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

Ambiente Baseado em Componentes para o Desenvolvimento de Sistemas Computacionais Microcontolados Distribuídos

por

CLÁUDIO VIANNA VILLELA

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

Prof. Dr. Carlos Eduardo Pereira
Orientador

Porto Alegre, junho de 2001.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Villela, Cláudio Vianna

Ambiente Baseado em Componentes para o Desenvolvimento de Sistemas Computacionais Microcontolados Distribuídos / por Cláudio Vianna Villela. - Porto Alegre: PGCC da UFRGS, 2001.

103f.:il.

Dissertação (Mestrado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 2001. Orientador: Pereira, Carlos E.

1. Sistemas Distribuídos. 2. Componentes de Software. 3. Modelagem. 4. Protocolos de Comunicação. 5. Ambiente de Desenvolvimento. 6. Geração de Código. I. Pereira, Carlos E. II. Título.

Dedico este trabalho aos meus pais, Wellington e Maria Laura, as minhas irmãs Ana Laura e Izabel e a minha noiva Adriana.

Agradecimentos

Gostaria fazer um agradecimento especial ao meu orientador, Prof. Carlos Eduardo Pereira, pela consideração, respeito e amizade conquistados ao longo deste trabalho, e ao Prof. Flávio Wagner pelo apoio e compreensão dados no momento da conclusão deste trabalho. Da mesma forma gostaria de agradecer aos colegas Leandro Becker e Carlos Mitidieri, pelo valoroso apoio e amizade prestados ao longo deste período. Também gostaria de agradecer aos colegas integrantes do grupo SIMOO-RT Cristiano Brudna, Wilson Pardi Jr., Leandro Tibola, Isabel Fernandes, Rafael Wild, Marcelo Göetz e João Pacheco, que de muitas formas colaboraram com o bom desenvolvimento do meu trabalho. Ainda da mesma forma gostaria de agradecer aos representantes dos diversos segmentos da Universidade Católica de Pelotas, em especial aos colegas e amigos da Escola de Engenharia e Arquitetura, pelo apoio dado durante o período do meu mestrado. Por fim, deixo um agradecimento a todas aquelas pessoas não mencionadas, mas que de uma forma ou de outra colaboraram para o desenvolvimento deste trabalho.

Sumário

Lista de Abreviaturas

AO/C++	*Active Objects C++*
CAN	*Controller Area Network*
CASE	*Computer Aided Software Engineerin*g
CCM	*CORBA Component Model*
COM	*Components Object Model*
CORBA	*Common Object Request Broker Architecture*
DCOM	*Distributed Components Object Model*
DDE	*Deployment Diagram Editor*
DII	*Dynamic Invocation Interface*
DSI	*Dynamic Skeleton Interface*
ECB	*Event Channel Broker*
ECH	*Event Channel Handler*
FC	*Freqüência Cardíaca*
FR	*Freqüência Respiratória*
GIOP	*General Inter ORB Protocol*
GUI	*Graphics User Interface*
GUID	*Globally Unique identifier*
IDL	*Interface Definition Language*
IIOP	*Internet Inter ORB Protocol*
IO	*Input/Output*
JVM	*Java Virtual Machine*
LOO	*Language Object Oriented*
MET	*Model Editor Tool*
MIDL	*Microsoft's Interface Description Language*
OCX	*Open Compact Exchange*
OLE	*Object linking and Embedding*
OMG	*Object Management Group*
ORB	*Object Request Broker*
ORPC	*Object Remote Procedure Call*
PID	*Proporcional, Integral e Derivativo*
POA	*Portable Object Adapter*
P/S	*Publisher/Subscriber*
RPC	*Remote Procedure Call*
RMI	*Remote Method Invocation*
ROOM	*Real-time Object-Oriented Modeling*
TCP/IP	*Transmission Control Protocol/Internet Protocol*
TP	*Temperatura*
UCRMPF	*Unidade de Calor Radiante com Monitorização de Parâmetros Fisiológicos*
UML	*Unified Modeling Language*
UNIX-TR	*UNIX de Tempo Real*

Lista de Figuras

Resumo

A modelagem e desenvolvimento de sistemas embarcados ("*embedded systems*") de forma distribuída, tende a ser uma tarefa extremamente complexa, especialmente quando envolve sistemas heterogêneos e sincronização de tarefas. Com a utilização do modelo de componentes de software é possível descrever, de uma forma simplificada, todos os elementos de distribuição e de comunicação para este tipo de sistemas. Neste sentido, a especificação de uma ferramenta capaz de auxiliar na modelagem e no desenvolvimento deste tipo de aplicação, certamente irá tornar o trabalho mais simples. Esta dissertação inicia por uma análise comparativa entre as tecnologias passíveis de serem utilizadas na definição de sistemas distribuídos heterogêneos, focando-se principalmente nas metodologias de modelagem, e nos mecanismos e *middlewares* de comunicação. Dos conceitos formados a partir desta análise é descrita uma ferramenta, baseada em componentes de software. A ferramenta é uma extensão do projeto SIMOO-RT, onde foram adicionados os conceitos de componente de software, biblioteca de componentes e diagrama de implantação. Além disso, foram realizadas modificações no sistema de geração de código, para dar suporte aos novos conceitos da ferramenta. A dissertação termina com a descrição de alguns estudos de caso utilizados para validar a ferramenta.

Palavras-Chave: Sistemas Distribuídos, Componentes de Software, Modelagem, Protocolos de Comunicação, Ambiente de Desenvolvimento e Geração de Código.

Abstract

Title: “Framework for Component-Based Development of Distributed Microcontrolled Systems”

Distributed embedded systems design and modeling can be a very complex task, specially when including heterogeneous systems and concurrent tasks synchronization. The software component model proposes interesting concepts to describe all distributed elements and their communication links. In this sense, specific framework can help dealing with such complexity. This dissertation begins with a comparative analysis between some technologies used in distributed heterogeneous systems, mainly focusing on their core concepts, such as the model methodologies, the proposed communication mechanisms middleware. Based on this evaluation a framework has been developed to support the modeling and implementation tasks of component-based applications. The framework is an extension of SIMOO-RT project by including the concept of software component, a component library template, a deployment diagram and a new code generator. The present text describes into details the specification of the framework, its implementation as an extension to SIMOO-RT, as well as some case studies.

Keywords: Distributed Systems, Software Components, Modeling, Communication Protocols, Develop Environment and Code Generation.

1 Introdução

1.1 Sistemas de Automação

Uma análise da evolução dos sistemas de automação industrial permite identificar um crescente uso de sistemas computacionais, em especial de sistemas microcontrolados, nos quais sensores e atuadores são interligados em redes de comunicação industrial, caracterizando um sistema computacional distribuído. Estes dispositivos oferecem informações sobre o estado do sistema e permitem aos controladores atuarem para modificar o estado do mesmo. A natureza deste tipo de sistema tende a impor requisitos temporais para tomadas de decisões e de reação a determinados eventos. Desta forma, os sistemas de automação encontram-se inseridos no contexto dos sistemas distribuídos de tempo real [FRI97].

Com a evolução dos microprocessadores e microcontroladores, estes passaram a ser incorporados nos dispositivos sensores e atuadores, aumentando enormemente a capacidade local de processamento destes dispositivos, sem necessariamente implicarem em grandes alterações visuais dos dispositivos. Diz-se que estas capacidades encontram-se "imersas" - ou "*embedded*" nos dispositivos. Desta forma, um dispositivo sensor não é mais somente responsável pela aquisição do sinal, mas também pode realizar localmente operações mais elaboradas, como a conversão e digitalização dos valores lidos em valores de engenharia, bem como a sinalização de alarmes. Esta abordagem diminui o número de informações que irão transitar pela rede, o que permite respostas mais rápidas na comunicação dos elementos distribuídos.

A evolução da indústria da microeletrônica tem melhorado a relação custo benefício dos sistemas microcontrolados. Os microcontroladores disponíveis atualmente possuem capacidade de processamento similar ou maior do que sistemas considerados como de grande porte há menos de uma década atrás, possuindo custos bastante acessíveis. Os avanços tecnológicos não se refletem somente num aumento da capacidade de

processamento localizado de informações mas também numa maior velocidade para troca de informação entre dispositivos distribuídos. Neste sentido pode-se idealizar sistemas onde, ao invés de grandes controladores coordenando dezenas ou centenas de dispositivos de IO, utilizar-se dispositivos de IO com capacidade computacional de controlar e coordenar operações de forma sincronizada entre eles. Tem-se uma descentralização da lógica de controle, tornando desnecessária a presença de um controlador central.

Entretanto, a fim de se obter uma melhor utilização destas arquiteturas microntroladas distribuídas, novas técnicas de projeto e implementação dos programas computacionais deve ser desenvolvida. Técnicas tradicionalmente utilizadas, como metodologias *ad-hoc* e linguagem assembler apresentam uma série de limitações. Um exemplo de proposta para o uso de modernas técnicas de programação para o desenvolvimento destas aplicações é apresentada em Brudna [BRU2000], que com uma arquitetura baseada em um versão reduzida do sistema operacional Linux, chamada μClinux, executando sob um microcontrolador Motorola, possibilita o desenvolvimento de programas concorrentes usando o conceito de objetos distribuídos, os quais se comunicam usando protocolos de comunicação do tipo *Controller Area Network* (CAN).

O desenvolvimento de aplicações microcontroladas distribuídas favorece a ampliação e manutenção das aplicações, além de facilitar o desenvolvimento de aplicações mais confiáveis seja pela possibilidade do uso de sistemas redundantes ou pelo fato de que não se tem mais um ponto único de falha no sistema. Para obter estas vantagens o projetista da aplicação deverá resolver alguns problemas adicionais, tais como a definição de garantias de sincronização, segurança e integridade das informações disponíveis por todos os elementos distribuídos, o que irá aumentar a complexidade da aplicação [TAM92].

Além desta complexidade, em diversas aplicações torna-se necessária a conexão entre diversos tipos de plataformas de hardware e software, que possuem características distintas que dependem da funcionalidade e do fabricante. Por exemplo, uma aplicação que possui sensores com plataformas dedicadas, atuadores e controladores baseados em sistemas microcontrolador, e um supervisório sendo executado em uma plataforma PC.

Desta forma estes tipos de sistemas de automação podem ser classificados como Sistemas Heterogêneos Distribuídos de Tempo Real.

Infelizmente, métodos convencionais de engenharia de software mostram-se ineficientes no desenvolvimento de sistemas heterogêneos distribuídos de tempo real, uma vez que estes, além de não considerarem requisitos temporais, usualmente não abordam o projeto de hardware e software de forma integrada. Diversos autores assim como Kauler [KAU99], Shlaer [SHL92] e Pereira [PER94] sugerem que para resolver problemas com tantas características e restrições, deve-se utilizar os conceitos da orientação a objetos na concepção da solução. Estes autores demostram que o objeto é uma unidade natural de distribuição, permitindo a descrição de todos os elementos de distribuição e seus relacionamentos. Mais recentemente, a partir do conceito da orientação a objetos foram definidas propostas baseadas no conceito de componentes [SZY99]. Uma das diferenças entre estes dois modelos, é que o modelo de componentes obriga uma definição mais rigorosa dos relacionamentos entre os componentes.

Esta definição é realizada através de uma interface, a qual é normalmente descrita por uma linguagem específica chamada *Interface Definition Language* (IDL). Além disso, para a definição completa do componente é necessário informar quais são as suas dependências de contexto, ou seja, quais são as interfaces de entrada que obrigatoriamente devem ter as suas requisições preenchidas para que o componente funcione corretamente. Em 1996 na Conferência Européia sobre programação orientada a objetos (ECOOP), Szyperski formulou a seguinte definição: *"Um componente de software é uma unidade de composição com especificações contratuais de interfaces e independência de contexto. Um componente de software pode ser desenvolvido independentemente e ser submetido para a composição por outro participante"*.

1.2 Motivações

Utilizando-se os conceitos da orientação a objetos é possível representar de forma bastante clara e eficiente um sistema de automação industrial [PER96]. Fazendo-se uma analogia entre o paradigma da orientação a objetos e o de componentes, pode-se representar cada objeto como um componente de software. Neste sentido as diferenças entre os modelos ficam reduzidas às formas de comunicação. No modelo de objetos, cada objeto pode estar relacionado aos demais através de associações ou agregações, o que obriga que esse objeto possua uma referencia explícita aos objetos a ele relacionados. Já no modelo de componentes, os componentes relacionam-se apenas através de interfaces bem definidas de comunicação. Para uma informação ser lida, escrita ou executada em um outro componente, este deve interagir com uma de suas interfaces, a qual irá requisitar esse comando às demais interfaces conectadas a ela. A forma da comunicação, bem como o formato da interface, devem ser os mesmos para todos os componentes relacionados. Alguns dos padrões mais conhecidos são o CORBA Componente Model da OMG [OMG2000], o JavaBeans da Sun [SUN00, JAV2000] e o DCOM da Microsoft [MIC2000].

Para efetiva adoção de uma metodologia baseada em componentes para análise e projeto de sistemas, torna-se necessária a 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 Engeneerin*g), 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 de 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.

O uso de componentes de software na modelagem de sistemas distribuídos baseados em PCs e/ou microcontroladores pode proporcionar subsídios para a construção de modelos mais simplificados e um melhor reaproveitamento destes componentes. Já a concepção de uma ferramenta capaz de modelar a partir de componentes, e auxiliar na geração de código

para sistemas distribuídos heterogêneos, certamente irá facilitar a construção de aplicações em sistemas de automação industrial.

O aumento da capacidade computacional e de interconexão dos novos microcontroladores, que impulsionaram um aumento da construção de aplicações *embedded* [SCH01, LEE93], e o crescente interesse da comunidade científica demonstrado pelo aumento de publicações na área, como por exemplo o visto em [BRU00, FRI97, KAM99, KAU99, LEE00, MOC92, SIE00, PER2001], são outros indicadores da relevância e importância do tema abordado por esta dissertação.

A presente proposta foi definida no contexto de um projeto de pesquisa desenvolvido na UFRGS e que versão sobre o uso de conceitos de objetos distribuídos e componentes no desenvolvimento de sistemas tempo-real distribuídos, tais como os sistemas de automação industrial. Mais especificamente, a ferramenta de suporte computacional a ser desenvolvida deverá ser integrada ao ambiente SIMOO-RT, o qual inclui recursos para a modelagem, simulação e geração de código para sistemas de tempo real.

1.3 Objetivos

Com base no contexto apresentado anteriormente, esta dissertação tem como objetivo principal a proposta de uma metodologia e suporte computacional para desenvolvimento de aplicações distribuídas usando conceitos de componentes. O ambiente proposto estende as funcionalidades da ferramenta SIMOO-RT, permitindo a criação de componentes de software. Alem disso, objetiva-se auxiliar no mapeamento para a arquitetura de hardware alvo da aplicação, bem como permitir a definição do protocolo de comunicação entre os componentes. Também foi criado um conjunto de componentes genéricos para sistemas automatizados, a partir dos quais foram desenvolvidos estudos de caso para a validação do modelo. De forma mais específica, o desenvolvimento desta dissertação visou atingir os seguintes objetivos:

- Criar uma biblioteca de componentes para automação industrial (atuadores, sensores e controladores);

- Implementar componentes para a construção de estudos de caso (barramento CAN, PC rodando Linux e Microcontroladores rodando μClinux);

- Avaliar o modelo de componentes para a descrição de sistemas computacionais microcontrolados distribuídos, em termos da capacidade e facilidades de descrição de aplicações complexas e da modularidade do código gerado;

- Comparar o modelo de objetos com o modelo de componentes, a partir dos estudos de caso desta dissertação, sob os aspectos da reusabilidade do sistema como um todo ou de suas partes, da facilidade de modelagem e da performance da aplicação, apontando as vantagens e desvantagens do modelo proposto.

- Estender a ferramenta SIMOO-RT a fim de incorporar as seguintes funcionalidades:

    - Criar o suporte à descrição e configuração da arquitetura de Hardware e Software, para possibilitar a descrição dos sistemas computacionais que fazem parte do modelo da aplicação;

    - Modificar o gerador de código para possibilitar a criação dos componentes a partir dos modelos;

1.4 Organização do Texto

O restante desta dissertação encontra-se organizado da seguinte forma: o capítulo 2 apresenta uma análise do estado da arte, discutindo as diferenças entre as metodologias baseadas em objetos e em componentes, através de uma análise dos conceitos e dos mecanismos de comunicação e de distribuição dos dois modelos; o capítulo 3 apresenta o projeto conceitual de uma ferramenta baseada em componentes; o capítulo 4 descreve a implementação da proposta desenvolvida e a sua incorporação ao ambiente SIMOO-RT. O capítulo 5 aborda a realização de alguns estudos de caso, que validam a implementação realizada e, finalmente, o capítulo 6 apresenta as conclusões observadas e indica os trabalhos futuros para continuidade desta pesquisa.

2 Análise do Estado da Arte

2.1 Introdução

Atualmente sistemas computacionais baseados em microcontroladores podem ser encontrados em uma grande quantidade de dispositivos utilizados não só na área industrial, como na comercial e residencial [BOH95, TAK99]. Originalmente, estes dispositivos foram desenvolvidos para trabalhar de uma forma isolada, ou seja, realizando apenas o processamento local de informações recebidas e transmitidas através de suas interfaces locais com sensores e atuadores. Atualmente, existe uma forte demanda para que estes dispositivos sejam capazes de integrarem-se a uma rede de dispositivos, trocando informações e atuando sincronizadamente com outros dispositivos no processamento de informações para realização de diferentes funcionalidades, tais como operações de controle, supervisão, e diagnostico de falhas.

Sendo assim, aplicações são inerentemente distribuídas, e devem ser formadas pela composição de diversos destes dispositivos. A simples modificação desta composição pode alterar os objetivos desta aplicação. Este conceito favorece o desenvolvimento de aplicações que estejam baseadas nos mesmos dispositivos, pois o projetista terá que criar este dispositivo apenas uma única vez. Em sistemas de automação industrial esta característica é bastante comum, onde numa planta industrial diversos sensores ou atuadores devem trabalhar de forma sincronizada.

Apesar desta vantagem, questões adicionais relacionadas à forma da distribuição e da sincronização destes dispositivos devem ser definidas antes da implementação destes conceitos. Neste sentido, este capítulo irá apresentar alguns destes conceitos básicos e o estado atual das tecnologias utilizadas para o desenvolvimento destas aplicações. Inicialmente são apresentados os conceitos básicos sobre sistemas distribuídos. Na terceira seção deste capítulo são apresentadas duas das metodologias mais utilizadas para a definição de aplicações distribuídas, a baseada em objetos e baseada em componentes.

Apesar destas metodologias já serem consagradas, os conceitos fundamentais destas variam de acordo com a bibliografia utilizada, sendo fundamental para este trabalho a definição de um conceito único. Na seção seguinte são apresentados três tipos de mecanismos de comunicação, RMI, portas e canais, quem podem ser implementados tanto no modelo de objetos como no de componentes. Na penúltima seção são apresentados cinco *middlewares* de comunicação, CORBA, COM, Java-RMI, AO/C++ e *Publisher/Subscriber*. Esta seção descreve a forma como estes *middlewares* dão o suporte necessário para a implementação das metodologias de modelagem de sistemas e dos mecanismos de comunicação. Ou seja, um mecanismo de comunicação define as regras de comunicação no nível de infra-estrutura, o que significa a forma de implementar a comunicação entre os elementos de distribuição. Já o *middleware* de comunicação define as regras que dizem respeito à identificação dos componentes da comunicação, sincronização entre comunicações e manutenção de características especiais (Ex.: Temporais). Na última seção deste capítulo é feito um resumo do mesmo e são realizadas algumas considerações voltadas para as aplicações em automação industrial.

2.2 Sistemas Distribuídos

Uma definição simplista de computação distribuída é a de vários sistemas computacionais independentes trabalhando em conjunto. Pode se dizer ainda que um sistema distribuído é constituído por três características fundamentais [MUL95]:

- Múltiplos Sistemas Computacionais: Numa aplicação distribuída sempre existirá mais de um sistema computacional independente, onde certamente cada um deles irá possuir uma porção de processamento, alguma memória local dedicada e possivelmente algumas interfaces de entrada e saída. Em alguns casos pode-se considerar que processos que estejam executando sob um mesmo processador, mas que possuam áreas de memória independentes, possam ser considerados sistemas computacionais independentes;

- Conexões: Cada sistema computacional independente possui um sistema de comunicação padronizado.

- Estado Compartilhado: Os sistemas computacionais devem cooperar para a manutenção de um conjunto de estados compartilhados. Isto significa que um estado específico de uma aplicação distribuída é formado pelo conjunto de estados individuais de cada sistema computacional, e que estes sistemas computacionais, devem manter estes estados coordenados para a manutenção da correção dos estados gerais da aplicação;

Para a interconexão de sistemas computacionais independentes é necessário manutenção de mais quatro características adicionais [MUL95]:

- Falhas Independentes: Quando ocorrer uma falha geral em um ou mais destes sistemas computacionais, a aplicação deve procurar manter o seu funcionamento total ou parcial. Esta característica deve ser mantida apenas quando isto não acarretar perdas importantes de desempenho ou funcionalidade, definidas pelo projetista da aplicação.

- Comunicação Confiável: O sistema deve ser capaz de tentar recuperar conexões ou mensagens perdidas. Caso não o consiga deve ser capaz de informar o ocorrido.

- Comunicação Segura: O meio de comunicação deve disponibilizar formas para garantir a confidencialidade e integridade das informações, permitindo que apenas os usuários que possuam direitos sobre estas informações possam acessá-las.

- Custo da Comunicação: A conexão entre sistemas computacionais pode não suprir as exigências de largura de banda necessárias, ou possuir um tempo de atraso muito grande, ou ainda ter um custo financeiro muito elevado.

Além da manutenção destas características existem muitas outras dificuldades no desenvolvimento de sistemas distribuídos, especialmente para aplicações com requisitos temporais. Heterogeneidade dos sistemas computacionais, sincronização de relógio nos diferentes nodos, atrasos impostos pelos meios de comunicação e procedimentos não determinísticos impostos pelo meio de comunicação, são exemplos destes problemas.

A fim de reduzir a complexidade na programação e uso do sistema, o modelo adotado deve tornar transparente tanto a localização física dos objetos (ou seja, chamadas locais devem ser idênticas a chamadas remotas) quanto à concorrência no processamento de serviços, sejam estes locais ou distribuídos. Um problema deste modelo está no gerenciamento do sistema distribuído. Isto ocorre pois este gerenciamento provavelmente também estará distribuído, obrigando a sincronização dos elementos de gerenciamento.

A subdivisão de um conjunto de problemas em níveis de abstração facilita a compreensão e a implementação da solução. Neste sentido, podemos dividir a comunicação entre objetos distribuídos em 4 níveis diferentes [JOH94], mostrados na Figura 2.1.

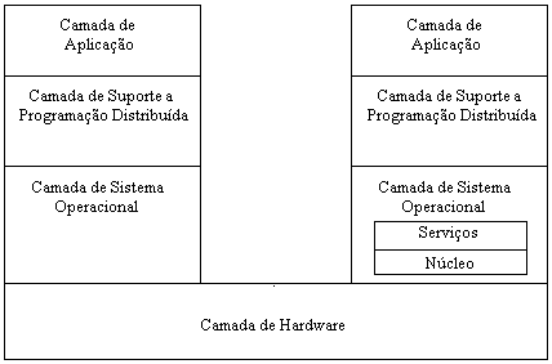

FIGURA 2.1 - Modelo básico de comunicação de um Sistema Distribuído

Na camada de aplicação são implementadas as operações relacionadas diretamente com a aplicação corrente, onde um objeto visualiza um outro objeto, distribuído ou local, da mesma forma. Na camada de suporte à programação distribuída as requisições da camada de aplicação são tratadas, identificando-se o receptor e a forma de comunicação que deve ser realizada. A camada do sistema operacional é responsável pelo encaminhamento das requisições da camada anterior, e pode estar dividida nas subcamadas de Serviços e Núcleo. Nela as operações realizadas com objetos distribuídos são encaminhadas para a transmissão em um protocolo de comunicação comum aos objetos relacionados nesta comunicação (subcamada de serviços). Se os objetos forem locais a comunicação é mapeada para o modelo interno de comunicação do sistema operacional (subcamada do núcleo). Por fim, na camada de hardware é realizada a transmissão

propriamente dita, onde ela é responsável pela detecção, sinalização e se possível correção de erros durante esta transmissão.

Apesar de melhorar a compreensão da aplicação, a simples divisão em níveis de abstração não é suficiente para identificar aspectos importantes do modelo desta aplicação. Por exemplo, os tipos de entidades existentes na aplicação (objetos, componentes, processos, ou outros) e os mecanismos pelos quais elas se relacionam (chamadas RMI, portas ou canais de eventos).

2.3 Metodologias de Modelagem de Sistemas Distribuídos

Metodologias baseadas em objetos e mais recentemente em componentes vem sendo adotadas com sucesso no desenvolvimento de sistemas distribuídos em diversas áreas de aplicação. Recentemente, a comunidade cientifica adotou o termo *Distributed Object Computing* (computação com objetos distribuídos) para descrever esta área de pesquisa, sendo que o numero de trabalhos e conferências nesta área vem aumentando significativamente nos últimos anos. Em função de ser uma área recente, os conceitos fundamentais tendem a variar de acordo com a bibliografia de referência utilizada. Neste sentido, as próximos seções procurarão definir estes conceitos que servirão de base para esse trabalho.

2.3.1 Orientação a Objetos

Os elementos básicos desta metodologia são os objetos e as classes. Para a sua total compreensão é necessária a definição específica destes conceitos. Uma classe especifica a organização e representação interna dos dados de um objeto, bem como as operações que o mesmo é capaz de executar. Enquanto um objeto é uma entidade concreta que desempenha um papel no sistema, uma classe captura a estrutura e o comportamento comuns a todos os objetos relacionados. Classes, em geral, não têm sentido isoladas e por isso mantêm relacionamentos com as demais classes do sistema. Relacionamentos entre classes indicam uma forma de compartilhamento ou alguma forma de relacionamento semântico.

