233424-7

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

Ambiente de Modelagem e Implementação de Sistemas Tempo Real usando o Paradigma de Orientação a Objetos

por

LEANDRO BUSS BECKER

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

UFRGS
SABi
05222922

Prof. Dr. Carlos Eduardo Pereira
Orientador

Porto Alegre, março de 1999.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Becker, Leandro Buss

Ambiente de Modelagem e Implementação de Sistemas Tempo Real usando o Paradigma de Orientação a Objetos / por Leandro Buss Becker. - Porto Alegre: PPGC da UFRGS, 1999.

80f.:il.

Dissertação (Mestrado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 1999. Orientador: Pereira, Carlos E.

1. Sistemas de Tempo Real. 2. Orientação a Objetos. 3. Modelagem. 4. Simulação 5. Ambiente de Desenvolvimento. 6. Linguagem de Programação 7. Geração de Código. I. Pereira, Carlos E. II. Título.

Informática - SBU
Sistemas: Tempo real
Orientação: Objetos
Simulação
CNPq 1.03.04.00-2

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA
N.º CHAMA.: 681.32.066(043) B395A
REG: 38288
22.06.99
ORIGEM: D
DATA: 14.06.99
R$ 20,00
FUNDO: II
FORN.: II

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Pro fª. Wrana Panizzi
Pró-Retor de Pós-Graduação: Franz Rainer Semmelmann
Diretor do Instituto de Informática: Prof. Philippe Navaux
Coordenador do CPGCC: Profª. Carla Dal Sasso Freitas
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro
Bibliotecária responsável pela normalização de documentos: Ida Rossi

Dedico este trabalho aos meus pais, Carlos Celeste e Maria Teresa, aos meus avós Adão e Dila, e à minha madrinha Tia Cláudia

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Sistema de Bibliotecas da UFRGS

38238

681.32.066(043)
B395A

INF
1999/233424-7
1999/06/22

Agradecimentos

Gostaria fazer um agradecimento especial ao meu orientador, Prof. Carlos Eduardo Pereira, pela atenção e dedicação na orientação deste trabalho e principalmente pela amizade formada neste período. Da mesma forma gostaria de agradecer aos bolsistas Guilherme Holsbach, João Jornada e Tiago Avancini pela ajuda prestada no desenvolvimento desta dissertação. Também gostaria de agradecer a colaboração dos integrantes do grupo SIMOO, especialmente dos professores Flávio Wagner e Bernardo Copstein. Ressalto também a amizade e a colaboração dos professores João Netto e Renato Ventura e dos colegas Wilson Pardi Jr., Andre Cordenonsi, Carlos Mitidieri, Joel Bortolotto, Mairon Bataghlin, Marcio Kreütz, Adriana Pereira, Leandro Tibola, Isabel Fernandes, Aline Flores, Rafael Wild, Marcelo Göetz, João Pacheco e Sérgio Süess. Gostaria de agradecer ainda ao amigo e colega Luciano Ferreira, por ter me ensinado o "caminho das pedras" dentro do CPGCC. Por fim, deixo um agradecimento à todas aquelas pessoas não mencionadas, mas que me deram força para conduzir este trabalho.

Sumário

Resumo **10**

Abstract **11**

1 Introdução **12**

1.1 Sistemas de Tempo Real **12**

1.2 Motivações **13**

1.3 Objetivos **14**

1.4 Organização do Texto **15**

2 Análise do Estado da Arte **17**

2.1 Introdução **17**

2.2 Ferramentas CASE **17**

2.2.1 Rational Rose 17

2.2.2 ObjecTime 18

2.2.3 SIMOO 19

2.2.4 Comparação entre as Ferramentas 20

2.3 Linguagens de Programação para Tempo Real **22**

2.3.1 Linguagem AO/C++ 22

2.3.2 Modelo de Objetos RTO.k 24

2.3.3 Linguagem Java Tempo Real 26

2.3.4 Comparação das Principais Características das Linguagens 28

3 Ambiente SIMOO-RT **33**

3.1 Introdução **33**

3.2 Especificação dos Requisitos Temporais **33**

3.3 Geração de Código Executável **35**

3.3.1 Criação de Programas AO/C++ a partir da Especificação 35

3.3.2 Geração das Classes AO/C++ 38

3.3.3 Mecanismos de Inicialização da Aplicação 39

3.3.4 Geração Automática do Projeto (Makefile) 42

4 Especificação de Comportamento **50**

4.1 Introdução **50**

4.2 Funcionalidades dos Diagramas de Transição de Estados **50**

4.3 Implementação do Mecanismo de Controle **52**

5 Estudos de Caso **56**

5.1 Introdução **56**

5.2 Estudo de Caso 1: Tanque de Armazenamento **56**

5.3 Estudo de Caso 2: Célula de Manufatura Robotizada **62**

5.4 Estudo de Caso 3: Controle de Elevadores **64**

5.5 Análise dos Resultados **67**

6 Conclusões e Trabalhos Futuros **74**

6.1 Considerações Gerais **74**

6.2 Trabalhos Relacionados **76**

Bibliografia **77**

Lista de Figuras

FIGURA 2.1 - Célula de Manufatura Robotizada 17
FIGURA 2.2 - Visão geral dos diagramas do Rational Rose 19
FIGURA 2.3 - Visão geral da ferramenta ObjecTime. 21
FIGURA 2.4 - Editor de modelos (MET) do SIMOO. 22
FIGURA 2.5 - Painel de controle e cenário da simulação no SIMOO. 23
FIGURA 2.6 - Configuração dos processos QNX em tempo de execução 27
FIGURA 2.7 - Estrutura de um objeto RTO.k 28
FIGURA 3.1 - Especificações de *deadline* e período 35
FIGURA 3.2 - Características temporais de uma aplicação de tempo real 36
FIGURA 3.3 - Diferenciação entre as classes ativas e as classes passivas 37
FIGURA 3.4 - Possibilidades de comunicação entre objetos ativos e passivos 37
FIGURA 3.5 - Estrutura do programa gerado em relação ao modelo SIMOO 38
FIGURA 3.6 - Atributos da classe 'Sensor' 39
FIGURA 3.7 - Atributos da classe 'Celula' 39
FIGURA 3.8 - Atributos dos objetos ativos 40
FIGURA 3.9 - Visão geral do código gerado e do seu processo de compilação 41
FIGURA 3.10 - Estrutura do arquivo *header* AO/C++ 41
FIGURA 3.11 - Disparo e estabelecimento de comunicação dos objetos AO/C++ 42
FIGURA 3.12 - Formato dos métodos com restrições temporais em AO/C++ 44
FIGURA 3.13 - Programa que ativa o objeto agregador do modelo 45
FIGURA 3.14 - Transformações provocadas pelo pré-processador AO/C++ 47
FIGURA 3.15 - Exemplo de 'Makefile' 48
FIGURA 3.16 - Opções de configuração do projeto 49
FIGURA 4.1 - Visão geral da ferramenta STDE 51
FIGURA 4.2 - Janela de especificação das transições 52
FIGURA 4.3 - Janela de especificação dos estados 52
FIGURA 4.4 - Exemplo do mecanismo de controle dos métodos 53
FIGURA 4.5 - Estrutura do método 'Select' 55
FIGURA 4.6 - Estrutura dos arquivos gerados com o uso do STDE 55
FIGURA 5.1 - Visão geral da bancada de controle do tanque 57
FIGURA 5.2 - Diagrama de classes para o sistema de controle do tanque 57

FIGURA 5.3 - Diagrama de transição de estados para a classe Tanque 58
FIGURA 5.4 - Estados do elemento de visualização da classe Tanque 59
FIGURA 5.5 - Simulação do sistema do tanque em andamento 60
FIGURA 5.6 - Exemplo da especificação de métodos com restrições temporais 60
FIGURA 5.7 - Diagrama de classes adaptado para o ambiente de execução 61
FIGURA 5.8 - Método DesligarBomba() antes e depois da geração da aplicação 62
FIGURA 5.9 - Código do método cíclico Amostrar() 63
FIGURA 5.10 - Código de definição das classes 63
FIGURA 5.11 - Código gerado para o diagrama de transição de estados 64
FIGURA 5.12 - Diagrama de transição estados da classe Sensor 66
FIGURA 5.13 - Configuração do sistema de controle da CMR 67
FIGURA 5.14 - Refinamento do diagrama de classes da CMR 67
FIGURA 5.15 - Configuração do sistema para controle de elevadores 69
FIGURA 5.16 - Diagramas da classe PainelExt 70
FIGURA 5.17 - Diagramas da classe Elevador 70
FIGURA 5.18 - Diagrama de transição de estados para a classe Elevador 71
FIGURA 5.19 - Animação do sistema de elevadores no ambiente SIMOO-RT 71

Lista de Tabelas

TABELA 5.1 - Estatísticas do modelo de controle do tanque 65
TABELA 5.2 - Estatísticas do modelo da célula de manufatura robotizada 68
TABELA 5.3 - Estatísticas do sistema para controle de elevadores 72

Resumo

Este trabalho descreve o desenvolvimento de um ambiente integrado para modelagem, simulação e implementação de sistemas de tempo real distribuídos (STRD), especialmente aqueles voltados para automação industrial. O ambiente proposto faz uso do paradigma de orientação a objetos, sendo baseado no ambiente SIMOO, desenvolvido no âmbito de uma tese de doutorado no CPGCC. A motivação para a realização deste trabalho surgiu através de alguns estudos de caso, que constataram que as ferramentas CASE existentes não incorporavam todas as propriedades desejadas em termos de suporte para modelagem e implementação dos STRD. Dentre estas propriedades, destacam-se a capacidade para especificação de restrições temporais, o suporte à simulação do modelo desenvolvido e a capacidade de geração automática de código para a aplicação final. O ambiente proposto tem por objetivo suprir as carências observadas, adicionando ao ambiente SIMOO original facilidades para a descrição de restrições temporais e facilidades para descrição de comportamento do modelo desenvolvido. Além disso, é incorporada ao ambiente a capacidade de geração automática de código em uma linguagem que suporte as restrições temporais descritas no modelo. Este trabalho foi desenvolvido no contexto do projeto ADOORATA (*A Distributed Object-Oriented Architecture for Real-Time Automation*), como parte do Programa de Cooperação entre Brasil e Alemanha, financiado pelas agências CNPq e DLR.

Palavras-Chave: Sistemas de tempo real, orientação a objetos, modelagem, simulação, ambiente de desenvolvimento, linguagem de programação, geração de código.

TITLE: "MODELING AND IMPLEMENTATION ENVIRONMENT FOR THE DEVELOPMENT OF REAL-TIME SYSTEMS USING OBJECT ORIENTED PARADIGM".

Abstract

This work describes the development of an integrated object-oriented environment for modeling, simulation and implementation of distributed real-time systems (DRTS), especially those conceived for industrial automation. This work extends the SIMOO environment, conceived as part of a Ph.D. thesis in the CPGCC at Federal University of Rio Grande do Sul. Its motivation began during some case studies, which concluded that existing CASE tools don't incorporate all the desired features for modeling and implementation of DRTS. Among these features, capacities to specify timing constraints, to simulate/animate the model and to automatically generate the final application code are highlighted. The proposed environment intends to overcome these lacks, adding to the original environment features for the specification of timing requirements and the application behavior, allowing the creation of an object-oriented simulation model. Additionally it automatically generates the application executable code, which makes use of the incorporated specifications to guarantee its correctness. This work has been developed within the context of the ADOORATA project (A Distributed Object-Oriented Architecture for Real-Time Automation), as part of the Brazilian-German Cooperation Program, sponsored by CNPq and DLR.

Keywords: Real-time systems, object oriented, modeling, simulation, development environment, programming language, code generation.

1 Introdução

1.1 Sistemas de Tempo Real

Um sistema computacional é classificado como sendo de tempo real, quando seu correto funcionamento não depende apenas do processamento de sinais de entrada ou estímulos e sua transformação em sinais ou variáveis de saída, mas especialmente da satisfação de requisitos temporais [STA88; HAL91]. Estes requisitos temporais determinam, por exemplo, intervalos de tempo máximos nos quais os sinais de saída devem ser gerados ou o período de amostragem dos sinais de entrada. Como exemplos de sistemas tempo real podem ser citados: sistemas de automação industrial, sistemas de controle para aviação, controle de usinas de geração de energia elétrica, controle de processos químicos, controle de tráfego aéreo, etc.

Os recentes avanços nas áreas de eletrônica, informática e microeletrônica têm possibilitado a utilização destes sistemas numa série de processos que outrora não eram economicamente viáveis ou tecnicamente possíveis. As tendências indicam que os sistemas tempo real do futuro devem ser ainda maiores, mais complexos e com longos ciclos de vida, exibindo um comportamento dinâmico e altamente adaptativo às diferentes condições de operação [BIH92]. Estes requisitos adicionais tornam o projeto de sistemas tempo real mais complexos do que o projeto de sistemas convencionais, pois além de terem que satisfazer requisitos temporais, trabalham com uma alta escala de componentes de software e hardware distribuídos ao longo da aplicação em questão.

Entretanto, até recentemente o desenvolvimento de sistemas tempo real era desprovido de qualquer metodologia. Sistemas tempo real eram desenvolvidos de maneira *ad-hoc* e, em geral, as fases iniciais do desenvolvimento de sistemas computacionais, a análise de requisitos e o projeto estruturado, eram relegadas a um segundo plano. O desenvolvimento concentrava-se nas fases de codificação dos programas - geralmente escritos em Assembler para microcontroladores. À medida que sistemas tempo real foram se tornando mais e mais complexos, seus desenvolvedores obrigaram-se a familiarizarem-se os conceitos adotados pelas metodologias de projeto, como modularização, hierarquia, abstração, encapsulamento de informação, etc.

Uma metodologia de projeto baseia-se fundamentalmente no uso de modelos. Um modelo é uma representação abstrata de um elemento ou sistema a fim de facilitar a

compreensão do elemento/sistema sendo modelado. O ato de abstrair é uma capacidade humana fundamental que permite lidar com complexidade. Desta forma, durante o processo de modelagem tem-se que detalhes não essenciais são negligenciados e características relevantes para compreensão são enfatizados. Para construir sistemas complexos, desenvolvedores devem geralmente abstrair diferentes visões do sistema. Além disso, devem verificar se os modelos satisfazem os requisitos do sistema e acrescentar detalhes gradativamente para transformar modelos em uma implementação. Resumidamente, uma metodologia de projeto define uma heurística de criação de modelos (representados sob a forma de diagramas) e um seqüenciamento de atividades a serem realizadas na criação de um sistema.

Infelizmente, métodos convencionais de engenharia de software mostram-se ineficientes no desenvolvimento de sistemas tempo real, uma vez que estes, além de não considerarem requisitos temporais, usualmente não abordam o projeto de hardware e software de forma integrada. Segundo reportado em [KEM92], uma metodologia de projeto para sistemas tempo real deve possuir aspectos para contemplar tanto requisitos temporais quanto complexidade, além de auxiliar os desenvolvedores durante todo o ciclo de vida do sistema, desde a engenharia de requisitos (*requirements engeneering*), até as fases de projeto (*design*), implementação, teste e colocação em operação.

Atualmente, as metodologias influenciadas pelo paradigma de orientação a objetos [COX86; MEY88, COA90, BOO91, RUM91, SHL92] têm recebido um maior destaque, justamente por possibilitar efetivamente o tratamento de sistemas complexos. O uso de objetos como elementos encapsuladores de informações (dados e comportamento), aliado à possibilidade de abstrair-se objetos com estrutura e comportamento semelhantes na forma de classes, além do uso de conceitos como herança, instanciamento, agregação, etc., constituem uma poderosa ferramenta de modelagem para sistemas complexos. Apesar do paradigma de orientação a objetos ter se tornado popular no desenvolvimento de sistemas computacionais sem requisitos temporais, experiências em estudos de caso relatadas por Pereira em [PER94a], demonstram que o paradigma de orientação a objetos é também uma ótima alternativa para a modelagem de sistemas tempo real, permitindo obter resultados superiores àqueles obtidos com métodos convencionais.

A fim de complementar os métodos orientados a objeto existentes na literatura com relação às suas deficiências na especificação de sistemas tempo real, Pereira em

[PER96b] apresenta o método OORTAC - *Object-Oriented Real-Time and Control*. Este método é baseado no conceito de objetos ativos, no qual cada objeto é visto como uma unidade de processamento autônoma capaz de comunicar-se com os demais objetos para realização de tarefas. Especialmente no que tange a descrição de aspectos temporais, OORTAC propõe extensões significativas aos métodos de análise existentes [PER94c].

1.2 Motivações

A adoção de uma metodologia de análise e projeto leva à necessidade da existência de ambientes/ferramentas computacionais apropriadas para auxiliar os desenvolvedores, principalmente na fase de modelagem da aplicação. Estas ferramentas são denominadas CASE (*Computer Aided Software Engeneering*), pois oferecem suporte computacional para criação de diagramas e especificações, que representam as entidades constituintes do problema e a maneira como elas se relacionam e se comportam. Além auxiliar na criação de diagramas e especificações, as ferramentas CASE com capacidade de geração automática de código facilitam significativamente o processo de implementação da aplicação.

Em contrapartida, não basta que sejam usadas metodologias de desenvolvimento sem que os conceitos adotados sejam refletidos durante o processo de implementação da aplicação. Sendo assim, o uso de uma linguagem de baixo nível como o Assembler na codificação das aplicações, além de não refletir os conceitos utilizados, torna ainda mais dispendioso e suscetível a erros o processo de implementação. Desta forma, torna-se extremamente interessante o uso de linguagens de programação de alto nível, que possam representar os conceitos adotados nas fases de análise e projeto e que ofereçam construções especiais para representar os requisitos temporais presentes neste tipo de aplicação.

Diante deste contexto, em [BEC97] foi realizado um estudo comparativo envolvendo tanto ferramentas de modelagem como linguagens de programação que fazem uso do paradigma da orientação a objetos. Os resultados obtidos neste estudo comprovaram a carência de ferramentas CASE adequadas para o desenvolvimento de sistemas de tempo real. Por outro lado, foi certificado que o uso de linguagens de programação de alto nível facilita os processos de implementação e manutenção das

aplicações de tempo real. A realização do trabalho citado motivou o desenvolvimento desta dissertação, cujo principal objetivo é a obtenção de uma ferramenta integrada para desenvolvimento de sistemas tempo real orientados a objetos, envolvendo aspectos de modelagem, simulação e implementação. O suporte a simulação baseia-se na idéia de que uma abordagem experimental pode ser útil para auxiliar a tomada de decisões [PID94], permitindo eventuais alterações no sistema ainda na fase de projeto.

A presente dissertação está inserida no contexto de um projeto de cooperação entre Brasil, Alemanha e Portugal, denominado ADOORATA - *A Distributed Object-Oriented Approach for Real-Time Systems Development*, que aborda diversos pontos relacionados com o desenvolvimento de sistemas tempo real. Devido a variedade deste assunto, outros trabalhos também estão sendo desenvolvidos pelo grupo de pesquisa do Laboratório de Automação Industrial da UFRGS, dentre os quais podem ser citados: monitoração e análise de barramentos industriais [WILD98], especificação de protocolos de comunicação entre objetos distribuídos [FLO98], desenvolvimento de sistemas supervisórios para automação industrial [PAR98] e especificação de um metamodelo para a linguagem UML com extensões para tempo real [SOU98].

1.3 Objetivos

O objetivo principal desta dissertação é obter um ambiente integrado, que evite descontinuidades no processo de desenvolvimento de aplicações tempo real orientadas a objetos. A busca pelo entendimento e detalhamento do problema inicial resulta no modelo conceitual do mesmo. Uma vez que o projetista consegue fazer esta definição, o mesmo modelo poderá ser aproveitado tanto para a simulação quanto para a implementação das entidades que compõem o problema.

O ambiente proposto, denominado SIMOO-RT, será baseado na ferramenta SIMOO [COP97] e deverá acrescentar a ela funcionalidades que permitam a especificação de requisitos temporais dentro do próprio modelo e a geração do programa executável. Aspectos como a geração de código utilizando linguagens que possibilitem a distribuição da aplicação através de vários processadores também são considerados. Além disso, as possibilidades de integração da ferramenta com os outros trabalhos em desenvolvimento no grupo, citados na seção anterior, também são verificadas neste trabalho. Embora tenham de ser oferecidas facilidades para a

simulação dos requisitos temporais especificados, pela necessidade de se delimitar o escopo do trabalho, ainda que cientes de sua importância, não serão abordados os aspectos de simulação dos modelos.

1.4 Organização do Texto

O restante desta dissertação encontra-se organizado da seguinte forma: o capitulo 2 discute uma análise do estado da arte de ferramentas CASE e linguagens de programação que fazem uso do paradigma de orientação a objetos; o capítulo 3 apresenta e descreve as principais características do ambiente SIMOO-RT; o capítulo 4 aborda uma maneira alternativa de representação do comportamento das entidades do modelo; o capítulo 5 descreve alguns estudos de caso e, finalmente, o capítulo 6 exibe as conclusões observadas e indica os trabalhos futuros para continuidade desta pesquisa.

2 Análise do Estado da Arte

2.1 Introdução

Atualmente, metodologias de projeto orientadas a objetos têm sido bastante utilizadas para a modelagem de sistema tempo real complexos [PER94a]. Com este preceito, foram concebidas diversas ferramentas CASE de âmbito acadêmico e comercial que oferecem suporte computacional a estas metodologias. Conforme relatado em [BEC97], "embora estas ferramentas possuam algumas características distintas, variando de acordo com a finalidade para que foram projetadas, elas são bastante semelhantes em termos de representação dos conceitos de orientação a objetos". Este capítulo descreve os principais resultados obtidos no trabalho em questão, onde foram estudadas e comparadas algumas ferramentas CASE e algumas linguagens de programação para tempo real que fazem uso do paradigma da orientação a objetos.

Como estudo de caso para esta comparação escolheu-se uma célula de manufatura robotizada, disponível no Laboratório de Automação Industrial do Departamento de Engenharia Elétrica da UFRGS (vide fig. 2.1). A célula utilizada contém os seguintes componentes: uma esteira de transporte, um braço de robô, uma mesa de armazenagem, uma câmera de vídeo e um sensor de presença.

