncias da classe *AsyncEventHandler*. Apesar desta diferença considerável no modo como o RTSJ trata as respostas, as mesmas são sempre escalonadas por ordem de chegada. Uma outra característica do RTSJ é o fato dele permitir a associação de um tratador de exceção para o caso da *thread* tempo real não terminar a atividade antes do *deadline*. Isto é feito através da passagem de uma instância da classe *AsyncEventHandler* para o método *setDeadlineMissHandler()* da classe *PeriodicParameters*.

Por outro lado, no RT-CORBA não existe uma estrutura pré-definida para facilitar a programação destas atividades. Cabe ao programador codificar as chamadas ao ORB tempo real para informar que um evento periódico deverá ser criado junto a um objeto servidor. O programador também necessita relacionar o código que representa a resposta com a captura do evento periódico a ser gerado pelo ORB, conforme mostrado na FIGURA 3.1. No que diz respeito a outros tipos de ativação, o tratamento é semelhante àquele definido pelo AO/C++.

Continuando a análise do modelo de escalonabilidade do UML-TR, chega-se no estereótipo *«SAEngine»*, que se refere a uma entidade capaz de suportar a execução dos recursos escalonáveis. Portanto, o mesmo poderia se referir ao SO do AO/C++ (QNX ou Linux), ao SO do TMO (Windows), ao ORB tempo real do RT-CORBA, ou ainda à RT-JVM do RTSJ. O estereótipo em questão reúne um conjunto de marcas que caracterizam as propriedades destes ambientes subjacentes, sendo que o detalhamento dos mesmos foge do escopo desta tese.

Também é constatada a presença de outros estereótipos no modelo, que são utilizados como modo alternativo de se especificar algumas marcas já definidas. Estas marcas não possuem representação em código, visto que são úteis para a análise de escalonabilidade. Os estereótipos em questão são: *«SAusedHost»*, *«SAuses»*, *«SAsituation»*, *«SAOwns»* e *«SAPrecedes»*.

Ao finalizar a análise proposta para esta seção se nota que muitos conceitos presentes no subperfil de escalonamento do UML-TR não são suportados pelas tecnologias estudadas. Justifica-se esta ausência pelo fato de que muitos destes parâmetros estão voltados para uma análise utilizada tipicamente antes da execução do sistema, denominada análise off-line, que visa verificar se o sistema é ou não escalonável.

4.4 Contribuições da Análise Efetuada

Conforme observado no final do capítulo 2 percebe-se que o perfil UML-TR possui um caráter genérico, i.e. permite a modelagem dos mais variados mecanismos de suporte para a programação de aplicações tempo real. Nota-se contudo, que muitas vezes as alternativas de modelagem para alguns requisitos tratados de forma bastante simples nas tecnologias estudadas requerem modelos com um certo grau de complexidade. Identifica-se portando uma contradição: o modelo de alto nível é mais complexo do que o modelo de programação.

Isto decorre da constatação que os elementos de modelagem presentes no perfil UML-TR estão voltados para as bibliotecas de programação presentes nos SO-TR. Contudo, deseja-se neste trabalho realizar implementações usando APIs com maior poder de abstração, que livra o programador de detalhes de implementação.

Conseqüentemente, uma vez que a programação de requisitos temporais é simplificada, a sua especificação deve sofrer o mesmo processo de simplificação.

Para contornar esta situação, foram introduzidos novos estereótipos e marcas ao longo das seções que antecederam esta análise, de modo a aumentar seu o poder de expressão e simplificar a especificação de requisitos comumente encontrados em aplicações tempo real. Além disso, também se abordou alterações na semântica de alguns elementos do perfil UML-TR, principalmente por considerá-los muito abstratos e com uso limitado. A seguir são listadas as observações e as modificações propostas ao longo deste capítulo:

1. Alteração da semântica do *«RTaction»* e seus derivados para incluir operações: esta alteração é motivada pelo fato de se considerar uma operação como passível de ser caracterizada pelas marcas existentes neste estereótipo.
2. Inclusão do estereótipo *«RTtimedAction»*: este elemento estende *«RTaction»* para representar uma ação com restrição de tempo, incluindo assim as marcas *RTdeadline* e *RTdeadMissHandler*. A primeira foi incluída por ser uma informação de grande relevância no contexto de aplicações tempo real (outrora definida somente no modelo de análise de escalonabilidade). Adicionalmente, verifica-se que algumas tecnologias de programação oferecem tratamento de exceção para violações de *deadline*, logo as ações também devem acrescentar uma referência para a operação que deverá tratar a violação, representada pela segunda marca adicionada.
3. Falta de expressividade da notação TVL: observa-se a carência da notação TVL para representar a geração de eventos periódicos durante intervalos de tempo pré-estabelecidos, aonde poderia ser usada a sintaxe do TMO ou do PEARL.
4. Estereótipo *«CRaction»* deveria ser derivado de *«RTtimedAction»*: pela sua semântica, *«CRaction»* pode ser considerado uma especialização de *«RTaction»*. Não obstante, por estar se tratando de aplicações tempo real, ele deve herdar as características do estereótipo *«RTtimedAction»*.
5. Alteração da semântica dos estereótipos «*CRsynch*» e «*CRasynch*» para incluir operações: esta alteração é motivada pelo fato de se considerar uma operação como passível de ser caracterizada pelas marcas existentes neste estereótipo.
6. Acréscimo das marcas *CRcallTimeout* e *CRcallTimeoutHandler* ao estereótipo «*CRsynch*»: estas marcas permitem, respectivamente, associar um tempo máximo de espera com uma chamada síncrona e também uma referência ao código de tratamento de exceção, acionado caso a chamada demorar mais tempo para retornar do que fora especificado pelo *timeout*.
7. Acréscimo das marcas *SAbloqTimeout* e *SAbloqTimeoutHandler* ao estereótipo «*SAresource*»: estas marcas permitem, respectivamente, associar um tempo máximo de bloqueio ao solicitar um recurso compartilhado e também uma referência ao código de tratamento de exceção, acionado caso a espera demorar mais tempo do que fora especificado pelo *timeout*.
8. Estereótipo *«SAschedulable»* deveria ser derivado de *«CRconcurrent»*: semelhante à observação feita no item 4, aqui também se verifica uma semelhança semântica entre estes dois estereótipos.
9. Estereótipo *«SAaction»* deveria ser derivado de *«RTtimedAction»*: com o acréscimo desta relação, o primeiro poderia reaproveitar a marca *RTdeadline* e também possuiria uma referência ao código de tratamento de exceção por violação de *deadline*.

10. Caracterização das exceções por falhas temporais: observa-se que as exceções geradas por falhas temporais podem receber níveis de importância distintos, portanto elas deveriam ser caracterizadas quanto a este aspecto. Para tanto, poderia-se relacionar as mesmas com um estereótipo do tipo *«EXtiming»*, (derivado, por exemplo, de um *framework* para tratamento de exceções), contendo alguma marca cuja semântica permita o controle sobre a importância da exceção.

Outro ponto observado no perfil UML-TR é a carência de um estereótipo para representar requisitos "fim-a-fim", que relacione um conjunto de entidades de escalonamento dependentes entre si. Este carência foi reconhecida inclusive pelo grupo de desenvolvimento do perfil UML-TR, que deverá acrescentar nas próximas versões do mesmo um novo estereótipo denominado *«SAendToEnd»* para estes fins.

Imagina-se também a existência de uma outra situação, onde várias unidades de escalonamento executem de forma independente e possuam uma relação não-casual, i.e. a seqüência de processamento apresenta um caminho conhecido, porém é gerada por alguma interação complexa dentro do sistema, a qual pode depender de elementos externos (envolvendo trocas de mensagens). Contudo, sabe-se o tempo máximo no qual o conjunto de atividades deve estar finalizado. Verifica-se novamente a carência de um estereótipo para expressar esta informação, que atualmente pode ser expressa somente através de anotações não padronizadas (conseqüentemente não mapeáveis). No estudo de caso presente no capítulo 8 depara-se com uma situação envolvendo este tipo de complexidade, a qual é responsável pelo algoritmo de coordenação entre os veículos autônomos.

Para concluir esta seção são apresentados na TABELA 4.1 os estereótipos identificados como "mapeáveis" durante a análise do perfil UML-TR, ou seja, que podem ser representados em código usando somente as tecnologias estudadas. Defende-se portanto que modelos UML decorados com estes estereótipos podem ser automaticamente mapeados para uma implementação usando como alvo as tecnologias estudadas.

TABELA 4.1 – Elementos do perfil UML-TR mapeáveis para tecnologias estudadas

Elemento do perfil UML-TR	Descrição
«RTaction»	Estereótipo que delimita ação realizada por um bloco de código ou um método.
«RTtimedAction»	Estereótipo que delimita um bloco de código ou um método com tempo limite de execução (*deadline*).
~.RTdeadline	Marca utilizada para definir o tempo limite de execução da ação.
~.RTdeadMissHandler	Marca que contém referência para o código de tratamento de exceção por violação de *deadline*.
«RTdelay»	Estereótipo que caracteriza uma ação para bloquear por um tempo determinado a entidade que a chama.
«RTevent»	Estereótipo que caracteriza um evento que serve, por exemplo, para disparar uma ação.
~.RTat	Marca que contém informações sobre o padrão de ativação do evento.
«RTnewTimer»	Estereótipo que caracteriza a criação de um *timer*.

«RTtimer»	Estereótipo que define um objeto do tipo *timer*.
«CRaction»	Estereótipo derivado de *«RTevent»*, o qual indica que a ação pode executar de modo concorrente.
«CRasynch»	Estereótipo que caracteriza uma chamada assíncrona.
«CRsynch»	Estereótipo que caracteriza uma chamada síncrona.
~.CRcallTimeout	Marca que informa o tempo máximo de espera em uma chamada síncrona.
~.CRcallTimeoutMissHandler	Marca que contém referência para o código de tratamento de exceção por violação de *timeout*.
«CRconcurrent»	Estereótipo que caracteriza um objeto como concorrente.
«CRimmediate»	Estereótipo para caracterizar chamadas síncronas com atendimento imediato.
~.CRthreading	Quando esta marca for do tipo 'remote' a chamada executa na *thread* do objeto invocador; já quando for 'remote' executa numa *thread* local do objeto chamado.
«CRdeffered»	Estereótipo para caracterizar chamadas cujo instante atendimento é definido pelo objeto invocado.
«SAresource»	Este estereótipo caracteriza um recurso cujo acesso deve ser compartilhado por diversos recursos compartilhados, incluindo mecanismos de exclusão mútua.
~.SACapacity	Esta marca define o número máximo de *threads* que podem acessar simultaneamente um recurso compartilhado.
~.SAAccessControl	Marca que 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»	Este estereótipo define o método que implementa a requisição de acesso exclusivo a um recurso compartilhado.
~.GRMBlocking	Marca que define se a requisição de acesso exclusivo a um recurso compartilhado deverá bloquear até o recurso ser liberado ou não.
~.SAbloqTimeout	Marca que caracteriza o tempo máximo de bloqueio ao esperar por um recurso
~.SAbloqTimeoutMissHandler	Marca que contém referência para o código de tratamento de exceção por violação de *timeout*.
«GRMrelease»	Este estereótipo define o método que implementa a liberação de um recurso compartilhado.
«SAschedulable»	Estereótipo que caracteriza um objeto que constitui uma unidade de escalonamento.
«SAtrigger»	Estereótipo que caracteriza o padrão de ativação de uma "resposta", executada por uma unidade de escalonamento.
~.RTat	Marca que contém informações sobre o padrão de ativação da "resposta" e, conseqüêntemente, da sua entidade de escalonamento.
«SAresponse»	Estereótipo que caracteriza a "resposta" propriamente dita. Em conjunto com *«SAtrigger»* define o padrão de ativação e as propriedades de escalonamento de uma entidade de escalonamento.
~.SAAbsDeadline	Marca utilizada para caracterizar o tempo limite de execução de uma "resposta".

~.SAPriority	Marca utilizada para definir a prioridade de execução de uma "resposta".
~.RTduration	Atributo utilizado para caracterizar a duração de uma "resposta", ou seja, seu tempo efetivo de execução.
«SAaction»	Estereótipo que delimita um bloco de código ou um método.
~.SARelDeadline	Marca utilizada para definir o tempo limite de execução do bloco de código ou método.

Analisando os elementos selecionados para o mapeamento proposto, verifica-se que muitos deles são especializações de elementos definidos anteriormente. Na sua maioria, os estereótipos são derivados de elementos provenientes do *framework* para modelagem de tempo. Com isto, a proposta de mapeamento efetuada para o elemento mais genérico acaba por se repetir quando se trata do elemento especializado. Além disso, nota-se também que o mapeamento de alguns estereótipos tem sentido apenas se utilizados em conjunto com outros estereótipos. Como exemplo, cita-se a situação de criação de *timers*, que só tem sentido se a classe do objeto que implementa o método estereotipado com *«RTnewTimer»* for estereotipada com *«RTtimer»*. Assim criou-se a TABELA 4.2 aonde procurou-se identificar os estereótipos que se "sobrepõem" (i.e. cuja proposta de mapeamento encontra-se duplicada na TABELA 4.1) e também aqueles utilizados em conjuntos com outros estereótipos.

TABELA 4.2 – Resumo dos elementos mapeáveis para as tecnologias de programação OO

Identificador	Elemento do perfil UML-TR
1.1	*«RTaction»*
1.2	*«RTtimedAction»*
1.3	*«RTdelay»*
1.4	*«RTevent»*
1.5	*«RTnewTimer» + «RTtimer»*
(≈1.2)	*«CRaction»*
2.1	*«CRasynch»*
2.2	*«CRsynch»*
2.3	*«CRconcurrent»*
2.4	*«CRimmediate»*
2.5	*«CRdeffered»*
3.1	*«SAresource» + «GRMacquire» + «GRMrelease»*
(≈2.3)	*«SAschedulable»*
3.2	*«SAtrigger» + «SAresponse»*
(≈1.2)	*«SAaction»*

5 Mecanismo de Escalonamento para o TAFT

Este capítulo apresenta um novo mecanismo de escalonamento para o escalonador TAFT, cujo principal objetivo é manter um índice elevado de utilização de CPU além de garantir que deadlines são sempre cumpridos. Além disso, o mecanismo adotado permite tratar situações de sobrecarga transiente provendo códigos de tratamento de exceção, negociando funcionalidade por garantias de execução. A proposta de escalonamento em questão é utilizada como o mecanismo capaz de oferecer garantifas voltadas para o cumprimento dos requisitos temporais especificado durante a etapa de modelagem de uma aplicação tempo real, os quais foram discutidos no capítulo anterior. Para leitores interessados em uma visão geral sobre algoritmos de escalonamento, recomenda-se a leitura de [AUD 90; FAR 2000; LIU 2000; STA 98].

Na próxima seção aborda-se a política de escalonamento TAFT de uma maneira geral, apresentando os conceitos relacionados e também os resultados previamente obtidos, os quais serviram como base para a elaboração da proposta de escalonamento contida neste trabalho. Posteriormente, apresenta-se o esquema de escalonamento proposto, o qual constitui a principal contribuição desta tese no que se refere à execução de aplicações tempo real. Também descreve-se a implementação de um protótipo de ambiente de execução, o qual altera o escalonador de um SO-TR inserindo o esquema de escalonamento proposto,a fim de demonstrar que o modelo proposto é factível e também facilmente incorporado em ambientes reais. Além disso, são apresentados os resultados dos experimentos realizados para validar a estratégia proposta.

5.1 Política de Escalonamento TAFT

Na política de escalonamento TAFT (*Time-Aware Fault-Tolerant*) [STR 94; NET 97; NET 2001], cada tarefa é designada como um par, denominado *TaskPair* (TP), o qual é caracterizado por um único *deadline*. Este par de tarefas é constituído por uma parte principal (*MainPart* - MP) e uma parte de exceção (*ExceptionPart* - EP), conforme ilustrado na FIGURA 5.1. Estas partes são usadas para refletir o aspecto tolerante a falhas da política TAFT: enquanto a MP é sempre executada, pois contém a funcionalidade da aplicação propriamente dita, a EP tem que ser executada se e somente se a sua MP correspondente não puder ser finalizada antes do *deadline* do TP. Esta é a principal diferença entre a EP e a chamada "parte mandatória" da computação imprecisa [LIU 94], ou seja, enquanto a parte mandatória é sempre executada a EP executa somente quando a MP não terminar no prazo.

Assim, o TAFT deve garantir que a EP será concluída antes do deadline, ou seja, o intervalo de tempo entre o instante t_2 de ativação da EP e o instante t_3 determinado pelo deadline da tarefa deve ser maior que o tempo de execução de pior caso da EP. Diferentemente da MP, que contém o código responsável pelo progresso computacional da aplicação, a funcionalidade encontrada na EP deve ser mínima, servindo apenas para assegurar que:

- a aplicação controlada fique em um estado seguro;
- o sistema de controle esteja em um estado consistente.

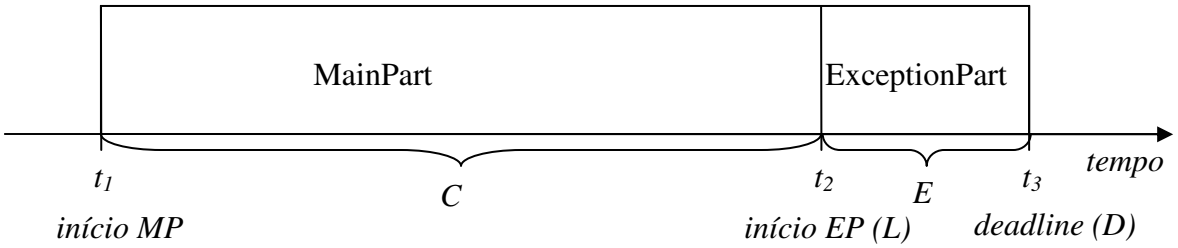


FIGURA 5.1 - Estrutura de um *TaskPair*

De uma maneira formal uma aplicação TAFT pode ser definida como sendo um conjunto $\prod$, constituído por τ_i tarefas independentes entre si e projetadas como TPs. Cada TP_i é uma tupla, composta por uma MP_i e uma EP_i, com deadline D_i e com tempo de ativação T_i. Este último pode ser interpretado como período nos TPs periódicos, ou como intervalo mínimo de chegada para TPs esporádicos. O tempo de execução E_i da EP_i é caracterizado como "tempo de execução de pior caso" (WCET), enquanto na MP_i o tempo de execução C_i é interpretado como "tempo esperado de execução" (*Expected Case Execution Time* - ECET). Logo, tem-se que:

C_i = ECET(MP_i) tempo de execução estimado da MP da tarefa *i*

E_i = WCET(EP_i) pior caso de execução da EP da tarefa *i*

Como na maioria dos casos ECETs são consideravelmente menores que WCETs, tem-se que normalmente a soma dos tempos C_i e E_i é muito menor que o tempo que seria obtido caso uma estimativa de pior caso fosse determinada para a MP (ou seja, $C_i + E_i << \text{WCET}(MP_i)$). Além disso, considerando que a complexidade de uma Epi tende a ser bem menor do que sua Mpi correspondente, tem-se ainda que o tempo médio de execução da Mpi tende a ser bastante superior ao pior caso de execução da EPi, ou seja, Ci >> Ei. Desta forma, mesmo estimativas bastante pessimistas de Ei não implicam em utilização de recursos comparáveis com a MP. Como conseqüência direta, consegue-se chegar a escalonamentos factíveis mesmo em casos onde um escalonamento tradicional baseado em WCETs não encontraria solução. Em suma, descreve-se uma aplicação TAFT composta por *n TaskPairs* como:

$$\prod = \{ \tau_i = (T_i, D_i, C_i, E_i), i = 1 \text{ até } n \}.$$

Do ponto de vista do escalonador, ambas as partes (MPs e EPs) devem ser tratadas como entidades de escalonamento distintas, cada uma com seus próprios parâmetros. O escalonador do TAFT deve sempre garantir que ou a MP ou a EP será finalizada antes do *deadline D*, refletindo assim o aspecto tolerante a falhas e a adequação do escalonador para aplicações tempo real críticas. Conforme já mencionado, uma EP deverá ser executada se e somente se a MP correspondente não terminar até o instante de tempo mais tardio em que a EP deve ser disparada sem violar *D*. Conseqüentemente, a execução de uma EP implica em abortar a MP respectiva.

Claramente pode-se perceber duas estratégias distintas para o escalonamento das MPs e EPs, o que leva a um escalonamento dito hierárquico: criar um escalonamento para as EPs implica em determinar o seu instante de ativação mais tardio (denotado por *L*), sendo que o instante *L* é usado como *deadline* para a respectiva MP. Em relação ao escalonamento das MPs, devem ser utilizadas estratégias que maximizem a utilização

do processador e que mantenham um alto nível de execuções bem-sucedidas. Assim, podem ser usados algoritmos de escalonamento com estratégia de melhor-esforço, a fim de otimizar o uso do sistema (porém sem oferecer garantias de execução), ou então algoritmos de escalonamento tempo real dinâmicos, que mantém uma alta utilização do processador com a vantagem de contar com garantias de execução.

5.1.1 Determinação dos Tempos de Execução das MainParts

Nos trabalhos de Gergeleit et al [GER 2001; NET 2001] é apresentado um componente de monitoração utilizado para coletar os tempos de execução dos TPs e gerar estimativas a respeito do "tempo de execução esperado" das MPs.

Neste trabalho, a cada MP pode ser associado um parâmetro *C*, o qual representa o tempo necessário para que um certo percentual de instâncias de MPs de um TP complete com sucesso, i.e. sem que seja necessário executar-se a sua EP. Quanto mais próximo este valor estiver do WCET, maior será o grau de sucesso da MP. Caso *C* seja igual ao WCET, tem-se 100% de sucesso, ou seja, todas as MPs terão tempo de execução < Ci e nenhuma EP é necessária.

O valor $C_{j,\alpha}$ é o tempo de CPU que deve ser atribuído à MP do TP τ_j de modo a obter uma probabilidade α que τ_j finalize sem executar sua exceção (ou seja, que MP_j finalize sua execução antes de L_j e que EP_j não necessite ser executado). O valor $C_{j,\alpha}$ pode ser derivado de uma função de distribuição de probabilidade (*Probability Distribution Function* – PDF). Considerando a PDF f_j para a MP_j do TP τ_j, tem-se que:

$$f_j : \{1,2,...,m\} \rightarrow [0,1], onde \sum_{x=1}^{m} f_j(x) = 1$$

onde o valor inteiro $m = WCET(MP_j)$ (dado que τ_j possui algum WCET) com a interpretação que $f_j(x)$ é a probabilidade a qual define que o tempo de execução da instância seja exatamente o tempo discreto *x*. Desta função, $C_{j,\alpha}$ pode ser computado por:

$$C_{j,\alpha} = Min\left(q \in \{1,2,...,m\} | \sum_{r=1}^{q} f(r) \geq \alpha \right)$$

Esta função é ilustrada na FIGURA 5.2. Os pontos denotam os valores da PDF f_j, e as áreas sombreadas representam as somas das probabilidades até aquele tempo de execução (i.e. a distribuição de probabilidade). Esta soma excede a probabilidade α para o C requisitado no eixo do tempo de execução (a área sombreada escura). Para ser mais preciso, α é o quantil (ou separatriz) da PDF.

A proposta de Gergeleit também defende que ao assinalar-se diferentes valores de α para diferentes tarefas é possível expressar uma noção de importância para cada tarefa. Segundo a estratégia de escalonamento prevista, o $C_{j,\alpha}$ de uma tarefa deve ser associado com a sua importância, de forma que quanto maior for o valor de α maior é a importância relatia da tarefa.

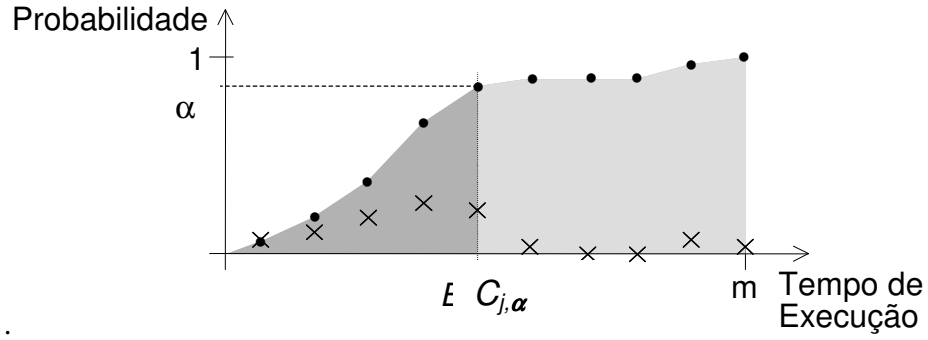


FIGURA 5.2 – Curva da PDF para estimação do parâmetro C

Na próxima seção apresenta-se o mecanismo de escalonamento proposto nesta tese para suportar a política de escalonamento tempo real tolerante a falhas do TAFT.

5.2 Mecanismo de Escalonamento Proposto

A fim de implementar os princípios de escalonamento do TAFT detalhados na seção anterior propôs-se uma estratégia de escalonamento de dois níveis[16], onde o primeiro nível escalona as EPs e o segundo as MPs. De acordo com esta proposta, a saída produzida pelo primeiro nível sempre terá uma maior prioridade do que a saída produzida pelo segundo nível, uma vez que mesmo em situação de pior caso, isto é, quando nenhuma MP consegue completar antes de seu deadline, todas as EPs precisam ser executadas. A razão principal para se usar duas estratégias de escalonamento se dá pelo fato de que MPs e EPs são vistas como entidades distintas, com diferentes critérios de escalonabilidade. Em um lado se tem a MP, que deve terminar antes do instante de ativação da sua respectiva EP, ou seja, o instante máximo para ativação da EP torna-se o *deadline* para a MP. Não são feitas restrições relativas ao seu instante de ativação. Por outro lado, as EPs possuem uma restrição severa em relação ao seu instante de ativação L_i que, conforme já mencionado, deve ocorrer o mais tarde possível, para maximizar o tempo disponível para as MPs, porém cedo o suficiente para garantir sua finalização antes do deadline D_i. A formalização da estratégia proposta é feita na definição 1.

Definição 1: O escalonador proposto é um escalonador de dois níveis, que funciona para um conjunto tarefas periódicas $\prod = \{ \tau_i = (T_i, D_i, C_i, E_i), i = 1 \text{ até } n\}$ projetadas como TPs, sendo que o *deadline* D_i é igual ao período T_i, o tempo de execução C_i da MP é caracterizado como ECET e o tempo de execução E_i da EP é caracterizado como WCET.

Primeiramente destaca-se o segundo nível do escalonamento, o qual é encarregado de escalonar as MPs. Neste nível é usado o algoritmo EDF, principalmente devido a sua optimalidade: se existir um algoritmo capaz de gerar um escalonamento onde todas as tarefas cumprem os seus *deadlines*, então o EDF também irá gerar um escalonamento factível. Adicionalmente, o EDF maximiza a utilização da CPU devido à sua característica de escalonar as tarefas o mais cedo possível. Portanto, assegura-se com

[16] De acordo com [Liu00], uma estratégia *n*-níveis indica o uso em conjunto de mais de uma política de escalonamento.

isto um maior número possível de MPs sendo executadas. Além disso, o uso do EDF se enquadra de maneira adequada na estratégia de escalonamento usada para as EPs, permitindo o desenvolvimento de um teste de aceitação único para os TPs, conforme detalhado na próxima seção.

A equação 1 [LIU 73] apresenta a condição necessária e suficiente para que o escalonador de segundo nível garanta a execução das MPs. No entanto, esta esquação funciona corretamente para valores de C baseados em WCET. Portanto, verifica-se a necessidade de adequá-la à proposta do TAFT, que caracteriza as MPs com tempo de execução C baseado em ECET.

$$\sum_{j=1}^{n} \frac{C_j}{T_j} \leq 1 \qquad (1)$$

A proposta inicial do TAFT propunha escalonar as EPs utilizando um escalonamento baseado em um calendário fixo (*calendar-based scheduling*) [STR 94], o qual armazena previamente todos os instantes de ativação das EPs. Entretanto, a implementação desta estratégia de escalonamento apresenta diversos problemas. O mais óbvio é que este tipo de implementação é dispendioso em termos de consumo de CPU para modificar e manter dinamicamente. Este fato tem conseqüência direta no teste de aceitação das tarefas, visto que garantir a execução de tarefas nesta abordagem implica em encontrar lacunas de tempo disponível no calendário ou, em outras palavras, computar todo o escalonamento.

A fim de superar os problemas mencionados acima, considerou-se neste trabalho o uso do algoritmo *Latest Release Time* (LRT) [LIU 2000], ou *Earliest Deadline as Late as possible* (EDL) [CHE 89], para escalonar as EPs, representando o primeiro nível de escalonamento. Basicamente, este algoritmo pode ser interpretado como um *Earliest Deadline First* (EDF) reverso, tratando tempo de ativação como *deadline*, e *deadline* como tempo de ativação. As EPs são escalonadas de trás para frente, iniciando a partir do superperíodo P do conjunto de TPs (mínimo múltiplo comum - MMC - entre todos os períodos) e se retrocedendo até o instante t_b, onde todas as EPs encontram-se escalonadas. Esta propriedade, denominada "ciclicidade", impõe o uso de um conjunto de TPs periódicos, onde o escalonamento gerado tem a mesma configuração no tempo $t \geq 0$ que ele terá no tempo $t + kP$ ($k = 1, 2, \ldots$). Desta forma, encontrar o escalonamento gerado pelo LRT em um tempo infinito se reduz a encontrar o escalonamento nos intervalos $[kP, (k + 1)P]$, $k = 0, 1, 2, \ldots$, sendo cada um deles denominado de janela. O LRT é provado ser ótimo (vide [CHE 89]) dentro das mesmas condições que o EDF é ótimo: é possível escalonar um conjunto de EPs periódicas independentes entre si, preemptivas e com *deadline* igual ao período, sempre que o fator de utilização do conjunto for menor ou igual a 1. Portanto, o escalonamento das EPs é "pessimista" no sentido de que assume a condição de pior caso, em que todas as EPs precisam ser executadas além das MPs. Este critério é fundamental para a elaboração do teste de aceitação discutido na próxima seção.

A equação 2 [CHE 89] apresenta a condição necessária e suficiente para que o escalonador de primeiro nível garanta a execução das EPs.

$$\sum_{j=1}^{n} \frac{E_j}{T_j} \leq 1 \qquad (2)$$

O planejamento inicial do algoritmo LRT é feito off-line para não causar sobrecarga no sistema, conforme destacado por Chetto [CHE 89]. Iniciando em *P*, quando todas as tarefas têm o mesmo *deadline*, um escalonamento baseado em prioridades é montado para a primeira janela de tempo ([0, P]), onde as prioridades são definidas de acordo com o instante de ativação da tarefa: quanto mais tarde for o instante de ativação, maior é a prioridade. Para aquelas tarefas com múltiplas ocorrências em *P*, todas as suas entradas são consideradas quando o escalonamento for gerado. Também cabe destacar a eficiência deste algoritmo, que possui uma complexidade de ordem O^n.

O pseudo-código do algoritmo contendo o mecanismo de escalonamento proposto é apresentado na FIGURA 5.3. O funcionamento deste algoritmo pode ser descrito como segue. Inicialmente a fila de EPs é verificada a fim de buscar a próxima EP a ser executada (linha 2). Caso exista alguma EP programada, verifica-se então se a MP correspondente terminou ou não a execução (linha 5). Caso a MP já tenha terminado, então a EP lida da fila é descartada da fila de EPs (linhas 11 e 12), caso contrario a mesma é definida como a próxima tarefa a ser executada (linhas 5 a 7) e a fila de EPs é atualizada. Caso não existir EP pronta para execução, o escalonador seleciona a próxima MP a ser executada (linhas 13 e 14). Finalmente, o escalonador faz o chaveamento de contexto, preemptando o TP atual na CPU e ativando a MP ou a EP selecionada (linha 15).

```
//Procedimento escalona TaskPairs
Inicio
1:  novaMP = nulo
2:  proximaEP = retorna próxima EP da fila-LRT
3:  Se (proximaEP não é nulo) então
      Inicio
4:      Se (tempo atual >= tempo de inicio da próximaEP) então
           Inicio
5:           Se (MP correspondente à proximaEP não terminou) então
               Inicio
6:                novaMP = MP da proximaEP
7:                Se (status da novaMP == SAVE) então
8:                                    status da novaMP = DEADL;
9:                atualiza a fila-LRT
.              Fim
10:          Senão
               Inicio
11:               novaMP = nulo;
12:               remove proximaEP da fila-LRT
               Fim
             Fim_se
           Fim_se
13: Se (novaMP não é nulo){
14:        novaMP = retorna próxima EP da fila-EDF
15: faz preempção para novaMP
Fim
```

FIGURA 5.3 – Algoritmo do mecanismo de escalonamento proposto para o TAFT

5.2.1 Teste de Aceitação

Nas equações (1) e (2) foram apresentadas as condições necessárias e suficientes para o primeiro e o segundo nível de escalonamento. Todavia, estas equações partem do princípio que os algoritmos operam isoladamente, sendo que a solução de escalonamento para os TPs apresentada na seção anterior requer um teste de aceitação, ou teste de escalonabilidade, que funcione para ambos algoritmos operando em conjunto. O objetivo é definir-se um teste de aceitação que garante que: (i) caso todas as MPs executem dentro de seu tempo de execução estimado, então o conjunto é escalonável; (ii) em caso de sobrecarga, nos quais a MP necessita um tempo maior do que o estimado, deve-se garantir que a EP correspondente consiga executar e finalizar antes do deadline.