Segundo Booch [BOO94], um modelo baseado em objetos deve possuir sete propriedades, sendo quatro fundamentais (abstração, encapsulamento, modularização e hierarquia), e 3 complementares (tipagem, concorrência e persistência):

- A Abstração é uma simplificação que enfatiza os aspectos mais importantes e negligencia os demais.

- O Encapsulamento é um conceito complementar ao da abstração. A abstração se preocupa com o comportamento observável de um objeto, enquanto que o encapsulamento focaliza o suporte que dará a manutenção deste comportamento. O encapsulamento gerencia a complexidade na medida que somente informações realmente necessárias a outros módulos são feitas visíveis na interface dos módulos, todas as outras são "escondidas" internamente aos módulos. O encapsulamento permite uma clara separação entre aspectos de uso e de implementação, e é, em geral, obtido através do que se chama de *information hiding*, que é o processo de esconder as informações de um objeto que não contribuem para o entendimento de suas características essenciais, vistas externamente. Tipicamente a estrutura e a implementação dos métodos de um objeto são restritas ao escopo interno a um objeto.

- Modularização é o ato de particionar um programa em elementos individuais, fracamente acoplados, com o objetivo de reduzir a sua complexidade criando um conjunto de limites bem definidos. Módulos servem de repositórios onde se armazenam classes e objetos relacionados.

- Hierarquia é a forma de relacionamento entre as classes de um programa. Os dois tipos de hierarquias mais importantes dentro de um sistema são as hierarquias de herança que descrevem relações do tipo “é um” ou “é do tipo de” e as hierarquias de agregação que descrevem relações como “contém” ou “é parte de”. Existe ainda um terceiro tipo bastante comum que é a associação. Neste caso as relações de conhecimento implicam apenas que um objeto conhece outro, ou seja, pode enviar mensagens para ele mas não é responsável pelo mesmo [BRO90, GAM95].

- Tipagem é a propriedade que a metodologia possui de formar modelos de referência. Na orientação a objetos cada classe define um tipo diferente que pode ser instanciado na forma de um ou mais objetos. De maneira complementar, pode-se dizer que uma classe descreve um conjunto de objetos que compartilham uma estrutura e um comportamento comuns, ou seja, compartilham a mesma implementação [GAM95].

- A Concorrência ocorre quando partes de um mesmo programa são executadas simultaneamente em fluxos de execução distintas (também denominados de *threads*). Objetos são unidades naturais para execução concorrente, permitindo que aplicações do mundo real com concorrência intrínseca sejam modeladas naturalmente. Além disso, o modelo de objetos concorrente é muito eficaz na descrição de sistemas distribuídos e paralelos.

- Persistência é a propriedade de um objeto cuja existência transcende tempo e/ou espaço. Um objeto em software ocupa certo espaço e existe por um certo período de tempo. Persistência está relacionada a objetos cujo tempo de vida transcende o tempo de vida de um programa individual. Não apenas o estado do objeto deve persistir como também sua classe, de maneira que o estado seja re-interpretado sempre da mesma forma.

2.3.2 Componentes

Um componente típico está organizado internamente em diversos outros componentes, variáveis, rotinas tradicionais (procedures/bibliotecas) e/ou classes/objetos [HAR98A, HAR98D]. Normalmente, ele pode ser construído a partir de um módulo que contenha apenas algumas classes ou procedimentos relacionadas. Apesar disso, o conceito de componente não está restrito ao conceito de um módulo. A modularidade é apenas uma propriedade fundamental deste modelo, sendo um pré-requisito indispensável para a construção de componentes. Algumas das características dos componentes são [SZY99]:

- É uma unidade de desenvolvimento independente. Precisa estar bem separado no seu meio dos outros componentes, encapsulando desta forma suas características;

- É uma unidade de composição, onde as partes desta composição podem ser construídas de forma terceirizada. Neste caso são necessárias especificações claras do que cada componente da composição necessita e disponibiliza. Em outras palavras, um componente necessita encapsular sua implementação e interagir com seu ambiente através de interfaces bem definidas.

- É indistinguível de suas próprias cópias. Toda e qualquer cópia de um componente possui o mesmo conjunto de interfaces não sendo por isso distinguíveis entre si. A distinção pode ser construída apenas através do significado das ligações entre as interfaces.

Construtivamente, classes e componentes são similares. Obviamente existem algumas diferenças, relacionadas principalmente aos tipos de elementos de composição, que são facilmente contornadas no modelo de objetos. A principal diferença entre estas duas metodologias está no relacionamento dos elementos de distribuição. Na orientação a objetos é possível realizar diversos tipos de relacionamentos entre objetos de uma mesma aplicação (por exemplo, associação ou agregação). Já no modelo de componentes, cada componente relaciona-se com os demais somente através de suas interfaces.

As Interfaces permitem a clientes deste componente, normalmente outros componentes, acessarem os serviços fornecidos por ele. Normalmente, um componente pode possuir várias Interfaces correspondendo a diferentes pontos de acesso a sua estrutura interna. Cada interface pode fornecer um serviço diferente, suprindo necessidades de diferentes clientes [SZY99]. No conceito puro de interface (em linguagem de programação Java), ela é uma classe sem implementações que possui declarações de operações [CAL99] e é através das declarações das operações dos componentes em sua Interface que esta possibilita interagir com outros componentes. Na prática uma interface é um conjunto de operações que podem ser chamadas pelo cliente, que tem a função de atuar como um *middleware* entre o componente e o seu cliente. A semântica de cada operação é

especificada, e desta especificação saem as regras que regem as comunicações entre o componente e o seu cliente. A principal vantagem da definição destas forma de relacionamento está na total independência proporcionada entre os diversos componentes de uma mesma aplicação. Isso significa maior possibilidade de reutilização de código e uma maior simplicidade na construção de aplicações baseadas em sistemas computacionais distribuídos e heterogêneos.

A reutilização de código, segundo o modelo de componentes, deve ser realizada através da implementação de um repositório ou biblioteca de componentes. Assim sendo, seria possível construir uma aplicação utilizando-se apenas alguns componentes desta biblioteca, interligando suas interfaces de acordo com a aplicação pretendida, sem necessidade de nenhuma codificação especial. Obviamente novos componentes também poderiam ser criados na aplicação, para serem ou não adicionados à biblioteca, mas isso exigiria a codificação destes novos componentes.

Existem três principais competidores no que tange a propostas de padronização de componentes [HAR98A, SZY99]: CORBA *Component Model* promovido pelo *Object Management Group* (OMG), um consórcio de mais de 700 empresas; DCOM proposto pela Microsoft e JavaBeans proposto pela empresa SUN.

Como já sugerido pelo nome o CORBA *Component Model* (CCM) é baseado na arquitetura *Common Object Request Broker Architecture* (CORBA), sendo um modelo direcionado para a definição de componentes servidores e suas interações, através de pacotes de desenvolvimento para múltiplas linguagens e com a atenção a conceitos como segurança, transações, eventos e persistência [SIE2000]. No CCM um componente pode ser comparado a uma especialização do *Portable Object Adapter* (POA) do modelo CORBA convencional. Interfaces são definidas na *Interface Definition Language* (IDL) com mapeamentos disponíveis para linguagens de programação C, C++, Smaltalk e Java.

O *Microsoft's Components Object Model* (COM) foi desenvolvido pela empresa Microsoft, a fim de possibilitar que o seu *Object linking and Embedding* (OLE), sucessivamente para componentes e componentes para a Internet (ActiveX), pudesse ser construído em várias linguagens. Existem neste modelo dois elementos chave: as interfaces

COM e o sistema para registro e troca de mensagens entre estas interfaces [HAR98A]. As interfaces COM são assinaturas codificadas em *Microsoft's Interface Description Language* (MIDL). Uma diferença para o modelo CORBA é que um programador não necessita descrever a sua interface nesta linguagem própria (MIDL), ele deve escrever uma aplicação numa linguagem tradicional, como C++ ou Visual Basic, e depois utilizar um compilador próprio para MIDL. Esta interface COM é implementada como um objeto COM, ou seja, objetos OLE, OCX ou ActiveX. Em qualquer um destes casos, um objeto COM difere do modelo tradicional da orientação a objetos, pois este objeto pode ser um módulo de procedimentos, uma composição de objetos ou até mesmo uma aplicação inteira.

Por fim, a Sun Microsystems desenvolveu padrões centralizados para Internet e sua popular linguagem de programação orientada a objetos Java. Os componentes são chamados de *Java Beans*, escritos em linguagem Java, e são portáveis para qualquer ambiente que suporte *Java Virtual Machine* (JVM). Segundo a SUN [SUN2000] "Um JavaBean é um componente reutilizável de software que pode ser manipulado visualmente em uma ferramenta de construção". Basicamente o JavaBeans é um pacote de programas, funcionalidades e documentações, que estendem os conceitos do Java, possibilitando a execução de componentes. Este pacote, através da adição de um conjunto de características, transforma classes Java em componentes. Alguns JavaBeans podem ser uma interface GUI simples, tais como um botão ou uma barra de rolagem, ou um sofisticado componente de visualização de dados, como alguns utilizados em banco de dados. Nem todos os JavaBeans possuem uma interface GUI, mas todos são construídos a partir de uma ferramenta visual. Estas ferramentas visuais modificam um pouco a forma e os conceitos destes componentes, mas de forma geral elas possuem as seguintes características [HAR98A., HAR98C]:

- *Introspection*: capacidade da análise de como o componente trabalha;
- *Customization*: capacidade de criar padrões de interfaces e comportamentos;
- *Events*: capacidade de comunicar-se através de eventos;
- *Properties*: capacidade de associar propriedades a elementos de *customizatiom* e de programação;
- *Persistence*: capacidade de existência de um objeto, transcendendo o tempo e/ou espaço (similar à persistência da orientação a objetos).

2.4 Mecanismo de Comunicação

2.4.1 RMI

Este é o modelo de comunicação adotado pela maior parte das aplicações orientadas a objetos, sendo suportado por ambientes de execução (*middleware*) orientados a objetos, como por exemplo o CORBA, o DCOM e o Java-RMI (vide próxima seção). De acordo com este paradigma, todos os relacionamentos entre objetos devem ser descritos na etapa de projeto. Quando este relacionamento for uma associação entre objetos, significa que deve existir uma comunicação distinta destes objetos. Isso caracteriza uma comunicação ponto a ponto em um estilo cliente/servidor, onde o objeto cliente deve conhecer a localização do objeto servidor [PER2001]. A chamada de um método é sintaticamente similar à chamada de um procedimento em uma linguagem convencional, com a diferença que em um sistema orientado a objetos a chamada de um método existe no contexto de uma instância de uma classe. Esta estratégia vem sendo proposta em metodologias como Rumbaugh [RUM91] e Booch [BOO94].

Este modelo de interação pode ser executado de forma síncrona ou assíncrona. No primeiro caso, um sistema similar à chamada remota de procedimento (RPC) deve ser adotado [COD88]. Quando um objeto chama um método de outro objeto, o fluxo de execução é bloqueado, e o controle é transferido para o sistema de invocação de métodos remotos. Por outro lado, quando a interação é assíncrona, não existe a necessidade desta transferência de controle. A Figura 2.2 descreve o conceito exposto:

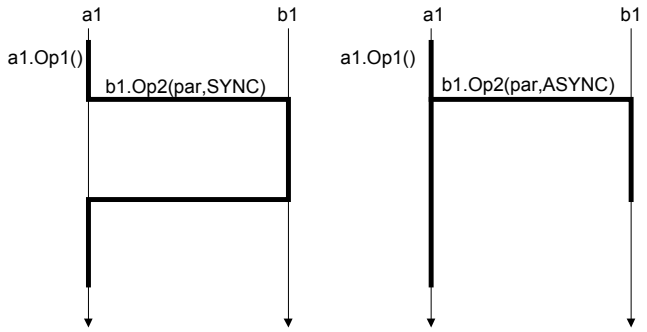

FIGURA 2.2 - Modelo RMI

2.4.2 Portas

Uma forma alternativa ao RMI para a comunicação entre objetos distribuídos é a definida na metodologia ROOM [SEL94]. Esta metodologia define o conceito de porta de comunicação, como sendo o único meio de relacionamento entre objetos. Cada objeto pode possuir inúmeras portas, sendo que cada porta está relacionada a um conjunto de mensagens de entrada e ou saída. Internamente o objeto é descrito através de uma máquina de estados, que pode ter o seu estado corrente alterado, conforme o tipo das mensagens recebidas através das portas. Esta metodologia também suporta o conceito de protocolo, que em geral significa que as mensagens de saída de uma porta devem ser aceitas como mensagens de entrada em uma outra porta conectada. No caso do ROOM, as portas de um protocolo são denominadas portas conjugadas. Em geral, o estabelecimento de tais conexões deve ser realizado antes da execução do sistema, em tempo de projeto ou de configuração.

Um dos principais objetivos das portas é o de criar um mecanismo de comunicação que permita, de forma facilitada, a modelagem de sistemas sem a necessidade da definição de dependências externas aos objetos. Quando uma dependência for inevitável ela deve ser descrita através de portas. Esta independência possibilita a conexão de várias portas, e quando uma mensagem for enviada por uma porta de saída, todas as portas de entrada conectadas a ela irão receber esta mensagem. Nesse sentido não existe mais uma relação direta entre objetos, ou seja, quando um objeto envia uma mensagem, ele não sabe quem ou quantos objetos irão receber, o que característica um estilo produtor/consumidor [PER2001]. Além disso, a interação é normalmente realizada de forma assíncrona, não havendo a interrupção do fluxo de execução para realizar a transferência do controle para o sistema de invocação.

Apesar de existirem algumas diferenças, o modelo ROOM de portas é bastante similar ao modelo genérico de componentes [SZY99]. Ambos modelos possuem objetivos similares, como a não utilização de referências externas aos objetos, e a definição dos tipos de mensagens de cada porta (ou interface), o que define um protocolo de comunicação [SEL99].

2.4.3 Canal

Em um canal de comunicação a conexão ponto a ponto oferecida pelos modelos RMI e de Portas é substituída por uma conexão em forma de barramento, onde vários elementos distribuídos podem estar conectados a um mesmo canal [SCT95]. Quando um destes elementos distribuídos transmite uma mensagem, ela é recebida por todos os demais, sendo o canal responsável pelo gerenciamento destas transmissões. Obviamente, cada aplicação pode possuir um ou mais canais, e a implementação destes canais é facilitada quando implementada em um protocolo do tipo *Broadcast* ou *Multicast*. Neste modelo de comunicação não existe a chamada direta de um método, o mecanismo de recepção da mensagem percebe um identificador e o relaciona a um método interno de um objeto. Assim, é responsabilidade do receptor filtrar as mensagens que chegam pelo canal. Este modelo é flexível o suficiente para permitir que novos elementos sejam conectados ao canal durante a execução da aplicação, pois ele é implementado através de um sistema que interage entre os elementos distribuídos e o meio físico de ligação dos mesmos.

Apesar de possuir similaridades com o modelo de portas, existem diferenças fundamentais entre os modelos destes mecanismos. No modelo de portas a conexão entre as portas deve ser definida em tempo de projeto, e somente é possível realizá-la entre portas conjugadas. Quando um canal é implementado, qualquer objeto que queira receber ou enviar as mensagens definidas para aquele canal pode conectar-se a este. Além disso, depois do canal criado, a qualquer momento, em tempo de projeto ou execução, novos objetos podem ser conectados a este canal [PER2001].

O aspecto negativo do uso dos canais está na manutenção dos mesmos. Um canal exige gerenciamento constante, o que acarreta num processamento adicional. Além disso, se o protocolo de comunicação no qual o canal está baseado não suportar o envio de mensagens em *broadcast* ou *multcast*, a próprio canal deverá realizar esta função, o que certamente acarretará em retransmissão de mensagens e aumento do volume de processamento realizado pelo canal.

Assim como o modelo de portas pode-se implementar o modelo de comunicação dos componentes utilizando-se canais. Para isso basta associar um canal a cada interface de

componente. Comparada ao modelo de portas, essa forma de ligação facilita a montagem dos diagramas, bem como o gerenciamento das mensagens entre os objetos distribuídos.

2.5 Middleware de Comunicação

2.5.1 CORBA

O CORBA [FAY00, HAR98A, OMG00, VIN99] é um conjunto de especificações criadas pela OMG, com o objetivo de facilitar a comunicação entre objetos distribuídos em sistemas heterogêneos. Inicialmente pode-se pensar que ele se aplica apenas ao modelo de objetos, mas, na realidade, estas especificações não apresentam restrições a qualquer outro tipo de elemento distribuído. A arquitetura CORBA é formada por um *Object Request Broker* (ORB), responsável pela gerência das comunicações de forma transparente, e por elementos de serviços (voltados ao usuário final) e de facilidades (voltados aos elementos da aplicação). A OMG fornece as especificações de um conjunto de padrões de serviços e facilidades. A Figura 2.3 mostra o modelo completo da arquitetura CORBA.

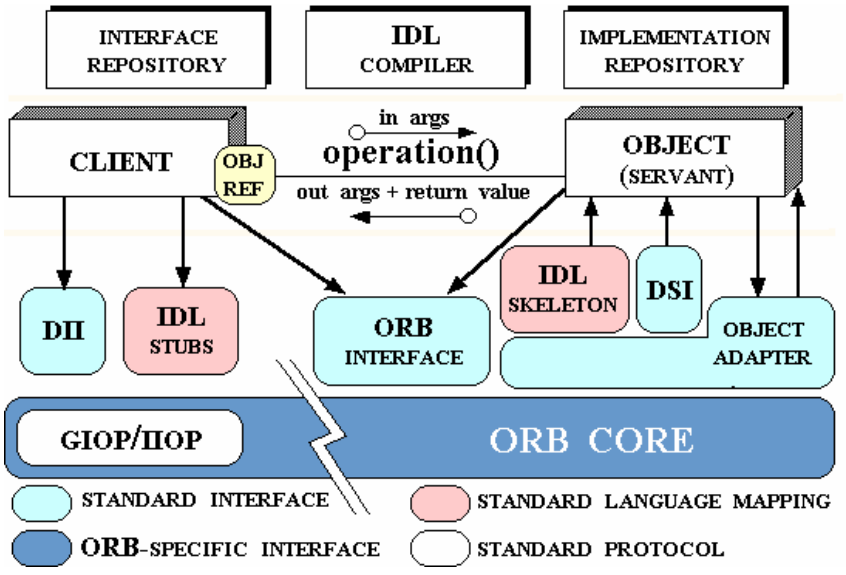

FIGURA 2.3 - Modelo de referência CORBA [OMG2000]

Para uma melhor compreensão como este *middleware* implementa a comunicação é necessário verificar os conceitos fundamentais de cada parte do modelo apresentado na Figura 2.3:

- *Object Implementation* (Servidor): Este elemento define as operações descritas pela interface IDL CORBA. Um objeto pode ser implementado por diversas linguagens de programação, tais como C, C++, Java, Smalltalk e Ada.

- *Client*: é a entidade que representa o programa que solicita um serviço (uma operação) de um Servidor. O acesso aos serviços do objeto remoto deve ser transparente, ou seja sua sintaxe deve ser similar àquela usada no caso não distribuído (obj->metodo(param)).

- *Object Request Broker*: O ORB provê um mecanismo para dar a transparência na comunicação entre o cliente e o servidor. O ORB simplifica a programação distribuída pela intermediação da chamada de serviços remotos. Quando um cliente chama uma operação, o ORB é responsável por achar o servidor, executar a chamada da operação e retornar, se necessário, a resposta ao cliente.

- Interface ORB: Um ORB é uma entidade lógica que pode ser implementada de várias maneiras, tais como um ou mais processos, ou um conjunto de bibliotecas. Para separar os detalhes de aplicação dos de implementação, a especificação CORBA define uma interface abstrata para os ORBs, com um conjunto de funções de uso geral.

- CORBA IDL *stubs* e *skeletons*: Estes dois elementos têm o objetivo de unir o cliente e o servidor através do ORB. A transformação das definições do CORBA IDL para a linguagem de programação da aplicação deve ser realizada por um compilador CORBA IDL específico.

- *Dynamic Invocation Interface* (DII): Esta interface permite que o cliente acesse diretamente o mecanismo de chamada de mensagens provido pelo ORB, sem a necessidade de um IDL *stub*. A diferença entre estas duas interfaces está nas formas possíveis de chamada. Quando a chamada de um método é realizada através de uma IDL, somente será permitida a chamada do tipo RPC. Já na DII é possível realizar chamadas do tipo RPC, síncrona e de único sentido (*send-only*).

- *Dynamic Skeleton Interface* (DSI): Está é a versão para o servidor, análoga da DII do cliente. A DSI permite que um ORB entregue requisições a um servidor que não saiba em tempo de compilação que tipo de objeto ele está implementando. O cliente que realiza a chamada não sabe se irá se comunicar com uma IDL *Skeleton* ou com uma DSI.

- *Object Adapter*: Ele auxilia o ORB com a entrega das requisições ao servidor e com a inicialização do servidor. Um *Object Adapter* associa um servidor a um ORB. Ele pode ser especializado para dar suporte para certos estilos de servidores, tais como OODB para suportar persistência e bibliotecas para clientes não remotos.

Para realizar uma chamada remota, o cliente deve primeiramente obter uma referência ao servidor. Após, é usado o mesmo código que serviria para realizar uma chamada local, apenas substituindo o objeto de referência pelo servidor remoto. Quando o ORB verifica que o objeto de referência é um objeto remoto, ele envia os argumentos e rotas de invocação pela rede até o ORB da máquina onde está executando o objeto remoto.

Esse processo foi padronizado pela OMG em dois níveis. No primeiro o cliente e o servidor sabem que tipo de objeto está sendo chamado através do *stub* do cliente e o *skeleton* do servidor, pois ambos são gerados a partir da mesma IDL. Isso quer dizer que o cliente sabe exatamente que operações podem ser executadas, quais são os parâmetros de entrada e onde ele deve buscar o serviço. No segundo, o ORB do cliente e o ORB do servidor devem concordar em se comunicar com um mesmo protocolo. Apesar da OMG oferecer um padrão bem definido chamado *Internet Inter ORB Protocol* (IIOP), pode ser utilizado qualquer outro protocolo seguindo o modelo mais geral chamado de *General Inter ORB Protocol* (GIOP).

2.5.2 COM e DCOM

Um objeto no modelo COM [HAR98A, MIC00, EDD98] é uma entidade funcional que obedece ao princípio de encapsulamento de orientação a objetos. Os clientes não manipulam os objetos diretamente, ao invés disso o objeto exporta para os seus clientes vários conjuntos de ponteiros para funções conhecidas como Interfaces. Cada objeto ou

interface possui um Identificador Global Único (GUID - *Globally Unique identifier*), que possibilita nomeação de objetos e interfaces livres de colisão. Todos os objetos COM devem suportar a interface mais básica chamada *Iunknown*. Esta interface suporta três métodos que fornecem a funcionalidade básica para todos os objetos, o *QueryInterface*, que permite que um cliente consulte quais interfaces um objeto suporta, e o *AddRef* e o *Release*, os quais gerenciam a contagem de referência para os objetos.

O COM define como os componentes de software e os seus clientes (outros componentes) interagem. Esta interação é definida de forma que o componente cliente e o servidor possam se conectar sem a necessidade de qualquer recurso adicional. Um cliente COM interage com um objeto COM adquirindo um ponteiro para uma das interfaces do objeto e chamando métodos através desse ponteiro, como se o objeto residisse no espaço de endereço do cliente. O COM especifica que qualquer interface deve seguir um layout de memória padrão, que é o mesmo que a tabela de funções virtuais do C++. Já que a especificação fica no nível binário, ela permite a integração de componentes binários possivelmente escritos em linguagens de programação diferentes tais como C++, Java e Visual Basic. A Figura 2.4 ilustra como as bibliotecas COM fornecem esta ligação entre cliente e componente.

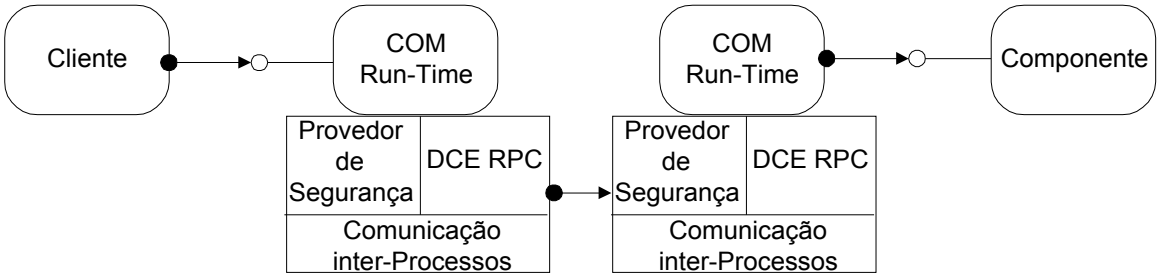

FIGURA 2.4 - Modelo COM [MIC2000]

O DCOM [HAR98B, MIC00, EDD98] é a extensão distribuída do COM que adota uma camada de chamada de procedimento de objetos remotos (*Object Remote Procedure Call* - ORPC). Quando o cliente e o componente residem em máquinas diferentes, o DCOM simplesmente substitui o mecanismo de comunicação inter-processo local por um protocolo de rede. Nem o cliente nem o servidor ficam sabendo do meio que os interconecta. A Figura 2.5 mostra a arquitetura DCOM de um modo geral.

