UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

UM SUPORTE PARA PROGRAMAÇÃO DISTRIBUÍDA EM MÚLTIPLAS LINGUAGENS

Dissertação submetida à Universidade Federal de Santa Catarina para obtenção do grau de Mestre em Engenharia

ELIANE LÚCIA BODANESE

Florianópolis, 12 junho de 1992

UM SUPORTE PARA PROGRAMAÇÃO DISTRIBUÍDA EM MÚLTIPLAS LINGUAGENS

ELIANE LÚCIA BODANESE

ESTA DISSERTAÇÃO FOI JULGADA PARA A OBTENÇÃO DO TÍTULO DE

MESTRE EM ENGENHARIA

ESPECIALIDADE ENGENHARIA ELÉTRICA, ÁREA DE CONCENTRAÇÃO SISTEMAS DE CONTROLE E AUTOMAÇÃO INDUSTRIAL, E APROVADA EM SUA FORMA FINAL PELO CURSO DE PÓS-GRADUAÇÃO

Prof. Joni da Silva Fraga, Dr.
Orientador

p/ Roberto Salgado

Prof. João Pedro Assumpção Bastos, Dr. D'Etat
Coordenador do Curso de Pós-Graduação em Engenharia Elétrica

BANCA EXAMINADORA:

Prof. Rogério Drummond B. P. de Mello Filho, PhD.

Prof. Jean Marie Farines, Dr. Ing.

Prof. Joni da Silva Fraga, Dr.
Orientador

Eraldo Silveira e Silva, M. Eng.
Co-Orientador

A Meus Amados Pais

AGRADECIMENTOS

A Deus, que me deu saúde, oportunidades e o amor de minha família.

A meus pais e irmãos pelo incondicional apoio e compreensão.

Ao Professor Joni da Silva Fraga pela orientação e apoio demonstrados no decorrer do trabalho.

A Eraldo Silveira e Silva pela co-orientação e amizade.

Aos meus colegas pela amizade, e a todos que de alguma forma contribuíram para o enriquecimento deste trabalho.

À UFSC e a CAPES pelo apoio financeiro.

SUMÁRIO

NOTAÇÃO

Nas especificações de sintaxe utilizamos o formalismo derivado da notação BACKUS NAUR FORM estendida, onde são usadas as seguintes convenções:

1. ::= definido por.
2. ¦,(,),[,],+ e * são caracteres especiais.
3. Símbolos terminais são cadeias de quaisquer caracteres limitados por '" "', sem pertencerem ao símbolo terminal.
4. Símbolos não terminais são cadeias de caracteres não especiais.
5. Palavras reservadas são escritas em letras maiúsculas e em negrito.
6. Itens alternativos são separados por uma barra vertical '¦'.
7. Itens opcionais são colocados entre colchetes '[ ]'.
8. Itens são agrupados dentro de parênteses '( )'.
9. Um asterísco '*' após um item significa zero ou mais ocorências deste item.
10. Um sinal de mais '+' após um item significa uma ou mais ocorências deste item.

Para maior clareza, o item ou itens que podem ser repetidos devem ser agrupados entre parênteses.

Notas Léxicas da Linguagem DIM:

1. Comentários começam com '/*' e terminam com '*/'.
2. Espaços em branco servem para separar itens, senão são ignorados.
3. Um identificador é uma letra seguida por uma sequência de letras, dígitos ou "underscores" '_'. Onde existe diferença entre letras maiúsculas e minúsculas, sendo que somente 8 caracteres são significativos.
4. Palavras reservadas são escritas em letras maiúsculas.

5. Uma constante é uma seqüência de um ou mais dígitos decimais, precedidos ou não por um sinal de menos.

Notas Sintáticas da Linguagem DIM:

1. Somente constantes sem sinal podem ser usadas para especificações de arrays. Se um identificador for utilizado, deve ter sido declarado previamente como uma constante sem sinal em uma definição "const".

2. Identificadores de constantes e tipos dentro do escopo de uma especificação devem ser unicamente declarados. Declarações de especificação de estruturas ou uniões criam novos escopos.

RESUMO

O objetivo deste trabalho é fornecer ao programador de aplicações distribuídas, ferramentas que facilitem a programação e integração de componentes de forma independente de linguagens e máquinas. Nesse sentido, são discutidos as necessidades e os aspectos relacionados ao desenvolvimento de um suporte de programação distribuída em múltiplas linguagens para ambientes heterogêneos (SPML).

O SPML é composto de ferramentas de programação e de um suporte de tempo de execução. Duas linguagens integram as ferramentas: uma responsável pela descrição das interfaces de componentes e outra pela configuração da aplicação. O suporte de tempo de execução implementa as abstrações definidas em um modelo de programação distribuída e se utiliza de uma representação de dados padrão para as transferências de informação entre componentes, de modo totalmente independente da heterogeneidade de linguagens e máquinas que caracteriza um ambiente distribuído.

ABSTRACT

The objective of this work is to facilitate the distributed applications programming by providing tools which make the programming independent of languages and machines. The aspects and requirements related to development of a mixed language distributed programming support (SPML) for heterogeneous environments are discussed.

The SPML is composed of programming tools and a run-time support. A components interface description language (DIM) and a configuration language (LINCS) are supplied. A run-time support incorporates the distributed programming model abstractions and uses a standard data representation for information exchanges between components.

CAPÍTULO 1

INTRODUÇÃO

Nos últimos anos tornou-se crescente o uso de ambientes de computação distribuídos, onde programas são alocados a estações de execução e integrados no sentido de atender as necessidades computacionais das aplicações. As aplicações distribuídas tornaram-se mais difundidas e determinaram novas necessidades em termos de modelos e ferramentas, distantes daqueles utilizados em linguagens seqüenciais tradicionais.

Na programação distribuída acredita-se que a melhor abordagem é a de separar as necessidades dos que desenvolvem programas de um componente do sistema daqueles que desenvolvem as aplicações que utilizam estes componentes. Por isso o princípio da Decomposição Modular [DeRemer 76] tem sido largamente defendido para a construção de sistemas distribuídos flexíveis.

Nos últimos anos inúmeras linguagens de programação distribuída têm sido propostas. Algumas com características voltadas para programação em larga escala, mas sem uma separação total da construção de componentes do sistema e da configuração do mesmo (MESA [Sweet 85], PRONET [Maccabe 82], etc.), trazendo limitações de flexibilidade ao sistema construído. Outras possuem uma separação completa entre a programação em pequena e larga escala, tal como a linguagem do CONIC, permitindo a construção de sistemas flexíveis e bastante confiáveis. Porém neste último exemplo, a implementação dos componentes do sistema é realizada através de uma única linguagem de programação. Esse tipo de abordagem ignora esforços anteriores de programação onde softwares se apresentam implementados em linguagens como C, Pascal, LISP, Prolog, etc.

Assim, como uma nova forma de implementação de aplicações distribuídas, vem-se estudando meios de manter as características das linguagens de programação distribuída na programação em larga escala e, trazer a simplicidade das linguagens seqüenciais para a programação em pequena escala, levando-nos então à abordagem da programação distribuída por "múltiplas linguagens".

A programação distribuída por múltiplas linguagens possibilita maior flexibilidade de escolha ao programador, que pode optar pela linguagem que melhor se adapta a determinado módulo ou reutilizar softwares já desenvolvidos. Porém

existem vários obstáculos para se conseguir programar em múltiplas linguagens. Os principais problemas estão ligados a incompatibilidade entre linguagens, trazendo problemas que incluem diferenças de sistemas de tipos, conceituais, de implementação, etc.

Nossa proposta consiste num suporte multi-linguagem para programação distribuída que tenta reduzir os problemas citados, assumindo separação completa da programação em pequena e larga escala.

Em nossa abordagem associamos a cada módulo do sistema uma representação "textual" e uma representação de "interface"; também utilizamos uma linguagem de descrição de tipos e uma representação padrão de dados. A representação textual corresponde ao código do módulo escrito em determinada linguagem (linguagem hospedeiro), ficando a cargo das ferramentas já existentes para esta linguagem o tratamento dos detalhes sintáticos, de geração de código, etc. A representação de interface descreve todos os objetos exportados e importados pelo módulo e as demais informações julgadas pertinentes do ponto de vista global. Todos os tipos descritos na representação de interface, bem como os dados pertencentes a comunicação inter-módulos obedecem a representação padrão. Um compilador converte a representação de interface em um código na mesma linguagem da sua representação textual, fornecendo facilidades de comunicação, sincronização e conversão de dados. Com todos os módulos compilados passamos a fase de construção do sistema utilizando uma linguagem de configuração com características que permitem a construção de sistemas distribuídos flexíveis.

Este trabalho está dividido em cinco capítulos. O capítulo dois apresenta um estudo sobre as características principais da programação distribuída relacionada ao tratamento da heterogeneidade máquina/linguagem. O capítulo três apresenta a descrição de um suporte para programação por múltiplas linguagens para sistemas distribuídos, seus aspectos funcionais e características sintáticas. O capítulo quatro apresenta os aspectos relevantes do projeto do suporte de programação distribuída em múltiplas linguagens (SPML). O capítulo cinco apresenta as conclusões do trabalho e suas perspectivas futuras.

CAPÍTULO 2

PROGRAMAÇÃO EM SISTEMAS DISTRIBUÍDOS

2.1 Introdução

Este capítulo apresenta as principais características da programação distribuída e de alguns sistemas que proporcionam a construção de aplicações em sistemas distribuídos. É ressaltada também a problemática envolvida na programação multi-linguagens em ambientes heterogêneos, apresentando-se um estudo sucinto de sistemas que abordam esse tipo de programação. Por fim, são apresentadas as principais técnicas utilizadas na programação nestes ambientes caracterizados pelas diferenças de linguagens e máquinas.

2.2 Programação Distribuída

Neste item apresentamos a noção de Sistemas Distribuídos e introduzimos as abordagens para programação nestes sistemas.

2.2.1 Sistemas e Aplicações Distribuídos

Encontramos na literatura muitas definições sobre sistemas distribuídos, sendo que estas diferem um pouco a respeito de como ele é constituído. Porém todas concordam num ponto: todo sistema distribuído requer a presença de múltiplos processadores. A definição que utilizaremos neste trabalho é a de [Bal 89]:

"Um sistema distribuído consiste de múltiplos processadores autônomos que não compartilham memória primária, mas cooperam através de envio de mensagens sobre uma rede de comunicação."

A rede de comunicação determina a velocidade e a confiabilidade da comunicação inter-processos, entre processadores, e a distribuição espacial destes processadores. Tradicionalmente uma arquitetura onde a comunicação é rápida e confiável e onde os processadores estão fisicamente próximos é chamada "fortemente acoplada" (ex. redes *transputer*, *hipercube*). Sistemas onde a

comunicação é lenta e pouco confiável entre processadores dispersos são denominados "**fracamente acoplados**" (ex. redes LAN e WAN).

As aplicações distribuídas apresentam características e finalidades diferentes dependendo do tipo de sistema distribuído em que se executam. Uma característica importante é a granularidade de paralelismo das aplicações. A granularidade é dada pela quantidade de processamento entre comunicações. Aplicações que apresentam comunicação freqüente e rápida, possuem uma granularidade fina de paralelismo e são mais adequadas em sistemas fortemente acoplados, uma vez que nos fracamente acoplados o custo do *overhead* da comunicação pode tornar-se proibitivo. Os programas distribuídos que despendem a maior parte de seu tempo em processamento e eventualmente em comunicação, possuem largas granularidades de paralelismo e são mais adequados a sistemas fracamente acoplados, onde a comunicação se caracteriza pela baixa velocidade e pelos atrasos diferenciados.

O hardware distribuído propicia a implementação de aplicações com diferentes graus de descentralização. Por exemplo, algumas aplicações são mais eficientes e confiáveis quando suas funções são separadas em uma coleção de serviços especializados, onde cada serviço utiliza um ou mais processadores de forma dedicada.

2.2.2 Suporte para a Programação Distribuída

Idealmente, o suporte para programação distribuída deve apresentar mecanismos que possibilitem a comunicação e a sincronização entre processos, o mapeamento físico destes e que permitam ações de reconfiguração. Este suporte pode ser fornecido pelo sistema operacional ou por uma linguagem especialmente projetada para programação distribuída.

No primeiro caso as aplicações são construídas como uma coleção de programas seqüenciais, apresentando rotinas de biblioteca responsáveis pelas chamadas ao sistema operacional e que permitem acesso ao sistema de comunicações e outras facilidades (espera por eventos, mudança de prioridade, etc.). Porém, existem problemas nesta abordagem, principalmente quanto a comunicação e a manutenção da aplicação. Em geral, as interfaces de comunicação são complexas, as primitivas de IPC não são uniformes apresentando semânticas diferentes para interações internas e remotas. O controle de estruturas e tipos de dados quando construído sobre base de programas seqüenciais, é inadequado para a programação distribuída. Na maioria dos sistemas, o sistema operacional fornece pouco suporte

para a configuração da aplicação tanto para a construção inicial quanto ao controle posterior.

A outra abordagem utilizada na construção de programas distribuídos é a utilização de uma linguagem: **linguagem de programação distribuída.** Neste caso, é importante que a estrutura da linguagem incorpore conceitos de modularidade, concorrência, comunicação e sincronismo, juntamente com aspectos de configuração e comunicação remota. A verificação de tipos e a realização de outros testes em tempo de compilação, neste caso, também são realizados na configuração do sistema. Outro ponto de interesse é a visão homogênea e consistente em termos de referências de comunicação, tanto para efeito local como remoto. A abordagem por linguagem normalmente torna mais simples e seguro o desenvolvimento das aplicações distribuídas.

2.2.3 Princípio da Decomposição Modular

O princípio da Decomposição Modular (*Programming-in-the-Large vs Programming-in-the-Small*) apresentado em [DeRemer 76] separa o desenvolvimento de softwares complexos em dois níveis que requerem por vezes, linguagens, ferramentas e metodologias diferentes. A idéia central está na distinção das atividades envolvidas entre a programação de componentes de software (Programação em Pequena Escala), geralmente denominados módulos, e a estruturação do sistema através da interconexão destes componentes (Programação em Larga Escala).

A programação em pequena escala envolve todos os aspectos relacionados à implementação de componentes, definindo os recursos fornecidos e utilizados por estes. Na prática, os recursos correspondem a constantes, tipos, estruturas de dados e procedimentos. O módulo é classicamente dividido em interface e corpo. A interface especifica as relações de importação e exportação de recursos e o corpo do módulo corresponde à realização (implementação) destes. Esta separação permite que a implementação e testes de um módulo se processe de maneira independente dos demais, facilitando a manutenção da aplicação.

Um arranjo de componentes constituindo um sistema ou aplicação é conhecido na literatura como **configuração.** A programação em larga escala trata dos aspectos relacionados com o gerenciamento da configuração de um sistema. A linguagem que tratar com aspectos de programação em larga escala, deve fornecer mecanismos que descrevam formalmente as operações de composição entre os

componentes, e no caso de Sistemas Distribuídos, descrever como as estruturas lógicas devem ser mapeadas sobre os recursos físicos. Em tempo de compilação, esta linguagem deve fornecer meios para a verificação da integridade destas composições, tanto a nível das conexões entre componentes, como das possibilidades de realização do mapeamento lógico/físico.

2.2.4 Linguagens de Programação Distribuída

Existe uma grande variedade de linguagens de programação distribuída que apresentam diferentes modelos e características de programação. A grande diversificação é conseqüência dos diferentes ambientes de programação e das características das aplicações alvo.