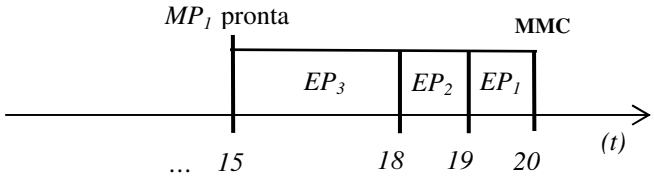
Inicialmente, assume-se um cenário otimista (i), onde todas as MPs executam sem ultrapassar o tempo de execução estimado C. Neste caso, tem-se que a condição de escalonabilidade do EDF, apresentada na equação (1), é condição necessária e suficiente.

Entretanto, diferentemente do EDF, que trabalha sempre com tempos de execução de pior caso, na solução de escalonamento adotada é possível que algumas (senão todas) instâncias de MPs ultrapassem os tempos de execução esperados. Nestes casos, o algoritmo deve garantir que as EPs correspondentes venham a ser executadas. Considera-se inicialmente o agrupamento dos testes necessários e suficientes apresentados nas equações (1) e (2) para a solução deste problema, conforme exibido na equação (3).

$$\sum_{j=1}^{n} \frac{C_j}{T_j} + \sum_{j=1}^{n} \frac{E_j}{T_j} \leq 1 \Rightarrow \sum_{j=1}^{n} \frac{C_j + E_j}{T_j} \leq 1 \qquad (3)$$

Destaca-se que o teste apresentado considera que as MPs e EPs de todas as tarefas são executadas, o que não ocorre na prática, uma vez que quando uma MP finaliza sua execução antes do início previsto de sua EP, a EP correspondente não precisa ser executada. Além disso, verifica-se que a equação (3) é uma condição necessária porém não suficiente para o escalonamento dos TPs. A mesma é necessária porque segue o principio básico que o processador não consegue escalonar um conjunto de tarefas quando estas ultrapassarem 100% de ocupação. Este é o caso quanto todas MPs ultrapassarem os seus ECET, causando uma conseqüente ativação de todas EPs.

Demonstra-se a insuficiência da equação (3) através de um exemplo, o qual é ilustrado na FIGURA 5.4. Neste exemplo é exibido o escalonamento das EPs a partir do MMC do conjunto de tarefas (t = 20), cujas características são mostradas na figura. A coluna U_i indica a utilização de cada um dos TPs, onde cada linha representa um dos somatórios calculados através da equação (3), sendo que a utilização total é de 0,8. Portanto, pelo resultado da equação (3), este conjunto de TPs deveria ser aceito para execução. Entretanto, observa-se que no instante t = 15 a MP_1 está pronta para executar, porém não existe tempo disponível até o instante t = 19, quando a sua EP é ativada. Logo, a MP_1 dispõe de um tempo de CPU menor que o seu tempo estimado de execução C1 e não consegue completar antes do inicio da sua exceção. Com isto, prova-se através deste exemplo que a condição apresentada na equação (3) não é suficiente para determinar se um conjunto de TPs é escalonável.



TP	T_i	C_i	E_i	U_i
1	5	1	1	0,4
2	10	1	1	0,2
3	20	1	3	0,2
			Total:	0,8

FIGURA 5.4 – Exemplo de escalonamento usando o TAFT

Analisando a FIGURA 5.4 verifica-se que a falta de tempo para execução da MP_1 ocorre justamente dentro do seu período crítico (*crictical period – CP*) [CHE 89], também chamado de instante crítico por Liu e Layand [LIU73]. Um TP se encontra no seu CP quando o mesmo estiver na sua última ativação dentro do superperíodo *P*, pois este é o instante onde todas as EPs estão prontas para executar (e uma vez que o escalonamento de EPs é prioritário com relação ao escalonamento de MPs, esta é a pior situação para execução de uma MP). Com base nesta observação definiu-se um novo teste de aceitação para o TAFT. O objetivo deste teste é garantir aos TPs aceitos um percentual mínimo α de sucesso das MPs, juntamente com a garantia de execução das EPs daquelas MPs que não finalizarem a sua execução dentro do tempo de execução estimado.

Contudo, deve ficar claro que o fato de uma MP exceder o seu tempo de execução C_i não determina a execução imediata da EP correspondente. Tal execução só deve ocorrer no momento em que o escalonador detectar que a MP atingiu o instante *L*, i.e. o último instante possível para ativação da EP de modo que a mesma complete antes do deadline *D*.

O teste proposto é baseado no cálculo de demanda apresentado por Klien et al [KLI 93]. O mesmo consiste em calcular o chamado fator máximo de utilização (*maximum utilization factor* - MUF) Ω para cada TP, sempre que uma nova tarefa, ou melhor, TP, desejar ingressar no sistema. O MUF representa um índice relativo de utilização do processador, o qual é definido como a utilização esperada de CPU quando o TP estiver executando dentro do seu CP. Este cálculo leva em consideração as possíveis múltiplas ocorrências das tarefas de menor período dentro do CP das tarefas de maior período, sendo que o número de ocorrências pode ser calculado pelo teto da divisão entre o maior e o menor período. O cálculo do MUF é apresentado na definição 2.

Definição 2: Dado o conjunto de TPs $\prod = \{ \tau_i = (T_i, D_i, C_i, E_i)$, $i = 1$ até n, $\forall i$: $(T_i \leq T_{i+1}$ e $D_i = T_i\}$, ordenado em ordem crescente de *deadline*, o qual é igual ao período, define-se o MUF Ω_i para o cada TP τ_i como sendo a soma de todas as EPs mais a soma da MP_i pelas MPs de maior prioridade[17], considerando as suas múltiplas ocorrências dentro do CP do TP *i*, conforme exibido abaixo:

$$\Omega_i = \sum_{j=1}^{n} \left(\left\lceil \frac{T_i}{T_j} \right\rceil E_j \right) + \sum_{j=1}^{i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil C_j \right) \leq T_i \qquad (4)$$

[17] Assume-se como MPs de maior prioridade àquelas MPs cuja ocorrência no conjunto ordenado de TPs $\prod$ aparecem antes do índice *i*.

Teorema 1: Dado a definição 2, se $\forall \tau_i \in \prod \Rightarrow \Omega_i \leq T_i$, i.e. se o MUF do TP *i* for menor do que o seu período, então existe uma condição suficiente para que o conjunto de TPs $\prod$, com períodos harmônicos, seja escalonável através do escalonador proposto para o TAFT (nível 1: escalona EPs com LRT; nível 2: escalona MPs com EDF).

Prova: Assumindo que o conjunto $\prod$ possui somente TPs com períodos harmônicos entre si, é possível estabelecer as seguintes relações:

1) $\forall i,j \,/\, j \leq i \Rightarrow T_j \leq T_i$ e Ti = $k*Tj$, onde $k \in$ N $\Rightarrow \left\lceil \frac{T_i}{T_j} \right\rceil = \frac{T_i}{T_j} = k$, e

2) $\forall i,j \,/\, j > i \Rightarrow T_j \geq T_i \Rightarrow T_i/T_j \leq 1 \Rightarrow \left\lceil \frac{T_i}{T_j} \right\rceil = 1$,

pode-se então reescrever a equação (4) como:

$$\Omega_i = \left[\sum_{j=1}^{i} \left(\frac{T_i}{T_j} E_j \right) + \sum_{j=i+1}^{n} E_j + \sum_{j=1}^{i} \left(\frac{T_i}{T_j} C_j \right) \right] / T_i \leq 1 \qquad (5.1)$$

$$\Omega_i = \sum_{j=1}^{i} \frac{E_j}{T_j} + \frac{1}{T_i} \sum_{j=i+1}^{n} E_j + \sum_{j=1}^{i} \frac{C_j}{T_j} \leq 1 \qquad (5.2)$$

$$\Omega_i = \sum_{j=1}^{i} \frac{C_j + E_j}{T_j} + \frac{1}{T_i} \sum_{j=i+1}^{n} E_j \leq 1 \qquad (5.3)$$

Um conjunto de tarefas harmônicas se constitui na pior das configurações possíveis para o período crítico, pois todas as MPs estão prontas para executar no instante $P - T_n$ (quando inicia o CP da tarefa *n* de maior período) e todas as EPs estão prontas para executar no instante *P*. Dado que todos TPs passíveis de sofrerem distúrbios encontram-se prontos para execução no período crítico, é possível assumir sem perda de generalidade que se um TP consegue ser escalonado neste momento, então ele poderá ser escalonado em qualquer outro período. A prova da equação (5.3) é feita por indução:

1) Para o TP τ_1 de menor período, tem-se que:

$$\Omega_1 = \frac{C_1 + E_1}{T_1} + \frac{1}{T_1} \sum_{j=2}^{n} E_j \leq 1$$

No intervalo [$P - T_1$, P] todas EPs estão prontas para executar, sendo que as mesmas têm uma maior prioridade de execução que qualquer MP. Como o TP τ_1 terá o tempo de ativação mais tardio (*latest release time*), o LRT irá escalonar, a partir de *P*, primeiramente EP_1 e após todas as outras EPs (em ordem inversamente proporcional ao valor do seu período). Uma vez que MP_1 é escalonada pelo algoritmo EDF, a mesma encontra-se pronta para executar no instante $P - T_1$, e consegue atender o seu ECET se existir tempo de CPU suficiente no intervalo [$P - T_1$, $P - \sum_{j=1}^{n} E_j$]. Tal situação procede

se $\Omega_1 \leq 1$, isto é, a utilização de CPU considerando todas tarefas existentes não é maior que 1. Visto que todos TPs tem o mesmo *deadline* no período crítico, MP_1 não possui necessariamente a maior prioridade de execução, porém neste caso o EDF permite uma reordenação arbitrária.

2) Dado que para $i > 1$ o teorema permanece válido para τ_j, $1 \leq j \leq i-1$, mostra-se que ele também é válido para τ_i:

$$\Omega_i = \sum_{j=1}^{i} \frac{C_j + E_j}{T_j} + \frac{1}{T_i} \sum_{j=i+1}^{n} E_j \leq \sum_{j=1}^{n} \frac{E_j}{T_j} + \sum_{j=1}^{i} \frac{C_j}{T_j} << \sum_{j=1}^{n} \frac{E_j}{T_j} \leq 1$$

Visto que as MPs não conseguem alterar o escalonamento das EPs devido às suas menores prioridades, a fórmula acima indica por prova do LRT [CHE 89] que todas as EPs podem ser executadas se o seu fator de utilização não for maior que 1. Novamente, como MP_i é escalonada via EDF, ela encontra-se pronta para execução em $P - T_i$, conseguindo atender o seu ECET se existir tempo de CPU suficiente no intervalo [$P - T_i$,
$P - \sum_{j=1}^{n} \left\lceil \frac{T_i}{T_j} \right\rceil E_j + \sum_{j=1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j$]. Isto é válido se $\Omega_i \leq 1$, isto é, a utilização de CPU causada por todas as EPs e MPs em execução no CP do TP τ_i não é maior que 1.

■

Teorema 2: Dado a definição 2, se $\forall \tau_i \in \prod \Rightarrow \Omega_i \leq T_i$, então existe uma condição suficiente para que o conjunto de TPs $\prod$, com períodos não-harmônicos, seja escalonável através do escalonador proposto para o TAFT (nível 1: escalona EPs com LRT; nível 2: escalona MPs com EDF).

Prova: Conforme mencionado anteriormente, um conjunto de tarefas harmônicas assume a pior configuração possível dentro do período crítico. Dado um conjunto de tarefas não-harmônicas, percebe-se que o mesmo pode ser trabalhado como se fosse harmônico, como faz justamente a equação (4) ao usar o teto da divisão entre os períodos dos TPs para calcular o número de ocorrências dos TPs com menor período naqueles de maior período. Considere as definições a seguir.

$$\forall i,j \,/\, j \leq i \Rightarrow T_j \leq T_i \leq k.T_j \Rightarrow 1 \leq \frac{T_i}{T_j} \leq k$$

$$\frac{T_i}{T_j} = \left\lfloor \frac{k.T_j}{T_j} \right\rfloor = \left\lceil \frac{k.T_j}{T_j} \right\rceil = k \text{ , onde } k \in \mathrm{N}$$

logo, reescrevendo a equação (4) percebe-se que:

$$\Omega_i = \left[\sum_{j=1}^{n} \left(\frac{k.T_i}{T_j} E_j \right) + \sum_{j=1}^{i} \left(\frac{k.T_i}{T_j} C_j \right) \right] / T_i \leq 1 \qquad (6.1)$$

$$\Omega_i = \sum_{j=1}^{i} \frac{k.E_j}{T_j} + \frac{1}{T_i} \sum_{j=i+1}^{n} E_j + \sum_{j=1}^{i} \frac{k.C_j}{T_j} \leq 1 \qquad (6.2)$$

$$\Omega_i = \sum_{j=1}^{i} \frac{k.(C_j + E_j)}{T_j} + \frac{1}{T_i} \sum_{j=1}^{n} E_j \leq 1 \qquad (6.3)$$

A prova da suficiência da equação pode ser feita da mesma forma como fora efetuado anteriormente para o conjunto de TPs harmônicos. Neste momento prova-se através de um contra-exemplo que esta condição não é necessária para o escalonamento de um conjunto de TPs não-harmônicos, conforme ilustrado na FIGURA 5.5. Neste exemplo, o resultado do teste de aceitação indica que o conjunto de tarefas não é escalonável, embora exista CPU ociosa (vide instante t = 5 na figura). Isto se deve ao fato de transformar-se o conjunto não-harmônico em um conjunto harmônico, que representa uma situação de carga maior do que a carga real imposta pelo conjunto não-harmônico.

	M1	M2	E1	M1	--	E1	E2	M1	E1	M1	M2	E1	M1	E2	E1	M1	M2	E1	M1	E2	E1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

TP	T_i	C_i	E_i	U_i
1	3	1	1	0,67
2	7	1	1	0,29
			Total:	0,96

$\Omega_1 = 1+1+1 = 3 \leq 3$

$\Omega_2 = 3*1+1+3*1+1 = 8 \leq 7$ (falso)

FIGURA 5.5 – Escalonamento de um conjunto de TPs não-harmônicos

5.2.2 Escalonamento Adaptativo para as MPs

Conforme descrito até o presente momento, dois algoritmos de escalonamento - LRT e EDF - coexistem no esquema proposto para escalonar os TPs. O algoritmo LRT escalona as EPs num primeiro nível, enquanto o EDF trabalha num segundo nível escalonando as MPs. As prioridades geradas como saídas destes escalonamentos encontram-se divididas em duas classes distintas, de modo similar ao proposto em [DAV 95]. Isto significa afirmar que uma vez existindo alguma EP pronta para executar, esta terá prioridade de execução sobre qualquer MP. Logo, MPs serão executadas se e somente se não existirem EPs prontas para a execução.

Entretanto, com o esquema de escalonamento e despacho descrito, perdem-se as garantias oferecidas pelo teste de aceitação caso alguma MP exceda o seu tempo de execução (C_i). Um mecanismo de despacho baseado puramente nas prioridades do EDF pode resultar no conhecido "Efeito Dominó[18]", onde todas tarefas subseqüêntes sofrem pelo fato de uma determinada tarefa ter perdido o seu *deadline*.

[18] Chama-se "Efeito Dominó" o fato de uma tarefa que não cumpre o *deadline* continuar executando e "roubar" o tempo de CPU destinado a outras tarefas, também levando as mesmas à perda de seus *deadlines*.

Para previnir os TPs do indesejável “Efeito Dominó”, alterou-se o mecanismo de despacho implementado para acrescentar um mecanismo de adaptação. Através deste mecanismo, quando uma MP exceder o seu ECET sua prioridade é decrescida para um nível inferior às prioridades assinaladas pelo EDF àquelas MPs que não violaram o seu ECET. Na verdade, o qualificador do TP passa do estado “garantido” para o estado “melhor-esforço”. A MP pode continuar usando a CPU, a fim tentar completar sua execução antes do início da EP correspondente, se e somente se não existirem MPs “garantidas” prontas para executar. Novamente, enfatiza-se que a execução das EPs independe deste mecanismo, estando a execução das mesmas garantidas devido a maior prioridade do escalonador de primeiro nível (LRT).

5.3 Validação

A fim de validar o mecanismo de escalonamento desenvolvido incorporou-se a proposta de escalonamento TAFT ao sistema operacional tempo real RTLinux [BAR 97], executando em um microcomputador com processador Pentium. A escolha do RTLinux como plataforma alvo se deu pelo fato do mesmo possuir um mecanismo de despacho bem estruturado e estritamente baseado em prioridades, como na maioria dos SO tempo real disponíveis, e também devido ao seu código fonte aberto, o que permite a realização de modificações direto no código do escalonador.

A implementação efetuada decorreu em algumas modificações nas estruturas de dados originais do RTLinux, dentre as quais encontram-se:

1. Alterações na estrutura das *threads* tempo real (*struct rtl_thread_struct*) para acrescentar informações referentes ao “instante de ativação da EP” (L_i), “*deadline* do TP” (D_i), “tempo esperado de execução da MP” (C_i), “pior caso de tempo de execução da EP” (E_i) e “tempo decorrido de uso efetivo da CPU”.
2. Alterações na estrutura que mantém os dados do escalonador (*struct rtl_sched_cpu_struct*) para manter informações referentes ao índice de utilização gerado pelo conjunto de TPs que passaram pelo teste de aceitação (informação proveniente do próprio teste de aceitação).

Não obstante, as principais alterações realizadas dizem respeito à função de escalonamento do RTLinux (procedimento *rtl_schedule()*). A implementação efetuada modifica o escalonador original para suportar o mecanismo de escalonamento descrito na seção anterior e cujo algoritmo é apresentado na FIGURA 5.3. Além disso, acrescentaram-se funcionalidades que permitem controlar o tempo de execução efetivo (i.e. de uso de CPU) de cada TP, a fim de prover suporte para o mecanismo de decréscimo de prioridade daquelas MPs que excedem o seu tempo de execução C_i. Adicionalmdente, foi necessário implementar uma estratégia para suportar o cancelamento das MPs que violam o *deadline* L_i e o respectivo chaveamento de contexto da aplicação para o código da EP correspondente.

Para garantir que o escalonador modificado do RTLinux trabalhe com uma boa precisão, manteve-se a estratégia original de utilização de um *timer* interno. Este estará sempre programado para invocar o escalonador no instante em que ocorrer o próximo evento de interesse. Como exemplos de eventos, citam-se o instante de tempo que caracteriza o período na próxima MP que virá a executar, o instante que caracteriza a expiração do tempo esperado de execução C_i e o instante L_i em que a MP deve ser abortada para ativar a EP. É importante ressaltar que está previsto um intervalo de

tempo mínimo entre ativações do escalonador, que caracteriza o chamado *tick*. Na implementação atual, o valor do *tick* é de 5*ms*. Desta forma, garante-se que o maior atraso (*jitter*) previsto para o atendimento a um determinado evento não será superior a este valor.

Outra funcionalidade importante, porém que não pertence ao escalonador propriamente dito, é o teste de aceitação dos TPs que ingressam no sistema. Esta funcionalidade foi implementada no escopo da função *static int add_to_task_list()*, que acumula a utilização calculada para o TP que está ingressando no sistema, caso haja CPU disponível. Caso contrário, é retornado um código de erro informando que o TP não fora aceito para execução.

Um dos pontos mais trabalhosos na implementação da política TAFT diz respeito à estratégia para cancelar a MP e ativar a EP assim que for detectada a violação do *deadline* L_i. Para tanto, dois pontos essenciais tiveram que ser trabalhados:

1. Detecção do instante exato em que a MP precisa ser abortada, diferenciando-o claramente dos instantes de preempção, sendo que neste momento a tarefa precisa ser colocada em um estado que permita o seu recomeço;
2. Eficiência do mecanismo de tratamento de exceção, o qual se procurou otimizar implementando-o no mesmo contexto (i.e. mesma *thread*) em que executa a parte principal. Com isto, o estado da parte abortada permanece disponível para o tratador de exceção, sendo portando desnecessário manter informações a respeito de duas *threads* distintas ao invés de somente uma.

Com vista nos requisitos expostos, a solução mais intuitiva encontrada para solucionar esta questão foi a utilização de um mecanismo do tipo *setjump()/longjump()*, proveniente do Unix/C. Todavia, não é possível utilizar estas funções diretamente, uma vez que elas atuam no contexto da aplicação e não do escalonador, como requer o presente caso. Assim, tornou-se necessário fazer uma adaptação deste mecanismo para o cenário atual.

A solução encontrada requer a criação um estado inicial para thread que executa o código do TP. Isto implica no salvamento dos registradores de hardware, do contador de programa (*programm counter*) – utilizado para indicar a próxima instrução a ser executada - e também outras informações relevantes, como as variáveis alocadas na pilha da função que contém o corpo do TP. Isto é feito antes de iniciar a execução da MP, que representa um bloco de código dentro desta mesma função. Em caso de violação de *deadline* durante a execução da MP, a thread é abortada, as variáveis que indicam o status do TP são atualizadas pelo escalonador e o contador de programas é recolocado no estado inicial, sendo que este será o ponto de partida para a EP a ser executada. Em contraste com o mecanismo proposto no Unix/C, a operação *longjump*() é na verdade invocada pelo escalonador, e não pela *thread* em execução propriamente dita. Já o mecanismo *setjump*() é implementado na própria *thread*, salvando-se o ponteiro da pilha (*stack pointer*) de modo que o *longjump*() possa recuperá-lo e reconstituir o estado da pilha. Por questões de implementação, o processo de recuperação é sempre realizado no contexto de uma outra *thread*, pois é muito difícil manipular a pilha de uma *thread* estando dentro do seu próprio contexto.

Para implementar este mecanismo foi necessário inserir outras quatro variáveis na estrutura das *threads* tempo real, sendo três delas usadas para manipular a pilha e a quarta utilizada para armazenar um dos quatro possíveis estados de execução do TP, os quais são exibidos abaixo:

1. NONE: indica que o TP fora criado e não salvou o ponteiro da pilha;
2. SAVE: indica que o TP encontra-se em um estado "seguro", i.e. já salvou o estado da pilha e não está executando nem a MP nem a EP;
3. MAINP: indica que o TP está pronto para executar a MP;
4. EXCEPTP: indica que a condição de exceção foi alcançada.

Com a intenção de facilitar a programação deste mecanismo, criou-se um conjunto de macros, o qual é descrito na TABELA 5.1. Estas macros são utilizadas durante a programação dos TPs, onde o programador deve especificar explicitamente as partes constituintes do TP enquanto programa o código das tarefas. Para tanto, deve ser seguido um estilo de programação com uma semântica para tratamento de exceções semelhante ao pseudo-código da FIGURA 5.6, o qual representa o código de um TP periódico típico. A função *guarantee()* é responsável por realizar o teste de aceitação descrito na seção anterior, que visa determinar se o escalonador consegue ou não atender a estratégia de escalonamento proposta, com base nos parâmetros informados.

TABELA 5.1 - Macros para programação dos *TaskPairs*

Macro	Código

TP_SAFE_REGION	`(pthread_self())->save_stack_size = 0; \`
	`(pthread_self())->tp_sp =&dummy; \`
	`(pthread_self())->tp_status = SAVE; \`
	`(pthread_self())->mp_exec = FIRST_EXEC; \`
	`(sched_yield();`

BEGIN_MAIN_PART	`if(pthread_self()->tp_status == MAINP) {`
END_MAIN_PART	`(pthread_self()->sched_param.sched_priority =` `priority_MP; }`
BEGIN_EXCEPTION_PART	`else if(pthread_self()->tp_status == EXCEPTP) {`

END_EXCEPTION_PART	`(pthread_self()->tp_status = MAINP; \`
	`(pthread_self())->sched_param.sched_priority =`
	`priority_MP;`
	`}`

```
struct taskpair{
    hrtime_t period;     //T_i
    hrtime_t tp_deadl;   //D_i
    hrtime_t mp_ecet;    //C_i
    hrtime_t ep_wcet;    //E_i
};

void Procedure_TP (struct taskpair TP){
    if(guarantee (TP)){
    while (true) {
           pthread_wait_np_ex();

           TP_SAFE_REGION

           BEGIN_MAIN_PART
                  MP();
           END_MAIN_PART

           BEGIN_EXCEPTION_PART
                  EP();
           END_EXCEPTION_PART
           }
    }
```

FIGURA 5.6 - Código para programação de um *TaskPair*

5.3.1 Suporte para Programação no TAFT

Na FIGURA 5.6 ilustrou-se uma estrutura genérica que reflete a maneira com um TP é programado. Contudo, existem alguns procedimentos a serem seguidos para a programação dos TPs no RTLinux, os quais fazem chamadas a funções específicas da API de programação deste último. A FIGURA 5.7 exibe um trecho de código completo, o qual representa a maneira como *threads* tempo real (que representam os TPs neste caso) são implementadas no RTLinux. Este código é executado de acordo com os seguintes passos:

1. Ao ser carregado, o módulo da aplicação[19] inicia a execução da função *init_module()*, encarregada de criar a(s) *thread*(s) no sistema através da chamada à função *pthread_create(...)*; esta última faz chamada à função *test_tp(...)*, que executa o teste de aceitação para o TP, retornando *true* em caso de sucesso e *false* caso contrário.
2. O código da *thread* encontra-se descrito em uma função separada no sistema (função *start_thr(...)* neste exemplo), cujo endereço é passado como parâmetro para a função que cria a *thread* no sistema. Ao ser criada, a *thread* precisa inicializar seus parâmetros de ativação, como prioridade e período (no caso de threads periódicas), o que é feito respectivamente através das funções *pthread_setschedparam(...)* e *pthread_make_periodic_np(...)*. Logo após, entra-se no laço que controla a execução periódica da *thread* (ou TP neste caso), conforme fora abordado na FIGURA 5.6.
3. Ao ser removido do sistema, o módulo da aplicação executa a função *cleanup_module()*. Nesta última, encontram-se chamada(s) ao método *pthread_delete_np(pthread t)*, encarregado de remover da aplicação a *thread* passada como parâmetro.

É importante ressaltar que algumas das funções da API do RTLinux tiveram que ser alteradas, sendo que outras funções precisaram ser criadas, conforme consta a seguir:

- **`int pthread_create (..., int importance, hrtime_t period, hrtime_t mp_ecet, hrtime_t ep_wcet);`** Função da API do RTLinux para criação de um TP, cuja alteração se dá pelo fato da mesma receber parâmetros adicionais que são utilizados para configurar o TP: importância, período, tempo de execução da MP, tempo de execução da EP. Tem as mesmas condições de retorno da sua equivalente no RTLinux, i.e. *true* em caso de sucesso e *false* em caso de falha.
- **`int test_tp (pthread_t p);`** Função criada para implementar o teste de aceitação do TP, sendo que é retornando *true* em caso de sucesso e *false* em caso de falha. Esta função pode ser chamada explicitamente pelo programador, ou utilizada internamente no escopo da função mostrada anteriormente para criar um TP.
- **`hrtime_t getAccExecutionTime(void);`** Função criada para retornar o tempo de execução efetivo decorrido para a *thread* que fez a chamada.

Fazendo uma análise crítica em relação ao código da aplicação, verifica-se que o mesmo não apresenta uma estruturação adequada, conforme se defende através do uso do paradigma de orientação a objetos. Embora seja possível utilizar a linguagem C++

[19] Arquivo objeto gerado a partir da compilação do código da FIGURA 5.7.

para implementar a estrutura mostrada na FIGURA 5.7, mesmo assim é mantido um programa voltado para a programação estruturada. Juntando-se esta constatação ao estudo discutido na seção anterior, que prevê o mapeamento do perfil UML-TR para APIs OO, decidiu-se por projetar uma biblioteca de programação OO para o ambiente desenvolvido. Esta biblioteca é discutida em detalhes no próximo capítulo.

```
...
pthread_t thr;

void *start_thr(void *arg) {
    int dummy; //required by TAFT
    struct sched_param p;

    p.sched_priority = priority_MP;
    pthread_setschedparam (pthread_self(), SCHED_OTHER, &p);

    pthread_make_periodic_np (pthread_self(), gethrtime(), PERIOD);

    //laço análogo aquele mostrado na FIGURA 5.6
}

int init_module(void) {
    int ret;

    ret = pthread_create(&thr, NULL, start_thr, (void *)1,
                         HIGH, PERIOD, MP_ECET, EP_WCET);
    if(ret==true)
           rtl_printf("TP refused! ");
    else
           rtl_printf("TP accepted! ");
    ...
}

void cleanup_module(void) {
    pthread_delete_np (thr);
}
```

FIGURA 5.7 – Programação de TPs no RTLinux/TAFT

5.4 Modelagem dos *TaskPairs* no UML-TR

No capítulo anterior discutiu-se o uso do perfil UML-TR na modelagem de sistemas tempo real e também as alternativas de mapeamento dos elementos deste perfil para o nível de programação. Nesta seção apresenta-se uma alternativa à especificação de TPs em diagramas UML. Considerando que um TP é um tipo especial de tarefa, pois além do código que trata do fluxo normal de execução (MP) também inclui um código de tratamento de exceção (EP), verifica-se a necessidade de criação de um novo estereótipo para representação do mesmo. Este estereótipo denomina-se *«TAtaskPair»*, onde o prefixo TA provém de TAFT. Propõe-se que este estereótipo seja uma especialização do estereótipo *«SAschedRes»* do perfil UML-TR.

Também se sugere a criação do estereótipo *«TAmainPart»*, a ser utilizado em conjunto com a ação tomada em resposta ao evento de ativação do TP, o qual deriva do estereótipo *«SAresponse»*. Esta ação representa a MP, i.e. o código que trata do fluxo normal de execução to TP. Para representar a EP definiu-se um estereótipo chamado *«TAexceptionPart»*. Este estereótipo é usado com a mesma operação estereotipada com *«TAmainPart»*. A diferença entre ambos está no evento de ativação: enquanto o estereótipo *«TAmainPart»* é ativado por um *«TAtrigger»*, o estereótipo *«TAexceptionPart»* é ativado por uma exceção especial proveniente do escalonador do sistema, a qual é estereotipada com *«TAmpTimeout»* e indica o chaveamento entre MP e EP. Também foram definidas duas novas marcas: *«TAmainPart».TAecet*, usada para caracterizar o tempo de execução da MP e *«TAexceptionPart».TAwcet*, usada para caracterizar o tempo de execução da EP.

Apresenta-se na FIGURA 5.8 um diagrama de seqüência da UML, o qual representa um TP com ativação periódica a cada 500 ms e com ECET da MP de 150 ms. Chama-se a atenção para o fato de que o cenário que representa a EP é modelado em separado, conforme exibido na FIGURA 5.8. Os estereótipos e marcas definidos nesta seção encontram-se especificados no anexo 1.

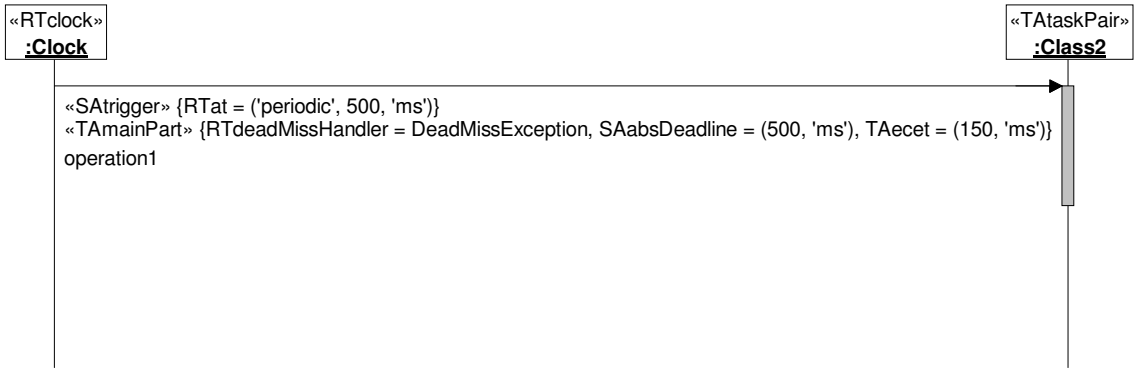


FIGURA 5.8 – Uso dos estereótipos propostos para modelagem de um TP

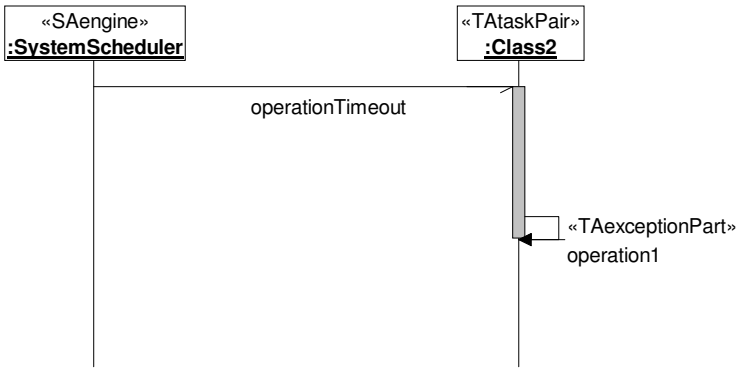


FIGURA 5.9 – Modelagem da EP correspondente

5.5 Testes e Resultados Obtidos

Os testes efetuados para avaliar experimentalmente o comportamento e a eficiência do ambiente proposto, especialmente em situações de sobrecarga, basearam-se no

conjunto de tarefas e na quantidade de carga propostos no *benchmark Hartstone* [Wei89; Wei92]. Mais precisamente, optou-se por utilizar os conjuntos de tarefas que formam as séries denominadas *PH* e *PN*. A primeira é composta por um conjunto de cinco tarefas puramente periódicas e com freqüências harmônicas. Já a segunda contém tarefas também periódicas, porém com freqüências não-harmônicas. O conjunto básico de tarefas de ambas as séries (vide TABELA 5.2) produz uma utilização nominal de 80%. A proposta do *benchmark* utilizado é aumentar sucessivamente o índice de utilização do conjunto básico e monitorar o nível de perda de *deadlines*. Com a série *PH* é possível usar dois experimentos distintos para aumentar o índice de utilização. No primeiro deles a freqüência da tarefa de maior freqüência é aumentada em múltiplos de 8 Hz. Já no segundo os tempos de execução são escalonadas por 1,1, 1,2, 1,3 e assim respectivamente, até que se atinja os limites de carga desejados. Com a série *PN* é possível utilizar somente o segundo experimento. Cada tarefa executa um laço de controle que termina exatamente quando o tempo de CPU consumido pela tarefa atinge os valores desejados. Os experimentos foram conduzidos no próprio ambiente de execução implementado.