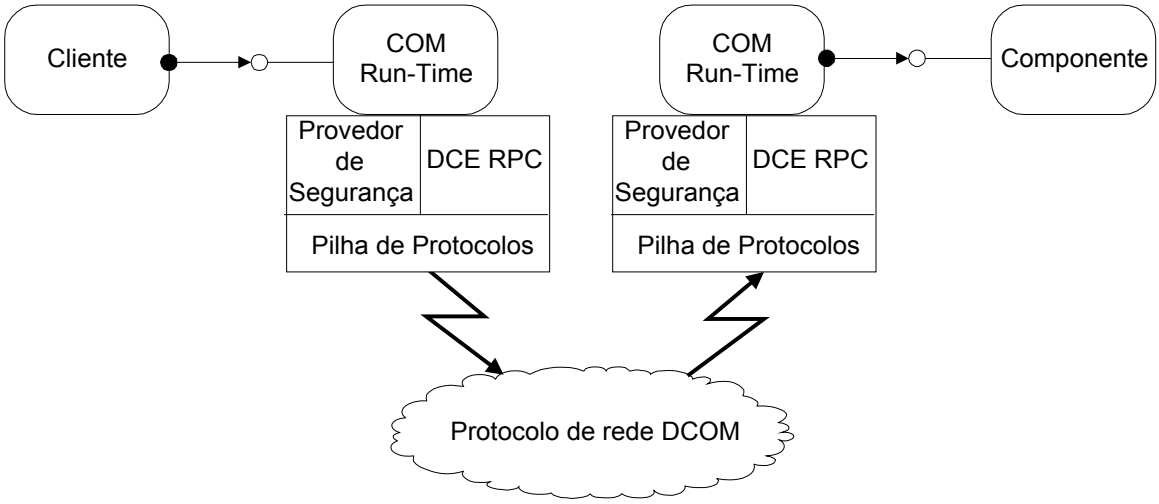

FIGURA 2.5 - Modelo DCOM [MIC2000]

O DCOM gerencia conexões tanto para componentes que são dedicados a um único cliente, bem como para componentes que são compartilhados por múltiplos clientes, mantendo uma contagem de referência em cada componente. Quando um cliente estabelece uma conexão com um componente, o DCOM incrementa o contador de referência do componente. Ao liberar a conexão, ele decrementa o contador. No momento que o contador alcança zero, o componente pode ser destruído.

Para a detecção dos clientes ativos o DCOM envia uma mensagem periódica através do protocolo *ping*. O DCOM considera uma conexão interrompida se mais do que três acessos de *ping* ocorrem sem que o componente receba uma mensagem. Se a conexão é quebrada ele decrementa a contagem de referência e libera o componente se a contagem alcançar zero. Se a rede se restabelece antes do intervalo de *timeout* definido, o DCOM restabelece as conexões automaticamente. Quando os clientes detectam a falha de um componente, eles podem tentar conectar-se novamente ao mesmo componente de referência que estabeleceu a primeira conexão. O componente de referência possui informações sobre quais servidores não estão mais disponíveis e provê o cliente automaticamente com uma nova instância do componente em execução em uma outra máquina. Neste caso, as aplicações poderão perder informações anteriores, bem como a sua consistência, devendo tratar isso em níveis superiores.

2.5.3 Java-RMI

Java [JAV2000] é uma linguagem de programação que oferece portabilidade e facilidades para a programação de bases WEB, criando um ambiente ideal para o desenvolvimento de aplicações para a Internet. Programas Java são compilados em um código binário (*bytecode*), que é executado sob uma *Java Virtual Machine* (JVM). Uma JVM é um pequeno ambiente de execução, portável para qualquer tipo de plataforma de software e hardware, que disponibiliza um conjunto de funcionalidades padronizado. Atualmente é possível encontrar JVMs, de forma nativa, na maioria das plataformas computacionais ou nos navegadores para a Internet. As JVMs interligam-se apenas através do protocolo de comunicação utilizado pela Internet, o TCP/IP.

O Java-RMI [JAV2001] permite que objetos Java possam invocar métodos de objetos executados em JVMs distintas, que normalmente estão localizadas em sistemas computacionais diferentes. Para isso o programador deve inicialmente definir as interfaces para cada objeto remoto. Uma Interface Java é equivalente a uma classe abstrata contendo apenas métodos abstratos. Como nas interfaces IDL CORBA, uma interface Java é definida de forma independente da sua aplicação, o que normalmente não é suportado pelas linguagens orientadas a objetos. Uma classe Java pode implementar nenhuma, uma ou várias interfaces, mas ela pode apenas estender uma simples superclasse.

Uma vantagem da utilização da interface Java em relação às IDL CORBA é que a primeira não impõe restrições de tipos aos elementos que são transferidos numa chamada remota. Isto é possível porque o Java-RMI pode utilizar qualquer classe Java tanto para parâmetros de entrada como para parâmetros de saída. O Java-RMI é o suporte a comunicação entre objetos distribuídos em Java, através de uma chamada remota de processos, entre diferentes JVMs, baseado no mecanismo RMI. A facilidade da programação de ambientes por RMI, proposta pelo Java, possibilita algumas vantagens adicionais. Por exemplo: é possível transmitir objetos completos como parâmetros de um método RMI, e receber uma resposta; é possível padronizar o estado de um objeto, e repassá-lo a outros objetos remotos; é possível transmitir o *bytecode* da classe do objeto.

Apesar desta diferença, o Java-RMI possui semelhanças ao modelo CORBA na base de sua arquitetura. Por exemplo, a camada de transporte é responsável pela transferência de baixo nível, a camada acima trata o mapeamento das referências locais a objetos remotos utilizados nos clientes, o *stub* permite o acesso do cliente às referências remotas e o esqueleto disponibiliza a interface ao servidor.

A última versão do Java-RMI possui novos serviços de negociação, transações e eventos [BEC2000]. O serviço de negociações permite que objetos negociem e estabeleçam contratos entre si para a utilização de recursos. Um contrato define um conjunto de regras que regem a comunicação entre dois objetos. Os tipos mais comuns de regras são as de acessibilidade ao objeto, solicitação de notificações de eventos, solicitação de armazenamento persistente ou solicitação da disponibilidade do objeto.

O conceito de transação adicionou ao Java-RMI características importantíssimas para o bom funcionamento de aplicações distribuídas. Transações são técnicas que forçam a consistência sobre um conjunto de operações. Quando uma destas operações não puder ser executada, ocasionada por algum tipo de erro, a transação é abortada e a aplicação retorna ao estado anterior à transação (*rollback*). Este processo garante a consistência dos dados após a transação.

Por fim, o serviço de eventos permite que os objetos registrem o seu interesse nas mudanças internas de estado de outros objetos de uma aplicação distribuída.

2.5.4 AO/C++

A linguagem de programação AO/C++ [PER94, BEC99] foi idealizada com o objetivo de unir os conceitos de orientação a objetos com as características de sistemas operacionais de tempo real, para possibilitar a produção de sistemas mais compactos e legíveis, e de manutenção facilitada. Esta linguagem implementa um conjunto de serviços semelhante àqueles oferecidos pelos *middleware* citados anteriormente. Tais serviços possibilitam a localização de um servidor remoto e o mapeamento de chamadas de funções locais para chamadas remotas.

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 e Linux[1]. 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 *parse*r. As classes ativas são criadas em dois arquivos separados, sendo um arquivo de cabeçalho (*.ph) e outro com a definição dos métodos (*.pc). Todas as classes ativas herdam as propriedades de uma classe básica, a classe tempo real. Essa classe possui uma referência a um processo UNIX-TR, onde a computação (métodos da classe) é executada, e é responsável por mapear as chamadas de funções feitas pelas classes C++ para mensagens.

Um construtor especial para as classes ativas tem a capacidade de gerar, 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, ou seja assemelham-se aos conceitos de "stubs" e "adapters" de CORBA. 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, que 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

[1] O Linux-TR não possui uma versão estável, capaz de suprir todas as necessidades das aplicações de teste. Por esse motivo tem se utilizado Linux e μClinux para a realização dos mesmos.

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 sistema operacional. No receptor, a mensagem é então decodificada e a função correspondente é ativada.

Para este mapeamento automático da interface é decisivo o tipo de retorno definido para a classe ativa. Se o valor de retorno for um '*void*' (linguagem C/C++) ou se não houver tipo definido, a troca de mensagens é mapeada assíncrona. Se houver um outro tipo definido qualquer, 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.

Apesar de ser utilizado o termo interface para a definição das funções que podem ser ativadas remotamente pela linguagem AO/C++, o seu significado é um pouco diferente do que define uma interface de um componente. No contexto do AO/C+ uma interface é um mascaramento da chamada de um método remoto de um objeto, baseada num mecanismo de comunicação do tipo RMI, através da utilização de um processo fornecido por um sistema operacional de tempo real. Neste sentido, a utilização de componentes como elementos de distribuição fica limitada a alguns tipos de aplicação, pois não é possível implementar o conceito completo de uma interface de componente (item 2.2.2.).

2.5.5 Produtor/Consumidor para Tempo Real

Kaiser e Moch [KAI99] apresentam uma nova proposta de *middleware* voltado para sistemas de automação industrial, envolvendo a comunicação entre microcontroladores e PC's. A coordenação entre objetos é baseada em compartilhamento de informações em vez de transferência de controle, motivando a utilização de um padrão de interação produtor/consumidor entre objetos distribuídos, em substituição ao modelo convencional cliente/servidor. O padrão produtor/consumidor não somente suporta autonomia de controle, mas também reflete as necessidades de sistemas de controle distribuídos com respeito à distribuição de informação. Enquanto o modelo cliente/servidor define uma relação um para um, o modelo produtor/consumidor facilmente implementa uma relação

um para muitos. Esse tipo de relação se adapta melhor ao tipo de comunicação imposto por sensores e atuadores inteligentes.

Este modelo de comunicação está baseado em canais de eventos (vide item 2.3.3), onde o consumidor assina certo canal de comunicação que representa um certo tipo de informação, em vez de um produtor específico. Um produtor, por sua vez, publica instâncias deste tipo de informações no respectivo canal, e todos os consumidores inscritos para aquele canal receberão a mesma mensagem.

A comunicação no modelo de componentes pode ser descrita pela configuração de um canal para cada interface de componente, e um evento para cada mensagem enviada ou recebida por está interface. Já para realizar a descrição do modelo de objetos, pode-se utilizar canais específicos para implementar os relacionamentos entre objetos remotos, onde cada relacionamento é representado por um canal, e cada chamada de método, por uma mensagem específica do canal. Nesta representação, apesar de estar baseada em canais, toda a comunicação se comportará como no mecanismo RMI.

Todo o controle dos canais de comunicação é mantido por dois tipos de elementos o *Event Channel Handler* (ECH) e o *Event Channel Broker* (ECB). O ECB, que é instanciado em cada barramento que compõe a aplicação, tem o objetivo de gerenciar em tempo de execução a configuração dos nós (de forma independente do hardware), a criação dos canais de comunicação e relacionamento dos consumidores com cada canal. Neste sentido ele destina um identificador diferenciado para cada canal e nodo da aplicação. Já o ECH, que é instanciado em cada nó que compõe a aplicação, tem a responsabilidade de filtrar as mensagens que chegam, criando um vínculo dinâmico entre as mensagens e os eventos cadastrados no nodo, e de transmitir as mensagens dos objetos do nodo.

Para ser inicializada uma aplicação deve-se em primeiro lugar dar o suporte ao *middleware* de comunicação, ou seja, devem ser instanciados os ECBs, os ECHs e por fim todos os demais objeto da aplicação (seguindo a ordem descrita pela necessidade da própria aplicação). Para poder receber as mensagens, cada objeto deve subscrever a quantos canais quiser, em qualquer instante da execução da aplicação. Apesar disso o mais comum é que

os objetos subscrevam os canais no momento da sua criação. A arquitetura completa deste modelo é mostrada na Figura 2.6.

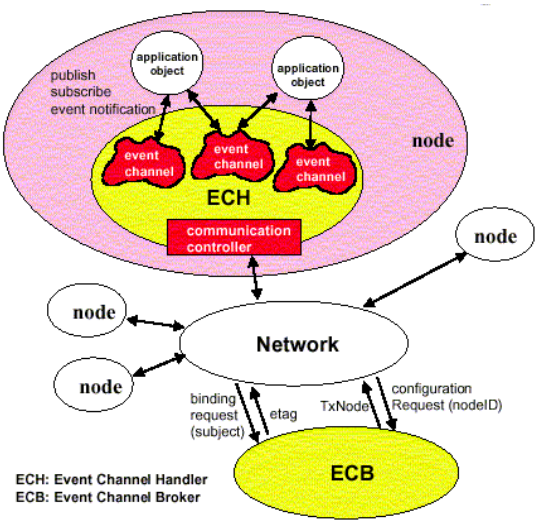

FIGURA 2.6 - Modelo Publisher/Subscriber Kaiser

Quando um ECH é instanciado ele deve cadastrar o seu nó no ECB, para receber um identificador do nó para aquele barramento. Quando um objeto se subscreve em um canal, ele envia essa requisição ao ECH que transmite ao barramento, onde um ECB deve estar conectado. O ECB verifica a requisição, e se o canal referenciado não existir, ele gera um novo identificador, cadastrando o novo canal juntamente com o seu consumidor relacionado. No final da operação ele retorna ao barramento a confirmação com o identificador do canal. Se o canal já existir ele apenas cadastra o novo consumidor relacionado e transmite a confirmação. Quando o ECH recebe a confirmação da existência do canal, ele cadastra uma nova relação local que servirá de base para os algoritmos de filtragem das mensagens vindas pelo barramento.

Apesar de ser uma forma simples e eficiente de implementação de um *middleware* de comunicação, este mecanismo está fortemente relacionado às características do protocolo de comunicação. Ou seja, certamente é mais simples implementá-lo com um protocolo baseado em mensagens do tipo *Broadcast* ou *Multicast* (como por exemplo, o *Control Area Network* (CAN)), do que em um protocolo baseado em mensagens endereçadas (como por exemplo, o protocolo TCP/IP). Apesar deste fato, Kaiser e Moch

implementaram este modelo neste dois protocolos. Um outro exemplo da utilização desta proposta, com detalhes de implementação no protocolo CAN, foi realizado por Brudna [BRU2000] durante sua dissertação de mestrado no CPGEE da UFRGS.

2.6 Resumo e Considerações

Segundo o descrito no início deste capítulo, sistemas de automação industrial possuem características que os classificam como sistemas distribuídos heterogêneos. Isto significa que eles são descritos a partir de entidades de software e hardware distintas, e que de alguma forma coordenam as suas tarefas em benefício da execução de uma aplicação. Neste sentido, este capítulo iniciou realizando um levantamento básico dos conceitos que dão suporte a este tipo de aplicação. O ponto de partida deste estudo foram os conceitos de sistemas distribuídos, seguido pelo levantamento de duas das principais metodologias de modelagem de sistemas distribuídos (orientação a objetos e a de componentes de software), dos principais mecanismos de comunicação passíveis de implementação nestes modelos (RMI, Portas e Canais) e de alguns dos principais *middlewares* de comunicação existentes (CORBA,COM, Java-RMI, AO/C++ e P/S).

Em termos de um modelo ideal para a definição de aplicações distribuídas e heterogêneas, uma análise complementar pode ser realizada sobre os aspectos da forma de acesso às entidades da aplicação. A possibilidade de permitir a modelagem em níveis diferenciados de acesso a estas entidades e a de tratar o relacionamento entre elas de forma não explicita, é sem sombra de dúvida uma forma de simplificar este modelo.

No modelo de objetos, uma classe define um conjunto de propriedades e métodos, que possuem as mesmas características de acessibilidade, para qualquer instância desta classe. Isso significa que um objeto disponibilizará os mesmos recursos a todos os demais objetos que estiverem relacionados a ele, não importando questões como que tipo de informação o objeto cliente quer ou tem permissão de acessar. É claro que de certa forma esta questão pode ser contornada utilizando-se classes abstratas, com graus de visão diferente de um objeto, sendo utilizadas como ponteiros para estes objetos. Uma outra restrição deste modelo, refere-se ao fato dele estar baseado no mecanismo de comunicação RMI, o que obriga que o objeto cliente conheça o nome e a localização do objeto servidor.

Em algumas linguagens, como Java-RMI, esta obrigação é repassada para mecanismos auxiliares, facilitando a sua utilização, no caso do Java-RMI as máquinas virtuais Java (JVMs) realizam esta tarefa.

No modelo de componentes as interfaces definem o grau de acessibilidade ao componente. Isto significa que cada componente cliente poderá ter uma visão diferente do componente servidor, e que uma mesma visão pode ser compartilhada por muitos clientes. Além disso, as interfaces criam um mecanismo único de acesso aos componentes, possibilitando que nem o cliente ou o servidor conheçam exatamente qual, ou quais, os componentes que estão relacionados a eles.

Além destas constatações é possível retirar alguns pontos de comparação que são de fundamental importância para definição da forma de construção dos sistemas distribuídos. Na orientação a objetos o elemento básico de construção de modelos é a classe, sendo que estas podem ser formadas através de relações com outras classes, tais como associações ou agregações. A chamada de métodos de classes distribuídas deve ser realizada por instâncias da classe através de RMI. Já no modelo de componentes, o elemento básico de construção de modelos é o próprio componente. Ele pode ser formado por classes, objetos, procedimentos ou pela composição de outros componentes. Os componentes se relacionam através de interfaces, as quais podem implementar um mecanismo similar ao de portas de comunicação ou canais de eventos.

Em termos de modelagem pode-se perceber que é possível utilizar-se qualquer uma das metodologias apresentadas, mas percebe-se também que o modelo de componentes aproxima-se mais do modelo proposto, sendo por isso mais indicado a utilização. Além disso, características como o de possibilidade de reutilização destas entidades, são mais naturais no modelo de componentes. Isso se deve ao fato de que um componente não possui, por definição, dependências de contexto internas. Se houver necessidade, esta dependência deve ser repassada através de uma interface.

O próximo capítulo trata do projeto conceitual da ferramenta baseada em componentes. Nele são discutidos os conceitos básicos que regem esta ferramenta, tendo como base as tecnologias apresentadas neste capítulo.

3 Projeto Conceitual

3.1 Introdução

Edward Lee [LEE2000] define que uma ferramenta CASE voltada para auxiliar o desenvolvimento de aplicações complexas deve possuir quatro tipos de categorias de serviços básicos: *Ontologia* onde é definido o que é um componente (ex: sub-rotina, objeto, processo ou etc.); *Epistemologia* onde são definidos os estados de conhecimento de cada componente e da aplicação, ou seja, o que cada um sabe sobre o outro; *Protocolos* onde são definidos os mecanismos de interação entre os componentes (ex: mensagens assíncronas, semáforos, monitores, eventos temporais, transferência seqüencial de controle ou etc.); e *Léxico* onde é definido o vocabulário de interação entre os componentes (ex.: IDL CORBA).

De forma resumida, uma ferramenta que pretenda descrever aplicações baseadas no modelo de componentes deve possibilitar a compreensão do que é um componente, como ele interage com os demais, como é possível reutilizar um componente e se possível auxiliar na geração de código do mesmo. Além disso, essa ferramenta deve possuir diagramas que estejam de acordo com os padrões internacionais, possibilitando a compreensão dos mesmos por um conjunto maior de pessoas.

Até 1995 existiam diversas metodologias baseadas nos conceitos da orientação a objetos. Destas, a metodologias de Booch [BOO94] e OMT (*Object Modeling Technique*) [RUM91] se destacaram pelo reconhecimento mundial obtido, tanto no setor acadêmico como no setor empresarial. A partir deste ano os autores destas metodologias, respectivamente Grady Booch e James Rumbaugh, juntaram seus esforços e em outubro de 1995 lançaram o primeiro rascunho do método unificado. Nesta mesma data Ivar Jacobson juntou-se a eles fundindo o método OOSE (Object-Oriented Software Engineering) [JAC92]. A principal motivação desta união de esforços era a de criar uma linguagem de modelagem unificada que tratasse assuntos de escala inerente a sistemas complexos e de

missão critica, que se tornasse poderosa o suficiente para modelar qualquer tipo de aplicação de tempo real, cliente/servidor ou outros tipos de software padrão.

A partir deste trabalho muitas empresas reconheceram a importância estratégica desta padronização e a UML (*Unified Modeling Language*) [RUM99] como fora batizada posteriormente, recebeu o apoio de parceiros importantes como a Microsoft, Hewlett-Packard, Oracle, IBM,e muitos outros mais. Essa colaboração produziu em janeiro de 1997 a UML 1.0 e em setembro do mesmo ano a UML 1.1. Em novembro de 1997 a OMG (*Object Management Group*) [OMG2000] aprova a UML colocando um ponto final na disputa por um padrão sobre o modelo de objetos.

O objetivo fundamental da UML é o desenvolver uma linguagem padronizada para a especificação, visualização, documentação e construção de artefatos de um sistema. É importante salientar que apesar de possuir estes objetivos, a UML não é uma metodologia. As metodologias consistem, pelo menos em princípio, de uma linguagem de modelagem e um procedimento de uso desta linguagem. Na UML não é descrito explicitamente esse procedimento. Em muitas formas, a linguagem de modelagem composta por sintaxe e semântica é a parte mais importante do método. Apesar de ser valiosa a existência de uma linguagem de modelagem padrão, não há uma necessidade comparável a de uma metodologia padrão, muito embora alguma harmonização no vocabulário permaneça desejável. Metodologias formais conduzem freqüentemente a um investimento desnecessário de tempo em detalhes periféricos, sendo normalmente difíceis de entender, manipular e de se relacionar com uma linguagem de programação. A UML pode ser utilizada para:

- Mostrar as fronteiras de um sistema e suas funções principais utilizando atores e estudos de caso;
- Ilustrar a realização de estudos de casos com diagramas de interação;
- Representar uma estrutura estática de um sistema utilizando diagramas de classe;
- Modelar o comportamento de objetos com diagramas de transação de estados;

- Revelar a arquitetura de implementação física com diagramas de comportamento e implantação;

- Estender sua funcionalidade através de estereótipos.

A UML vai além de uma simples padronização em busca de uma notação unificada, uma vez que contém conceitos novos que são encontrados em outros métodos orientados a objetos. Em seu estágio atual, a UML define uma notação e um metamodelo. A notação é o material gráfico visto nos modelos, o metamodelo é a sintaxe da linguagem de modelagem. Por exemplo, com base nestas definições foram realizados estudos para a ampliação das possibilidades de modelagem com a UML para modelar sistemas de tempo real [DOU99] e para a modelar a partir de componentes [RUM99].

Na próxima seção são definidos os diagramas da UML que devem ser utilizados para a descrição das características propostas por Edward Lee. Na seção seguinte é definida uma biblioteca de componentes para automação industrial. Na última seção é apresentado um resumo do capítulo com algumas observações sobre os diagramas e a biblioteca que formam a ferramenta proposta.

3.2 Definição dos diagramas a serem utilizados na ferramenta proposta

3.2.1 Diagrama de Componentes

Este diagrama é fundamental para a descrição do modelo de componentes, Nele é definida a ontologia de um componente, ou seja, o seu conceito, características e formas de utilização. Como já foi descrito no item 2.2, um componente pode ser criado pela composição de outros componentes, ou pela utilização de objetos e procedimentos, juntamente com a definição individual de cada uma de suas interfaces. A UML possui um diagrama chamado *Component Diagram*, que possibilita a descrição completa de um componente.

Como pode ser visto na figura 3.1, neste diagrama são mostrados além dos elementos que formam o componente, todas as relações estáticas formadas por estes elementos. Estas relações são como dependências de compilação entre programas, e são

mostradas como setas tracejadas que vão do elemento cliente até o elemento servidor. Como elementos para a formação de um componente pode-se utilizar outros componentes, classe, objetos ou procedimentos (bibliotecas).

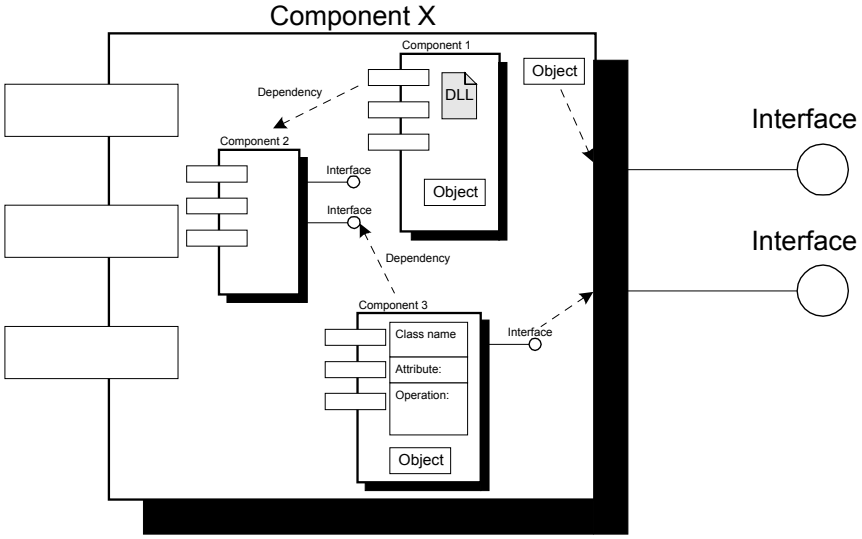

FIGURA 3.1 - UML Component Diagram

3.2.2 Diagrama de Distribuição

Um diagrama de distribuição define a forma como partes de uma mesma aplicação, sendo executadas em elementos computacionais distintos, conseguem interagir entre si. No caso de componentes, este diagrama define estes mecanismos de interação entre os componentes (*Protocolos*) e a forma de cada sistema computacional que dá o suporte a distribuição da aplicação. Por exemplo, neste diagrama são definidos o tipo de hardware e software dos sistemas computacionais que implementam a aplicação, a forma como estes sistemas estão interligados e onde será executado cada componente.

O *Deployment Diagram* definido pela UML (vide figura 3.2), possibilita esta descrição a partir de três elementos fundamentais:

- *Node*: representa um sistema computacional com um hardware específico (Ex.: PC ou microcontrolador) e um sistema operacional de base (Ex.: QNX ou µClinux);

- *Link*: representa o meio de ligação entre dois *Nodes*. No caso de sistemas de automação industrial, esta ligação irá representar um barramento ou rede industrial, com um protocolo específico (Ex: CAN, Profibus ou TCP/IP);

- Dependência: representa as dependências de contexto dos componentes da aplicação, ou seja, identifica quais são os relacionamentos entre os componentes. As dependências são representadas por linha tracejada, com indicação do sentido na extremidade do servidor.