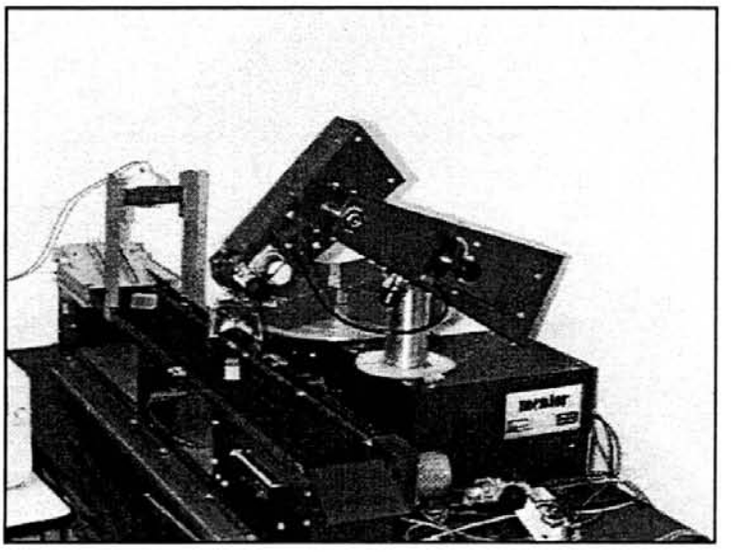


FIGURA 2.1 - Célula de Manufatura Robotizada

2.2 Ferramentas CASE

2.2.1 Rational Rose

O Rational Rose é uma ferramenta CASE projetada e distribuída pela empresa *Rational Software Incorporation Ltd.* As metodologias empregadas nesta ferramenta (OMT [RUM91], Booch [BOO91] e UML[RAT97; DOU98]) recomendam o uso de visões estáticas e dinâmicas de um modelo lógico, e um modelo físico para representar os componentes definidos nas fases de análise e projeto orientadas a objetos. Utilizando a sua notação, a ferramenta permite ao usuário criar e refinar estas visões dentro de um modelo global, que representa o domínio do problema. O modelo global contém classes, objetos, operações, subsistemas, módulos, processadores, dispositivos e os relacionamentos entre os mesmos.

Um modelo também contém diagramas e especificações, os quais possibilitam uma maneira de visualizar e manipular os componentes do próprio modelo e as suas propriedades. A interface gráfica do usuário do Rational Rose permite visualizar, criar, modificar e manipular os componentes de um modelo, representados através de diferentes diagramas. Os diagramas suportados pela ferramenta são os seguintes: diagrama de classes, diagrama de objetos ou cenários, diagrama de estados, diagrama de componentes e diagrama de desenvolvimento.

De acordo com os princípios de desenvolvimento estabelecidos pelo Rational Rose, inicialmente foram especificados os componentes que constituem o diagrama de classes do modelo, identificando os atributos das entidades e quais delas necessitam comunicar-se entre si. O comportamento individual de cada classe foi definido através do diagrama de estados da ferramenta. Os objetos do sistema e as situações (eventos) em que eles interagem foram especificados com a utilização do diagramas de cenários.

Uma vez que o objetivo desta dissertação é a modelagem de sistemas de tempo real, foi levantado o suporte dado pela ferramenta para permitir a representação das restrições temporais da aplicação. A única restrição temporal encontrada diz respeito ao tempo de execução dos métodos das classes, que é informado a partir do campo 'Time' da sua janela de especificações.

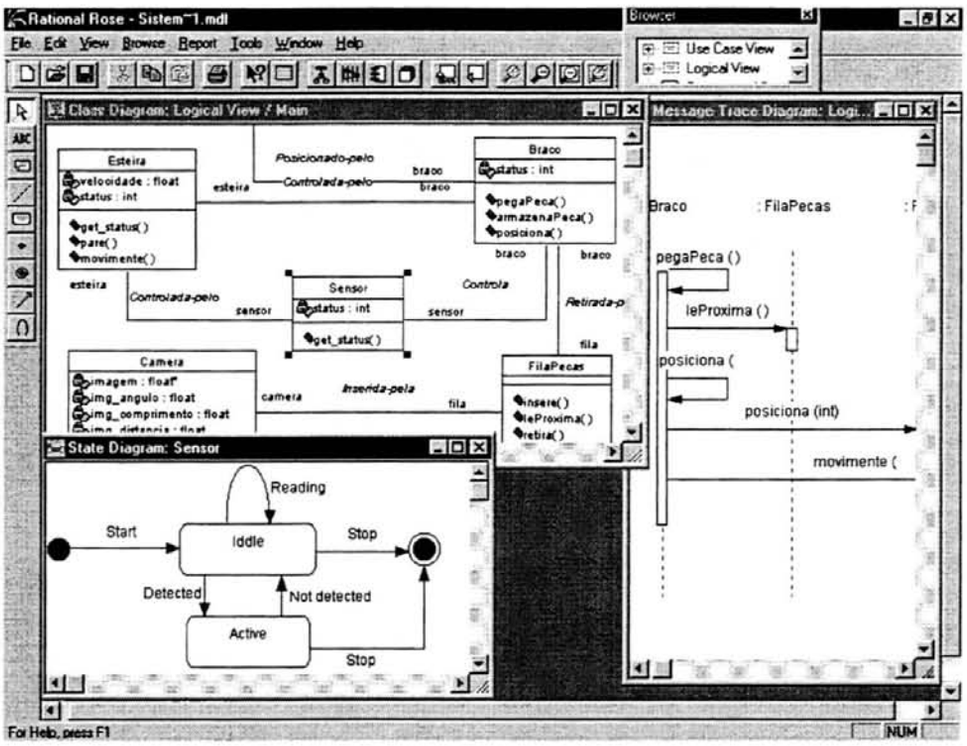


FIGURA 2.2 - Visão geral dos diagramas do Rational Rose

Outra característica estudada foi a capacitação da ferramenta para realizar geração automática de código. Utilizando o gerador de código do Rational, foi produzido o código fonte em C++ a partir das informações contidas no modelo desenvolvido. Observou-se que o código produzido continha somente a estrutura das classes, sem nenhuma consideração sobre o comportamento especificado. Os relacionamentos de conhecimento foram representados através de ponteiros e as informações adicionais (e.g. tempo de execução de um método) foram inseridas no código como comentários, precedendo a declaração da função membro. É importante ressaltar que o código gerado ainda não é o programa final, mas somente um esqueleto do mesmo.

2.2.2 ObjecTime

ObjecTime é uma ferramenta de CASE comercializada pela empresa canadense ObjecTime Ltd., baseada na metodologia ROOM (*Real-Time Object-Oriented Modeling*) (vide [SEL94]), que foi especialmente concebida para o desenvolvimento de sistemas de tempo real. Ela é usada para a criação de modelos orientados a objetos que,

após terem sido implementados, podem ser executados na própria ferramenta ou portados para uma arquitetura alvo que executa uma máquina virtual especial.

O processo de desenvolvimento compreende a construção de dois modelos hierárquicos: modelo estrutural de alto nível (*high-level structure modeling*) e modelo comportamental de alto nível (*high-level behavior model*). Dois tipos de objetos são utilizados na ferramenta: atores (*actors*) e dados (*data-objects*). Os primeiros são entidades ativas que podem executar concorrentemente, enquanto o segundo tipo corresponde a objetos passivos de linguagens de programação orientada a objetos, tais como C++, ou seja como unidades de encapsulamento de dados e de suas respectivas funções de acesso. A fim de possibilitar a comunicação entre atores, estes possuem pontos de comunicação em sua interface, aos quais são associados protocolos.

Os protocolos especificam quais as mensagens que podem ser trocadas entre os atores através dos pontos de comunicação, determinando ainda a direção e a seqüência válidas destas mensagens. As mensagens trocadas entre os atores também podem conter um objeto de dados, que representa uma variável encapsulando informações.

A ferramenta ObjecTime faz uso de dois diagramas principais, que vem a ser: diagrama de estrutura e diagrama de comportamento. O diagrama de estrutura contém os relacionamentos de comunicação entre os atores e também informações estruturais do tipo "contém" (espécie de agregação), onde um ator complexo pode ser refinado em estruturas mais simples. Para descrição do comportamento de atores a ferramenta sugere o uso de máquinas de estado hierárquicas, que permitem a definição de ações, expressas em C++ ou RPL (linguagem própria da ferramenta), associadas à chegada de mensagens.

Segundo a metodologia da ferramenta, todo ator deve possuir necessariamente uma máquina de estados definida. A interação entre os atores pode ser vista através de diagramas de seqüência de mensagens (*message sequences charts*), semelhantes ao diagrama de cenários do Rational Rose. A descrição de requisitos temporais é possível através do uso de instâncias de uma classe *timer*, as quais geram mensagens de *timeout* no caso de que o tempo máximo especificado seja ultrapassado.

Quanto à geração de código, o ObjecTime gera código C++ separado para cada elemento do modelo. Este código é utilizado para gerar um programa executável que irá

rodar na máquina virtual. A figura 2.3 mostra uma visão geral da ferramenta ObjecTime, contendo alguns diagramas e a janela de controle da simulação do modelo.

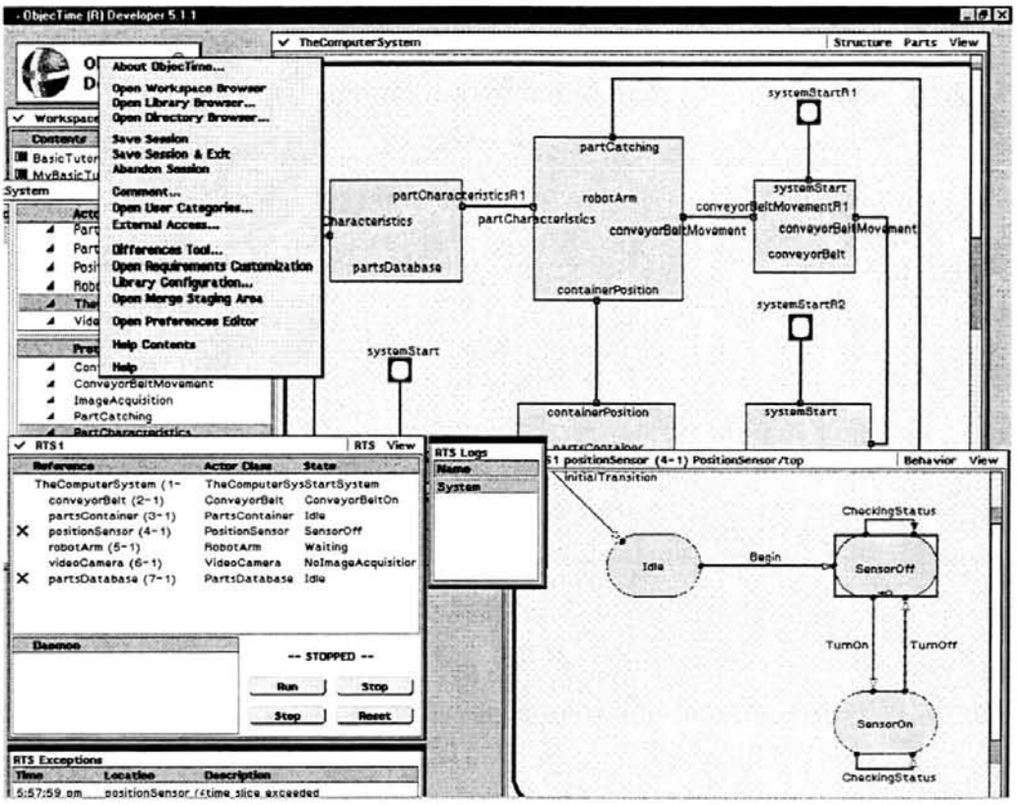


FIGURA 2.3 - Visão geral da ferramenta ObjecTime.

2.2.3 SIMOO

SIMOO [COP97] é um *framework* que foi desenvolvido como parte de uma tese de doutorado no CPGCC/UFRGS. Ele permite a construção de modelos de simulação discreta OO, que incorporam recursos para permitir uma interação máxima entre o usuário e o modelo.. O mapeamento de entidades de simulação em elementos autônomos [WIL94], baseados na idéia de objeto ativo, oferece não apenas uma unidade de distribuição que permite a construção de modelos escalonáveis, como incentiva a construção de bibliotecas de entidades reusáveis. Entre os objetivos propostos para este ambiente está a possibilidade de se especificar modelos de simulação usando o paradigma de orientação a objetos.

O ambiente SIMOO adota uma abordagem hierárquica, permitindo que o modelo seja detalhado por níveis conforme a necessidade. Além disso, o uso de elementos de interface e monitores de visualização permite ao usuário projetista manter

a separação entre o domínio do modelo e os recursos de visualização e coleta de resultados. Para permitir a construção de modelos de simulação, a partir da especificação da estrutura do *software* que implementa o modelo, o SIMOO oferece uma ferramenta de edição de modelos, denominada MET (*Model Editor Tool*) (fig. 2.4). O editor permite ainda a especificação gráfica da estrutura estática das entidades do modelo, através do diagrama de instâncias. A descrição do comportamento das entidades é feita diretamente em linguagem de programação C++. No que diz respeito a especificações de restrições temporais, o SIMOO não possui nenhum tipo de suporte.

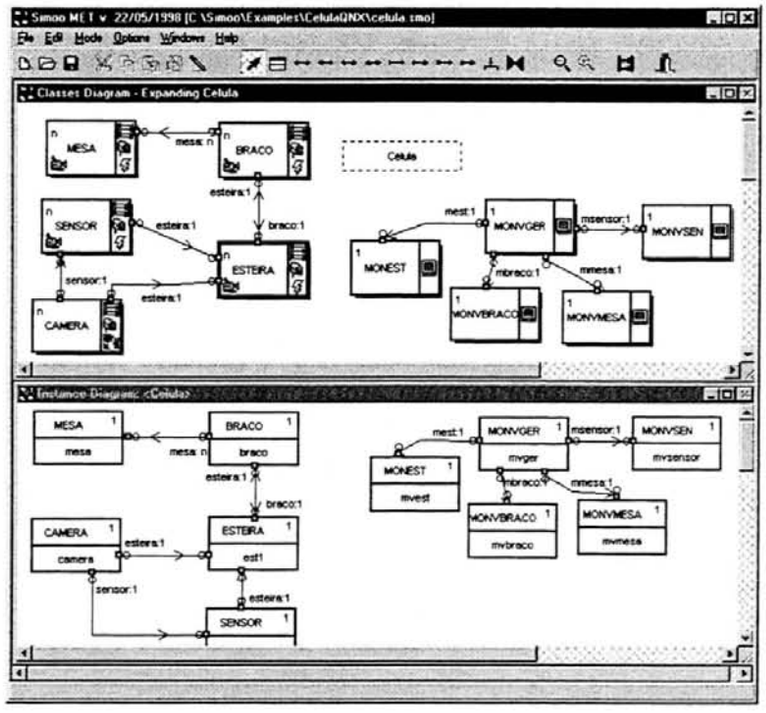


FIGURA 2.4 - Editor de modelos (MET) do SIMOO.

Como pode ser verificado, o diagrama de classes desenvolvido no SIMOO para o estudo de caso é bastante semelhante aos diagramas das outras ferramentas analisadas. Eles diferem pela notação gráfica utilizada e pela necessidade da construção de um modelo hierárquico por parte do SIMOO, cujo diagrama mostrado na figura 2.4 é derivado de uma classe básica 'Célula', que agrega o restante do sistema. Outra característica deste diagrama é a presença de elementos autônomos, representando as entidades que fazem parte do domínio do problema, e de elementos de interface (representados por um ícone de um monitor no lado direito), que são de uso exclusivo do ambiente de simulação e servem para monitorar as mensagens recebidas pelas

classes às quais eles são associados. As classes com algum EI associado são diferenciadas pela presença do ícone de uma câmera no canto inferior esquerdo. Esta propriedade permite a definição de cenários animados para a simulação.

Também merece destaque a capacidade do SIMOO de gerar o código executável da simulação, a partir da descrição feita no MET. Neste código, são acrescentados automaticamente os recursos necessários à interação do usuário com o modelo. Entre os aspectos acrescentados pode-se destacar o painel que permite o controle do andamento da simulação, elemento básico de interação com o usuário, e a capacidade de auto-conhecimento das entidades. Este último aspecto permite ao usuário, durante a execução da simulação, questionar e alterar diversos aspectos das entidades presentes no modelo. O cenário gerado para o modelo desenvolvido, juntamente com a janela de controle da simulação, são mostrados na figura 2.5.

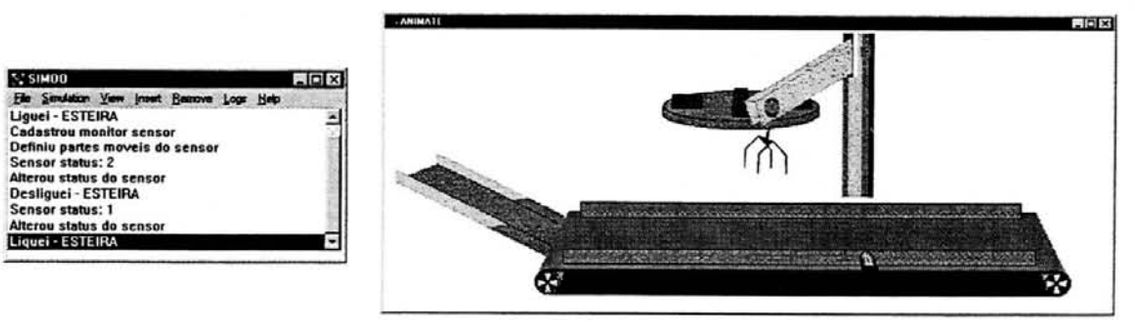


FIGURA 2.5 - Painel de controle e cenário da simulação no SIMOO.

2.2.4 Comparação entre as Ferramentas

Apesar das diferenças encontrada nas ferramentas estudadas, são vários os pontos comuns que podem ser observados comparando-se as mesmas. Por exemplo, todas as ferramentas enfocam o sistema sendo analisado sob mais de um ponto de vista, sendo que uma abordagem comum é descrever-se o sistema sob três diferentes perspectivas: estrutural, comportamental e funcional.

O primeiro critério de comparação se dá em relação à perspectiva estrutural, ou seja, os recursos oferecidos pelas ferramentas para fazer a modelagem dos objetos, classes e suas diferentes formas de relacionamento (herança, agregação, uso, etc.). De acordo com este critério, constatou-se que as três ferramentas possuem recursos bastantes semelhantes no que diz respeito à inserção de classes e seus relacionamentos

no modelo, apesar de existirem diferenças conceituais entre os modelos, principalmente no definido pelo ObjecTime. Uma desvantagem do SIMOO em relação às outras ferramentas é que ele não oferece ao usuário campos para inserção de comentários a respeito das classes e dos relacionamentos, o que é bastante desejável em termos de documentação. Entretanto, uma vantagem do SIMOO em relação às outras ferramentas ocorre pela utilização de um diagrama de instâncias. Este diagrama permite eliminar ambigüidades presentes em diagramas que incluem somente classes.

No que diz respeito à descrição de comportamento, são encontradas semelhanças entre as ferramentas. Apesar do SIMOO não adotar diagrama de transições de estado para representar o comportamento interno dos objetos como as outras ferramentas, o que é uma desvantagem, ele possibilita o mapeamento direto de mensagens para operações. Em relação à capacidade de representação da interação entre os objetos que constituem o modelo, no Rational Rose e no ObjecTime é adotado o princípio dos diagramas de eventos, que permitem representar o comportamento de um determinado conjunto de objetos no momento da chegada de uma nova mensagem. No SIMOO isto fica por conta do próprio usuário do ambiente, que deve especificar através de código em C++ o comportamento dos objetos no momento da chegada de um evento. No geral, todas as ferramentas consideram o sistema como um conjunto de objetos concorrentes, os quais podem operar em paralelo e ainda comunicar-se uns com os outros.

Naturalmente que, por ser um ambiente projetado para simulação, o SIMOO oferece a facilidade de permitir a observação do comportamento do modelo construído através de uma simulação/animação. O ObjecTime também possui esta característica, com a desvantagem de não permitir a animação do modelo, mas somente a visualização das transições de estados. A capacidade de simular um modelo de objetos tem como principal vantagem proporcionar a visualização do comportamento do sistema antes da sua implementação, possibilitando a correção de eventuais erros que possam aparecer. Esta correção, em etapa anterior à implementação, traz uma redução significativa no custo do sistema em geral.

No que diz respeito à especificação dos requisitos temporais exigidos pelo sistema, todas as ferramentas são carentes de recursos neste aspecto. Pelo lado do Rational Rose, as poucas informações possíveis de serem especificadas ficam restritas apenas para a documentação do sistema. No SIMOO não existe a possibilidade de realizar-se tais especificações. Por fim, os mecanismos de timer oferecidos pelo

ObjecTime possuem limitações: os baixos níveis de abstração adotados forçam os analistas a descrever uma solução para o problema, não a sua especificação.

Finalmente analisa-se o código gerado por cada uma das ferramentas. Os códigos gerados pelo Rational Rose e pelo SIMOO são bastante parecidos em termos de estrutura, pois incluem a definição dos atributos, dos métodos e também das classes no qual o objeto faz referência. O código gerado pelo SIMOO é um pouco maior, pois inclui o suporte para simulação do modelo. Por sua vez, o código gerado pelo ObjecTime é bem mais extenso e complexo, pois inclui muitas chamadas às suas bibliotecas e estruturas de dados internas para possibilitar a execução do modelo.

Em [BEC97] também foram estudadas algumas linguagens de programação orientadas a objetos que suportassem a especificação de requisitos temporais. Dando continuidade a esta dissertação, nas próximas seções são descritas as principais características destas linguagens e uma comparação entre as mesmas é efetuada.

2.3 Linguagens de Programação para Tempo Real

2.3.1 Linguagem AO/C++

A linguagem AO/C++ [PER94b; SAN97] foi idealizada durante o desenvolvimento de alguns estudos de caso. Os resultados deste trabalho mostraram que a união dos conceitos de orientação a objetos com as características de sistemas operacionais de tempo real conduzem 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 de tempo real (UNIX-TR) orientado a processos, caso do QNX. Para tanto é utilizado o conceito de objetos ativos, i.e. possuem a capacidade de operar concorrentemente, assumindo assim que cada objeto será mapeado para um processo UNIX-TR.

Os objetos ativos em AO/C++ surgem pela criação de uma instância de uma classe ativa, após ter sido pré-processada pelo programa *parser*. As classes ativas devem ser criadas em dois arquivos separados (exclusivos para cada classe): a estrutura da classe é declarada em um arquivo de cabeçalho (*.ph) e a definição dos seus métodos é

feita em um segundo arquivo (*.pC). Todas as classes ativas herdam as propriedades de uma classe básica, a classe tempo real, que 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. 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. Este construtor é uma característica particular da linguagem, que permite combinar o poder dos compiladores C++ existentes com o modelo concorrente de características tempo real dos processos QNX.

As instruções e funções definidas por 'public:' são consideradas interfaces dos objetos ativos. 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. O pré-processador da linguagem torna estas chamadas remotas transparentes, modificando o corpo das funções de interface. Devido a esta característica, quando um objeto ativo chama uma função de outro, a chamada é convertida para uma mensagem do tipo 'send' fornecida pelo QNX. No receptor, a mensagem é então decodificada e a função correspondente é ativada. A configuração dos processos do sistema em tempo de execução, obtida pela utilização de objetos ativos, é mostrada na figura 2.6.

Para este mapeamento automático da interface é decisivo se a operação na classe ativa espera um valor de retorno definido ou não. Se o valor de retorno for um tipo definido 'void' ou se não houver tipo definido, a troca de mensagens é mapeada assíncrona. Se houver tipo definido, a comunicação é mapeada síncrona. No caso de comunicação síncrona, o objeto cliente fica bloqueado até que o objeto servidor processe a mensagem e retorne os dados resultantes.