TABELA 5.2 - Parâmetros das tarefas utilizadas nos testes

	Série Harmonica (PH)		Série Não-Harmônica (PN)	
Id Tarefa	Frequência	Tempo Exec.	Frequência	Tempo Exec.
T0	1 Hz	160,00 ms	2 Hz	80,00 ms
T1	2 Hz	80,00 ms	3 Hz	53,28 ms
T2	4 Hz	40,00 ms	5 Hz	32,00 ms
T3	8 Hz	20,00 ms	7 Hz	22,85 ms
T4	16 Hz	10,00 ms	11 Hz	14,54 ms

A fim de explorar as propriedades do escalonador TAFT, foram introduzidas algumas modificações no conjunto básico de tarefas do *Hartstone*. Aqui todas tarefas são consideradas TPs, onde o tempo de execução previsto no *benchmark* é associado à MP e um tempo adicional de 5% sobre este valor é atribuído à EP. Além disso, visto que o TAFT é voltado para tratar possíveis variações no tempo de execução das MPs (razão pela qual se trabalha com ECETs), utiliza-se de uma função de distribuição de probabilidade para modelar variações neste tempo. Assume-se para os testes que os tempos de execução variam entre 50% e 100% do WCET, que é o tempo de execução sugerido pelo *benchmark*. Esta variação é considerada conservativa, visto que na prática ela tende a ser ainda maior. Assim, define-se o ECET C_i como o quantil α (e.g. o quantil 0,95) da PDF utilizada. As PDFs beta (com parâmetros $a = 2$ e $b = 3$) e uniforme foram selecionadas para o experimento, onde valores de 0,90, 0,95 e 0,99 para α foram utilizados. As equações da FIGURA 5.10 resumem as relações mencionadas.

$ET_{MP(i)} = beta(2, 3)\ [WCET(MP_i) * 0.5,\ WCET(MP_i)]$ *ou*

$ET'_{MP(i)} = uniform\ [WCET(MP_i) * 0.5,\ WCET(MP_i)]$

$C_{\alpha i} = \alpha\text{-quantil de } ECET(MP_i)$

$E_i = WCET(MP_i) * 0.05$

FIGURA 5.10 - Relações entre os parâmetros utilizados nos testes propostos

Optou-se pelo uso das distribuições uniforme e beta por motivos distintos. A primeira é amplamente utilizada em experimentos de simulação (e.g. voltados para avaliação de desempenho [JAI 91]), principalmente para caracterizar tempos de execução que se encontram igualmente distribuídos em uma determinada faixa, obtendo uma média de 0,5 em relação aos valores mínimo e máximo (vide ilustração na FIGURA 5.11*a*). Por outro lado, acredita-se que a distribuição beta (com parâmetros $a = 2$ e $b = 3$) se aproxima do caso real, onde os valores se encontram um pouco mais próximos da média, que neste caso é de 0,47 em relação aos valores minimo e máximo, tendo assim uma menor probabilidade de ocorrerem próximos ao pior caso (vide ilustração na FIGURA 5.11*b*).

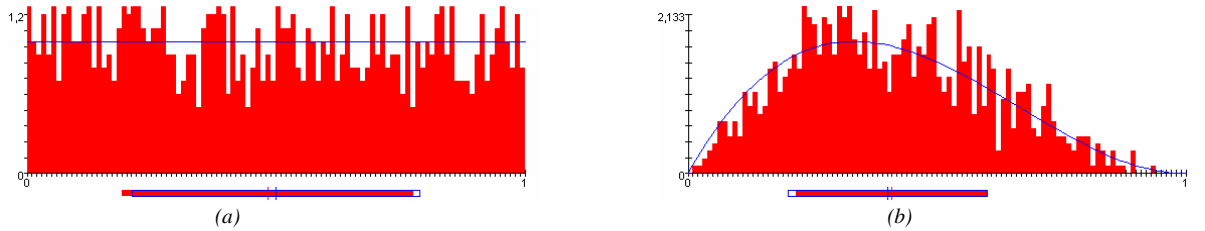


FIGURA 5.11 - Ilustração da distribuição de valores nas PDFs (*a*) uniforme e (*b*) beta(2,3)

Como conseqüência direta do uso das PDFs, tem-se que os tempos de execução utilizados nos experimentos são somente uma fração dos WCETs definidos pelo benchmark. Conseqüentemente, a utilização efetiva[20] de CPU é uma fração do valor alcançado se todas as tarefas computassem com o seu WCET, ou seja, é somente parte da chamada utilização nominal. A FIGURA 5.12 exibe uma comparação entre os índices de utilização nominal e efetiva das MPs obtidos no experimento Beta(2,3). A diferença entre utilização nominal e utilização efetiva indica a quantidade de CPU ociosa que seria verificada caso se utilizasse somente WCETs. Analisando o gráfico, nota-se uma diferença em torno de 30% menor para a utilização efetiva, valor este que representa a média dos valores gerados pela PDF Beta(2,3). Nos experimentos usando a PDF uniforme, esta diferença diminiu para 25%.

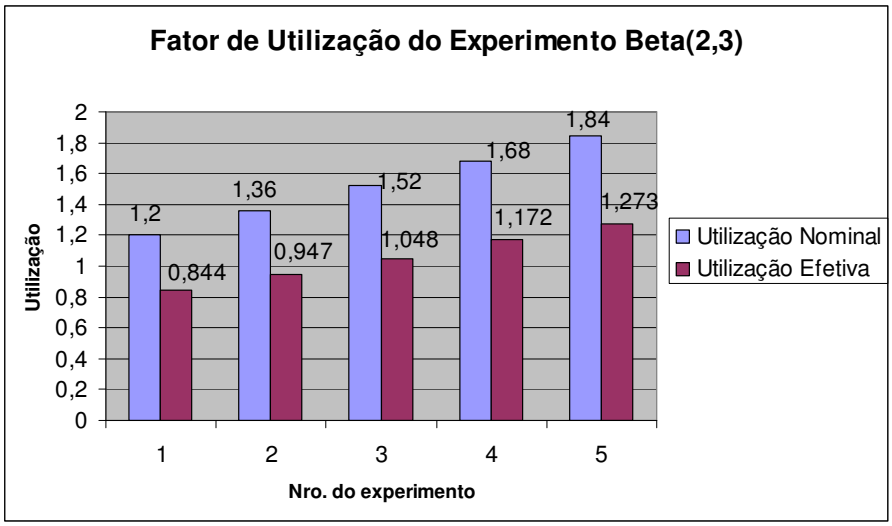


FIGURA 5.12 - Utilização nominal e utilização efetiva das MPs no experimento Beta(2,3)

[20] Refere-se a utilização obtida na prática, i.e. com base nos tempos reais de execução de cada tarefa.

Analisando os valores de utilização mostrados no gráfico da FIGURA 5.12, verifica-se que o algoritmo EDF não poderia ser utilizado nos experimentos montados, uma vez que o mesmo trabalha com WCET e, portanto, baseia-se na utilização nominal, que não pode ser superior a 100%. Isto significa que o conjunto de tarefas com carga máxima aceito pelo EDF deixa cerca de 30% de CPU ociosa. Entretanto, apesar de que em situações reais não é possível utilizar o EDF com utilização nominal acima de 100%, conforme garantido pelo seu teste de aceitação (que é ótimo), o mesmo foi usado como métrica para se comparar com o TAFT: até 100% de utilização efetiva, nenhum outro algoritmo terá melhor performance do que o EDF em termos de número de tarefas completadas. A FIGURA 5.13 exibe o comportamento do algoritmo EDF para escalonar os TPs gerados com a PDF beta para a série *PN* (os resultados para os outros casos são semelhantes): até em torno de 85% de utilização efetiva praticamente todos os *deadlines* são cumpridos (embora não estejam garantidos). Entre 85% e 100%[21] o número de perdas tem um suave aumento. Acima de 100% esta taxa aumenta drasticamente e o chamado "Efeito Dominó" conduz a um comportamento temporal imprevisível e leva instâncias de tarefas a perderem acima de 80% dos *deadlines*. Também é possível observar que aquelas tarefas com períodos maiores sofrem mais perdas do que aquelas com períodos menores, sendo que acima de 120% de utilização as instâncias das tarefas com períodos maiores são simplesmente descartadas, deixando assim mais tempo disponível para as tarefas de períodos menores (por isso se percebe uma pequena melhora em relação a taxa de perdas das mesmas).

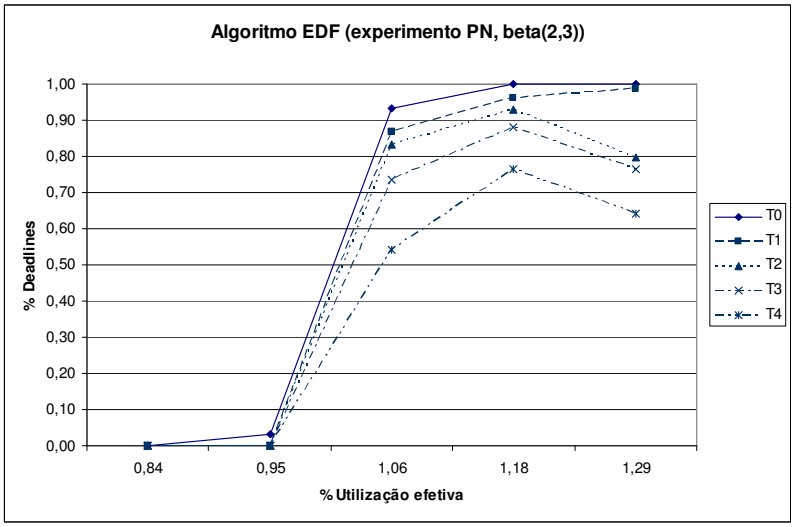


FIGURA 5.13 – Comportamento do algoritmo EDF com o benchmark Hartstone

Contrapondo-se à situação apresentada na FIGURA 5.13, observa-se o comportamento do escalonador TAFT em situação semelhante. Todavia, cabe ressaltar novamente que a execução de uma EP possui um significado totalmente diferente daquele associado com a perda de um *deadline*. Enquanto uma EP termina antes do *deadline* do TP e deixa o mesmo em um estado seguro, uma perda de *deadline* é um erro que poderá deixar o sistema vulnerável. Ressalta-se também que, dependendo da aplicação, é necessário alcançar um determinado índice de sucesso para as MPs a fim de garantir algum progresso computacional. Este índice, i.e. a taxa média de sucesso das

[21] Este limite de 100% de utilização efetiva valor foi comprovado por experimentos adicionais.

MPs, é expresso através do quantil α associado com o parâmetro C_i do TP, sendo garantido através do teste de aceitação apresentado. Dependendo do valor de α, este teste consegue aceitar uma carga maior que o EDF, negociando funcionalidade por garantias de tempo e tratando situações críticas através da invocação de EPs.

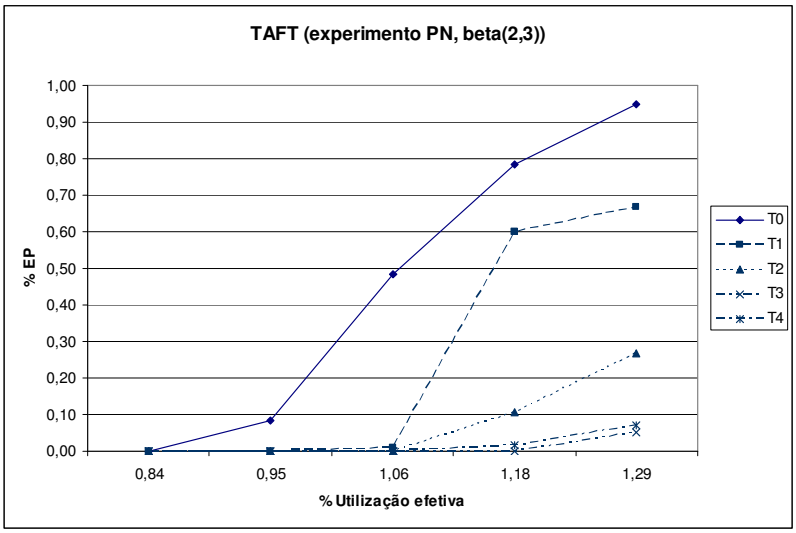


FIGURA 5.14 – Comportamento do TAFT (sem o mecanismo de adaptação) com o benchmark Hartstone

Feita a ressalva, apresenta-se na FIGURA 5.14 o desempenho observado para o TAFT – sem o mecanismo de adaptação – para o mesmo experimento feito com o EDF. Uma vez que mecanismo de adaptação não fora adotado, assume-se que o valor de α é 1, dado que a MP continua executanto mesmo após ultrapassar o seu ECET. Observa-se neste experimento que até 85% de utilização efetiva não existe a ocorrência de exceções. Entre 85% e 100%[22] apenas o TP de maior período (*T0*) apresenta um pequeno aumento no número de execuções de EPs. Acima de 95% de utilização, verifica-se um grande aumento no percentual de EPs para o TP *T0*, sendo que acima de 105% também o TP *T1* apresenta um forte aumento no número de exceções.

Repetiu-se a mesma experiência com o quantil α ajustado em 0,9. Para tanto, ressalta-se a necessidade da presença do mecanismo de adaptação no escalonador para controlar o tempo de execução da MP. O resultado obtido é exibido no gráfico da FIGURA 5.15. Observa-se neste gráfico uma tendência semelhante para os TPs de modo geral àquela obtida com a versão do TAFT sem o mecanismo de adaptação. Entretanto, nota-se que o fato do escalonador diminir a prioridade da MP após expirar o ECET, conforme explicado na seção 5.2.3, faz com que o seja criado uma espécie de "teto máximo" para o percentual de MPs completado como sucesso. Neste caso, o teto é de 90%, justamente o valor do parâmetro α. Isto acaba sendo indesejado, pois apesar de se ter CPU disponível (conforme comprovado pelo experimento anterior) o fato do escalonador reduzir o status da MP para "melhor-esforço" acaba inviabilizando a continuação da sua execução após a mesma ultrapassar o ECET. Entretanto, este efeito não deve ocorrer em situações onde a sobrecarga é transiente, i.e. causada por

[22] Apesar do gráfico FIGURA 5.15 exibir os valores de perdas nos pontos com 95% e 106% de utilização, realizaram-se experimentos com valores de utilização intermediários, a fim de comprovar que o número de perdas cresce substanciamente a partir de 100% de utilização.

computações esporádicas, e não pelo aumento de carga em todo o conjunto de tarefas, conforme feito no experimento mostrado.

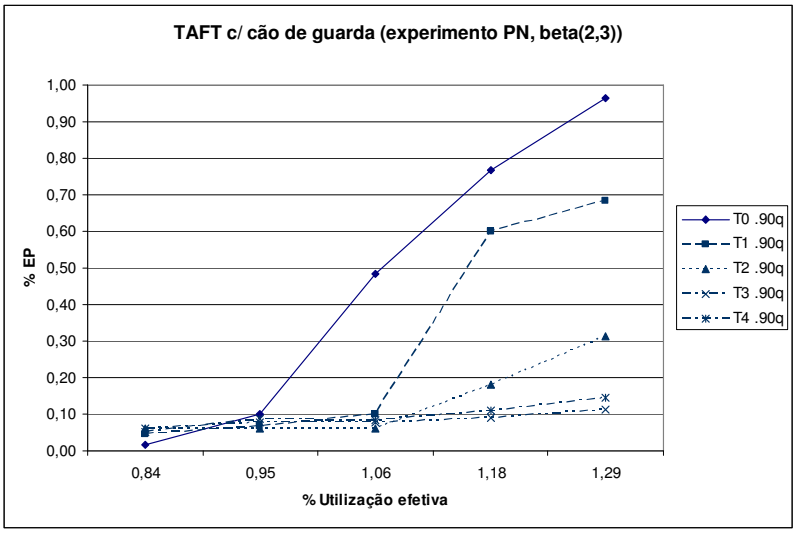


FIGURA 5.15 – Comportamento do TAFT com o mecanismo de adaptação no mesmo experimento

Para facilitar a comparação de desempenho entre os experimentos efetuados, calculou-se o chamado índice de acertos (*hit value ratio*) [BUT 95] para cada algoritmo, o qual representa o percentual de tarefas (MPs no caso do TAFT) que completaram com sucesso. Os índices calculados para o EDF, TAFT e TAFT com o mecanismo de adaptação (TAFT-MA) podem ser observados no gráfico da FIGURA 5.16. Através deste gráfico, observa-se que até 100% de utilização efetiva o TAFT possui um comportamento muito próximo do ótimo, o qual é determinado pelo EDF. Ressalta-se novamente que em casos reais o EDF não poderia ser usado com estes níveis de utilização, visto que os valores de utilização nominal estão acima de 100%. Acima desta faixa, como esperado, o EDF apresenta uma forte degradação no seu desempenho, enquanto o TAFT consegue manter um bom nível em termos de MPs completadas e garantindo a execução dos TPs sem perda de *deadlines*. Já a versão do TAFT com o mecanismo de adaptação apresenta um desempenho cerca de 10% inferior ao apresentado pela versão do TAFT sem o mecanismo de adaptação. Conforme fora discutido anteriormente, esta diferença se dá devido ao decréscimo da prioridade da MP no momento em que a mesma ultrapassa o ECET.

Devido ao uso do EDF para escalonar as MPs, verifica-se que os TPs de maior período são aqueles mais penalizados em situações de sobrecarga, pois recaem em muitas ativações de EPs. Isto é indesejável no sentido que períodos longos não significam necessariamente pequena importância[23]. Entretanto, um escalonador simplesmente baseado em prioridades, como o EDF, não possui meios para expressar importância de maneira independente da prioridade de despacho. Com base nesta constatação, realizou-se um experimento onde a idéia é utilizar o parâmetro α-quantil para expressar a importância do TP, permitindo o ajuste da taxa de sucesso das MPs de maneira ortogonal ao seu período.

[23] No jargão de escalonamento, "importância" indica o parâmetro utilizado para decidir qual tarefa dentre aquelas de mesma prioridade que deverá ser escolhida para execução.

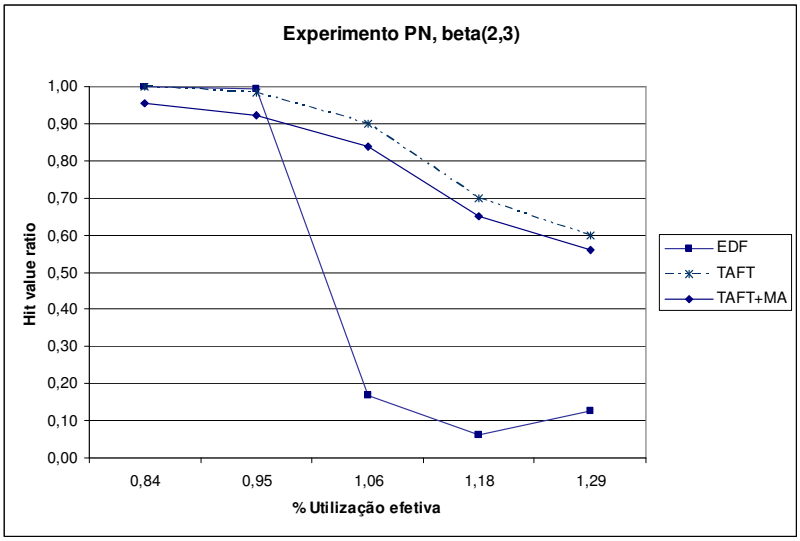


FIGURA 5.16 – Comparação entre a taxa de sucesso obtida nos experimentos efetuados

A FIGURA 5.17 exibe os resultados da execução de uma versão semelhante do experimento discutido anteriormente. A diferença é que foi selecionado um quantil α de 0,99 para os TPs *T0* e *T1* (com maiores períodos e tempos de execução) e 0,50 para os TPs *T2, T3* e *T4*. Analisando-se os resultados é possível verificar que até 95% de utilização efetiva a taxa de sucesso de *T0* e *T1* fica em torno de 100%, enquanto nesta configuração os TPs de menor período têm uma taxa de ativação de EPs consideravelmente alta, sendo afetadas pelo mesmo fenômeno observado no experimento com quantis iguais para todos os TPs. Em situações de sobrecarga, i.e. utilização efetiva acima de 100%, o mecanismo de controle de importância baseado no α-quantil acaba se tornando ineficiente, pois a taxa de exceções de *T0* e *T1* acaba por alcançar e até mesmo superar os índices de exceções dos TPs com menor α.

O último experimento demonstra que usar o parâmetro α-quantil para expressar importância não é uma boa alternativa. Com isto, implementou-se em caráter experimental uma nova versão para o escalonador das MPs utilizando o algoritmo *Highest Density First* (HDF) a fim de explorar uma nova alternativa de se expressar importância. Ao analisar as MPs prontas para execução, o HDF escolhe aquela com maior densidade, que é definida pela divisão do parâmetro denominado importância pelo tempo de execução restante da tarefa até atingir o ECET. A importância é um valor definido pelo programador, sendo que representa a importância relativa da MP em relação às outras MPs no sistema.

Com isto, realizou-se um novo experimento onde a importância relativa do TP é inversamente proporcional ao seu período (uma estratégia similar à do *Rate Monotonic*). Os resultados mostram que o HDF consegue superar o problema do EDF na atribuição da importância, i.e. consegue dar maior prioridade às tarefas de maior período, conforme mostra o gráfico da FIGURA 5.18. Todavia, ao se comparar o desempenho do HDF em relação ao EDF em termos de índice de acertos, verifica-se que o primeiro é mais eficiente somente durante situações de sobrecarga (vide FIGURA 5.19).

Como continuação dos testes efetuados vislumbra-se o uso do ambiente proposto em um estudo de caso real, envolvendo uma aplicação capaz de fazer uso dos benefícios apresentados pelo TAFT. Atualmente esta etapa já se encontra em desenvolvimento, com a implementação de uma aplicação voltada para futebol de robôs, onde as

características do TAFT são utilizadas em algoritmos para fusão de dados provenientes de sensores distribuídos [GER 2003]. Estes algoritmos são caracterizados como "*anytime*" [DEA88], isto é, são capazes de serem interrompidos a qualquer momento e retornar um resultado cuja qualidade aumenta com o passar do tempo. Naturalmente que quanto mais tempo o algoritmo executar, melhor a qualidade do resultado obtido.

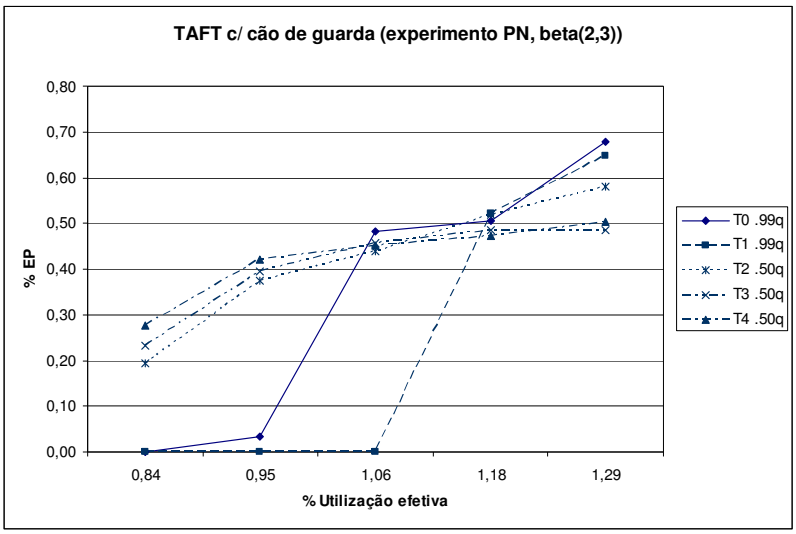


FIGURA 5.17 - TAFT com diferentes quantis α em situação de sobrecarga transiente

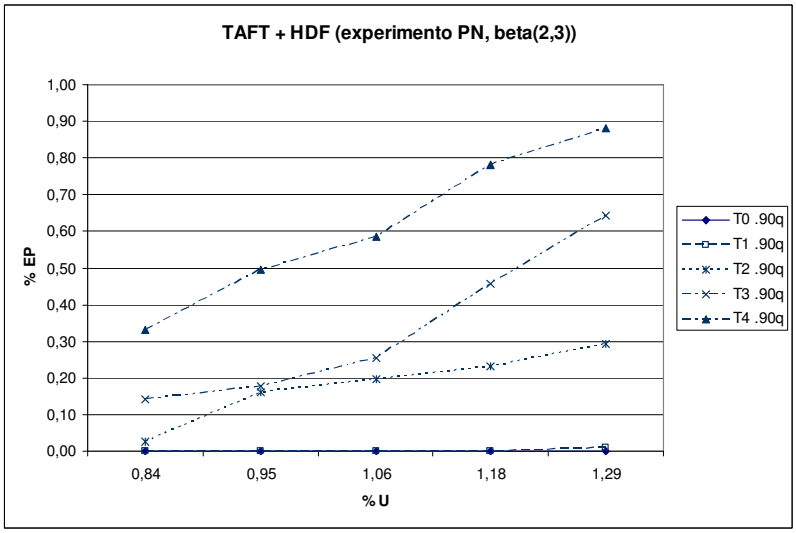


FIGURA 5.18 – Desempenho do TAFT com o algoritmo HDF para escalonar as MPs

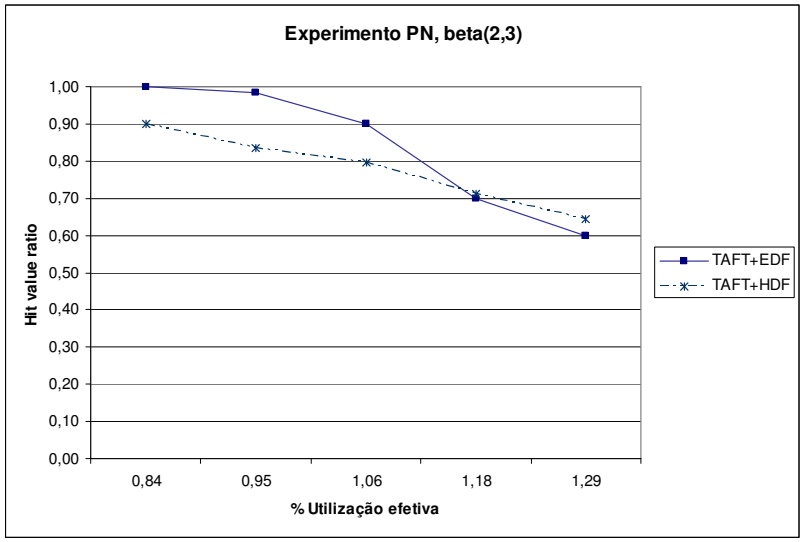


FIGURA 5.19 – Comparação de desempenho entre os algoritmos EDF e HDF no TAFT

5.6 Trabalhos Relacionados

Dois trabalhos recentes também abordaram o uso de algoritmos de escalonamento distintos para tratar situações de sobrecarga. Richardson et al [RIC 2001] apresentam o algoritmo de escalonamento chamado *Adaptative Deadline Monotonic* (ADM), que garante a execução de tarefas críticas em situações de sobrecarga. Para tanto, *pseudodeadlines* são associados com cada tarefa, onde cada vez que o mesmo for violado um novo teste de escalonabilidade é ativado e, caso a condição de escalonabilidade não seja mais válida, a condição de escalonabilidade é passada de baseada no tempo (nos *deadlines*) para baseada em prioridades (com prioridades fixas). Uma vez executando no modo baseado em prioridades o algoritmo sustenta a execução de tarefas críticas, embora suprimindo tarefas menos importantes. Comparando-se com a abordagem proposta nesta tese, no ADM é necessário recalcular a importância da tarefa cada vez que um *pseudodeadline* for violado, aumentando a sobrecarga do escalonador em tempo de execução. Além disso, ao contrário do TAFT, o mesmo não garante que o conjunto completo de tarefas aceitas irá cumprir os *deadlines*.

Bernart e Cayssials [BER 2001] também propõem uma abordagem baseada em dois modos de operação, denominada *Bi-Modal Scheduler* (BMS). No modo normal as tarefas podem ser escalonadas através de um escalonador genérico (como melhor-esforço), onde uma taxa pré-definida de perdas de *deadlines* pode ser tolerada. Quando o sistema atinge as taxas preestabelecidas ele passa a operar no modo pânico, o qual garante que *deadlines* serão sempre cumpridos através do uso de um escalonador baseado em prioridades fixas, com tarefas cujos WCETs são conhecidos. A transição entre o modo normal e o modo pânico de operação acontece quando a criticalidade da tarefa (*task criticality*) computada exceder um determinado limite. Comparando-se o BMS com o TAFT é possível verificar que o último não possui a sobrecarga de calcular a criticalidade da tarefa em cada invocação da mesma. Além do mais, o BMS se baseia no uso de WCET, cujas desvantagens já foram mencionadas.

Outra abordagem que merece ser destacada é o trabalho de Butazzo e Stankovic no desenvolvimento de técnicas de tolerância a falhas associadas ao algoritmo EDF. Estes autores propuzeram uma nova versão para o EDF denominada *Robust Earliest Deadline* (RED). Nesta nova proposta existem dois *deadlines* associados a cada tarefa: um primário e outro com um nível de tolerância. Caso uma tarefa não possa ser escalonada com o seu *deadline* primário, uma nova tentativa é feita junto ao *deadline* estendido com tolerância. A tarefa será rejeitada somente no caso de falha para ambos os testes. Comparando-se com o TAFT, este trabalho também se baseia na existência de WCETs e não oferece alternativas para abortar tarefas já inicializadas.

6 Programação de Modelos UML-TR no Ambiente RTLinux/TAFT

Ao longo do capítulo 4 discutiram-se os componentes do perfil UML-TR, enfatizando aqueles elementos que representam conceitos comumente presentes em sistemas tempo real e que são passíveis de serem mapeados para o nível de programação. Já no capítulo anterior apresentou-se uma estratégia de escalonamento que visa otimizar o uso da CPU sem perder as garantias necessárias para a execução de aplicações tempo real. O objetivo deste capítulo é discutir a integração entre os modelos UML-TR descritos no capítulo 4 com o modelo de *TaskPairs* e o escalonamento adaptativo discutido no capítulo anterior. Como resultado desta integração vislumbra-se a possibilidade de realizar-se o mapeamento dos modelos UML-TR para uma interface de programação (API) que além de suportar a programação usando o conceito de *TaskPairs*, permita a definição de requisitos temporais de maneira explícita.

6.1 Mapeamento do Modelo de Objetos para Tarefas

O mapeamento de modelos baseados no conceito de orientação a objetos para ambientes de execução baseados em tarefas concorrentes tem sido abordado por diversos autores nos últimos anos ([GOM 93; PER 94; BUR 95; AWA 96; GOM 2000]). Apesar da ampla discussão sobre este assunto, a mesma não se esgota devido ao grande dinamismo e ao crescente interesse da área, o qual é atestado pelo surgimento de novos padrões, como o perfil do UML-TR e o RTSJ. Esta seção procura discutir o nível de concorrência e paralelismo interno a um objeto, a fim de definir o mapeamento de modelos UML-TR baseados em *TaskPairs* para unidades autônomas de execução.