Citamos aqui, algumas linguagens que se enquadram nas características de programação distribuída discutidas até o momento. São linguagens e sistemas que apresentam características voltadas ao princípio da decomposição modular.

Sistema Mesa

O sistema MESA [Sweet 85] foi um dos primeiros a adotar o princípio da decomposição modular, suportando portanto, uma programação modular com forte verificação de tipos em tempo de compilação. Os aspectos de configuração são resolvidos por um carregador e um ligador em tempo de execução. Embora existam duas linguagens neste sistema, C/Mesa e P/Mesa, para a programação da configuração e dos módulos respectivamente, esta separação não é completa: um módulo em MESA pode conter comandos de criação de instâncias.

Argus

A linguagem de programação Argus [Liskov 87] fornece suporte para tolerância a faltas, utilizando um modelo baseado em transações atômicas. Sua principal característica é os *guardians*: módulos que encapsulam dados e procedimentos. O *guardian* apresenta características de concorrência interna, sendo formado por um processo iniciador, processos gerenciadores, tratadores de exceções, etc. A configuração permite então o mapeamento de *guardians* em processadores do sistema.

Emerald

O sistema **Emerald** [Black 86] apresenta um modelo de programação "baseado a objeto", destinando-se à construção de aplicações distribuídas em ambientes homogêneos. Emerald suporta um modelo a objetos aplicável a todas as estruturas de dados do sistema, desde inteiros até sistemas de arquivos completos, distintos em objetos ativos (contendo processos) e passivos (estrutura de dados somente). Cada objeto Emerald tem quatro componentes: um identificador único no sistema, uma representação (dados locais ao objeto), um conjunto de operações que são chamadas para se executarem sobre a representação do objeto e um processo opcional. Objetos com processos "invocam" outros objetos locais ou remotos no sistema.

O sistema Emerald apresenta dois aspectos importantes: alguns objetos são completamente móveis entre os nós do sistema, mesmo durante uma invocação. O segundo é o aspecto da persistência: um objeto, uma vez criado, existe em um espaço de objetos enquanto existirem referências ao mesmo. Mecanismos são fornecidos na linguagem de modo a aumentar a sua persistência, mesmo diante de *crashes* de nó.

Conic

Um sistema de programação que tem influenciado inúmeros trabalhos em programação distribuída é o ambiente **Conic** [Magee 87]. As aplicações distribuídas no sistema Conic são construídas segundo uma abordagem hierárquica, usando duas linguagens: linguagem de configuração e linguagem de programação. A linguagem de programação é usada na construção de módulos (tipos módulos). Um "nó lógico" corresponde a uma especificação de subsistema, criada a partir de um conjunto de módulos, usando uma declaração de "grupo" da linguagem de configuração. O nó no sistema Conic, constitui-se na unidade de configuração. A linguagem de configuração deve então construir um programa distribuído como uma composição de um conjunto de nós lógicos, interconectados e mapeados na estrutura física do sistema. As interfaces nas abstrações do modelo (nós lógicos e módulos) são definidas através de mecanismos de troca de mensagem. Mudanças dinâmicas na configuração do programa distribuído são realizadas usando o suporte da linguagem de configuração que permite então criar, interconectar e controlar nós lógicos durante a execução do programa distribuído (Configuração Dinâmica [Kramer 85]). Além das linguagens e do suporte de configuração o sistema fornece facilidades de depuração e suporte para execução de programas distribuídos.

2.3 Programação em Múltiplas Linguagens

Na última década surgiram inúmeras linguagens de programação distribuída com diferentes características dependendo dos ambientes e finalidades para os quais foram projetadas. As tentativas em traduzir um considerável número de programas já construídos em linguagens seqüenciais ou concorrentes para estas novas linguagens demonstraram, em muitos casos, serem de alto custo e, para certos fins, os programas ou partes de programas eram mais eficientes quando implementados por linguagens seqüenciais (ex. programas de cálculo de autovalores e autovetores em Fortran e Algol). Por outro lado, a abordagem de programação distribuída, tendo como base uma única linguagem, restringe também a concepção de softwares distribuídos a ambientes homogêneos de programação, o quê de certa forma, implica numa limitação de uso desta abordagem em sistemas heterogêneos e abertos. Então, no final da década de 80, começou-se a pesquisar outras abordagens de programação distribuída que possibilitassem a reutilização de softwares existentes e que fossem mais adequadas às características usuais de heterogeneidade nestes sistemas. Uma destas abordagens é a "Programação Distribuída por Múltiplas Linguagens" [Jones 85] [Hayes 86] [Barbacci 89], também conhecida como "Programação Mista", na qual um programa distribuído é configurado com componentes de software (programas seqüenciais e/ou concorrentes), construídos a partir de diferentes linguagens.

Existem dois tipos de abordagens no desenvolvimento de programação mista. A primeira trata de compor programas complexos a partir de diversos subprogramas escritos em diferentes linguagens. Nesta abordagem, o programador terá como encargo o trabalho árduo da construção de rotinas de interface que farão o mapeamento, em tempo de execução, entre os diferentes conceitos e limitações de implementação envolvidos com as várias linguagens utilizadas. A segunda abordagem, que corresponde a uma solução mais geral e mais adequada do problema, baseia-se na existência de um suporte para programação mista. Nesta abordagem, os subprogramas serão integrados, executando-se em espaços próprios, determinados pelas abstrações definidas pelo suporte (por exemplo, processos); os aspectos ligados a referências ou comunicações externas serão tratados de maneira uniforme pelo suporte. Esta última abordagem é plenamente adequada a Sistemas Distribuídos com características de heterogeneidade, sendo neste contexto que se insere este trabalho.

2.3.1 Programação Mista: Principais Obstáculos

Existem vários obstáculos para se conseguir programar Sistemas Distribuídos com várias linguagens. O trabalho do programador é árduo e de custo considerável, resultando na maioria das vezes em implementações de difícil evolução e pouco seguras. Por isto, vários grupos de pesquisa vem buscando formas de simplificar a participação do programador de um software complexo em uma aplicação distribuída, principalmente em ambientes heterogêneos.

Para abordarmos os problemas envolvidos na programação multi-linguagem, mostramos a seguir os tipos de dificuldades [Einarsson 84] e exemplos que as esclarecem.

Incompatibilidade entre Linguagens

Em primeiro lugar destacamos os problemas associados à **incompatibilidade entre linguagens**. Neste caso as dificuldades advêm essencialmente das diferenças existentes nos conceitos e modelos de linguagens. Os exemplos são bastante comuns:

1. Estrutura de dados: Neste caso, estruturas de dados ou tipos definidos em uma linguagem não encontram correspondência em uma outra linguagem. Exemplo: Uma função Fortran não pode retornar uma variável complexa a um procedimento Pascal, porque em Pascal não existe tal tipo.

2. Diferenças conceituais: Neste caso os modelos de linguagem assumem diferentes abstrações para o mesmo recurso. Como exemplo podemos citar as abstrações definidas para entrada e saída ou ainda serviços de arquivos. Exemplo: A linguagem Fortran assume que as operações de entrada/saída ocorrem sobre registros discretos, em outras linguagens estas operações se efetuam sobre byte streams.

3. Diferenças semânticas na passagem de parâmetros: Algumas linguagens não suportam mais do que um tipo de acesso ao dado armazenado, como na semântica copy in/copy out. Linguagens como o C, por exemplo, suportam apenas semânticas do tipo "chamada por valor", dificultando desta maneira, a chamada por outras linguagens que utilizam a semântica de "chamada por referência" e que não possuem um operador de endereço (Fortran, Pl/1 e Pascal Standard).

4. Diferenças em ligações: As referências de endereço em linguagens são tratadas diferentemente. Em algumas as ligações são dinâmicas e em outras estáticas. Os arquivos, por exemplo, podem ser abertos e fechados dinamicamente

durante a execução do programa em determinadas linguagens, em outras estes são manipulados em tempo de compilação.

5. Diferentes formas de tratar exceções: Quando em linguagens diferentes ocorre a mesma exceção, geralmente os suportes de cada linguagem estipulam ações diferentes para tratar o mesmo tipo de exceção.

6. Diferentes aspectos conceituais: Outros aspectos conceituais podem evidenciar diferenças entre linguagens, dificultando a programação mista, podemos citar entre estes:

- Linguagens fortemente tipadas, com atribuições de tipos de forma estática e linguagens não tipadas ou com alguma forma de polimorfismo (herança, *generic*, etc.), permitindo de forma mais flexível uma atribuição dinâmica de tipos.

- Diferenças em modelos de comunicação e sincronismo em linguagens com suporte para concorrência. Os modelos "chamada de procedimento" (semáforos, monitores) são certamente incompatíveis com modelos "troca de mensagem".

Implementações Incompatíveis de Linguagens

Outro tipo de dificuldades é conseqüência de **implementações incompatíveis** de linguagens:

7. Diferenças em representações: As diferentes formas que as linguagens armazenam e alinham os dados de determinados tipos. Como exemplo, podemos citar a linguagem Fortran que armazena *arrays* multidimensionais por colunas, enquanto outras linguagens os armazenam por linhas.

8. Dependências dos suportes subjacentes: As diferenças, em se tratando de implementações de linguagens, refletem também as características diferenciadas dos serviços de camadas subjacentes. Estas características são reflexos principalmente das diferenças no gerenciamento de memória e nisto, podemos incluir as diferentes políticas e mecanismos para áreas de *heap*, pilhas, níveis de indireção usados no endereçamento e alocação de memória virtual. Estas características dos serviços subjacentes determinam diferenças nos suportes e compiladores das linguagens, implicando por vezes, no envolvimento explícito do programador na definição das necessidades para execução de seu programa.

9. A granularidade do uso de linguagens em programação mista: A granularidade na qual linguagens podem ser usadas em programação mista implica em limitações. Se o uso destas na programação de componentes se limita ao nível de *procedures*, isto pode não ser realizável para compiladores que não suportem compilação separada.

Diferenças nas Arquiteturas das Máquinas

10. Diferenças de representação de dados em diferentes arquiteturas: Outros problemas são devidos a existência de ambientes heterogêneos. Por exemplo se construirmos um programa produtor/consumidor utilizando *pipes* no sistema operacional UNIX e escrevemos os dados, digamos os inteiros 0 1 2 3 4 5 em uma máquina SUN, e realizarmos a leitura em um VAX obteremos:

0 166777216 33554432 50331648 67108864 83886080,

este problema ocorre devido as diferenças na ordenação de bytes nas duas máquinas. Logo, para que máquinas diferentes compartilhem dados é necessário a "portabilidade" dos mesmos, ou seja, que exista uma representação externa para os dados compartilhada pelas diferentes máquinas. Os aspectos envolvendo transformações entre a representação externa e representações internas de máquinas, devem ser perfeitamente conhecidos e facilmente realizáveis.

O próximo item explicita as soluções encontradas por diversos grupos de pesquisa para o problema da programação mista.

2.4 Estudo de Casos

Apresentamos a seguir alguns suportes de sistemas desenvolvidos para facilitar a programação em múltiplas linguagens.

2.4.1 Durra

Durra [Barbacci 86] [Barbacci 89] é uma linguagem de programação em larga escala, projetada para suportar o desenvolvimento de aplicações distribuídas em ambientes heterogêneos. As "tarefas" (componentes) da aplicação são escritas em diferentes linguagens (ex. C, Ada) e podem se executar sob diferentes sistemas operacionais (ex. UNIX, VMS) e diferentes arquiteturas de máquina (ex. VAX, SUN). Isto é conseguido através do conceito de **descrição de tarefa** e **declaração de tipos.**

Um programa distribuído especificado em Durra é um conjunto de declarações de tipos e descrições de tarefas que descrevem a configuração e aspectos de execução em uma rede heterogênea. As **declarações de tipos** definem a estrutura dos dados produzidos ou consumidos pelas tarefas, seguindo uma sintaxe particular. A declaração de tipos abrange um conjunto de tipos que vai de uma seqüência de bits de tamanho fixo ou variável aos tipos mais complexos como arrays multidimensionais e registros de tipos mais simples.

As descrições de tarefas fornecem informação sobre:

A Interface: com declarações de portos que especificam a direção e o tipo de dados que se movem através do porto.

Atributos: especificam várias propriedades da tarefa. Ex. autor, número da versão, linguagem, tipo de processador, nome do arquivo, etc.

Informação de Comportamento: especifica propriedades funcionais e de tempo sobre a tarefa.

Estrutura: descreve a estrutura interna da tarefa. Contendo declarações de processos, de filas, ligações e declarações de reconfiguração.

Para descricão da aplicação e de tarefas compostas de outras tarefas mais simples utiliza-se a mesma descrição de tarefa. A diferença é que a descrição de uma aplicação (ou uma tarefa composta) possui a parte de estrutura e não possui o atributo de implementação; para uma tarefa simples ocorre o contrário.

Uma transformação de dados é necessária quando os tipos dos portos de entrada e saída são incompatíveis. As transformações devem ser escritas como tarefas separadas e instanciadas como processos. Isto permite sanar as diferenças entre linguagens e máquinas.

A configuração de uma aplicação é realizada através de um conjunto de comandos de alocação de recursos e de escalonamento, resultantes da compilação das descrições de tarefas que compõem a aplicação. Na configuração o executivo Durra carrega as implementações de tarefas nos processadores e de acordo com os comandos executa os programas. No ambiente de execução Durra existem dois componentes ativos: as tarefas da aplicação e o executivo Durra.

A comunicação entre tarefas é realizada via executivo através de uma interface por mensagens (troca de mensagens). A ativação de primitivas de serviço do executivo (também uma tarefa) pelas tarefas de aplicação se dá por meio de

mecanismos de RPC. Estes serviços incluem operações como: pedir identificadores de portos, testar o conteúdo das filas associadas aos portos da tarefa,etc. Do ponto de vista de um código em linguagem hospedeiro (tarefa de aplicação), a comunicação entre tarefas e os serviços relacionados são conseguidos através de uma interface troca de mensagens, formada por primitivas presentes no mesmo espaço de endereçamento (mesma tarefa).

Para finalizar podemos dizer que o objetivo principal da linguagem Durra é ser uma linguagem de programação em larga escala, geral o suficiente para permitir a programação de aplicações em ambientes heterogêneos.

2.4.2 Mixed Language Programing (MLP)

O suporte MLP apresentado em [Hayes 86],[Hayes 88],[Hayes 89] constrói programas distribuídos tendo como base mecanismos de RPC (*Remote Procedure Call* [Nelson 81]). O suporte MLP está fundamentado sobre um Sistema de Tipos Universal (*Universal Type System* - UTS), formado por uma representação padrão e uma linguagem de expressão de tipos. A linguagem UTS é usada para descrever as interfaces dos diversos componentes (procedimentos) do programa. A representação padrão de dados especifica o formato dos dados transferidos como argumentos ou resultados entre procedimentos de forma independente de linguagem e de máquina. Um dado representado neste formato é conhecido como um valor UTS.

Um programa em MLP é constituído de componentes, cada um possuindo procedimentos escritos em uma linguagem, conhecida como a **linguagem hospedeiro** do componente. Uma descrição de interface escrita em linguagem UTS, traz o identificador e os tipos dos parâmetros de cada procedimento importado ou exportado por um componente.