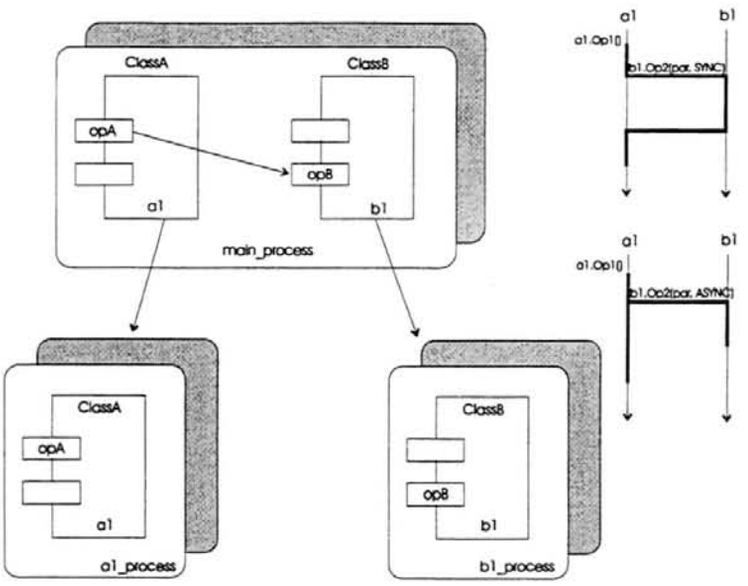


FIGURA 2.6 - Configuração dos processos QNX em tempo de execução

Construtores especiais, semelhantes aos utilizados em linguagens de tempo real de alto nível (como PEARL), também foram adicionados ao AO/C++ para dar mais poder à linguagem. Estes construtores permitem a definição de processos com ativação periódica, garantindo os seguintes recursos:

- ativação cíclica de funções dos objetos;
- definição de *deadlines* e *timeouts* (tanto no lado cliente como no lado servidor), associados a blocos de instruções.

2.3.2 Modelo de Objetos RTO.k

O modelo de objeto RTO.k [KIM94a; KIM94b] foi desenvolvido para auxiliar a construção dos chamados *hard real-time distributed computer systems* (DCS), que são sistemas onde os aspectos temporais são de vital importância. A principal característica na qual se baseia este modelo são as facilidades para descrever e garantir o atendimento a aspectos temporais, oferecendo aos objetos serviços tempo-dependentes.

O desenvolvimento do modelo de objetos RTO.k adota as seguintes metas principais: o princípio da estruturação uniforme dos programas, tanto para os DCS de tempo real quanto para suas aplicações em ambientes de simulação, e a garantia de oferecer aos usuários a capacidade de definir serviços com tempo de duração

determinado durante o projeto. Quatro motivos essenciais tornam o modelo de objetos RTO.k uma extensão do modelo clássico:

1) Clara diferenciação entre os métodos de objeto ativados por instantes de tempo (*time-triggered object methods*), também chamados métodos espontâneos (*spontaneous methods – SpM's*), dos convencionais métodos de objeto ativados por mensagens (*message-triggered object methods*), também chamados métodos de serviço (*service methods – SvM's*).
2) Existência de um "inibidor" (*constraint*) de concorrência, que impede conflitos entre os *SpM's* e os *SvM's*. Basicamente, a execução de um serviço ativado por um cliente é permitido somente quando o potencial *SpM* conflitante não estiver sendo executado. Esta restrição é denominada *basic concurrency constraint* (BCC). Desta forma, os métodos ativados por instantes de tempo recebem maior prioridade de execução do que os métodos ativados por mensagens.
3) Imposição de um *deadline* (tempo máximo de execução) a todo método de um objeto RTO.k.
4) Por fim, os dados em tempo real, representados pelos atributos de objetos RTO.k, tornam-se inválidos após expirar seu "prazo de validade", condição esta chamada *maximum validity duration*.

A figura 2.7 apresenta a estrutura básica de um objeto RTO.k. Em cada objeto são definidos os métodos espontâneos e de serviço, a capacidade de acesso a outros objetos RTO e o espaço dos dados utilizados pelo objeto (*object data store* - ODS). Nessa figura também estão representadas as filas de espera dos SpM's e a fila de chamadas de clientes para os SvM's.

Os tempos de disparo (*triggering times*) para os SpM's devem ser completamente especificados no momento do projeto do sistema. Estas constantes de tempo real aparecem no primeiro parágrafo na especificação de um SpM, sendo conhecido por seção de *autonomous activation condition* (AAC). O mecanismo de execução usa o AAC de um SpM para gerar listas de execução de eventos futuros para o SpM nos tempos apropriados e coloca-os na fila de reserva (*reservation queue*) do SpM. Cada fila de reserva é uma fila ordenada onde o próximo evento da lista está no inicio da mesma.

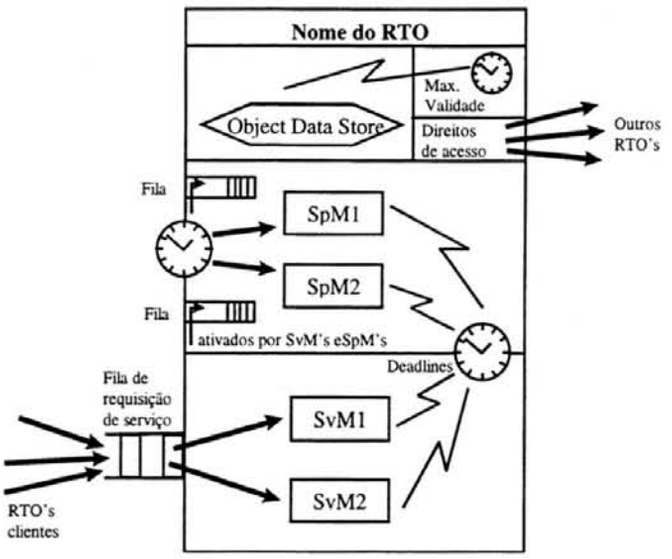


FIGURA 2.7 - Estrutura de um objeto RTO.k

Uma filosofia subjacente de projeto do modelo de objeto RTO.k é a que um DCS de tempo real sempre terá a forma de uma rede de objetos RTO.k. Objetos RTO.k interagem através de chamadas de objetos clientes para SvM's em objetos servidores. Tanto um SpM, quanto um SvM, em um objeto cliente pode fazer esta chamada. Para facilitar operações altamente concorrentes entre objetos servidores e clientes, chamadas (solicitações de serviços) não-bloqueantes (ou simplesmente assíncronas) juntamente com as convencionais chamadas bloqueantes podem ser feitas para os SvM's.

A abordagem adotada no modelo de objeto RTO.k para controle de concorrência exige a declaração explícita da porção do ODS a ser usada por cada objeto. Além disso os métodos devem especificar o tipo de acesso a esta região que poderá ser efetuado: leitura, escrita ou leitura/escrita. Isto permite a detecção de prováveis conflitos entre métodos. Caso o método não especifique o tipo de acesso ao ODS, será adotado um monitor CREW (*concurrent-read & exclusive-write*) para fazer o controle de acesso.

2.3.3 Linguagem Java Tempo Real

Diferente da maioria das linguagens projetadas para a programação em tempo real, Java Tempo Real [NIL95; NIL96a; NIL96b], ou Java-TR, simplifica a programação para permitir aos desenvolvedores escreverem códigos que requisitem o cumprimento seguro das restrições temporais envolvidas. Apesar de muitas manifestações contrárias, Java oferece muitas funcionalidades aos sistemas de tempo

real. Combinando implementações especiais da máquina virtual Java com o código escrito para cumprir as convenções especiais para descrição das atividades tempo real, é possível desenvolver aplicações em conformidade com as restrições de execução exigidas pelas aplicação que possuem um determinismo temporal.

Java-TR proíbe certas construções legais em Java, além de oferecer à linguagem semântica e sintaxe adicionais relacionadas à definição de restrições temporais. Esta extensão é implementada por um pré-processador especial (RTJPP - *Real-Time Java Preprocessor*) que converte o código Java-TR em código Java. Este código é então traduzido para *byte-codes* tradicionais (espécie de código intermediário) por um compilador Java qualquer. Assim, os *byte-codes* resultantes estão prontos para a execução em uma máquina virtual Java tradicional (*Java Virtual Machine*) ou em uma máquina virtual especialmente projetada para tempo real (*Real-Time Java Virtual Machine*).

No ambiente Java-TR, existem diferentes tipos de atividades de tempo real, que podem ser individualmente caracterizadas como cíclicas, esporádicas, espontâneas ou *ongoing*. As definições de cada uma destas atividades são mostradas a seguir:

- As tarefas periódicas ou cíclicas são invocadas em períodos regulares fixos durante o decorrer da execução da atividade de tempo real.
- As tarefas esporádicas, por outro lado, não têm um período de execução fixo. Estas tarefas ocorrem em resposta a eventos externos. Quando as respostas a estes eventos são processadas dentro de um tempo específico, a tarefa é descrita como sendo uma tarefa de tempo real esporádica.
- Tarefas espontâneas são similares às esporádicas pois também atuam em resposta a eventos externos. Porém não há limite máximo na freqüência de execução destas tarefas. O executivo de tempo real permite a execução deste tipo de tarefa somente se existirem recursos suficientes para tal.
- As tarefas contínuas ou *ongoing*, diferentemente dos outros tipos de tarefas, caracterizam-se por serem suspensas e retomadas, em vez de reiniciadas a cada período de execução da atividade.

A linguagem Java-TR consiste de uma combinação de bibliotecas de classes especiais, protocolos de comunicação padronizados fornecidos pelas bibliotecas e

também de algumas estruturas adicionadas à sintaxe padrão de Java. A ampliação do Java 1.1 é por conta de duas estruturas de controle, que são mostradas a seguir:

- *Timed*: identifica um segmento de código, cuja execução tem tempo limitado.
- *Atomic*: identifica um segmento de código atômico, i.e. ou a execução é realizada por completo ou não é efetuada. Adicionalmente, atomicidade implica em sincronização em termos de objetos dentro do qual o segmento de código atômico aparece.

O pacote de Java para tempo real oferece uma biblioteca de classes adicionais para suportar a implementação das diferentes atividades citadas anteriormente. Esta biblioteca é denominada PERC API, e pode ser referenciada em [NIL97].

2.3.4 Comparação das Principais Características das Linguagens

No que diz respeito à capacidade de distribuição das aplicações, cada linguagem faz uso de um modelo de referência distinto para implementar esta característica, o que torna evidente as diferenças entre as três abordagens. A linguagem AO/C++ faz o mapeamento de um modelo orientado a objetos 'logicamente distribuído' para o modelo 'fisicamente distribuído' de um sistema operacional UNIX-TR orientado a processos, transformando instâncias de classes em processos e chamadas de funções em mensagens. Os recursos de comunicação e sincronização são disponibilizados pela sua biblioteca e pelo sistema operacional QNX. Recursos de controle de acesso a dados não são fornecidos. Já o modelo de objetos RTO.k tem como plataforma um micro *kernel* chamado DREAM (*Distributed Real-time Ever Avaliable Micro-computing*) [KIM95], que roda em microcomputadores conectados por uma rede local executando o sistema operacional Windows-NT. Uma biblioteca de classes [KIM96] é oferecida para suportar comunicação entre os processos, mecanismos de ativação e sincronização e controle de acesso a dados compartilhados. A linguagem Java-TR, por sua vez, é independente de plataforma e o seu ambiente de execução consiste de máquinas virtuais, que podem estar distribuídas ao longo da rede Internet. Todo o suporte para comunicação entre processos, incluindo ativação e sincronização, está embutido na linguagem Java.

Uma limitação do AO/C++ é a ausência de chamadas assíncronas de serviços que retornam um valor, pois um objeto AO/C++ sempre fica bloqueado esperando a resposta do objeto que executa o método chamado. O modelo RTO.k permite chamadas

síncronas e assíncronas de serviços. Além disso, possui um sistema de transferência de chamada que faz que um método cliente possa solicitar um serviço de um método servidor, e receber o resultado de outro servidor, que pode estar localizado em um objeto diferente daquele onde estava o primeiro servidor chamado, pois este modelo permite mensagens entre processos. Java-TR também suporta tanto chamadas síncronas como assíncronas, sendo que o servidor pode estar localizado em qualquer nó da rede.

Em relação ao suporte para a especificação de restrições temporais, todas as linguagens permitem que uma operação seja executada repetidamente em intervalos de tempo específicos: AO/C++ possui blocos especiais para definição de operações cíclicas; no modelo RTO.k são utilizados os métodos SpM juntamente com uma cláusula chamada 'every' para determinar intervalos de tempo constante entre cada execução do método; em Java-TR isto é possível se a atividade relacionada for definida como cíclica. AO/C++ também apresenta construtores de alto nível para permitir a especificação de *deadlines* e *timeouts*. Restrições de tempo também são apresentadas no modelo RTO.k, incluindo duração da validade dos dados. Na linguagem Java-TR as restrições são implementadas através da constante 'Timed', que especifica uma janela de tempo para uma determinada seqüência de operações.

A principal diferença entre as linguagens na especificação de restrições temporais se dá no modelo RTO.k, que implementa métodos espontâneos os quais são ativados por um relógio interno em intervalos de tempo bem específicos, definidos durante o projeto. Uma carência encontrada no RTO.k e na linguagem Java-TR é a falta de um mecanismo para tratamento de exceções, pois uma vez que expira um determinado intervalo de tempo, é de interesse que o programa possa tomar uma ação preventiva. AO/C++ oferece tratamento de exceções através de um bloco de instruções que é executado caso o *deadline* de uma função tenha sido ultrapassado.

Para finalizar, considera-se o grau de dificuldade presente no aprendizado de cada linguagem. Apesar das três linguagens serem consideradas uma extensão, RTO.k e Java-TR utilizam bibliotecas sobre as quais o programador deve possuir um elevado grau de conhecimento, pois é nas mesmas que são acrescentadas a maioria das facilidades destas linguagens. Já no AO/C++ o programador não precisa ter conhecimento das chamadas de funções feitas à sua biblioteca, mas somente dos construtores de alto-nível que foram adicionados ao C++ para especificar a definição das classes ativas e das restrições temporais, o que facilita o seu aprendizado.

3 Ambiente SIMOO-RT

3.1 Introdução

A partir da análise do estado-da-arte descrita de forma sucinta nos capítulos anteriores e discutida em maiores detalhes em [BEC97], observou-se a carência de um ambiente/ferramenta que incorporasse as características "desejáveis" num ambiente de desenvolvimento para sistema de tempo real: modelagem OO, especificação de restrições temporais, simulação/animação e geração de código executável.

Diante deste contexto, nesta dissertação é proposta a criação de um ambiente integrado denominado SIMOO-RT, que visa estender a ferramenta SIMOO de maneira a torná-la mais adequada ao desenvolvimento de sistemas de tempo real. A extensão aqui proposta visa suprir as deficiências observadas no que diz respeito à representação de requisitos temporais e também à geração de código executável em uma linguagem que suporte os requisitos adicionados.

A decisão em utilizar uma ferramenta já existente como base para a realização deste trabalho foi motivada pelo fato de ser inviável a concepção de um ambiente na sua totalidade, partindo da estaca zero, dentro do prazo proposto. Além disso, um conjunto de fatores influenciaram na escolha do SIMOO como base para a extensão proposta, dentre os quais destacam-se: a adequação da sua notação para a criação de modelos OO, o suporte à simulação/animação do modelo, a capacidade de geração de código voltado para simulação e finalmente devido à possibilidade de extensão deste ambiente, uma vez que o seu código fonte encontra-se disponível.

Nesta primeira versão, a linguagem alvo explorada é o AO/C++, escolhida por possuir assertivas simples e adequadas para representação dos principais requisitos temporais existentes em uma aplicação de tempo real. Somado a isto, a capacidade de distribuição da aplicação oferecida por esta linguagem, através do uso de objetos ativos, torna ainda mais conveniente a sua utilização. Contudo, deve ficar claro que o ambiente como um todo deve ser independente de linguagem.

Nas próximas seções, as facilidades para especificação de restrições temporais, juntamente com a geração de código em AO/C++ pelo ambiente desenvolvido, são apresentadas e discutidas em detalhes.

3.2 Especificação dos Requisitos Temporais

Uma vez definida a linguagem AO/C++ como alvo da geração de código, acrescentou-se ao SIMOO-RT facilidades para descrição dos requisitos temporais incorporados pela mesma. Conforme citado anteriormente na seção 2.3.1, o AO/C++ suporta a especificação de operações periódicas/cíclicas e operações com *deadlines* (tempo máximo de execução). A partir do valor de tempo informado pelo usuário e através do controle feito em tempo de execução, estas operações permitem, respectivamente, definir métodos de classe com ativação cíclica e métodos com código de tratamento especial às operações que não cumprem o *deadline*.

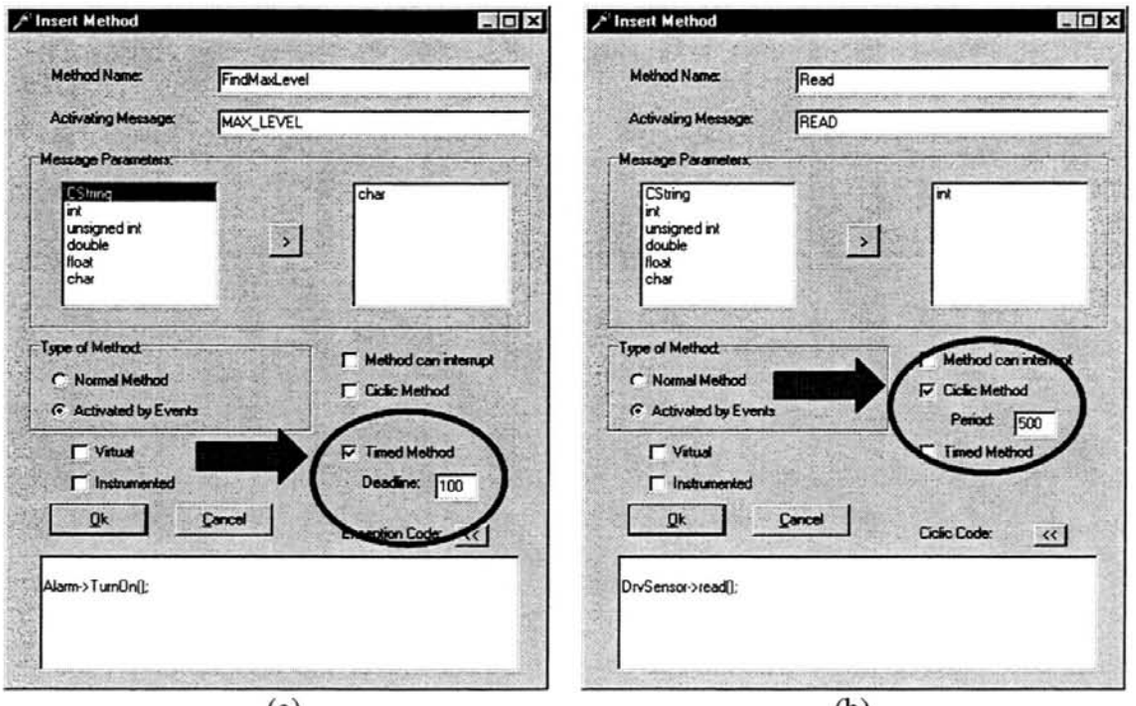
Ambos os requisitos são adicionados ao modelo através da janela de definição dos métodos, conforme exibido na figura 3.1. A seleção da opção 'timed method' (fig. 3.1a) indica a definição de uma operação com tempo máximo de execução (*deadline*). Esta opção permite ao usuário especificar um código de tratamento de exceção, que será executado caso o *deadline* não seja cumprido. Enquanto o campo 'exception code' oferece acesso à janela de edição do código de exceção, a determinação do tempo máximo de execução é feita no campo 'deadline'.

De maneira análoga à restrição citada anteriormente, a definição de uma operação periódica também é feita através da seleção de uma opção especial, neste caso denominada 'cyclic method' (fig. 3.1b). Assim como o código de tratamento de exceções, o código a ser executado periodicamente também é definido nesta janela, porém através do campo 'cyclic code'. Na janela de edição do método é permitido ao usuário definir o código de inicialização da operação, a ser executada antes da primeira chamada ao código restrito do ciclo. O campo 'Period' permite a entrada do período de ativação do ciclo.

Comparando-se o mecanismo de especificação das restrições temporais incorporado ao SIMOO-RT com aqueles adotados por outras ferramentas da área, como o ObjecTime, constata-se que o primeiro oferece ao usuário maior facilidade para acrescentar as restrições ao modelo. Isto se deve ao fato da capacidade de associar as restrições diretamente aos métodos das classes, ao contrário do ObjecTime, que sugere a programação de *timers* para monitorar os requisitos temporais.

Para inserir as informações a respeito das restrições temporais no SIMOO-RT, além da redefinição das janelas de edição, necessitou-se alterar algumas estruturas de

dados da ferramenta original. As alterações promovidas constaram da adição de 'flags' especiais nas estruturas, que servem para indicar se os métodos possuem ou não algum requisito temporal associado. Também foi necessário acrescentar variáveis para armazenar os valores de temporização informados durante a criação do método e os códigos especiais associados aos métodos. Esta alteração refletiu-se em grande parte do código original da ferramenta SIMOO, principalmente nos módulos de tratamento de arquivos, que tiveram de ser reescritos para se adaptarem às modificações incorporadas.



(a) (b)

FIGURA 3.1 - Especificações de *deadline* e período

Considerando os diferentes níveis de detalhamento que uma aplicação de tempo real pode possuir, verifica-se que os modelos projetados no ambiente desenvolvido pertencem a um nível mais elevado de abstração. Portanto, deve ser ressaltado que as restrições temporais incorporadas na ferramenta SIMOO-RT não esgotam o conjunto de restrições que podem ser impostas e/ou monitoradas em uma aplicação de tempo real. Para fins de documentação, considera-se oportuno citar algumas destas restrições.

Analisando uma aplicação de tempo real, verifica-se que os atributos das classes devem ser normalmente atualizados dentro de períodos específicos de tempo. Muitas vezes o valor contido em algum destes atributos pode representar um ponto crítico que influencia no bom andamento do sistema. Portanto, garantir a validade destes dados é

uma necessidade em um sistema de tempo real. Uma restrição que poderia ser utilizada para garantir a consistência destes dados seria definir para cada atributo da classe um tempo máximo de atualização, que torna o seu valor inválido após expirar este tempo. Algumas linguagens de programação voltadas para aplicações de tempo real, como o modelo RTO.k discutido na seção 2.3.2, adotam esta restrição. Particularmente no RTO.k ela é chamada de *maximum validity duration*.