Segundo a proposta de Gomaa [GOM 93], cada objeto ativo da aplicação é mapeado para uma unidade de execução independente no sistema operacional. Esta unidade pode ser caracterizada ou por um processo, o qual executa em um espaço de endereçamento próprio, ou então por uma *thread* (“processo leve”), que apesar de possuir um fluxo de controle independente encontra-se na mesma região de memória das outras *threads*. Tecnologias de programação como o AO/C++ e o RT-CORBA mapeiam cada objeto ativo para um processo. Já o Java-TR baseia-se no uso de *threads* para implementar os objetos ativos.

Na seção 5.4 definiu-se que os objetos ativos que representam um *TaskPair* devem ser estereotipados com *«TAtaskPair»*. Aqui se discute como este conceito é mapeado para nível de programação e execução. Como o RTLinux é baseado no escalonamento de *threads*, procurou-se utilizar neste trabalho o modelo de concorrência no qual um objeto ativo estereotipado com *«TAtaskPair»* incorpora uma única *thread* de controle. Esta *thread* tem o seu código dividido entre parte principal (MP) e parte de exceção (EP). De maneira mais específica, o que se propõe aqui é a subdivisão do método que contém o corpo da *thread* utilizando o conjunto de macros apresentado na seção 5.3.

O uso de *threads* no RTLinux é viabilizado através da biblioteca de programação para *threads* definida pela norma POSIX 1003.1c [IEEE 2001]. Conforme detalhado na seção 5.3.1, algumas das funções POSIX implementadas no RTLinux foram devidamente modificadas para permitir a criação e gerência dos *TaskPairs*. O ponto de

interseção entre o modelo de escalonamento (baseado em *threads*) e os modelos UML é encontrado a partir do uso de uma linguagem orientada a objetos. Para a implementação de programas orientados a objetos o RTLinux permite a programação em C++.

Defende-se aqui a necessidade de se criar uma API para oferecer conceitos de programação de alto nível e evitar que os programadores tenham que executar as tarefas – bastante suscetíveis a erro – de programar usando conceitos de baixo nível. Isto porque o C++ não é nem uma linguagem concorrente nem com características tempo real. Assim, os programas atualmente escritos em C++ no RTLinux deixam a cargo do desenvolvedor a programação de baixo nível para permitir a definição das *threads* concorrentes e também dos requisitos temporais. .

Na próxima seção é apresentada uma análise sobre o mapeamento de elementos do perfil UML-TR para o nível de programação, com o objetivo de estabelecer as características desejáveis para a API do ambiente RTLinux/TAFT a ser definida neste trabalho.

6.2 Programação dos Estereótipos do UML-TR

Na TABELA 4.2 exibiu-se aqueles elementos definidos pelo perfil UML-TR que eram passíveis de serem mapeados para os mecanismos de programação descritos no capítulo 3, i.e. RTSJ, AO/C++, TMO e RT-CORBA. Esta seção procura identificar o grau de adequação destes mecanismos à representação das restrições do perfil UML-TR. Mais especificamente, identificou-se o mecanismo mais adequado para representar cada um dos estereótipos em questão. Para tanto montou-se a TABELA 6.1, onde são apontados os mecanismos selecionados para a representação dos elementos do perfil UML-TR passíveis de serem mapeados para código. Os critérios utilizados para seleção do mecanismo mais adequado foram os seguintes:

1. Clareza na separação entre programação do requisito e funcionalidade da aplicação;
2. Grau de dificuldade para implementação da construção proposta no RTLinux;
3. Facilidade para programação do requisito;

Analisando os elementos apresentados na TABELA 6.1, verifica-se que não existe um mecanismo de programação que incorpore todos os conceitos considerados como relevantes, i.e. que seja a melhor alternativa de mapeamento para os estereótipos apresentados. A API ideal é aquela que agrega características de todos os mecanismos de programação abordados. Na FIGURA 6.1 ilustra-se os elementos que deveriam constituir esta API, com base nos resultados do estudo efetuado.

Apesar de ainda não ser a API “ideal”, identifica-se o RTSJ como uma boa alternativa para representar as restrições do perfil UML-TR. Isto pode é atestado pelas facilidades de se especificar restrições temporais, como padrões de ativação e deadline. O RTSJ também facilita a criação de *threads*, uma vez que o mesmo incorpora as facilidades inerentes à linguagem Java. Além disso, o RTSJ é padronizado. As carências apresentadas pelo RTSJ se concentram na especificação de requisitos relacionados com restrições de QoS na comunicação, tipicamente usados na modelagem de aplicações distribuídas. Isto é justificado pelo fato do RTSJ não suportar a programação de aplicações distribuídas. Todavia, esta carência deverá ser brevemente superada, visto que já se encontram em discussão propostas para a elaboração da versão distribuída do Java-TR. A mais concreta destas propostas é apresentada por Wellings et al [WEL 2002].

TABELA 6.1 – Suporte das APIs estudadas à programação de requisitos temporais provenientes do perfil UML-TR

Estereótipo do perfil UML-TR	**Elemento selecionado para representar o estereótipo**
«RTaction»	Igualmente suportado no RTSJ, AO/C++, TMO e RT-CORBA
«RTtimedAction»	Métodos *timed* do AO/C++
«RTactionDead-MissHandler»	Trecho de código delimitado pelas macros "begin_exception" e "end_exception" nos métodos *timed* do AO/C++
«RTdelay»	Igualmente suportado no RTSJ, AO/C++, TMO e RT-CORBA
«RTevent» / *«SAtrigger»*	AACs do modelo TMO
«RTnewTimer» + *«RTtimer»*	Classe *Timer* do RTSJ
«CRasynch»	Invocações assíncronas no RT-CORBA
«CRsynch»	Invocações síncronas no RT-CORBA ou AO/C++
«CRconcurrent» / *«SAschedRes»* / *«TAtaskPair»*	*RealtimeThread* do RTSJ
«CRimmediate»	Igualmente suportado no RTSJ, AO/C++, TMO e RT-CORBA
«CRdeffered»	Invocações remotas no AO/C++
«SAresource» + *«GRMacquire»* + *«GRMrelease»*	Monitores do RTSJ
«SAresponse»	Método *run()* da *thread* tempo real do RTSJ

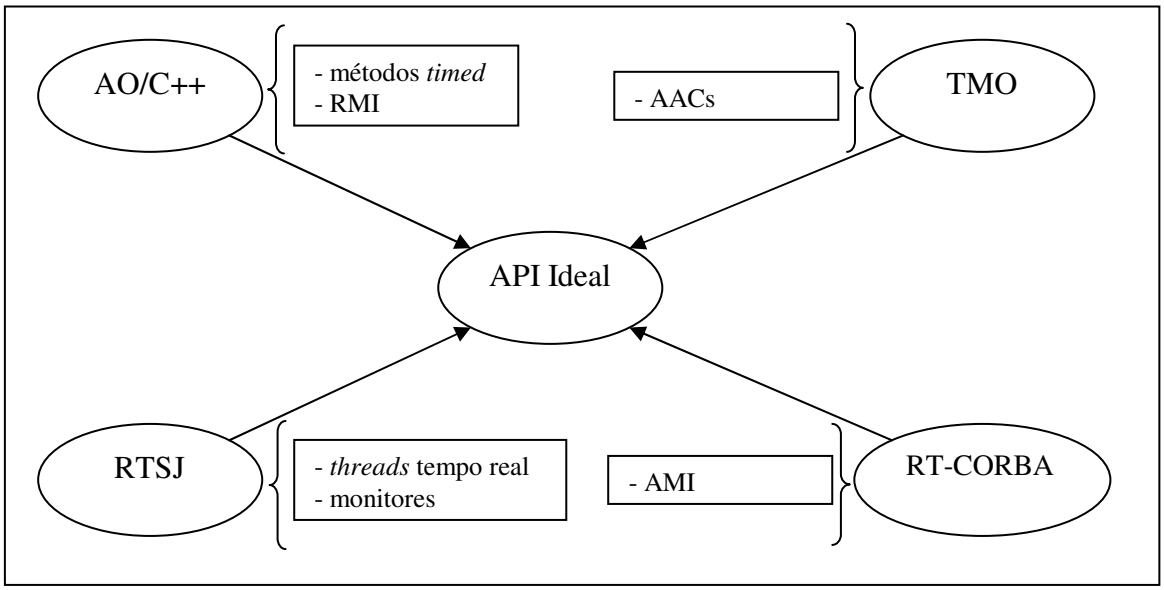


FIGURA 6.1 – Características de uma API "ideal"

Uma vez justificada a potencialidade do RTSJ como alvo do mapeamento proposto para o perfil UML-TR, a mesma foi escolhida como base para a API orientada a objetos desenvolvida para o ambiente TAFT, a ser referenciada por TAFT-API. Na próxima seção apresentam-se os detalhes da API em questão.

6.3 Descrição da TAFT-API

Dado que escalonador TAFT baseia-se no escalonamento de *threads*, adotou-se na TAFT-API um modelo de concorrência semelhante aquele adotado pela linguagem Java, onde todo objeto ativo tem sua própria *thread*, sendo o corpo da mesma definido no método *run()*. Um detalhe a respeito deste modelo é que um objeto ativo é ao mesmo tempo passivo, pois com a exceção do método *run()*, os outros métodos podem ser chamados concorrentemente. Neste caso, os métodos executam no contexto da *thread* do objeto que chamou o método. Isto traz um problema de sincronismo durante o acesso aos atributos, cabendo ao programador identificar as regiões críticas e protegê-las com mecanismos de exclusão mútua.

No desenvolvimento da TAFT-API procurou-se definir uma API que fosse bastante expressiva com relação aos requisitos temporais, mas que também permitisse a geração de programas cujo comportamento temporal fosse determístico e computacionalmente eficiente. Sendo a API proposta baseada no paradigma de orientação a objetos, procurou-se limitar o uso de algumas características que pudessem levar a um indeterminismo do comportamento temporal da aplicação (em relação ao uso de linguagens não OO). Como exemplo destas características, destaca-se a verificação dinâmica de tipos (*dynamic casting*), a ligação dinâmica de funções virtuais (*dynamic binding*) e também o mecanismo de tratamento de exceções (*exception handling*).

Conforme definido anteriormente, a TAFT-API é fortemente baseada na API do Java-TR, todavia na implementação do experimento de validação das idéias propostas nesta tese, optou-se pela linguagem C++. O principal motivo por optar-se pela linguagem C++ ao invés do Java é o fato de não existir uma máquina virtual Java com as extensões tempo real propostas pelo RTSJ implementada na plataforma RTLinux. Além disso, o fato da RTSJ ser padronizada leva a crer que a mesma terá uma grande popularização entre programadores de sistemas tempo real nos próximos anos. Encontram-se listadas abaixo as classes provenientes do RTSJ as quais foram implementadas em C++ no ambiente RTLinux, de modo a constituir os elementos propostos para a TAFT-API (a definição completa da mesma encontra-se no anexo 2):

1. *ReleaseParameters*: Classe abstrata da qual derivam classes como *AperiodicParameters, SporadicParameters* e *PeriodicParameters*, que representam os parâmetros que definem as condições de liberação para execução de objetos ativos. Na TAFT-API, esta classe contém um parâmetro e também dois métodos adicionais em relação ao RTSJ. Os métodos servem para modificar e ler o parâmetro, que é usado para armazenar informações estáticas a respeito do "tipo" do parâmetro (i.e. periódico, aperiódico ou esporádico), evitando assim o uso do mecanismo para verificação dinâmica de tipos, o qual se baseia na classe instanciada.
2. *PeriodicParameters*: Classe derivada de *ReleaseParameters* que condiciona a execução periódica de objetos ativos, definindo que o método *waitForNextPeriod()* do mesmo será desbloqueado no início de cada período.
3. *SporadicParameters*[*]: Classe derivada de *ReleaseParameters* que caracteriza o padrão de execução dos objetos ativos como esporádico, definindo o tempo

[*] estas classes apesar de definidas na TAFT-API, não são usadas devido as atuais restrições previstas no atual modelo de escalonamento.

mínimo decorrido entre duas ocorrências, ou seja, a máxima freqüência de ocorrência dos eventos esporádicos.

4. *AperiodicParameters*[*]*:* Classe derivada de *ReleaseParameters* que caracteriza o padrão de execução dos objetos ativos como não-periódico.
5. *SchedulingParameters:* Classe abstrata da qual derivam classes como *PriorityParameters* e *ImportanceParameters*, as quais contém como atributos informações utilizadas pelos escalonador tempo real, tais como prioridade e importância.
6. *PriorityParameters*[*]: Instâncias desta classe são usadas como parâmetros na criação de objetos ativos, usada por alguns algoritmos de escalonamento para indicar a prioridade da *thread* que contém a sua execução.
7. *ImportanceParameters*[*]: Instâncias desta classe são usadas como parâmetros na criação de objetos ativos, sendo que no escalonador TAFT esta informação é utilizada para expressar a importância do *TaskPair*.
8. *RealtimeThread*: Esta classe permite a criação de *threads* tempo real usadas para implementar-se o comportamento dos objetos ativos. O contrutor desta classe contém parâmetros que permitem informar ao sistema suas demandas de processamento e restrições temporais, e.g. instância da classe *ReleaseParameters* e suas derivadas e também de *SchedulingParameters* e suas derivadas. A classe *RealtimeThread* contém o método abstrato *run()*, o qual deve ser redefinido nas classes derivadas da mesma para representar o código do TP. Conforme mencionado, este código deverá conter as macros usadas para realizar a separação entre o código da parte principal (MP) e o código de tratamento de exceção (EP). Diferentemente do RTSJ, a classe *RealtimeThread* da TAFT-API não necessita de um objeto adicional para representar o código a ser executado em caso de violação de deadline. Este fato representa outra alteração proposta pela TAFT-API em relação ao RTSJ.
9. *RelativeTime*: Esta classe representa um instante de tempo relativo. Mais especificamente, a mesma é usada para representar um intervalo de tempo com resolução de nanossegundos. Exemplos destes intervalos são os períodos, deadlines, tempos de execução, etc.
10. *Timer*: Classe abstrata da qual derivam classes que possibilitam a criação de temporizadores, como *OneShotTimer* e *PeriodicTimer*.
11. *OneShotTimer*: Classe que representa um temporizador que executa uma uma única vez.
12. *PeriodicTimer*: Classe que representa um temporizador que executa periodicamente.
13. *AsyncEventHandler*: Classe abstrata, a qual define o método *handleAsyncEvent()*, usado para representar o código executado caso o evento de timeout provocado pelo término do *timer* seja disparado.
14. *PriorityCeiling*: Classe que representa a política *priority ceiling*.
15. *PriorityInheritance*: Classe que representa a política *priority inheritance*.

[*] estas classes apesar de definidas na TAFT-API, não são usadas devido as atuais restrições previstas no atual modelo de escalonamento.

Também foram definidas duas novas classes para suportar a implementação de objetos com acesso controlado, conforme caracterizado pelo estereótipo *«SAresource»* do perfil UML-TR. Estas classes encontram-se descritas a seguir:

1. *Semaphore*: Classe utilizada para prover o mecanismo de exclusão mútua necessário para os elementos do modelo estereotipados com *«SAresource»* através de um semáforo. Implementa os métodos *acquire()* e *release()*, que suportam respectivamente os estereótipos *«GRMacquire»* e *«GRMrelease»*. Além disso, implementa também o método *timeoutHandler()*, executado caso o recurso não seja liberado dentro do tempo definido pelo atributo *bloqTimtout*. Esta classe é usada nos casos onde a marca *SAaccessControl* não estiver definida.
2. *Mutex*: Possui funcionalidade semelhante à classe anterior, porém utiliza um *mutex* para controlar as regiões críticas condicionais. Esta classe oferece suporte à políticas de controle sobre inversões de prioridade.
3. *MutexControl*: Classe cuja funcionalidade é alusiva à classe MonitorControl proposta no RTSJ. Instâncias da mesma podem ser associadas com objetos de classes derivadas de Mutex, a fim de estabelecer uma política de controle de acesso para a região de acesso exclusivo. As políticas suportadas são *priority inheritance*, definida na classe *PriorityInheritance*, e *priority ceiling*, definida na classe *PriorityCeiling*.

Outra classe não definida no RTSJ, porém implementada na presente API é a classe *Util*, a qual foi usada nos estudos de caso apresentados em [HÖL 2002] e [BEC 2002]. Esta classe foi criada para suportar algumas funções utilitárias, como a solicitação de bloqueio por tempo limitado (*delay*) e também a impressão de mensagens no console.

A próxima seção descreve a relação entre os componentes da TAFT-API e os estereótipos do perfil UML-TR apontados como passíveis de serem mapeados para código, permitindo assim a transição.

6.4 Mapeamento do Perfil UML-TR para a TAFT-API

Esta seção objetiva completar o ciclo "especificação-implementação-execução" através do mapeamento dos elementos do perfil UML-TR para a TAFT-API. Para tanto, os estereótipos apresentados na TABELA 6.1 são mapeados para trechos de código que servem para exemplificar o mapeamento proposto.

Inicia-se esta descrição pelo estereótipo *«RTaction»*. Este estereótipo não requer nenhuma construção especial ao nível de programação, pois representa simplesmente um conjunto de ações que podem ser implementadas em um bloco de código ou método. Já o estereótipo *«RTtimedAction»* requer o uso da API proposta, pois caracteriza um conjunto de ações controladas no tempo. Para exemplificar a implementação deste estereótipo, mostra-se na FIGURA 6.2 o código usado para programar a restrição da FIGURA 4.2. Também se destaca nesta figura a presença do código executado em caso de timeout, o qual é representado no modelo pelo estereótipo *«RTactionDead-MissHandler»*.

```
class TimerHandler: public AsyncEventHandler{
  void handleAsyncEvent(){
    //Código do timeout (representado pelo estereótipo
«RTactionDeadMissHandler»)
  }
};

void Class1::controledAction(){
  TimerHandler m_timeout;
  m_pmonitor = new TAPI_TimeoutMonitor( 1000, m_timeout );
    //código normal da operação
  m_ pmonitor.normalEnd();
}
```

FIGURA 6.2 – Programação do estereótipo *«RTtimedAction»* na TAFT-API

Os estereótipos *«RTevent»* e *«SAtrigger»* são usados para representar o padrão de ocorrência de um evento. Os mesmos podem estar associados com *timers* de um só disparo, conforme exemplificado anteriormente, ou com *timers* periódicos, que são instâncias da classe *PeriodicTimer*. Além disso, estes estereótipos também são usados com o evento que caracteriza a execução de um objeto ativo. Nestes casos, os mesmos são implementados através das classes derivadas da classe *ReleaseParameters*. Destaca-se na FIGURA 6.3 o uso da classe *PeriodicParameters* (derivada de *ReleaseParameters*) da TAFT-API para representar o estereótipo *«RTevent»* usado na FIGURA 4.3, o qual determina um padrão de ocorrência periódico para o cenário exibido.

```
Class2::Class2(): RealtimeThread(
    new PriorityParameters(NORM_PRIORITY),
    new PeriodicParameters( 0 /*ReleaseTime*/, 500 /*RTevent.RTat*/,
                            120 /*SAResponse.RTduration*/,
                            500 /*RTtimedAction.SAAbsDeadline*/)
    ) {
    //código de inicialização do objeto
}
```

FIGURA 6.3 – Programação do padrão de ocorrência de um evento

Para representar os estereótipos *«CRconcurrent»*, *«SAschedRes»* e *«TAtaskPair»* são utilizadas *threads* tempo real. Para criar uma *thread* tempo real é necessário instanciar uma classe derivada da classe *RealtimeThread*. Este classe deverá necessariamente redefinir o método *run()*, o qual contém o corpo da thread. No caso de se programar *TaskPairs*, é necessário acrescentar ao método *run()* as macros usadas para delimitar as partes principais (MP) e o código de exceção (EP), conforme exibido na FIGURA 6.4. Nota-se que o código apresentado na FIGURA 6.3 é justamente o construtor desta classe.

Tem-se ainda o conjunto de estereótipos usados para representar recursos protegidos, i.e. *«SAresource», «GRMacquire»* e *«GRMrelease»*. Objetos passíveis deste tipo de restrição são implementados na TAFT-API como instância de classe derivada de *Semaphore* ou *Mutex*. A primeira representa um mecanismo de sincronização primitivo, o qual inclui os métodos *P()* e *V()*, utilizados respectivamente para testar e incrementar o semáforo. O uso deste recurso é exemplificado na FIGURA 6.5. Já a classe *Mutex* representa um mecanismo de controle de acesso mais sofisticado, pois além das primitivas de acesso e liberação do recurso também inclui uma variável de sincronização, tornando-se semelhante ao conceito de monitor. A diferença é que os métodos da classe derivada de *Mutex* não são sincronizados automaticamente, cabendo

aos programadores esta tarefa. Além dissso, é possível associar à objetos derivados de Mutex uma instância da classe MutexControl, permitindo a definição de uma política para controle sobre inversões de prioridade. Na FIGURA 6.6 é exibido um exemplo de utilização para a classe Mutex, o qual representa a implementação da situação exibida na FIGURA 4.7.

```
class Class2: public RealtimeThread(){
   ...
};

void Class2:run(){
    while( m_isRunning ) {
        waitForNextPeriod();
        BEGIN_MAIN_PART
            //código da MP
        END_MAIN_PART
        BEGIN_EXCEPTION_PART
            //código da EP
        END_EXCEPTION_PART
    }
}
```

FIGURA 6.4 – Definição de uma *thread* tempo real

```
class Data: public Semaphore {
    int d;
};

void Data::writeData(int d){
    p()
    this->d = d;
    v();
}

int Data::readData(){
    return d;
}
```

FIGURA 6.5 – Implementação de um recurso protegido

Por fim destaca-se o estereótipo *«RTdelay»*, utilizado para bloquear o TP por um tempo determinado. A implementação do mesmo na TAFT-API é feita através do método estático *delay(<time>)* da classe *Util*.

```
class Data: public Mutex {
    int d;
};

void Data::timeoutHandler(){
    // código executado caso o mutex não seja liberado no tempo solicitado
}

void Data::writeData(int d){
    lock(100 /*Deadline*/)
    this->d = d;
    unlock();
}

int Data::readData(){
    return d;
}
```

FIGURA 6.6 – Implementação de um recurso protegido

Na TABELA 6.2 são resumidos os elementos da TAFT-API usados na implementação dos estereótipos proveniente do perfil UML-TR e passíveis de serem mapeados para código.

TABELA 6.2 – Resumo do mapeamento do perfil UML-TR para a TAFT-API

Elemento do Perfil UML-TR	Elementos da TAFT-API usados na programação do estereótipo
«RTaction»	nenhum
«RTtimedAction»	Classe *TAPI_TimeoutMonitor*.
«RTdelay»	Método estático *delay()* da classe *Util*.
«RTevent» / *«SAtrigger»*	Classe *TAPI_TimeoutMonitor* ou classes derivadas de ReleaseParameters.
«CRconcurrent» / *«SAschedRes»* / *«TAtaskPair»*	*Thread* tempo real criada através da instanciação de classe derivada de *RealtimeThread*.
«SAresource» + *«GRMacquire»* + *«GRMrelease»*	Implementadas como especialização das classes que promovem mecanismos de exclusão mútua, como *Semaphore* e *Mutex*, as quais provêm métodos para suportar os estereótipos *«GRMacquire»* e *«GRMrelease»*.
«SAresponse»	Método *run()* da classe *RealtimeThread*.

6.5 Considerações Finais

A TAFT-API se oferece uma alternativa ao uso de bibliotecas que acessam diretamente as funcionalidades do sistema operacional na especificação de tarefas concorrentes e de requisitos temporais. Com isto, consegue-se uma melhor separação entre definição das funcionalidades da aplicação e a especificação dos seus diversos requisitos, melhorando assim a legibilidade das aplicações desenvolvidas. Além disso, os elementos do UML-TR permitem uma representação destes requisitos em alto-nível, ou seja, durante a etapa de modelagem da aplicação. Desta forma, consegue-se realizar uma transição suave entre as etapas de especificação, implementação e execução, conforme exibido na seção anterior.

A API projetada permite demonstrar que o mapeamento dos requisitos temporais e aspectos de concorrência descritos nos modelos orientados a objetos é viável. Todavia a API definida não deve ser encarada como uma API completa e certamente outras características poderiam ser incorporadas. Portanto, caracteriza-se como trabalho futuro a extensão da TAFT-API proposta nesta tese. Entre os pontos não abordados pela API atual encontram-se o suporte aos estereótipos relacionados com comunicação, como o *«CRasynch»* e o *«CRsynch»* e as suas devidas marcas. Na versão atual da TAFT-API, suporta-se somente chamadas síncronas.

Também fica como sugestão a modificação das classes que premitem a especificação do padrão de ocorrência de requisitos temporais, a fim de que seja possível utilizar requisitos mais elaborados, como por exemplo as expressões AAC definidas pelo modelo TMO.

Como principal fator a ser abordado por esta API está a inclusão de alternativas para a especificação de requisitos fim-a-fim, tema este que tem sido abordado por diversos autores (e.g. grupo do RT-CORBA TAO [SCH 98]), porém cuja solução continua em aberto.

7 Mapeamento do Perfil UML-TR para a TAFT-API: Estudo de Caso

Neste capítulo apresenta-se um estudo de caso que aborda o mapeamento do perfil UML-TR para em a interface de programação desenvolvida para o ambiente de execução do escalonador TAFT, ou seja a TAFT-API. Através deste estudo, objetiva-se exemplificar a utilização dos recursos de modelagem e implementação apresentados na tese. Este capítulo inicia pela apresentação do sistema utilizado como estudo de caso.

7.1 Apresentação do Estudo

O sistema selecionado para o estudo proposto provém de uma aplicação real, composta por um conjunto de veículos autônomos [SCH 2001], encarregados de cooperar entre si para evitar colisão enquanto os mesmos estiverem se deslocando em circuitos com interseções, conforme ilustrado na FIGURA 7.1. Observa-se nesta figura a existência de uma área deonminada zona de aproximação (*approaching zone*). a qual caracteriza o espaço que o veículo tem para mudar a sua velocidade a fim de evitar colisões na chamada zona crítica (*hot spot*), que é justamente o espaço onde pode haver colisões.

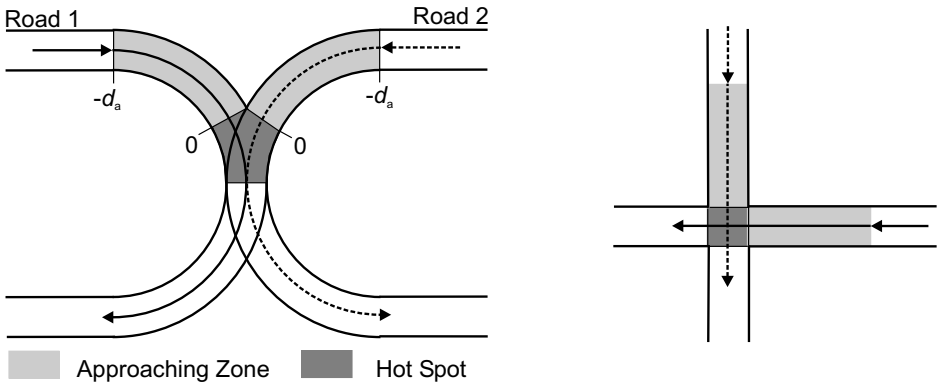


FIGURA 7.1 – Exemplo de circuitos onde os veículos pederão trafegar

Os veículos modelados no estudo de caso são do tipo *Kurt2* [KURT 2001]. Este veículos representam uma aplicação tempo real embutida, pois são dotados de um processador Pentium, um microcontrolador Infineon C167, uma interface de comunicação para redes sem fio (*wireless*), uma interface de comunicação CAN-bus, uma câmera digital para captura imagens, um encoder para monitoração de deslocamento e velocidade, e também de outros sensores não relevantes para este estudo, como sensores de colisão e sensores infravermelhos. Além do conjunto de sensores, o sistema também possui dois motores de passo, encarregados de impulsionar, frear e direcionar o veículo, de acordo com os comandos provenientes do sistema de controle. A FIGURA 7.2 exibe uma visão geral de um *Kurt*. Convém salientar que a execução da aplicação projetada não utilizou a plataforma real dos veículos, mas sim

um computador *desktop*, além de que a execução da aplicação se deu de forma simulada. Nas seções que seguem apresenta-se, respectivamente, a modelagem do sistema, o mapeamento do modelo para o ambiente de execução e também os resultados relativos à sua execução e posterior validação.

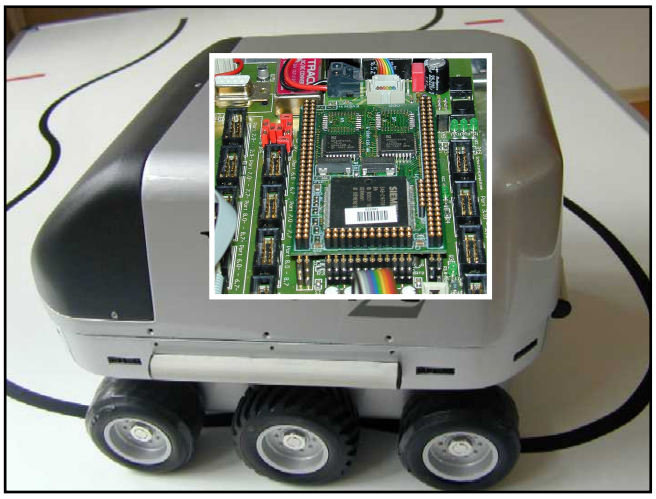


FIGURA 7.2 - Visão geral do veículo autônomo *Kurt*

7.2 Modelagem do Sistema de Veículos Autônomos

Seguindo o método proposto nesta tese, o estudo de caso teve início com o processo de análise do domínio do problema. O resultado desta fase é um modelo orientado a objetos usado no projeto do sistema real. Para construção deste modelo, utilizou-se a ferramenta *Real-Time Studio* da Artisan Sw, cedida pela empresa para fins de avaliação e testes. Justifica-se a opção por esta ferramenta pelo fato da mesma suportar o uso do perfil UML-TR, além de oferecer facilidades que permitem a configuração do seu gerador de código.

Na primeira etapa do processo de análise, identificaram-se as classes provenientes do domínio da aplicação, ou seja, as classes que representam a interface entre os elementos físicos do veículo (como sensores e atuadores) e os elementos lógicos presentes no *software* sendo projetado. A FIGURA 7.3 contém os elementos identificados, que representam os primeiros componentes do Diagrama de Classes do modelo.

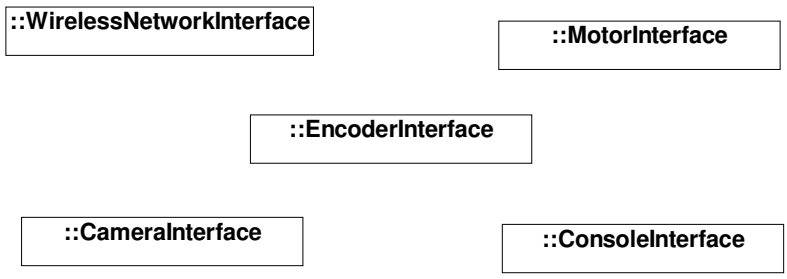


FIGURA 7.3 – Classes de interface entre os elementos físicos do veículo e a aplicação projetada

Após a definição das classes de interface, voltou-se o foco do estudo para a especificação das funcionalidades presentes no sistema. Estas funcionalidades foram listadas através de casos de uso (*use cases*). As mesmas foram agrupadas em um Diagrama de Use Cases, o qual relaciona os *use cases* com os elementos físicos do domínio do problema (chamados de atores) com os quais os mesmos interagem. O diagrama desenvolvido é mostrado na FIGURA 7.4. Encontram-se neste diagrama seis funcionalidades principais, a saber: inicialização (*Initialization*), mudança de pista e de velocidade (*Track & Speed Change*), processamento de imagens (*Image Processing*), sistema de odometria (*Odometry*), sistema de navegação (*Cruise Control*) e por fim o sistema de coordenação (*Coordination*), que utiliza as funcionalidades de localização (*Self Location*) e de processamento de estados (*State Processing*).