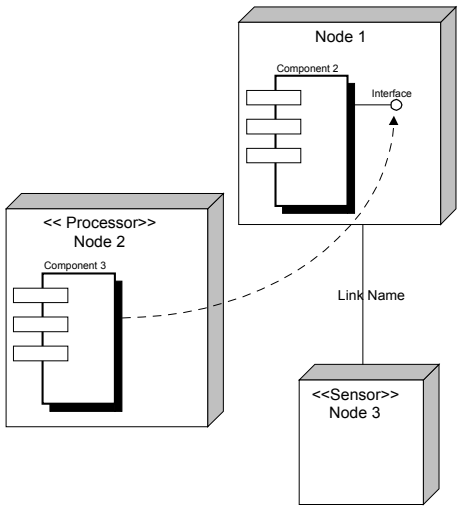

FIGURA 3.2 - UML Deployment Diagram

3.2.3 Diagramas de Comportamento

Alem do diagrama de componentes, que define a ontologia, o diagrama de distribuição, que define os *Protocolos*, é necessário definir o comportamento (epistemologia) destes componentes, diante de seus relacionamentos. Existem várias formas possíveis de descrever um comportamento. A UML define alguns diagramas com esta finalidade, tais como o Diagrama de Seqüência (*Sequence Diagram)* e o Diagrama de Estados (baseado na notação *Statechart* [HAR87]).

O Diagrama de Seqüência é utilizado para descrever o comportamento da aplicação em relação ao tempo. Este diagrama é especialmente importante em se tratando de aplicações em automação industrial, ou que, de alguma forma, possuam características

temporais mais rígidas. Os principais elementos deste diagrama estão relacionados à relação temporal entre a troca de mensagens entre as suas interfaces. A Figura 3.3 mostra este diagrama com alguns destes elementos.

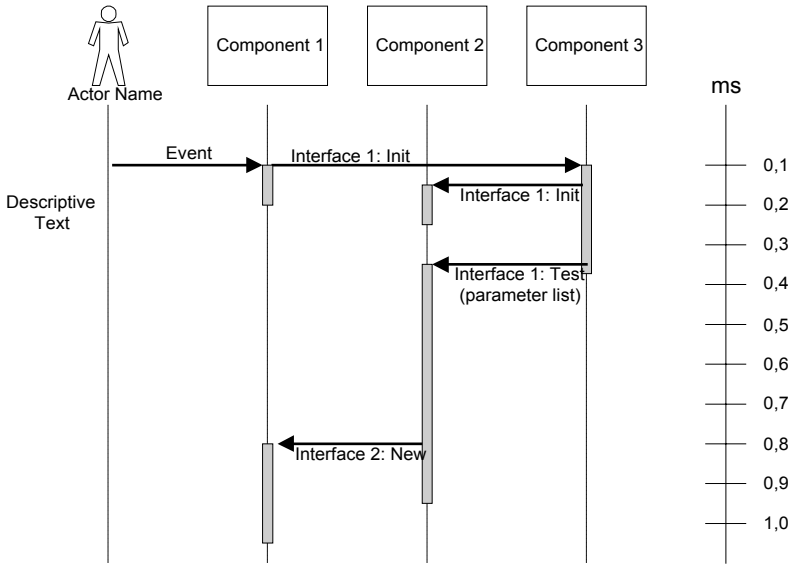

FIGURA 3.3 - Diagrama de Seqüência (*Sequence Diagram*)

Já o Diagrama de Estados, mostrado na figura 3.4, descreve a aplicação a partir de estados e transições. Nele uma aplicação, ou parte dela, pode ser descrita objetivando-se um detalhamento de um conjunto de características específicas que formam um estado, e o conjunto de transações ou eventos que alteram este estado.

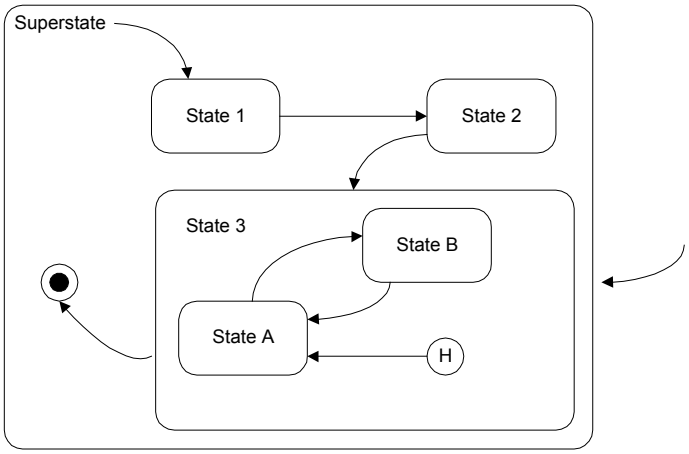

FIGURA 3.4 - Diagrama de Estados (*Statechart)*

3.3 Proposta de Biblioteca de Componentes para Automação Industrial

A independência de contexto, e a decorrente possibilidade de reuso de partes de aplicações de forma simplificada é um dos principais argumentos dos defensores das tecnologias baseadas em componentes. Como mencionado anteriormente (seção 2.3.2), isso implica na necessidade de criação de um repositório ou biblioteca, e consequentemente de uma ferramenta auxiliar que permita o armazenamento e a busca de componentes ou tipos de componentes.

Para possibilitar um melhor reaproveitamento de código, o ideal é que esta biblioteca deva seguir um modelo preestabelecido conforme as necessidades do tipo de aplicação alvo. Isso significa definir uma estrutura inicial e a forma do crescimento desta biblioteca. Por exemplo, para aplicações em um modo geral são indispensáveis as definições de interfaces com o usuário (Ex: *display* e teclado). Já para aplicações voltadas para sistemas de automação industrial, elementos como sensores, atuadores e controladores são igualmente indispensáveis. Neste sentido uma biblioteca com pretensões de auxiliar o desenvolvimento de aplicações em automação industrial devem definir a forma e os relacionamentos destes componentes básicos. Além disso, esta forma e estes relacionamentos devem ser simples de implementar, para possibilitar a construção de modelos de aplicações igualmente simples.

A questão principal tratada em sistemas de automação industrial envolve os problemas de controle de um ou mais dispositivos. Nesta situação este sistema de controle capta informações do ambiente da aplicação, processa algum algoritmo pré-definido e atua novamente sobre este ambiente. Esta forma de relacionamento é uma constante em sistemas de automação industrial, modificando-se apenas a forma de captação das informações, o algoritmo de controle e a forma de atuação sobre o ambiente da aplicação. Além disso, cada algoritmo de controle pode captar uma ou mais informações e pode atuar sobre um ou mais aspectos deste mesmo ambiente. Assim como cada informação captada pode ser utilizada por mais de um algoritmo e cada sistema de atuação pode ser controlado por mais de um algoritmo.

A construção da biblioteca deve iniciar a partir destes três elementos básicos o algoritmo (ou controlador), o sistema de captação de informação (ou sensor) e o sistema de atuação no ambiente (ou atuador). Sendo que um controlador pode estar relacionado a um ou vários sensores e atuadores, e cada sensor ou atuador pode estar relacionado igualmente a um ou vários controladores.

Obviamente que, como qualquer sistema, os sistemas de automação industrial necessitam da definição das formas de interação com o usuário. Ou seja, como as informações serão lidas dos dispositivos de entrada (Ex. Teclado) ou escritas nos dispositivos de saída (Ex. *Display*). Normalmente o maior problema está relacionado ao grande volume de informações captadas, e/ou a necessidade de visualização instantânea destes dados. Isto significa que os dispositivos de saída necessitam um relacionamento direto com os dispositivos de captação das informações. Além disso, um dispositivo de entrada de dados também deve possuir um relacionamento com um ou mais algoritmos de controle, para possibilitar alterações na forma como estes atuam no ambiente da aplicação. Preferencialmente este relacionamento deve ser realizado por um conjunto de regras que é independente da forma de atuação do dispositivo interface com o usuário. A Figura 3.5 mostra a proposta com os 5 tipos básicos de componentes para sistemas de automação industrial.

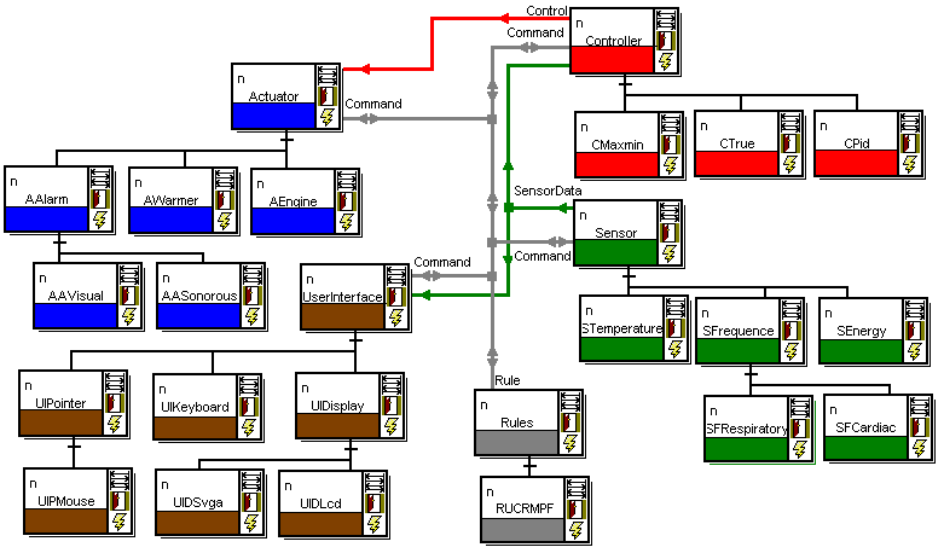

FIGURA 3.5 - Elementos básicos para a biblioteca de componentes

- **Sensor:** Representa um sensor de qualquer tipo, onde um sinal é adquirido ou amostrado, e convertido de um valor bruto (conversor AD), para um valor de engenharia (Ex.: temperatura). Para isso são implementadas as interfaces de comunicação com os componentes do tipo *Controllers* e *UserInterface*, onde são publicados os valores da temperatura (Interface do tipo *SensorData*), com os componentes do tipo Rules, onde são recebidas as configurações do sistema de conversão de valor bruto para valor de engenharia e são enviadas as mensagens de erro (Interface do tipo *Command*). A partir deste componente são construídas algumas especializações para sensores de temperatura, de falta de energia (booleano), e de freqüência. O sensor de freqüência é especializado novamente em sensor de freqüência cardíaca e em freqüência respiratória. Estas especializações diferenciam-se apenas na forma de condicionamento do sinal e transformação do valor de bruto no valor de engenharia;

- **Controller:** Representa um controle de qualquer tipo, onde a partir das informações de um ou mais sensores, e/ou de interferências externas definidas pelo usuário, mensagens de controle são geradas. Para isso são implementadas as interfaces com os componentes do tipo *Sensor*, para o recebimento dos dados dos sensores (interface do tipo *SensorData)*, componentes do tipo *Actuator*, para a transmissão das mensagens de controle (interface do tipo *Control*), e componentes do tipo *Rules*, para o recebimento de os comandos externos definidos pelo usuário e a transmissão de mensagens de erro (interface do tipo *command*). A partir deste componente são construídas algumas especializações para controle do tipo PID (CPID), máximo e mínimo (CMaxmin), e booleano (CTrue). A única diferença entre as especializações destes componentes está na inicialização e no algoritmo de controle;

- **Actuator:** Representa um atuador de qualquer tipo, que acionará um dispositivo físico, conforme o comando de acionamento recebido. Para isso são implementadas as interfaces com os componentes do tipo *Controller*, onde são recebidas as diretivas de controle (Interface do tipo *Control*) e componentes do tipo *Rules*, onde são recebidas as informações de configuração dos dispositivos e são enviadas as mensagens de erro (Interface do tipo *Command*). A partir deste componente são

construídas algumas especializações de atuadores do tipo máquina ou motor (AEngine), aquecedor (AWarmth) e alarmes (AAlarm) do tipo visual (AAVisual) ou sonoro (AASonoro);

- **UserInterface:** Representa uma interface com o usuário, que pode ser de entrada, como um teclado, ou de saída como um display. Para isso são implementadas as interfaces com os componentes do tipo *Sensor*, onde são recebidas as informações dos sensores (Interface do tipo *SensorData*) e componentes do tipo *Rules*, onde onde são recebidas as informações de configuração dos dispositivos e estado dos demais componentes, e são transmitidas as informações de saída dos dispositivos (ex.: teclado) e de erro (Interface do tipo *Command*). A partir deste componente são construídas algumas especializações de interfaces de entrada como teclados (UIKeyboard), de mouse (UIPointer, UIPMouse), ou interfaces de saída como *display* (UIDSvga, UIDLcd);

- **Rules:** Representa um gerente da aplicação, contendo um conjunto de regras para inicialização e restauração da aplicação, bem como o gerenciamento de erros dos componentes. Para isso é implementada a interface de comunicação *Command* que é utilizada por todos os demais componentes da biblioteca.

Neste conjunto básico de elementos são definidas as regras básicas de construção e de comunicação de componentes como sensores, atuadores, controladores, interfaces com o usuário e regas das aplicações. Isto significa que se um novo sensor ou atuador for construído, ele deverá ser implementado a partir do componente básico do seu tipo, ou a partir de um componente derivado do mesmo. Desta forma fica facilitada a construção de novos componentes ou aplicações.

Cada um destes componentes possui um conjunto de funcionalidades que é independente da aplicação. A definição das regras da aplicação deve ser construída a partir de um componente *Rules*. Neste componente devem ser definidas regras especiais como o que fazer quando ocorrer algum erro específico.

3.4 Middleware de Comunicação e Geração de Código

Na seção 2.4 é apresentado um estudo sobre três dos principais mecanismos de comunicação. Conforme descrito anteriormente, o RMI normalmente relaciona a transferência de dados com a de controle, violando princípios básicos da autonomia de controle para sistemas distribuídos. Além disso, quando se utiliza RMI é necessário conhecer a localização do objeto servidor. Deste modo, o programador não pode desenvolver objetos de forma independente. Como resultado, qualquer mudança nas relações de comunicação poderá afetar um número de objetos, o que poderá dificultar a localização destes.

No mecanismo de comunicação baseado em portas, uma porta simplesmente define mensagens de entrada e saída para um objeto. Deste modo, um projetista de um objeto não necessita saber quais outros objetos estão envolvidos na comunicação, como no caso do RMI. Uma característica importante suportada por este mecanismo é a do anonimato, ou seja, quem envia uma mensagem não necessita conhecer quem está recebendo a mesma. Da mesma forma que receber uma informação não necessita, obrigatoriamente, conhecer quem a gerou, o que facilita a modularidade e as alterações no sistema. Isso implica que não há a transferência de controle da aplicação, apenas os dados é que são transferidos. É possível o estabelecimento de comunicações entre objetos através de portas. Para isso deve ser adicionada uma ligação a mais entre eles. Esta ligação irá suportar os conceitos de modularidade, extensibilidade e modificabilidade, para que a comunicação entre estes objetos mantenha este mecanismo de comunicação.

Apesar das portas possuírem uma definição similar à da comunicação implementada por componentes, as configurações de portas podem apenas ser realizadas estaticamente. Isto quer disser que em tempo de execução portas não podem ser incluídas, excluídas ou modificadas. Contudo, às vezes é necessário um sistema de controle sem interrupção, o qual requisita de forma *on-line* modificações no sistema [OKI93]. O modelo de canais de eventos foi desenvolvido como alternativa a esta questão. De forma geral ele se baseia em uma comunicação do tipo *broadcast*, onde uma mensagem é transmitida para todos os elementos distribuídos da aplicação, que examinam e decidem o que fazer com a

mensagem. Os canais deste mecanismo evitam qualquer tipo de ligação fixa, deixando todo este serviço para os sistemas de filtragem de mensagens. Isto é possível porque ele converte uma mensagem em um identificador para a rede, deixando o canal de eventos muito parecido com uma porta. As diferenças vão estar nas ligações dinâmicas que ocorrem em tempo de execução, onde o objeto necessita estabelecer as suas ligações antes de executar as suas funcionalidades, e na não obrigatoriedade da especificação das ligações em tempo de projeto ou configuração. É claro que, para possibilitar estas ligações dinâmicas, serão necessárias modificações na estrutura da rede [PER2001].

No nível de aplicação é necessário definir um conjunto maior de regras de comunicação, que dizem respeito à identificação dos componentes da comunicação, sincronização entre comunicações e manutenção de características especiais durante a comunicação (Ex.: Temporais). Na seção 2.5 estão descritos 5 *middlewares* de comunicação, onde procurou-se vislumbrar os conceitos de modelos amplamente difundidos e também de propostas alternativas. O objetivo deste estudo foi relacionar estas propostas com as idéias de entidades de distribuição e com os mecanismos de comunicação. Para isto, pode-se tomar como base quatro questões básicas:

- **Elementos de distribuição utilizados:** A maioria destes modelos consegue trabalhar tanto com objetos como com componentes. No CORBA, por exemplo, todo o relacionamento dos elementos distribuídos é realizado através de uma IDL, não importando se o que estiver atrás desta IDL seja um componente ou um objeto. Já no caso do Java, o suporte a componentes é realizado através da adição dos conceitos JavaBens. Neste caso apenas a estrutura dos elementos é alterada, o restante do mecanismo é todo mantido. No caso do COM, toda a chamada remota é mapeada para um processo do sistema operacional, não importando qual o tipo do elemento distribuído. Da forma similar, o P/S mapeia cada chamada remota para uma mensagem de um canal específico. O único modelo que ainda não suporta os conceitos de componentes é o AO/C++.

- **Formas de definição das interfaces:** Tanto o COM como o CORBA descrevem estes relacionamentos através de uma IDL. De forma similar, o AO/C++ utiliza arquivos de cabeçalho (*.h), que são utilizados tanto pelo cliente como pelo

servidor. O Java utiliza para isso uma interface descrita na própria linguagem de programação e o P/S utiliza os *subscribes* dos eventos para definir estes relacionamentos.

- **Mecanismos de comunicação utilizados:** Tanto Java como o AO/C++ implementam apenas o mecanismo RMI. Já o P/S implementa o conceito de canal de eventos. Em termos de modelo o COM possibilita a utilização de qualquer um dos mecanismos, mas este mecanismo é sempre mapeado para uma chamada de processo do sistema operacional. O único capaz de implementar realmente qualquer um dos mecanismos é o CORBA.

- **Suporte a comunicação:** Todos estes *middlewares* possuem sistemas auxiliares para dar suporte a um ou mais mecanismos de comunicação. Por exemplo, o CORBA utiliza o ORB, o DCOM utiliza o ORPC, o Java-RMI utiliza a JVM, o P/S utiliza o ECH e o ECB, e o AO/C++ utiliza diretamente o suporte do SO.

Tendo em vista que o objetivo desta dissertação é o do estudo de uma proposta baseada em componentes a serem utilizados em sistemas microcontrolados, pode-se observar características que eliminam alguns dos modelos propostos. Por exemplo, a utilização de um mecanismo com a transferência tanto de dados como do fluxo de controle. Um exemplo disso é o AO/C++, que apesar de permitir a modelagem sem a transferência de controle, igualmente permite a modelagem baseada na transferência do controle, o que poderia acarretar um modelo de componente inconsistente.

Um outro exemplo é o dos *middlewares* como o COM/DCOM, que apesar de sua implementação no sistema operacional tempo-real VxWorks, não possibilita o controle destas características temporais. Por fim, apesar da existência de *middlewares* como Java-RMI com suas máquinas virtuais Java (JVMs) para algumas plataformas de microcontroladores, como a da *iButton* [IBU2001] ou ainda CORBA com projetos como o da *Highlander Engineering Inc* [HIG2001], estes necessitam de uma grande capacidade de processamento e memória, não disponível na maioria das configurações de microcontroladores utilizados pela industria na automação.

O *middleware* P/S proposto por Kaiser e Mock, além de possuir todas as características desejadas para este trabalho, possui um conjunto de serviços, que apesar de ser mínimo, é suficiente para atingir-se os objetivos propostos nesta dissertação. A definição do *middleware* de comunicação descreve finalmente o vocabulário *Léxico* de interações entre os componentes.

3.4.1 Geração de código Automática

Com a definição da forma da comunicação entre os componentes, juntamente com o modelo e informações contidas em cada um dos diagramas da proposta, pode-se então definir a forma que o código fonte deverá ter para implementar todos estes conceitos.

Inicialmente deve se ter bem claro que cada componente é uma parte de execução independente da aplicação, não sendo passível à distribuição. Quando o componente for formado por uma composição de outros componentes, todos os componentes que formam esta composição devem estar em um mesmo nodo, sob o mesmo fluxo de execução. De forma mais simplificada pode-se dizer que cada componente é um programa sendo executado de forma independente.

Para implementar a proposta P/S devem ser criados os componentes auxiliares ECH e ECB. O ECH é responsável pela sincronização das mensagens dentro de um nodo, sendo por isso necessária a existência de um em cada nodo. Já o ECB é responsável pela definição dos identificadores de mensagens que serão utilizadas pela aplicação e por um barramento específico, sendo por isso necessário um para cada barramento.

3.5 Resumo e Considerações

Neste capítulo estão descritos os conceitos e diagramas de um *Framework* para componentes. O ponto de partida são os conceitos de ontologia, epistemologia, protocolos e léxico, definidos por Edward Lee, e os diagramas da UML. A partir disto foram apresentados alguns dos diagramas da UML, sendo estes relacionados com os conceitos definidos por Lee. O diagrama de componentes descreve a ontologia dos componentes, o diagrama de aplicação descreve os protocolos de comunicação e o ambiente de execução

dos componentes, os diagramas de seqüência e de estados definem a epistemologia dos componentes, e finalmente a definição do *middleware* de comunicação possibilita a descrição do vocabulário léxico. Ainda é apresentada uma proposta de biblioteca de componentes, a serem utilizados em sistemas de automação industrial.

A UML define um conjunto muito maior de diagramas que são passíveis de serem utilizados no modelo de componentes. A intenção com os diagramas utilizados foi disponibilizar um modelo simples, mas capaz de caracterizar grande parte dos aspectos abordados em sistemas de automação industrial. Certamente, para alguns tipos de aplicações seria mais conveniente a utilização de algum outro diagrama da UML, mas isto acarretaria num acúmulo de diagramas, que seriam desnecessários para a maioria das aplicações. No capítulo 5 é descrita a implementação realizada a partir deste projeto conceitual.

4 Extensão do Ambiente SIMOO-RT

4.1 Introdução

SIMOO [COP97] é um *framework* originalmente concebido para a construção de modelos de simulação discreta orientados a objetos. O mapeamento de entidades de simulação em elementos autônomos, 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.

Posteriormente SIMOO foi estendido para permitir suporte ao desenvolvimento de sistemas tempo-real. Por exemplo, em SIMOO-RT [BEC99] foram adicionadas extensões para permitir a descrição de requisitos temporais e a especificação do comportamento dos objetos através de máquinas de estados e a capacidade de geração de código em uma linguagem de programação para aplicações de tempo real.

Mesmo na sua versão básica, o SIMOO define dois tipos de classes que se distinguem pela forma de comunicação com as demais. Elas podem se relacionar através de mensagens ou portas, como pode ser visto na Figura 4.1. Classes que utilizam mensagens descrevem um mecanismo de comunicação RMI, em um modelo semelhante ao definido pelo paradigma de orientação a objetos. Na Figura 4.1 esta relação aparece entre os elementos *Active Class* e *Passive Class*. Já as classes que se relacionam através de portas seguem um modelo similar ao das interfaces definidas no modelo de componentes [VIL2001]. Na Figura 4.1 este relacionamento aparece entre os elementos *Component* 1 e 2. Como se pretende estender o SIMOO-RT para trabalhar com componentes distribuídos, o modelo de classe que será utilizado nas aplicações será aquele no qual a classe se relaciona através de portas. Assim, a classe representará o componente, ao qual pode-se adicionar um numero finito de portas, onde cada porta representará uma interface.

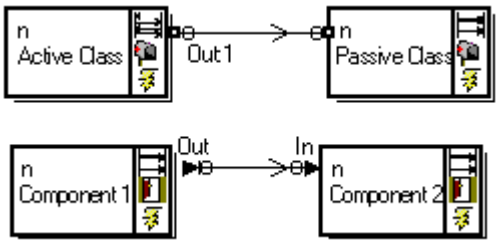

FIGURA 4.1 - Tipos de classes do SIMOO-RT

Neste sentido o conceito de componente dentro do SIMOO-RT limita-se à definição de uma classe, ou a composição de classes, que utilizam portas para a troca de mensagens. No SIMOO, uma aplicação é representada por classes em diferentes níveis hierárquicos, sendo o primeiro nível a visão mais abstrata da aplicação até o último nível numa visão mais detalhada da aplicação. Para a implementação dos componentes, considerou-se que as classes de segundo nível são as que representam os componentes da aplicação. Os níveis internos destas classes representam apenas a composição deste componente, não sendo sujeitas a distribuição. Com isso, é possível descrever de forma parcial o conceito de componente dentro do SIMOO-RT. Para possibilitar uma descrição mais completa do modelo de componentes são necessárias algumas alterações no conjunto de diagramas do SIMOO-RT, como por exemplo possibilitar a adição de alguns elementos complementares, tais como o diagrama de distribuição e a biblioteca de componentes.

4.2 Diagramas SIMOO-RT

4.2.1 Introdução

Tanto a construção deste novo diagrama, como a alteração do diagrama atual, envolvem uma reestruturação do SIMOO, pois a adição destes novos conceitos deve ser realizada nos modelos de base da ferramenta. Como um dos objetivos deste trabalho é o de estender a ferramenta SIMOO-RT para possibilitar a modelagem a partir de componentes, optou-se por adotar o conceito parcial de componente, pois mesmo não sendo completo, o diagrama atual é suficiente para validar o modelo. Esta escolha justifica-se pela manutenção da compatibilidade entre as demais versões do SIMOO, e em prol das demais modificações que envolvem a especificação e implementação de recursos que possibilitem

o reuso, a configuração da forma da distribuição da aplicação e a nova geração de código. A Figura 4.2 mostra o diagrama implementado no SIMOO-RT. No diagrama da esquerda são representadas as ligações estáticas entre os tipos de componentes. Já no diagrama da direita são representadas as ligações dinâmicas entre os componentes da aplicação propriamente ditos.