Voltando a atenção para a execução dos métodos da classe, para garantir a execução dos requisitos temporais impostos, seria interessante monitorar certas características (vide [FLO98]), tais como: momento de chegada, momento de liberação, *jitter* de liberação (latência) e tempo de resposta. Momento de chegada é o instante de tempo em que a aplicação solicita a execução do método. Já o momento de liberação ocorre quando o sistema operacional inicia efetivamente o processamento das suas instruções. A latência é a diferença de tempo entre o momento de chegada e o momento de liberação. Por fim, o tempo de resposta é o tempo entre o momento de chegada do método e o término da sua execução. A figura 3.2 ilustra a ocorrência destas características.

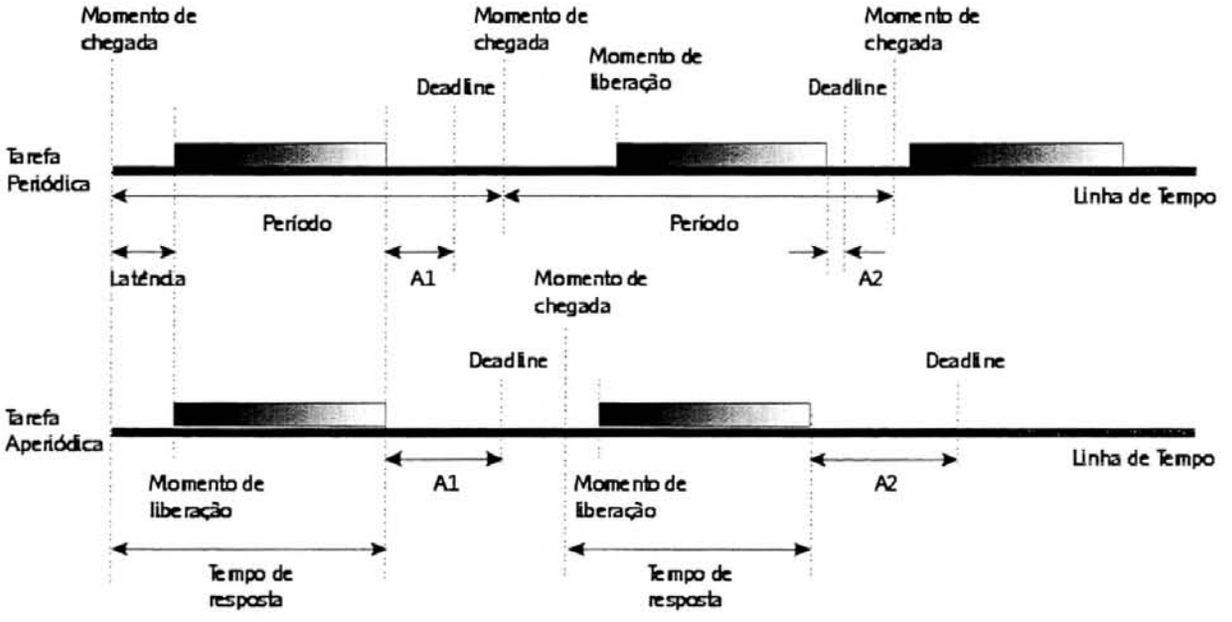


FIGURA 3.2 - Características temporais de uma aplicação de tempo real

Para finalizar esta seção, considera-se importante em uma aplicação de tempo real a incorporação de regras para garantir que certos eventos só sejam atendidos mediante algumas circunstâncias pré-definidas, que poderia ser a execução de um outro

evento. Por exemplo, no estudo de caso apresentado no capítulo 2, o manipulador só poderia pegar a peça da esteira após o aviso gerado pelo sensor. Estas regras são chamadas de causalidade/relação de precedência.

3.3 Geração de Código Executável

3.3.1 Criação de Programas AO/C++ a partir da Especificação

Para facilitar a descrição das etapas de geração do código executável, inicialmente é feita uma consideração a respeito da estrutura dos programas AO/C++ em relação aos modelos definidos no ambiente SIMOO-RT. O primeiro aspecto a ressaltar é a diferenciação entre as classes ativas e as classes passivas, definida no momento da sua criação no ambiente de modelagem. Estas classes se distinguem pelo ícone do canto direito superior, conforme exibido na figura 3.3.

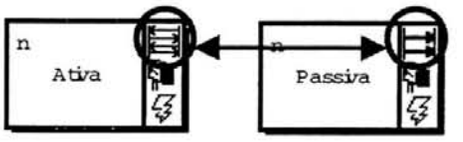


FIGURA 3.3 - Diferenciação entre as classes ativas e as classes passivas

No AO/C++, as classes ativas oferecem mecanismos para operar de maneira concorrente e distribuída, podendo ser referenciadas por objetos localizados em outros processos do sistema operacional, os quais podem encontrar-se situados em qualquer nó da rede. Em contrapartida, as classes passivas são similares às classes C++ usuais, uma vez que não passam pelo pré-processador e podem se comunicar somente com as classes/objetos implementados no mesmo processo. A figura 3.4 ilustra as possíveis configurações da comunicação entre dois objetos, sendo os mesmos ativos ou passivos.

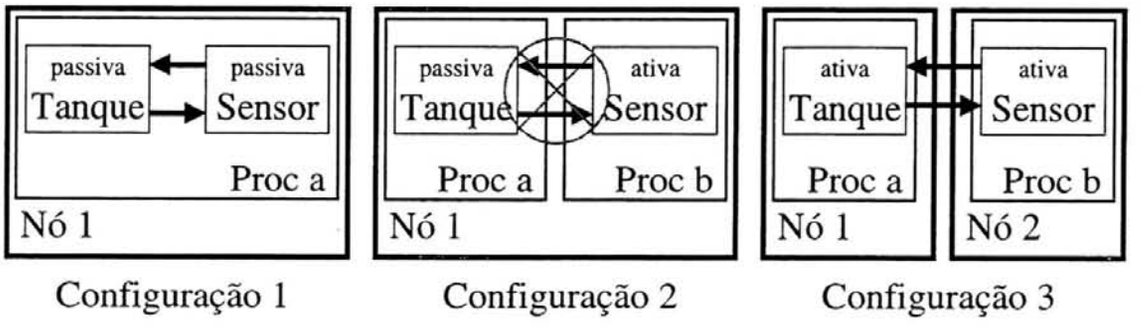


FIGURA 3.4 - Possibilidades de comunicação entre objetos ativos e passivos

Outro ponto a ressaltar diz respeito à estruturação dos modelos. Similarmente ao proposto em SIMOO, os modelos SIMOO-RT são construídos de maneira hierárquica, sendo que diversos tipos de associações entre as classes podem ser identificadas: agregação, conhecimento, herança, etc. Classes definidas em um determinado nível são consideradas "componentes agregados" da classe de mais alto nível que está sendo detalhada. Desta forma, um modelo SIMOO-RT é representado por uma única classe que inclui todas as demais.

Na linguagem AO/C++ cada classe é definida em dois arquivos distintos: um arquivo *header* contendo suas definições (<Classe>.ph) e outro contendo o corpo dos métodos (<Classe>.pC). Em virtude destas propriedades, cada classe do modelo é representada por dois arquivos AO/C++ distintos, conforme ilustrado na figura 3.5.

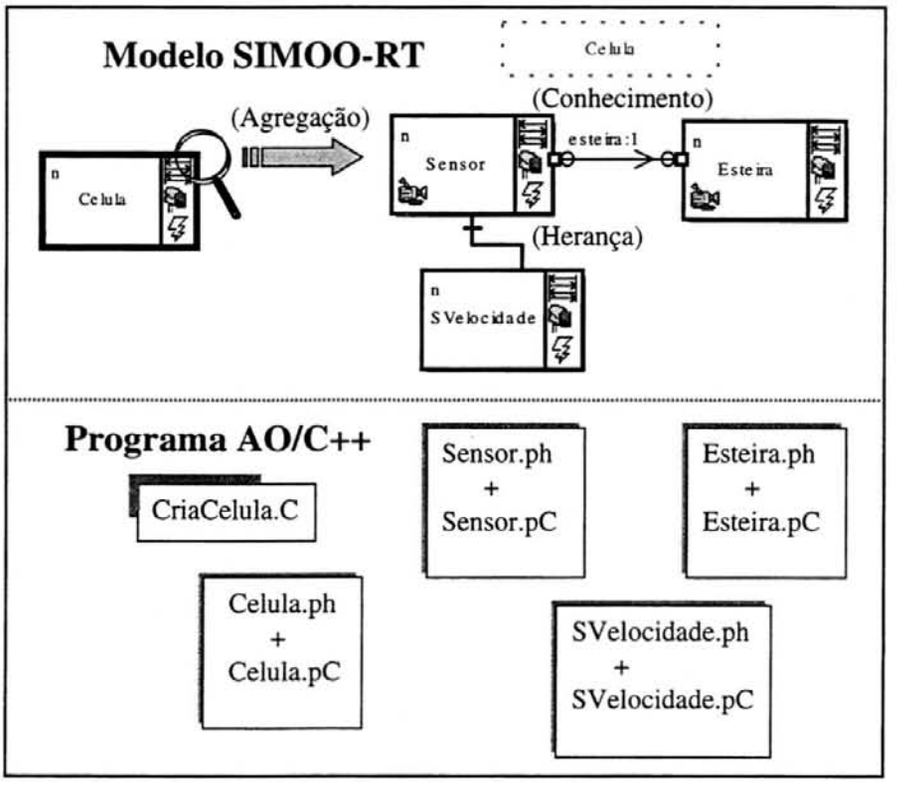


FIGURA 3.5 - Estrutura do programa gerado em relação ao modelo SIMOO

No SIMOO-RT, as associações das classes e as instâncias pertencentes ao sub-nível derivado do detalhamento da mesma são representadas pelos chamados "atributos de sistema". Na figura 3.6 são mostrados os atributos da classe 'Sensor' (fig. 3.5), dentre os quais destaca-se o atributo de sistema derivado da sua associação de conhecimento

com uma instância da classe 'Esteira'. Já a figura 3.7 apresenta os atributos da classe 'Celula' (fig. 3.5), destacando os atributos de sistema resultantes da sua associação de agregação, dado o diagrama de instâncias derivado do seu detalhamento. Além destes atributos de sistema, classes modeladas em SIMOO-RT também poderão possuir atributos definidos pelo usuário.

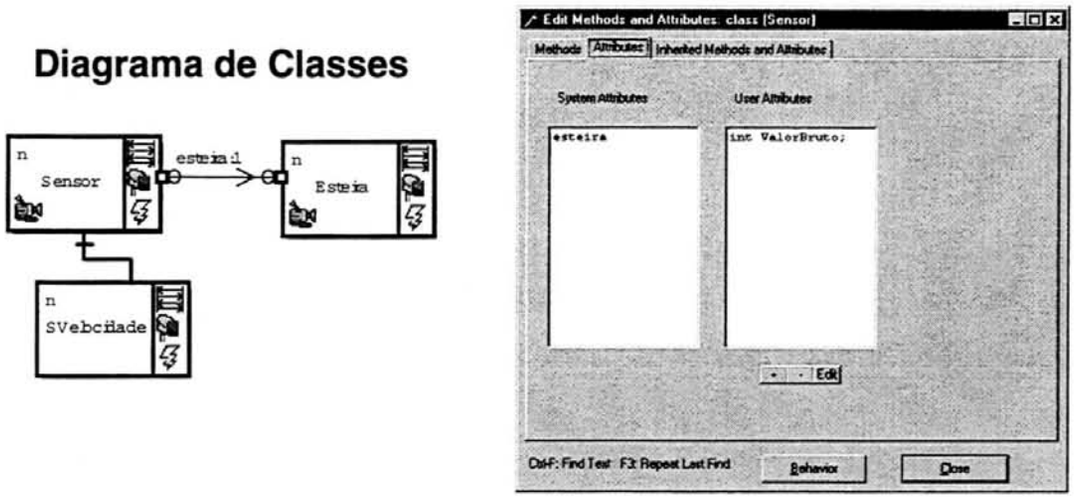


FIGURA 3.6 - Atributos da classe 'Sensor'

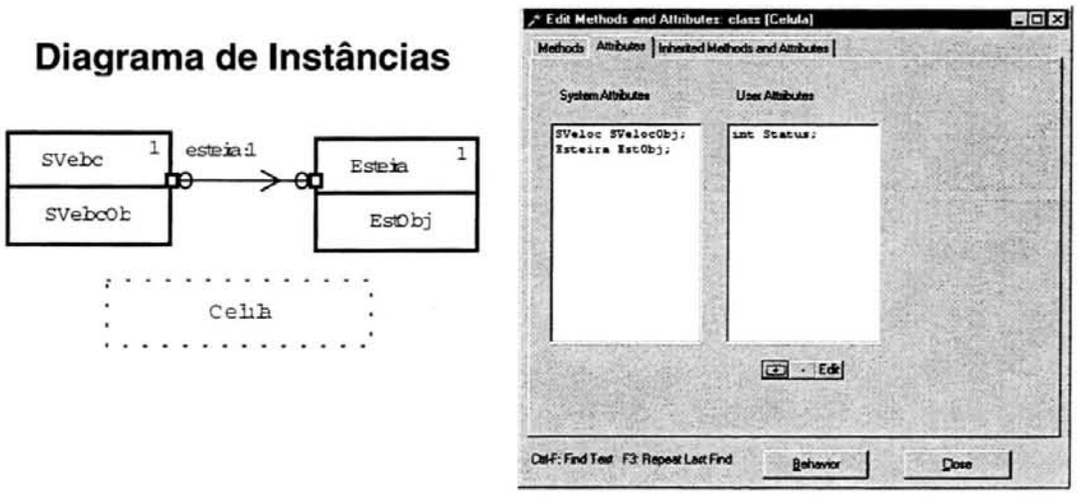


FIGURA 3.7 - Atributos da classe 'Celula'

Conforme já mencionado, instâncias de classes ativas podem executar de forma concorrente e/ou distribuída com outras instâncias ativas. Estas instâncias são geralmente mapeadas para tarefas do sistema operacional tempo-real sobre o qual executam os programas gerados automaticamente pelo ambiente SIMOO-RT. Assim sendo, 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. Suas especificações são permitidas através da janela de criação de instâncias, conforme exibido na figura 3.8.

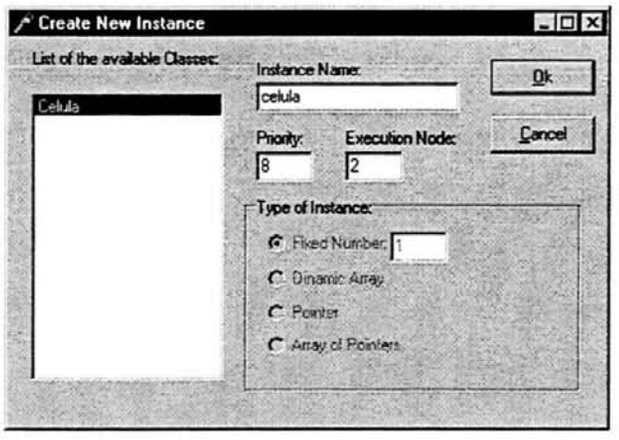


FIGURA 3.8 - Atributos dos objetos ativos

3.3.2 Geração das Classes AO/C++

A geração do código da aplicação QNX, cuja visão geral é exibida na figura 3.9, é feita percorrendo a estrutura de dados do SIMOO-RT de maneira *bottom-up*, a fim de analisar individualmente cada classe do modelo. Iniciando pelas classes com maior nível de profundidade, primeiramente é produzido o arquivo *header* (<classe>.ph), que contém a definição de todos os atributos e métodos. Neste arquivo, as associações da entidade são expressas em termos de código AO/C++. Nas associações de agregação, os objetos agregados são definidos como instâncias pertencentes à classe agregadora. Já as associações de conhecimento são expressas através de referências especiais (vide seção 2.3.1), que "apontam" para a classe do relacionamento. Por fim as associações de especialização ou herança são expressas no mesmo formato padrão da linguagem C++.

Após a geração dos atributos de sistema, são gerados os atributos definidos pelo usuário e a declaração dos métodos da classe. A geração da declaração dos métodos é feita em duas etapas: num primeiro momento são gerados automaticamente os métodos responsáveis pela criação, gerência e comunicação dos objetos ativos, a serem discutidos posteriormente nesta seção. Após, são produzidos os métodos definidos pelo usuário, que devem sofrer uma verificação prévia a fim de checar se os mesmos têm ou não alguma restrição temporal, i.e. têm um *deadline* ou período associado. Caso o

método possua alguma restrição ele deve ser definido com um parâmetro especial, dependendo da temporização associada (vide fig. 3.1). A geração da definição dos métodos finaliza a definição do arquivo *header*, que terá o formato apresentado na figura 3.10.

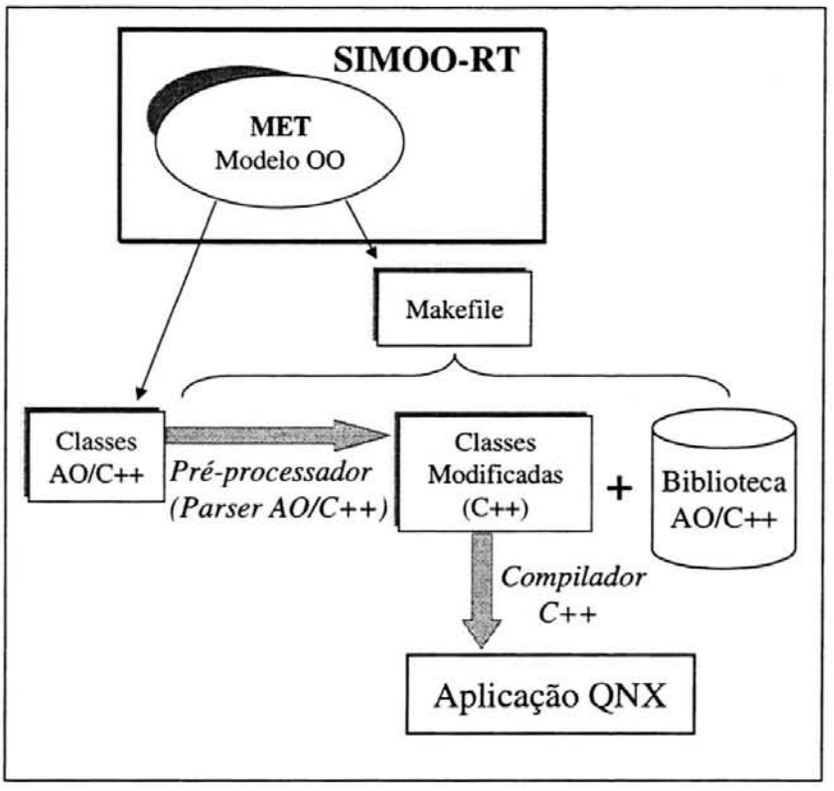
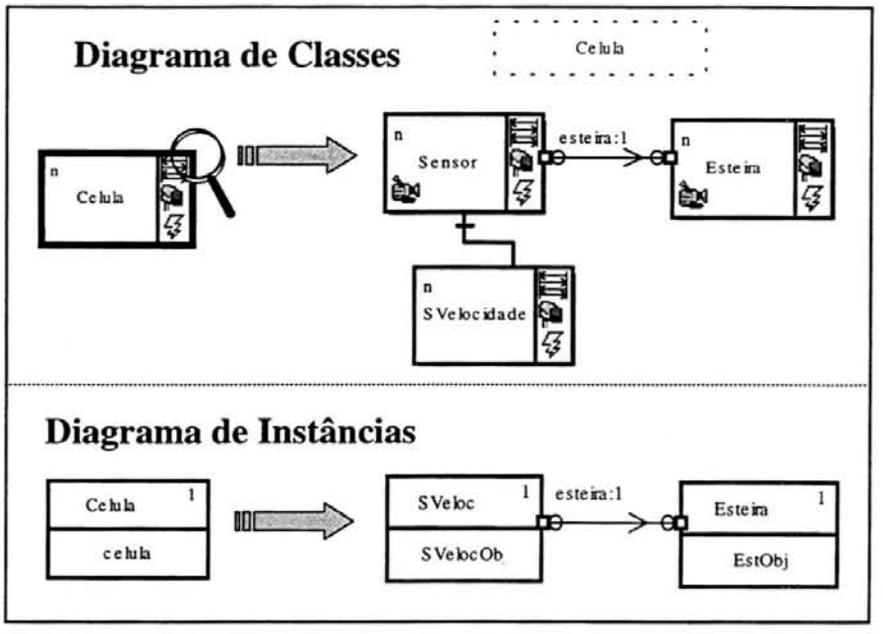


FIGURA 3.9 - Visão geral do código gerado e do seu processo de compilação

```
//Arquivo Sensor.ph
active class Sensor{
    ... // atributos do usuário
    protected:
    <classe> <Obj>; // atributos das associações de agregação
    <classe> <*pointer>; // atributos das associações de conhecimento
    public:
    ... // métodos automáticos
    void <nomeOperacao>(cycle_t); // métodos do usuário (cíclico)
    <tipo> <nomeOperacao>( dead_l); // métodos do usuário (deadline)
}
```

FIGURA 3.10 - Estrutura do arquivo *header* AO/C++

A segunda etapa do processo de geração de código consta da criação do arquivo (<classe>.pC) que contém o "corpo" dos métodos definidos anteriormente. Nesta fase são gerados primeiramente os métodos automáticos, dentre os quais destaca-se o método 'Start(char *pai)', encarregado de ativar e estabelecer a comunicação entre os objetos que encontram-se no sub-nível de hierarquia derivado do refinamento da classe que estiver sendo processada (associação de agregação), caso os mesmos existam.




```
// Arquivo Celula.pC
void Celula::Start(char *pai) {
    //Cria instancia concatenando o nome do PAI
    EstObj.create(pai + "_EstObj");
    SVelocObj.create(pai + "_SVelocObj");
    //Estabeleca as ligacoes entre as instancias:
    SVelocObj.Conecta_esteira(pai + "_EstObj");
    ... }

//Arquivo Sensor.pC
void Sensor::Conecta_esteira(char *nome_esteira) {
    while(esteira->link(nome_esteira)==-1) delay(1000);
}
```

FIGURA 3.11 - Disparo e estabelecimento de comunicação dos objetos AO/C++

A criação dos objetos ativos é feita através da invocação do método 'create(nome)', definido pela biblioteca AO/C++. Este método aceita como parâmetro o nome do processo correspondendo ao objeto ativo no sistema. Já para estabelecer a comunicação entre os objetos do modelo foi necessário definir um método especial, denominado 'Conecta_<ponto>(char *)'. Como no AO/C++ a comunicação entre objetos é feita através de referências para as classes ativas, este método contém a chamada do método 'link(nome)' (também da biblioteca AO/C++) que implementa a definição destas referências. Uma classe irá possuir tantos métodos de conexão quantos forem os seus pontos de ligação, i.e. com quantas classes ela deverá se comunicar. A figura 3.11 exemplifica a utilização das funções 'Start' e 'Conecta_<ponto>'.