As especificações de interface são analisadas por um gerador de código conhecido como "*stub generator*". O conceito de *stub* foi introduzido por [Nelson 81] como suporte para RPC. Os *stubs* devem encapsular todos os detalhes de conversão e comunicação remota durante uma chamada de procedimento remoto, mantendo para o código chamador as mesmas características de uma chamada local. Para cada linguagem hospedeiro, o *stub generator* gera procedimentos ou *stub procedures* que atuam sob cada rotina importada ou exportada do componente. Os *stubs* no contexto MLP, são rotinas que além de converterem os dados locais em tipos UTS e vice-versa (funções de "serialização" e "deserialização"), servem de

interface entre o código do usuário e o sistema de tempo de execução do MLP. Os subprogramas e os *stubs* de cada componente são compilados e ligados à biblioteca do MLP através dos compiladores de cada linguagem hospedeiro, produzindo arquivos executáveis.

As rotinas de conversão de dados nos *stubs* são responsáveis pela conversão automática entre tipos UTS e da linguagem hospedeiro que possuam mapeamento direto. Nos tipos UTS subespecificados e aqueles não suportados pela linguagem hospedeiro, que não podem ser traduzidos diretamente para um valor na linguagem, são usados os tipos *representatives*. *Representatives* são tickets ou capacidades para valores UTS. Os *representatives* são declarados na linguagem hospedeiro como variáveis de tipo REPRESENTATIVE, onde nenhuma operação é permitida sobre estas variáveis, a não ser quando manipuladas por rotinas especiais da biblioteca UTS. Estas rotinas UTS, presentes nos *stubs*, são então responsáveis pela conversão dos valores associados ao *representative* para valores suportados pelo UTS. Também são usados *representatives* na passagem de procedimentos como parâmetros, existindo rotinas especiais que chamam um procedimento que tenha sido passado como parâmetro.

A comunicação entre processos é executada usando o mecanismo de RPC, com dados no formato UTS. Uma funcionalidade pertencente ao suporte de tempo de execução MLP, chamado mlp_server, é responsável pelo envio e recepção de chamadas e pela coordenação dos *stubs*.

A característica principal do sistema MLP é a simplicidade. Permite o uso de muitas linguagens para a programação distribuída, fornecendo tipos que são comuns a maioria das linguagens seqüenciais existentes. Os pontos semânticos mais refinados são deixados para o programador, ajudado por um esquema flexível de procedimentos pré-definidos pelo sistema.

2.4.3 Matchmaker

Matchmaker [Jones 85] é uma ferramenta para especificação de interface e geração de código para permitir comunicações interprocessos, em um ambiente heterogêneo, independentemente de linguagem. Esta ferramenta é um suporte para o núcleo do sistema operacional *Accent* [Rashid 81] que forma a base do ambiente SPICE [Dannenberg 82] desenvolvido na Universidade Carnegie Mellon (Pittsburgh USA).

As interfaces em Matchmaker definem toda a comunicação inter-processos, apresentando diversas semânticas e utilizando a noção de portos para o envio e recepção de mensagens. Esta abordagem permite servir também a linguagens baseado a objeto.

Matchmaker possui também um conjunto de tipos próprios a fim de obter uma representação padrão de dados.

Em resumo, este suporte fornece uma linguagem de descrição de interface, utilizada para construir uma especificação de interface de módulos. Cada especificação é traduzida de acordo com a linguagem de implementação do módulo, fornecendo como saída um código de conversão, recepção e envio de mensagens. Os códigos resultantes de todos os componentes da aplicação se comunicam pelo núcleo *Accent* na execução da aplicação distribuída. Matchmaker se propõe a ser uma ferramenta de fácil utilização.

2.4.4 Horus Stub Generator

Horus é um sistema RPC para ambientes heterogêneos, desenvolvido nos laboratórios da *Hewlett-Packard*, que utiliza uma linguagem de especifição de interface onde o programador escreve declarações da interface remota, contendo definições de tipos e declarações de procedimentos remotos. Baseado nesta declaração de interface um gerador de código, denominado *stub generator* [Gibbons 87], produz um subprograma para o servidor do serviço, chamado *server stub*, e produz também subprogramas de clientes chamados, *client stubs* . Estes *stubs* contêm código para converter argumentos e para chamar e monitorar o envio de mensagens sobre a rede. O gerador também produz cabeçalhos de arquivos que definem os tipos especificados nas descrições de interface que são utilizados pelos procedimentos nas linguagens hospedeiro.

Em tempo de execução, a chamada ao serviço remoto é interceptada por um *stub* cliente, este converte (serializa) os argumentos da chamada em uma representação externa (sintaxe de transferência), envia a mensagem ao servidor e espera pelo retorno da mensagem. O *stub* servidor recebe a mensagem, converte (deserializa) os argumentos na representação interna do servidor (sintaxe interna) e chama o servidor. No retorno do serviço, o caminho inverso é percorrido pelos resultados, envolvendo os mesmos *stubs* e os processos correspondentes de conversão de representações. O *stub* cliente ao deserializar a mensagem de

resposta, retorna o resultado ao cliente. A figura 2.1 ilustra uma chamada de procedimento remoto entre C e Pascal.

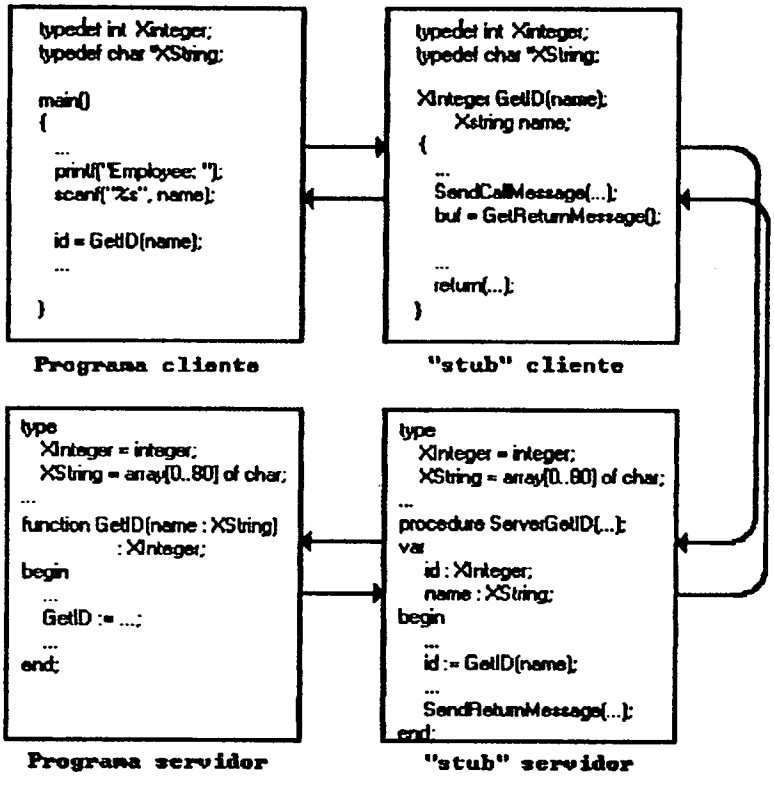

Fig. 2.1 Chamada RPC entre Componentes em C e Pascal

Para resolver problemas de heterogeneidade, Horus implantou sua representação de dados externa, que define uma sintaxe padrão de tipos de dados comuns a linguagens e máquinas. Os algoritmos de conversão estão envolvidos nos *stubs*, ou seja, estes convertem os argumentos na linguagem hospedeiro para a representação padrão, e vice-versa.

O gerador de código é único e independente das linguagens e das máquinas. Toda a informação sobre construção de linguagem e formato dos dados de máquina é armazenada em especificações. Uma **especificação de linguagem** contém um conjunto de instruções que especificam completamente tudo que o gerador deve saber sobre a linguagem, para gerar os códigos de serialização e deserialização entre a linguagem especificada e a sintaxe de transferência. Uma **especificação de máquina** contém todo o conhecimento dependente da máquina necessário para a conversão de argumentos. Assim para a execução de uma aplicação que utiliza L linguagens de programação e M tipos de máquinas, na entrada do gerador de stubs haverão L especificações de linguagem e M especificaçoes de máquina e mais toda informação de interface fornecida pelo usuário, para a criação de todos *stubs* necessários.

Todo o processo de criação e manutenção de aplicações são realizadas pelo ambiente Horus.

2.5 Comentários e Comparações entre as Abordagens que Tratam do Problema da Heterogeneidade Linguagem-Máquina

Das propostas para permitir a programação distribuída com múltiplas linguagens, percebemos a dificuldade no tratamento dos problemas inerentes a estes sistemas e que as soluções implementadas são reguladas por uma relação de custo-benefício. Porém, todas foram unânimes em alguns pontos: a necessidade da separação entre o corpo e a interface do componente de software (subprograma), a criação de especificações de interface onde todas as informações de interesse global são inseridas, e a necessidade de uma padronização na representação externa de dados, durante a comunicação entre componentes. As soluções na bibliografia envolvem, neste sentido a definição de um sistema de tipos próprio e a utilização de uma representação externa de dados padronizada. Essas medidas conseguem contornar a maioria dos problemas citados no item 2.3.1.

A utilização das especificações de interface difere pouco entre as abordagens: todas utilizam-na como elemento de entrada para a geração de um código de conversão de tipos e dados e algumas também a utilizam para propósitos de configuração da aplicação distribuída.

A conversão dos dados na maioria das propostas, baseia-se em diferentes implementações de *stubs* (mecanismo para comunicação RPC [Birrell 84]), que consistem de códigos dependentes da linguagem hospedeiro a que estão ligados.

A geração dos *stubs* e suportes é similar nos sistemas descritos: um tradutor recebe a especificação de interface e gera o código na linguagem hospedeiro, a partir do conhecimento necessário desta linguagem e da máquina em que se executará o componente de software ligado ao *stub*. A principal diferença entre as diversas proposições da literatura, encontra-se na obtenção do conhecimento sobre as interfaces e nas necessidades de entrada do gerador dos códigos que realizam a serialização. A MLP possui um gerador para cada linguagem hospedeiro e o conhecimento necessário para a geração de código é intrínseco. Matchmaker e Horus possuem um único gerador para todas as linguagens. Horus recebe o conhecimento nécessário através de uma especificação de linguagem e de máquina e Matchmaker utiliza uma abordagem de compilação capaz de produzir código para muitas linguagens sem o uso de especificações.

Nosso trabalho se assemelha em muitos aspectos apresentados nesta análise e serão abordados no capítulo três. A tabela 2.1 descreve sucintamente os mecanismos, as vantagens e as diferenças entre as diversas soluções para a programação mista encontradas na literatura.

Sistemas	Mecanismo de comunicação	Linguagem de especificação de interface	Sintaxe de transferência	Geração automática de algoritmos	Heterogeneidade de ling./máquina	Vantagens	Desvantagens
Durra	Troca de mensagem	Descrição de Tarefa	Não completa	Não	Possível, porém parcela a cargo do programador	Generalidade	Compatibilidade dos dados na comunicação é deixada ao programador
MLP	RPC	Expressões UTS	UTS	Sim. Um gerador para cada linguagem	Sim	Simplicidade de utilização	Não é tão abrangente p/ programação distribuída quanto os demais
Match-maker	RPC síncrono e assíncrono	Especificação Matchmaker	Própria	Sim. Único gerador p/ todas linguagens	Sim	Várias semânticas de sincronização para RPC	Mudanças e extensões no gerador de código são mais difíceis.
Horus	RPC	Declaração de Interface	Própria	Sim. Único gerador p/ todas linguagens	Sim	Todo conhecimento sobre lings. e máqs. separado em especificações	Usando especificações leva mais tempo para gerar stubs de uma decl. de int.

Tabela 2.1 Quadro Comparativo Entre os Sistemas Descritos

Um outro aspecto que convém salientar é a diferença entre os suportes para programação distribuída em ambientes heterogênenos e as facilidades existentes em protocolos de comunicação. No segundo caso, o usuário é em seus programas (processos de aplicação) envolvido com a ativação das primitivas da camada de apresentação, no sentido de converter seus tipos na sintaxe de transferência. No primeiro caso, onde se inclui este trabalho, os algoritmos de conversão de tipos são gerados automaticamente e farão parte do suporte dos programas distribuídos, deixando os algoritmos de aplicação distantes dos problemas ligados com a heterogeneidade de ambientes distribuídos.

CAPÍTULO 3

UM SUPORTE PARA A PROGRAMAÇÃO DISTRIBUÍDA EM MÚLTIPLAS LINGUAGENS (SPML)

3.1 Introdução

Este capítulo primeiramente resgata a experiência obtida no desenvolvimento de software para aplicações distribuídas através do ADES - Ambiente de Desenvolvimento e Execução de Software Distribuído [Fraga 89], descrevendo sua metodologia e modelo de programação.

Com o conhecimento adquirido na execução de várias aplicações utilizando este ambiente, verificamos a necessidade de trazer facilidades de programação em múltiplas linguagens para o mesmo. Abordamos então uma evolução do ADES - um sistema que mantém suas principais características trazendo porém, a concepção de um suporte de programação multi-linguagem.

Assim, o novo sistema é descrito, detalhando todos seus aspectos funcionais e toda a sintaxe envolvida nas descrições de interface de módulos e configuração do sistema.

3.2 O Sistema ADES

O Ambiente de Desenvolvimento e Execução de Software Distribuído - ADES concebido no LCMI-UFSC, é resultado de um trabalho de cinco anos de pesquisa e foi construído com o intuito de permitir o desenvolvimento de aplicações distribuídas em tempo real, segundo uma metodologia baseada no princípio da decomposição modular [DeRemmer 76]. O ambiente de execução consiste de um conjunto de estações interconectadas por uma rede de comunicação. Uma das estações do sistema, denominada Estação de Trabalho, engloba as funções de desenvolvimento e configuração do sistema, as demais são chamadas de Estações de Execução, nas quais o software é carregado e executado. Um suporte de tempo de execução, o núcleo de tempo real - NTR [Nacamura 88], é responsável pela implementação do ambiente multi-tarefas distribuído que executa a aplicação. A comunicação e a sincronização entre tarefas apresenta uma interface uniforme, independente da distribuição das tarefas no sistema.

Modelo de Programação

Uma aplicação distribuída é constituída por um conjunto de módulos interconectados entre si. O módulo encapsula uma ou mais tarefas que constituem as unidades básicas de concorrência. A visibilidade externa de um módulo e mesmo de uma tarefa é fornecida por portos. As ligações de portos de saída a portos de entrada formam os canais de comunicação e sincronização, onde dados são importados e exportados. Os canais de comunicação são definidos externamente aos módulos de maneira a tornar a configuração completamente independente do comportamento interno dos componentes elementares do sistema. Outras características do modelo são:

- Uma configuração pode conter várias instâncias de um mesmo módulo.
- Mudanças na configuração podem ser obtidas em tempo de execução (configuração dinâmica).
- Um módulo pode ser programado para alterar a sua configuração interna. As tarefas podem modificar em tempo de execução suas ligações e estados, reconfigurando o seu módulo internamente.

As abstrações definidas no modelo de programação são implementadas com a linguagem LIS - Linguagem de Interconexão de Sistema, composta de duas sub-linguagens:

- Uma Linguagem de Componentes Elementares (LINCE) [Silva 88] que permite a definição de tipos módulos. A LINCE é uma linguagem procedural que contém as construções da linguagem Pascal, acrescida de extensões para comportar o modelo multi-tarefas.
- Uma Linguagem de Configuração de Sistemas (LINCS) que possibilita a criação de instâncias de tipos módulos, interconexão e o mapeamento dos mesmos nas unidades de processamento [Souza 88] e [Abreu 91].