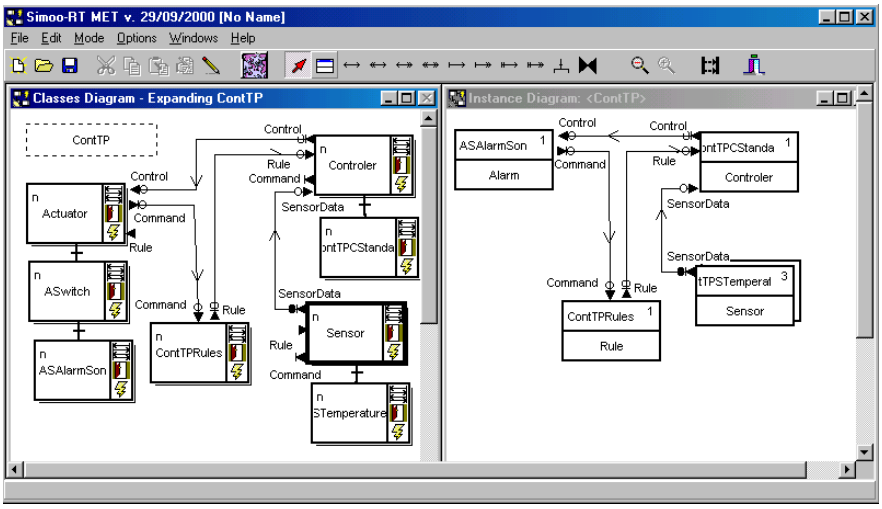

FIGURA 4.2 - Diagrama de Componentes SIMOO-RT

Para facilitar a compreensão dos diagramas propostos, será utilizado um estudo de caso de um controlador de temperatura composto por um sistema de controle, que a partir da média de três temperaturas, define se um alarme deve ou não ser disparado. Os diagramas apresentados na Figura 4.2 representam os relacionamentos destes componentes. Por exemplo, o componente *ASAlarmSom* herda as características do componente *ASwitch*, que herda as características do componente *Actuator*. Sendo assim as interfaces *Control*, *Command* e *Rule* também fazem parte do componente *ASAlarmSom*. Como as interfaces do componente *Actuator* estão relacionadas com as interfaces de mesmo nome dos componentes *Controler* e *ConTPRules*, estes relacionamentos também são herdados pelo componente *ASAlarmSom*.

Os dois diagramas que descrevem a epistemologia da aplicação também já fazem parte do modelo definido por Becker e já compõem os diagramas do SIMOO-RT [BEC99]. Estes diagramas são o editor de cenários, que corresponde ao *Sequence Diagram* definido pela UML, e o editor de fluxo de dados, correspondente ao *Statechart* da UML.

4.2.2 Diagrama de Distribuição

De acordo com a ferramenta proposta é necessária a descrição da forma da distribuição do modelo. Neste sentido, foi adicionada ao SIMOO-RT uma ferramenta baseada no *Deployment Diagram* definido pela UML (vide seção 3.2.2), denominada *Deployment Diagram Editor* (DDE).

O DDE define um subconjunto dos elementos descritos pela UML. A diferença deste diagrama para o definido pela UML está na definição das dependências entre componentes distintos. No modelo da UML esta dependência é mostrada neste diagrama. Já no caso do SIMOO, ela é evidenciada no diagrama de instâncias, não sendo necessário repeti-la no novo diagrama.

Os demais conceitos definidos pela UML, como o *Node* e *Link* foram implementados neste diagrama. A Figura 4.3 mostra a forma destes elementos relacionando-os com os descritos pela UML.

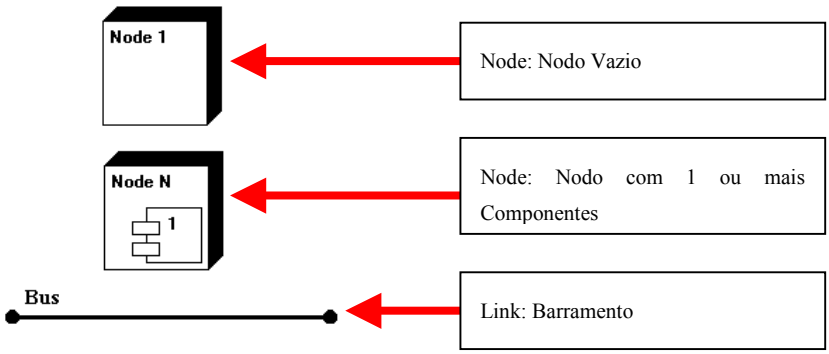

FIGURA 4.3 - Elementos implementados no DDE

A Figura 4.4 mostra o editor implementado no SIMOO-RT, onde pode-se observar as cinco áreas diferenciadas definidas para o DDE:

- Área de Trabalho: Área onde é desenhado o diagrama. É a maior área do editor, e está localizada à direita do mesmo.

- Área de Informações: Área expressa na lateral esquerda do editor, que contém as informações do diagrama, do nodo e modo de trabalho corrente.

- Área de Mensagens: Área destinada a mensagens relacionadas às ações correntes do usuário. Ela está localizada na parte inferior do editor.

- Área de Menus: Área destinada aos menus. Ela está localizada acima das áreas de trabalho e informação.

- Área de Título: Área destinada ao nome do diagrama, juntamente com o nome do arquivo que contém o diagrama que está sendo editado. Ela está localizada na parte superior do editor.

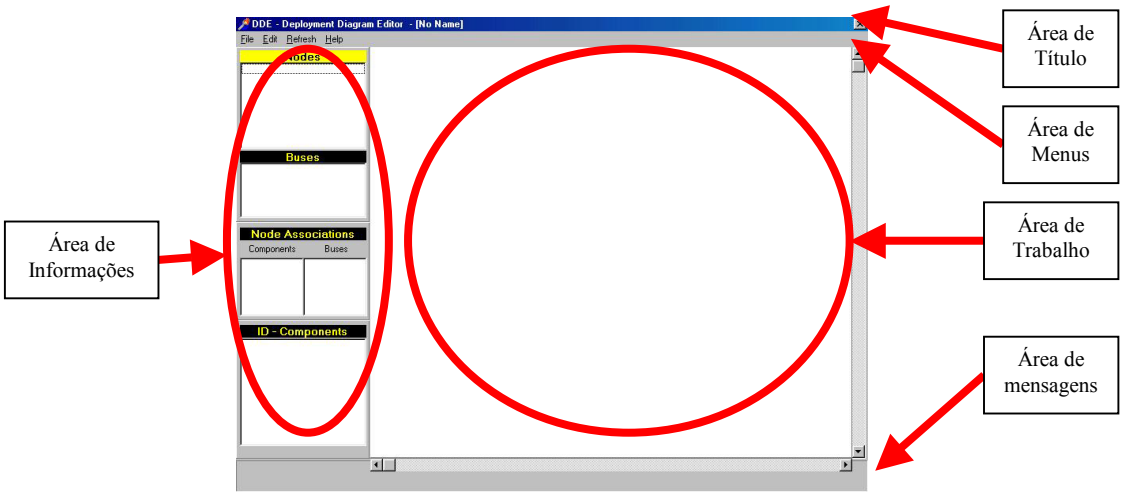

FIGURA 4.4 - Deployment Diagram Editor

O usuário somente conseguirá utilizar o DDE, se possuir uma aplicação já modelada no MET. Quando o usuário acessa o DDE, a área de trabalho é deixada vazia e os nomes das instâncias de segundo nível, descritas no diagrama de instâncias do MET, são automaticamente adicionadas à lista de componentes para a distribuição, podendo ser visualizadas na caixa *ID-Components* da área de informações (Figura 4.4).

Para realizar os procedimentos de construção do diagrama, o usuário deve selecionar um dos quatro modos de trabalho descritos na área de informação, através de um *click* com o mouse sobre o nome do modo: *Nodes*, *Buses*, *Node Association* ou *ID-Components*.

Nos modos *Nodes* e *Buses* é possível incluir, excluir e editar um nodo ou barramento. Para isso basta utilizar o mouse, selecionando pontos na área de trabalho. Quando um nodo ou barramento é incluído ou editado, informações adicionais ao elemento são perguntadas através das janelas auxiliares mostradas na Figura 4.5.

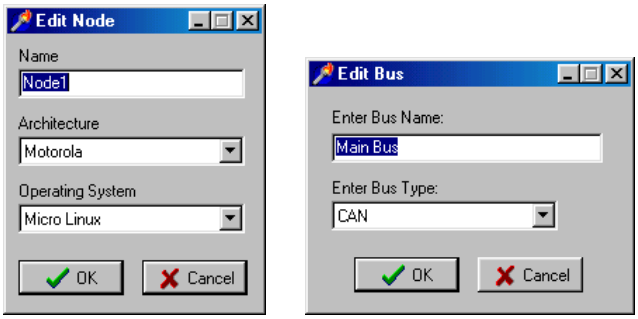

FIGURA 4.5 - Janelas de Edição

Desta forma, quando um nodo é adicionado, pode-se definir qual é a plataforma de hardware e software implementada por este nodo e qual é o nome deste nodo. De forma similar, quando um barramento é adicionado, pode-se definir qual é o protocolo de comunicação implementado e o nome que será dado a este barramento.

No modo *node Associations* é possível criar e excluir associações entre nodos e barramentos. Para isso, o usuário deve selecionar um barramento para adicionar um ponto de associação. Esta seleção se dá através de um *click* do mouse em qualquer ponto do barramento. Logo após, o usuário deve arrastar este ponto até o nodo desejado, completando a associação. Neste modo também são possíveis as edições dos nodos e dos barramentos, bem como a exclusão das associações com componentes.

Por fim, no modo *ID-Components* é possível adicionar associações de componentes aos nodos, através de um procedimento do tipo *Drag and Drop*, no sentido do componente para o nodo. Este modo também permite a edição dos nodos e dos barramentos.

Os dados deste modelo são salvos em arquivo, sendo que os modelos de distribuição gerados podem ser reaproveitados para qualquer outra aplicação. Isso se deve ao fato de que o arquivo gerado pelo DDE é subdividido em duas partes. A primeira contém apenas as informações do modelo de distribuição, ou seja, dos nós e dos barramentos. A segunda

parte contém as informações das associações. Esta segunda parte será ignorada toda vez que se tentar abrir um modelo que não foi gerado inicialmente para a aplicação atual. O DDE é vinculado ao diagrama de instancias de uma aplicação, sendo que cada aplicação pode ter N modelos gerados pelo DDE.

Uma característica importante do DDE implementado é da manutenção da consistência dos diagramas. Esta consistência é mantida por um conjunto de regras implementadas pelo DDE. Por exemplo, os componentes são definidos apenas pelo diagrama de instâncias do MET, por isso nunca existirá um diagrama com componentes inexistentes no modelo da aplicação. Ou ainda, sempre antes de salvar o arquivo do diagrama é verificado se todos os relacionamentos entre interfaces são possíveis de implementar no diagrama descrito pelo DDE. Se foram possíveis, o arquivo é simplesmente salvo. Se não, um aviso com as inconsistências encontradas no diagrama é mostrado.

A Figura 4.6 mostra uma proposta para a arquitetura do estudo de caso do controlador de temperatura. Nesta proposta existem dois nodos sendo o Nodo1 um PC/Linux, executando os componentes *Alarm*, *Rule* e *Controler*, e um Nodo2 um Motorola/µClinux, executando os três sensores.

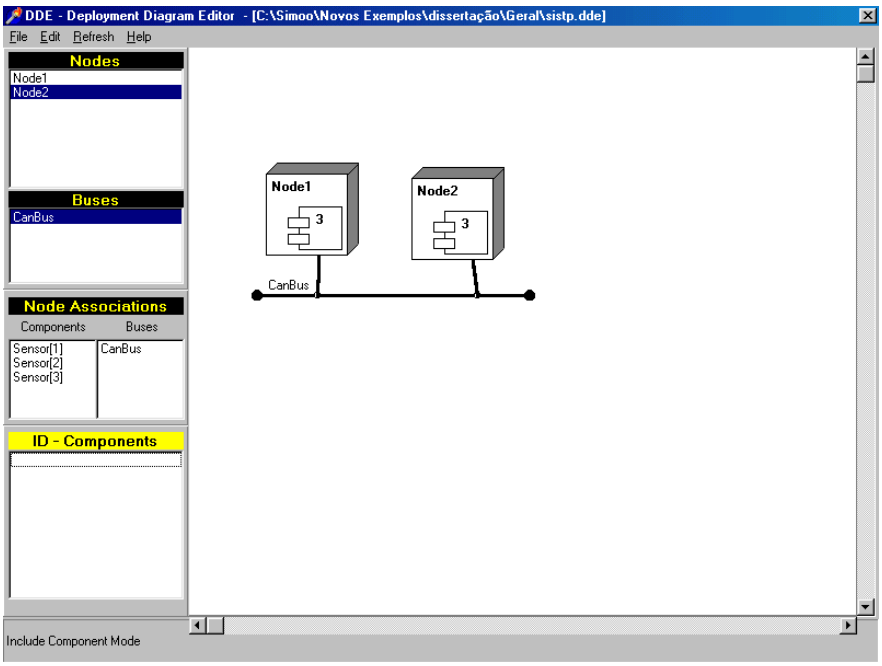

FIGURA 4.6 - DDE Controlador de Temperatura

4.3 Biblioteca de Componentes

Em sua versão original o SIMOO-RT possibilitava a importação e exportação de classes a partir do Diagrama de Classes do editor de modelos (MET). Apesar disto funcionar de forma eficiente, tratando não só da estrutura, mas como também dos relacionamentos de uma classe, nenhuma informação adicional que auxilie a pesquisa ou o reconhecimento desta classe é adicionada. Além disso, os arquivos são armazenados sem um critério específico, o que dificulta a procura por elementos individuais. Sendo assim o modelo original do SIMOO-RT não provê um suporte adequado para a reusabilidade.

A fim de solucionar-se esta deficiência, foi adicionado ao SIMOO-RT um sistema de gerenciamento de bibliotecas de componentes. Este sistema, além de permitir a criação de componentes, auxilia a pesquisa por determinados componentes na biblioteca. A solução proposta utiliza um arquivo padrão de exportação do SIMOO e uma tabela de um banco de dados para armazenar informações adicionais dos componentes. São armazenadas as informações de descrição do componente, de suas interfaces e de suas dependências de contexto, bem como o *link* para o arquivo SIMOO relacionado. A escolha da utilização desta forma de armazenamento deve-se ao grande número de informações que definem o código de um componente ou classe. Se toda esta informação fosse armazenada na biblioteca, a performance do banco de dados certamente seria prejudicada. Uma outra vantagem da utilização deste arquivo para o código fonte está na possibilidade do aproveitamento deste código por outras versões do SIMOO que não incorporaram esta biblioteca. A Figura 4.7 mostra a interface desta biblioteca.

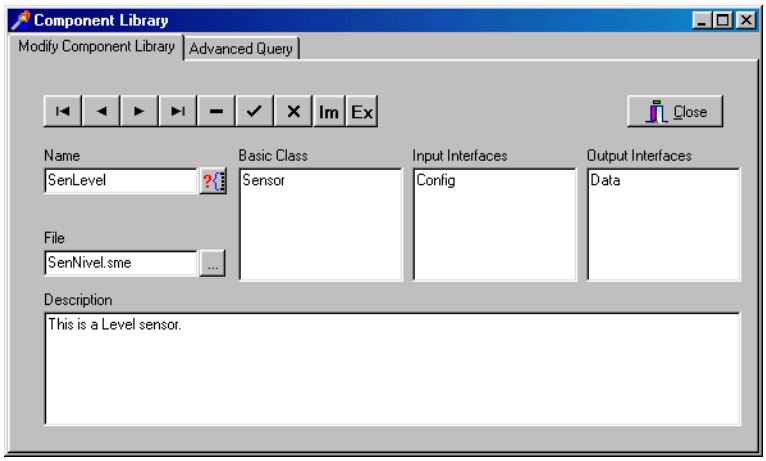

FIGURA 4.7 - Biblioteca de Componentes

Um usuário do SIMOO-RT pode importar (Im) ou exportar (Ex) componentes entre a biblioteca de componentes e o Diagrama de Classes do MET. No processo de exportação o usuário deve selecionar uma classe (ou componente) no diagrama de classes do MET. Após esta identificação o usuário seleciona o comando de exportação através do botão (Ex) mostrado na Figura 4.7. A partir disso a biblioteca automaticamente determina um nome para o arquivo (nome do componente) e quais são as interfaces e dependências de contexto do componente, ficando apenas a cargo do usuário o preenchimento da descrição do mesmo. Se o componente for uma especialização (herança) de algum outro componente, este componente base também será adicionado à biblioteca. A cada processo de exportação é realizada uma validação, onde se o componente a ser exportado para a biblioteca já estiver cadastrado, o usuário será notificado através da requisição de uma solução para o problema, ou seja, o cancelamento da exportação ou o *upgrade* do componente na biblioteca.

No processo de importação, o usuário deve chamar a biblioteca, selecionar um componente através da pesquisa por nome (vide figura 4.8), e pressionar o botão (Im). A partir disso a biblioteca colocará automaticamente o componente no MET. Com isto, o componente deve aparecer no diagrama de classes MET, na mesma posição onde ele foi exportado originalmente. Se o componente for uma especialização, o componente de origem também será importado para o MET. Isto ocorrerá de forma recursiva até encontrar o componente mais elementar. Se algum destes componentes já fizer parte do Diagrama de

Classes do MET, será requisitada a escolha de *Upgrade* do componente ou cancelamento da importação.

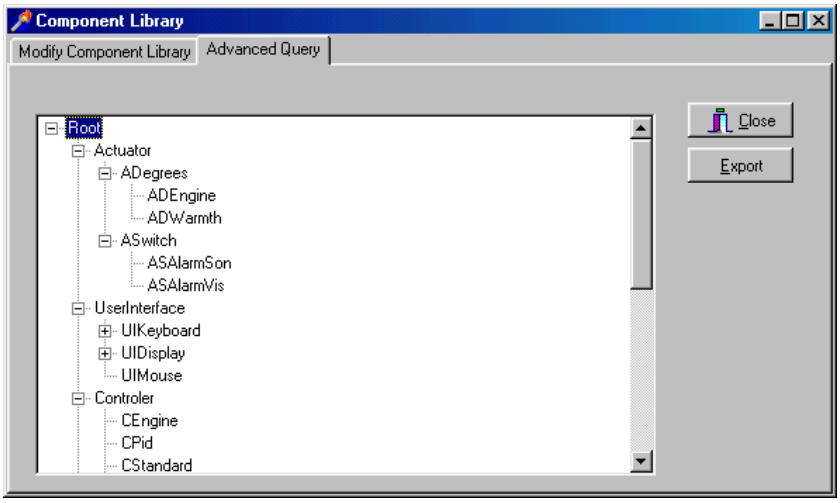

FIGURA 4.8 - Consulta Avançada

No estudo de caso do controlador de temperatura todos os componentes que fazem parte deste modelo, exceto o componente *ComTPRules*, foram importados da biblioteca de componentes.

4.4 Geração de Código Executável

Para completar o ambiente proposto foram adicionadas novas características ao gerador de código. Mantendo a característica original da ferramenta [BEC99], parte do código fonte é gerada através das informações dos diagramas e o restante através de codificação em C++ efetuada pelo usuário. Basicamente, a única alteração visível aos usuários da ferramenta é a da necessidade de identificação de qual *Deployment Diagram* deverá ser utilizado na geração de código. Apesar desta semelhança estrutural, no novo código gerado existem algumas diferenças fundamentais na estrutura das classes do código C++ e no sistema de suporte à comunicação.

As formas dos diagramas para esta nova geração de código já foram descritas nos itens anteriores deste capítulo, e referem-se principalmente a adição do *Deployment Diagram*. Já o sistema de suporte à comunicação está baseado no modelo

Publisher/Subscriber visto no item 2.5.5. A diferença deste para o modelo original é que, neste modelo, apenas o ECH foi construído. Isso ocorre porque todas as definições de canais e mensagens são descritas em tempo de projeto, pelos diagramas ou códigos fonte escritos dentro do SIMOO-RT, o que torna desnecessária a implementação do ECB. A simplicidade da estrutura e a independência dos elementos de distribuição foram os principais motivos da sua escolha. Estas vantagens possibilitam a implementação desta estrutura em plataformas menores de hardware e software, tais como sistemas microcontrolados.

Por fim a nova estrutura do código C++, utilizada para descrever os componentes do SIMOO-RT, difere da anterior pela forma de interação entre os elementos de distribuição. No modelo anterior, baseado em objetos, quando um objeto estava relacionado a outro, o primeiro poderia executar os métodos do segundo realizando uma chamada direta do método do objeto específico. Caso este objeto estivesse executando em um nodo distinto ao do objeto que chama o método, a estrutura do AO/C++ transformava esta chamada em uma RMI. Na nova estrutura, os componentes não possuem relacionamentos diretos. Todo o relacionamento é realizado através das interfaces e do *Publisher/Subscriber*. Isto significa que se um componente desejar um serviço, como a execução de um método qualquer de outro componente, ele deve publicar este desejo no canal específico. Da mesma forma, se ele desejar receber a resposta, ou ser informado quando este método específico foi executado, ele deve se subscrever no canal específico.

Neste modelo de comunicação toda interface irá publicar uma mensagem que será lida e retransmitida pelo ECH para todas as interfaces cadastradas neste canal e mensagem específicos. O ECH irá retransmitir esta mensagem através do barramento utilizando os recursos do protocolo relacionado, e para qualquer outro componente do mesmo nodo, que obviamente estiver cadastrado neste canal para esta mensagem.

Neste sentido, observa-se que a alteração efetuada no *middleware* de comunicação é realizada em dois níveis. O primeiro nível é definido pelo ECH, onde é mantida uma lista atualizada de canais e mensagens. Esta lista relaciona estes canais e mensagens aos componentes da aplicação, definindo o barramento e a arquitetura do nodo de execução. A Figura 4.9 mostra as classes ECH, SYSTEM, BUS e TABLE implementadas. A classe

BUS é responsável pela comunicação no nível do protocolo de rede (Ex.: TCP/IP). A classe SYSTEM é responsável pela comunicação no nível do sistema operacional do nodo (ex.: sockets). A classe TABLE é responsável pela manutenção das informações sobre os relacionamentos entre componentes, canais e eventos. A classe ECH é responsável pelo gerenciamento de toda a comunicação e está diretamente associada a um BUS, um SYSTEM e um TABLE.

```
typedef struct {
    lond int ChannelID;
    long int MessageID;
    char     Msg[8];
    char     Send;
   } Message;

class SYSTEM {
     public:
      SYSTEM();
      ~SYSTEM():
      accept();
      bind();
      listen();
      CreateThread();
      int  read(Message *msg);
      int  write(Message *msg);
      int  proxy(long int ID,char *msg,int sizemsg,int x);
      int  proxy_detach(long int senderID);
     };

class BUS {
      int  ID;
      int  txnode;
      void *Config;
     public:
      BUS(void *cfg);
      ~BUS();
      void TurnON(void);
      void TurnOFF(void);
      int  ConfigRequest(int nodeID);
      int  BindResquest(int txnd,int eventID);
      int  ReadMessage(Message *msg);
      int  WriteMessage(Message *msg);
     };
```

```
typedef struct {
     long int ChannelID;
     long int MessageID;
     long int ComponentID;
     char     Location;
    } ROW;

class TABLE {
      ROW *Table;
      int NumRow;
     public:
      TABLE();
      ~TABLE();
      void add(long int CID,long int MID,long int CID,char Loc);
      void clear(void);
      int  FindFirst(long int ChID, long int MID, long int *CID,
                     char *Loc);
      int  FindNext(long int ChID, long int MID, long int *CID,
                     char *Loc);
      int  RowCount(void);
    };

class ECH {
     TABLE *table;
     BUS *bus;
     SYSTEM *system;
    public:
     ECH(TABLE *t,BUS *b,SYSTEM *s);
     ~ECH();
     void TableClear(void);
     void TableAdd(long int ChID,long int MID,long int CID,char Loc);
     void Monitor(void);
     int Send(Message *M);
    };
```

FIGURA 4.9 - Cabeçalho do ECH

O segundo nível está localizado em cada componente da aplicação. Nele um *loop* de execução monitora a chegada de mensagens, ativando o processo relacionado a cada mensagem de cada canal. A Figura 4.10 representa a estrutura básica deste *loop* e a inicialização do componente sensor utilizado no estudo de caso do controlador de temperatura..

```
main(int argc, char *argv[])
{
 STemperatura Sensor;
 short continue_loop = OK;
 char rcv_msg[250];

 System.qnx_name_attach(0, "Sensor");
 strcpy(Sensor.name,"Sensor");
 continue_loop = 1;
 while(continue_loop) {
     rcv_msg[1] = -1;
     Sensor.sender_pid = System.Receive(0, rcv_msg, sizeof(rcv_msg));
     if (Sensor.Select(rcv_msg) == SENSOR_END )
       continue_loop = 0;
      else
       System.reply(Sensor.sender_pid, &continue_loop, sizeof(short));
  }
}
```

FIGURA 4.10 - Estrutura do Programa Principal de um componente Sensor

A Figura 4.11 mostra o método de seleção implementado por este mesmo componente sensor. O método de seleção é responsável pela verificação das mensagens enviadas pelo ECH e pela execução do método do componente correspondente ao evento identificado no corpo da mensagem.

```
int Sensor::Select(char *recv_msg)
{
 int sel_flag = 1, event = 0,channel;
 event = recv_msg[6];
 channel  = event>>8;
 swicth (channel) {
    case RULE:
            switch(event) {
              case RULE _START:
                        Sensor_Start_msg* P_Sensor_Start_msg;
                        P_Sensor_Start_msg = (Sensor_Start_msg*) (recv_msg);
                        System.proxy_detach(sender_pid);
                        Start( P_Sensor_Start_msg->string1  );
                        sel_flag = SENSOR _START;                    break;
              case RULE_END:
                        Sensor_End_msg* P_Sensor_End_msg;
                        P_ Sensor_End_msg = (Sensor_End_msg*) (recv_msg);
                        System.proxy_detach(sender_pid);
                        End( );
                        sel_flag = SENSOR_END;                    break;
              case RULE_CONFIG:
                        Sensor_msg* P_Sensor_Config_msg;
                        P_Sensor_Config_msg = (Sensor_Config_msg*) (recv_msg);
                        System.proxy_detach(sender_pid);
                        Config( P_Sensor_Config_msg->par1 ,P_Sensor_Config_msg->par2 );
                        sel_flag = SENSOR_CONFIG;                    break;
            }
   }
 return sel_flag;
}
```

FIGURA 4.11 - Processo SELECT do Componente Sensor

A base deste mecanismo de comunicação entre os nodos está na filtragem das mensagens, sendo que a sua implementação irá depender do tipo de protocolo utilizado. Em protocolos baseados em mensagens do tipo *broadcast* ou *multicast*, como por exemplo o CAN-bus, esta filtragem pode ser realizada automaticamente por configuração do hardware do nodo receptor. Em protocolos que se baseiam em endereços, como por exemplo o TCP/IP, isto não é possível, sendo então responsabilidade do ECH que está enviando a mensagem localizar os ECHs dos nodos receptores da mesma.