Observando o código da função 'Start(char *pai)' exibido anteriormente (fig. 3.11), percebe-se que ela recebe como parâmetro uma *string* denominada 'pai'. Esta variável contém o nome do objeto que está disparando as instâncias, i.e. o nome do objeto agregador das mesmas. De posse deste nome, ele é concatenado ao nome da instância criada, de maneira a constituir uma hierarquia unívoca de nomes no ambiente de execução. Esta hierarquia é iniciada pela instância do primeiro nível de refinamento do modelo, passando por todos os níveis intermediários até chegar ao objeto disparado.

Logo após a geração dos métodos automáticos, são produzidos os métodos definidos pelo usuário. Nestes, de maneira análoga ao procedimento efetuado durante a geração do arquivo *header*, é feita uma nova verificação para determinar se os mesmos possuem ou não restrições temporais. Caso alguma restrição seja verificada, um código especial é gerado a fim de possibilitar que a linguagem ofereça o tratamento adequado ao método. Na figura 3.12 são mostrados de maneira reduzida os códigos produzidos para os métodos 'Read()' (cíclico) e 'FindMaxLevel()' (com deadline), definidos na figura 3.1.

3.3.3 Mecanismos de Inicialização da Aplicação

Até o presente momento, foram tratadas apenas as estruturas individuais das classes do modelo, sem nenhuma referência a mecanismos de criação ou disparo da aplicação. Considerando que o ambiente proposto adota uma modelagem hierárquica e que cada classe contém mecanismos para disparar os objetos agregados num nível inferior, basta que seja criado o objeto do primeiro nível para que o restante das

entidades do modelo sejam automaticamente criadas e ativadas. Motivado por esta característica do ambiente, foi adotada a geração do programa 'Cria<objeto>.C', que contém as instruções necessárias para invocar a criação do objeto agregador do modelo. A figura 3.13 exibe o código do programa 'CriaCelula.C', gerado a partir do modelo ilustrado na figura 3.5.

- Operações cíclicas:

```
void Read(cycle_t){
    //codigo de inicialização
    ...
    begin_cycle
    //código do ciclo
    DrvSensor->Read();
    end_cycle
}
```

- Operações com *deadline*:

```
int FindMaxLevel( dead_l){
    begin_operation
    // Operação Normal
    ...
    end_operation
    begin_exception
    // Rotina de Exceção
    Alarm->TurnOn();
    end_exception
}
```

FIGURA 3.12 - Formato dos métodos com restrições temporais em AO/C++

```
#include "Celula.h"

void main(void){
    Celula celula(8, 2); //(prioridade, nó_execução)
    char *nome;
    nome = "/celula";
    celula.create(nome); //ativa a aplicação
    cout << "Objeto " << nome << " criado!" << endl;
    celula.Start(nome);
    while(getch() != 27) delay(500);
    celula.End( );
    cout << "Objeto " << nome << " destruido!" << endl;
    celula.stop( ); //encerra a aplicação
}
```

FIGURA 3.13 - Programa que ativa o objeto agregador do modelo

A figura 3.13 exibe duas funções especiais da linguagem AO/C++: a função 'create(nome)' e a função 'stop()'. Todas classes ativas herdam a operação 'create(nome)', responsável pela criação do processo concorrente UNIX-RT associado a um objeto ativo. Somente após a execução desta operação é que os métodos de uma classe ativa podem ser chamados. Para destruir um objeto ativo (seu processo), não é suficiente o término do mesmo no programa 'main' (ou no escopo onde ele foi criado). Antes disto deve-se chamar a função 'stop()', que faz com que o processo em que o objeto ativo está rodando termine.

Outro ponto que merece ser destacado no código mostrado na figura 3.13 é a operação de criação da instância da classe Celula. 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, conforme definido na figura 3.8. A indicação do nó da rede é que permite que o objeto execute de maneira distribuída. Porém, o fato do código gerado considerar os objetos do modelo como atributos da classe de nível superior inviabiliza a utilização deste mecanismo. Isto ocorre por uma limitação do compilador C++ do QNX, que impede a passagem de parâmetros ao construtor de classes instanciadas dentro de outras classes. Desta forma, caso fosse gerado um conjunto de classes AO/C++ com a mesma estrutura do modelo em SIMOO-RT, ou seja, um modelo hierárquico partindo de uma única classe representando o sistema, não seria possível distribuir a aplicação.

Várias alternativas são vislumbradas para solucionar o problema apresentado, dentre as quais três merecem ser destacadas. A primeira delas é implementar a ativação dos objetos de forma individual, i.e. gerar um programa 'Cria<objeto>' para cada instância/classe da aplicação. Porém, esta alternativa pode implicar na geração de um número grande de programas, dependendo do tamanho do modelo, consumindo assim mais espaço em disco e mais memória. Esta solução também tem como conseqüência uma descaracterização da aplicação em relação ao modelo, pois termina com o conceito de agregação.

Outra alternativa seria implementar um programa que ative todos os objetos da aplicação. Além de não adotar o conceito de agregação, esta alternativa aumenta a dificuldade de depuração dos programas, pois torna difícil isolar o objeto/processo que pode estar causando algum erro.

Por fim, a terceira possibilidade considerada é alterar o mecanismo de parametrização do AO/C++, tirando do construtor e passando a outro método (e.g. método 'create()') os parâmetros que contém as informações de inicialização do objeto. Apesar desta parecer a melhor das três alternativas, para ser efetivada ela necessita de alterações no pré-processador do AO/C++. Como tais alterações fogem ao escopo desta dissertação, implementou-se a geração de programas individuais de disparo para cada objeto. Todavia, também foi conservado o mecanismo original de ativação dos objetos do modelo.

Como o SIMOO-RT executa no ambiente MS-Windows e o código objeto de AO/C++ roda em QNX, é necessário transportar o código AO/C++ gerado para o ambiente QNX. Isto pode ser feito utilizando uma área de disco compartilhada entre o Windows e o QNX ou transportando o código gerado para uma máquina QNX via 'ftp'. Com o código produzido no sistema operacional QNX, basta invocar um arquivo 'Makefile' gerado automaticamente (vide próxima seção) que os arquivos fontes serão compilados e a aplicação executável será gerada.

3.3.4 Geração Automática do Projeto (Makefile)

Gerados os arquivos correspondentes ao modelo projetado, é necessário compilá-los para criar os executáveis que constituem a aplicação no sistema operacional QNX. Com o objetivo de facilitar este processo, juntamente com os arquivos citados anteriormente é também gerado um arquivo em lote do tipo 'Makefile' (padrão UNIX), que contém as informações necessárias para o pré-processador AO/C++ e o compilador C++ produzirem os programas objetos e executáveis que fazem parte da aplicação.

Indubitavelmente, a criação do arquivo 'Makefile' é uma das tarefas mais dispendiosas na programação em AO/C++, uma vez que devem ser informados todas as dependências existentes entre os diferentes arquivos que fazem parte da aplicação, sendo que algumas destas dependências advém dos relacionamentos entre as classes e outras dos arquivos gerados. Portanto, a geração automática deste arquivo no ambiente de desenvolvimento facilita de maneira significativa o trabalho dos programadores. Antes de descrever o processo de geração do 'Makefile' é necessário descrever as modificações nos arquivos fontes provocas pelo pré-processador (parser) da linguagem AO/C++.

O trabalho do *parser* é transformar as classes ativas definidas em AO/C++ para classes C++ acrescidas de chamadas à biblioteca de tempo real. Logo são geradas duas classes C++ com a mesma interface, porém com implementações distintas. A primeira, que permanece com o nome idêntico (e.g. class Sensor), apenas monta e envia as mensagens. Já a outra, cujo nome é adicionado da palavra 'Proc' (e.g. class SensorProc), contém os métodos a serem executados no processo onde a instância da respectiva classe atua. As duas novas classes geradas são definidas num mesmo arquivo (e.g. Sensor.h) e seus códigos são declarados em arquivos distintos, que seguem a mesma denominação das classes (e.g. Sensor.C e SensorProc.C). O processo UNIX-RT que executa os métodos de uma classe ativa possui o mesmo nome da classe acrescido da palavra 'Main' (e.g. SensorMain). A figura 3.14 ilustra o pré-processamento das classes ativas fazendo uso da classe Sensor.

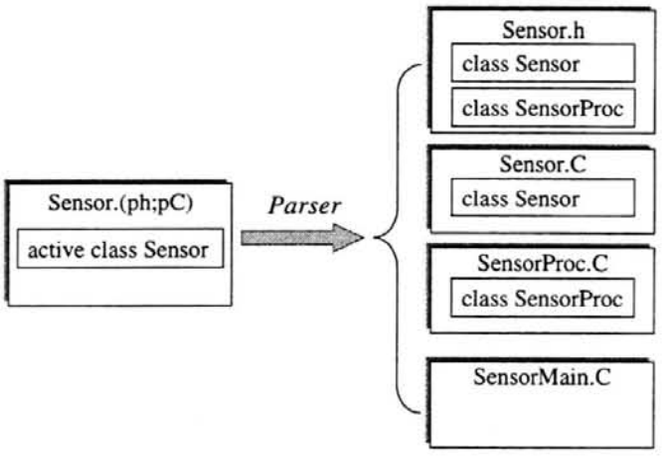


FIGURA 3.14 - Transformações provocadas pelo pré-processador AO/C++

Cabe ao 'Makefile' definir as regras (comandos + parâmetros) de pré-processamento das classes ativas e também as regras de compilação dos arquivos resultantes. Desta forma, na primeira linha do 'Makefile' é gravada uma instrução contendo os programas executáveis que são gerados ao final da sua execução. Para fins de esclarecimento, cada classe ativa do sistema corresponde a um programa executável, que corresponde ao processo UNIX-TR encarregado de realizar a computação das funções do objeto (e.g. SensorMain). Nas linhas seguintes são gravadas as instruções individuais de cada classe, contendo os comandos para invocar o pré-processador AO/C++ (parser) e o compilador C++ do QNX (Watcon). Na figura 3.15 é exibido, de maneira reduzida, o 'Makefile' produzido para o modelo da figura 3.5.

```
1a: all: CriaCelula EsteiraMain SensorMain SVelocMain CelulaMain

2a: CriaCelula: CriaCelula.o CriaCelula.C Celula.o AO.o
2b:     cc -o CriaCelula CriaCelula.C Celula.o AO.o

3a: CriaCelula.o: CriaCelula.C Celula.o
3b:     cc -c CriaCelula.C

4a: Sensor.h: Sensor.ph
4b:     parser Sensor -n

5a: Sensor.C: Sensor.pC Sensor.ph
5b:     parser Sensor -n

6a: SensorMain.o: SensorMain.C
6b:     cc -c SensorMain.C

7a: SensorProc.o: SensorProc.C
7b:     cc -c SensorProc.C

8a: Sensor.o: Sensor.h Sensor.C Esteira.h
8b:     cc -c Sensor.C

9a: SensorMain: AO.o SensorMain.o SensorProc.o Sensor.o Esteira.o
9b:     cc -o SensorMain AO.o SensorMain.o SensorProc.o Sensor.o Esteira.o
...

10a: CelulaMain: AO.o CelulaMain.o CelulaProc.o Celula.o Esteira.o Sensor.o SVeloc.o
10b: cc -o CelulaMain AO.o CelulaMain.o CelulaProc.o Celula.o Esteira.o Sensor.o SVeloc.o
```

FIGURA 3.15 - Exemplo de 'Makefile'

A fim de facilitar o entendimento do processo de compilação do código produzido, algumas considerações sobre o 'Makefile' exibido na figura 3.15 devem ser observadas. As linhas que antecedem a invocação do comando (terminação 'a') informam quais arquivos já devem ter sido processados antes da sua execução. Isto permite que eventuais alterações nos arquivos sejam detectadas pelo 'Makefile'. Nas instruções 2 e 3 é compilado o programa que dispara a instância da classe agregadora do sistema. As instruções 4 e 5 invocam o pré-processador AO/C++ para a classe 'Sensor', passando os parâmetros definidos pelo usuário no momento da geração do código (fig. 3.16). As instruções de 6 a 9 contêm os comandos de compilação dos arquivos produzidos após o pré-processamento. Deve ser dada uma atenção especial à geração dos arquivos de envio e recebimento das mensagens (instruções 8 e 9 respectivamente),

pois é neste momento que devem ser informadas as classes associadas àquela sendo processada. Por exemplo, para a classe 'Sensor' deve ser informada a sua associação com a classe 'Esteira'. A instrução 10 foi mantida para mostrar as associações da classe 'Celula'.

O gerador de código do SIMOO-RT permite ao usuário selecionar algumas opções de configuração do projeto. Estas opções se referem ao mecanismo de ativação dos objetos da aplicação e também aos parâmetros do pré-processador AO/C++. Maiores informações a respeito dos parâmetros de pré-processamento podem ser obtidos em [SAN97]. Aconselha-se àqueles usuários sem muita experiência a adotarem os parâmetros *default*.

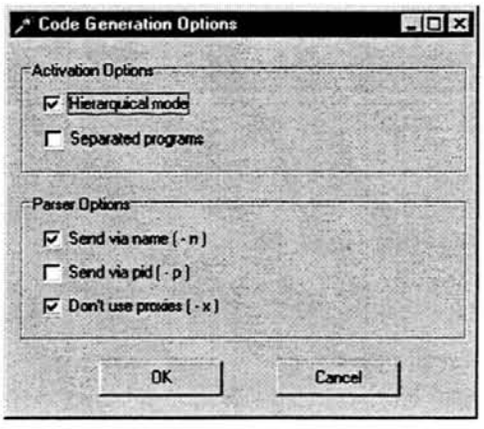


FIGURA 3.16 - Opções de configuração do projeto

4 Especificação de Comportamento

4.1 Introdução

Apesar das funcionalidades do ambiente SIMOO original, a comparação com outras ferramentas do gênero, descritas no capítulo 2 desta dissertação, comprova a sua carência em termos de facilidades para representação do comportamento das entidades do modelo. Esta deficiência tornou-se ainda mais evidente quando foi realizada a geração do código executável para os modelos de simulação desenvolvidos, como o modelo da Célula de Manufatura Robotizada apresentada no capítulo 2.

O principal problema encontrado foi a ausência de um formalismo para relacionar uma seqüência de atividades à chegada de um evento. No SIMOO, o comportamento do modelo de simulação, definido pelo usuário através de código C++, mistura código de tratamento de eventos (paradigma de simulação orientado a eventos) com código que representa as atividades das entidades. Este fator praticamente impossibilita o usuário de reutilizar o modelo de simulação, pois o código com o "comportamento desejado" da entidade confunde-se com o código de tratamento aos eventos. Na ferramenta ObjecTime este problema não é verificado, uma vez que a utilização de máquinas de estados restringe o código definido pelo usuário à computação de dados e a troca de mensagens entre os atores que compõem a aplicação.

Através da metodologia definida por Pereira em [PER94c], percebe-se que a utilização de diagramas de estados é uma solução bastante adequada para representar o "comportamento desejado" dos objetos, sendo aplicável tanto aos objetos simulados quanto aos objetos reais. Desta forma, para solucionar o problema apresentado foi incorporada ao ambiente SIMOO-RT uma ferramenta para edição de diagramas de transições de estados, denominada STDE - *State and Transitions Diagram Editor*, a qual é exibida na figura 4.1. As características desta ferramenta são detalhadas nas próximas seções.

4.2 Funcionalidades dos Diagramas de Transição de Estados

A ferramenta STDE não foi incorporada ao SIMOO-RT somente para fins de documentação, mas principalmente para formalizar a definição do comportamento das

entidades. Para tanto, alterou-se a filosofia de projeto utilizada no SIMOO original, onde o usuário, após definir as classes da aplicação, partia diretamente para a codificação dos métodos destas classes. Com a ferramenta STDE, o usuário inicia a especificação através da descrição dos possíveis estados de execução das classes/objetos e dos eventos que provocam as transições entre estes estados. Somente após esta formalização é que o código é especificado.

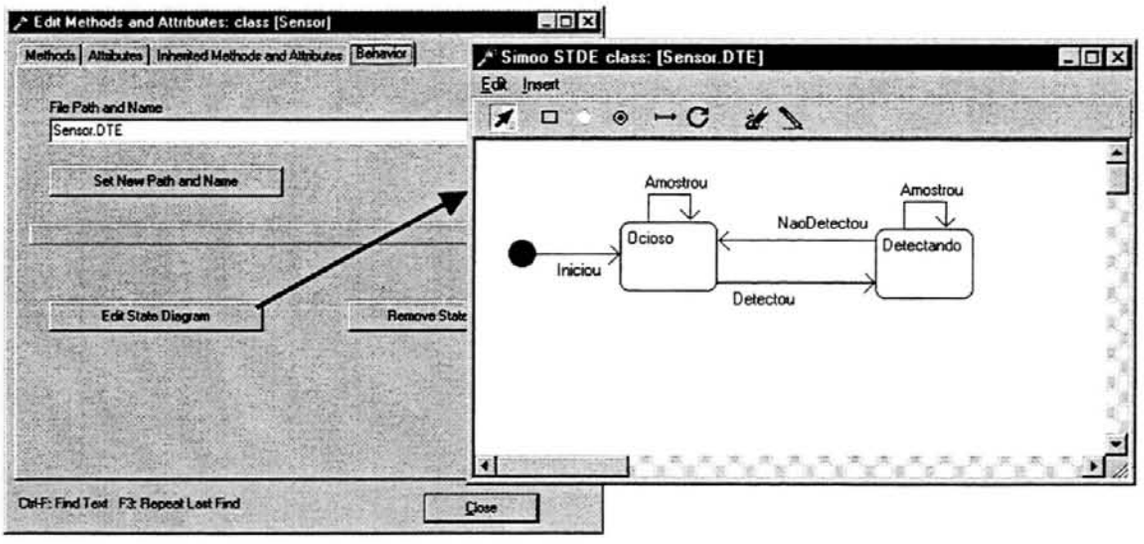


FIGURA 4.1 - Visão geral da ferramenta STDE

Uma transição é disparada (*triggered*) pela chegada de uma mensagem que, de acordo com a filosofia de desenvolvimento do ambiente, estará sempre associada a um único método da classe. Através da janela de especificação das transições (vide figura 4.2) o usuário pode inserir ou editar os métodos da classe, selecionando aquele que serve de *trigger* à transição que está sendo editada. Deve ficar claro que ao editar o código da transição, o usuário estará modificando o código do método correspondente à mensagem de disparo.

Em relação aos estados de execução da classe, a ferramenta permite a definição de três tipos de estados: estado inicial, estado final ou estado normal. Aos estados do tipo normal é permitido associar uma seqüência de atividades (vide figura 4.3), que são executadas assim que ocorrer a transição para o mesmo. Não é possível associar atividades aos estados inicial e final, uma vez que estes são associados, respectivamente, à transição 'Start' (associada ao método de criação da instância) e à transição 'End' (associada ao método de destruição da instância).

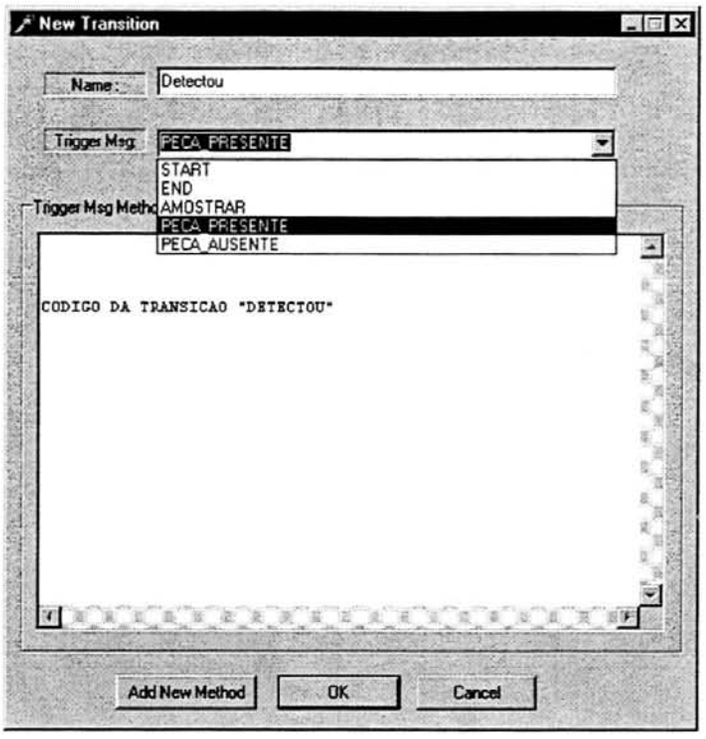


FIGURA 4.2 - Janela de especificação das transições

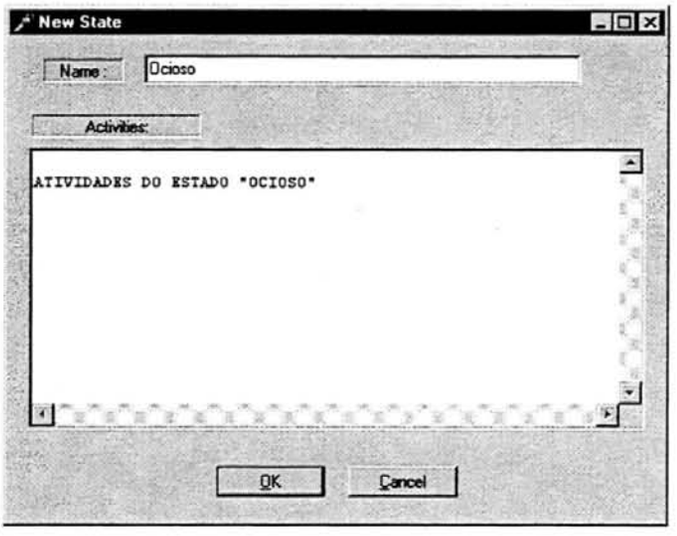


FIGURA 4.3 - Janela de especificação dos estados

4.3 Implementação do Mecanismo de Controle

Com a incorporação da ferramenta STDE no ambiente SIMOO-RT, foi necessário implementar nos códigos de simulação e execução um mecanismo de controle para os diagramas de transição de estado, o qual será encarregado de mapear as

mensagens aos métodos a serem executados quando as mesmas forem recebidas. É necessário ressaltar que o mecanismo implementado está longe de ser o ideal, mas oferece algumas funcionalidades adicionais ao código da aplicação, como restrições de atendimento aos eventos conforme o estado de execução do objeto.