A programação de aplicações distribuídas realizada através da linguagem LIS é bastante flexível e permite o desenvolvimento rápido e seguro de aplicações distribuídas.

3.3 Proposição do Suporte para Programação Distribuída em Múltiplas Linguagens (SPML)

Desenvolvendo aplicações no ambiente ADES, verificamos a existência de um grande número de programas, já desenvolvidos nas diversas áreas do laboratório, que poderiam compor diversas aplicações, porém estes programas foram implementados em linguagens tradicionais e forçosamente necessitariam ser traduzidos para a linguagem LINCE, a fim de integrar alguma aplicação distribuída. Desta forma, para facilitar a construção de componentes de software em aplicações distribuídas a partir de ambientes heterogêneos de programação, direcionamos a pesquisa para o desenvolvimento de um novo sistema, baseado na experiência adquirida com o ADES e nas pesquisas realizadas sobre a problemática da heterogeneidade linguagem-máquina em ambientes de programação distribuída.

O trabalho aqui proposto visa construir um suporte de programação multi-linguagem composto por:

Ferramentas de Programação:

- Uma linguagem de programação de interface que permitirá ao programador descrever as interfaces de componentes de software do sistema. A tradução de uma descrição de interface deve gerar arquivos contendo todas as informações sobre o componente julgadas pertinentes do ponto de vista global (portos, mensagens e atributos). Estes arquivos serão usados para a configuração da aplicação distribuída e pelo suporte de tempo de execução.

- Uma linguagem de configuração que permitirá descrever o programa distribuído em termos de uma composição de componentes. As informações geradas na tradução desta descrição permitirão, através do suporte de tempo de execução, o carregamento, a instanciação e a interconexão de componentes, caracterizando então a configuração da aplicação distribuída no sistema.

Suporte de Tempo de Execução:

- O suporte deverá implementar as abstrações definidas no modelo de programação distribuída. As interfaces dos serviços de comunicação e sincronismo deverão atender os requisitos clássicos de uniformidade, garantindo a total transparência (e independência) da distribuição dos códigos de aplicação no sistema.

- O suporte de tempo de execução deve se utilizar de uma representação padrão de dados para as transferências de informação entre componentes, de modo totalmente independente da heterogeneidade de linguagens e máquinas que caracteriza o ambiente distribuído. As rotinas que farão as conversões entre representações locais e de transferência, serão parte do suporte e estarão codificadas na mesma linguagem do componente correspondente (linguagem hospedeiro).

3.3.1 Organização do Sistema e Paradigma de Programação

Inicialmente está prevista a construção do sistema sobre uma rede heterogênea, composta de um conjunto de estações dos tipos PC e Estações de trabalho SPARC-SUN, interconectados por uma rede Ethernet. Mantém-se as noções de Estação de Trabalho e Estações de Execução, presentes no ambiente ADES, com o executivo distribuído, utilizando-se das facilidades fornecidas pelo sistema operacional UNIX*.

O modelo de programação é similar ao do ambiente ADES. Mantém-se a visibilidade através dos portos dos módulos e os mecanismos de comunicação e sincronização.

O módulo constituí a unidade básica de configuração e de concorrência; cada instância de módulo corresponde a um único processo, o que é próprio para o caso de linguagens hospedeiro seqüenciais. As necessidades de concorrência interna na implementação de módulos, usando linguagens concorrentes, podem ser atendidas desde que o conceito de processo fornecido pelo suporte de tempo de execução, apresente subunidades internas de processamento (*threads*). A figura 3.1 ilustra o paradigma de programação.

Os próximos itens descrevem as principais características do suporte, segundo dois aspectos: **estrutura funcional do Suporte de Programação Distribuída em Múltiplas Linguagens (SPML)** e a **visão do usuário.** Os aspectos funcionais mostram como os diversos componentes do sistema interagem, desde a construção de cada módulo até a execução da aplicação. Na visão do usuário SPML, são tratados todos os aspectos relativos a programação de uma aplicação distribuída: a descrição de interface de módulo e a especificação da configuração da aplicação.

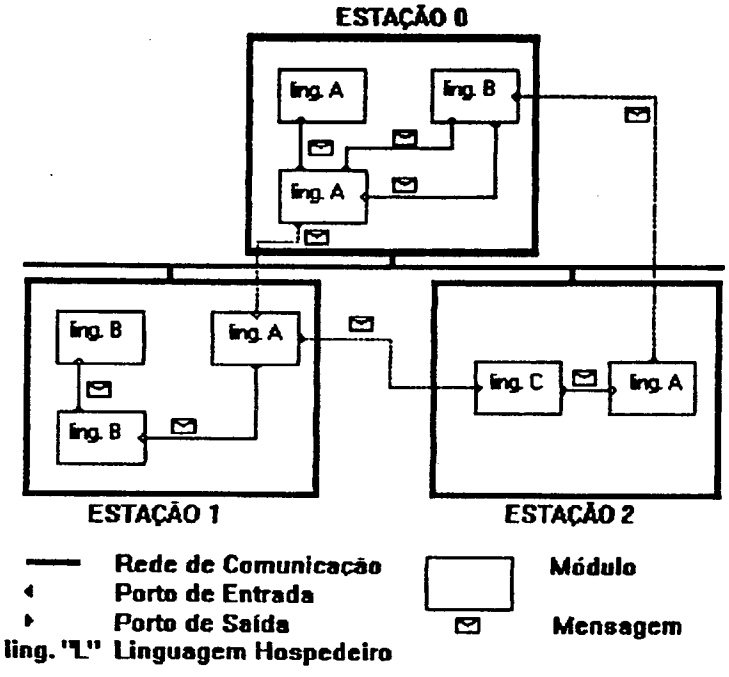

Fig. 3.1 Paradigma de Programação do SPML

3.4 Aspectos Funcionais do SPML

A construção e execução de programas distribuídos segundo a abordagem do SPML se utiliza de ferramentas e suportes envolvidos em três fases distintas: construção dos módulos, configuração e gerenciamento em tempo de execução.

Construção dos Módulos:

Inicialmente cada módulo da aplicação possui duas unidades de compilação, uma contendo o código responsável por parte da aplicação, em linguagem escolhida pelo programador (linguagem hospedeiro), a qual chamaremos de **implementação de módulo**, e a outra que fornece a interface referente a este código, a **descrição de interface**. Cada descrição de interface passa por um tradutor especial que gera os procedimento denominados **Rotinas de Interface**, responsáveis pelo tratamento da heterogeneidade linguagem-máquina e pelo acesso às primitivas de comunicação do suporte de tempo de execução (**Interface de Serviços**), e ainda um arquivo de dados a ser utilizado pelo processador LINCS na configuração. O código executável correspondente a um tipo módulo é o resultado da edição de *link* dos seguintes códigos objeto: Código da implementação de módulo, Rotinas de Interface, Interface de Serviços e da biblioteca da linguagem hospedeiro. A figura 3.2 mostra as etapas da construção dos módulos.

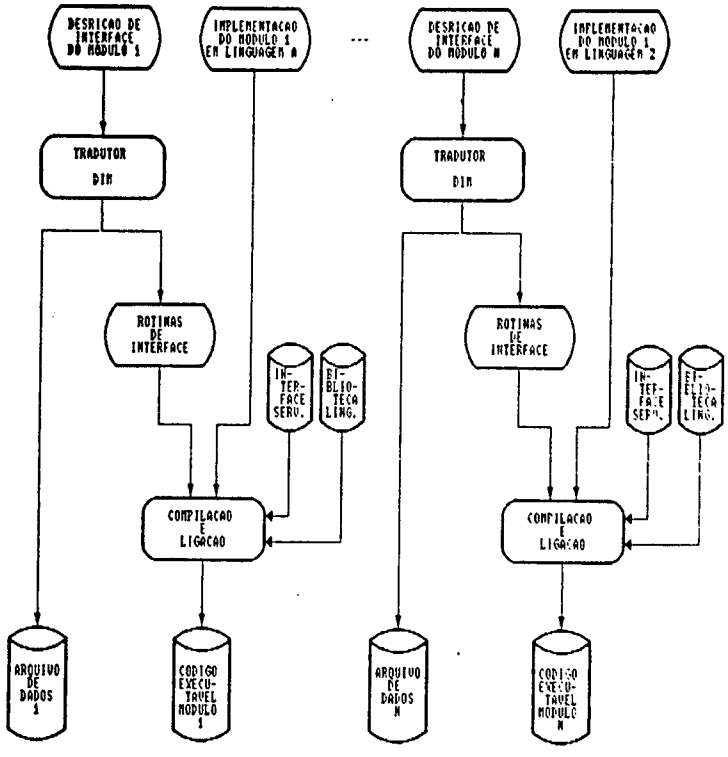

Fig. 3.2 Construção dos Módulos

Configuração:

A configuração de uma aplicação é descrita através de uma declaração System em linguagem LINCS do sistema ADES [Abreu 91]. Esta descrição permite: identificar os módulos da aplicação, determinar a criação de instâncias de módulos, interconectar estas instâncias através de canais de comunicação e sincronização e também informar em que entidades físicas serão executadas estas instâncias.

A tradução da declaração System da linguagem LINCS resulta em uma tabela de configuração para cada estação do sistema, e em uma base de dados do sistema. Estes resultados fornecem a utilitários do sistema (construtor) as diretivas para a configuração. Após a tradução, o carregador realiza a carga dos tipos módulos e do suporte de tempo de execução, nas estações de execução do sistema. Seguindo as informações da tabela de configuração da estação, uma função de iniciação do suporte de tempo de execução, ativa as instâncias de módulos, estabelece os canais de comunicação da aplicação distribuída e, transfere o controle para o escalonador da estação (suporte de tempo de execução da estação), finalizando então o processo de configuração estática. A figura 3.3 ilustra o processo de configuração.

Execução:

O suporte de tempo de execução implementa as abstrações do modelo de programação (módulos, portos, etc.), fornecendo os mecanismos necessários para a concorrência e comunicação no sistema. Cada módulo corresponde a um processo que é escalonado segundo uma política de pre-empção, baseada em eventos e com prioridades estáticas.

O suporte de tempo de execução, depois de instalado, fornece também serviços de remoção de módulos, destruição de instâncias e desconexão de portos. Estes serviços são úteis principalmente no caso de configuração dinâmica.

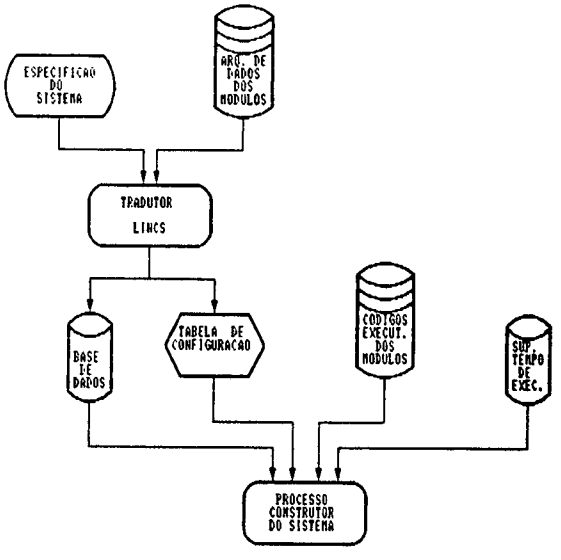

Fig. 3.3 Configuração do Sistema

Durante uma comunicação o processo emissor de uma mensagem aciona a operação de envio através de uma chamada à Rotina de Interface correspondente. Esta rotina executa a conversão dos dados para o formato da sintaxe de transferência e ativa a primitiva de comunicação do suporte (através da Interface de Serviços). Da mesma forma, no processo destino, o acionamento da primitiva de recepção deve ativar a Rotina de Interface para a conversão dos dados da representação externa para a sintaxe local.

3.5 Programação de Aplicações Utilizando o SPML

O sistema separa as atividades da programação em pequena e larga escala de uma aplicação. A programação em pequena escala compreende basicamente a construção de módulos e se divide em duas partes: A **implementação do módulo** em qualquer linguagem hospedeiro suportada pelo SPML e a sua **descrição de interface**, a DIM - Descrição de Interface de Módulo, escrita utilizando uma sintaxe específica. A programação em larga escala consiste na **configuração da aplicação**, utilizando uma linguagem de configuração de sistemas.

3.5.1 Implementação do Módulo

Para construir um módulo é o programador quem define a linguagem hospedeiro que melhor se adapta a tarefa a ser implementada ou ainda, simplesmente reutilizar um programa já codificado em uma determinada linguagem, realizando pequenas alterações.

As primitivas de comunicação do suporte de tempo de execução são os pontos de ligação entre a implementação do módulo e sua descrição de interface. Referências no código de implementação do módulo às funções de envio e recepção (Rotinas de Interface), implicam na ligação das primitivas de comunicação do suporte de tempo deexecução com o código do módulo. Como nas Rotinas de Interface estão embutidos os algoritmos que tratam da conversão entre a sintaxe local e a sintaxe de transferência e vice-versa, a codificação destas deve também ser realizada na linguagem hospedeiro, no sentido de facilitar estas manipulações de conversão. A idéia básica é portanto, que para cada linguagem hospedeiro seja gerada uma interface nesta mesma linguagem.

Para comportar todos os tipos de comunicação suportados pelo sistema, esta interface deve apresentar procedimentos ou funções para o envio assíncrono, envio síncrono, recepção simples e seletiva; operações originalmente presentes no sistema ADES. A figura 3.4 ilustra as possíveis combinações destas operações no sentido de definir diferentes tipos de canais de comunicação. Estas operações possuem parâmetros com as seguintes características em comum:

- A mensagem é passada por referência.
- A mensagem deve ser uma variável simples ou estruturada, cujo tipo é compatível com o do porto.

- Para identificar o porto, seu nome é passado através de uma variável do tipo cadeia de caracteres.

- O tamanho da mensagem corresponde ao número de bytes definidos pelo seu tipo.

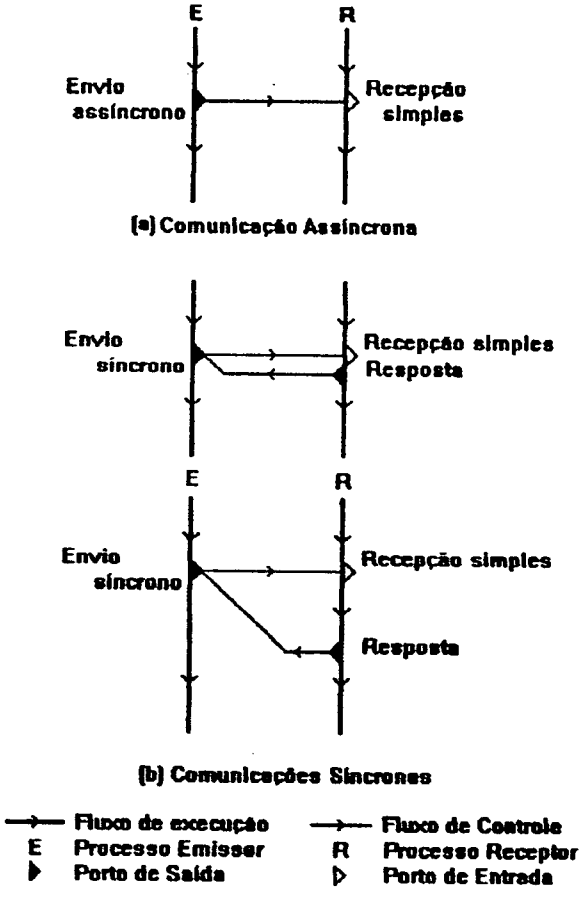

Fig. 3.4 Combinação de Operações de Comunicação

Para se obter a generalidade necessária em ambiente multi-linguagem, as operações de envio e de recepção correspondem a chamadas de funções ou procedimentos, identificadas igualmente por todas as linguagens e possuindo mesmo número de parâmetros. Uma sintaxe informal é definida para cada operação de comunicação, presente no sistema:

A. Envio Assíncrono

envass(nome_porto, end_msg, tam_msg, falha)

O envio assíncrono especifica o porto de saída que transmitirá a mensagem, o endereço da mensagem, seu tamanho e uma condição de falha.