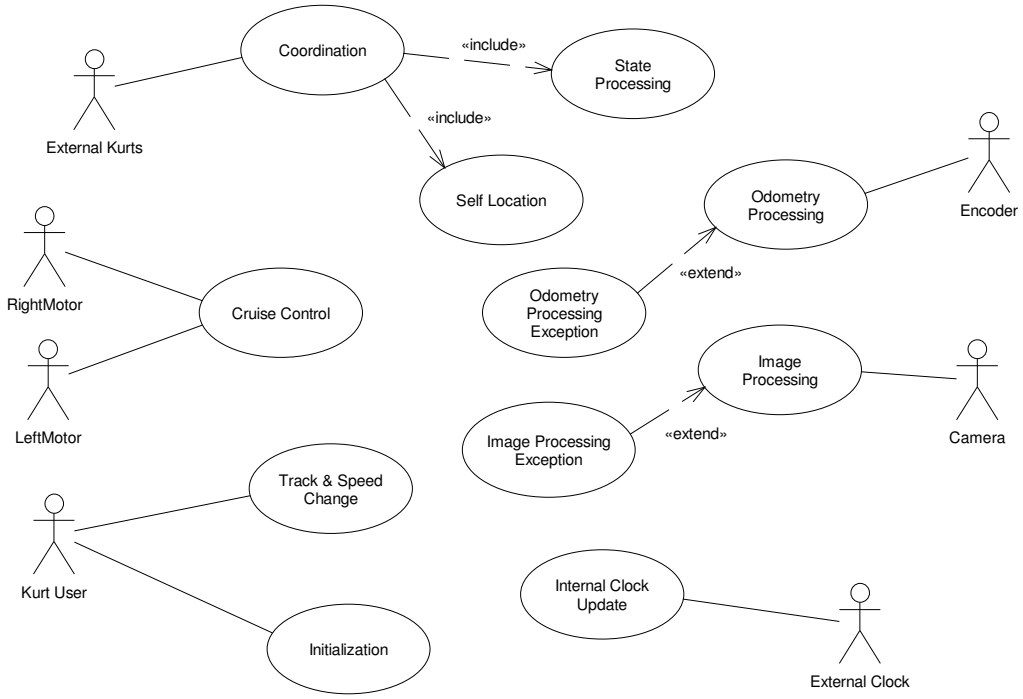


FIGURA 7.4 - Diagrama de Use Case do sistema Kurt

Posteriormente, cada *use case* identificado foi detalhado através de Diagramas de Colaboração. Conforme Gomaa [GOM 2000], uma visão mais detalhada das funcionalidades do sistema é importante por permitir a identificação das entidades lógicas encarregadas de promover a realização das funcionalidades propostas. Os Diagramas de Colaboração permitem descrever as instâncias de classe (objetos), ou somente as classes, que colaboram entre si para prover uma determinada funcionalidade. Além disso, define-se o seqüenciamento das operações realizadas. Tanto as operações como os elementos que as realizam podem ser decorados com as restrições temporais presentes no perfil UML-TR.

O grau de complexidade apresentado pelas colaborações é bastante variado. Embora existam colaborações simples, como a inicialização do sistema, existem também colaborações bastante complexas, como a que define o sistema de coordenação. No total, o sistema é composto por 14 diagramas de colaboração. Optou-se por discutir neste estudo de caso somente duas colaborações, a fim de não tornar a discussão muito

exaustiva. As colaborações escolhidas são o processamento de imagem e o sistema de coordenação.

O processamento de imagem é encarregado de periodicamente coletar uma imagem e processá-la a fim de determinar o alinhamento do veículo em relação à linha que determina o circuito e também a presença de marcas que indicam a aproximação das chamadas áreas de risco (*critical area*), que são os pontos aonde podem ocorrer colisões entre os veículos. A determinação do alinhamento do veículo é usada pelo sistema de navegação (*Cruise Control*) para manter o mesmo na pista. Já a detecção de marcas é enviada para o sistema de localização (*SelfLocator*), que por sua vez determina se indica a entrada ou a saída da região crítica, interagindo com o sistema de colaboração entre os veículos. A colaboração projetada é exibida na FIGURA 7.5.

Esta colaboração é cíclica, sendo iniciada por um evento proveniente do relógio do sistema, que ativa o método *processImage()* da classe *ImageProcessor*. A caracterização deste evento é feita com o uso do perfil UML-TR, que o define com o estereótipo *«SAtrigger»*, usando a marca *RTat* para caracterizá-lo como "periódico" e com ocorrência a cada 20ms. A colaboração também apresenta a seqüência de ações, estereotipadas com *«SAaction»*, a serem executadas. O tempo de execução e o *deadline* para execução de todas as atividades encontram-se definidos, respectivamente, nas marcas *RTduration* e *SAabsDeadline* do estereótipo *«SAresponse»*. Outro detalhe que merece ser ressaltado é o uso da classe *ImageInfoStorage*, que por necessitar de um mecanismo de controle de acesso é caracterizada pelo estereótipo *«SAresource»*, utilizando o protocolo "*PriorityInheritance*" (marca *SAaccessControl*) e com capacidade igual a 1 (marca *SAcapacity*).

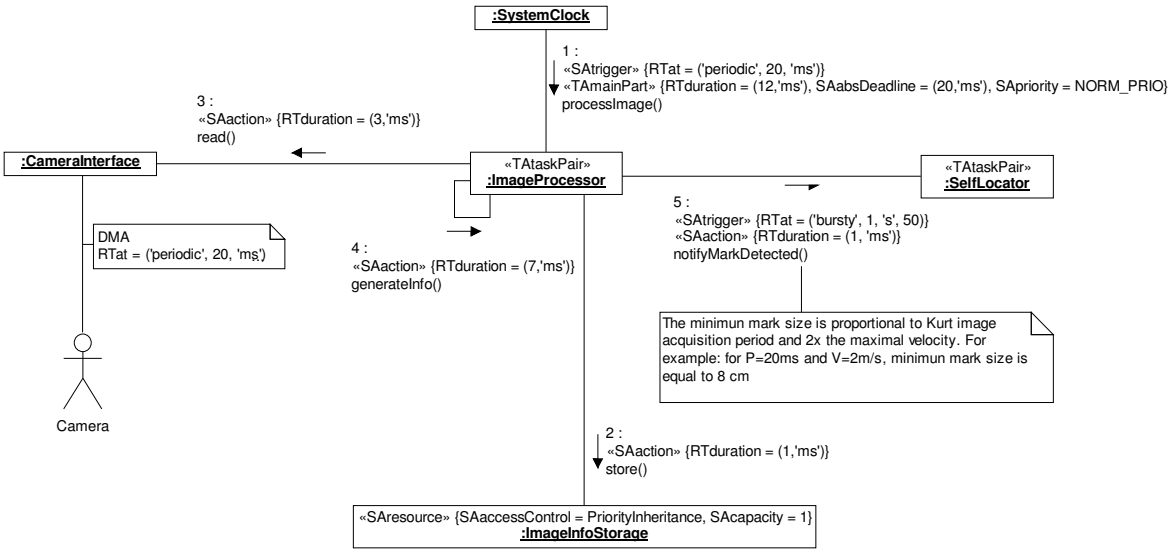


FIGURA 7.5 – Diagrama de colaboração do processamento de imagem

Um detalhe que merece atenção é o fato de nem todos os diagramas de colaboração representarem o fluxo normal de execução. Existem os chamados "fluxos de exceção", que são acionados no momento em que se verifica a ocorrência de situações especiais, como por exemplo a perda de um *deadline*. Como alguns dos elementos do sistema *Kurt* necessitam de um comportamento temporal rígido, são aplicadas situações de tratamento de exceção nos mesmos. O processamento de imagens é um exemplo. Nesta funcionalidade, a exceção está associada com o método *processImage()*, uma vez que ela está diretamente relacionada com o seu tempo de processamento.

Uma nova colaboração é criada para representar esta situação especial, conforme mostrado na FIGURA 7.6. O método *processExceptionHandler()* realiza um processamento para verificar se houveram mais de *k* exceções seguidas. Se esta condição for verdadeira, então o veículo é parado. Caso contrário, o resultado do último processamento de imagem é mantido. O valor de *k* é determinado dinamicamente, variando conforme a velocidade do veículo, porém garantindo que as marcas da área crítica e também o circuito serão sempre detectados.

A colaboração que descreve o sistema de cooperação provém do protocolo proposto por Schemmer et al [SCH 2001]. A idéia principal deste protocolo é permitir que cada veículo conheça o estado corrente de todo o sistema, tendo capacidade de decidir por eventuais alterações de velocidades naqueles veículos que estiverem ingressando nas áreas com risco de colisão. A colaboração projetada acaba por ser complexa porque além de disparar outras colaborações (e.g. nas operações *calculateIntermediateState()* e *calculateFinalState()*), o protocolo é dividido em três etapas, enumeradas por *A, B* e *C* na FIGURA 7.7. A etapa *A* é disparada localmente em um único *Kurt*, quando o mesmo estiver ingressando na área de risco. O inicio desta operação é determinado pela funcionalidade de localização (encontrada na classe *SelfLocator*), que dispara o evento assíncrono *notifyCriticalAreaEntrance*. As outras duas etapas são disparadas por eventos publicados no meio de comunicação (*WirelessNetworkInterface*), sendo proveniente de outros *Kurt* ou também por ele próprio. Os eventos das etapas B e C são, respectivamente, *notifyRequestGroupJoinArrival* e *notifyGroupInsertion*. Conforme ressaltado no capítulo 4, o perfil UML-TR não apresenta um estereótipo capaz de relacionar estas etapas através de um requisito temporal.

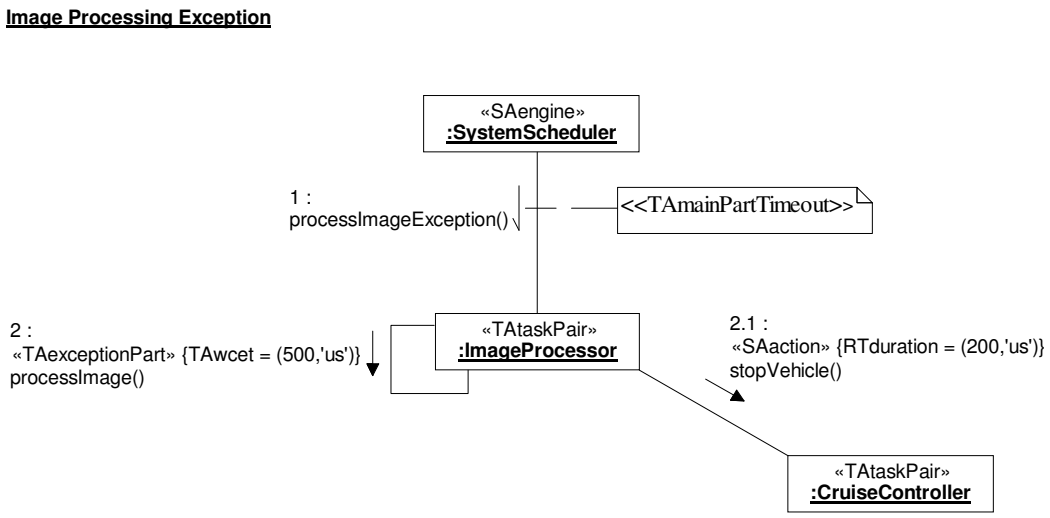


FIGURA 7.6 – Procedimento de exceção no processamento de imagem

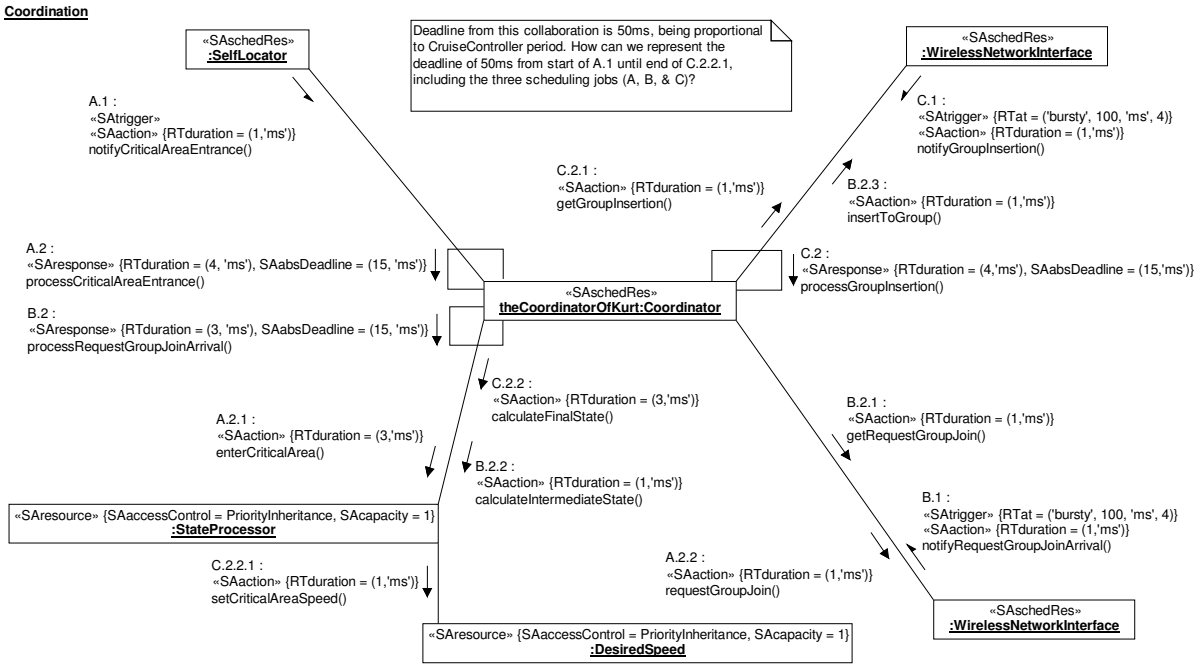


FIGURA 7.7 – Diagrama de colaboração do sistema de cooperação

Conforme pode ser observado na FIGURA 7.7, os elementos desta colaboração também são decorados pelas restrições provenientes do perfil UML-TR. O objeto *theCoordinatorOfKurt* por exemplo, por ser classificado com o estereótipo *«SAschedRes»*, representa um elemento escalonável no ambiente de execução, de modo similar ao conceito de objeto ativo. Por definição do perfil UML-TR, objetos com esta característica são associados com uma ou mais tarefas de escalonamento (*scheduling jobs*). Neste caso, existem três tarefas de escalonamento associadas, representadas no diagrama por *A, B* e *C*, que são justamente as etapas do protocolo descritas anteriormente. Cada uma delas representa um conjunto evento/ação ou, utilizando-se a notação do RT-UML, *«SAtrigger»* e *«SAresponse»*. Apesar da ocorrência destes eventos poderem estar relacionadas entre si, isto não é um requisito, tendo em vista que o primeiro evento é proveniente do próprio veículo e os outros podem vir de outros veículos. Com base na dinâmica do sistema é possível limitar o padrão de ativação das atividades. Por isso, caracterizou-se a ocorrência das mesmas como *burst*, ou seja, cuja ocorrência não é exatamente conhecida, porém deve se manter na faixa especificada.

No presente estudo as atividades são decompostas em um conjunto de ações *«SAaction»*, que também incorporam parâmetros que caracterizam a sua execução.

Destaca-se também neste diagrama a presença do estereótipo *«SAresource»*, associado com a classe *StateProcessor*. Revisando este conceito, verifica-se que ele representa um elemento cujo acesso deve ser feito com o auxílio de um mecanismo de controle sobre inversões de prioridade. Como este recurso possui capacidade igual a 1 (*SAcapacity = 1*), somente uma atividade poderá acessá-lo num determinado instante.

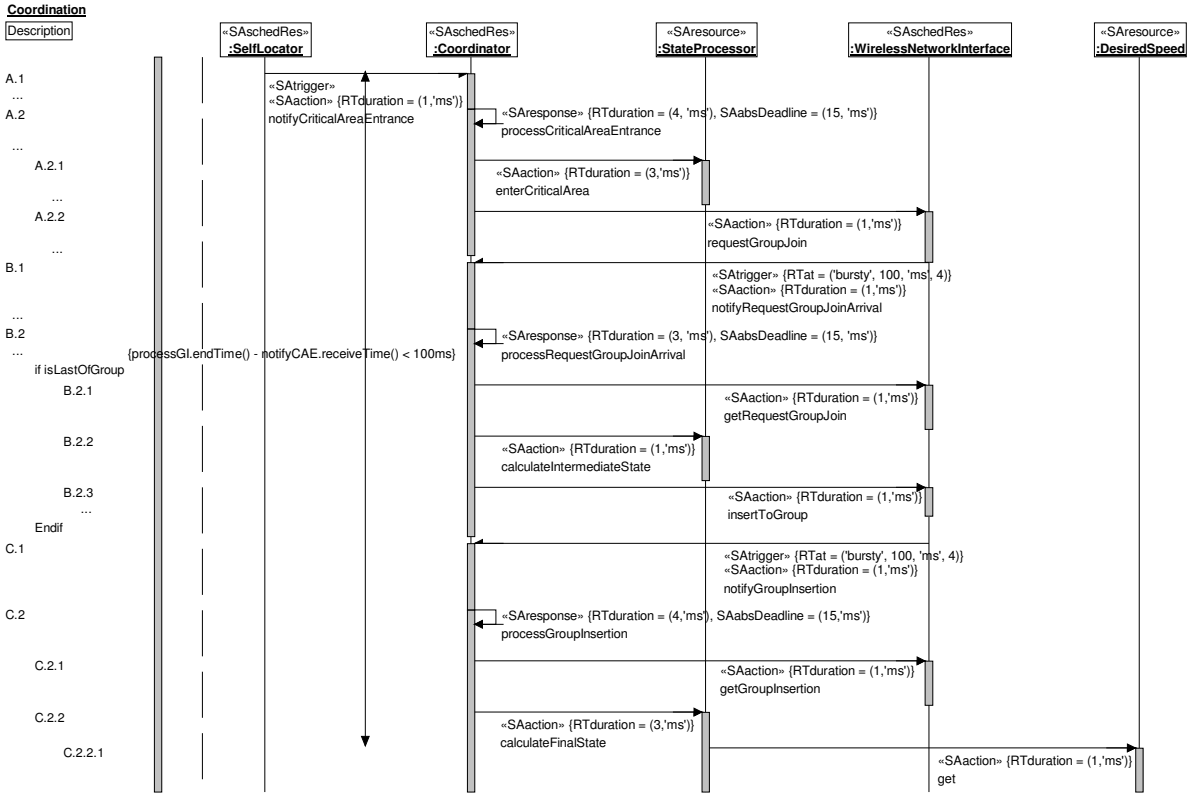


FIGURA 7.8 – Diagrama de seqüência do algoritmo de cooperação

Uma alternativa ao Diagrama de Colaboração para mostrar a interação entre os objetos que compõem uma dada funcionalidade é o Diagrama de Seqüência. Este último tem como principal vantagem o fato de ordenar a ocorrência dos eventos ao longo do tempo, deixando explícita a sua ordenação e facilitando a identificação do tempo decorrido. No caso do algoritmo de colaboração, uma vantagem em se utilizar um diagrama de seqüência para representá-lo é possibilitar a inserção de requisitos temporais tipo fim-a-fim. Por exemplo, nos casos onde o próprio veículo identifica a sua entrada numa região crítica e executa o algoritmo de cooperação, consegue-se inserir uma restrição desde a chegada do evento que dispara a ação inicial até o final da última ação. Na FIGURA 7.8 é possível expressar esta restrição no diagrama de seqüência utilizando recursos do *framework* para modelagem de tempo do perfil UML-TR: "*processGI.endTime() – notifyCAE.receiveTime() < 100ms*". Esta restrição pode ser inserida na forma de comentário no diagrama de colaboração, conforme mostrado na FIGURA 7.8, todavia a sua utilidade não fica tão explícita como se fosse utilizado um estereótipo para tanto.

Terminada a etapa de definição detalhada das atividades, obtêm-se as classes e as suas respectivas associações, constituindo assim o diagrama de classes do sistema. Este diagrama apresenta uma visão estática do sistema, sendo esta uma informação útil na geração de código da aplicação. O diagrama de classes do sistema *Kurt* é formado por 28 classes, sendo que destas, 5 são elementos de interface, 7 são recursos escalonáveis e as 16 restantes são usadas para armazenar dados, sendo 10 de acesso compartilhado. A FIGURA 7.9 exibe o diagrama de classes desenvolvido.

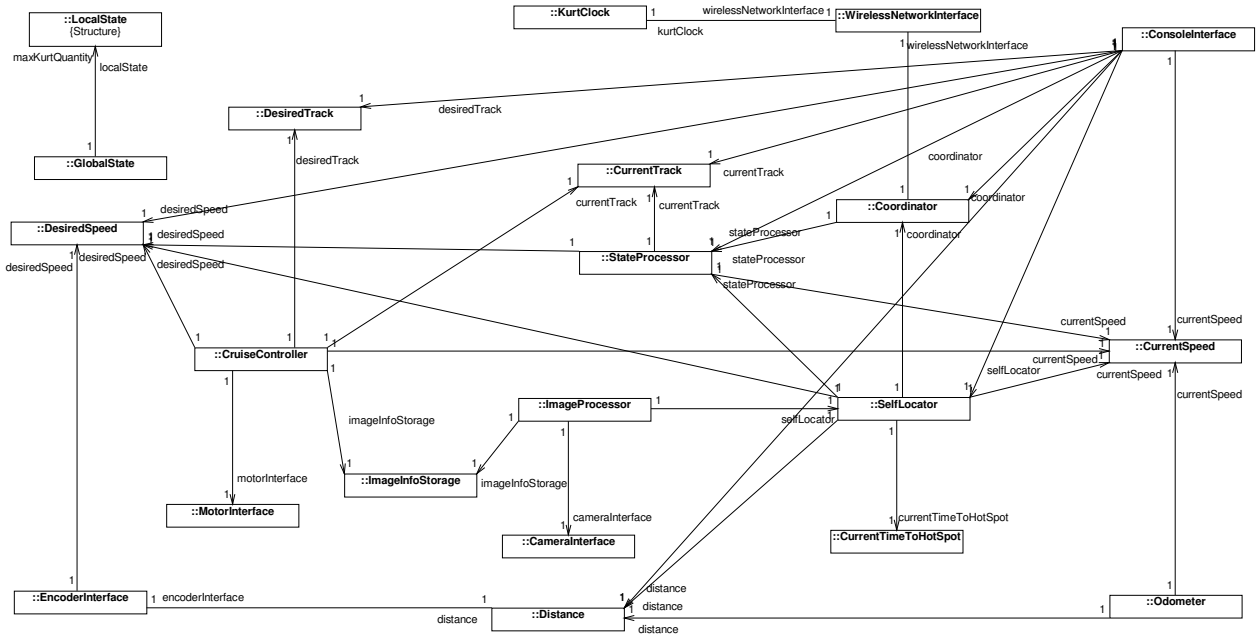


FIGURA 7.9 – Diagrama de classes do sistema *Kurt*

7.3 Mapeamento para o Ambiente de Programação

O mapeamento do modelo para o ambiente de programação se deu conforme as definições feitas no capítulo 4, e discutidas também em [HÖL 2002] e [BEC 2002]. Com isto, a estrutura estática do sistema, definida pelo diagrama de classes, foi mapeada para uma aplicação C++. Além disso, os estereótipos e marcas provenientes do perfil UML-TR foram implementados através de construções definidas na TAFT-API descrita no capítulo 6. Neste estudo, utilizou-se basicamente as definições de recurso escalonável, recurso compartilhado, tarefa escalonável e ações, cujo mapeamento pode ser resumido da seguinte forma:

1. Os *«TAtaskPair»* foram mapeados para *RealtimeThread*;
2. Os *«SAresource»* foram mapeados para classes Java associadas com um objeto da classe *MutexControl*;
3. As tarefas escalonáveis (*«SAtrigger»* + *«TAmainPart»*), quando ativadas por um relógio, foram mapeadas para o corpo principal da *thread* tempo real (método *run()*);
4. As EPs, representadas pelo estereótipo *«TAexceptionPart»*, são mapeadas para o trecho de exceção no corpo principal da *thread* tempo real;

Inicia-se a discussão do mapeamento efetuado pela estrutura estática do sistema, representada pelo diagrama de classes na FIGURA 7.9. O mapeamento efetuado seguiu o procedimento adotado no ambiente SIMOO-RT, o qual é detalhado em [BEC 99]. Este procedimento pode ser resumido da seguinte forma:

- Para cada classe, primeiramente é gerado o arquivo *header* (<classe>.h), que contém a definição de todos os seus atributos e métodos, e depois o arquivo contendo a definição dos métodos (<classe>.cpp);

- No arquivo *header* são expressas as associações da entidade: nas associações de agregação, os objetos agregados são definidos como instâncias pertencentes à classe agregadora; já nas associações de conhecimento são utilizados ponteiros, que referenciam a classe do relacionamento; associações de especialização ou herança são expressas no formato padrão da linguagem C++;
- Além dos métodos definidos pelo usuário, as classes do modelo também possuem métodos responsáveis por estabelecer a ligação entre os objetos associados, os quais simplesmente inicializam os ponteiros que representam os relacionamentos; estes métodos são denominados, de modo genérico, *link<nome do relacionamento>*.

No estudo de caso apresentado, assume-se que todas as classes do sistema são agregadas de uma classe genérica denominada *Kurt*. Esta classe contém referências para todas as classes definidas no modelo, que são instanciadas com o construtor da classe *Kurt*. Criadas as instâncias, são chamados os métodos encarregados de estabelecer as ligações entre as classes associadas. O procedimento descrito é exibido na FIGURA 7.10. O disparo das *threads* responsáveis por prover as funcionalidades do sistema (*consoleInterface, imageProcessor, selfLocator* e *cruiseController*) é feito no método *run()* da classe *Kurt*.

```
public Kurt() {
    cameraInterface = new CameraInterface();
    coordinator     = new Coordinator();
// Criação das outras instâncias...

    coordinator.linkStateProcessor(stateProcessor);
    coordinator.linkWirelessNetworkInterface(wirelessNetworkInterface);
// Definição das outras ligações...

};
```

FIGURA 7.10 – Inicialização do sistema no construtor da classe *Kurt*

Para exemplificar o mapeamento dos requisitos provenientes do perfil UML-TR utilizados neste estudo de caso, discute-se inicialmente o diagrama de colaboração apresentado na FIGURA 7.5. A implementação desta colaboração é resumida da seguinte forma:

- O *stereotype* «SASchedulable» aplicado à classe *ImageProcessor* indica que a mesma é um recurso escalonável, com sua própria *thread* de controle; portanto a classe *ImageProcessor* é subclasse de *RealtimeThread* da TAFT-API.
- A classe *SystemClock* está inclusa na API, e os eventos de tempo gerados pela mesma estão implícitos no suporte à execução da *RealtimeThread*;
- O *stereotype* «SATrigger» aplicado à mensagem *processImage* com a marca *RTat=('periodic', 20, 'ms')* representa o disparo a cada 20 ms do método *processImage()* da classe *ImageProcessor*; nesta mesma mensagem também se utiliza o *stereotype* «SAResponse» com a marca *SAAbsDeadline=(20,' ms')* indicando que o tempo limite para execução completa desta seqüência de ações é idêntico ao respectivo período de disparo; já a marca *RTduration* indica o tempo de execução esperado para o conjunto de atividades, que é de 12 ms;

- A entidade[24] *ImageInfoStorage* foi implementada como uma classe passiva (sem *thread* própria de controle), utilizando os elementos da TAFT-API para promover o controle de acesso;
- O processamento de imagem realizado na operação *generateInfo()*, simula o reconhecimento da trilha e também a detecção das marcas de entrada e saída na região crítica; a detecção é feita com base nas informações provenientes do sistema de odometria;
- A mensagem assíncrona *notifyMarkDetected* enviada pela instância de *ImageProcessor* à instância da classe *SelfLocator* foi implementada como uma chamada ao método *notifyMarkDetected()*, que por sua vez dispara o evento *processMarkDetected* da classe *SelfLocator*;

A TAFT-API permite aos objetos escalonáveis (subclasses de *RealtimeThread*) informar ao escalonador suas condições de liberação através de parâmetros, ou seja, se a liberação para execução é periódica, aperiódica ou esporádica. O objeto ativo periódico da FIGURA 7.5 (instância da classe *ImageProcessor*) é então mapeado para uma *RealtimeThread* periódica, sendo esta característica definida através da definição dos *ReleaseParameters* da mesma como periódicos. No construtor da classe *ImageProcessor*, é criado um objeto *PeriodicParameters*, com o período de 20 ms, duração de 12 ms e deadline de 20 ms, sendo o mesmo atribuído ao objeto de *RealtimeThread* que está sendo construído, conforme pode ser visto em destaque no código apresentado na FIGURA 7.11. Informações sobre importância também podem ser passadas ao construtor da *RealtimeThread*, conforme também mostrado na mesma figura.

```
ImageProcessor::ImageProcessor(): RealtimeThread{
    new ImportanceParameters(NORM_IMPORTANCE/*SAAction.SAImportance*/ ),
    new PeriodicParameters( 0 /*ReleaseTime*/, 20/*SATrigger.Period*/,
                            12 /*SAResponse.RTduration*/,
                            20 /*SAResponse.SAAbsDeadline*/)
    );

}
```

FIGURA 7.11 – Construtor da classe *ImageProcessor*

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 *ImageProcessor* corresponde a um laço no qual é simulado o processamento de imagem, seguidos da espera pelo novo período, conforme pode ser visto em destaque na FIGURA 7.12. Esta figura também destaca as macros utilizadas para representar a separação entre os código da parte principal (MP) e o código de tratamento de exceção (EP).

Na colaboração do processamento de imagens, a instância da classe *ImageInfoStorage* representa a entidade responsável por manter os dados obtidos do processamento, sendo esta entidade utilizada pelos demais objetos para armazenamento e consulta. Este elemento é modelado como um tipo de recurso protegido, no qual os acessos concorrentes seguem uma disciplina, delimitando seu acesso com chamadas aos

[24] Entidades são classes cujo papel principal consiste em armazenar dados [GOM 2000].

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 7.13, no qual o controle de acesso ao recurso protegido é definido através da classe *MonitorControl* da TAFT-API.

```
void ImageProcessor:run(){
    while( m_isRunning ) {
        waitForNextPeriod();
        BEGIN_MAIN_PART
            processImage();
        END_MAIN_PART
        BEGIN_EXCEPTION_PART
            //código da EP
        END_EXCEPTION_PART
    }
}
```

FIGURA 7.12 – Método *run()* da classe *ImageProcessor*

```
class ImageInfoStorage: public Mutex {
    ...
};

ImageInfoStorage::ImageInfoStorage () {
    MutexControl *myMControl = new MutexControl;
    myMControl->setMonitorControl( this,
                new PriorityInheritance()/*SAResource.SAAccessControl*/ );
    }
    // ...
}

void Data::store(Imageinfo &i){
    lock()
    ...
    unlock();
}
```

FIGURA 7.13 – Implementação de um recurso protegido

Para finalizar esta seção, discute-se o mapeamento do diagrama de colaboração da FIGURA 7.7. Para implementar a colaboração foi criada uma classe denominada *EventChannel*, responsável por implementar o broadcast das mensagens enviadas pelos *Kurt* através da instância da classe *WirelessNetworkInterface*.

O chamado *EventChannel* simula a propagação de mensagens através de *broadcasts*. Para tanto, o mesmo mantém uma lista contendo referências para os elementos *WirelessNetworkInterface* de cada *Kurt* presente no sistema. Com isto, cada vez que uma mensagem é enviada ao *WirelessNetworkInterface* ela é automaticamente repassada ao *EventChannel*, que por sua vez é encarregado de propagá-la a todos os outros *Kurts*.

8 Arquitetura para Validação dos Requisitos Temporais

A monitoração é uma das técnicas utilizadas para validar a execução de aplicações tempo real. Esta abordagem vem sendo empregada com sucesso em diversos projetos, como, por exemplo, em [CHO 91] e em [LAN 92]. Contudo, apesar de convenientes para os seus propósitos, tais projetos não permitem aos usuários concluir diretamente sobre as características orientadas a objetos da aplicação e também sobre as suas restrições temporais. Uma alternativa para tratar esta restrição foi proposta em [BEC 2001], permitindo ao projetista a monitoração de eventos relacionados com os objetos de classe, como a invocação de métodos e a atualização de atributos. Apesar disso, a validação dos requisitos temporais definidos durante a modelagem continua dificultada pelo fato de que o experimentador se deparar com muitos dados, difíceis de serem avaliados qualitativamente.

Com o objetivo de suprir a carência verificada, apresenta-se neste capítulo uma arquitetura para verificar o cumprimento das restrições temporais especificadas durante a modelagem. O objetivo é facilitar a detecção de eventuais falhas no sistema, sejam elas falhas de projeto ou causadas por anomalias durante a execução [WIL 2000a]. O processo de validação na arquitetura proposta é dividido em três etapas: (*a*) geração do conjunto de requisitos temporais a serem validados; (*b*) monitoração do ambiente de execução; (*c*) validação dos requisitos. Esta arquitetura combina uma ferramenta voltada para a validação de requisitos temporais em protocolos de comunicação, desenvolvida no âmbito da dissertação de mestrado de Wild [WIL 2000], com o ambiente de monitoração descrito em [BEC 2001], que é baseado no trabalho de Gergeleit [GER 2001]. A inovação do ambiente de monitoração presente na arquitetura proposta está no fato do mesmo ter sido portado para o SO RTLinux, plataforma base do ambiente de execução desenvolvido nesta tese descrito no capítulo 5. Nas próximas seções a arquitetura proposta é descrita em detalhes.

8.1 Monitoração da Aplicação

Para monitorar o ambiente de execução desenvolvido utilizou-se a ferramenta Jewel++ [GER 2001], que é capaz de armazenar eventos provenientes do SO (e.g. interrupções e chaveamentos de contexto), juntamente com eventos definidos pelo usuário. De maneira genérica, a ferramenta Jewel++ é formada por três componentes, conforme exibido na FIGURA 8.2: um *driver* específico para o SO, uma infra-estrutura de comunicação e um módulo de interface com o usuário.