O mecanismo de controle implementado segue o mesmo princípio tanto para o código de simulação quanto para o código de execução. Ele se baseia no fato que as classes ativas possuem mecanismos de decodificação das mensagens recebidas, mapeando-as para métodos de classe. Desta forma, adicionou-se aos mecanismos de decodificação um conjunto de instruções que restringe a chamada do método conforme o estado de execução do objeto. Assim, a execução do método só é permitida se ele estiver associado a alguma transição que parte do estado corrente. Para tornar mais claro este procedimento, na figura 4.4 é mostrado de maneira reduzida a função de decodificação 'TranslateMsg()' (presente no código de simulação), acrescida das instruções geradas pelo SIMOO-RT para o diagrama de transição de estados exibido da figura 4.1.

```
void Sensor::TranslateMsg(MSGEA m){
   ...
   if (m.GetIdMsg().Cmp("PECA_PRESENTE") == 0){
         switch(Estado){
            case OCIOSO:
               PecaPresente(m); // Executa ação da transicao
               Estado = DETECTANDO; // Altera estado corrente
               EstadoDETECTANDO( ); // Executa atividades do novo estado
               break;
         } //fim switch ESTADO
         return;
   }
   if (m.GetIdMsg().Cmp("PECA_AUSENTE") == 0)
   ...
} //fim da função
```

FIGURA 4.4 - Exemplo do mecanismo de controle dos métodos

Analisando o trecho de código destacado na figura 4.4, percebe-se que a chamada do método 'PecaPresente()' (associado à mensagem de mesmo nome) só é realizada caso o objeto esteja no estado Ocioso. Isto ocorre porque no diagrama de transição de estados da fig. 4.1 a transição Detectou (disparada pela mensagem PecaPresente - fig. 4.2) parte somente deste estado. Caso a transição seja disparada, logo após executar o código do método, o estado corrente do objeto passa a ser o estado de destino da transição (no exemplo em questão é o estado Detectando) e as atividades associadas ao novo estado são executadas. No caso do estado Detectando, as suas atividades são automaticamente mapeadas para um método denominado 'EstadoDetectando()'.

Conforme citado anteriormente, o mecanismo adotado no controle do código de execução segue o mesmo princípio. A diferença é que o método que controla a seleção das mensagens só aparece após o pré-processamento das classes ativas AO/C++ (as classes passivas não adotam este mecanismo). O método em questão denomina-se 'Select()' e é inserido pelo *parser* na classe C++ ativa resultante do pré-processamento, cujo nome recebe a extensão 'Proc' (vide seção 3.3.4).

Para acrescentar as devidas alterações no código da função 'Select()' foram vislumbradas duas maneiras distintas. A primeira consistia na geração de um arquivo intermediário com informações sobre os diagramas de transição de estados a ser lido pelo pré-processador, que através destas informações geraria o código adequado. Uma segunda opção seria utilizar a ferramenta de geração de código do SIMOO-RT para substituir o pré-processador na geração da classe em questão, já com o código adaptado aos diagramas de transição de estados. Diante das possibilidades levantadas escolheu-se a segunda opção, i.e. gerar o código da função 'Select()'. O código gerado para o método 'Select(char *msg)', correspondente ao diagrama da figura 4.1, é mostrado de maneira resumida na figura 4.5.

Deve ficar claro que o ambiente SIMOO-RT não obriga que todas as classes tenham um diagrama de transição de estados definido, ao contrário do que acontece em outras ferramentas do gênero, como o ObjecTime. Esta característica torna o ambiente mais flexível, pois permite deixar a cargo do usuário a verificação da necessidade da existência de um diagrama para restringir a chamada dos seus procedimentos. Na figura 4.6 é exibida a nova estrutura do processo de geração de código com a inclusão dos diagramas de transição de estados.

```
void SensorProc::Select(char *msg){
   switch(msg){
      case PECA_PRESENTE:
         switch(Estado){
            case OCIOSO:
               PecaPresente(m); // Executa ação da transição
               Estado = DETECTANDO; // Altera estado corrente
               EstadoDETECTANDO( ); // Executa atividades do novo estado
               break;
         } //fim switch ESTADO
         break;
      case PECA_AUSENTE:
      ...
   } //end switch msg
} //end function
```

FIGURA 4.5 - Estrutura do método 'Select'

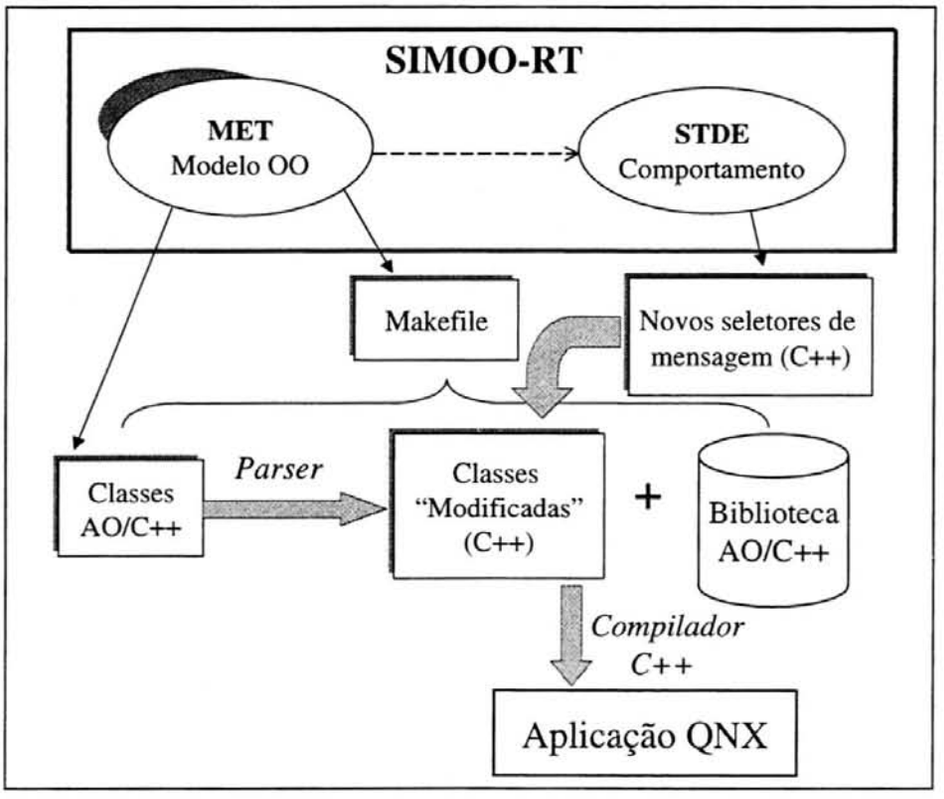


FIGURA 4.6 - Estrutura dos arquivos gerados com o uso do STDE

5 Estudos de Caso

5.1 Introdução

Esta capítulo visa avaliar a potencialidade do ambiente SIMOO-RT no desenvolvimento de sistemas de tempo real orientados a objetos, através da sua utilização em alguns estudos de caso. O primeiro deles é descrito na seção 5.2 e consiste na modelagem, simulação e implementação de um sistema de controle para uma bancada didática, que permite monitorar e atuar sobre o nível de líquidos em um mini-tanque. Logo após, na seção 5.3, apresenta-se em detalhes o sistema de controle da célula de manufatura robotizada citada no capítulo 2. Por fim, na seção 5.4 é feita a descrição do estudo de caso que trata do desenvolvimento de um sistema distribuído para controle de elevadores.

5.2 Estudo de Caso 1: Tanque de Armazenamento

Este estudo de caso trata da utilização do ambiente SIMOO-RT no desenvolvimento de um sistema para controle de nível em um mini-tanque de armazenamento. Por ser um sistema para fins didáticos, esta bancada de controle faz uso de um hardware bastante simples, que pode ser facilmente adaptado para uma tecnologia mais complexa, como as utilizadas nos ambientes industriais reais. Também por motivos didáticos, para demonstrar a possibilidade de distribuição de objetos, utiliza-se de dois computadores ligados em rede para realizar o controle do sistema.

A monitoração do tanque é feita através de um sensor de nível, constituído por lâminas metálicas paralelas eletricamente isoladas, cuja capacitância varia de acordo com o nível de água entre elas. A variação da capacitância se reflete na variação da freqüência de um oscilador, que altera a tensão nas entradas de um conversor analógico/digital conectado à porta paralela de um PC. Para atuar sobre o nível do líquido no tanque o sistema faz uso de uma bomba, também conectada a um PC via porta paralela. A figura 5.1 exibe a configuração do sistema em questão.

Segundo a metodologia adotada pelo ambiente SIMOO-TR, a primeira etapa do processo de desenvolvimento consiste na criação do modelo de simulação do sistema em questão. Desta forma, através da ferramenta MET, foram definidos os diagramas de

classes e de instancias do modelo, os quais encontram-se subdivididos em apenas dois níveis de hierarquia. O primeiro nível contém a classe 'SistTanque', que não tem outra utilidade senão agregar o restante do modelo. Expandindo o primeiro nível são encontradas as entidades que constituem a aplicação sendo modelada, i.e. as classes 'Tanque', 'SenNivel' (sensor de nível) e 'Bomba', como mostra a figura 5.2. Também pode ser verificada neste diagrama a presença da classe 'Liquido', que simula o conteúdo do tanque, e dos elementos de interface para os componentes da aplicação, responsáveis pela animação da simulação. O diagrama de instâncias correspondente não necessita ser mostrado, devido à relação 1 para 1 entre as classes e as suas instâncias.

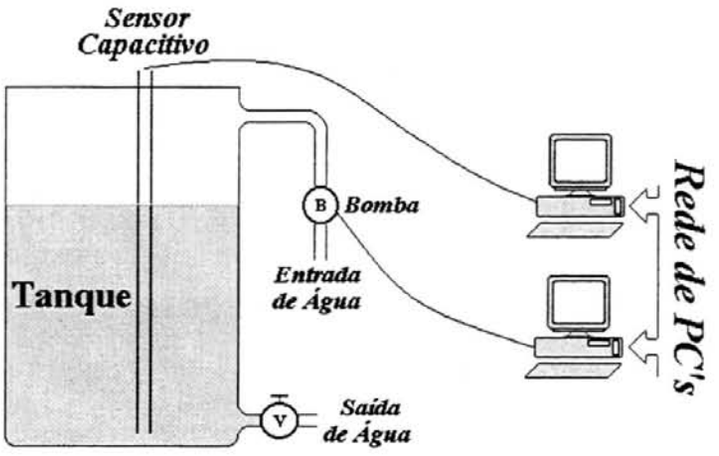


FIGURA 5.1 - Visão geral da bancada de controle do tanque

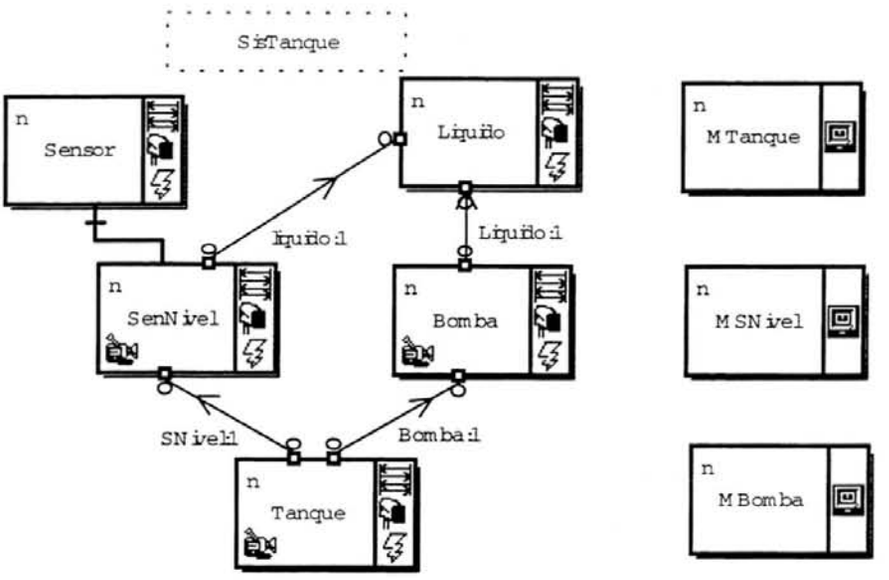


FIGURA 5.2 - Diagrama de classes para o sistema de controle do tanque

Após a definição do diagrama de classes, o próximo passo para a implementação do modelo de simulação é a definição do comportamento destas entidades. O ambiente SIMOO-RT sugere o uso de diagramas de transição de estados para representar o comportamento "desejado" das entidades do modelo. Desta forma foi definido o diagrama da classe Tanque, cuja incumbência é manter o nível do líquido contido no tanque dentro de uma faixa previamente especificada. Para tanto, foram definidos dois estados de execução para esta classe: Ocioso e Enchendo.

O estado Ocioso representa o tanque perdendo líquido, e é disparado (*triggered*) pelas mensagens START (*default*), gerada no inicio da execução do sistema, ou DESLIGAR_BOMBA, gerada quando o líquido atinge o nível máximo permitido. Por sua vez, o estado Enchendo representa o tanque recebendo líquido, sendo disparado pela mensagem LIGAR_BOMBA. As mensagens especificadas representam um método da classe, cujo código é especificado na janela de edição da classe. O diagrama de transição de estados da classe Tanque, juntamente com a definição das mensagens de disparo das transições, é exibido na figura 5.3.

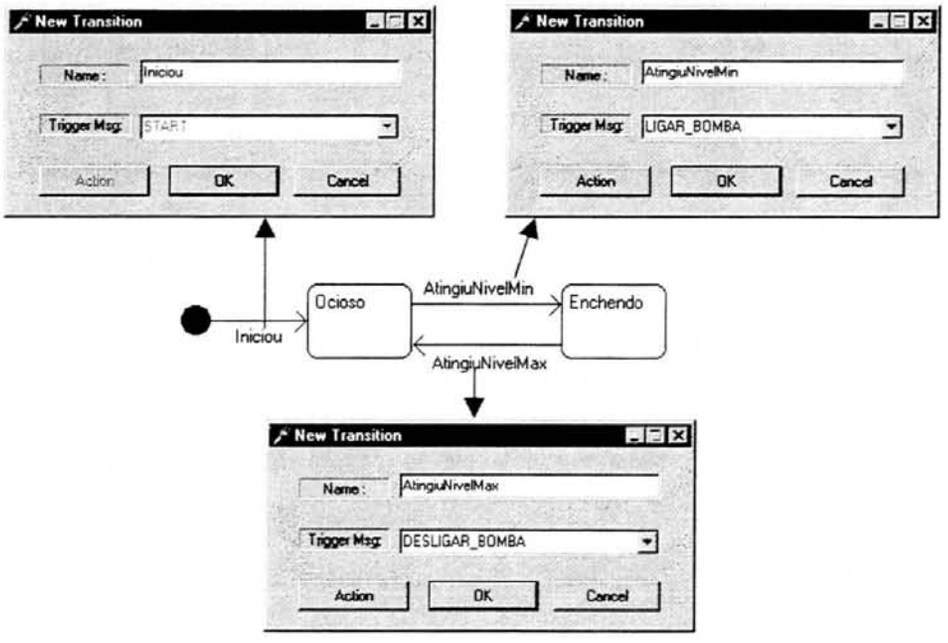


FIGURA 5.3 - Diagrama de transição de estados para a classe Tanque

As outras duas classes presentes no diagrama da figura 5.2 não tiveram o seu comportamento representado por um diagrama de transição de estados devido a sua trivialidade. A classe 'SenNivel' simplesmente monitora o nível do líquido e informa à

classe Tanque quando perguntada. Já a classe Bomba simplesmente liga ou desliga a bomba, conforme solicitação da classe 'Tanque'.

Além de servir para descrever o comportamento desejado das entidades, o diagrama de transição de estados também poderia ser utilizado, caso o ambiente tivesse suporte para tanto, para representar os estados de visualização dos elementos de interface associados aos elementos autônomos. Para exemplificar este conceito, na figura 5.4 é mostrado um possível diagrama de transição de estados para o monitor associado à classe 'Tanque', juntamente com as diferentes seqüências de visualização associadas ao mesmo. No caso da simulação desenvolvida, este procedimento foi descrito manualmente através de código C++. O cenário final da simulação é apresentado na figura 5.5.

Apesar do modelo de simulação não fazer uso dos requisitos temporais acrescentados ao ambiente, é desejável que nesta fase já sejam definidos todos os processos com comportamento periódico e também os que possuem *deadlines* associados. Na aplicação em questão, verificou-se que algumas operações são críticas para um bom andamento do sistema, como as operações de ligar e desligar a bomba, presentes na classe Bomba. Também foram verificados alguns processos cíclicos, como o processo de amostragem do sensor (classe Sensor) e o processo de controle do sistema (classe Tanque). A figura 5.6 exibe a definição do método 'Liga()', responsável pelo acionamento da bomba, e do método 'Amostrar()', utilizada para fazer a leitura do sensor de nível.

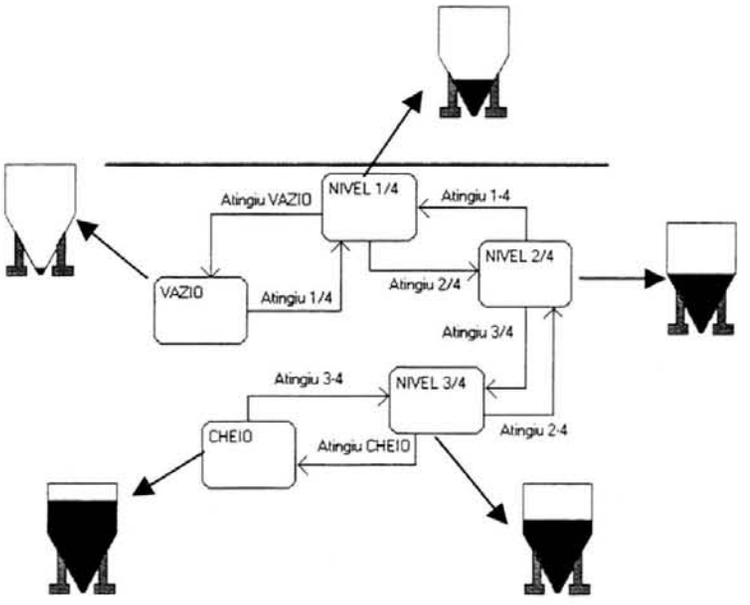


FIGURA 5.4 - Estados do elemento de visualização da classe Tanque

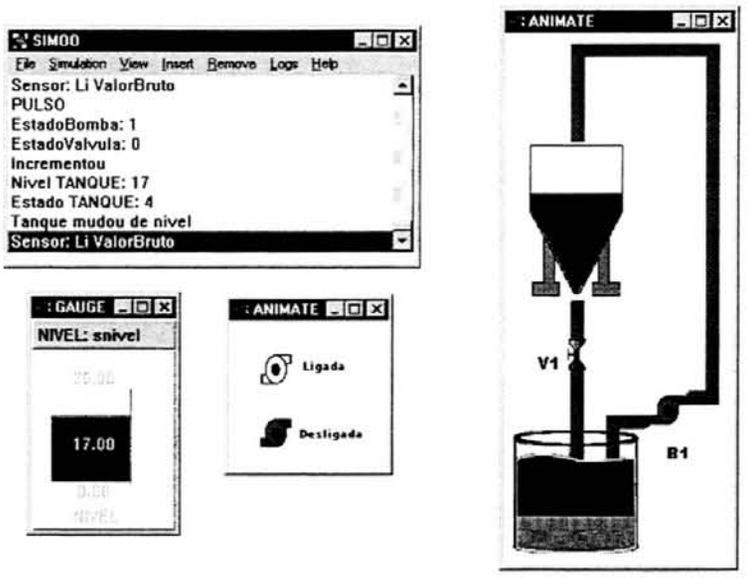


FIGURA 5.5 - Simulação do sistema do tanque em andamento

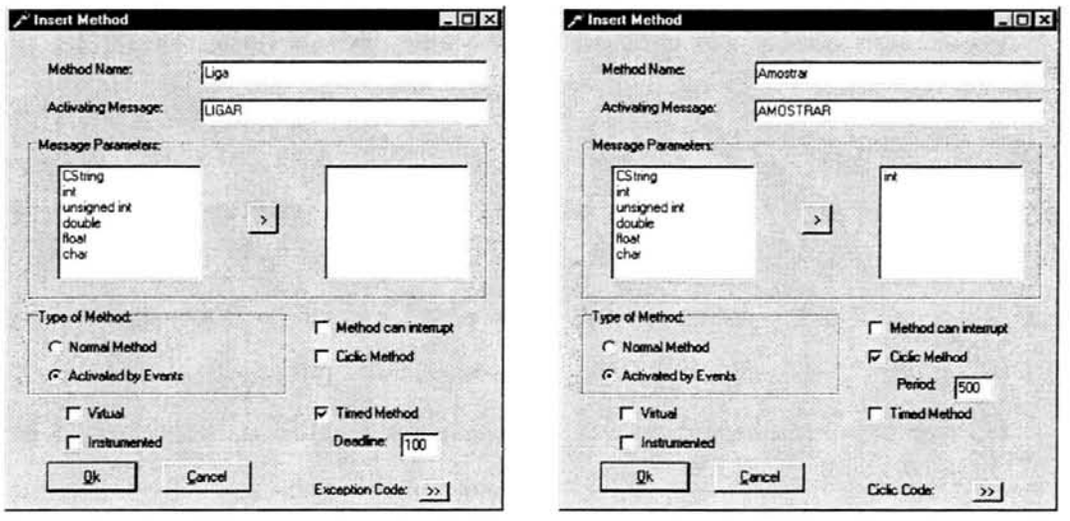


FIGURA 5.6 - Exemplo da especificação de métodos com restrições temporais

Terminada a etapa de simulação da aplicação, realizou-se a transição do modelo de simulação para o "modelo de execução" correspondente, redefinindo os diagramas de classes e de instâncias de acordo com as necessidades do ambiente de execução da aplicação. Esta redefinição constou da retirada da classe 'Liquido', que servia somente para simular o líquido no tanque, e da adição de três novas classes: 'DrvSensor', 'DrvBomba' e 'PortaPar'. As duas primeiras são classes de interface entre o objeto lógico e o objeto real que está sendo controlado/monitorado. A terceira classe representa a porta paralela do computador, tendo em vista que este é o dispositivo de E/S utilizado para fazer a comunicação entre a aplicação computacional e as entidades reais.

Outra modificação foi a substituição dos elementos de interface do modelo de simulação por elementos autônomos. Esta modificação se deve ao fato do código gerado não suportar o conceito de elementos de interface, dado que a comunicação entre os objetos da aplicação e os objetos de visualização devem ser definidas explicitamente pelo usuário. A figura 5.7 ilustra o novo diagrama obtido. Quanto ao diagrama de instâncias correspondente, para permitir que o controle do sensor de nível e da bomba sejam feitos de maneira distribuída são necessárias duas instâncias da classe 'PortaPar', uma vez que são executadas em computadores distintos.