Nesta operação, o fluxo da comunicação é unidirecional e o processo continua a sua execução logo após o envio. Este tipo de envio permite um maior grau de paralelismo entre os processos envolvidos, porém apresenta uma menor confiabilidade que o envio síncrono, uma vez que não é sinalizado no processo emissor o sucesso ou não da transferência assíncrona. O único tipo de sinalização de excessão fornecido no processo emissor é a de "falta de ligação" (porto não ligado) que é realizado através de um argumento de retorno que denominamos "falha". Um valor 1 indica a existência apropriada do canal de comunicação; e 2 indica a detecção da inexistência do canal de comunicação (porto de saída não conectado): seguindo os valores deste parâmetro, o programador pode, se desejar, executar uma ação de tratamento de exceção. Abaixo são apresentadas as sintaxes possíveis para a chamada de envio assíncrono em diversas linguagens:

```
Em C:          envass(porto,&msg,sizeof(msg),fail);

Em Fortran:    CALL ENVASS(PORTO,MSG,N,FAIL)
                      onde: N é o número de bytes ocupados por MSG.

Em Pascal:     envass(porto,msg,SizeOf(msg),fail):
                      onde: msg é definida como var;
```

B. Envio Síncrono

envsin(nome_porto, end_msg, tam_msg, timeout, end_rsp, tam_rsp, falha)

Além dos parâmetros comuns às operações de comunicação, o envio síncrono apresenta parâmetros relativos à resposta esperada e ao sucesso da operação.

A mensagem de resposta deve ser uma variável de tipo compatível ao do porto síncrono de saída. No envio a mensagem é passada por referência, juntamente com seu tamanho em *bytes*.

O argumento de retorno "falha" é uma variável inteira que identifica dois tipos de excessões: esgotamento de tempo e inexistência de ligação no porto de saída. Quando se deseja que o processo não espere indefinidamente por uma resposta, especifica-se uma constante inteira positiva no argumento "*timeout*". Assim se o tempo de espera pela resposta ultrapassar o período pré-estabelecido, o argumento "falha" retorna com valor 0, indicando o esgotamento de tempo. O

programador pode, no tratamento da excessão, reativar o processo bloqueado, tomando os devidos cuidados quanto às consequências desta reativação. O valor -1 para o argumento "timeout" é utilizado para inibir esse mecanismo, isto é, fazer o processo esperar indefinidamente pela resposta. A deteção de falta na ligação é análoga à descrita na operação anterior, excetuando-se apenas que o retorno de "falha" com valor 1 indica o sucesso da operação. A codificação desta operação em diferentes linguagens é apresentada abaixo:

```
Em C:
penvsin(porto,&msg,sizeof(msg),-1,&rsp,sizeof(rsp),fail);

Em Fortran:
CALL ENVSIN(PORTO,MSG,NMSG,2,RSP,NRSP,IFAIL)
                onde:  NMSG é o número de bytes ocupados por MSG.
                 NRSP é o número de bytes ocupados por RSP.

Em Pascal:
envsin(porto,msg,SizeOf(msg),1,rsp,SizeOf(rsp),fail)
                onde: msg e rsp são declaradas como var;
```

C. Recepção

recebe(nome_porto, end_msg, timeout, 0, 0)

A recepção possue cinco argumentos: O nome do porto de entrada que recebe a mensagem, a especificação de uma localização de memória onde será armazenada a mensagem, o tempo de espera máximo para a recepção e ainda outros dois parâmetros que serão utilizados somente na recepção seletiva. O parâmetro "*timeout*" é um valor inteiro positivo na existência de espera de tempo e o valor -1 para indicar espera indefinida. Exemplos de sintaxe da chamada **recebe** são mostradas abaixo:

```
Em C:
recebe(porto,&msg, -1, 0, 0);

Em Fortran:
CALL RECEBE(PORTO,MSG, 10, 0, 0)

Em Pascal:
recebe(porto,msg, 0, 0, 0);
                onde: msg é declarada como var;
```

D. Recepção Seletiva

A recepção seletiva, apresentada como mecanismo de codificação de não determinismo, está presente no suporte de tempo de execução, com duas funções que permitem a programação de recepções seletivas:

recebe (nome_porto, end_msg, timeout, sel, clausula)

A função **recebe** habilita à recepção em um subconjunto de portos determinados na recepção seletiva. É possível implementar condições de guarda em cada porto (através de expressões booleanas) permitindo ou não a execução da função **recebe**, ou seja, habilitando ou não o porto à recepção. A função **recebe** habilita somente o porto para recepção da mensagem e retorna ao programa chamador, sem que a recepção propriamente dita tenha sido executada. Todos os portos habilitados por funções recebe dentro da recepção seletiva são colocados em uma fila.

Os três primeiros argumentos são utilizados da mesma forma que na recepção não seletiva. A existência de várias partes elegíveis de tempo de espera faz com que o menor seja escolhido. O argumento sel é uma variável inteira que possui valor 1 significando que a recepção é seletiva. O argumento final é uma cláusula que numera cada declaração recebe dentro de uma recepção seletiva.

recepção_sel(porto);

A função **recepcao_sel** percorre a lista de portos habilitados e verifica a existência ou não de mensagem. Ao encontrar a primeira mensagem realiza a recepção propriamente dita. Porém se não existe mensagem na fila, esta função programa uma espera até o fim do menor *timeout* fornecido nas funções **recebe**. Se o menor *timeout* for zero, o processo deve continuar imediatamente a sua execução; se todos tempos de espera são iguais a -1 o processo permanece em espera até a chegada de uma mensagem. O argumento de retorno "porto" é uma *string* cujo valor indica o porto de entrada de onde a mensagem procede. Se no retorno da função, este argumento for nulo temos duas possibilidades: esgotamento de *timeout* (recepção bloqueante) e inexistência de mensagem (recepção não bloqueante).

O exemplo em linguagem C a seguir, ilustra o uso destas funções em uma recepção seletiva. No exemplo, verificamos que o porto A está guardado (comando if). O próximo conjunto de portos é guardado por um comando de escolha, habilitando um porto do conjunto. O porto B não possue comando de guarda e por isso sempre é habilitado. Após a habilitação dos portos é realizada a recepção da mensagem e a identificação do porto de onde a mensagem procede, desencadeando a ação definida pelo programador para aquela recepção.

```
...
     strcopy(portoA, "porto_A")
     strcopy(portoB, "porto_B")
     strcopy(porto1, "porto_1")
     strcopy(porto2, "porto_2")
     strcopy(porto3, "porto_3")
...
     if (a == 2) {
          recebe ( portoA. &msgA, -1, 1, clausula);
          clausula ++;
     }
     switch (c) {
         case 1: /* instrucao
                    recebe (porto1, &msg1, -1, 1, clausula);
                    clausula ++;
                    instrucoes ... */
                    break;
         case 2: /* ... */

                    break;
...
         case n: /* instrucao
                    recebe (porton, &msgn, -1, 1, clausula);
                    clausula ++;
                    instrucoes ... */
                    break;
         default:/* Nao se deseja recepcao em nenhum porto*/
                    break;
     }
     recebe(portoB, &msgB, 10, 1, clausula);
     recepcao_sel (porto);
     switch (porto) { /* instrucoes para cada porto,
                 o default pode ser utilizado para tratamento de
                 excecoes (esgotamento de tempo) */
     }
```

E. Resposta

resposta(nome_porto, end_msg)

Na comunicação síncrona, após o processo receptor haver completado a recepção, este deve responder imediatamente ou após um bloco de instruções, fazendo uso da operação resposta. Esta primitiva está vinculada à recepção e o programador deve atentar cuidadosamente para esse fato, a fim de evitar erros.

O primeiro argumento corresponde ao nome do porto de entrada que recebeu a mensagem. Isto é possível já que a resposta deve seguir a mesma via da mensagem recebida. Em seguida é fornecido o endereço da mensagem de resposta que deve ser uma variável de tipo compatível ao especificado pelo porto.

Exemplos:

```
Em C:
resposta(porto,&rsp);

Em Fortran
CALL RESPOSTA(PORTO,RSP)

Em Pascal:
resposta(porto,rsp);
                onde: rsp é declarada como var;
```

3.5.2 Descrição de Interface de Módulo (DIM)

A Descrição de Interface de Módulo é uma unidade de compilação que define as propriedades externas da implementação do módulo. A DIM possui uma sintaxe própria para especificar as seguintes informações:

- Identificação do módulo
- Informações de interface
- Atributos

As informações de interface definem os portos do módulo usados para comunicação, através de declarações de portos. A declaração de um porto especifica a forma de sincronismo possível, a direção e o tipo dos dados da mensagem que são veiculados sobre este porto. As informações de atributo descrevem propriedades do módulo.

Como as interfaces dos módulos devem ter uma notação independente de linguagem e de máquina, adotamos um sistema de tipos padrão na DIM que engloba a maior parte dos conjuntos de tipos apresentados nas linguagens correntes. Certos aspectos foram ponderados para a escolha do conjunto de tipos e da representação padrão dos dados, os quais apresentamos no próximo item.

3.5.2.1 Definição de Uma Notação para a DIM

Para o desenvolvimento de uma especificação de interface, onde se deseja uma linguagem de descrição de interface que forneça um denominador comum entre as linguagens de programação dos diversos ambientes da aplicação, alguns passos devem ser seguidos:

1. Selecionar um razoável conjunto comum de tipos de dados. Estes tipos de dados em linguagem comum poderiam, por exemplo, incluir tipos como booleano, *cardinal*, inteiro, real, enumerado, *string*, *array*, *record*, *sequence* e apontador.

2. Determinar como e para que estes tipos podem ser combinados, levando em consideração as linguagens hospedeiros e as características de comunicação.

3. Adotar uma notação sintática para definir tipos envolvidos nas interfaces de módulos.

Em um ambiente heterogêneo uma comunicação trata com três tipos de representação de dados: sintaxes fonte e destino (ambas representações locais) e a sintaxe de transferência (representação externa). Estas sintaxes são representações ditas concretas e, possivelmente diferentes, dos mesmos dados.

As interfaces dos módulos, envolvidas com os dados e tipos compartilhados no sistema heterogêneo, devem ser descritas usando uma sintaxe **abstrata**, que abstraia o programador dos formatos das três sintaxes concretas citadas acima. A linguagem DIM é, neste sentido, introduzida como notação única que permite descrever as três sintaxes concretas, independentemente de linguagens e de máquinas.

A linguagem de Descrição de Interface de Módulo (DIM) é então a notação usada para especificar as estruturas de dados presentes nas interfaces dos módulos do SPML. No sentido de facilitar a sua implementação, a linguagem DIM incorpora a notação XDR - *External Data Representation* [Sun Microsystems 90] definida no contexto da rede Arpanet [McQuillan 77]. Desta forma, as descrições de interfaces se utilizarão de tipos e regras presentes na notação XDR, mais extensões próprias às necessidades impostas pelo paradigma de programação adotado.

Fatores Determinantes para a Escolha do XDR

XDR inclui tipos como: inteiro, real, booleano, enumerado, *string*, *array*, estruturado, união, apontador e outros como *void* e *opaque data*. Estes tipos são suficientes para formar um conjunto entre as linguagens usuais. A similaridade destes com os tipos da linguagem C permite também certas facilidades de programação em ambiente UNIX.

Outro fator importante para sua escolha foi a possibilidade de conhecer e aproveitar ferramentas de conversão de dados disponíveis em ambiente UNIX, que facilitam a conversão de tipos XDR em tipos de linguagens como o C. A representação interna nestas ferramentas dos tipos XDR serve de sintaxe de transferência. Isto tudo diminui as necessidades de implementação, na construção do suporte.

O emprego da linguagem ASN1 - *Abstract Sintax Notation One*, definida pela ISO e CCITT, foi cogitado. Esta notação se difunde cada vez mais nas descrições

de formatos (sintaxe abstrata) dos protocolos da camada de aplicação no modelo OSI/ISO [ISO 87a] [ISO 87b]. Também a sintaxe de transferência definida no contexto do ASN1 é recomendada como representação externa, para uso nos protocolos de apresentação nas especificações MAP [McGuffin 87]. Porém, o uso do ASN1 como base da DIM foi abandonado devido às dificuldades em se obter os últimos adendos onde se encontra a definição do tipo ponto flutuante e outras questões pendentes quanto à sua normalização. Dentro deste aspecto, o fato da notação XDR apresentar uma completa documentação e ferramentas já disponíveis em ambiente UNIX, foi decisivo para nossa escolha.

A sintaxe XDR completa, que fará parte das especificações DIM, é apresentada no apêndice B e maiores detalhes podem ser obtidos em [Sun Microsystems 90].

3.5.2.2 Sintaxe DIM

No desenvolvimento de uma aplicação, após a decomposição modular, o programador identifica, para cada módulo, os portos de comunicação e transcreve na notação DIM os tipos de dados das mensagens a serem veiculados nestas interfaces. A transcrição deve ser realizada com cautela, uma vez que não há verificação da compatibilidade entre os tipos da linguagem de implementação do módulo (linguagem hospedeiro) e a linguagem de Descrição de Interface de Módulo.

A seguir especifica-se passo a passo toda a sintaxe das declarações envolvidas na DIM, que permitem a instalação de um programa distribuído em um ambiente heterogêneo de linguagens.

Linguagem de Descrição de Interface de Módulo:

1 - Palavras Reservadas e Identificadores Pré-definidos

A Descrição de Interface de Módulo apresenta o seguinte conjunto de palavras reservadas:

ATTRIBUTE, ENDMODULE, INCLUDE, IN, MODULE, PORT, OUT, REPLY.

Os identificadores pré-definidos na DIM correspondem aos mesmos reservados na sintaxe XDR não podendo ser usados de outra forma. Sintaticamente são limitados por aspas '" "' (símbolos terminais), correspondendo a:

Bool, case, const, default, double, enum, float, hyper, int, long, opaque, short, string, struct, switch, typedef, union, unsigned, void.

2 - Declarações DIM

Na linguagem é previsto um conjunto de declarações que permitem especificar as interfaces de módulos:

Declaração da Descrição de Interface de Módulo

```
desc_modulo     ::=   "MODULE" nome_modulo
                          [decl_inclusão]
                          [decl_tipo]
                          [decl_porto]
                          [decl_atributos]
                      "ENDMODULE" "."

nome_modulo     ::=   identificador
```

As palavras reservadas "MODULE" e "ENDMODULE" encapsulam uma descrição de módulo definindo sua interface e atributos. O nome do módulo define um único módulo no sistema. Os itens subseqüentes descrevem todas as declarações que formam a DIM. Exceto o nome do módulo, todas as declarações são opcionais na descrição de interface.