Em sua versão atual o gerador de código permite criação de programas para 4 plataformas de hardware e software distintas: Linux com TCP/IP; o Linux com CAN; o QNX com TCP/IP; e o μClinux com CAN. Mesmo não representando um número quantitativamente expressivo de combinações (uma vez que o número de combinações possíveis com sistemas comerciais pode chegar à ordem de centenas), tem-se que as combinações escolhidas são bastante representativas, uma vez que combinam o uso de sistemas microcontrolados e baseados em PC e dois protocolos de comunicação largamente utilizados como o TCP/IP e CAN.

A inicialização da aplicação deve ser realizada a partir dos ECHs de cada nodo, para que sejam construídos os mecanismos de comunicação que darão suporte à aplicação. Após deve ser realizada a inicialização dos componentes que fazem parte de cada nodo, onde cada componente irá informar ao ECH quais os canais e mensagens ele está interessado. Esta inicialização deve estar descrita no construtor do componente (no SIMOO-RT corresponde ao método *Start*). A Figura 4.12 mostra como é feito o cadastro para o recebimento das mensagens do componente *Controler*, no estudo de caso do controlador de temperatura.

```
subscribe(Rule,"EVENT_PROG");
subscribe(SensorData,"EVENT_DATA");
```

FIGURA 4.12 - Código para Subscrição de Mensagens

Um componente da aplicação ainda pode ser formado pela composição de outros componentes e/ou classes. Neste caso o relacionamento interno é mantido por um componente auxiliar semelhante ao ECH, com o objetivo principal de manter os canais

internos de comunicação do componente. Apesar de possuírem a mesma funcionalidade geral, este componente auxiliar difere-se do ECH pelo fato de que ao invés de estar conectado ao barramento, ele está conectado às interfaces externas do componente. Ou seja, quando uma mensagem chegar por uma das interfaces externas, este ECH interno irá replicar esta mensagem a todas as interfaces internas de mesmo nome. Quando um componente interno transmitir uma mensagem, ela será lida por todas as interfaces internas de mesmo nome, e, se existir, pela interface externa igualmente de mesmo nome. Ao perceber a chegada desta mensagem a interface externa irá replicar esta mensagem ao ECH do nodo.

O resultado final de todo o processo de geração de código é definido por dois tipos de conjuntos de arquivos, os que definem um componente e os que definem um nodo. Quando um componente é composto apenas de uma classe, o resultado do processo de geração de código deste será quatro arquivos: o arquivo de cabeçalho (*.h); o arquivo de código do componente (*.c); o arquivo de inicialização do componente (*Main.c); e o arquivo de projeto (*makefile*). Entretanto, quando ele é formado pela composição de outros componentes, definidos nos níveis internos do diagrama de classe, o resultado da geração de código será os quatro arquivos do componente final, mais dois arquivos para cada componente de composição, sendo um de cabeçalho (*.h) e outro com o fonte (*.c) e mais o componente auxiliar (*ech.c e ech.h). Obviamente, no caso de composição de componentes, o arquivo de projeto deverá ser modificado para possibilitar a geração de código a partir de todos os arquivos da composição.

No estudo de caso do controlador de temperatura serão gerados 9 conjuntos de arquivos fonte (*.c e *.h), referentes aos componentes e suas composições, 1 conjunto (*ech.c e *.ech.h), referente à composição do componente *Sensor* (único composto por outros componentes), 4 conjuntos de arquivos principais de componentes (*main.c e *main.h), 1 conjunto de arquivos para a construção do nodo PC/Linux e outro para o nodo Motorola/μClinux e 6 arquivos de construção (*makefile*), sendo um para cada componente ou nodo.

5 Estudos de Caso

5.1 Introdução

Para a validação da proposta apresentada, bem como das características e versatilidades do ambiente proposto, foram realizados três estudos de caso, baseados em sistemas de automação reais com características funcionais distintas. O primeiro é um robô, com características típicas para sistemas de automação industrial. O robô Janus [PER2001] é constituído de um sistema de visão com duas câmeras digitais, apoiadas sobre um sistema articulado com dois graus de liberdade, com a função similar ao de um pescoço, e dois braços articulados com oito graus de liberdade cada.

O segundo estudo de caso é o de um sistema de controle para múltiplos elevadores. O sistema deve ser modular para possibilitar o controle de 1 a N elevadores dentro de uma mesma edificação. Um dos aspectos básicos a ser considerado refere-se à alocação de elevadores para atendimento a chamadas, o que deve ser feito baseado nos estados atuais dos elevadores (andar em que se encontram, andar a que se destinem e relação destes com o andar e sentido desejado pela pessoa que pressionou o botão de chamada). Este estudo de caso já foi utilizado em outras dissertações de mestrado relacionadas ao projeto SIMOO-RT [BEC99, BRU2000].

Por fim, o último estudo de caso trata de uma aplicação hospitalar: o controle e monitoração de parâmetros fisiológicos em uma unidade de calor radiante UCRMPF [VIL99]. Este sistema tem como finalidade principal manter a temperatura corporal de um neonato dentro de uma faixa de normalidade previamente definida. Além da temperatura, este sistema é capaz de monitorar a freqüência cardíaca e a freqüência respiratória, acionando alarmes quando valores preestabelecidos foram ultrapassados. Estes estudos de caso são descritos nas próximas seções.

5.2 Estudo de Caso 1: Robô Janus

5.2.1 Arquitetura Janus

Janus é um sistema robótico estacionário consistindo de um sistema de visão e dois braços mecânicos complexos como manipuladores, como pode ser visto na Figura 5.1. O sistema de visão é montado sobre um pescoço com dois graus de liberdade. Para pegar ou manipular objetos, os dois braços mecânicos do Janus são subdivididos em 8 juntas. Todas as juntas são monitoradas por sensores óticos, que juntamente com o sistema de visão formam uma rede complexa de supervisão. A Figura 5.2 mostra a arquitetura de controle que será utilizada para este estudo de caso.

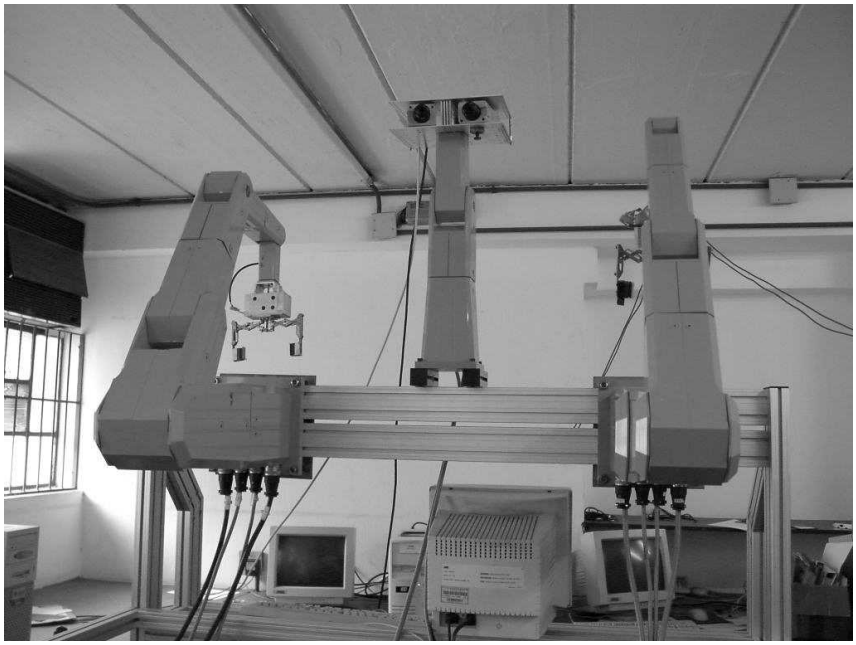

FIGURA 5.1 - Janus

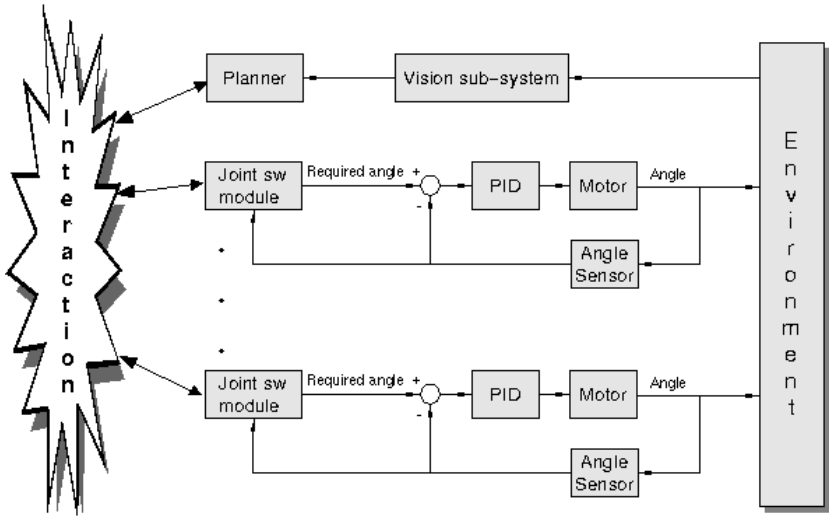

FIGURA 5.2 - A Arquitetura de controle do Robô Janus

- ***Planner*** - responsável pela definição dos objetivos e de como chegar a eles;

- ***Vision sub-system*** - responsável pela aquisição e tratamento das imagens captadas pelas câmeras digitais;

- ***Joint SW module*** - responsável pelo movimento individual de uma articulação de um dos membros. Cada *Joint*, ou junta, possui sensor de ângulo (*Angle Sensor*), um motor (*Motor*) e um sistema de controle do tipo PID;

- ***Member*** - apesar de não aparecer no modelo da Figura 5.2, este é um elemento auxiliar que é responsável pela sincronização dos movimentos de várias articulações (braço ou pescoço).

O modelo de controle utilizado para este estudo de caso é o mesmo utilizado no estudo de caso realizado por [PER2001]. Nesta proposta o controle é tratado com o modelo de multi-agentes, onde o sistema deve possuir dois níveis hierárquicos de controle, os níveis dos agentes reativos e o dos cognitivos.

No nível reativo cada agente, que é representado por um componente de software, representa o sistema de uma das juntas (*Joint SW Module*). No nível cognitivo, o *Planner* deve se comunicar com as juntas do nível reativo, para dar a elas os direcionamentos

individuais de cada junta e destino final do membro. Tanto o direcionamento com o destino são representados por coordenadas cartesianas tridimensionais.

Quando um plano de movimento para um dos membros está pronto, todos as juntas que compõem este membro são comunicadas, e o algoritmo de controle distribuído é executado. O funcionamento deste algoritmo é independente do número de dimensões da região de trabalho, por isto pode utilizar apenas duas dimensões para se construir uma rápida explicação dos seus conceitos. Como pode ser visto na Figura 5.3, o ciclo de movimento inicia pela junta mais externa até a mais interna, as quais se movimentam tentado diminuir o ângulo α. Quando uma junta termina de realizar um movimento ela comunica às demais a sua nova posição. A cada fase de movimento um novo destino pode ou não ser traçado, e quando o membro chega ao seu destino, mais nenhum movimento ou comunicação é realizado. Apesar de ser um algoritmo simples ele é eficaz e suficiente para testar os conceitos desta dissertação.

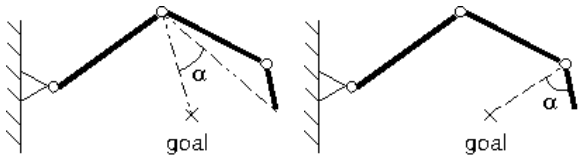

FIGURA 5.3 - Descrição do Algoritmo Distribuído

5.2.2 Modelo de Componentes SIMOO-RT

O primeiro passo, para a modelagem deste estudo de caso, é o da transposição destes elementos básicos para os diagramas de classe e instâncias do MET no SIMOO-RT. A Figura 5.4 mostra a implementação destes diagramas, onde cada elemento básico foi mapeado para um componente. No diagrama de classes são colocados os componentes para a implementação do Janus, definindo-se o relacionamento entre eles. No diagrama de instâncias são definidas as ligações especificas da aplicação proposta, que auxiliaram na configuração do gerador de código. Com a utilização das portas, a representação da multiplicidade dos relacionamentos é explicitada apenas através deste diagrama. Por exemplo, o pescoço (*Neck*) e suas duas juntas.

Para possibilitar a formação de um canal de comunicação, uma porta deve estar conectada a uma ou mais portas de outros componentes que possuam o mesmo nome. Isto também pode ser verificado nos diagramas da Figura 5.4.

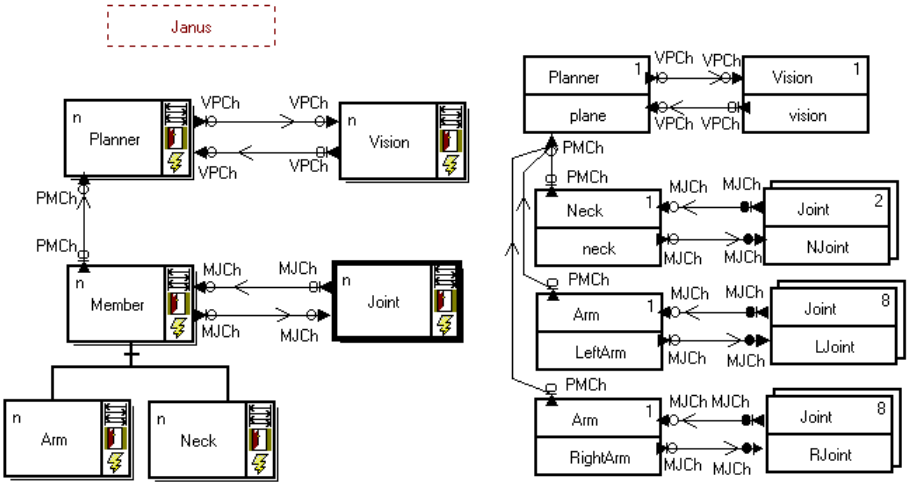

FIGURA 5.4 - Diagramas MET do SIMOO-RT para o Janus

Após a definição destes modelos é necessário descrever quais mensagens cada interface do componente deseja receber. Por exemplo, o *Planner* deve receber as mensagens com os dados do sistema de visão e dos membros e identificar que procedimento deve ser realizado quando a mensagem for recebida. A figura 5.5 ilustra esta configuração.

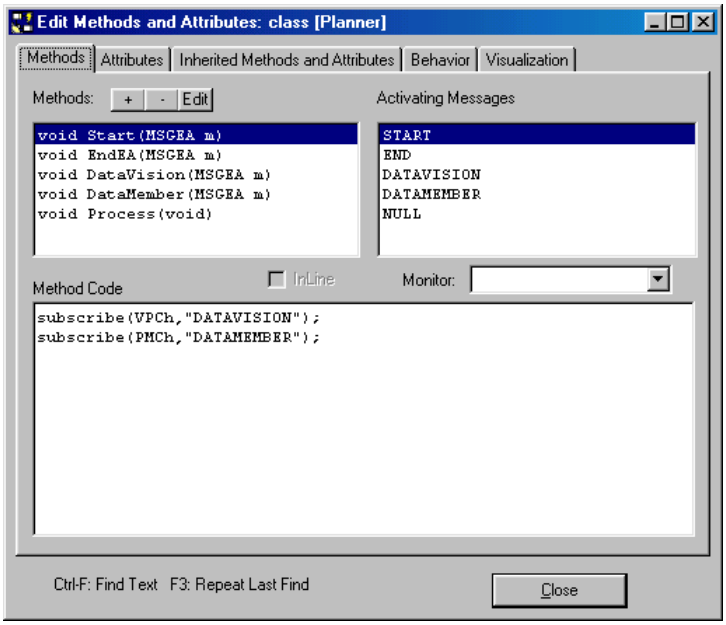

FIGURA 5.5 - Método Start do Planner

O Componente *Joint* da Figura 5.4, que representa uma junta do Janus, possui a sua borda realçada, pois é formado pela composição de outros componentes. A junta foi composta por elementos da biblioteca de componentes, motor, sensor de posição e controlador PID. A Figura 5.6 mostra esta composição.

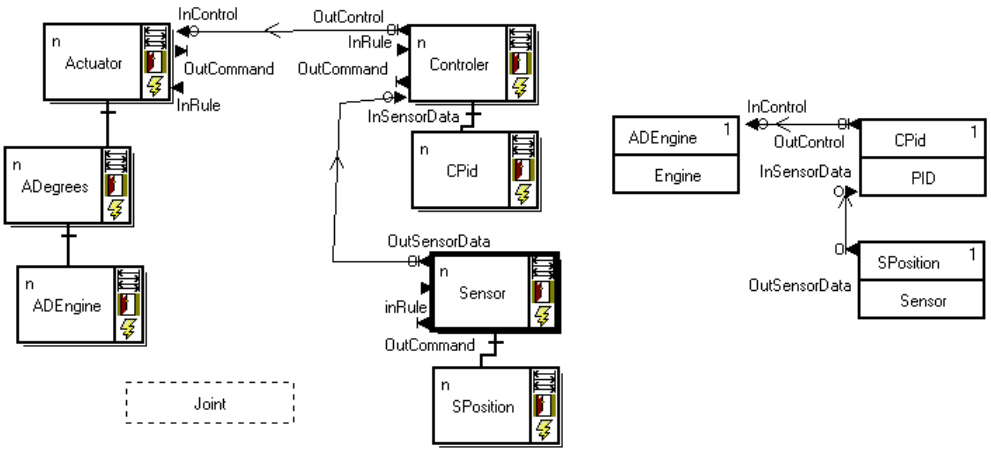

FIGURA 5.6 - Diagramas MET do SIMOO-RT - Composição das Juntas

A publicação das mensagens pode ser realizada de duas maneiras. Por exemplo, na junta uma mensagem pode ser transmitida ou ao final do algoritmo, como mostra a figura 5.7, ou através de uma solicitação explicita como mostra a figura 5.8.

```
if (fabs(Goal-Phase)>0.1) {
   PID.Process(Goal);
   Phase = Sensor.Read(();
   SndFlag = False;
  }
else {
      if (!SndFlag)
        publish(MJCh,"PHASE",Phase);
      SndFlag = True;
  }
```

FIGURA 5.7 - Método cíclico *Process* do componente *Joint*.

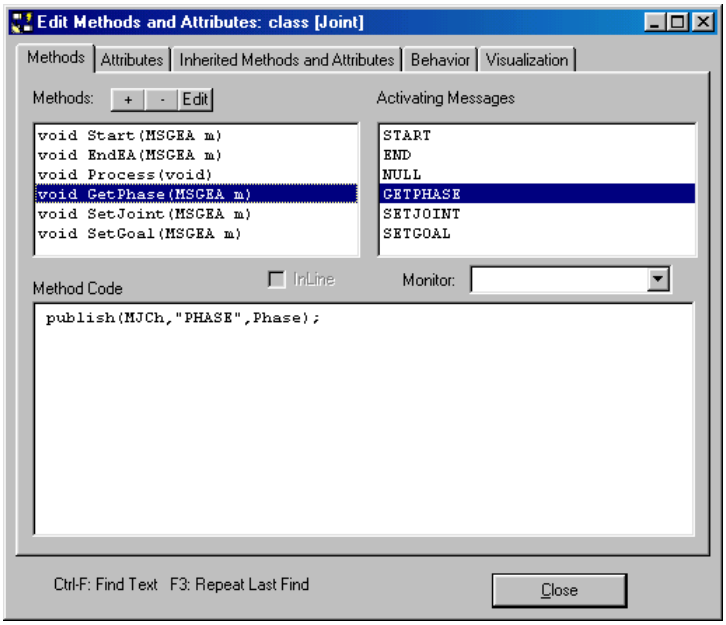

FIGURA 5.8 - Método *GetPhase* do *Joint*

Definido o algoritmo dos componentes e a forma da comunicação é necessária a definição da arquitetura alvo desta aplicação, ou seja, onde será executado cada componente da aplicação. A Figura 5.9 mostra uma possivel arquitetura de implementação para o Janus. Nesta implementação, foram colocados 4 nodos sendo 1 PC/Linux e 3 Motorola/µClinix, interligados através de um barramento CAN. Nele cada nodo Motorola executa os componentes de um dos membros. No nodo PC são executados os componentes de visão e planejamento.

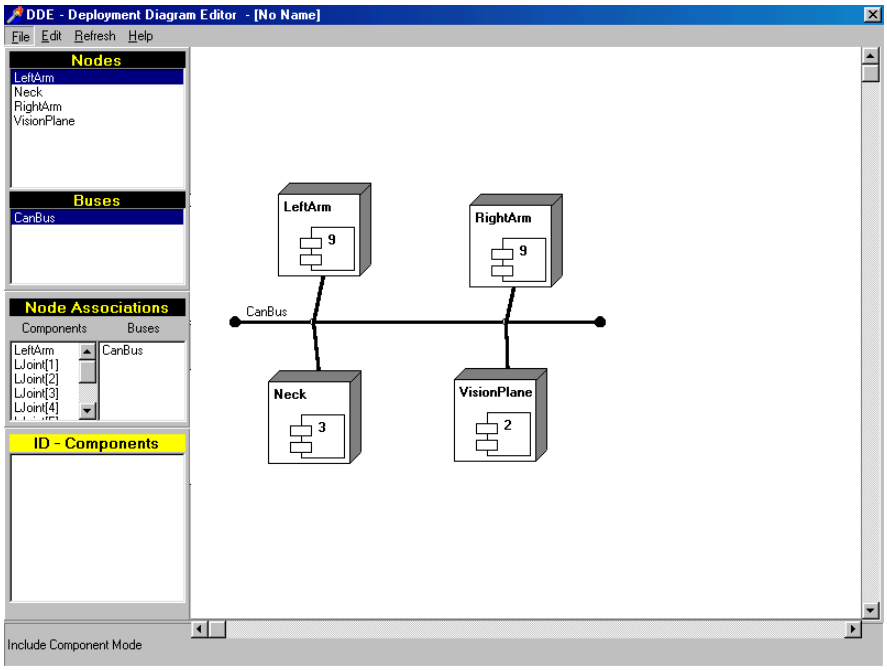

FIGURA 5.9 - DDE Janus Primeira Proposta

Como descrito no modelo, a partir do mesmo modelo do Janus é possível construir a aplicação com qualquer outro modelo de distribuição. A Figura 5.10 mostra outra possível arquitetura de implementação para o Janus. Nesta segunda implementação, foram colocados 6 nodos sendo 3 PC/Linux e 3 Motorola/µClinix, interligados através de um barramento CAN. Nele cada nodo Motorola executa os componentes das juntas de um dos membros (pescoço, braço esquerdo e direito). Já os nodos PC executam o componente de planejamento dos movimentos, o componente do sistema de visão e os componentes agregadores das juntas que formam os membros.

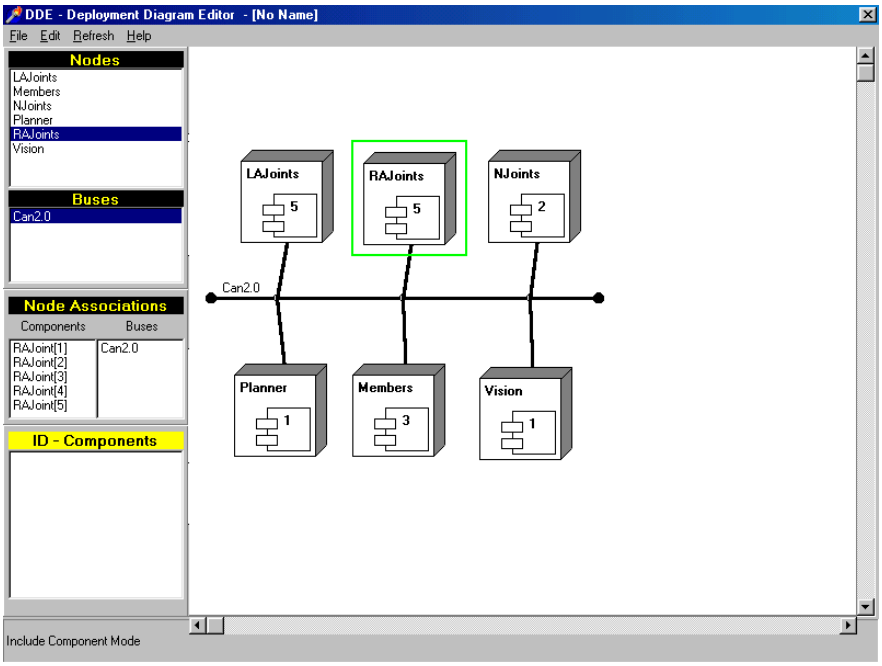

FIGURA 5.10 - DDE Janus Segunda Proposta

Ambas as propostas podem atender as exigências de hardware para o modelo Janus. Apenas a título de comparação sobre os aspectos da comunicação, pode-se verificar que a primeira proposta gera um número aproximadamente 2 vezes menor de mensagens que circulam pelo barramento. Isso ocorre, pois de acordo com o modelo, apresentado na Figura 5.4, o maior conjunto de mensagens é realizado entre o membro e suas juntas, ou entre o sistema de visão e o planejador. Na primeira proposta o conjunto de mensagens dos membros está confinada aos nodos *LeftArm*, *RigthArm* e *Neck*, e o conjunto de mensagens entre o sistema de visão e o planejador está confinado ao nodo *VisionPlane*. Já na segunda proposta, estas mesmas mensagens deverão obrigatoriamente ser transmitidas pelo barramento, o que obviamente cria um atraso nas comunicações.