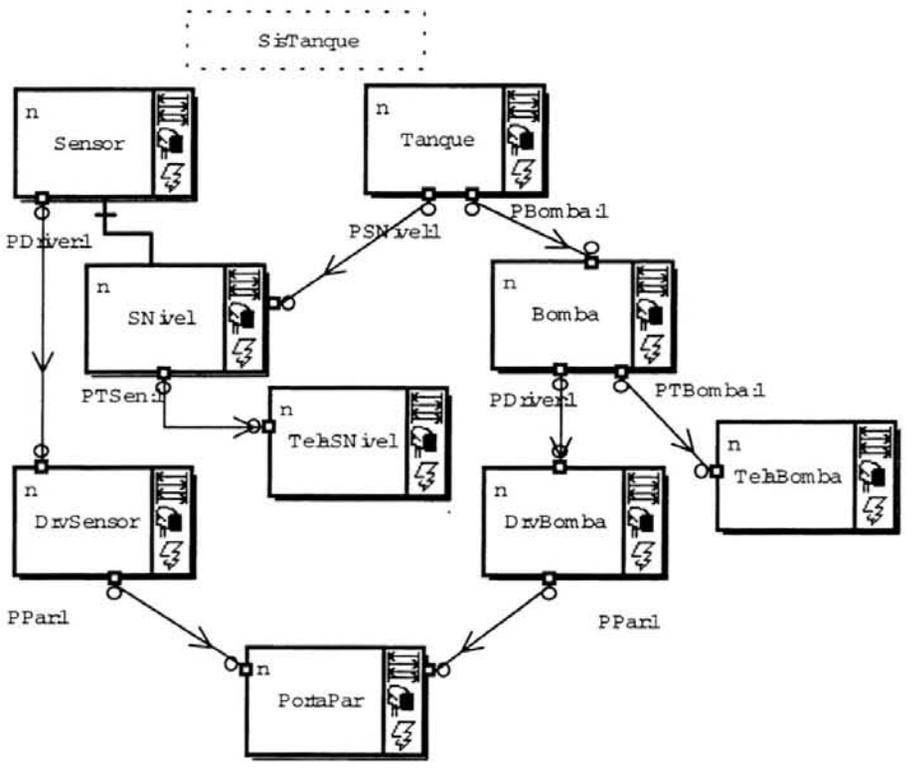


FIGURA 5.7 - Diagrama de classes adaptado para o ambiente de execução

Uma vez redefinido o diagrama de classes, para que o mesmo fosse portado para o QNX e não ocorressem erros na sua compilação, foram retiradas todas as instruções que faziam chamadas à biblioteca de simulação e deixadas apenas as instruções C++ de controle, comuns a ambos os modelos. Basicamente foram retiradas as instruções responsáveis pela troca de mensagens entre os objetos, que tiveram que ser substituídas pelo código AO/C++ correspondente (bastante simplificado). A figura 5.8 exemplifica as trocas realizadas, mostrando o código do método *DesligarBomba()* (classe Tanque) antes e depois da sua adequação para o ambiente QNX.

- **Código da simulação:**

```
void Tanque::
DesligarBomba(MSGEA m)
{
   Parametros par;

   m.Set(Id(),bomba,"DESLIGA",
         NORMAL_PR,par);
   Send(m);
}
```

- **Código da aplicação executável:**

```
void Tanque::DesligarBomba()
{
   bomba->Desliga();
}
```

FIGURA 5.8 - Método *DesligarBomba()* antes e depois da geração da aplicação

Comparando os códigos de simulação e execução, é possível verificar outras funcionalidades do código AO/C++ que simplificam o processo de implementação, dentre as quais destaca-se as operações cíclicas. No AO/C++ uma função definida como cíclica não necessita ter a sua próxima ativação reprogramada, uma vez que o sistema provém esta funcionalidade. Por outro lado, no ambiente de simulação, para que uma função seja executada periodicamente, o usuário deve programá-la para tanto. As diferenças citadas podem ser verificadas na figura 5.9, na qual são exibidos os códigos de simulação e execução do método cíclico *Amostrar()*, pertencente a classe Sensor.

Continuando o processo de comparação, analisando as definições das classes é possível verificar as diferenças entre os códigos de simulação e de execução. Enquanto no código de simulação é necessário explicitar a herança da classe básica (proveniente da biblioteca de simulação), no AO/C++ o uso da palavra chave 'active' (adicionada automaticamente pelo código gerado) é suficiente para que o pré-processador adicione automaticamente o código da herança. Outra diferença é o mecanismo de referência para as relações de conhecimento da classe. No código de simulação, as referências são instâncias de um tipo especial de dado definido pela biblioteca de simulação (e.g. CString Liquido). Já no código de execução as referências são expressas através de ponteiros (e.g. DrvBomba *PDriver). Por último cabe salientar a diferença no mecanismo de passagem de parâmetros das duas implementações. O AO/C++ usa praticamente o mesmo mecanismo do C++, exceto na passagem de vetores de caracter, onde o tamanho máximo deve ser previamente especificado (e.g. Start(char*25)). Em contrapartida, no código de simulação os argumentos são passados para os métodos através de um único parâmetro, que contém todos os dados da mensagem (e.g.

Start(MSGEA m)). A figura 5.10 apresenta o código de definição das classes nos ambientes de simulação e de execução.

- **Código da simulação:**

```
void Sensor::Amostrar(MSGEA m)
{
   Parametros par;
   int periodo;

   //Programar o novo evento CICLICO
    periodo = m.GetDados().GetParAsInt(0);
   par.Set(periodo);
   m.Set(Id(),Id(),"AMOSTRAR",
          NORMAL_PR,par);
   Send(periodo,m);

   //Código do ciclo
   m.Set(Id(),SimulNivel, "INFORMAR",
          NORMAL_PR,par);
   SendReceiveNow(&m);
  ValorBruto=m.GetDados().GetParAsInt(0);
}
```

- **Código de execução:**

```
void Sensor::Amostrar(cycle_t)
{
   //Codigo de inicializacao
begin_cycle //Codigo do ciclo
   ValorBruto = PDriver->Ler();
end_cycle
}
```

FIGURA 5.9 - Código do método cíclico *Amostrar()*

- **Código de simulação:**

```
class Bomba : public EVMSG
{
   protected:
   int EstadoBomba;
   CString Liquido;

   public:
   Bomba(CString *pid, char *pcl,
          CString ppai, CATEGORY cat);
   void Start(MSGEA m);
   void EndEA(MSGEA m);
   void Liga(MSGEA m);
   void Desliga(MSGEA m);
   void TranslateMsg(MSGEA m);
   void RecebeIdPonto(MSGEA m);
   void Simoo_Get_Attrs(MSGEA m);
   void Simoo_Get_Msgs(MSGEA m); };
```

- **Código de execução:**

```
active class Bomba
{
   protected:
   int EstadoBomba;
   DrvBomba *PDriver;

   public:
   Bomba();
   ~Bomba();
   void Conecta_PDriver(char*25);
   void Start(char*25);
   void End();
   void Liga(dead_l);
   void Desliga(dead_l);
};
```

FIGURA 5.10 - Código de definição das classes

- **Código de simulação:**

```
void Tanque::TranslateMsg(MSGEA m)
{
 switch (m.GetIdMsg()){
  case START:{
    switch(EstadoTanque){
      case INITIAL{
        Start(m); //Acao da transicao
        EstadoTanque = OCIOSO; //Altera o estado
        TanqueOcioso(); //Operacoes do novo estado
      } break;
    } //fim switch(ESTADO)
  } break;
  ... //outras mensagens
 } //fim switch(MSG)
 EVMSG::TranslateMsg(m);
}
```

- **Código de execução:**

```
int TanqueProc::Select(char *recv_msg)
{
 int sel_flag = 0;
 if(!strcmp(recv_msg, "Tanque")) switch (recv_msg[6]) {
    case TANQUE_START
    {
        Tanque_Start_msg* P_Tanque_Start_msg;
        P_Tanque_Start_msg = (Tanque_Start_msg*) (recv_msg);
        qnx_proxy_detach(receive_pid);
        switch(EstadoTanque){
            case INITIAL{
                Start(P_Tanque_Start_msg->string1); //Acao da transicao
                EstadoTanque = OCIOSO; //Altera o estado
                TanqueOcioso(); //Operacoes do novo estado
                sel_flag = 1;
            } break;
        } //fim switch(ESTADO)
    } break;
    ... //outras mensagens
  } //fim switch(MSG)
 return(sel_flag);
}
```

FIGURA 5.11 - Código gerado para o diagrama de transição de estados

Outro aspecto a ser comparado entre os códigos de simulação e de execução é o mecanismo de controle referente aos diagramas de transição de estados. Conforme mencionado no capítulo 4, o código de simulação faz uso do método *TranslateMsg()* para selecionar o método correspondente a cada mensagem. Já o código de execução utiliza o método *Select()*, que é gerado pelo pré-processador AO/C++. Todavia, analisando os códigos de simulação e execução gerados para a classe Tanque, verifica-se que ambos os códigos são bastante semelhantes. A figura 5.11 exibe os códigos gerados, realçando as instruções que definem a ativação do método *Start()* somente a partir do estado inicial.

Para finalizar este estudo de caso, são reportadas algumas estatísticas referentes aos modelos desenvolvidos e aos códigos gerados para a simulação e para a execução. A tabela 5.1 resume os dados levantados.

TABELA 5.1 - Estatísticas do modelo de controle do tanque

	Código de Simulação	**Código de Execução**	
Itens levantados	**C++/Windows**	**AO/C++**	**C++/QNX**
Linhas de código	2690	729	2308
Número de Classes	9	10	20
Número de Instâncias	9	11	22
Níveis de herança	(max=03/med=02)	(max=02/med=01)	(max=03/med=02)
Níveis de agregação	(max=02/med=02)	(max=02/med=02)	(max=02/med=02)

5.3 Estudo de Caso 2: Célula de Manufatura Robotizada

No segundo estudo de caso, foi repetido o processo de desenvolvimento citado na seção anterior, onde iniciou-se pelo desenvolvimento do modelo de simulação e terminou-se na definição do modelo executável. Desta vez, o alvo foi a célula de manufatura robotizada (CMR) apresentada como estudo de caso no capítulo 2. O funcionamento do sistema pode ser resumido nos seguintes passos: quando a peça passa pela câmera de vídeo, sua imagem é processada para a obtenção de alguns parâmetros tais como a sua largura e a sua posição na esteira. Assim que o sensor detecta a presença da peça a esteira é parada, a mesa de armazenagem é posicionada e o braço é acionado para capturar e armazenar a peça.

Como o alvo desta dissertação é o desenvolvimento de sistemas de tempo real, convém levantar os aspectos temporais deste estudo de caso. Apesar da dinâmica da CMR ser considerada lenta, i.e. os estímulos não requerem altas velocidade de resposta, é possível perceber a existência de alguns requisitos que necessitam de um tratamento adequado, dentre os quais destacam-se:

1) Processos periódicos: câmera amostra imagem; sensor verifica presença da peça;

2) Relações de precedência: braço pode capturar a peça somente após a chegada da mesma na zona de carga;

3) Tempo máximo de execução: caso a mesa não se posicionar no tempo máximo especificado, o processo de armazenagem da peça deve ser abortado.

O modelo de simulação para este estudo de caso foi apresentado anteriormente na figura 2.4. Este modelo sofreu algumas alterações no que diz respeito à definição do seu comportamento, onde cada uma das entidades teve um diagrama de transição de estados associado. Em relação ao estudo de caso apresentado previamente na seção 5.2, os elementos deste novo modelo se diferenciam em termos de comportamento. Isto pode ser percebido uma vez que não existe uma entidade de gerenciamento, pois cada um dos componentes da célula tem um comportamento autônomo bem definido.

Para tornar mais clara esta diferenciação, considera-se o exemplo da classe 'Sensor' que, ao invés de ser verificado pela classe 'Braco' através de um mecanismo de verificação repetida (*polling*), se encarrega de informar ao 'Braco' o momento em que foi constatada a chegada de uma nova peça. Na figura 5.12 é mostrado o diagrama de transição de estados da classe Sensor.

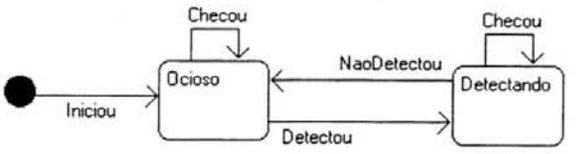


FIGURA 5.12 - Diagrama de transição estados da classe Sensor

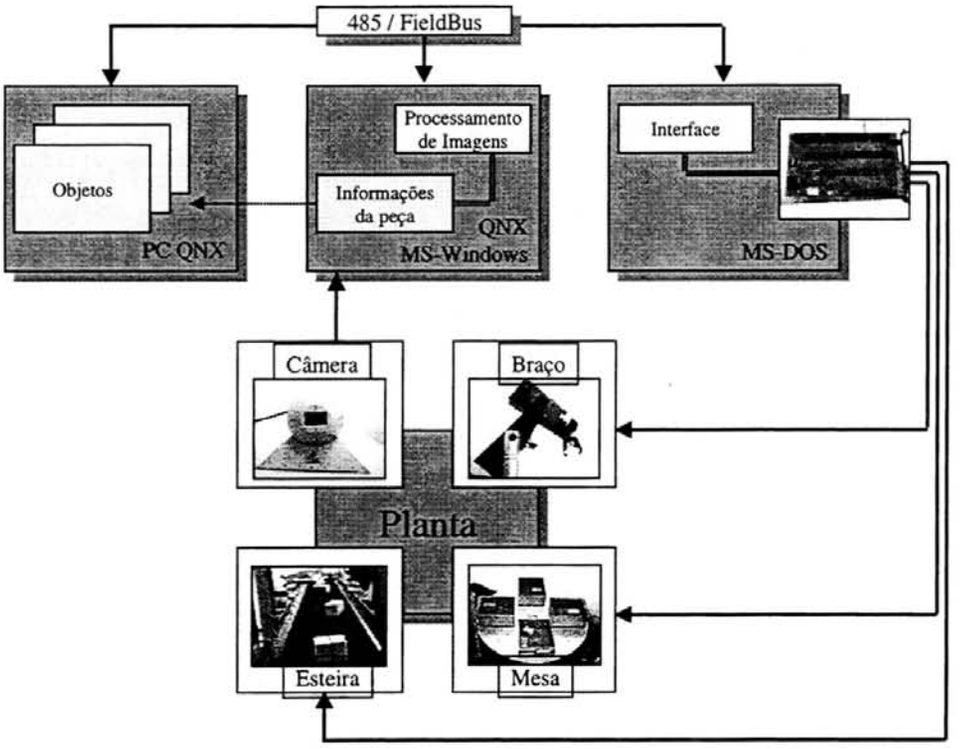


FIGURA 5.13 - Configuração do sistema de controle da CMR

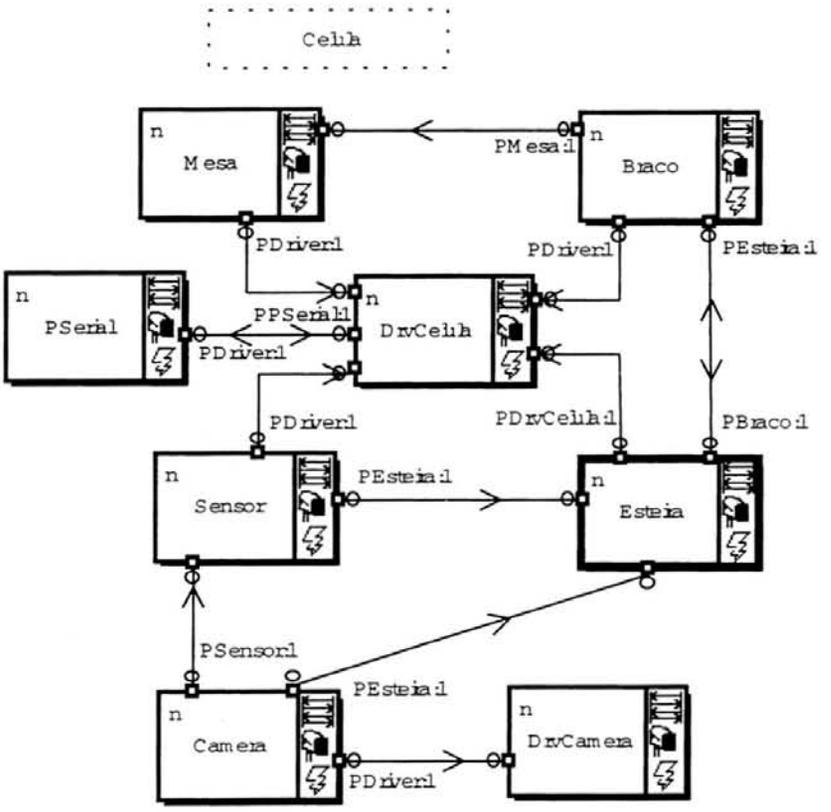


FIGURA 5.14 - Refinamento do diagrama de classes da CMR

A configuração final do sistema de controle da célula de manufatura robotizada é mostrada no diagrama da figura 5.13. Como pode ser verificado, a comunicação dos objetos lógicos com as suas 'contra-partes', ou seja, com os componentes reais da planta industrial ocorre através de um único objeto de interface (associado a uma placa de interface com o protótipo). Devido a isto, a exemplo do estudo de caso anterior, foi necessário refinar o modelo de simulação definido, a fim de adequá-lo à configuração do sistema que está sendo controlado, conforme mostrado na figura 5.14. As estatísticas referentes a este estudo de caso são exibidas na tabela 5.2.

TABELA 5.2 - Estatísticas do modelo da célula de manufatura robotizada

	Código de Simulação	**Código de Execução**	
Itens levantados	**C++/Windows**	**AO/C++**	**C++/QNX**
Linhas de código	3327	858	2561
Número de Classes	12	10	20
Número de Instâncias	12	10	20
Níveis de herança	(max=02/med=02)	(max=01/med=01)	(max=02/med=02)
Níveis de agregação	(max=03/med=02)	(max=03/med=02)	(max=03/med=02)

5.4 Estudo de Caso 3: Controle de Elevadores

Nesta seção é descrito o terceiro e último estudo de caso realizado, onde foi modelado um sistema de controle multi-elevadores baseado no trabalho de Brudna (vide [BRU98]). Convém ressaltar que o objetivo deste estudo de caso é consolidar as propriedades do ambiente desenvolvido e não testar algum algoritmo de controle. Para tanto, após o projeto do modelo de simulação, foi gerada a aplicação no sistema operacional QNX que, através da sua iteração com elementos gráficos, simula o controle do sistema real.

O modelo desenvolvido partiu de uma classe denominada 'MultiElev', encarregada de escolher o elevador que irá atender cada requisição externa recebida, de acordo com a configuração em que o sistema se encontra. Expandindo esta classe encontram-se as classes 'PainelExt' e 'Elevador' que representam, respectivamente, o painel com os botões externos (presentes nos andares) e o Elevador que está sendo controlado. Esta organização permite que se tenha um sistema flexível em termos de

número de elevadores e número de andares, uma vez que as quantidades destes elementos são determinadas no diagrama de instâncias. Para este estudo de caso, foi modelado um edifício de cinco andares com dois elevadores, conforme retratado no diagrama de instâncias da figura 5.15. Também na figura 5.15 pode ser observada a presença das classes 'MElev' e 'MPainelExt', encarregadas de promover a animação destes elementos durante a simulação.

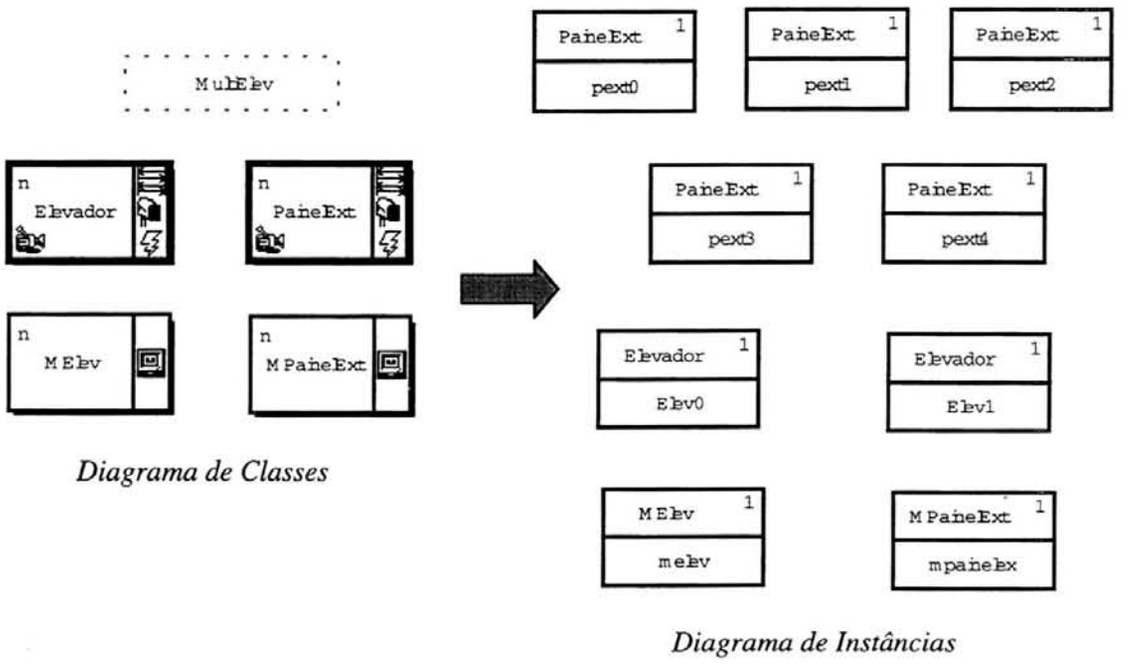


FIGURA 5.15 - Configuração do sistema para controle de elevadores

As bordas largas das classes presentes na figura 5.15 indicam que elas contêm elementos agregados. No caso da classe 'PainelExt' existe somente uma classe agregada, a classe passiva 'Botão', que foi modelada desta forma por não ter de atender a solicitações externas, apenas enviar mensagens avisando que foi pressionada. Para a classe 'Botão' foram definidas duas instâncias, 'subir' e 'descer', conforme exibido na figura 5.16.

A classe 'Elevador', por sua vez, agrega três novas entidades, constituídas das classes 'Porta', 'Motor' e 'PainelInt'. As duas primeiras são encarregadas de controlar a temporização associada às operações de abertura e fechamento das portas e à operação de movimentação do elevador, uma vez que o sistema sendo controlado não existe fisicamente. Já a classe 'PainelInt' tem praticamente a mesma função da classe 'PainelExt' mencionada anteriormente. Sua diferença é que ela representa as chamadas

dos usuários internos do elevador e o número de instâncias agregadas da classe botão é definido pelo número de andares do sistema. A figura 5.17 retrata os diagramas resultantes da expansão da classe 'Elevador'.