Declaração de Inclusão

```
decl_inclusão       ::="INCLUDE" lista_nome ";"
lista_nome          ::=identificador ["," identificador]*
```

Esta declaração permite a inclusão de unidades de definição de tipos (UDT's), contendo declarações de tipos compartilhados entre módulos do sistema. Os identificadores na declaração indicam os nomes das UDT's incluídas, sendo estes os mesmos nomes dos arquivos que as contêm.

Declaração de Tipos

decl_tipo ::= (def_tipo)+

Utilizada para declaração de tipos construídos que não se encontram em UDT's, mas são suportadas pelos portos do módulo. A declaração define identificadores ou etiquetas para os tipos seguindo a notação XDR.

Exemplo:

```
struct mens {        /* mens representa
        int n;                a estrutura definida */
        float dados<50>;
};
typedef int resposta[50]; /* resposta é um novo nome
                          de tipo para um array de
inteiros com 50 elementos */
```

Declaração de Portos

```
decl_porto      ::=  "PORT" (esp_porto) +
esp_porto       ::=  lista_porto ":" "IN"
                     "(" tipo_xdr ["REPLY" tipo_xdr] ")" [tam_buf] ";"
                     |lista_porto ":" "OUT"
                     "(" tipo_xdr ["REPLY" tipo_xdr] ")" ";"
lista_porto     ::=  identificador ["," identificador]*
tipo_xdr        ::=  identificador de tipo XDR
tam_buf         ::=  "unsigned" "int"
```

A declaração de portos define as interfaces do módulo em relação ao sistema. Os portos de entrada e saída são especificados pelas palavras reservadas "IN" e "OUT" respectivamente. O tipo de sincronismo, envolvido nas comunicações que utilizam estes portos, é caracterizado pela presença ou não da mensagem de resposta: "REPLY" e conseqüentemente pelo bloqueio do processo emissor esperando por uma resposta. O porto é síncrono se existe resposta e assíncrono em caso contrário.

O tipo da mensagem é declarado entre parênteses, correspondendo a tipos simples definidos pela linguagem XDR, como inteiros, reais, booleano e *void* ou tipos construídos declarados a nível de descrição de interface ou de unidades de definição de tipos. Para o porto síncrono a declaração do tipo da mensagem resposta é feita após a palavra reservada "REPLY".

Para o porto de entrada assíncrono pode ser associado após a declaração do tipo da mensagem um inteiro sem sinal que indica o número máximo de *buffers* de armazenamento de mensagens. Na ausência deste último é assumido por default o valor 1. Com exemplo:

```
        typedef opaque pacote[128];
PORT
      beta        :   OUT ( unsigned int REPLY bool);
      gama        :   IN  ( float) 10;
      delta       :   OUT (pacote);
```

Declaração de Atributos

decl_atributos	::=	"ATTRIBUTE" lista_atributos ";"
lista_atributos	::=	(Nomeatr "=" Valoratr)+
Nomeatr	::=	identificador
Valoratr	::=	tipo_enumerado|"unsigned" "int"

Os atributos especificam propriedades do módulo. O programador lista os possíveis valores das propriedades. Estas informações são utilizadas pelo tradutor da DIM e pelo processador da LINCS.

Os exemplos de atributos especificados até o momento estão relacionados com a identificação da linguagem de implementação do módulo (linguagem hospedeiro), a identificação do processador onde ocorrerá a tradução, a versão do sistema operacional utilizado, a indicação da prioridade do módulo e os tipos de possíveis parâmetros de entrada e saída que o módulo pode conter. No primeiro caso o nome de atributo escolhido foi "lang" que é uma variável enumerada pré-definida pelo suporte, contendo como conjunto de valores as linguagens atualmente suportadas pelo SPML. Em linguagem XDR corresponde a:

```
enum lang {
   C = 0, FORTRAN = 1, PASCAL = 2};
```

Então "Valoratr" para "lang" conterá um destes elementos, correspondente à linguagem de implementação do módulo.

Para a identificação da máquina onde vai ser traduzida a Descrição de Interface, também são utilizadas variáveis enumeradas pré-definidas, uma com o nome da máquina e a outra a versão do sistema operacional. Estas especificações são mostradas no apêndice B. Para a prioridade foi escolhida uma variável inteira sem sinal pré-definida denominada "prior".

Se existem parâmetros de entrada e/ou saída no módulo, seus tipos (em notação XDR) são listados após a variável "parametros", na mesma ordem que se apresentam na codificação do módulo. Como exemplo temos:

```
ATTRIBUTE
    lang              =  FORTRAN;
    processador       =  SUN4;
    versao_SO         =  UNIX 4.0;
    prior             =  5;
    parametros        =  (string, int, float, bool);
```

Unidade de Definição de Tipos

A unidade de definição de tipos (UDT) é uma unidade de compilação separada e serve para declarar tipos construídos que serão usados por diversos módulos. Cada módulo que utiliza tipos declarados em uma UDT deve incluí-la. As UDT's são compiladas com extensão .def.

definição_tipo ::= def_tipo +

A UDT segue a sintaxe XDR para a definição de tipos com "typedef" ou etiquetas seguindo as palavras-chaves "enum", "struct" e "union".

Exemplo:

```
        typedef int num_valores;
struct mens {
        num_valores n;
        float dados<50>;
};
enum disc {
        m = 0,
        med = 1,
        var = 2,
        dp = 3,
        cv = 4  };
union result switch (disc opted) {
   case m         :     float media;
   case med       :     float mediana;
   case var       :     float variancia;
   case dp        :     float desvio_padrao;
   case cv        :     float coef_variacao;
   default        :     string nex<30>;
};
```

3.5.3 Linguagem de Configuração

A ferramenta inicial utilizada no processo de configuração é a linguagem de configuração, que possui um conjunto de comandos destinados a descrever a estrutura do sistema imaginada pelo programador. Os diversos comandos identificam os módulos do sistema (definição de contexto), determinam a criação de instâncias de tipos módulos, interconectam estas instâncias formando canais de comunicação e

sincronização, e descrevem o mapeamento lógico/físico, ou seja, informam em que entidades físicas serão executadas as instâncias. Todo este conjunto de instruções organizadas, seguindo a sintaxe definida pela linguagem de configuração, constitui a especificação do sistema.

Em nossa abordagem, a linguagem de configuração escolhida para a especificação do sistema é a LINCS - Linguagem de Configuração de Sistemas, pertencente ao sistema ADES [Fraga 89]. A fim de suportar programação mista, foram inseridas algumas modificações, porém praticamente toda sua sintaxe foi mantida, mesmo a parte referente à configuração dinâmica que não está sendo objeto deste trabalho .

A LINCS é uma linguagem declarativa cujas características mais relevantes são:

- Configuração hierárquica : Permite definir subsistemas.
- Reutilização de software : Instanciação de módulos facilitada pelas características de parametrização formal.
- Separação de funções : Declarações independentes para cada operação necessária na configuração do sistema.
- Atribuição de módulos às unidades de execução : Definição explícita das estações onde serão carregados os módulos.
- Facilidades para a especificação de mudanças em configuração dinâmica : Proíbe a nomeação direta a entidades externas aos módulos, condição básica para mudanças em tempo de execução no sistema.

As declarações da linguagem LINCS são classificadas em: básicas, inversas, auxiliares e de estruturação.

As declarações básicas descrevem a estrutura do sistema, definindo contexto (USE), criando instâncias (CREATE) e interconectando módulos através de ligações de portos (LINK), destacando assim, a separação explícita de funções.

As declarações inversas estão relacionadas à mudanças na configuração, envolvendo remoção de tipos módulos do contexto (REMOVE), destruição de instâncias (DELETE) e desligamento de portos (UNLINK).

As declarações auxiliares definem constantes (CONST), famílias (FAMILY) e portos de módulos multi-tarefas (PORT). Estas duas últimas não são usadas no suporte SPML, pelo fato de não existir o conceito de módulos multi-tarefas.

Finalmente as declarações de estruturação englobam as especificações de configuração, correspondendo a especificação inicial do sistema (SYSTEM) e a especificação de mudanças (CHANGE). A declaração de sub-especificação (GROUP MODULE) não é suportada pelo sistema, nesta versão multi-linguagens.

A sintaxe dos comandos LINCS podem ser encontrados em [Souza,88] e [Abreu,91]. Destacamos a seguir, a sintaxe das duas declarações que apresentam modificações, para uso em programaçàо com multi-linguagens:

Declaração de Contexto

```
decl_contexto          ::=  USE contexto ";" [contexto ";"]*
contexto               ::=  (ctexto_tipo_modulo ¦ctexto_tipo_dado)
ctexto_tipo_modulo     ::=  id_tipo_modulo
ctexto_tipo_dado       ::=  id_un__def__tipo
```

Em nosso suporte utilizamos o conceito de Unidade de Definição de Tipos ao invés do Módulo de Definição.

Declaração de Instanciação

```
decl_instância         ::=  CREATE decl_insta
decl_insta             ::=  [localiza] lista_decl_insta
localiza               ::=  [AT [ ":" ] ] STATION id_estação ":"
id_estação             ::=  "[" inteiro_positivo "]"
lista_dc_insta         ::=  [instância ";"]+
instância              ::=  [lista_insta ] [ ":" ] def_tipo
def_tipo               ::=  id_tipo_mod [lista_param] [atribui_índice]
lista_instância        ::=  id_insta ["," id_instância]*
lista_param            ::=  "(" parâmetros_reais ")"
param_reais            ::=  parâmetro ["," parâmetro]*
parâmetro              ::=  cadeia_de_caracteres
atribui_índice         ::=  "<" inteiro_positivo ">"
```

Os parâmetros atuais de entrada e saída de um módulo, que foi originalmente escrito em linguagens clássicas como C, Pascal ou Fortran, são especificados na criação da instância do módulo, como mostrado no exemplo:

```
CREATE AT STATION [0] :
        serv : gserv1(5);
        geren(true) <1>;
        ociosa0 : ociosa;
        simula1;

CREATE AT STATION [1] :
        instal : tipo1(2,3,true);
```

3.6 Exemplo de Programação Utilizando o SPML

Após a apresentação do método de programação utilizando o SPML, ilustra-se seu emprego em uma aplicação simples composta de dois módulos em linguagens diferentes.

O exemplo constrói um programa de cálculo de autovalores de matrizes reais quadradas. A aquisição dos dados de entrada e o controle do programa distribuído é realizado por um módulo escrito em C. O cálculo dos autovalores é realizado por um módulo escrito em Fortran, que devolve os resultados ao módulo em C. A figura 3.5 ilustra a aplicação; os códigos presentes nos arquivos de implementação de módulo são mostrados nas figuras 3.6 e 3.7. As descrições de interface destes módulos são apresentadas nas figuras 3.8, por fim apresentamos a especificação do sistema em relação à essa aplicação (figura 3.9).

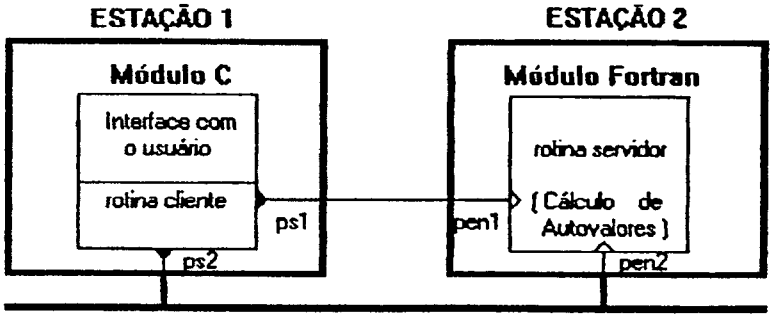

Fig. 3.5 Representação Esquemática da Aplicação

```
...
typedef float matel[100];
struct autoval {
float preal[10];
float pimag[10];
int ierr;
};
typedef struct autoval autoval;
...
main()
{
        ...
        calc_autoval(valores,apautoval);
        ...
}
void
calc_autoval(val,apval)
matel val;
autoval *apval;
{
        int timeout,falha,ndim,n,i;
        strcpy(ps1,"ps1");
        strcpy(ps2,"ps2");
        falha = 0;
        envass (ps1, &ndim, sizeof(ndim), &falha);
        ...
        envsin(ps2,val,sizeof(matel),timeout,apval,sizeof(autoval),falha);
        ...
        }
```

Fig. 3.6 Implementação do Módulo Cliente em C

```
implicit  real*8 (a-h,o-z)
dimension a(10,10)
dimension wr(10), wi(10), fv1(10)
integer   iv1(10), dim, timeout
character*10 porto
real*8 matel(100)
common/autov/ wr, wi, ierr

timeout = -1
porto = 'pen1'
dim = 0
call recebe (porto, dim, timeout,0,0)
...
porto = 'pen2'
call recebe (porto, matel, timeout,0,0)
l = 0
do 10 i = 1, dim
   do j = 1, dim
   l  = l + 1
   A(i,j) = matel[l]
   end do
end do
call rg (n, a, wr, wi, iv1, fv1, ierr)
call resposta (porto, autov)
stop
end
```

Fig. 3.7 Implementação do Módulo Servidor em Fortran

```
/* DIM DO MÓDULO DIRETOR */

MODULE  diretor

INCLUDE txdr

PORT
ps1 : OUT (int);
ps2 : OUT (matel REPLY autoval);

ATTRIBUTE
        lang      = C;
        processador = SUN4;
        versao_SO = UNIX 4.0;
        prior = 5;

ENDMODULE
```

```
/* DIM DO MÓDULO CALCULA */

MODULE      calcula

INCLUDE     txdr

PORT
pen1 : IN (int);
pen2 : IN (matel REPLY autoval);

ATTRIBUTE
   lang      = FORTRAN;
   processador = SUN4;
   versao_SO = UNIX 4.0;
   prior = 5;

ENDMODULE
```

```
                    /* UDT */


/* Arquivo txdr.def */


typedef float matel<100>;

struct autoval {
        float preal[10];
        float pimag[10];
        int   ierr;
};
```

Fig. 3.8 As DIM's dos Módulos do Exemplo e a UDT Incluída em Ambos

```
SYSTEM  exemplo;

USE     diretor, calcula;

CREATE  at station[0]:
                          diretor;

CREATE  at station[1]:
                          calcula;

LINK
diretor.ps1 TO calcula.pen1;
diretor.ps2 TO calcula.pen2;

END.
```

Fig. 3.9 Especificação do Sistema

3.7 Conclusão

Neste capítulo descrevemos a estrutura funcional de um suporte para Programação Distribuída em Múltiplas Linguagens (SPML). Este sistema é separado em ferramentas de programação (linguagens DIM e LINCS) e em um suporte de tempo de execução.

O estilo de programação do sistema ADES é assumido como paradigma no SPML e portanto o sistema apresenta características de programação modular. A separação entre interface e implementação de módulos permite que se trate as diferenças de linguagens e de máquinas em ambientes heterogêneos. A Descrição de Interface de Módulo é o meio utilizado para a resolução destas diferenças no SPML. A linguagem de Descrição de Interface de Módulo (DIM), baseada na notação XDR, permite a composição de programas distribuídos e ainda a transferência de informações neste ambiente multi-linguagens.