O *driver* é usado para instrumentar o *kernel* do SO a fim de interceptar as interrupções e os chaveamentos de contexto. Diferentemente das implementações anteriores [LAN 92], onde o *driver* se comunicava com o SO através de *hooks*, no presente trabalho o mesmo foi implementado diretamente no código fonte do escalonador TAFT, uma vez que o mesmo se encontrara disponível. Os eventos capturados são armazenados em descritores, que contém informações sobre o evento, o processo do SO que o originou, uma marca de tempo de alta precisão (o contador interno do processador *Intel Pentium*, com resolução < 10 ns), e também um parâmetro

adicional (presente somente nos eventos definidos pelo usuário). A FIGURA 8.1 mostra a estrutura do descritor de eventos gerados.

Cod. Evento	*Ident. de Processo*	*Marca de Tempo*	*Parâmetro*
(unsigned long)	*(unsigned long)*	*2 x (unsigned long)*	*(unsigned long)*

FIGURA 8.1 - Estrutura do descritor de eventos gerados pelo *driver* de instrumentação do SO

A geração e o armazenamento dos eventos introduz uma baixa sobrecarga no sistema. A primeira é feita a partir de poucas linhas de código, que inclui a leitura do contador e a inserção das informações pertinentes no descritor (vide FIGURA 8.3). Os eventos são armazenados em um buffer, como fora exibido na FIGURA 8.2. No caso particular da implementação atual, o *buffer* pertence a uma estrutura interna de comunicação do RTLinux conhecida por FIFO tempo real (*RTfifo*). Esta estrutura serve como alternativa de comunicação entre processos executando no modo tempo real (sujeitos à política de escalonamento do TAFT) e os processos executando no modo usuário, constituído pelas tarefas não tempo real. Esta estrutura baseada em *RTfifo* permite uma comunicação eficiente entre o *driver* de instrumentação do SO e o módulo de captura e tratamento dos dados (interface com o usuário). Isto porque a comunicação é feita somente quando o processador se encontrar ocioso, i.e. sem tarefas tempo real para executar.

Conforme mencionado anteriormente, a interface com o usuário é responsável por ler os eventos da *RTfifo* e fazer o tratamento das mesmas. No momento, existem duas implementações alternativas para o módulo de captura e tratamento dos dados: com ou sem interface gráfica. A versão com interface gráfica é indicada para aqueles casos onde um conjunto pequeno de experimentos é efetuado, uma vez que requer um processo "manual" (com interferência do usuário) de abertura e fechamento de seção, visualização dos dados e salvamento de arquivos. Já a versão sem interface gráfica é recomendada quando existir um número elevado de experimentos, que podem ser programados através de *scripts* e executados de maneira autônoma, i.e. sem necessitar da interferência do usuário.

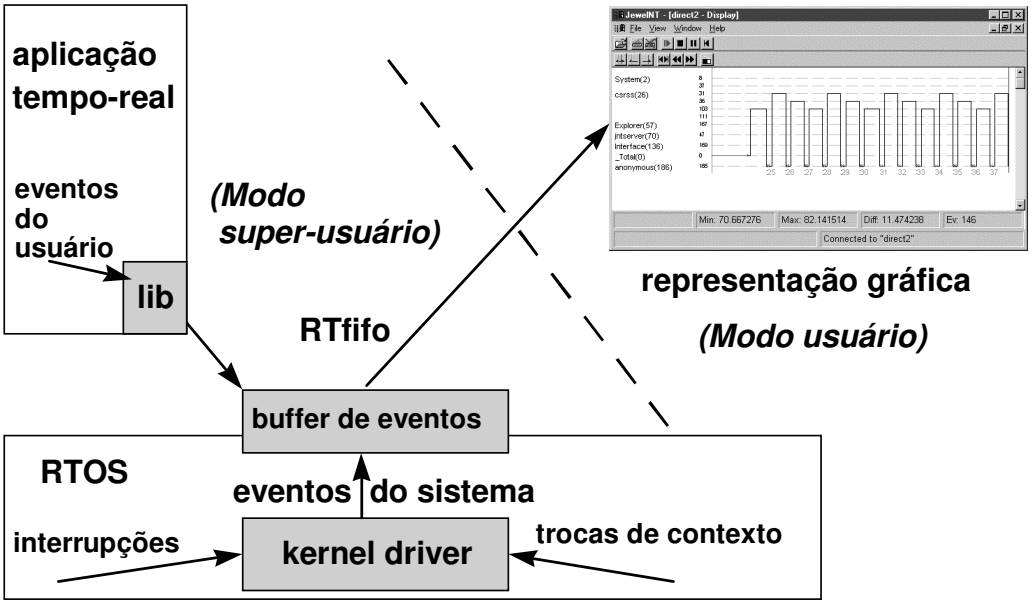


FIGURA 8.2 - Fluxo de dados na ferramenta Jewel++

```
rtl_gettime( &ts);
if (timespec_ge(&ts, &s->rtl_current->dl.it_value)) {
    ev.type = JEWEL_EV_USER+1;
    rdtsc(ev.time.time_l, ev.time.time_h);
    ev.thread_id = ev.proc_id = s->rtl_current;
    ev.param = s->rtl_current;
    rtf_put(3, &ev, sizeof(ev));
}
```

FIGURA 8.3 - Código para geração de evento a ser monitorado

Além das funcionalidades mencionadas, a versão com interface gráfica oferece uma visualização alternativa para os dados monitorados. A mesma contém um diagrama de Gantt, onde na coordenada *y* são listadas as tarefas tempo real (ou TPs quando usados com o TAFT). Estas tarefas são identificadas por um número identificador no sistema (*process-ID*) ou opcionalmente por nomes definidos pelo usuário. Na coordenada *x* é mostrado o tempo global desde o início do experimento. Cada tarefa é representada por uma linha no diagrama, que será mostrada sempre que a tarefa estiver utilizando a CPU. Deste modo, é possível visualizar claramente as trocas de contexto efetuadas durante a execução da aplicação. Na FIGURA 8.4 ilustra-se o uso deste diagrama junto com o escalonador TAFT, destacando-se o momento em que um TP tem a sua MP abortada e a respectiva EP é invocada.

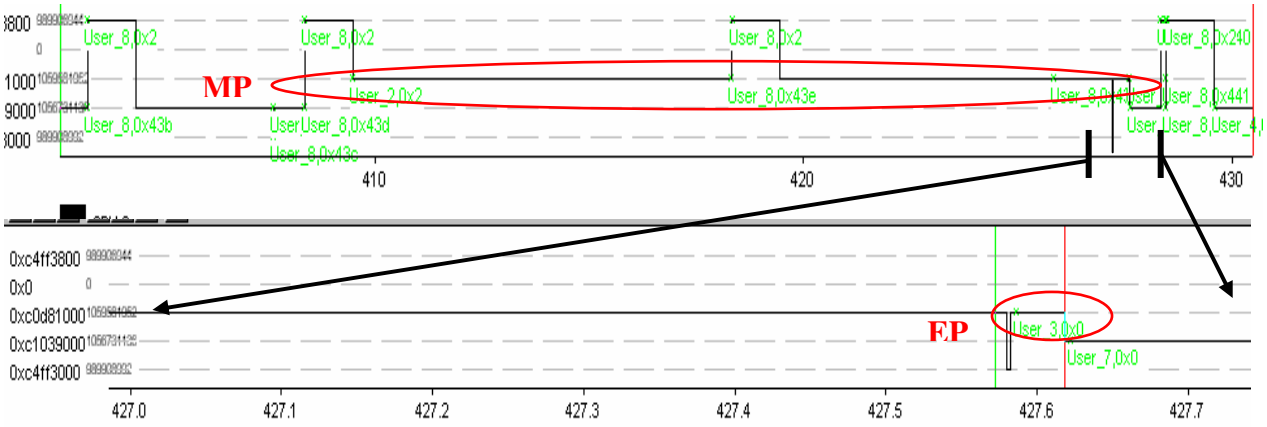


FIGURA 8.4 - Diagrama de Gantt do Jewel++

8.2 Geração e Validação dos Requisitos Temporais

Apesar da estrutura de monitoração oferecer uma visualização gráfica para os eventos, a mesma não possui qualquer tipo de validação em relação aos requisitos temporais modelados. Os requisitos temporais derivam das anotações feitas nos diagramas orientados a objetos construídos durante a fase de modelagem. Na arquitetura proposta, os requisitos do modelo UML devem ser transformados para à notação proposta por Pereira em [PER 95], que por sua vez é baseada na lógica temporal RTL (*Real-Time Logic*) [JAH 86]. O uso desta notação é justificado pelo fato da mesma

expressar de forma quantitativa, sucinta e de maneira declarativa, uma série de relações temporais entre eventos pontuais, isto é, que ocorrem em um instante de tempo, através de predicados simples.

As relações são expressas em alto nível, isto é, seu significado é similar ao de construções em linguagem natural, utilizando predicados como "antes" ou "ciclicamente" (vide TABELA 8.1). Conforme descrito em [WIL 2000], estas relações podem ser expressas de forma a incluir uma tolerância para a solução, isto é, a relação será satisfeita por um evento que ocorra dentro de um intervalo de tempo, e não somente em um instante preciso de tempo. As relações podem ser expressas de maneira absoluta (contagem de eventos e tempo a partir de um instante inicial) ou de maneira relativa (contagem a partir do tempo ou evento atual).

TABELA 8.1 - Notação temporal utilizada

Relação Temporal	Expressão da Relação	8.2.1.1..1 Exemplo
Seqüência temporal	(<ev1>AFTER<ev2>) <intervalo>	(Ligar_bomba AFTER Ler_sensor) <1s, 2s>
	(<ev1>BEFORE<ev2>) <intervalo>	(Ler_sensor BEFORE Ligar_bomba) <1s, 2s>
Sincronização	(SYNCHRON<lista_ev>) <duração>	(SYNCHRON sinal1, sinal2, sinal3) <500ms>
Periodicidade	(CYCLIC<ev1>, <ciclo>, <jitter>)	(CYCLIC AtualizaDisplay, 75ms, 10ms)

Através de um estudo de caso efetuado [BEC 2001b], verifica-se que as mesmas são bastante adequadas para expressar os requisitos temporais definidos durante a modelagem. Neste âmbito, o predicado mais trivial é o (CYCLIC <ev1>, <ciclo>, <jitter>), uma vez que o mesmo pode ser usado para representar a ocorrência de um «*SATrigger*» periódico, onde **<*ev1*>** representa o nome do método e **<*ciclo*>** denota o seu período (representado pela marca *RTat*).

Diferentemente de CYCLIC, que deriva de um único evento, os outros predicados estão relacionados com a ocorrência de dois ou mais eventos. Por exemplo, o predicado (SYNCHRON<lista_ev>) <duração> exige que todos os eventos da lista ocorram de forma síncrona, com o tempo máximo, expresso em <duração>, decorrido entre o primeiro e o último evento. Este predicado é útil para relacionar os eventos que fazem parte de uma «*SAResponse*».

Por sua vez, os predicados AFTER e BEFORE estabelecem relações básicas de seqüência. O primeiro, cuja sintaxe é (<ev1> AFTER <ev2> <Tmin, Tmax>), indica que o evento <ev1> deve ocorrer após o evento <ev2>. Além disso, o intervalo especifica quantitativamente uma restrição para esta ocorrência: o evento <ev1> deve ocorrer pelo menos tanto tempo após o extremo inferior do intervalo, e no máximo tanto tempo até o extremo superior do intervalo, iniciado com a ocorrência de <ev2>. O requisito (Ligar_bomba AFTER Ler_sensor) <1s, 2s> significa que o evento denominado Ligar_bomba deve ocorrer após o evento Ler_sensor. Também é exigido que o evento Ligar_bomba ocorra pelo menos 1s após, e no máximo 2s após Ler_sensor. A relação de seqüência BEFORE é construída de maneira similar, onde o primeiro evento deve ocorrer antes do segundo, dentro do intervalo especificado. Verifica-se que estes requisitos são úteis tanto para representar as seqüências de execução delimitada nos diagramas de seqüência, quanto eventuais

restrições do diagrama de transição de estados. Este requisito pode ser utilizado para validar o estereótipo «*SAPrecedes*» do RT-UML.

Os requisitos temporais obtidos a partir dos diagramas modelados são então expressos de acordo com a notação proposta e, juntamente com os dados provenientes da monitoração, servem de entrada para um módulo de validação, descrito detalhadamente em [WILD 2000]. A apresentação dos resultados do processo de monitoração e validação é feita através de diferentes gráficos: gráficos de área, histogramas de validação e diagramas de Gantt validados.

8.3 Exemplos de Uso da Arquitetura Proposta

8.3.1 Análise do Comportamento Temporal do Ambiente TAFT/RTLinux

Neste estudo de caso utiliza-se a arquitetura proposta para fazer a validação do ambiente TAFT/RTLinux. Os resultados obtidos na avaliação do ambiente TAFT/RTLinux (discutidos no capítulo 5) são utilizados como entrada neste estudo. De maneira mais detalhada, os dados obtidos através da monitoração do *benchmark* Hartstone são analisados através da ferramenta VCAT, usada para fazer a validação dos requisitos temporais. Apesar destes dados já terem sido alvo de uma análise qualitativa, o que permitiu a geração dos resultados discutidos na seção 5.4, aqui os mesmos são novamente interpretados, sob a perspectiva dos requisitos temporais a serem apresentados nesta seção, os quais foram definidos para permitir a caracterização das características temporais instrínsecas do ambiente TAFT/RTLinux. O procedimento para experimentação do ambiente proposto deu-se conforme descrito a seguir.

Inicialmente, configurou-se o conjunto de tarefas a serem executadas de acordo com o seu padrão de chegada, definido pelas séries PH e PN (vide TABELA 5.2), com a PDF utilizada para gerar os tempos de execução das MPs e também com o índice desejado para a utilização nominal de CPU. Optou-se por utilizar aqui a mesma configuração selecionada para análise na seção 5.4, i.e. a série PN com PDF Beta(2, 3).

Durante a realização dos experimentos, utilizou-se a versão sem interface gráfica do módulo de monitoração, discutido na seção 8.1. Para isto, desenvolveu-se um conjunto de *scripts* capazes de disparar o experimento e o monitor e também de coordenar o salvamento dos dados após o término do experimento. Os dados gerados, salvos em arquivos binários, são utilizados nesta seção como entrada para a análise efetuada.

A primeira análise efetuada mostra o comportamento do ambiente de execução proposto com o uso do EDF como algoritmo de escalonamento. Inicialmente é feita uma análise quantitativa sobre o número de tarefas efetivamente executadas em relação ao número previsto de execuções esperadas (vide TABELA 8.2). Escolheu-se a tarefa T0 como alvo deste estudo, sendo que os resultados obtidos são exibidos nos gráficos de torta da FIGURA 8.5. Estes gráficos exibem os valores absolutos de número de execuções esperadas e número de execuções finalizadas com sucesso, de forma análoga aos pontos desenhados no gráfico da FIGURA 5.13. O gráfico *a* representa o comportamento obtido para as utilizações efetivas de 84% e 95%. Já os gráficos *b, c* e *d* representam, respectivamente, as utilizações efetivas de 101%, 118% e 129%. O evento *T0-ready* representa o número de execuções esperadas para a tarefa *T0* e o evento *T0-end* representa o número de execuções finalizadas. Observa-se no gráfico *a* uma

situação ótima, i.e. exatamente o mesmo número de finalizações em relação às tarefas programadas. Nos outros gráficos é possível perceber através do percentual de ocorrências, uma diminuição no número de tarefas completadas com sucesso em relação ao número de tarefas programadas. Na legenda dos gráficos são mostrados os valores absolutos observados para cada evento.

TABELA 8.2 – Parâmetros associados com as tarefas da série PN

Id Tarefa	N. Execuções	Período	Tp.Execução
TP0	60	500,0 ms	80,00 ms
TP1	90	333,3 ms	53,28 ms
TP2	150	200,0 ms	32,00 ms
TP3	210	142,8 ms	22,85 ms
TP4	330	90,9 ms	14,54 ms

Os resultados apresentados na FIGURA 8.5, apesar de úteis para verificar o número de instâncias descartadas devido aos atrasos, não oferecem uma visão qualitativa sobre o comportamento do escalonador. Se apenas os resultados da análise absoluta fossem levados em consideração, ficaria a falsa impressão de que os experimentos com maior índice de utilização possuem um melhor desempenho, pois neste caso se aproximam do resultado ótimo (gráfico *a*). Mais especificamente, parece que os resultados do gráfico *d* são melhores do que os resultados do gráfico *c*, que por sua vez são melhores que os resultados do gráfico *b*. Somente uma análise qualitativa dos dados pode conduzir a resultados mais precisos.

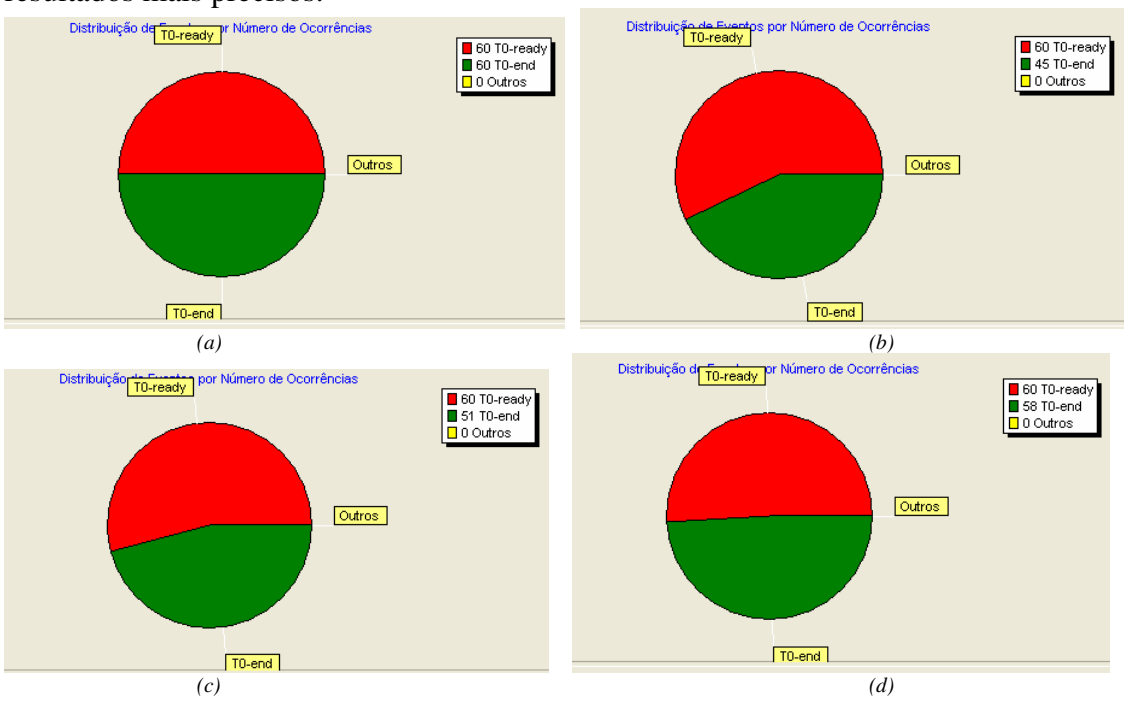


FIGURA 8.5 – Análise quantitativa do EDF, executando a série PN e a distribuição Beta(2, 3)

Assim, torna-se útil avaliar também as perdas de *deadlines*, que é justamente o parâmetro utilizado para se obter a taxa de sucesso das tarefas, discutido na seção 5.4. Nos experimentos utilizados, o próprio escalonador fora programado para fazer esta verificação, gerando eventos que indicavam a perda do *deadline*. Com o uso do VCAT, este procedimento se torna desnecessário, uma vez através das restrições temporais a

ferramenta é capaz de detectar as perdas de *deadline*. As restrições definidas para a realização desta análise indicam que o evento de término da tarefa (*TPx-endMP*), deve ocorrer após o inicio da mesma (evento *TPx-ready*), num intervalo de tempo entre o ECET da tarefa e o seu *deadline* (que é igual ao seu período). As restrições em questão são mostradas abaixo.

(‘TP0-endMP’ AFTER ‘TP0-ready’) [80000, 500000]
(‘TP1-endMP’ AFTER ‘TP1-ready’) [53280, 333333]
(‘TP2-endMP’ AFTER ‘TP2-ready’) [32000, 200000]
(‘TP3-endMP’ AFTER ‘TP3-ready’) [22850, 142857]
(‘TP4-endMP’ AFTER ‘TP4-ready’) [14540, 90909]

Os resultados da análise qualitativa, i.e. da validação dos requisitos temporais especificados, podem ser verificados de duas formas distintas: através de histogramas e através de diagramas de Gantt. Escolheu-se a tarefa TP0 para discutir os resultados obtidos na análise efetuada.

Na FIGURA 8.6 são exibidos os histogramas representando o comportamento da tarefa TP0 nos mesmos índices de utilização efetiva observados para as análises da FIGURA 8.5. Aqui, porém, o histograma *a* representa somente os resultados obtidos com utilização efetiva de 95%. O resultado da avaliação obtida para a utilização efetiva de 84% não é mostrado porque não foram detectadas perdas de *deadline*. Nestes histogramas fica claro que quanto maior o índice de utilização do conjunto de tarefas, maior é o número de *deadlines* perdidos, representados nos histogramas por aquelas ocorrências observadas fora da restrição imposta (barra em negrito plotada acima da linha de tempo). Desta forma, tem-se uma visão mais realista a respeito do comportamento do escalonador sob índices de utilização mais elevados.

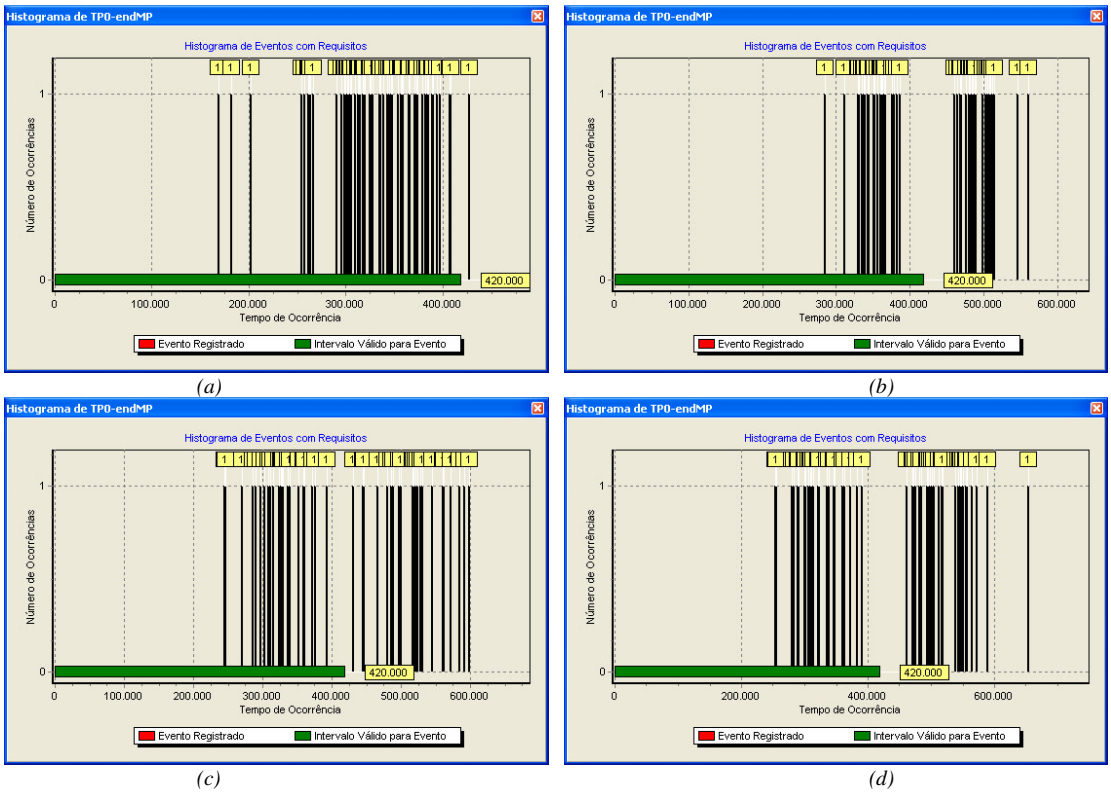


FIGURA 8.6 – Histogramas contendo os instantes de término da tarefa TP0

Outra forma de se analisar qualitativamente os requisitos temporais especificados é através de diagramas de Gantt. Este diagrama é uma ferramenta bastante útil para localizar no tempo o instante de ocorrência dos eventos especificados, bem como o atendimento ou não dos seus requisitos. Na FIGURA 8.7 o uso deste diagrama é exemplificado. O mesmo representa o resultado das avaliações da restrição da tarefa TP0 para a configuração com utilização de 95%. Com isto é possível achar os instantes de ocorrência dos *deadlines*, que são justamente os instantes em que o evento aparece fora da restrição (barra em negrito abaixo do evento), sendo demarcados por um X. Neste exemplo, o evento 2 representa o instante em que a tarefa é finalizada. Percebe-se no diagrama violações de *deadline* nos instantes 2,1s e 3,1s (a unidade de tempo do gráfico é us).

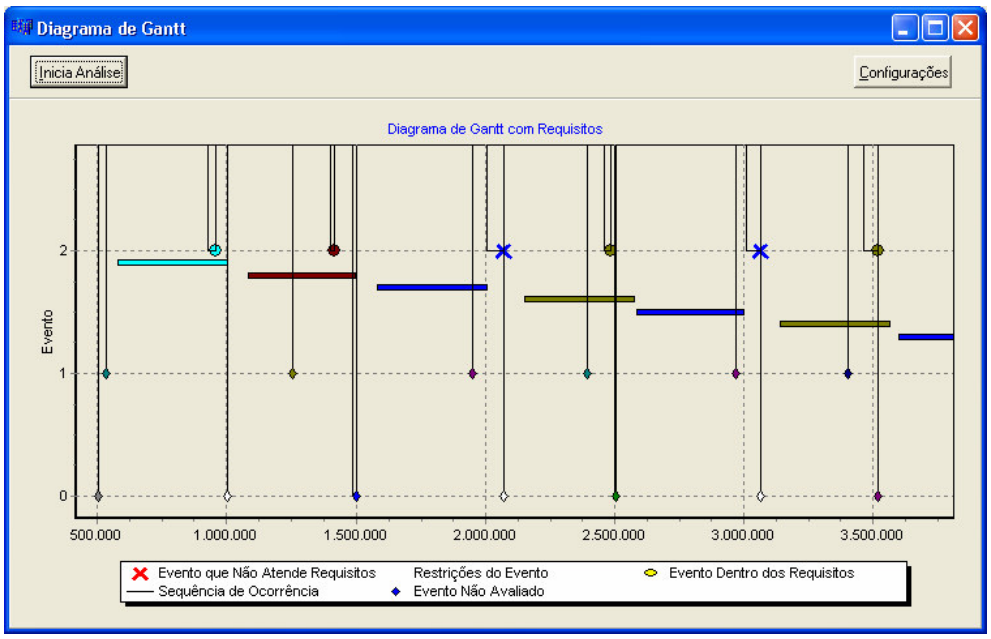


FIGURA 8.7 – Perdas de *deadline* exibidas no diagrama de Gantt do VCAT

Neste estudo de caso, realizou-se também uma análise do comportamento do ambiente de execução proposto executando o escalonador TAFT, na implementação sem o mecanismo de adaptação. Fez-se então uma análise quantitativa semelhante àquela mostrada na FIGURA 8.5, quanto se utilizou o escalonador EDF. Todavia, como aqui são escalonados TPs e não simplesmente tarefas, analisa-se o número de MPs e EPs completadas em relação ao número de ativações do TP, conforme mostrado na FIGURA 8.8. Os eventos *TP0-ready, TP0-endMP* e *TP0-endEP* representam respectivamente a ativação do TP0, o final da sua MP e o final da sua EP. Através destes gráficos percebe-se claramente o aumento no número de EPs (evento *TP0-endEP*) com o aumento no índice de utilização efetiva do conjunto de TPs.

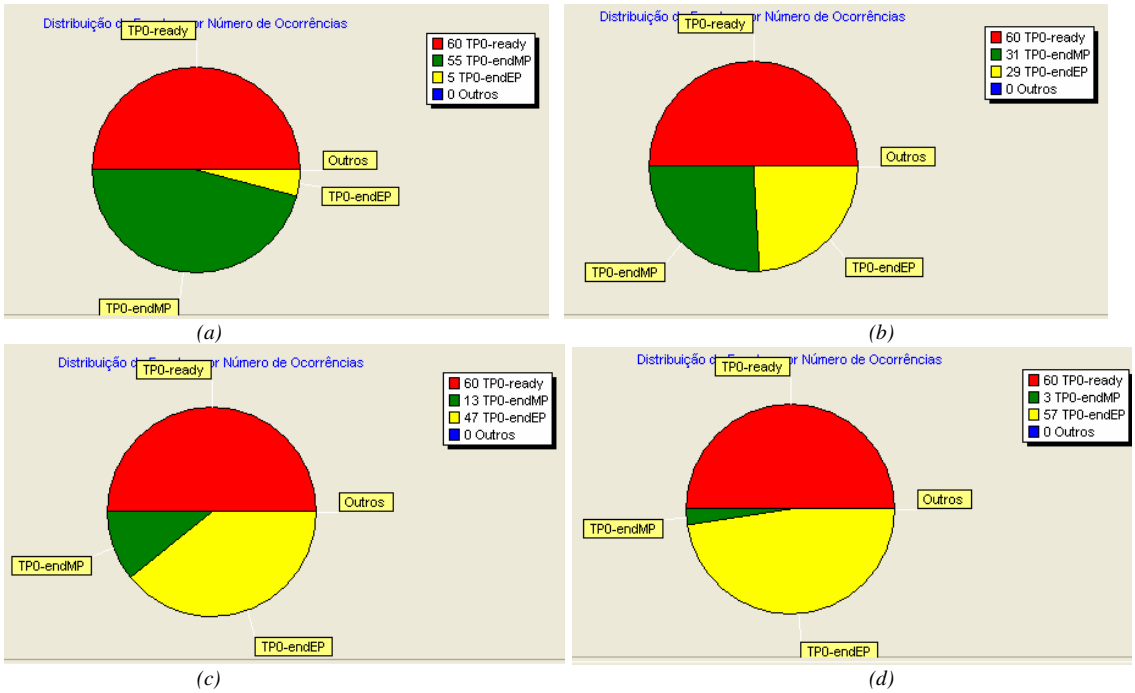


FIGURA 8.8 – Análise quantitativa do TAFT (sem o mecanismo de adaptação)

Na experimentação do escalonador TAFT, uma análise interessante é fazer a verificação da precisão do mesmo em relação às ativações das EPs. Esta análise permite concluir se os TPs estão realmente cumprindo com os seus *deadlines*, além é claro de verificar a precisão do escalonador na ativação das exceções. Para tanto, montou-se um conjunto de restrições para indicar que a exceção deve ocorrer antes do *deadline* e após o *deadline* menos o ECET associado com a MP. Nota-se que apesar deste não ser um requisito de escalonamento (visto que o LRT irá ordenar as EPs por ordem de prioridade de chegada), utilizou-se este valor por ser necessário um limite inferior para o requisito (vide tabela exibidos abaixo). A FIGURA 8.9 ilustra os histogramas que descrevem o comportamento verificado para TP0, nas utilizações efetivas de *(a)* 95%, *(b)* 101%, *(c)* 118% e *(d)* 129%. Conforme pode ser identificado nas figuras, à medida que a utilização aumenta, aumenta também o número de exceções. Entretanto, como pode ser observada, as mesmas sempre ocorrem dentro do intervalo definido, comprovando assim a precisão do mecanismo de escalonamento implementado.