O resultado final do processo de geração de código, para a primeira proposta, é a criação de diversos arquivos, sendo 16 com os códigos fonte (14 da aplicação mais o 2 para cada um dos tipos de ECH para os nodos), 16 com os cabeçalhos (14 da aplicação, o ECH e mais o arquivo *Heading.h*), 17 com os arquivos de execução dos componentes (um para cada instância de componente) e 25 arquivos de *makefile* (23 dos componentes existentes e 2 dos tipos de ECHs).

5.3 Estudo de Caso 2: Controle de Elevadores

5.3.1 Arquitetura do Sistema de Controle de Elevadores

Para a implementação deste estudo de caso foram modificados alguns dos conceitos do controle de elevadores definidos inicialmente por Brudna [BRU98]. A intenção destas modificações foi a de possibilitar uma melhor adaptação da aplicação ao modelo da biblioteca de componentes, sem que com isto fossem perdidos os objetivos iniciais.

A Figura 5.11 mostra de uma forma simplificada os elementos e relacionamentos que compões este estudo de caso. Todos os elementos são nomeados e identificados por um símbolo diferente, e todos os relacionamentos são representados por linhas e setas que indicam a direção e sentido das comunicações. Nesta figura é possível verificar três elementos principais, o elevador (*Elevator*), o gerente (*Manager*) e o andar (*Level*), onde: um elevador é composto de um sistema de movimento do elevador (EMotor , EControl e SPosition), um sistema de acionamento da porta (DMotor, Dcontrol e SBollean) e um sistema interno para requisição de andares (Keys); um andar é composto apenas por um ou dois botões de requisição dos elevadores; e um gerente é composto pelo algoritmo de escolha dos elevadores para atender as requisições dos andares. Para a comprovação do modelo, a forma do algoritmo é irrelevante, não sendo discutida neste estudo de caso.

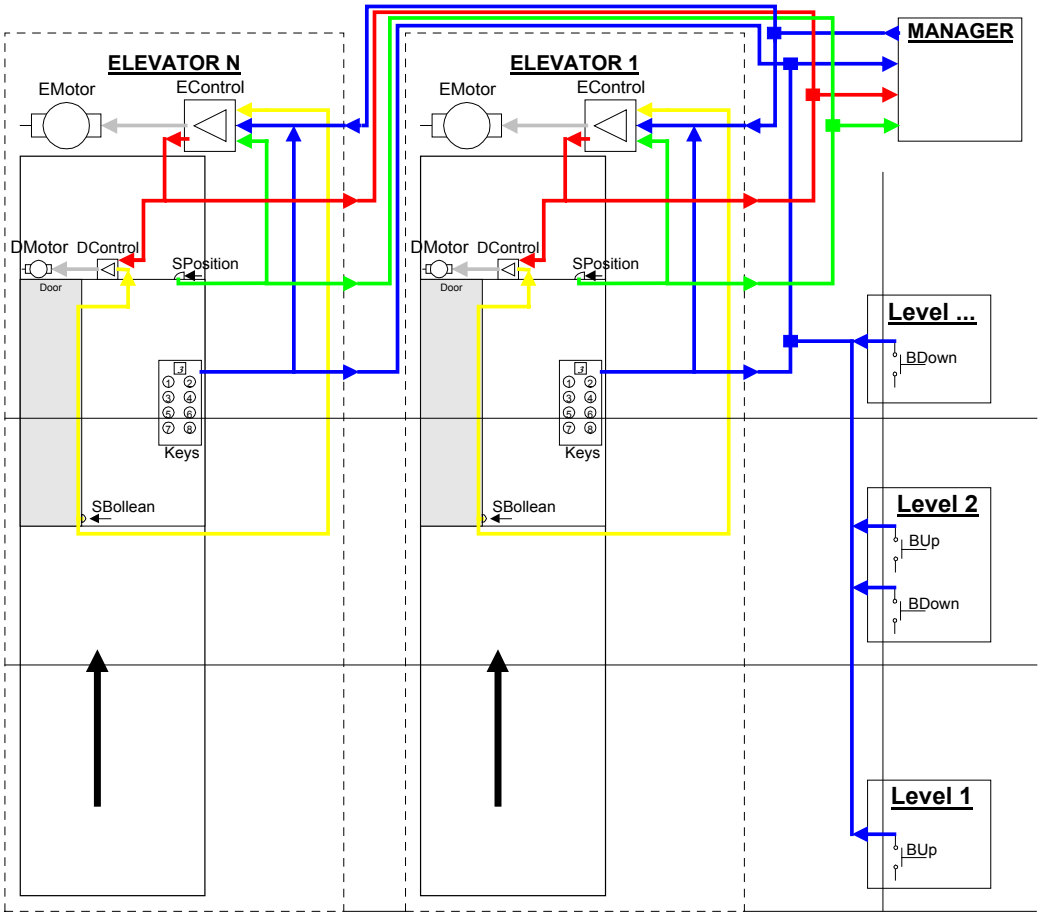

FIGURA 5.11 - Arquitetura do Sistema de Controle de Elevadores

Nesta arquitetura, cada elevador é capaz de funcionar de forma independente do resto da aplicação. Numa aplicação sempre existirá apenas um gerente, ao qual deverão estar conectados 2 ou mais elevadores e 2 ou mais andares. O gerente deve receber todas as requisições, tanto de andares como de elevadores, bem como ele deve saber das condições de cada elevador, os seja, sua posição e sentido atual do movimento. A partir destas informações, o algoritmo de escolha, implementado pelo gerente, irá definir qual elevador deverá atender a uma requisição. Esta interação entre os elementos que compõem a arquitetura pode ser verificada na Figura 5.11, sendo que o elevador possui 4 pontos de iteração (de cima para baixo: Requisições externas, sentido do movimento, andar atual e requisições internas), o gerente possui 4 pontos de iteração (de cima para baixo: Requisições internas, requisições externas, sentido do movimento e andar atual), e o andar possui 1 interação referente às requisições internas do andar.

5.3.2 Modelo de Componentes SIMOO-RT

A Figura 5.12 mostra o modelo implementado no SIMOO-RT do controlador de elevadores. Neste modelo foram mapeados os três principais componentes da arquitetura, o elevador, o nível e o gerente. Para esse estudo de caso foram criados 5 níveis de uma edificação, três elevadores para atender as requisições e o gerente das requisições (Figura 5.12).

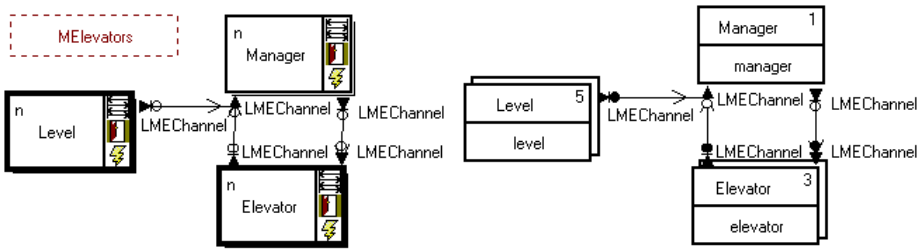

FIGURA 5.12 - Diagramas MET do SIMOO-RT para o Controle de Elevadores

Pelo modelo apresentado pela arquitetura, cada elevador é um sistema completo com capacidade de funcionamento independente. Segundo a arquitetura um elevador é subdividido em três sistemas básicos, o sistema de controle do movimento do elevador, o sistema de controle de abertura e fechamento da porta e o sistema interno de requisição de andares. Os relacionamentos destes três subsistemas podem ser vistos na Figura 5.13.

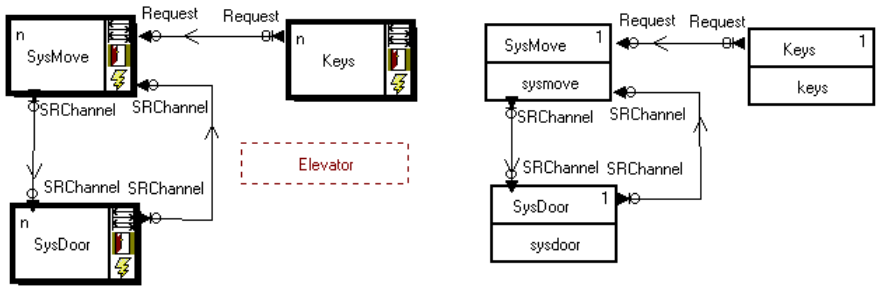

FIGURA 5.13 - Diagramas MET do SIMOO-RT - Composição do Elevador

Como pode ser observado na Figura 5.13 cada subsistema do elevador é formado pela composição de outros componentes. A Figura 5.14 mostra a composição do sistema de

movimento do elevador (*SysMove*), e que esta composição é formada a partir dos elementos da biblioteca de componentes.

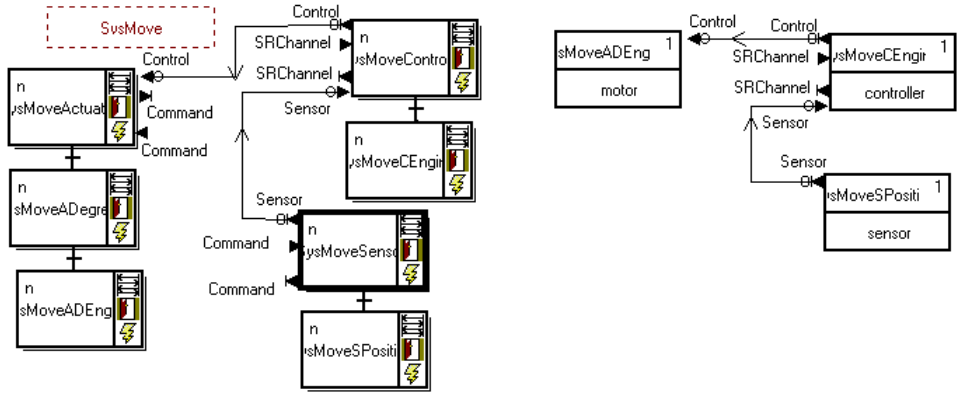

FIGURA 5.14 - Diagramas MET do SIMOO-RT - Composição do SysMove

Para o diagrama de implantação da aplicação (DDE) foram criados 8 nodos, sendo 5 baseados em Motorola/µClinux, executando os componentes dos níveis (*Level*), e 3 baseados em PC/Linux, executando os componentes dos elevadores. O gerente está sendo executado no mesmo nodo do elevador de número 1. Para interligação destes nodos foi utilizado o protocolo CAN. Este modelo implementado pelo DDE do SIMOO-RT pode ser visto na Figura 5.15.

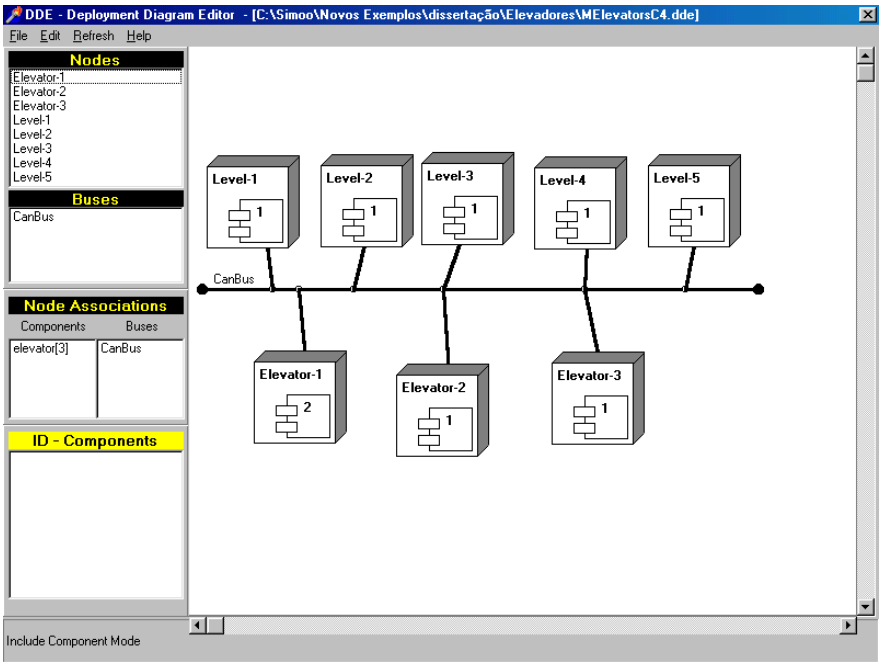

FIGURA 5.15 - Diagrama de Distribuição(DDE) do Controle de Elevadores

Por fim, o resultado do processo de geração de código é a criação de diversos arquivos, sendo 27 com os códigos fonte (22 da aplicação mais o 2 para cada um dos tipos de ECH para os nodos), 27 com os cabeçalhos (22 da aplicação, o ECH e mais o arquivo *Heading.h*), 9 com os arquivos de execução dos componentes (um para cada instância de componente) e 5 arquivos de *makefile* (3 dos componentes existentes e 2 dos tipos de ECHs).

5.4 Estudo de Caso 3: UCRMPF

5.4.1 Arquitetura UCRMPF

Os conceitos da Unidade de Calor Radiante com Monitorização dos Parâmetros Fisiológicos, ou simplesmente UCRMPF, foram desenvolvidos durante pesquisas realizadas pelo Grupo de Engenharia Biomédica da Universidade Católica de Pelotas [VIL99]. A Figura 5.16 mostra o modelo de um equipamento construído segundo estas definições.

FIGURA 5.16 - UCRMPF

O sistema proposto é composto por um microcontrolador que monitora a temperatura (TP), freqüência cardíaca (FC) e freqüência respiratória (FR) de um neonato, e a partir dos valores monitorados o sistema controla um aquecedor (a partir da temperatura) e alarmes sonoros e visuais segundo valores pré-programados (a partir de todos). Na Figura 5.17 é mostrada a arquitetura do UCRMPF, onde é possível identificar sete subsistemas. O primeiro é o de aquisição de sinais bioelétricos, onde são capturados os sinais de TP, FC e FR. Nos sub-sistemas chamados respectivamente de Temperatura, F. Cardíaca e F. Respiratória, são tratadas as questões de monitorização, visualização de valores e acionamento de alarmes da TP, FC e FR. A UCRMPF possui alarmes visuais e o sonoros, sendo um alarme visual tratado de forma independente para cada tipo de sinal bioelétrico, e o alarme sonoro de forma geral por todo o sistema. Para o acionamento destes alarmes é utilizado um controle de máximos e mínimos pré-definidos. O alarme sonoro ainda pode ser reativado quando após quinze minutos do seu desligamento a condição que originou o disparo do alarme ainda persistir. Por este motivo é idealizado um subsistema independente para o acionamento do alarme sonoro. No subsistema de Controle de Fluxo são tratadas as interfaces com o usuário e as regras gerais impostas pelas normas da ABNT (NBR IEC 601-1 e NBR IEC 601-2-21). Por fim o último subsistema é o do controle do aquecedor que é baseado num controle do tipo PID.

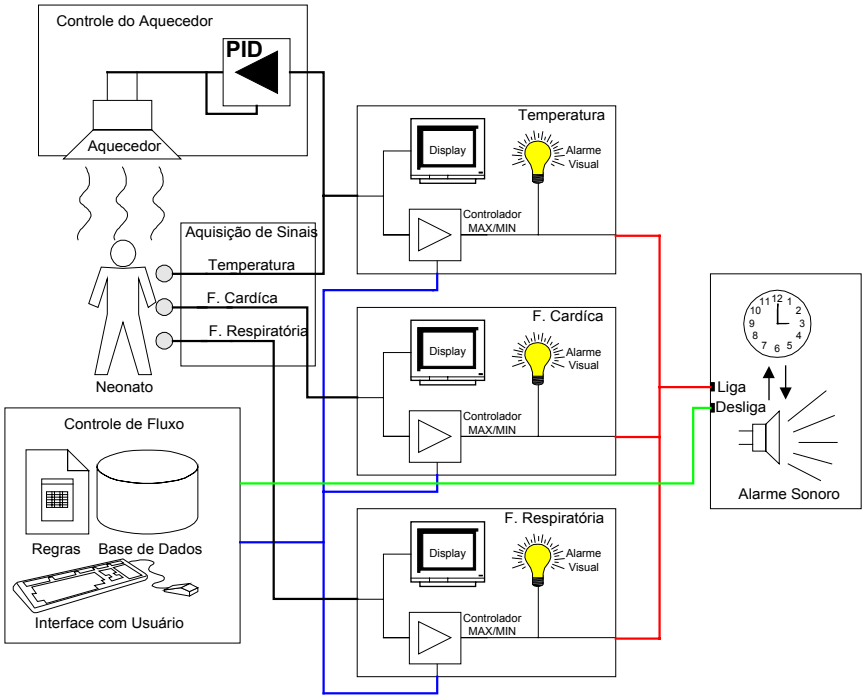

FIGURA 5.17 - Arquitetura de Controle UCRMPF

5.4.2 Modelo de Componentes SIMOO-RT

A Figura 5.18 mostra o modelo do UCRMPF implementado no MET do SIMOO-RT. Neste modelo é possível constatar que a maioria dos elementos da arquitetura da UCRMPF foram mapeados diretamente para um único componente. Por exemplo, os sistemas de TP, FR e FC (*SysTP*, *SysFR* e *SysFC*). Apenas o controle de fluxo teve a sua representação subdividida em dois componentes, o *SysRules* e o *SysUserInput*. Esta subdivisão foi acrescentada apenas para dar mais flexibilidade ao modelo, possibilitando a utilização de mais de um *SysUserInput*. O *SysUserInput* representa um teclado, sendo que no diagrama de instâncias do MET, mostrado na Figura 5.18, foram adicionadas três instâncias dos componentes: a *SysKeys*, responsável pela ativação e desativação da monitorização da TP, FR ou FC; a *ProgKeys*, responsável pelas teclas de programação dos parâmetros dos controladores; e a *CancelSom*, responsável pelo cancelamento do alarme sonoro. Neste mesmo diagrama é possível verificar ainda que foram criadas duas instâncias dos componentes *SysDisplayAux* e *SysSom*. Estas instâncias auxiliares serão utilizadas para a construção de um nodo auxiliar de visualização das informações.

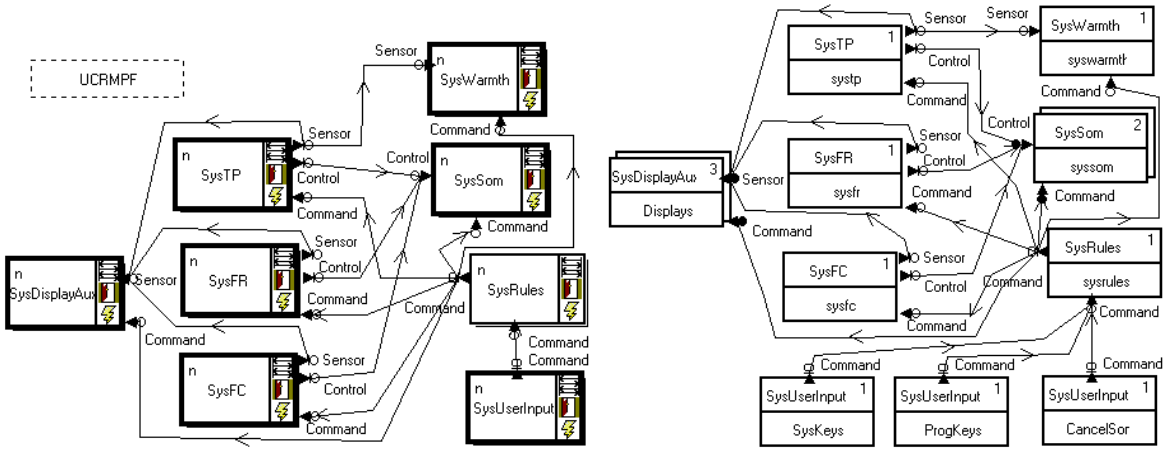

FIGURA 5.18 - Diagramas MET do SIMOO-RT para o UCRMPF

Como pode ser verificado na Figura 5.18, a maioria dos elementos é construída a partir da composição de outros componentes. A mais importante destas composições é a dos sistemas de monitorização da TP, FC ou FR. Na Figura 5.19 é mostrada a composição do componente *SysTP*. Novamente, como nos estudos de caso anteriores, pode-se observar a composição deste componentes a partir de elementos da biblioteca de componentes.

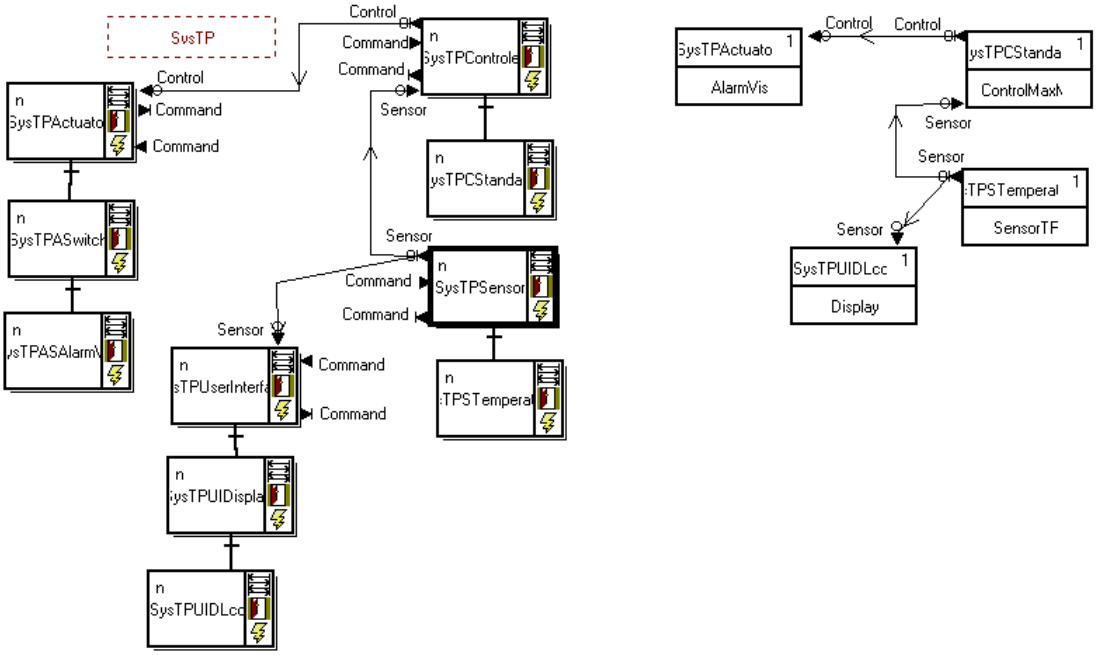

FIGURA 5.19 - Diagramas MET do SIMOO-RT - Composição do *SysTP*

Da mesma forma pode-se observar a composição de outros componentes como o *SysWarmth*, Figura 5.20, e o *SysSom*, Figura 5.21.

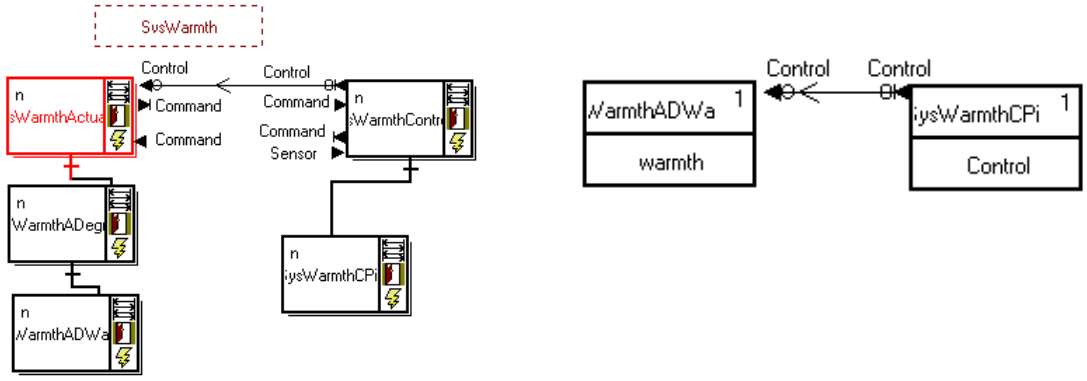

FIGURA 5.20 - Diagramas MET do SIMOO-RT - Composição do *SysWarmth*

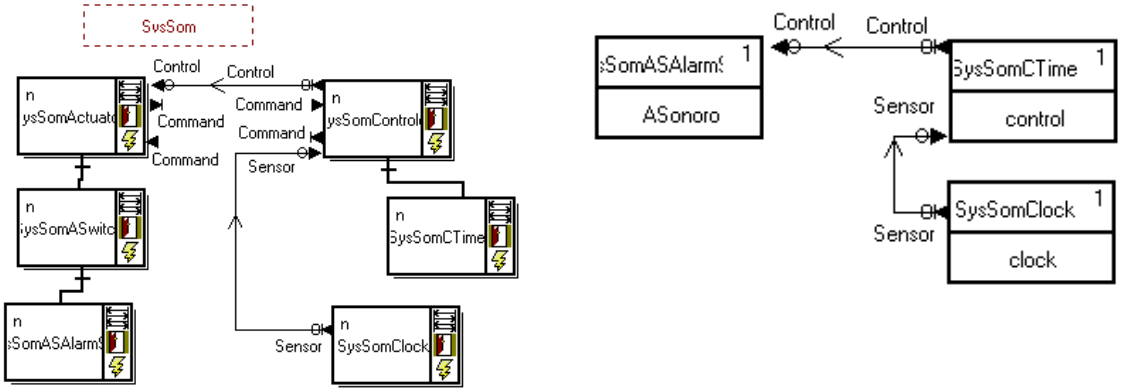

FIGURA 5.21 - Diagramas MET do SIMOO-RT - Composição do *SysSom*

Novamente cada componente deve indicar no método *Start* quais as mensagens ele tem interesse em receber. Assim como ele pode publicar mensagens a qualquer ponto de seu código.