CAPÍTULO 4

PROJETO DE UM PROTÓTIPO DO SUPORTE PARA PROGRAMAÇÃO DISTRIBUÍDA EM MÚLTIPLAS LINGUAGENS

4.1 Introdução

Neste capítulo são apresentados os aspectos de projeto e implementação do suporte para programação distribuída em múltiplas linguagens (SPML). O desenvolvimento de um programa distribuído no SPML envolve as seguintes fases:

- **Tradução da DIM:** fase envolvendo ferramentas de análise e produzindo estruturas de dados intermediárias, correspondentes à interface do módulo e gerando como saída um arquivo de informações para a configuração e os códigos das Rotinas de Interface do módulo.

- **Configuração do Sistema:** nesta fase o sistema é construído usando as ferramentas de programação (LINCS) para a especificação da configuração e a construção do sistema propriamente dita. O arquivo de informações gerado pela tradução da DIM é usado como entrada na tradução de especificações de configuração.

- **Gerenciamento em Tempo de Execução:** aborda principalmente os mecanismos envolvidos com o escalonamento e comunicação no sistema.

4.2 Tradução de Especificações DIM

A tradução de cada especificação DIM é realizada pelo tradutor DIM e corresponde à análise sintática e à geração de ações semânticas a partir de suas declarações: a análise sintática verifica se a seqüência de declarações obedece a certas convenções estruturais explícitas na definição sintática da DIM; as ações semânticas proporcionam a formação de estruturas de dados a serem utilizadas pelo Gerador de Rotinas de Interface e na geração de um arquivo de dados a ser utilizado pelo processador LINCS na configuração. As características do tradutor permite que este seja desenvolvido de acordo com a abordagens clássicas de compiladores [Setzer 85], [Kowaltowski 83] e [Aho 89]. Na figura 4.1 apresentamos o diagrama funcional do tradutor DIM.

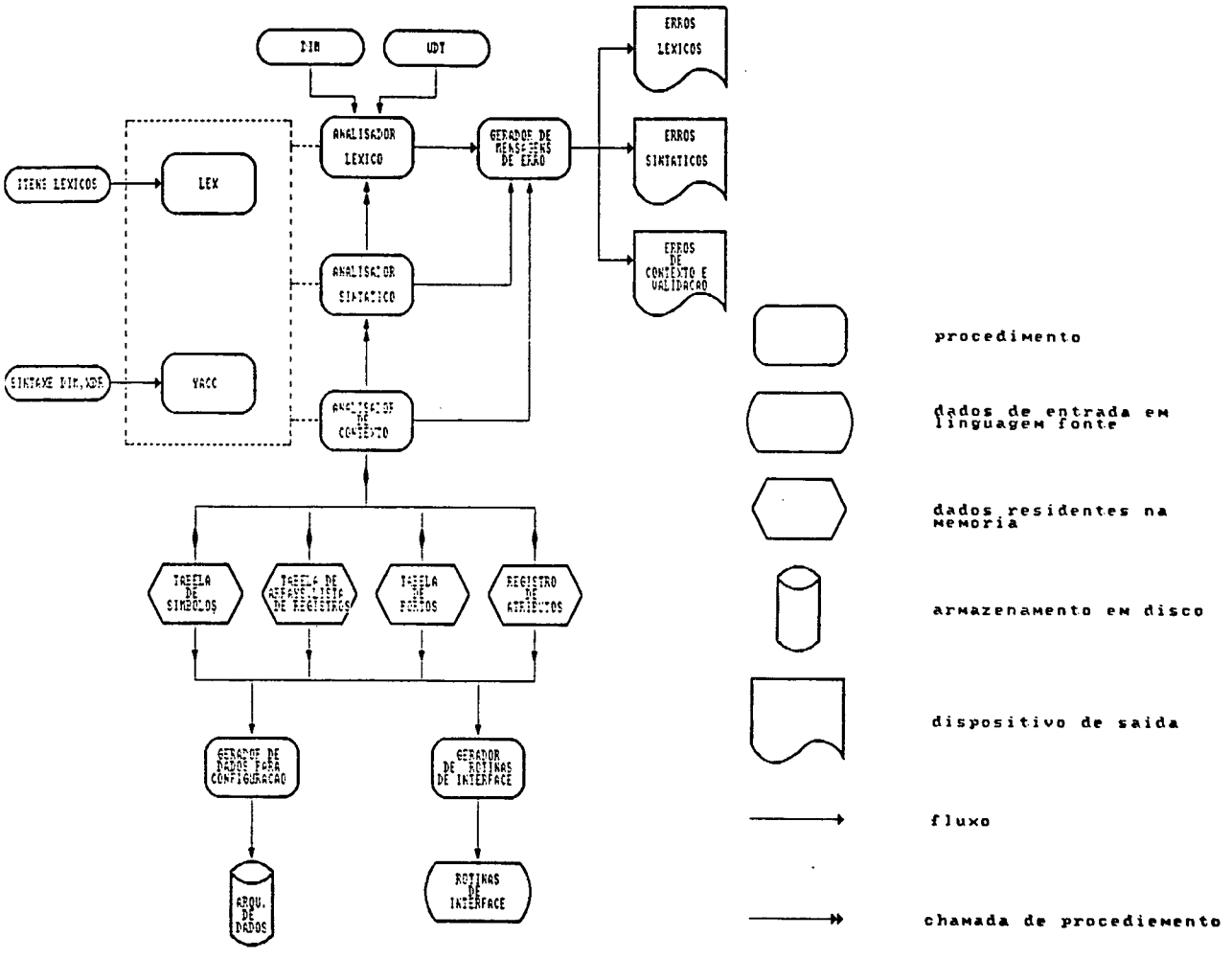

Fig. 4.1 Diagrama Funcional do Tradutor DIM.

Antes de descrever cada item que compõe o diagrama mostrado, precisamos fazer algumas considerações iniciais sobre as declarações contidas na DIM. As especificações DIM se caracterizam por serem códigos fonte onde são encontradas unidades sintáticas, descrevendo a estrutura gramatical das especificações de interface. Por exemplo, a declaração de um porto:

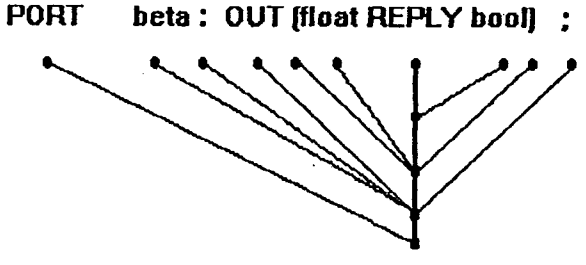

Fig. 4.2 Árvore Sintática da Declaração de Porto

Observando o que chamamos de árvore sintática da declaração de um porto, notamos que cada um de seus nós representa uma unidade sintática e que existem unidades compostas a partir de outras mais simples, indo até as elementares que estão no primeiro nível (PORT,beta,":",OUT,"(", etc.) e as quais denominamos itens léxicos.

4.2.1 O Analisador Léxico

O analisador léxico realiza a leitura da DIM, fazendo uma varredura linha a linha, agrupando os caracteres em itens léxicos. Para cada item léxico é definido um token que determina sua classe (identificador, número, palavra reservada) que é enviado ao analisador sintático. O analisador léxico tem também por tarefa o reconhecimento de erros léxicos, que pode ser, por exemplo, um símbolo especial inválido ou a formação incorreta de um identificador.

Para a implementação foi escolhida a ferramenta Lex [Lesk 82], que gera um analisador léxico a partir de uma especificação de entrada. Esta especificação compreende um conjunto de expressões regulares e fragmentos de programas. As expressões regulares definem os itens léxicos.

Deste modo o analisador léxico particiona os dados de entrada, no nosso caso o arquivo fonte da DIM, em cadeias de caracteres que se encaixam nas expressões regulares, classificando-as em *tokens*, através dos fragmentos de programa associados às expressões. Cada vez que o analisador léxico é chamado um item léxico é identificado e seu respectivo *token* é enviado ao analisador sintático.

4.2.2 O Analisador Sintático

O analisador sintático deve detectar erros sintáticos, ou seja, construções no programa fonte que não estão de acordo com as regras gramaticais da linguagem. O erro quando reconhecido, provoca a ativação, por parte do analisador sintático, do gerador de mensagens de erro para a emissão da mensagem correspondente.

Foi escolhida para ser utilizada na implementação do analisador sintático para DIM a ferramenta Yacc [Johnson 78] que fornece meios para se gerar analisadores sintáticos. Para tanto, devemos fornecer um arquivo de especificação contendo as regras gramaticais que definem a sintaxe da linguagem cujos códigos

serão analisados (no caso, a sintaxe da linguagem DIM). O analisador sintático se utilizará do analisador léxico definido acima para que este forneça as unidades sintáticas elementares (*tokens*) nos códigos fontes. Os *tokens* são organizados segundo as regras gramaticais especificadas, que ao serem reconhecidas desencadeiam ações correspondentes, executando códigos especificados pelo projetista do analisador.

No processo de análise, o analisador sintático percorre um código fonte, classificando *tokens* em unidades sintáticas elementares, agrupando-as, segundo regras gramaticais, em unidades gramaticais compostas, construindo então a árvore sintática correspondente à especificação DIM. Sempre que uma regra gramatical é reconhecida são ativadas ações semânticas do analisador de contexto.

4.2.3 O Analisador de Contexto

O analisador de contexto deve criar as estruturas de dados correspondentes às especificações de interface do módulo: as tabelas de símbolos, portos, *arrays*, *records* e atributos. Através destas estruturas o analisador insere, busca e classifica os identificadores contidos na DIM.

O analisador de contexto é responsável pela deteção de erros de contexto, como por exemplo, a deteção da inexistência de um identificador na tabela de símbolos, quando de acordo com a regra gramatical este já deveria ter sido declarado.

Estas ações semânticas, em parte, são realizadas pelas ações inseridas através do Yacc e em outra pelas ações indicadas na especificação do Lex.

4.2.4 Principais Estruturas de Dados Presentes no Tradutor DIM

Tabela de Símbolos

A tabela de símbolos é uma tabela que contém entradas para os identificadores de tipo definidos pelo usuário na especificação DIM, e todos os identificadores que se referem à comunicação: o tipo de sincronismo dos portos e os tipos das mensagens que estes suportam.

A tabela de símbolos é a estrutura de dados central do gerador das Rotinas de Interface, pois a esta ligam-se outras estruturas necessárias para completar

toda a informação sobre um identificador presente em uma especificação DIM. Nesta tabela cada item é composto por quatro campos: **nome, classe, tipo e indt,** conforme [Setzer 85]. O nome do identificador é a própria cadeia de caracteres reconhecida pelo analisador léxico. A classe revela o que o identificador significa: um módulo, um tipo, um campo de uma estrutura, um porto, um atributo ou uma constante. O tipo está relacionado à classe e complementa a informação sobre o identificador, determinando sua função ou a que tipo de estrutura este se refere. indt é um índice que pode referenciar outra tabela, fornecer informações sobre o tamanho do tipo ou ainda apontar para um outro item da tabela de símbolos.

Tabela de Portos

Em cada elemento desta tabela encontram-se as informações associadas a um determinado porto. Apresenta em cada entrada, ponteiros que indicam na tabela de símbolos, o módulo que possui o porto, o tipo de mensagem veiculada por este além de outras informações.

Tabela de Arrays

Quando é identificado um tipo array nas especificações DIM, este é inserido na tabela de símbolos; o campo indt possui um índice que referencia a entrada em uma tabela de arrays, onde são descritas as características do mesmo.

Tabela de Registros

De maneira análoga aos arrays, também existe uma tabela que descreve os registros (*records*) declarados na DIM.

Registro de Atributos do Módulo

Quando a tabela de símbolos está sendo formada, para incorporar os atributos do módulo é formada também uma estrutura que concentra todas as características descritas sobre os mesmos. Esta estrutura é o registro de atributos do módulo.

4.2.5 Gerador das Rotinas de Interface

O Gerador de Rotinas de Interface (GRI) é a ferramenta responsável pela geração dos algoritmos envolvidos com o tratamento da heterogeneidade linguagem/máquina e da interface de comunicação do suporte de tempo de execução, apresentando alguns aspectos similares à proposta do sistema Horus

[Gibbons 87]. Seu objetivo é liberar o programador do módulo das questões pertinentes à comunicação tais como: formato dos dados das mensagens, peculiaridades do sistema operacional, diferentes representações de tipos, etc, fornecendo ao usuário um nível de interface de mensagem padrão entre processos.

Para cada porto de comunicação está associada uma Rotina de Interface, que é fundamentalmente, um código composto de uma parte de declarações de tipos (correspondentes as informações de portos e mensagens), um cabeçalho do procedimento de envio ou recepção e o corpo deste procedimento, que contém ações cujo objetivo é:

- Serializar os dados da mensagem no caso de envio através do uso de rotinas de conversão para a sintaxe concreta do XDR;
- Invocar as rotinas de comunicação do suporte de tempo de execução;
- Desserializar os dados da mensagem no caso de recepção através das rotinas de conversão e passar a mensagem ao processo receptor.

Este código será gerado na mesma linguagem do módulo associado.

O Gerador de Rotinas de Interface é independente das linguagens e máquinas; todas as informações sobre as construções da linguagem hospedeiro e os formatos de máquina são armazenadas em **Especificações de Linguaguem** e **Especificações de Máquina**, respectivamente [Gibbons 87]. Desta forma a função do gerador é analisar a Tabela de Símbolos e conforme as especificações de linguagem e de máquina gerar o código das Rotinas de Interface. A figura 4.3 ilustra os componentes usados para a geração das rotinas.

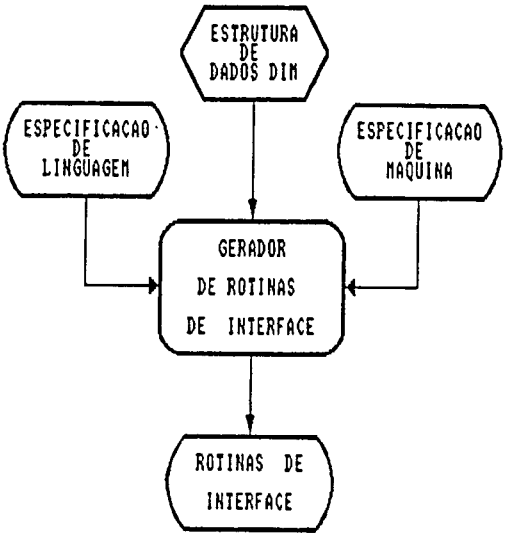

Fig. 4.3 Geração das Rotinas de Interface.

A estrutura funcional do Gerador de Rotinas de Interface segue também a estrutura de compiladores. Assim, o gerador inicia seu processamento seguindo passo a passo a Tabela de Símbolos do módulo e dependendo do elemento da tabela, aciona a interação dos analisadores sintático e léxico, que agem sobre uma parte determinada da especificação de linguagem e/ou máquina e, através das ações semânticas e de geração de código produzem os diversos códigos que correspondem às Rotinas de Interface.

O Gerador de Rotinas de Interface será escrito em C e utiliza analisadores sintático e léxico, implementado também a partir das ferramentas Yacc [Johnson 78] e Lex [Lesk 82], para o processamento das informações de entrada.

As especificações de linguagem e de máquina não são atribuições do programador de módulo, mas do responsável pelo suporte, que implementa para cada linguagem presente no sistema a respectiva especificação de linguagem e, analogamente, para as diferentes máquinas da rede. Detalhamos a seguir as diversas características da implementação das especificações citadas.