(‘TP0-endEP’ AFTER ‘TP0-ready’) [420000, 500000]
(‘TP1-endEP’ AFTER ‘TP1-ready’) [265000, 333333]
(‘TP2-endEP’ AFTER ‘TP2-ready’) [168000, 200000]
(‘TP3-endEP’ AFTER ‘TP3-ready’) [119000, 142857]
(‘TP4-endEP’ AFTER ‘TP4-ready’) [76650, 90909]

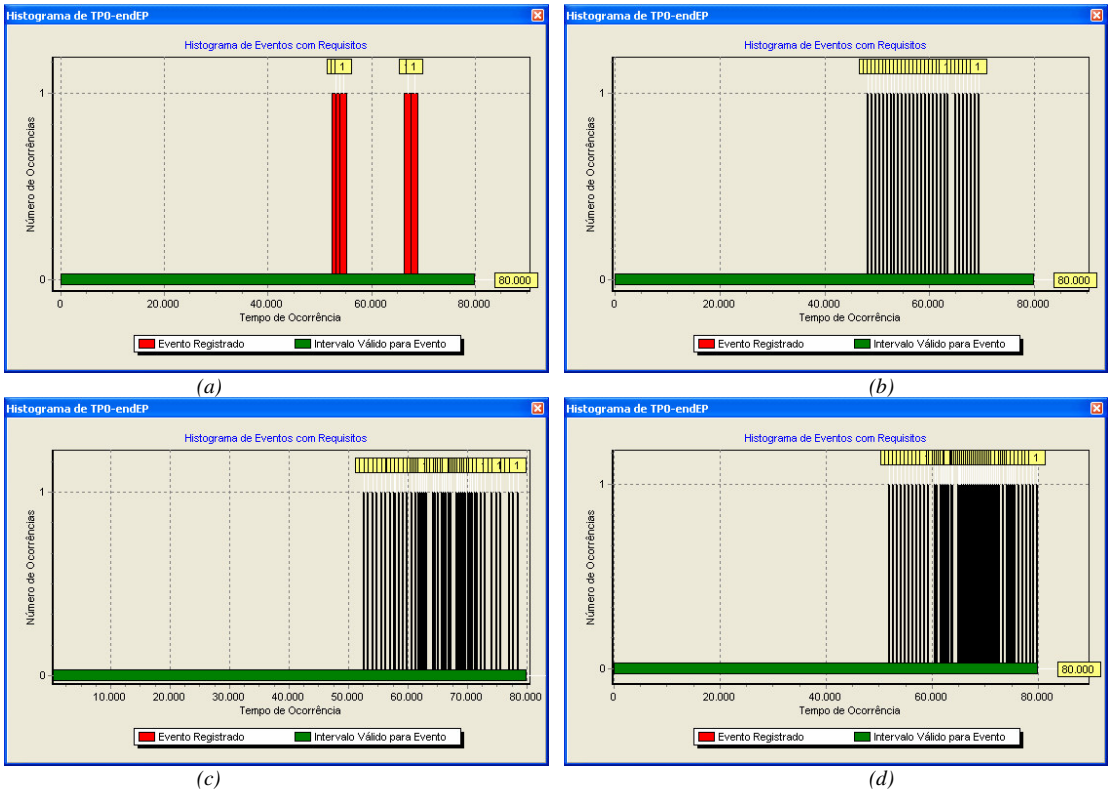


FIGURA 8.9 – Comportamento das EPs do TP 0

Por fim, realiza-se uma análise do diagrama de Gantt gerado para o TP0, com uma utilização efetiva de 95%. Através deste diagrama, é possível detectar o momento em que ocorreram exceções (evento 2) e também se as mesmas ocorrem dentro do intervalo estabelecido pelas restrições definidas. A FIGURA 8.10 ilustra o diagrama gerado.

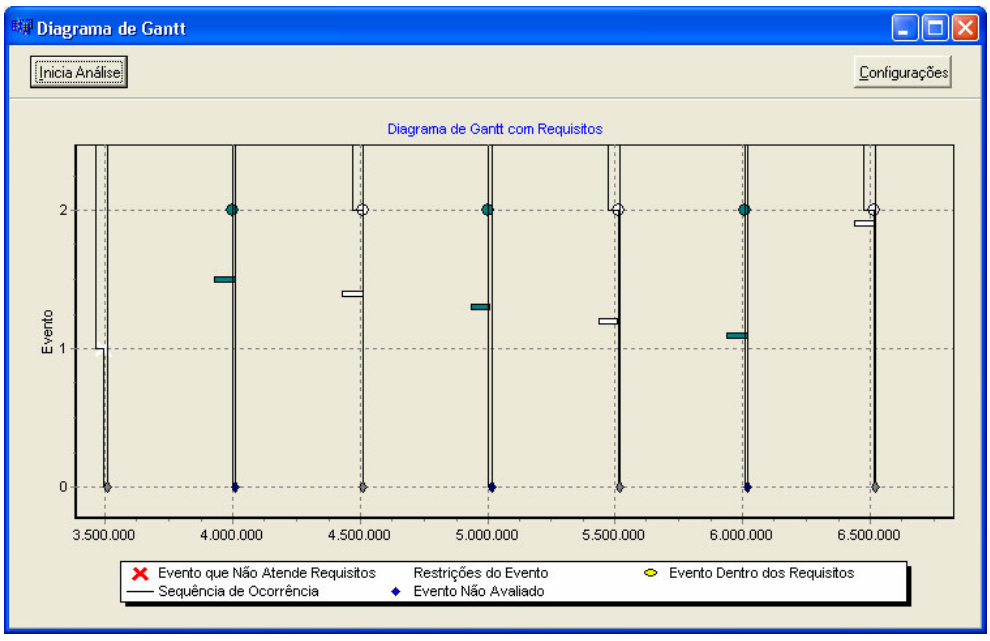


FIGURA 8.10 – Diagrama de Gantt para o TP0 com utilização efetiva de 95%

8.3.2 Sistema de Controle de Nível

Nesta seção descreve-se o segundo estudo de caso elaborado para testar a arquitetura de validação proposta. Utilizou-se um protótipo de uma planta industrial composto por um tanque, um sensor de nível e uma bomba. Este sistema, projetado originalmente com o uso da ferramenta SIMOO-RT (vide em [BEC 99]), foi remodelado utilizando-se a ferramenta *Real-Time Studio*, a fim de permitir a caracterização do modelo com os requisitos provenientes do perfil UML-TR.

O foco de interesse neste estudo de caso é o comportamento temporal da classe *Tank*, descrito com o uso de um diagrama de seqüência, o qual é exibido na FIGURA 8.11. Este diagrama mostra o comportamento do tanque durante um estado "Normal" de operação, i.e., onde não foram detectados eventuais erros. Definiu-se uma estratégia de controle para manter o nível de líquido no tanque conforme definido pelo usuário. Também foi definido que o controle poderia assumir três estados de operação, a constar:

1. **Normal**: indica que o sistema está operando normalmente;
2. **Exception**: acionado quando a instância da classe *Tank* não estiver conseguindo se comunicar com a instância da classe *Sensor*;
3. **Alarm**: acionado quando a instância da classe *Tank* não estiver conseguindo se comunicar com a instância da classe *Pump*;

Os estados de operação são determinados através de um diagrama de transição de estados, conforme mostrado na FIGURA 8.12.

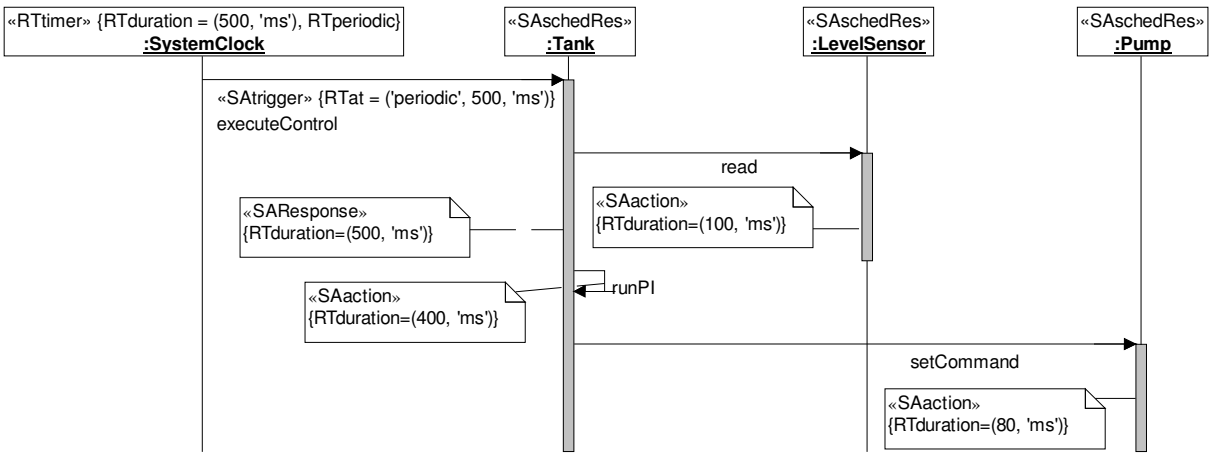


FIGURA 8.11 - Funcionalidade da classe *Tank* em estado "Normal" de operação

A partir do diagrama da FIGURA 8.11 selecionaram-se alguns eventos de interesse para serem monitorados, conforme definido na TABELA 8.3. Com isto, mapearam-se as restrições descritas no diagrama de seqüência que fazem uso dos eventos selecionados. O resultado deste mapeamento são as restrições mostradas na TABELA 8.4.

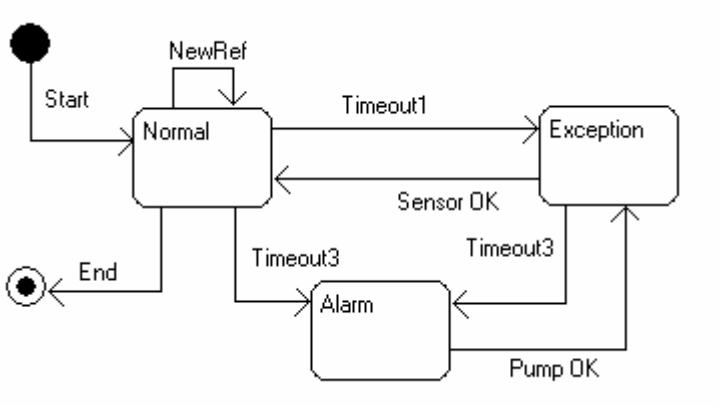


FIGURA 8.12 - Diagrama de transição de estados da classe *Tank*

TABELA 8.3 - Eventos de interesse nos experimentos

Evento	Descrição
E1	Execução do evento periódico que inicia o cenário
E2	RMI para ler o objeto sensor
E3	Retorno do objeto sensor
E4	Chamada local do método de controle
E5	RMI para método de comando
E6	Retorno do método de comando
E7	Retorno do método de controle
E8	Fim do cenário

TABELA 8.4 - Requisitos temporais gerados

Requisito temporal	Descrição
(CYCLIC E1@-1) [500, 10]	Execução do cenário (E1) será periódico, com período de 500ms e desvio (*jitter*) máximo de 10ms.
(E3@-1 AFTER E2@-1) [0, 100]	Sensor deve retornar o dado no máximo 100ms após a solicitação
(E7@-1 AFTER E4@-1) [0, 400]	Tempo máximo entre início e fim do método de controle
(E6@-1 AFTER E5@-1) [0, 80]	A bomba deve responder à solicitação de comando no máximo 80 ms após receber a chamada.

O programa gerado a partir do modelo desenvolvido foi configurado e executado (e monitorado) de três formas distintas, conforme descrição a seguir:

- Todos os objetos instanciados na mesma máquina;
- Com o objeto sensor instanciado em uma máquina remota;
- Semelhante à configuração anterior, porém com falha no meio de comunicação.

No próximo passo do estudo de caso, realizou-se a validação dos requisitos temporais (vide TABELA 8.4). Analisando o requisito 1, que define a periodicidade do cenário descrito no diagrama de seqüência da FIGURA 8.11 possui um comportamento semelhante nos três experimentos efetuados, o que mostra a adequação do ambiente de execução (neste caso o SO QNX) para tratar eventos tempo real. O período médio de ativação teve variação de 0,06 % (com variação de +/- 0,01%). O histograma da FIGURA 8.13 exibe as ocorrências da restrição 1 (número de ocorrências do evento em cada instante observado) no experimento 2.

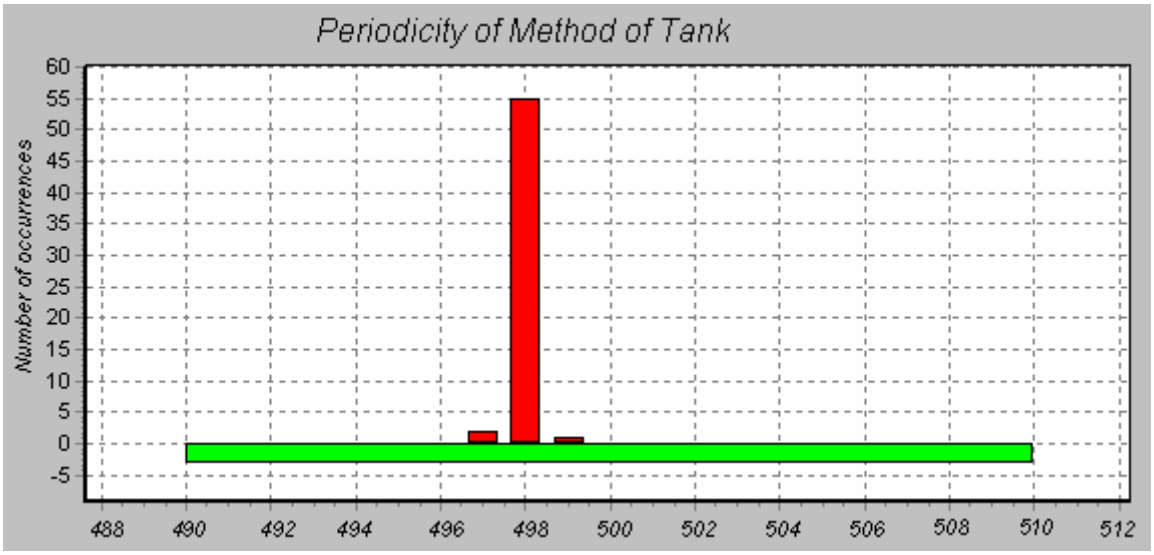


FIGURA 8.13 - Histograma com a periodicidade do cenário (restrição 1) no experimento 2

O comportamento da ação de controle (método *Control()*) apresentou-se em torno de 23% (15ms) mais demorado do primeiro para o segundo experimento, justificado pelo fato da leitura do sensor ser feita através de RMI. Este aumento no tempo pode ser claramente observado no diagrama de Gantt da FIGURA 8.14, onde se pode comparar a diferença entre os eventos E2 e E3 nos casos *a* (experimento 2) e *b* (experimento 1). Neste diagrama, as barras horizontais representam o intervalo de tempo válido para a ocorrência do evento.

Outro ponto analisado neste estudo de caso se concentra na falha induzida durante o terceiro experimento. A falha de comunicação propositalmente introduzida acaba por gerar um *timeout* no RMI *Read()* (evento E5), encarregado de ler o sensor de nível. Este evento é interessante por causar a transição do estado *Normal* para o estado *Exception*. Com isto, ressalta-se a atenção para uma eventual necessidade de se ter um novo conjunto de requisitos temporais. A FIGURA 8.15 exibe dois diagramas de *Gantt* que recebem a mesmo conjunto de dados como entrada, porém um contém os requisitos definidos para o estado *Normal* e o outro os requisitos para o estado *Exception*. O objetivo desta figura é o de realçar justamente o momento da transição entre os estados.

Os resultados obtidos neste estudo de caso demonstram que a arquitetura proposta se apresenta como uma boa alternativa para a realização de uma análise qualitativa dos

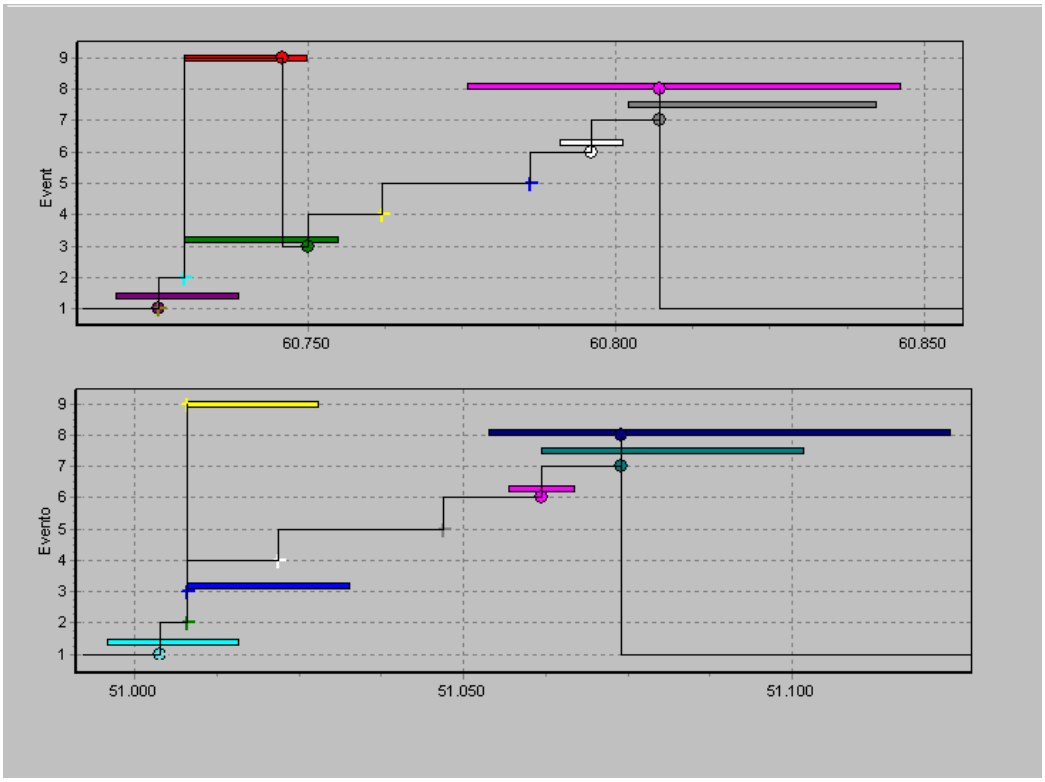


FIGURA 8.14 – Eventos E2 e E3 nos experimentos *a* (baixo) e *b* (cima)

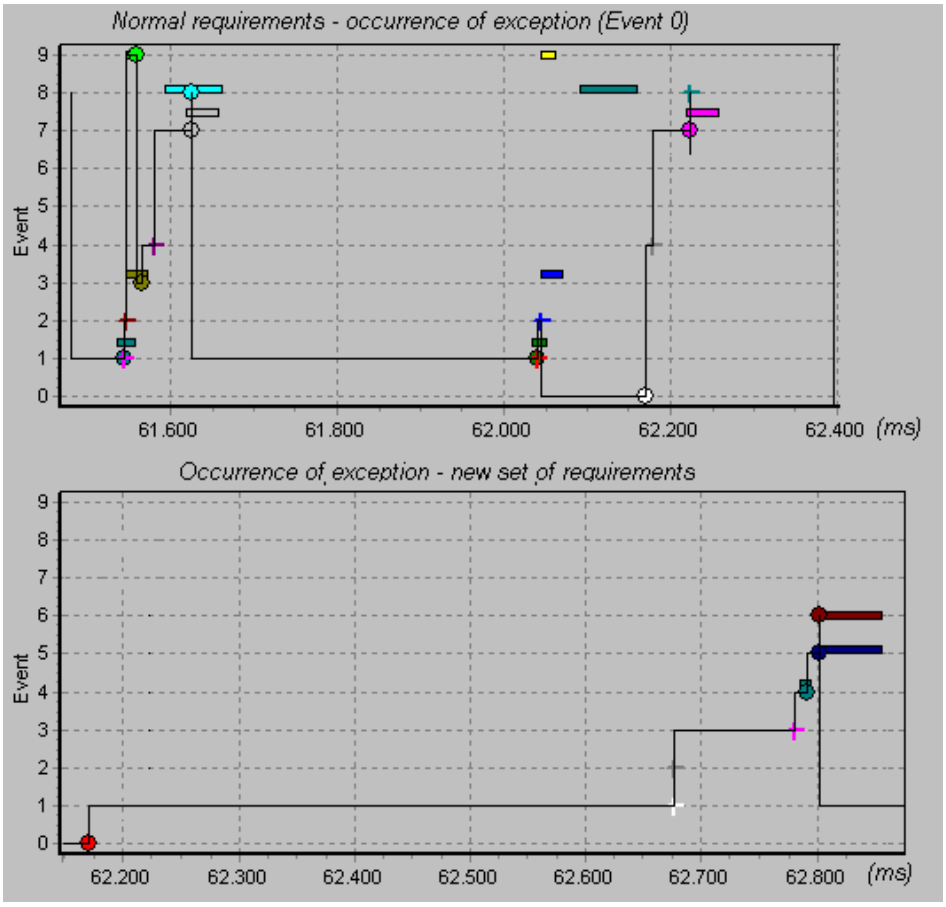


FIGURA 8.15 - Transição entre estados, causando mudança no conjunto de requisitos

sistemas projetados. Isto porque se permite a validação dos requisitos definidos durantea etapa de modelagem do sistema. Através das diversas opções de visualização oferecidas pela ferramenta a análise dos requisitos temporais torna-se intuitiva, facilitando seu entendimento e eventuais correções.

8.4 Considerações Adicionais

A arquitetura desenvolvida permite ao projetista fazer a validação dos requisitos impostos ao sistema durante a fase de modelagem. Isto é feito de maneira off-line, pois ocorre após a execução do sistema. Além da detecção de erros, esta análise proporciona ao projetista uma visão mais detalhada a respeito da relação temporal entre as tarefas. Por exemplo, o projetista pode vir a identificar a possibilidade de agrupar funcionalidades que executam periodicamente em uma mesma tarefa (ou TP), com o objetivo de minimizar as trocas de contexto e aumentar a eficiência do sistema.

Além deste tipo de validação, convém ressaltar que o objetivo da ferramenta de monitoração incorporada ao sistema proposta por Gergeleit [GER 2001] é permitir uma análise estatística a respeito dos tempos de execução das tarefas, a fim de realimentar o escalonador. Este elemento constitui o componente "*Time Aware*" do escalonador TAFT. Apesar de atualmente estes dois elementos não estarem integrados, esta integração é prevista como uma continuação deste trabalho.

9 O Método Proposto na Totalidade

Ao longo do texto foram apresentados de maneira individual os elementos que fazem parte da tese proposta: um método baseado no paradigma da orientação a objetos que aborda todo o ciclo de desenvolvimento de aplicações tempo real. Neste capítulo realiza-se uma discussão sobre o método proposto na sua totalidade, i.e. como as diversas etapas e componentes apresentados encontram-se relacionados. Além disso, discutem-se as possibilidades de integração das etapas e componentes em questão em uma ou mais ferramentas CASE.

9.1 Relacionamentos entre os Elementos do Método Proposto

O método proposto para abordar todo o ciclo de desenvolvimento de aplicações tempo real constitui-se dos seguintes elementos:

1. Etapa de modelagem usando UML-TR
2. Escalonador TAFT
3. Interface de programação para o TAFT (TAFT-API)
4. Etapa de mapeamento do modelo UML-TR para TAFT-API
5. Arquitetura para validação dos requisitos temporais

A etapa de modelagem consiste na criação de diagramas UML, os quais são decorados com os elementos do perfil UML-TR a fim de expressar as restrições – performance, escalonabilidade e tempo – associadas com a aplicação modelada. Os diagramas produzidos com o auxílio de uma ferramenta CASE são utilizados tanto por ferramentas de análise, e.g. análise de escalonabilidade, como por geradores automático de código.

O escalonador TAFT constitui-se no segundo componente do método proposto. O mesmo define um modelo de tarefas, o qual serve de entrada para o escalonador, um teste de aceitação e também um algoritmo de escalonamento. Apesar de não estar diretamente relacionado com a etapa de modelagem, o TAFT pode receber receber como entrada o modelo de objetos resultante da mesma.

A interface de programação para o TAFT, ou simplesmente TAFT-API, permite a mapear o modelo lógido de objetos resultante da etapa de modelagem para o modelo de escalonamento e execução baseado em threads do SO RTLinux, o qual fora utilizado como plataforma alvo para a validação do TAFT. Além disso, esta API permite elevar o nível de abstração das aplicações geradas, isolando em camadas mais abstratas os detalhes de programação do SO subjacente.

A etapa de mapeamento do modelo UML-TR para a TAFT-API consiste em um conjunto de definições. Estas definições estabelecem o relacionamento entre os estereótipos e marcas do perfil UML-TR e os elementos da TAFT-API capazes de proverem a sua implementação. Tal mapeamento constitui-se nas definições típicas de um gerador automático de código.

O último integrante do método proposto é a arquitetura para validação dos requisitos temporais. Esta arquitetura é capaz de prover uma análise qualitativa em relação ao cumprimento ou não dos requisitos temporais definidos durante a modelagem. Tal

análise é feita a partir dos dados obtidos através da monitoração do ambiente de execução TAFT, sendo os requisitos temporais derivados da fase de modelagem.

O diagrama apresentado na FIGURA 9.1 apresenta o relacionamento entre os diversos componentes que fazem parte do método proposto.

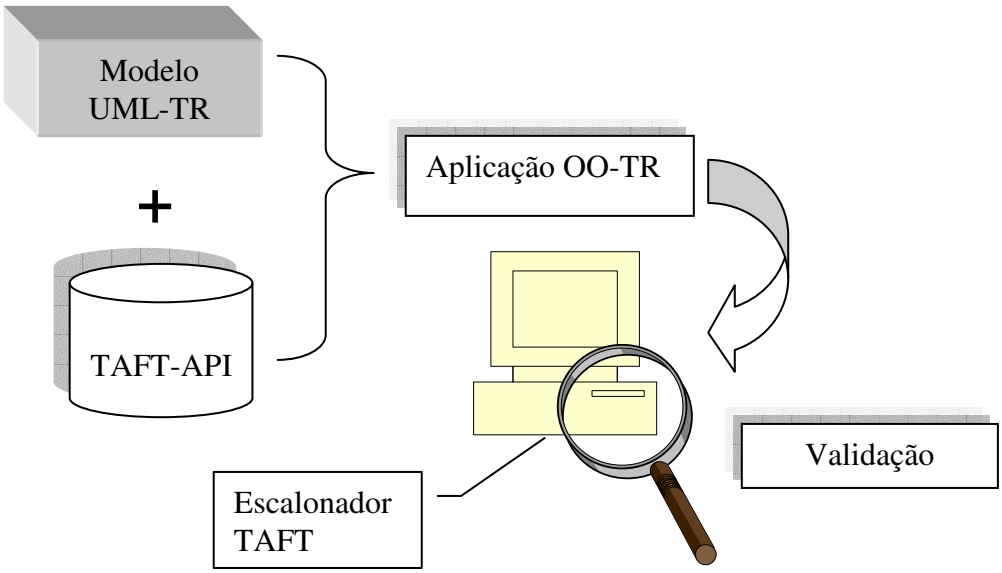


FIGURA 9.1 – Relacionamento entre os elementos do método proposto

9.2 Integração dos Elementos em Ferramentas de Apoio

Durante a discussão sobre o relacionamento entre os elementos da metodologia proposta realizada na seção anterior, antecipou-se alguns casos onde ferramentas CASE podem ser utilizadas. A seguir aponta-se em detalhes o uso destas ferramentas:

1. **Etapa de modelagem**: durante esta estapa faz-se necessário o uso de uma ferramenta com suporte para modelagem de diagramas UML, suportando também os elementos provenientes do perfil UML-TR.
2. **Escalonador TAFT**: com o uso de uma ferramenta para análise de escalonabilidade, é possível implementar o teste de aceitação do TAFT para que o mesmo faça a análise do modelo UML, decorado com as restrições do perfil UML-TR.
3. **Etapa de mapeamento**: nesta etapa pode ser feito o uso de uma ferramenta de geração de código, a fim de prover a transição automática do modelo UML-TR para um programa baseado na TAFT-API.
4. **Validação dos requisitos temporais**: a arquitetura de validação proposta constitui-se por uma ferramenta de monitoração e também por uma ferramenta que permite a análise dos requisitos e apresentação dos resultados.

9.3 Considerações sobre o Processo de Desenvolvimento

A tese proposta não foca na elaboração de uma metodologia de desenvolvimento, mas sim em prover ferramentas de apoio que podem ser usadas por projetistas independentemente da metodologia utilizada. Entretanto, realizou-se uma análise preliminar daquela que se considera uma seqüência de desenvolvimento "sugerida" para ser utilizada com o método descrito nesta tese, conforme publicado em [BEC 2002b].

Na abordagem proposta defende-se como idéia principal o entendimento do domínio do problema, o qual antecede as preocupações com detalhes tecnológicos de projeto (e.g. componentes de hw, protocolos de comunicação, etc). Portanto, considera-se necessário uma análise profunda do problema a ser solucionado. Além disso, para reduzir custos e minizar esforços, o modelo obtido na fase de análise é utilizado como ponto de partida para o projeto do sistema. Este modelo é obtido aplicando-se uma das atuais metodologias de projeto para sistemas tempo real voltadas para a UML (e.g. [DOU 98; GOM 2000]), permitindo assim a identificação de classes e objetos, bem como do relacionamento entre os mesmos. O comportamento das classes e dos objetos é descrito através de notações propostas pela UML, como o Statecharts e o diagrama de atividades. Já as funcionalidades do sistema são expressas através de diagramas de caso de uso, diagramas de colaboração e diagramas de seqüência.

De acordo com a filosofia seguida, a validação do comportamento do modelo gerado pode ser feita através de simulação, permitindo uma maior compreenção por parte do usuário do domínio do problema. É possível interpretar este modelo de simulação como uma representação de objeto, idéia ou sistema, o qual reflete os mesmos aspectos de comportamento presentes no sistema real. Em outras palavras, um modelo de simulação nada mais é do que uma abstração de entidades o qual enfatiza aspectos relevantes para o estudo em questão. Uma análise de escalonabilidade também é passível de ser realizada durante esta fase. O término da mesma completa a etapa de análise.

Considerando que o modelo corrente já reflete a estrutura estática e dinâmica dos componentes do sistema, define-se uma ligação clara entre os elementos computacionais (presentes no modelo) e os elementos físicos (presentes no processo físico sendo analisado). Tal ligação é obtida através da inserção de uma camada de abstração adicional, denominada "camada de drivers[25]", o qual é usada como elo de ligação entre os elementos físicos e lógicos.

No próximo passo, mapeia-se o modelo com os drivers para a arquitetura alvo utilizando um processo de geração automática de código, o qual é precedido pela compilação e posterior execução do mesmo. Durante a execução da aplicação, ativa-se o componente de monitoração para coletar informações sobre o comportamento temporal da aplicação. Os dados obtivos são posteriormente validados através da arquitetura de validação proposta, permitindo assim determinar o atendimento aos requisitos temporais. Ao término desta fase, encontra-se completo o ciclo de desenvolvimento para aplicações tempo real, conforme a seqüência proposta.

[25] Também referenciada na literatura como camada de interface.

10 Conclusões e Trabalhos Futuros

Ao longo do texto foram apresentados os elementos que fazem parte da tese proposta, isto é, um método baseado no paradigma da orientação a objetos que aborda o ciclo de desenvolvimento de aplicações tempo real. O trabalho apresentado cobre a área estudada em abrangência, pois inclui as etapas de modelagem, projeto, execução e validação de uma aplicação tempo real. O maior benefício do método proposto é o de abordar as etapas citadas de uma maneira integrada, evitando descontinuidades no processo de desenvolvimento. Muitas vezes estas descontinuidades dificultam a detecção de erros, bem como dificultando a validação do sistema e tendem a aumentar os custos de desenvolvimento. Analisando separadamente os elementos envolvidos nesta tese, verifica-se que os mesmos apresentam uma série de avanços para o estado da arte das tecnologias voltadas para o desenvolvimento de aplicações tempo real, conforme destacado nos próximos parágrafos.

Na proposta de mapeamento das restrições temporais presentes em modelos orientados a objetos para o nível de implementação, trabalha-se com elementos que constituem o estado da arte nas áreas de modelagem e de programação, como o perfil UML-TR e o RTSJ. No âmbito da presente tese de doutorado foram introduzidos novos estereótipos e marcas, de modo a aumentar o poder de expressão do perfil UML-TR e simplificar a especificação de requisitos comumente encontrados em aplicações tempo real. Por exemplo, propôs-se o acréscimo das marcas *CRcallTimeout* e *CRcallTimeoutHandler* ao estereótipo «*CRsynch*» e também o acréscimo das marcas *SAbloqTimeout* e *SAbloqTimeoutHandler* ao estereótipo «*SAresource*».

Além disso, também foram abordadas alterações na semântica de alguns elementos do perfil UML-TR, principalmente por considerá-los muito abstratos e com uso limitado, como é o caso do estereótipo *«RTaction»* e seus derivados. Também foram observadas falhas de relacionamento entre os elementos do perfil, onde o estereótipo *«CRaction»* deveria ser derivado de *«RTtimedAction»*, o estereótipo *«SAschedulable»* deveria ser derivado de *«CRconcurrent»* e o estereótipo *«SAaction»* deveria ser derivado de *«RTtimedAction»*. Dentre as conclusões adicionais obtidas com a análise efetuada, destacam-se a falta de expressividade da notação TVL, que poderia adotar a sintaxe do TMO ou do PEARL, a necessidade da inclusão do estereótipo *«RTtimedAction»* e finalmente a falta de um elemento para caracterizar explicitamente exceções causadas por falhas temporais (e.g. o não cumprimento de um *deadline*).

Outro ponto observado no perfil UML-TR é a carência de um estereótipo para representar *deadlines* especiais do tipo fim-a-fim, que relacione um conjunto de entidades de escalonamento dependentes entre si. Este carência foi reconhecida inclusive pelo grupo de desenvolvimento do perfil UML-TR, que deverá acrescentar nas próximas versões do mesmo um novo estereótipo denominado *«SAendToEnd»*. Entretanto, a proposta do grupo não cobre o problema apresentado no sistema de cooperação dos veículos *Kurt*, descrito no capítulo 8, onde várias unidades de escalonamento executem de forma independente e possuem uma relação não-casual, i.e. a seqüência de processamento apresenta um caminho conhecido gerado por alguma interação complexa dentro do sistema. Contudo, sabe-se o tempo máximo no qual o conjunto de atividades deve estar finalizado. Portanto, verifica-se novamente a carência de um estereótipo para expressar esta restrição, que atualmente pode ser expressa somente através de anotações não padronizadas (conseqüentemente não mapeáveis).