Na Figura 5.22 é possível verificar o modelo de implantação proposto para o UCRMPF. Nele foram adicionados 7 nodos, sendo o nodo *DisplayAux* um PC/QNX com TCP/IP, o nodo *DislplayMain* um PC/Linux com TCP/IP, o nodo Rules um PC/Linux com TCP/IP e CAN, e os demais nodos Motorola/µClinux com CAN.

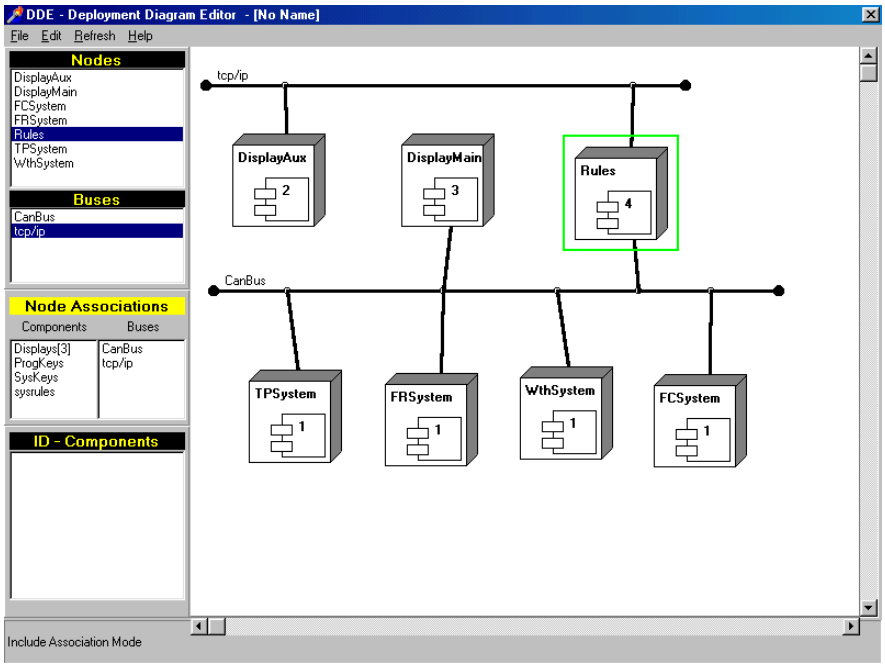

FIGURA 5.22 - Diagrama de Distribuição(DDE) UCRMPF

Por fim, como resultado do processo de geração de código são criados 56 arquivos de códigos fonte (53 da aplicação mais 3 para cada um dos tipos de ECH para os nodos), 55 com os cabeçalhos (53 da aplicação, o ECH e mais o arquivo *Heading.h*), 13 com os arquivos de execução dos componentes (um para cada instância de componente) e 16 arquivos de *makefile* (13 dos componentes existentes e 3 dos tipos de ECHs).

5.5 Análise dos Resultados

5.5.1 Avaliação do Modelo de Componentes Proposto

Através deste estudo teórico e prático do modelo é possível verificar que a descrição destas aplicações é bastante simplificada. Isto ocorre por não haver dependências internas entre componentes das aplicações, e de haver apenas um tipo de relacionamento entre eles, as interfaces. Além disso, a reutilização dos componentes, apoiada na biblioteca de componentes, auxilia na simplificação da construção destes modelos.

Nos estudos de caso deste capítulo pode-se observar que muitos dos componentes utilizados vieram da biblioteca de componentes apresentada no item 4.2. Por exemplo, no

estudo do sistema de controle para elevadores o componentes *SysMove* foi construído apenas com a utilização dos componentes definidos na biblioteca. Isto também ocorre com os componentes *SysTP*, *SysFC* e *SysFR*, do estudo de caso UCRMPF, e o componente *Joint* do Janus.

Outro aspecto importante adicionado à ferramenta é o da simplicidade e flexibilidade proporcionada pela estrutura implementada pelo ECH. Por exemplo, no estudo de caso do UCRMPF, que utiliza diversas arquiteturas para os nodos e mais de um protocolo nos barramentos, não foram necessárias modificações especiais nos componentes. É claro que algumas mudanças podem ser necessárias, principalmente se um componente necessita de recursos especiais do nodo, como por exemplo, um sensor de temperatura que necessita de um conversor AD.

Outro exemplo de flexibilidade adicionada à ferramenta é visto no estudo de caso Janus. Neste estudo são construídos dois *Deployment Diagrams* distintos, que foram criados a partir de um mesmo modelo básico de sistema (Janus). As possibilidades adicionadas pela inclusão deste diagrama são inúmeras. É possível, como o realizado neste estudo de caso, comparar os diagramas em função de características desejadas. Mas também é possível apenas criar novos diagramas, que descrevem ambientes diferentes para uma mesma aplicação.

De qualquer forma, além de possibilitar a descrição dos modelos de implantação de uma aplicação, o DDE oferece ao ambiente SIMOO-RT subsídios para a geração de código dos componentes e do *middleware* de comunicação, simplificando esta tarefa para o programador.

Apesar de ser possível uma descrição completa dos componentes e seus relacionamentos, para que o sistema possa ser mais adequado ao modelo proposto, são necessárias algumas melhorias nos diagramas que descrevem os modelos. Estas melhorias referem-se a alguns poucos ajustes nos diagramas e a mudanças na forma de descrição das interfaces. Atualmente, para manter a compatibilidade com os modelos anteriores do SIMOO-RT, a descrição de cada interface é realizada de forma dispersa pelos diagramas e código gerados pelo ambiente. No estágio atual, para descrever uma interface é necessário

relacionar os métodos com os canais e eventos através de código fonte. Esta descrição é realizada no método de inicialização de cada componente, onde o projetista deve conhecer os nomes corretos dos canais e eventos desejados. O sistema não disponibiliza nenhum tipo de registro global dos canais e eventos utilizados pela aplicação. O ideal seria que houvesse uma ferramenta auxiliar interligada a modelagem do componente, no diagrama de classes do SIMOO-RT. Esta ferramenta deveria permitir a definição de todas as interfaces de cada componente, a partir de um conjunto de interfaces conhecido, ou a construção de uma nova interface. Além disso, esta ferramenta deveria possibilitar que os eventos destas interfaces fossem relacionados aos métodos específicos de cada componente, e que todas estas informações fosse armazenada junto ao modelo.

5.5.2 Comparação dos Resultados

Um outro objetivo desta dissertação é o da comparação entre modelo de objetos e o modelo de componentes proposto, apontando as vantagens e desvantagens deste novo modelo. Esta comparação é baseada apenas nos resultados práticos dos estudos de caso, sendo que foram utilizadas três características básicas para a definição desta comparação: a capacidade de reusabilidade do sistema como um todo ou de suas partes; a facilidade de modelagem; e a performance geral da aplicação.

De uma forma mais conceitual, estas características utilizadas para a comparação já foram discutidas nos capítulos anteriores desta dissertação, servindo inclusive como base da tomada de decisões de implementação da proposta apresentada. Por exemplo, o re-uso de componentes é mais simplificado devido ao fato que este modelo não permite dependências externas de contexto, possibilitando com isso uma maior flexibilidade para a conexão dos componentes. Também por causa desta flexibilidade, o modelo de componente é mais adequado à modelagem de aplicações distribuídas heterogêneas. Já em termos da performance, pode-se afirmar inicialmente que ambos os modelos são totalmente dependentes do *middleware* de comunicação, e que qualquer comparação que se deseje realizar deve ser feita sob um modelo e *middleware* específicos.

Na prática a reutilização tanto de componentes como de objetos, em sistemas de automação industrial, fica restrita a um número limitado destes elementos. Por exemplo,

qualquer um dos elementos dos estudos de caso, tanto para objetos como para componentes, que se envolva diretamente na captação de sinais ou no acionamento de dispositivos, é praticamente impossível construir um componente que seja adaptável a qualquer plataforma de *Hardware/Software*. O melhor que pode-se construir é versões do mesmo elemento para cada plataforma. Mas para os demais elementos, que envolvam as questões de controle de processo e da aplicação, a independência de contexto definida pelos componentes proporciona melhores condições para o re-uso. Um exemplo disso pode ser observado no estudo de caso Janus. No modelo de objetos construído para o estudo de caso realizado em [PER2001], foi necessária uma reprogramação destes objetos, para possibilitar a alteração da localização e do relacionamento entre estes objetos. Já no modelo de componentes (seção 5.2), para a construção dos dois modelos de distribuição (Figura 5.9 e 5.10), não foram necessárias modificações nestes componentes. O exemplo mais claro deste re-uso está nos componentes *Neck*, *LeftArm* e *RightArm*, pois além de mudar a forma do relacionamento com os demais componentes, eles mudaram da plataforma μClinux/Motorola, para uma plataforma Linux/PC. Um outro exemplo de reuso de componentes pode ser visto no estudo de caso do controlador de elevadores. Neste estudo de caso tanto o *SysMove*, como o componente *SysDoor*, foram formados pelo mesmo conjunto básico de componentes: um motor, um controlador e um sensor.

Já para a forma de modelagem das aplicações, o que pôde ser constatado é que ambos modelos permitem de forma similar a definição das aplicações. Isso significa que qualquer característica da aplicação pode ser modelada tanto com objetos, como com componentes. A diferença entre as implementações SIMOO-RT destes modelos dá-se apenas na forma de relacionamento entre os elementos do modelo. Por exemplo, no estudo de caso do UCRMPF no modelo de objetos definido em [VIL99], existem associações entre os objetos *Display* e *SysTp*, ou *Display* e *SysFR*, ou *Display* e *SysFC*, que foram transformadas em interfaces no modelo de componentes. Obviamente esta associação direta dos objetos poderia ser igualmente contornada, mas isto não é o natural no modelo de objetos. O resultado do uso de interfaces ao invés de associações é o de tornar o elemento de distribuição o mais independente do contexto que o cerca, favorecendo a construção de aplicações distribuídas, que estejam baseadas em plataformas heterogêneas. Novamente o melhor exemplo disso são os dois modelos implementados pelo estudo de caso Janus.

A performance dos modelos implementados no SIMOO-RT é uma característica de difícil definição exata. Inicialmente o modelo de componentes implementado adiciona um *overhead* na execução do ECH. Obviamente este fator irá tornar o desempenho na comunicação dos componentes menos eficiente que a de objetos em circunstâncias similares. Mas o principal fator de definição é que a performance está diretamente relacionada ao protocolo de comunicação utilizado.

Em uma comunicação ponto a ponto (TCP/IP), os objetos possuem uma pequena vantagem de não possuírem o *overhead* imposto pelo ECH. Neste caso quando houverem múltiplos clientes (AO/C++) ou consumidores (*Publisher/Subscriber*), cada mensagem deverá ser replicada para cada cliente ou consumidor.

Já em uma comunicação do tipo *Broadcast* (CAN), os componentes possuem uma grande vantagem quando existirem múltiplos consumidores. Isso ocorre porque no modelo de objeto, mesmo sendo um protocolo *Broadcast*, o objeto servidor terá que enviar uma mesma mensagem a cada um dos múltiplos clientes. Já no modelo de componentes, o publicador irá apenas enviar uma única mensagem, e os ECHs de cada nodo irão replicar esta mensagem para cada um dos consumidores.

6 Conclusões e Trabalhos Futuros

Aplicações em sistemas de automação industrial exigem, pela sua complexidade e características especiais, a utilização de técnicas de desenvolvimento de software compatíveis com essas necessidades. O uso de técnicas baseadas no modelo de objetos, conforme a bibliografia sugere, consegue retratar de forma simplificada e eficiente conceitos complexos como encapsulamento, herança, concorrência e a forma de troca de mensagens entre os elementos distribuídos.

O principal problema do uso dos conceitos da orientação a objetos, para a modelagem deste tipo de aplicação, é que os objetos necessitam ter explicitas todas as suas dependências externas. Por exemplo, se um objeto necessita de um serviço, ele necessita saber qual é a classe que possui este serviço e qual a instância desta classe a qual ele estará associado, o que acaba tornando a aplicação menos flexível. A proposta de componentes obriga que estas dependências tenham, além um padrão definido por uma IDL, uma total independência de contexto, o que significa que o cliente irá saber que alguém irá servi-lo, não interessando quem ou onde está este servidor.

Obviamente que esta flexibilidade oferecida pelos componentes obriga a construção de intermediadores na comunicação, que certamente consomem recursos de memória e processamento. Através dos estudos teóricos realizados (seção 2.5), foi possível definir a proposta *Publisher/Subscriber* como a mais indicada para os tipos de aplicações que se desejava modelar. A estrutura do ECH desenvolvida mostrou-se, através dos estudos de caso, flexível, enxuta e eficiente. Flexível pelo fato de não possuir dependências explicitas com a plataforma de *hardware* e *software* utilizada, sendo assim facilmente implementável em qualquer composição desta plataforma. Enxuta, pois a classe ECH é pequena, não consome muitos recursos de memória. Eficiente, pois os algoritmos implementados pela classe não necessitam de muitos recursos de processamento.

A inclusão do DDE (*Deployment Diagram Editor*) aos diagramas do SIMOO-RT completou a descrição deste modelo de comunicação. Nele é possível definir quais são as

características de cada plataforma de *hardware* e *software* utilizada, bem como o local de execução de cada componente da aplicação. Estas funcionalidades do DDE possibilitaram a descrição destes modelos de forma transparente ao programador, eliminando com isso o uso de uma programação direta no código fonte da aplicação.

Um último aspecto de relevância desta dissertação refere-se às características para a reutilização dos componentes. Conforme apresentado nas seções 2.3, 2.6 e 5.5.2, a forma simplificada de re-uso de componentes é uma das principais vantagens do modelo de componentes sobre o modelo de objetos. Neste sentido, a inclusão da biblioteca de componentes, com a definição de uma estrutura básica de componentes interligados para automação industrial, possibilitou uma fácil modelagem de novos componentes, como comprovado pelos estudos de caso.

Dos resultados obtidos com os experimentos realizados pode-se concluir que o modelo de componentes é o mais indicado para a utilização em sistemas distribuídos heterogêneos, e com a adição da biblioteca de componentes e o sistema de geração automática de código, este modelo torna-se mais adequado aos sistemas de automação industrial baseados em plataformas com PC's e microcontroladores. Esta afirmação está embasada nos aspectos de independência de contexto dos componentes, na forma padronizada de relacionamento entre componentes e na forma simplificada de re-utilização de um componente.

Para finalizar, sugere-se como extensão deste trabalho um estudo sobre as formas de definição das características de qualidade de serviço, oferecidas pelas interfaces dos componentes. Obviamente que, para o contexto do SIMOO-RT, as características temporais seriam as mais relevantes. Uma segunda sugestão é a da expansão do ambiente construído, definindo-se novas plataformas de *hardware* e *software*, barramentos de comunicação, componentes para a reutilização e a construção de um conjunto maior de estudos de caso.

Bibliografia

[BEC99] BECKER, L. B. **Ambiente de modelagem e implementação de sistemas tempo real usando o paradigma de orientação a objetos.** 1999. Dissertação(Mestrado) – PPGC da UFRGS, Porto Alegre.

[BEC2000] BECKER, L. B. **Aplicação de Tecnologias de Computação com Objetos Distribuídos no Desemvolvimento de Sistemas Tempo-Real**. 2000. Exame de Qualificação(Doutorado) – PPGC da UFRGS, Porto Alegre.

[BOH95] BOHN, R. A. Semiconductores: An Industry in Transition. **Red Herring Magazine**. Disponivel em <http://www.herring.com/mag/issue23/transition.html>. Acesso em: Set. 1995.

[BOO94] BOOCH, G. **Object Oriented Analysis and Design**. New York. The Benjamin/Cummings, 1994.

[BRU98] BRUDNA, C et al. Sistema para Controle de Elevador em Tempo-Real. In: Seminário Interno do Departamento de Instrumentação e Controle. **Proceddings...** UFRGS, Porto Alegre, [S.l.:s.n.], p. 21-24, 1998.

[BRU2000] BRUDNA, C. **Suporte para Sistemas de Automação Industrial, Baseados em Objetos Distribuídos e no Barramento CAN.** 2000. Dissertação (Mestrado) – CPGEE da UFRGS, Porto Alegre.

[BRO90] BROCK, R. W; WILKERSON, B.; WIENER, L. **Designing Object Oriented Software**. Englewwod Clifs [S.l.:s.n.], Prentice Hall, 1990.

[CAL99] CALDER, C.; SHANNON B**. JavaBeans ™ Activation Framework Specification Version 1.0a**. Sun Microsystems Press, 1999.

[COD88] COULOURIS, G.F.; DOLLIMORE, J. **Distributed Systems :** concepts and design. Wokingham: Addison-Wesley, 1988.

[COP97] COPSTEIN, B.. **SIMOO: Plataforma Orientada a Objetos para Simulação Discreta Multi-Paradigma**. 1997. Tese (Doutorado) - PGCC da UFRGS, Porto Alegre.

[DOU99] DOUGLASS, B.P. **Doing Hard Time -** Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. 2nd ed. Reading, Massachusetts, Addison Wesley, 1999.

[EDD98] EDDON, G.; EDDON, H. **Inside distributed COM**. Redmond, Microsoft Press, 1998.

[FAY2000] FAY-WOLFE, V.; et al. Real-Time Corba. **IEEE Transaction on Parallel and Distributed Systems**. New York, v. 11, n. 10, p 1073-1089, Oct. 2000,

[FRI97] FRIGERI, A. H.; PEREIRA, C. E. **OORTAC:** Um Método de desenvolvimento de Sistemas de Automação em Tempo Real Usando Técnicas de orientação a Objetos. DELET-UFRGS, Porto Alegre, 1997.

[GAM95] GAMMA, E.; et al. **Design Patterns -** Elements of Reusable Object Oriented Software. USA, Addison Wesley, 1995.

[HAR87] HAREL, D. **On Visual Formalisms**. Pittsburgh, Caraegie Mellon University, 1987.

[HAR98A] HARMON, P. Components. **Components Development Strategies.** Arlington, v. 8, n. 7, Jul. 1998.

[HAR98B] HARMON, P. Distributed Component Systems. **Components Development Strategies.** Arlington, v. 8, n. 8, Jul. 1998.

[HAR98C] HARMON, P. Enterprise JavaBeans. **Components Development Strategies.** Arlington, v. 8, n. 9, Jul. 1998.

[HAR98D] HARMON, P. Component Methodologies. **Components Development Strategies.** Arlington, v. 8, n. 11, Jul. 1998.

[HIG2001] HIGHLANDER ENGINEERING INC. **Real-Time Embedded Systems**. Disponível em <http://www.highlander.com>. Acesso em: fev. 2001.

[IBU2001] IBUTTON **Java TINI Systems**. Disponível em <http://www.ibutton.com/TINI/index.html>. Acesso em: fev. 2001.

[JAC92] JACOBSON, I. **Object-oriented software engineering:** a use case driven approach. Wokingham. Addison-Wesley, 1992.

[JAV2000] JAVASOFT. **Java and JavaBeans Technologies**. Disponível em <http://www.javasoft.com>. Acesso em: dec. 2000

[JAV2001] JAVASOFT. **Java RMI Enhancements since JDK 1.2**. Disponível em <http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/index.html >. Acesso em: Out. 1999.

[JOH94] JOHANSEN, D.; RENESSE, R. V. Software Structures for Supporting Distributed Computing. **Distributed Open Systems.** Los Alamitos, [S.l.:s.n.], 1994.

[KAI99] KAISER, J.; MOCK, M. Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN). **Proceedings of the 2nd Int. Symp. on Object-oriented Real-time distributed Computing (ISORC99)**, Saint-Malo, France, [S.l.:s.n.], p. 172-181, May 1999.

[KAU99] KAULER, B. **Flow Design For Embedded Systems a radical new unified object-oriented methodology**. 2nd ed. Lawrence, R&D Books, 1999.

[LEE93] LEE, E.; KALAVADE, A. A Hardware Software codesign Methodology for DSP Applications. **IEEE Design & Test of Computers**. New York, [S.l.:s.n.], Sep. 1993.

[LEE2000] LEE, E. What's Ahead for Embedded Software? **Computer Magazine.** New York, v. 33, n. 9, p. 18-26, Sep. 2000.

[MIC2000] MICROSOFT. **COM and DCOM Technologies**. Disponível em <http://www.microsoft.com/com>. Acesso em: Dec. 2000

[MOC92] MOCHEL, T.; OBERWEIS, A. An Object Oriented Concept for the simulation of Embedded Systems. **Proceddings of Modeling and Simulation – ESM92.** York, [S.l.:s.n.], 1992

[MUL95] MULLENDER, S. **Distributed Systems**. 2nd ed. New York, Addison-Wesley, 1995.

[OKI93] OKI, B.; et al. The information Bus®- An Architecture for Extensible Distributed Systems. **14th ACM Symposium on Operating System Principles**. Asheville, [S.l.:s.n.], p.58-68, Dec 1993.

[OMG2000] OBJECT MANAGEMENT GROUP. Disponível em <http://www.omg.org>. Acesso em: Dec. 2000

[PER94] PEREIRA, C.; DARSCHT, P. Using Object-Orientation in Real Time Aplications: An Experience Report. In **Proc. Of TOOLS EUROPE '94**. Versailles, France, [S.l.:s.n.], 1994.

[PER96] PEREIRA, C.; FRIGERI, A.; DARSCHT, P.; HALANG W. Objet-Oriented Development of Real-Time industrial Automation Systems. **13th World Congress of IFAC.** San Fransisco, [S.l.:s.n.], Jun. 1996.

[PER2001] PEREIRA, C.; et al. On Evaluating Interaction and Communication Schemes for Automation Applications based on Real-Time Distributed Objects. **Proceedings of the 4th Int. Symp. on**

Object-oriented Real-time distributed Computing (ISORC01). Magdeburg, Germany, [S.l.:s.n.], May 2001.

[RUM91] RUMBAUGH, J.; et al. **Object Oriented Modeling and Design.** New Jersey, 1991.

[RUM99] RUMBAUGH, J.; BOOCH G.; JACOBSON I., **The Unified Modeling Language User Guide**. Reading, Massachusetts, Addison-Wesley, 1999.

[SCH2001] SCHLETT, M. **Trends in Embedded Microprocessor Design**. Disponível em <http://www.omimo.be/members/book_schlett.html>. Acesso em: Fev. 2001.

[SCT95] STAROVIC, G.; CAHILL, V.; TANGNEY, B. An Event Based Object Model for Distributed Programming. **In OOIS (Object-Oriented Information Systems) '95**, London, Springer-Verlag, [S.l.:s.n.], p. 72-86, Dec. 1995.

[SEL94] SELIC, B.; GULLEKSON, B.; WARD, P. **Real Time Object Oriented Modelling**. Reading, New York, John Wiley and Sons, 1994.

[SEL99] SELIC, B. Protocols and Ports: Reusable Inter-Objects Behavior Patterns. **Proceedings of the 2nd Int. Symp. on Object-oriented Real-time distributed Computing (ISORC99)**. Saint-Malo, France, [S.l.:s.n.], p. 332-339, May 1999.

[SHL92] SHLAER, S.; MELLOR, S. **Object Life Cycles - Modeling the World in States.** Reading, New Jersey, Yourdon Press, 1992.

[SIE2000] SIEGEL, J. Examine the new CORBA 3 specification. **In:Proceedings of The First Workshop on Real-Time and Embedded Distributed Objects Computing**. Fall Church, [S.l.:s.n.], Jul. 2000.

[SUN2000] SUN. **Java and JavaBeans Technologies**. Disponível em <http://java.sun.com>. Acesso em: Dez. 2000.

[SZY99] SZYPERSKI, C. **Component Software: Beyond Object-Oriented Programming**. Londres, Addison-Wesley, 1999.

[TAK99] TAKAHASHI, D. Embedded Systems Customizable Cores. **Electronic Business Magazine.** Disponível em <http://209.67.253.150/eb-mag/issues/1999/9909/0999emb.asp>. Acesso em: Sep. 1999.

[TAM92] TANENBAUM, A. S. **Sistemas Operacionais Modernos**. Rio de Janeiro, Prentice-Hall do Brasil, 1992.

[VIL99] VILLELA, C.; et al. Metodologia para Desenvolvimento de Software em Sistemas Baseados em Microcontrolador. **XIII Congreso Chileno de Ingeniería Electrica.** Santiago, Chile, [S.l.:s.n.], 1999.

[VIL2001] VILLELA, C.; BECKER, L. B.; PEREIRA, C. E. Framework for Component-Based Development of Distributed Real-Time Systems. **6Th Workshop on Object-Oriented Real-Time and Dependable Systems**. Rome, Italy, [S.l.:s.n.], Jan. 2001.

[VIN99] VINOSKI, S. **Advanced CORBA Programming with C++**. Reading, Massachusetts, Addison-Wesley, 1999.

Ambiente Baseado em Componentes para o Desenvolvimento de Sistemas Computacionais Microcontolados Distribuídos

por

Cláudio Vianna Villela

Dissertação apresentada aos Senhores:

_____
Prof. Dr. Flávio Wagner

_____
Prof. Dr. João Netto

_____
Prof. Dr. Bernado Copstein

Vista e permitida a impressão.
Porto Alegre, ___/___/___.