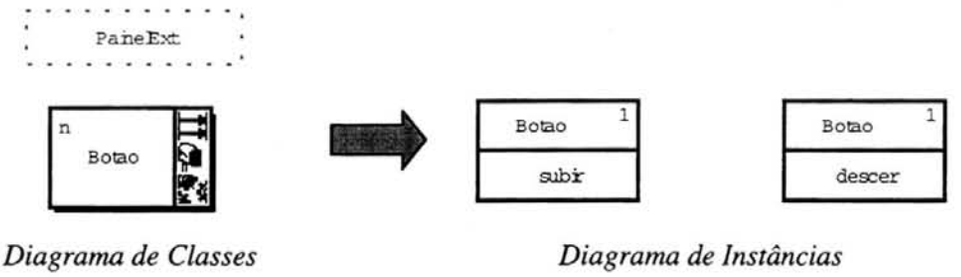


FIGURA 5.16 - Diagramas da classe PainelExt

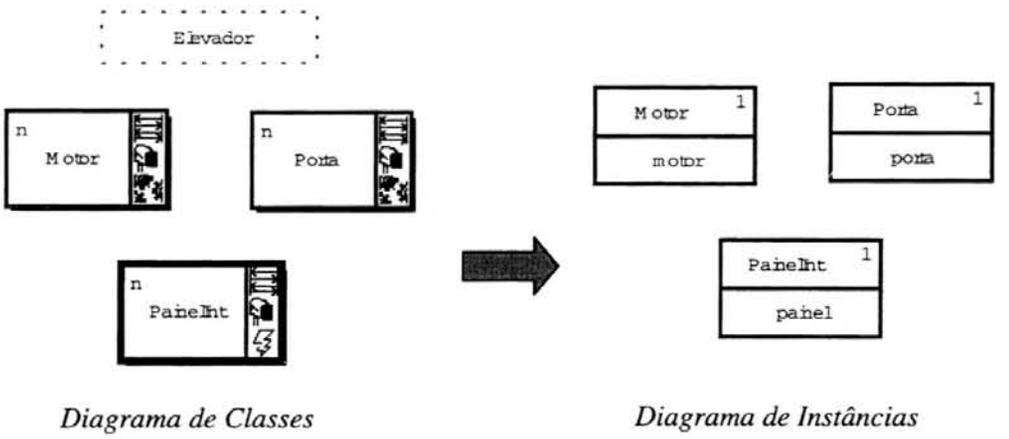


FIGURA 5.17 - Diagramas da classe Elevador

Como o SIMOO-RT suporta a utilização de máquinas de estados para descrever o comportamento das entidades, definiu-se inicialmente um diagrama de transição de estados para a classe 'Elevador', como mostra a figura 5.18. Este diagrama descreve os quatro possíveis estados de execução em que o elevador pode se encontrar, explicitando as mensagens que comandam as trocas de estado. Porém, a utilização deste diagrama teve de ser revista uma vez que as mensagens de solicitação de elevador, vindas dos painéis externos, estavam sendo perdidas quando o elevador não estivesse no estado 'Parado'. Desta forma, optou-se por implementar o algoritmo de controle do elevador sem a utilização de um diagrama de transição de estados. Como resultado da simulação realizada no ambiente SIMOO-RT, tem-se a animação da figura 5.19.

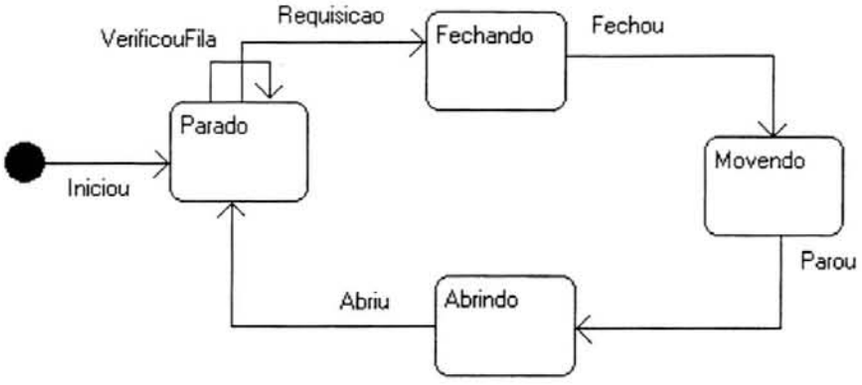


FIGURA 5.18 - Diagrama de transição de estados para a classe Elevador

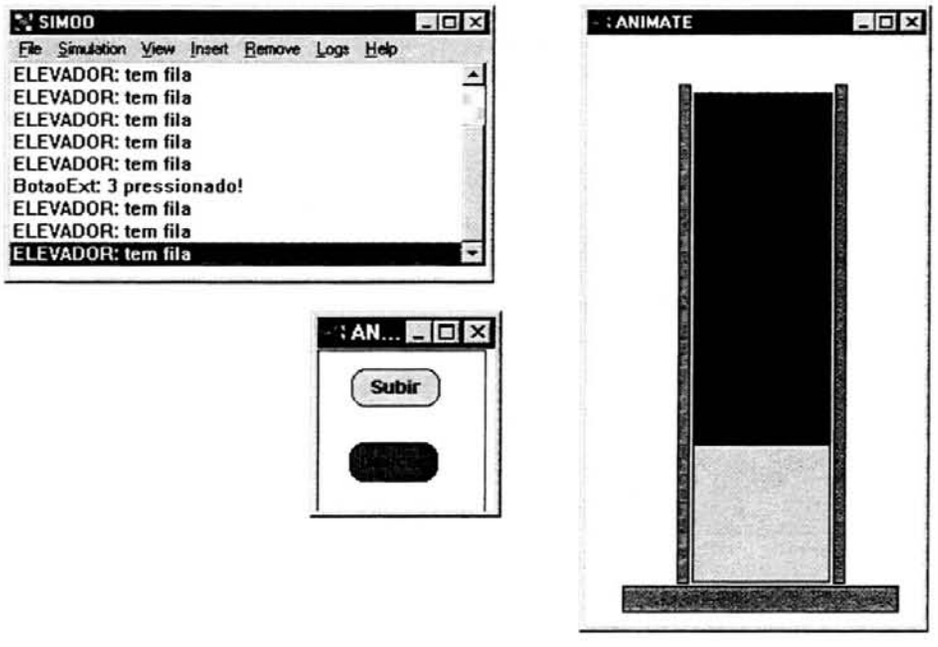


FIGURA 5.19 - Animação do sistema de elevadores no ambiente SIMOO-RT

Uma vez terminado o modelo de simulação para a aplicação em questão, da mesma forma como foi feito nos estudos de casos citados anteriormente, faz-se necessário refinar o modelo de simulação de acordo com as necessidades do ambiente de execução. Neste estudo de caso, o processo de refinamento do modelo é facilitado devido ao fato da aplicação sendo modelada não existir fisicamente. Portanto a adição de 'Drivers' de comunicação torna-se desnecessária, uma vez o modelo de execução irá na realidade simular o controle da aplicação, através da sua interação com elementos gráficos.

Os elementos gráficos de interação com a aplicação gerada foram programados manualmente dentro das classes 'Elevador', 'PainelExt' e 'PainelInt'. Os elementos de visualização associados às classes modificadas foram suprimidos da aplicação gerada, dado que as propriedades de recebimento automático de mensagens por parte destes elementos são uma propriedade restrita do ambiente de simulação. A tabela 5.3 resume as estatísticas referentes a este estudo de caso.

TABELA 5.3 - Estatísticas do sistema para controle de elevadores

	Código de Simulação	**Código de Execução**	
Itens levantados	**C++/Windows**	**AO/C++**	**C++/QNX**
Linhas de código	2123	713	2413
Número de Classes	8	8	16
Número de Instâncias	32	32	48
Níveis de herança	(max=03/med=02)	(max=02/med=01)	(max=03/med=02)
Níveis de agregação	(max=04/med=03)	(max=04/med=03)	(max=04/med=03)

5.5 Análise dos Resultados

A partir dos estudos de caso realizados constata-se que o ambiente SIMOO-RT facilita de maneira considerável o processo de desenvolvimento dos sistemas de tempo real. As facilidades do ambiente permitem a descrição tanto do modelo estático da aplicação sendo analisada (diagramas de classes e instâncias) quanto do seu modelo dinâmico (diagrama de transição de estados), os quais resultam em modelos OO "simuláveis". Estes modelos permitem a validação funcional do sistema anteriormente à implementação, permitindo a detecção de eventuais erros de projeto. Mesmo considerando-se que na versão atual SIMOO-RT não permite a validação do comportamento temporal, o uso da ferramenta permitiu a detecção de inconsistências durante a realização dos estudos de caso.

Uma vez definido o modelo de simulação, o processo de geração da aplicação é mais um refinamento deste modelo. Nele são adicionadas entidades que representam os módulos de interface com o processo físico, ou simplesmente *drivers*, que permitem a comunicação dos objetos computacionais lógicos com os objetos físicos que constituem o sistema a ser controlado. Refinado o modelo, o ambiente oferece ao usuário a

facilidade da geração automática de código na linguagem AO/C++, que incorpora propriedades para tratar as restrições temporais especificadas no modelo, além de facilidades que possibilitam a distribuição da aplicação. Conforme discutido anteriormente, o gerador automático de código também gera automaticamente os arquivos de projeto, arquivos *makefile*, os quais são essenciais para a geração da aplicação completa.

Cabe salientar que além da redefinição do modelo, o usuário necessita alterar os mecanismos de troca de mensagens especificados durante a criação do modelo de simulação, conforme mencionado nos estudos de caso. Este fato é indesejável, pois incompatibiliza os modelos de simulação e execução.

Por outro lado, verificando as estatísticas apresentadas nos estudos de caso, pode-se constatar a forte relação da linguagem AO/C++ com os modelos SIMOO-RT. O número de linhas de código dos programas AO/C++ gerados é bastante reduzido em relação aos seus programas C++ correspondentes, provenientes do seu pré-processamento ou mesmo da simulação.

6 Conclusões e Trabalhos Futuros

6.1 Considerações Gerais

Os principais pontos abordados por esta dissertação envolvem a modelagem e a implementação de sistemas de tempo real, fazendo uso do paradigma de orientação a objetos. A principal contribuição deste trabalho é a extensão do ambiente SIMOO, denominada SIMOO-RT, ao qual foram adicionados as seguintes funcionalidades:

- Possibilidade de descrição de restrições temporais no modelo OO, usando as novas opções e novos 'menus' acrescentados à ferramenta SIMOO, os quais encontram-se descritos no capítulo 3;
- Capacidade de especificar-se o comportamento dos objetos através de diagramas de transição de estados usando a ferramenta STDE (*State-Transiction Diagram Editor*), cujas propriedades foram apresentadas no capítulo 4;
- Geração automática de código para a linguagem AO/C++. Esta linguagem foi selecionada por incorporar primitivas de "alto nível" para especificar restrições temporais e por facilitar a distribuição da aplicação..

O trabalho desenvolvido foi motivado pelos resultados obtidos nas comparações efetuadas durante a análise do estado-da-arte tanto de ferramentas de modelagem OO quanto de linguagens de programação para aplicações tempo-real, onde observou-se a inexistência de um ambiente de desenvolvimento que integrasse as principais características desejáveis numa ferramenta voltada para a concepção de sistemas de tempo real: suporte a definição de requisitos temporais, capacidade de simulação e geração de código executável. A escolha do ambiente SIMOO como base deste trabalho partiu da possibilidade de extensão do mesmo, uma vez que foi concebido no ambiente acadêmico local e, consequentemente, o seu código fonte encontrava-se disponível para modificações. Porém, o principal fator que levou a utilização do SIMOO foi a capacidade de simulação e animação do modelo desenvolvido, possibilitando a detecção de erros de concepção antes da implementação da aplicação.

Através da realização de alguns estudos de caso relatados no capítulo 5, verificou-se que apesar destes diagramas facilitarem o mapeamento do comportamento específico do modelo, os códigos definidos para a simulação e para a execução são incompatíveis, pois fazem uso de bibliotecas distintas, impedindo a utilização do mesmo modelo para as duas finalidades. Também foi possível detectar a necessidade de um formalismo alternativo para se especificar a comunicação inter-objetos. O objetivo deste formalismo é unificar os mecanismos de troca de mensagens utilizados pela simulação aos mecanismos suportados pelo ambiente de execução. Devido às restrições de tempo para a conclusão desta dissertação, ficam em aberto as soluções para os problemas em questão.

Ao final dos estudos de caso, constatou-se que o modelo SIMOO-RT e a aplicação AO/C++ gerada são fortemente relacionados. Esta constatação pôde ser observada a partir da análise e comparação dos códigos resultantes dos modelos desenvolvidos. Verificou-se que o número de linhas de código AO/C++ para um determinado modelo é significativamente menor que a versão de simulação e o código C++/QNX efetivo, comprovando a constatação verificada.

Para finalizar, como sugestão para trabalhos futuros sugere-se a migração da biblioteca de simulação para o ambiente QNX. Isto possibilitaria a execução simultânea de objetos simulados com objetos reais, além de possibilitar suporte para a simulação em tempo-real. Para exemplificar a utilidade desta migração, imagine o terceiro estudo de caso, onde os elevadores poderiam ser objetos simulados que estariam interagindo com os objetos de controle. Esta técnica, denominada co-simulação, permite a experimentação do sistema de controle sem a necessidade da existência do processo físico real. Um outro recurso que poderia ser adotado pelo ambiente é o da engenharia reversa, possibilitando ao usuário manipular o código gerado no ambiente de execução e depois atualizá-lo diretamente no modelo, diminuindo o risco de inconsistências e facilitando a sua manutenção e reutilização.

6.2 Trabalhos Relacionados

Conforme citado anteriormente, esta dissertação está inserida no contexto do projeto ADOORATA. Assim, outros trabalhos relacionados estão sendo desenvolvidos, dentre os quais alguns merecem ser destacados. O primeiro deles também faz parte de

uma dissertação de mestrado em desenvolvimento junto ao CPGCC, conduzida pela aluna Isabel de Souza [SOU98], o qual visa a definição de um metamodelo baseado em UML-RT, que será integrado ao SIMOO-RT para permitir a verificação de consistência dos modelos projetados.

Também encontra-se em desenvolvimento um trabalho em cooperação com o GMD da Alemanha [BEC99b], o qual objetiva "instrumentar" o código gerado pelo SIMOO-RT com instruções especiais, visando a obtenção de tomadas de tempo que serão utilizadas para fins de análise de desempenho da aplicação.

Um outro projeto de cooperação em andamento, envolvendo a UFRGS e uma empresa do setor alimentício, propõe a utilização do SIMOO-RT na implementação de sistemas supervisórios [BEC99a]. A primeira fase deste trabalho consta da definição de elementos gráficos de visualização, para serem utilizadas durante a simulação dos modelos. Na segunda etapa, este trabalho visa estender o conceito dos 'elementos de interface' presentes no SIMOO original para o ambiente QNX, reutilizando os elementos gráficos utilizados na simulação para a supervisão do sistema real.

Bibliografia

[BEC97] 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).

[BEC99a] BECKER, Leandro et al. Proposal of an Integrated Object-Oriented Environment for the Design of Supervisory Software for Real-Time Industrial Automation Systems. In: WORKSHOP ON OBJECT-ORIENTED REAL-TIME DEPENDABLE SYSTEMS, 1999, Santa Barbara, USA. **Proceddings...** [S.l.:s.n.], 1999.

[BEC99b] BECKER, Leandro et al. An Integrated Environment for the Complete Development Cycle of an Object-Oriented Distributed Real-Time Systems. In: INTERNATIONAL SIMPOSIUM ON OBJECT-ORIENTED REAL-TIME COMPUTING, 1999, Saint-Malo, France. **Proceddings...** [S.l.:s.n.], 1999.

[BIH92] BIHARY, T.; GOPINATH, P. Object-Oriented Real-Time Systems: Concepts and Examples. **Computer**, New York. p. 25-32, Dec. 1992.

[BOO91] BOOCH, Grady. **Object-Oriented Development**. Redwood City: Benjamin/Cummings Publishing Company, 1991.

[BRU98] BRUDNA, Cristiano et al. Sistema para Controle de Elevador em Tempo-Real. In: SEMINÁRIO INTERNO DO DEPARTAMENTO DE INSTRUMENTAÇÃO E CONTROLE , 2., 1998, Porto Alegre. **Anais...** [S.l.:s.n.], 1998. p. 21-24.

[COA90] COAD, Peter; YOURDON, Edward. **Object-Oriented Analysis**. Englewood Cliffs, NJ: Prentice Hall, 1990.

[COP97] COPSTEIN, Bernardo. **SIMOO: Plataforma Orientada a Objetos para Simulação Discreta Multi-Paradigma**. Porto Alegre: CPGCC da UFRGS, 1997. Tese de Doutorado.

[COX86] COX, B. **Object-Oriented Programming**. Sandy Hook: Addison Wesley Publishing, 1986.

[DOU98] DOUGLASS, Bruce. **Real-Time UML**: Developing Efficient Objects for Embedded Systems. [S.l.]: Addison Wesley, 1998.

[FLO98] FLORES, Aline; PEREIRA, Carlos. Uma Análise de Metodologias Orientadas a Objetos para o Desenvolvimento de Sistemas Distribuídos e de Tempo Real. In: SIMPÓSIO DE SISTEMAS DE SIMULAÇÃO E CONTROLE, 1., Rio de Janeiro. **Anais...** [S.l.:s.n.], 1998. p. 7-12.

[HAL91] HALANG, W.; STOYENKO, A. **Constructing Predictable Real-Time Systems**. [S.l.]: Kluwer Academic Publishers, 1991.

[KEM92] KEMMERER, R. e GHEZZI, C. Guest Editor's Introduction: Specification and Analysis of Real-Time Systems. **IEEE Transactions on Software Engineering**, New York, v. 18, n. 9, p. 766-777, 1992.

[KIM94a] KIM, Kane et al. A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potentials. In: COMPSAC, 1994, Taipei. **Proceedings...** [S.l.:s.n.], 1994. p. 392-402.

[KIM94b] KIM, Kane et al. Distinguishing Features and Potential Roles of the RTO.k Object Model. In: WORKSHOP ON OBJECT-ORIENTED REAL-TIME DEPENDABLE SYSTEMS, 1994, Dana Point. **Proceedings...** [S.l.:s.n.], 1994. p. 36-45.

[KIM95] KIM, Kane et al. A Timeliness-Guaranteed Kernel Model - DREAM Kernel - and Implementation Techiniques. In: RTCSA, 1995, Tokyo. **Proceedings...** [S.l.:s.n.], 1995. p. 80-87.

[KIM96] KIM, Kane et al. The DREAM Library Support for PCD and RTO.k programming in C++. In: WORKSHOP ON OBJECT-ORIENTED REAL-TIME DEPENDABLE SYSTEMS, 1996, Laguna Beach. **Proceedings...** [S.l.:s.n.], 1996. p. 59-68.

[MEY88] MEYER, Bertrand. **Object Oriented Software Construction.** New York: Prentice-Hall, 1988.

[NIL95] NILSEN, Kelvin. **Issues in the Design and Implementation of Real-Time Java**. Center for Advanced Technology Development, Iowa State University. Disponível em http://www.newmonics. com/webroot/technologies/ java/RTJI.ps (15 out. 1997).

[NIL96a] NILSEN, Kelvin. **Invited Note: Java for Real-Time**. Center for Advanced Technology Development, Iowa State Univ. Disponível em http://www.newmonics.com/webroot/technologies/java/java-rtsj.ps (15 out. 1997).

[NIL96b] NILSEN, Kelvin. **Embedded Real-Time Development in the Java Languagem**. Center for Advanced Technology Development, Iowa State University. Disponível em http://www.newmonics. com/webroot/technologies/ java/javaesce.ps (15 out. 1997).

[NIL97] NILSEN, Kelvin; LEE, Steve. **PERC™ Real-Time API (Draft 1.2)**. NewMonics Inc. Disponível em http://www.newmonics.com/ webroot/ technologies/java/rtjava_api.ps (15 out. 1997).

[PAR98] PARDI JUNIOR, Wilson. **Simulação e Visualização de Plantas Industriais usando Técnicas de Orientação a Objetos**. Porto Alegre: IEE/UFRGS, 1998. Relatório parcial de bolsa DTI.

[PER94a] PEREIRA, Carlos; DARSCHT, Pablo. Using Object-Orientation in Real-Time Applications: An Experience Report. In: TOOLS EUROPE, Versailles, França. **Proceedings...** [S.l.:s.n.], 1994.

[PER94b] 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, Orlando, EUA. **Proceedings...** [S.l.:s.n.], 1994.

[PER94c] PEREIRA, Carlos. Enhancing Real-Time Object-Oriented Analysis Methods. In: EUROMICRO WORKSHOP ON REAL-TIME PROGRAMMING, 6., Vaesteraas, Sweden. **Proceedings...** [S.l.:s.n.], 1994. p. 104-109.

[PER95] PEREIRA, Carlos. Temporal Reasoning on Object-Oriented Real-Time Specifications by using Constraint Propagation Techniques. In: IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, 20., FL-USA. **Proceedings...** [S.l.:s.n.], 1995.

[PER96b] PEREIRA, Carlos. Métodos de Análise de Sistemas Tempo-Real Usando Técnicas de Orientação a Objetos. SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, São Carlos, SP. **Anais...** [S.l.:s.n.], 1996.

[PID94] PIDD, Michael. **An Introduction to Computer Simulation.** In: WINTER SIMULATION CONFERENCE, 1994. **Proceedings...** [S.l.:s.n.], 1994

[RAT97] RATIONAL SOFTWARE CORPORATION. **UML Notation Guide**: Versão 1.1. Disponivel em http://www.rational.com (nov. 1997).

[RUM91] RUMBAUGH, James et al. **Object Oriented Modeling and Design.** Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

[SAN97] SANTOS, Sandro. **Guia de Referência da Linguagem AO/C++**. Disponível em ftp://caracol.inf.ufrgs.br/pub/simoo/RT/ ManualAO.doc.

[SEL94] SELIC, Bran et al. **Real Time Object Oriented Modelling**. [S.l.]: Willey Professional Computing - John Wiley and Sons, 1994.

[SHL92] SHLAER, S.; MELLOR, S. **Object Life Cycles**: Modeling the World in States. Englewood Cliffs, NJ: Yourdon Press, 1992.

[STA88] STANKOVIC, J. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. **Computer**, New York. p. 10-19, Oct. 1988.

[SOU98] SOUZA, Isabel e PEREIRA, Carlos. Proposta de um Metamodelo para a Linguagem de Modelagem UML–RT. In: SEMANA ACADÊMICA DO CPGCC, 3., Porto Alegre. **Anais...** Porto Alegre: CPGCC da UFRGS, 1998. p. 33-36.

[WIL94] WILHELM, Bob. **Structural Implications of Concurrent Execution Models for OO Real Time Systems**. In: OOPSLA, Portland, USA. **Proceedings...** [S.l.:s.n.], 1994.

[WILD98] WILD, Rafael; PEREIRA, Carlos. Contribuição à Avaliaçã de Características Tempo-Real em Barramentos de Campo. In: SIMPÓSIO DE SISTEMAS DE SIMULAÇÃO E CONTROLE, 1., Rio de Janeiro. **Errata...** [S.l.:s.n.], 1998. p. 13-18.

Informática *UFRGS*

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

"Ambiente de Modelagem e Implementação de Sistemas Tempo Real Usando o Paradigma de Orientação a Objetos "

por

Leandro Buss Becker

Dissertação apresentada aos Senhores:

Prof. Dr. Flávio Rech Wagner

Prof. Dr. João César Netto

Prof. Dr. Bernardo Copstein (PUCRS)

Vista e permitida a impressão.
Porto Alegre, 19/04/99.

Prof. Dr. Carlos Eduardo Pereira,
Orientador.