No contexto do método de desenvolvimento proposto, o ambiente de execução apresentado contribuiu como uma plataforma de desenvolvimento capaz de oferecer garantias de execução às aplicações tempo real projetadas. Baseado na política de escalonamento TAFT, este ambiente é capaz de manter a CPU com índices de utilização elevado sem perder a garantia do cumprimento aos *deadlines* dos chamados TPs, que constituem a unidade de escalonamento em questão. Apresentou-se nesta tese um novo mecanismo de escalonamento para o TAFT, baseado em algoritmos do tipo *Earliest Deadline*. Este mecanismo é formado por um escalonador de dois níveis, onde no primeiro nível é usado o algoritmo LRT e no segundo o algoritmo EDF, constituindo-se num mecanismo eficiente em tempo de execução e com um teste de aceitação simples.

A grande vantagem do mecanismo proposto em relação às abordagens existentes está no fato do mesmo escalonar tarefas com tempos de execução estimados (ECET) e não baseado no pior caso (WCET). Como conseqüência, são obtidos índices de utilização de CPU mais elevados do que nas abordagens tradicionais, com a garantia de que os *deadlines* são sempre cumpridos, mesmo em situações de sobrecarga transiente. A contrapartida é que nestas situações de sobrecarga a aplicação sofre uma perda de funcionalidade, a fim de que os *deadlines* sejam cumpridos. Entretanto, isto não deve comprometer a aplicação a longo prazo, visto que tais situações são atingidas somente em casos excepcionais. Além disso, o mecanismo proposto oferece um teste de aceitação simples para os taskPairs.

O comportamento do mecanismo desenvolvido foi comprovado através da utilização do *benchmark* Hartstone. Através de diversos experimentos, constatou-se que o ambiente desenvolvido possui um desempenho muito próximo do ótimo em situações onde o índice de utilização da CPU está abaixo de 100%. Acima deste valor, o ambiente se mostra bastante estável, conseguindo manter um bom índice de tarefas finalizadas sem perdas de funcionalidade. O ambiente também se mostra imune aos problemas causados pelo efeito dominó, o qual é apresentado pelo EDF em situações de sobrecarga. Também através dos experimentos foi possível constatar que o parâmetro α-quantil, utilizado para expressar o grau de precisão relacionado com o tempo de execução, não se apresenta como uma boa alternativa para expressar a importância das tarefas. Testes adicionais realizados mostram que algoritmos “baseados em valores“ , como o HDF, apresentam-se como uma boa maneira de solucionar esta carência.

Como continuação do trabalho desenvolvido, verificam-se dois temas para serem tratados em detalhes. Um deles é justamente trabalhar no sentido de implementar uma nova versão do escalonador capaz de expressar o parâmetro “importância”, principalmente em situações de sobrecarga. Outro ponto pertinente é explorar os benefícios em se utilizar uma estratégia de escalonamento distribuída, ou seja, oferecer suporte para que as aplicações sejam capazes de se auto-reconfigurar com base nos índices de utilização da CPU e também da rede de comunicação. Além disso, vislumbra-se o uso do TAFT em aplicações reais. Atualmente encontra-se em desenvolvimento a implementação de uma aplicação voltada para futebol de robôs, onde as características do TAFT são utilizadas em algoritmos para fusão de dados provenientes de sensores distribuídos.

Para integrar a proposta de mapeamento do perfil UML-TR apresentada no capítulo 4 com o ambiente de execução do escalonador TAFT, elaborou-se a chamada TAFT-API. Esta API se oferece como alternativa ao uso de bibliotecas que acessam diretamente as funcionalidades do sistema operacional na especificação de requisitos temporais. Com isto, consegue-se uma melhor separação entre definição das

funcionalidades da aplicação e a especificação de requisitos temporais, melhorando assim a legibilidade das aplicações desenvolvidas. Além disso, os elementos do UML-TR permitem uma representação destes requisitos em alto-nível, ou seja, durante a etapa de modelagem da aplicação. Desta forma, consegue-se realizar uma transição suave entre a especificação, implementação e execução de aplicações tempo real.

A API elaborada mantém praticamente todas as construções do RTSJ usadas na especificação de requisitos temporais e que não estão direta ou indiretamente relacionadas com propriedades específicas da linguagem Java, pois a TAFT-API é escrita na linguagem C++. Conforme discutido, são evitados os mecanismos do C++ que causam sobrecarga de desempenho nas aplicações produzidas, como a determinação dinâmica de tipos e o tratamento de exceções.

Aliado aos componentes de modelagem e de execução, apresenta-se também uma arquitetura para verificar os requisitos temporais especificados durante a modelagem. Esta arquitetura, que consiste de um módulo de monitoração e também de um módulo de validação, propicia tanto uma análise quantitativa quanto qualitativa dos requisitos temporais modelados. A mesma permite ao projetista fazer a validação dos requisitos temporais especificados durante a fase de modelagem. A validação é feita off-line, sendo executada após a execução do sistema. Além da detecção de erros, esta análise proporciona ao projetista uma visão mais detalhada a respeito da relação temporal entre as tarefas. Por exemplo, o projetista pode vir a identificar a possibilidade de agrupar funcionalidades que executam periodicamente em uma mesma tarefa (ou TP), com o objetivo de minimizar as trocas de contexto e aumentar a eficiência do sistema.

Além deste tipo de validação, convém ressaltar que o objetivo da ferramenta de monitoração incorporada ao sistema - proposta originalmente por Gergeleit [GER 2001] - é permitir uma análise estatística a respeito dos tempos de execução das tarefas, a fim de prover importantes informações para o escalonador. Este elemento constitui o componente "*Time Aware*" do escalonador TAFT. Apesar de atualmente estes elementos não estarem integrados, esta integração é factível e encontra-se prevista como uma continuação deste trabalho.

Para finalizar as conclusões deste trabalho, são feitas algumas considerações adicionais no que diz respeito aos trabalhos futuros. A seguir são listados e descritos alguns trabalhos os quais podem ser realizados como continuação dos resultados obtidos com a tese apresentada:

- *Extensão do escalonador TAFT para operar com aplicações distribuídas*: o objetivo deste trabalho é abordar problemas de escalonamento tempo real distribuído, que são tratados atualmente de maneira *ad hoc* em muitas aplicações. A idéia desta proposta é manter o componente de monitoração do TAFT atento a situações de sobrecarga nos nodos de computação, tornando o mesmo capaz de decidir pela eventual migração de tarefas entre processadores, de modo a sempre garantir o cumprimento dos *deadlines*. A plataforma de robôs equipados com microcontroladores interligados por uma rede CAN-bus seria o alvo deste estudo.
- *Suporte em Hardware para o escalonador TAFT*: objetiva-se implementar o escalonador TAFT proposto nesta tese em um processador dedicado, com o objetivo de diminuir a sobrecarga causada pelo algoritmo de escalonamento. Com isto, torna-se-á possível a elaboração de mecanismos de escalonamento mais sofisticados, aumentando ainda mais o número de tarefas críticas

suportadas. Este trabalho poderia utilizar como plataforma base a arquitetura proposta por Götz [GÖT 2001].

- *Desenvolvimento de infra-estruturas de apoio à depuração de sistemas tempo real embutidos e tolerante a falhas*: este trabalho visa utilizar o suporte de monitoração e escalonamento apresentados com o TAFT utilizando as infra-estruturas de apoio à depuração, nomeadamente de macroblocos sintetizáveis, acessíveis através de pontos de acesso normalizados, e.g. IEEE 1149.1 ou IEEE 5001.
- *Emulação em FPGAs de sistemas tempo real embutidos*: esta proposta de trabalho visa permitir a emulação em FPGAs de sistemas tempo real embutidos. Para tanto, utilizar-se-ia o mapeamento proposto nesta tese para a criação de um gerador automático de código para especificações em UML e com o auxílio do perfil UML-TR. Esta proposta faria uso em um primeiro instante do *core* FPGA para o processador *PowerPC*.

Anexo 1 Estereótipos e Marcas Propostas neste Trabalho

- *«RTtimedAction»*: estereótipo que estende *«RTaction»* para representar uma ação com restrição de tempo.
  - *«RTtimedAction».RTdeadline*: marca para representar um deadline, o qual é uma informação de grande relevância no contexto de aplicações tempo real (outrora definida somente no modelo de análise de escalonabilidade).
  - *«RTtimedAction».RTdeadMissHandler*: marca que contém uma referência para a operação que deverá tratar uma possível violação de deadline.

  - «*CRsynch*».*CRcallTimeout*: marca que permite a associação de um tempo máximo de espera com uma chamada síncrona.
  - «*CRsynch*».*CRcallTimeoutHandler*: ao estereótipo «*CRsynch*»: marca que permite a associação de uma referência ao código de tratamento de exceção acionado caso a chamada demorar mais tempo para retornar do que fora especificado pelo *timeout*.

  - «*SAresource*».*SAbloqTimeout*: marca que permite a associação de um tempo máximo de bloqueio ao solicitar um recurso compartilhado.
  - «*SAresource*».*SAbloqTimeoutHandler*: marca que permite a associação de uma referência ao código de tratamento de exceção, acionado caso a espera demorar mais tempo do que fora especificado pelo *timeout*.

- *«EXtiming»*: estereótipo usado para caracterizar uma exceção gerada por falha temporal (derivado, por exemplo, de um *framework* para tratamento de exceções).
  - *«EXtiming».EXimportance*: marca cuja semântica permite definir a importância da exceção.

- *«TAtaskPair»*: estereótipo que representa a unidade de concorrência de uma aplicação TAFT; derivado de *«SAschedRes»*.
- *«TAmainPart»*: estereótipo utilizado com a ação tomada em resposta ao evento de ativação do TP; esta ação representa a MP, i.e. o código que trata do fluxo normal de execução do TP.
  - *«TAmainPart».TAecet*: marca usada para caracterizar o tempo de execução da MP.
- *«TAexceptionPart»*: estereótipo utilizado com a ação tomada em resposta à exceção temporal causada pela MP; esta ação representa a EP, i.e. o código de exceção do TP.
  - *«TAexceptionPart».TAwcet*: marca usada para caracterizar o tempo de execução da EP.
- *«TAmpTimeout»:* estereótipo que caracteriza a exceção gerada pela violação do tempo de execução da MP, procando o chaveamento entre MP e EP.

Anexo 2 Resumo da TAFT-API

```
class RelativeTime(int milliseconds, int nanoseconds)

class PriorityParameters( int priority )

class PeriodicParameters( int releaseTime, int period, int
deadline )
class AsyncEventHandler()
  virtual void handleAsyncEvent()

class TAPI_TimeoutMonitor( int deadline, void *handler );
  void normalEnd()

class RealtimeThread( SchedulingParameters sp, ReleaseParameters
rp)
  virtual void run()
  void waitForNextPeriod()

class SchedulingParameters()
class PriorityParameters: public SchedulingParameters
class ImportanceParameters: public SchedulingParameters

class ReleaseParameters()
class PeriodicParameters:  public ReleaseParameters
class SporadicParameters:  public ReleaseParameters
class AperiodicParameters: public ReleaseParameters

class Semaphore()
  void p()
  void v()

class Mutex()
  void lock(int deadline=INFINITE_TIME)
  void unlock()

class MutexControl()
  setMonitorControl( Mutex *obj, PriorityInheritance *pi)
  setMonitorControl( Mutex *obj, PriorityCeiling *pc)

class PriorityInheritance()

class PriorityCeiling()
```

Referências Bibliográficas

[AUD 90] AUDSLEY, N.; BURNS, A. Real-Time System Scheduling. **Report YCS 134**. York: Department of Computer Science, University of York, Jan. 1990.

[AWA 96] AWAD, M.; KUUSELA, J.; ZIEGLER, J. **Object-Oriented Technology for Real-Time Systems**: a Practical Approach using OMT ad Fusion. Englewood Cliffs, NJ: Prentice Hall, 1996.

[BAC 2001] BACELLAR, L.; NETTER, C. Assessing the Real-Time Properties of Windows CE 3.0. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 4., 2001, Magdeburg, Germany. **Proceedings…** Los Alamitos, CA, USA: IEEE Computer Society, 2001. p. 179-184.

[BAR 97] BARABANOV, M. **A Linux-based Real-Time Operating System**. 1997. M.S. Thesis, New Mexico Institute of Technology.

[BEC 97] BECKER, Leandro. **Uso de Objetos Ativos na Modelagem e Implementação de Sistemas Tempo real**: Análise do Estado-da-Arte e Estudo de Caso. Porto Alegre: CPGCC da UFRGS, 1997. (TI - 694).

[BEC 99] BECKER, Leandro Buss. **Ambiente de Modelagem e Implementação de Sistemas Tempo Real usando o Paradigma de Orientação a Objetos**. 1999. 80p. 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 2000a] BECKER, Leandro Buss; PEREIRA, Carlos. From Design to Implementation: Tool Support for the Development of Object-Oriented Distributed Real-Time Systems. In: EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS, 12., 2000, Stockholm, Sweden. **Proceedings...** Los Alamitos - USA: IEEE Computer Society, 2000. p.109-116.

[BEC 2001] BECKER, Leandro Buss; GERGELEIT, Martin; NETT, Edgar; PEREIRA, Carlos. An Integrated Environment for the Complete Development Cycle of an Object-Oriented Distributed Real-Time System. **International Journal of Computer Systems Science Engineering**, Leics, UK, v. 16, 2001, p.89-96.

[BEC 2001a] 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.

[BEC 2001b] BECKER, Leandro Buss; GERGELEIT, Martin. Execution Environment for Dynamically Scheduling Real-Time Tasks. In: IEEE REAL-TIME SYSTEMS SYMPOSIUM, 22., 2001, London. **Work in Progress Proceedings...** York: University of York, Department of Computer Science, 2001. p.13-16.

[BEC 2001c] BECKER, Leandro Buss; PEREIRA, Carlos; VILLELA, Claudio. Framework for Component-Based Development of Distributed Real-Time Systems. In: IEEE INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED REAL-TIME DEPENDABLE SYSTEMS, 6., 2001, Roma. **Proceedings…** Los Alamitos - USA: IEEE Computer Society, 2001. p. 85-90.

[BEC 2002] BECKER, Leandro Buss; HOLTZ, Rudy; PEREIRA, Carlos. On Mapping RT-UML Specifications to RT-Java API: Bridging the Gap. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING,

5., 2002, Washington. **Proceedings…** Los Alamitos, CA, USA: IEEE Computer Society, 2002. p. 348-355.

[BEC 2002a] BECKER, Leandro Buss; PEREIRA, Carlos. Proposta de mecanismo para o escalonamento dinâmico de tarefas tempo real tolerante a falhas. In: WORKSHOP DE TEMPO REAL, 4., 2002, Búzios. **Anais…** Rio de Janeiro: Nce/UFRJ, 2002. p. 19-25.

[BEC 2002b] BECKER, Leandro Buss; PEREIRA, Carlos. SIMOO-RT - An Object-oriented Framework for the development of real-time industrial automation systems. **IEEE Transactions On Robots Automation**, New York, v. 18, n. 4, p. 421-430, 2002.

[BER 2001] BERNAT, G.; CAYSSIALS, R. Guaranteed On-Line Weakly-Hard Real-Time Systems. In: IEEE REAL-TIME SYSTEMS SYMPOSIUM, 22., 2001, London. **Proceedings…** Los Alamitos, CA, USA: IEEE Computer Society, 2002. p. 25-35.

[BOL 2001] BOLLELLA, Greg; GOSLING, James; BROSGOL, Benjamin. **The Real-Time Specification for Java**. Reading, Massachusetts: Addison-Wesley, 2000. 195p. ISBN 0-201-70323-8. (Ida: este livro foi editado em 2000; entretanto os autores lançaram uma nova versão em 2001 – formato eletrônico PDF – que mantém o mesmo ISBN, sendo que eu gostaria de referenciar a nova versão.)

[BOO 91] BOOCH, Grady. **Object-Oriented Development**. Redwood City: Benjamin/Cummings Publishing Company, 1991.

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

[BUR 95] BURNS, A.; WELLINGS, A. **HRT-HOOD™: A Structured Design Method for Hard Real-Time Ada Systems**. Oxford , UK: Elsevier, 1995. 330p. ISBN 0 444 82164 3.

[BUR 98] BURNS, A.; RANDELL, B.; ROMANOVSKY, A.; STROUD, R.J.; WELLINGS, A.J.; XU, J. Temporal Constraints and Exception Handling in Object-Oriented Distributed Systems. Design for Validation (DeVa) - Third Year Report, Esprit LTR Project 20072 - DeVa, 1998.

[BUT 93] BUTTAZZO, G.; STANKOVIC, J. RED: a Robust Earliest Deadline Scheduling Algorithm. In: INTERNATIONAL WORKSHOP ON RESPONSIVE COMPUTING SYSTEMS. 3., 1993, Austin, USA. **Proceedings…** [S.l:s.n.], 1993.

[BUT 95] BUTTAZZO, G.; SPURI, M.; SENSINI, F. Value vs. Deadline Scheduling in Overload Conditions. In: IEEE REAL-TIME SYSTEMS SYMPOSIUM, 16., 1995, Pisa, It. **Proceedings…** Los Alamitos, CA, USA: IEEE Computer Society, 1995. p. 90-99.

[CHE 89] CHETTO, H.; CHETTO, M. Some Results of the Earliest Deadline Scheduling Algorithm. **IEEE Trans. on Software Engineering**, New York, v. 15, n. 10, p. 1261-1269, 1989.

[CHO 91] CHODROW, S.; JAHANIAN, F.; DONNER, M. Run-Time Monitoring of Real-Time Systems. In: IEEE REAL-TIME SYSTEMS SYMPOSIUM, 12., 1991. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 1991. p.74-83.

[COP 97] COPSTEIN, Bernardo. **SIMOO**: Plataforma Orientada a Objetos para Simulação Discreta Multi-Paradigma. 1997, Tese (Doutorado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[DEA 88] DEAN, T.; BODDY, M. An Analysis of time-dependent planning. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELIGENCE, 7., 1988. **Proceedings...** St. Paul, Minessota, USA: Morgan Kaufmann, 1988. p. 49-54.

[DOU 98] DOUGLASS, Bruce Powel. **Real-Time UML:** Developing Efficient Objects for Embedded Systems. Reading, Massachusetts: Addison-Wesley, 1998. 328p. ISBN 0-201-65784-8.

[DOU 98a] DOUGLASS, Bruce Powel. **Doing Hard Time:** Using Object-Oriented Programming Software Pattern in Real Time Application. Reading, Massachusetts: Addison-Wesley, 1999.

[EDG 2001] EDGAR, S.; BURNS, A. Statistical Analysis of WCET for Scheduling, In: IEEE REAL-TIME SYSTEMS SYMPOSIUM, 22., 2001, London, UK. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 2001. p. 215-224.

[FAR 2000] FARINES, J.M.; OLIVEIRA, R.; FRAGA, J. **Sistemas de Tempo Real**. São Paulo: Departamento de Computação, IME-USP, 2000. Disponível em: <http://www.das.ufsc.br/gtr/livro/principal.htm>. Acesso em: 29 abr. 2003.

[GER 2001] GERGELEIT, Martin. **A Monitoring-based Approach to Object-Oriented Real-Time Computing**. 2001. Ph.D. Thesis. Institute für Verteilte Systeme, Otto-von-Guericke-Universität Magdeburg, Magdeburg, Germany.

[GER 2003] GERGELEIT, Martin; BECKER, Leandro Buss; NETT, Edgar. Robust Scheduling in Team-Robotics. In: INTERNATIONAL WORKSHOP ON PARALLEL AND DISTRIBUTED REAL-TIME SYSTEMS, 11., 2003, Nice, France. **Proceedings…** Los Alamitos, CA, USA: IEEE Computer Society, 2003.

[GOM 93] GOMAA, Hassan. **Software Design Methods for Concurrent and Real-Time Systems**, Reading, Massachusetts: Addison-Wesley, 1993. 464p. ISBN 0-201-52577-1.

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

[GÖT 2001] GÖTZ, Marcelo. **Proposta de arquitetura de hardware e software para sistemas tempo-real distribuídos.** 2001. 96p. Dissertação (Mestrado em Engenharia Elétrica) – Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[HAL 91] HALANG, W.; STOYENKO, A. **Constructing Predictable Real-Time Systems**. Kluwer Academic: Boston, 1991. 352p. ISBN 0-7923-9202-7.

[HAR 87] HAREL, D. Statecharts: a Visual Formalism for Complex Systems. **Science of Computer Programming**. [S.l.:s.n.], v. 8, p. 231-274, 1987.

[HÖL 2002] HÖLTZ, Rudy Hamilton. **Desenvolvimento de Sistemas Tempo real usando Orientação a Objetos:** Estudo sobre o Mapeamento de Especificações para Linguagens de Programação. 2002. 75p. Trabalho de Conclusão (Mestrado Profissional em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[HUS 2003] HUSEMMAN, R. et al. Análise do Determinismo Temporal de Aplicações Distribuídas usando CORBA TAO. In: WORKSHOP DE TEMPO REAL, 5., 2003, Natal. **Anais…** Natal: UFRN, 2003.

[IEEE 90] IEEE Std 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*, 1990.

[IEEE 2001] IEEE Std 1003.1-2001. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX®). 2001.*

[ISH 90] ISHIKAWA, Y. et al. Object-Oriented Real-Time Language Design: Constructs for Timing Constraints. **SIGPLAN Notices**, New York, v. 25, n. 10, p. 289-298, 1990.

[JAH 86] JAHANIAN, F.; LEE, R.; MOK, A. Safety Analysis of Timing Properties in Real-Time Systems. **IEEE Trans. Software Eng**, New York, v. SE-12, p. 890-904, Sept. 1986.

[JAI 91] JAIN, R. **The Art of Computer Systems Performance Analysis**. New York: Wiley-Interscience, 1991.

[JC 2000] J CONSORTIUM. **Real-Time Core Extensions for the Java TM Platform**. International J Consortium Specification, [S.l.:s.n.], 2000.

[KIM 99] KIM, Kane. Real-Time Object-Oriented Distributed Software Engineering and the TMO Scheme. **Int'l Jour. of Software Engineering and Knowledge Engineering**, [S.l.], v. 2, p. 251-276, April 1999.

[KIM 2000] KIM, Kane. APIs for real-time distributed object programming. **IEEE Computer (special**

issue on OO RT Distributed Computing), New York, v. 33, n. 6, p. 72-80, June 2000.

[KLI 93] KLIEN, M.H. et al. **A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems**. Boston: Kluwer Academic, 1993.

[KURT 2001] KURT2 Roboter Homepage. Disponível em < http://www.kurt2.de>. Acesso em: 20 maio 2001.

[LAN 92] LANGE, F.; KRÖGER, R.; GERGELEIT, M. Design and Implementation of a Distributed Measurement System. **IEEE Trans. on Parallel and Distributed Systems**, New York, v. 3, n. 6, p. 657-671, April 1992.

[LEU 82] LEUNG, J.Y.T.; WHITEHEAD, J. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. **Performance Evaluation**, v. 2, n. 4, p. 237-250, Dec. 1992.

[LIU 73] LIU, C.L.; LAYAND, J.W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. **Journal of the Association for Computer Machinery**, v. 20, n. 1. p. 46-61, 1973.

[LIU 94] LIU, J.W.-S. et al. Imprecise Computations. **Proc. of the IEEE**, New York. v. 82, n. 1, p. 68-82, 1994.

[LIU 2000] LIU, J.W.-S. **Real-Time Systems**. Englewood Cliffs, New Jersey: Prentice-Hall, 2000. ISBN 0-13-099651-3.

[MOT 93] MOTUS, Leo. Time Concepts in Real-Time Software, **Control Engineering Practice**, ,v. 1, n. 1, p.21-33, Feb. 1993.

[NET 97] NETT, E.; STREICH, H. The GMD-Snake - Real-Time Scheduling of a Flexible Robot Application at Run-Time, In: INT. WORKSHOP ON PARALLEL COMPUTATION AND SCHEDULING IN COMPUTERS, 1997, Ensenada, Mexico. **Proceedings...** [S.l.:s.n.].

[NET 2001] NETT, E.; GERGELEIT, H.; MOCK, M. Enhancing O-O Middleware to become Time-Aware. **Real-Time Systems Journal (Special Issue on Real-Time Middleware in Real-Time Systems)**, v. 20, n. 2, p. 211-228, March, 2001.

[NET 2001a] NETT, E.; GERGELEIT, H.; MOCK, M. Mechanisms for a Reliable Cooperation of Vehicles. In: INTERNATIONAL SYMPOSIUM ON HIGH ASSURANCE SYSTEMS ENGINEERING, 6., 2001, Boca Ration, Florida, Oct. 2001. **Proceedings...** [S.l.:s.n.].

[NIS 99] NIST **Requirements for Real-time Extensions for the Java Platform:** Report from the Requirement Group for Real-time Extensions for the Java Platform**.** Special Publication 500-243, Sept. 1999. Disponível em: <http://www.itl.nist.gov/div897/ctg/real-time/rtj-final-draft.pdf>. Acesso em: 15 ago. 2000.

[OMG 98] OBJECT MANAGEMENT GROUP. **CORBA Specification v. 2.2**. 1998. (OMG Document formal/98-12-01).

[OMG 99] OBJECT MANAGEMENT GROUP **Real-Time CORBA**. 1999. (OMG document orbos/99-02-12).

[OMG 2002] OBJECT MANAGEMENT GROUP. **UML Profile for Schedulability, Performance, and Time Specification**. 2002. Disponível em: <http://www.omg.org/cgi-bin/doc?ptc/02-03-02>. Acesso em: 06 mar. 2003.

[OMG 2002a] OBJECT MANAGEMENT GROUP. **UML 1.5 with Action Semantics**. 2002. Disponível em: <http://www.omg.org/cgi-bin/doc?ptc/2002-09-02>. Acesso em: 06 mar. 2003.

[PER 94] PEREIRA, Carlos. Real Time Active Objects in C++/Real-Time UNIX. In: ACM SIGPLAN WORKSHOP ON LANGUAGES, COMPILER, AND TOOL SUPPORT FOR REAL-TIME SYSTEMS, 1994, Orlando, EUA. **Proceedings...** [S.l.:s.n.], 1994.

[PER 94a] PEREIRA, Carlos; DARSCHT, P. Using Object-Orientation in Real-Time Applications: an Experience Report. In: Proc. TOOLS EUROPE 94. 1994, Versailles, France. **Proceedings...** [S.l.:s.n.], 1994.

[PER 95] PEREIRA, Carlos. Temporal Reasoning on Object-Oriented Real-Time Specifications by using Constraint Propagation Techniques. In: IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, 20., 1995, FL, USA. **Proceedings...** [S.l.:s.n.], 1995.

[PER 96] PEREIRA, Carlos. Métodos de Análise de Sistemas Tempo Real Usando Técnicas de Orientação a Objetos. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, São Carlos, SP. **Anais...** [S.l.:s.n.], 1996.

[PER 97] PEREIRA, Carlos. Applying Object-Oriented Concepts to the Development of Real-Time Industrial Automation Systems. In: IEEE INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED REAL-TIME DEPENDABLE SYSTEMS, 3., 1997, Newport Beach, USA. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997, p. 264-270.

[PER 99] PEREIRA, Carlos et al. Quantitative evaluation o distributed object-oriented progamming environments for real-time applications. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2., 1999, Saint Malo, France. **Proceedings…** Los Alamitos, CA, USA: IEEE Computer Society, 1999. p. 133-138..

[RIC 2001] RICHARDSON, P.; SIEH, L.; ELKATEEB, A. Fault-Tolerant Adaptive Scheduling for Embedded Real-Time Systems. **IEEE Micro**, New York, v. 21, n. 5, p. 41-51. Sept./Oct. 2001.

[ROD 2000] RODRIGUES, C.; LOYALL, J.; SCHANTZ, R. Quality Objects (QuO): Adaptative Management and Control Middleware for End-to-End QoS. In: OMG WORKSHOP ON REAL-TIME AND EMBEDDED DISTRIBUTED OBJECT COMPUTING, 1., 2000, Falls Church, USA. **Proceedings…** [S.l.:s.n.], 2000.

[ROM 99] ROMANOVSKY, A.; XU, J.; RANDELL, B. Coordinated Exception Handling in Real-Time Distributed Object Systems. **Int. Journal of Computer Systems Science and Engineering (Special Issue on Object-Oriented Real-Time Distributed Systems)**, v. 14, n 4, p. 197-208, 1999.

[RUM 91] RUMBAUGH, James et al. **Object Oriented Modeling and Design.** Englewood Cliffs, New Jersey: Prentice-Hall 1991.

[SCH 2001] SCHEMMER, S.; NETT, E.; MOCK, M. Reliable Real-Time Cooperation of Mobile Autonomous Systems, In: IEEE SYMPOSIUM ON REALIABLE DISTRIBUTED SYSTEMS, 20., 2001. **Proceedings…**October 28-31, 2001, p. 238 – 246.

[SCH 98] SCHMIDT, D.; LEVINE, D.; MUNGEE, S.. The design of the TAO real-time object request broker. **Computer Communications**, v. 21, n. 4, April 1998.

[SEL 94] SELIC, Bran et al. **Real Time Object Oriented Modeling**. [S.l.]: John Wiley and Sons, 1994.

[SEL 99] SELIC, Bran. Turning clockwise: Using UML in the real-time domain. **Communications of the ACM**., New York, v. 42, n. 10, p. 46-54. Oct. 1999.

[SEL 2000] SELIC, Bran. A Generic Framework for Quantitative Modeling of Real-Time Systems in UML, In: IFAC WORKSHOP ON REAL-TIME PROGRAMMING, 25., 2000, Palma de Mallorca, Spain. **Proceedings…** p. 139-144.

[SEL 2002] SELIC, Bran. The emerging real-time UML Standard. **International Journal of Computer Systems Science Engineering**, Leics, UK, v. 17, n. 2, 2002.

[SHI 94] SHIN, K.G.; RAMANATHAN, P. Real-Time Computing: A New Discipline of Computer Science and Engineering. **Proceedings of the IEEE**, New York, v. 82, p. 6-24. Jan. 1994.

[SHO 2000] SHORK, E.; SHEU, P. Real-Time Distributed Object Computing: An Emerging Field. **IEEE Computer** (Guest Editors' Introduction), New York, USA, p. 45-46. June 2000.

[STA 88] STANKOVIC, J. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. **IEEE Computer**, New York, USA, p. 10-19. Oct. 1988.

[STA 98] STANKOVIC, J.; SPURI, M.; RAMAMRITHAM, K.; BUTTAZZO, G. **Deadline**

Scheduling for Real-Time Systems: EDF and Related Algorithms. Boston: Kluwer Academic, 1998. ISBN 0-7923-8269-2.

[STR 94] STREICH, H. TaskPair-Scheduling: an Approach for Dynamic Real-Time Systems. In: INTERNATIONAL WORKSHOP ON PARALLEL AND DISTRIBUTED REAL-TIME SYSTEMS, 2., 1994, Cancun, Mexico, **Proceedings…** [S.l.:s.n.] p. 28-29.

[TUR 99] TURLEY, Jim. Tensilica CPU bends to designers' will. Microprocessor Report, 8 March 1999.

[WAR 85] WARD, Paul; MELLOR, S.J. **Structured Development for Real-Time Systems**. New York: Yourdon, 1985. 3v.

[WEI 89] WEIDERMAN, N. Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications. Carnegie Mellon University, Technical Report CMU/SEI-89-TR-23, 1989.

[WEI 92] WEIDERMAN, N.; KAMENOFF, N. Hartstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time Systems. **Journal of Real-Time Systems**, v. 4. p. 353-382, 1992.

[WEL 2002] WELLINGS, A.; CLARK, R.; JENSEN, D.; WELLS, D. A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 5., 2002, Washington. **Proceedings…** Los Alamitos, CA, USA: IEEE Computer Society, 2002. p. 13-22.

[WER 89] WERUME, W.; WINDAUER, H., **Introduction to PEARL:** Process and Experiment Automation Realtime Language, Vieweg Verlag, 1989, ISBN 3-528-33590-4.

[WIL 2000] WILD, Rafael. **Proposta de Ferramenta para Validação Temporal em Barramentos de Campo**. 2000. Dissertação (Mestrado em Engenharia Elétrica) – Departamento de Engenharia Elétrica, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[WIL 2000a] WILD, R.; BECKER, L.B.; GÖTZ, M.; HUSEMANN, R.; PEREIRA, C. Tool support for evaluating temporal characteristics of industrial protocols, In: IEEE INTERNATIONAL WORKSHOP ON FACTORY COMMUNICATION SYSTEMS, 2000, Porto, Portugal, **Proceedings...** Los Alamitos, USA: IEEE Computer Society, 2000. p. 195-201.