4.2.5.1 Especificação de Linguagem

Como a descrição de interface de módulo possui todos os tipos definidos em XDR, cada especificação de linguagem hospedeiro deve conter um conjunto de comandos que especifica completamente a correspondência dos tipos XDR com os da linguagem hospedeiro, podendo apresentar também, um tratamento mais elaborado quando não existe correspondência direta, com comandos de composição e decomposição de tipos.

O conjunto de comandos que fazem parte da especificação de linguagem são processados pelo Gerador de Rotinas de Interface que emite código, segundo estes comandos, em linguagem hospedeiro para:

- Declarações de tipos;

- Serialização e desserialização dos tipos de interface;

- Geração do corpo das Rotinas de Interface.

Cada linha de comando consiste de uma declaração de geração e de uma linha de caracteres com espaços significativos. Esta linha é composta de

identificadores pré-definidos da linguagem alvo e de nomes de variáveis correspondentes a atributos de um elemento da Tabela de Símbolos.

O Gerador de Rotinas de Interface processa então o seguinte conjunto de comandos: WRITE, REPEAT, NEXT, MARSHAL e M_MEMBER. A seguir descrevemos os comandos do gerador, conforme são utilizados nas diferentes partes de uma especificação de linguagem (declaração de tipos, conversão de dados, corpo das Rotinas de Interface) para a geração de todo código necessário para formar as Rotinas de Interface.

Declaração de Tipos

A especificação de linguagem possui para cada tipo definido na DIM uma especificação de tipo, que contém todos os comandos necessários para gerar a declaração daquele tipo na linguagem hospedeiro.

Por exemplo a declaração de tipo para uma *string* em C, cujo identificador na DIM é "escolha" e tamanho 80, seguirá o seguinte comando:

```
string escolha<80>;
```

```
WRITE     typedef char $type_name[$value];
```

O comando WRITE emite a linha de caracteres correspondente ao texto fonte da Rotina de Interface. WRITE é a declaração básica de geração de código. No exemplo, o Gerador de Rotinas de Interface substituirá $type_name pelo identificador "escolha" e $value por 80. A Rotina de Interface conterá a seguinte linha após a efetuação deste comando:

```
typedef char escolha[80];
```

O comando WRITE é insuficiente para emitir declarações de tipos construídos que possuem um número arbitrário de elementos. Assim, mais duas declarações de geração são fornecidas: REPEAT e NEXT que oferecem a possibilidade da construção de um *loop* para declarar elementos de uma seqüencia arbitrária. REPEAT é uma declaração de controle do gerador, WRITE possui a linha de caracteres correspondente a declaração dos membros da seqüência e NEXT separá-os, podendo

interpretar os caracteres de formatação do C como \n para nova linha e \t para tabulação, e também marca o final do *loop*. As variáveis na linha associada ao WRITE referem-se a identificadores (na Tabela de Símbolos) do membro daquela iteração.

Considerando uma DIM que define um registro chamado "tabela" consistindo de uma string denominada "nome" e de um inteiro "identidade", a seguinte declaração de geração seria então utilizada para a linguagem Pascal:

```
WRITE     type $type_name = record
REPEAT          (n_fields)
WRITE     $field_name : $field_type_name;
NEXT
WRITE     end;
```

O gerador emitiria o seguinte código para a Rotina de Interface em questão:

```
type tabela = record
        nome : string[20];
        identidade : integer
end;
```

É importante perceber que todo o conhecimento sobre um tipo particular é isolado dentro da especificação para aquele tipo, uma mudança em um tipo acarreta somente a modificação da especificação associada a ele, o restante permanece inalterado.

Conversão de Dados

Para a conversão dos dados das mensagens a especificação de linguagem utiliza a mesma estrutura da declaração de tipos, com a diferença da inclusão de alguns novos comandos.

A conversão é realizada por "filtros" existentes na biblioteca XDR, que corresponde a um conjunto de funções que convertem dados entre as representações local e *streams* XDR (representação de transferência). Os filtros possuem função tríplice: codificar, decodificar e liberar memória que um filtro possa ter alocado. Assim, os comandos de geração emitem chamadas a estes filtros de acordo com o tipo a ser convertido.

A biblioteca XDR fornece os chamados "filtros primitivos" correspondendo aos tipos básicos suportados pelas linguagens clássicas. A figura 4.4 lista estas funções correspondentes aos tipos da linguagem C:

C Type	Filter	XDR Type
char	xdr_char ( )	int
short int	xdr_short ( )	int
unsigned short int	xdr_u_short ( )	unsigned int
int	xdr_int ( )	int
unsigned int	xdr_u_int ( )	unsigned int
long	xdr_long ( )	int
unsigned long	xdr_u_long ( )	unsigned int
float	xdr_float ( )	float
double	xdr_double ( )	double
void	xdr_void ( )	void
enum	xdr_enum ( )	int

Fig. 4.4 Filtros Primitivos

Os filtros apresentam dois argumentos, o primeiro é um apontador para o controlador da "fonte XDR", o segundo é o endereço do dado de interesse (obtido na passagem de parâmetros do módulo quando é requisitado a comunicação).

O meio onde os filtros lêem e escrevem os dados na representação padrão denomina-se "fonte XDR" ou "*stream* XDR", que pode residir em qualquer meio: memória, disco, *tape*, etc. A biblioteca XDR possui três tipos de *streams*: *standard I/O*, *memory* e *record streams*, e também podem ser criados *streams* usuários.

O controlador XDR suporta a abstração de *stream* XDR. Ele possui informação sobre a operação sendo aplicada ao *stream*, um vetor de operações e alguns dados privados (que não devem ser alterados). O vetor de operações contém o endereço das funções que realmente lêem, escrevem dados e destroem *streams*. A operação é definida na criação do *stream*. Os *streams* XDR constituem a sintaxe concreta, utilizada como representação de transferência na comunicação entre ambientes heterogêneos. Maiores detalhes sobre *streams* e o controlador XDR podem ser obtidos em [Corbin 90].

Em adição aos filtros primitivos, a biblioteca XDR fornece filtros para tipos construídos comumente usados: *string*, *byte*,vetor, *array*, *union*. A sinopse destes filtros são descritos no apêndice B.

Notamos que a biblioteca XDR não fornece um filtro para registros, porque seria difícil saber antecipadamente os tipos dos membros da estrutura. Os filtros para registros são denominados *custom filters* e se comportam como procedimentos que chamam outros mais primitivos. Para implementar esses filtros utilizamos comandos de geração na especificação de linguagem.

Após estes esclarecimentos podemos descrever como é especificada a conversão de dados, através dos comandos de geração.

O código para codificar/decodificar os dados depende do tipo do dado e, se for tipo construído, depende também dos tipos de seus membros. Para emitir o código que utiliza um filtro XDR, introduzimos um novo comando de geração o MARSHAL, onde a linha de caracteres associada é uma chamada a um filtro XDR.

Para exemplificar começamos com um tipo primitivo. Considerando na DIM, a declaração de uma mensagem cujo tipo é um enumerado, o seguinte comando é especificado:

```
MARSHAL xdr_enum(xdrs, objp)
```

onde xdrs é o controlador de *stream* e objp o endereço do dado. É importante perceber que é usado o mesmo comando MARSHAL tanto para a serialização quanto para a deserialização, o filtro sabe qual é a operação desejada devido a definição da mesma na criação do *stream*.

Para um tipo construído, digamos a *string* "exemplo" de tamanho 80, teremos o seguinte comando de geração:

```
MARSHAL xdr_string(xdrs,&objp, $value)
```

onde &objp é o endereço do apontador da *string*, $value o tamanho. No caso de uma string variável $value seria de tamanho máximo. Logo para a *string* "exemplo" o gerador emite:

```
xdr_string(xdrs,&objp,80)
```

Na montagem de filtros de estruturas para tipos registro introduzimos o comando M_MEMBER que redireciona para o filtro correspondente ao tipo do campo

definido por $type_fiiden (valor retirado da Tabela de Símbolos). As linhas de comando para a geração de um *custom filter* são:

```
WRITE     bool_t
MARSHAL  xdr_$type_name(xdrs, objp)
WRITE            XDR   *xdrs;
WRITE            $type_name *objp;
WRITE     {
REPEAT           (n_fields)
WRITE            if (!
M_MEMBER  xdr_$type_fiiden ) {
WRITE            return(FALSE);
WRITE            }
NEXT
WRITE            return(TRUE);
WRITE     }
```

O filtro será chamado pela Rotina de Interface como um procedimento, com a seguinte linha de comando:

```
MARSHAL  xdr_$type_name(xdrs, objp)
```

Como exemplo, suponhamos a estrutura definida na Unidade de Definição de Tipos (UDT) do exemplo do capítulo 3:

```
struct autoval {
        float preal[10];
        float pimag[10];
        int ierr;
};
```

A rotina de conversão emitida pelo gerador será:

```
bool_t
xdr_autoval(xdr, objp)
XDR *xdrs;
autoval *objp;
{
if (!xdr_vector(xdrs, (char *)objp->preal, 10, sizeof(float), xdr_float)) {
return(FALSE);
}
if (!xdr_vector(xdrs, (char *)objp->pimag, 10, sizeof(float), xdr_float)) {
return(FALSE);
}
if (!xdr_int(xdrs,&objp->ierr)) {
return(FALSE);
}
return(TRUE);
}
```

Corpo das Rotinas de Interface

Também na especificação de linguagem estão as linhas de comando para a emissão do código que declara a Rotina de Interface como um procedimento.

Para cada tipo de comunicação existe uma Rotina de Interface associada. O gerador analisa os portos descritos na Tabela de Símbolos para emitir as Rotinas de Interface necessárias. Em todas são incluídas a biblioteca XDR e as definições de tipos necessários; também são criados os *streams* e definidas as operações de conversão de dados (de acordo com o sincronismo da comunicação).

O apêndice C traz uma especificação de linguagem completa da emissão de Rotinas de Interface para todos os tipos de portos.

A figura 4.5 ilustra sinteticamente os arquivos gerados a partir da especificação de linguagem C, para a estrutura "tabela" definida em uma DIM, e considerando a parte das Rotinas de Interface que trata do envio assíncrono desta variável.

4.2.5.2 Especificação de Máquina

As especificações de máquina contêm informações sobre a ordenação dos *bytes* na máquina, o tamanho e a representação dos tipos primitivos de hardware como: caracter, inteiros com e sem sinal de vários tamanhos e, para os tipos de ponto flutuante. Desta forma, toda a informação para conversão de dados que depende da máquina é fornecida. No apêndice C, mostramos um exemplo de especificação de máquina para estações de trabalho SPARC-SUN (escopo do trabalho simplificado na primeira versão).

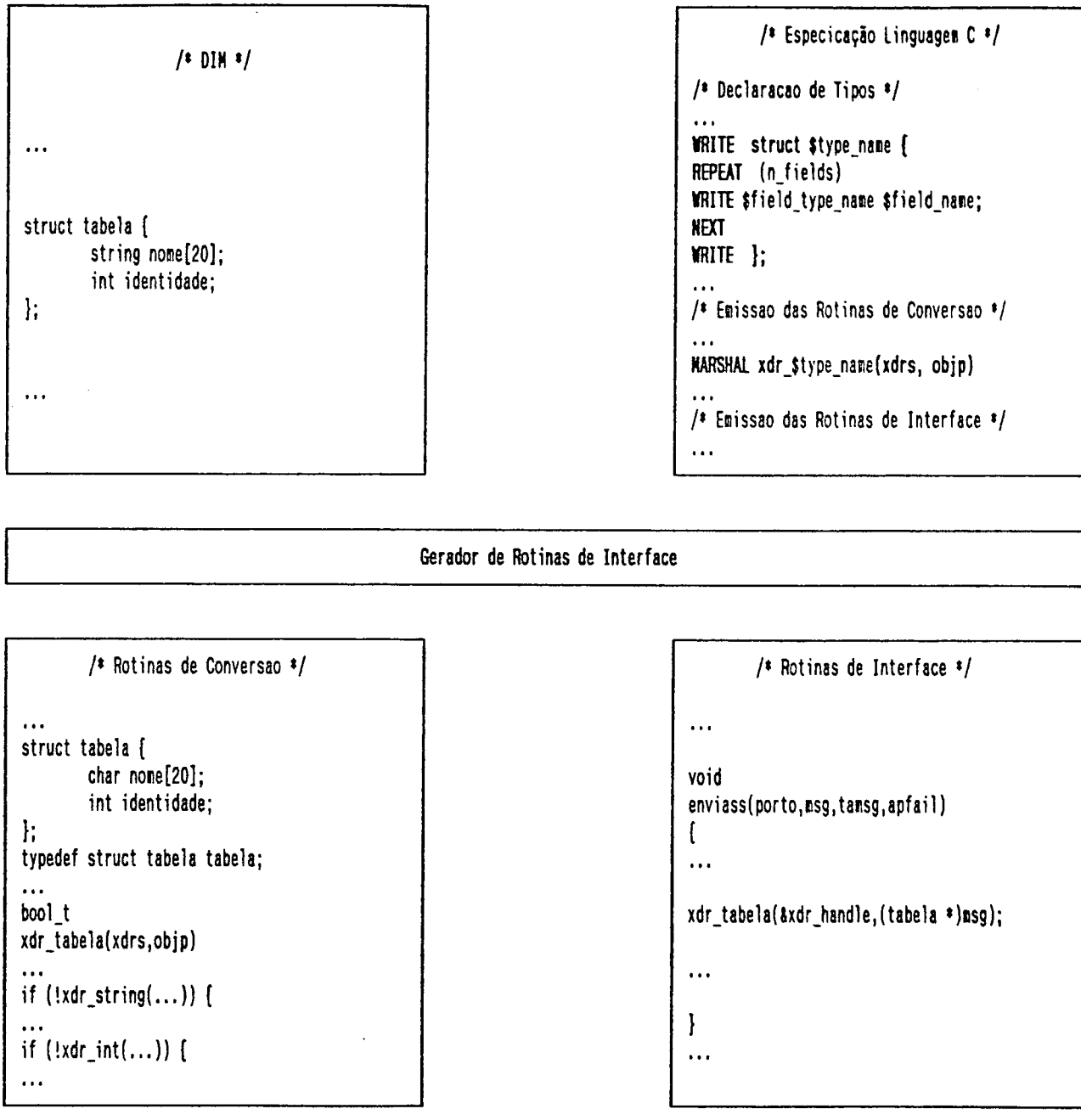

Fig. 4.5 Especificações de Entrada do Gerador e Códigos Gerados

4.3 Geração de Dados para Configuração

O gerador de dados para configuração é a entidade responsável pela obtenção de informações das tabelas de símbolos, de portos e do registro de atributos, e pela conseqüente preparação destas informações como dados de entrada necessários para a configuração. A extração das informações destas tabelas é realizada por um código de programa que é ativado quando da sinalização da não ocorrência de erros nas análises léxica, sintática e de contexto no código fonte da DIM.

As estruturas geradas para a configuração contêm as seguintes informações:

Informações de Portos do Módulo

- nome do porto
- tipo da mensagem
- unidade de definição
- prioridade
- tipo do porto
- tamanho do buffer

Informações dos Atributos do Módulo

- linguagem de implementação do módulo
- processador onde foi realizada a tradução
- versão do sistema operacional utilizado
- prioridade do módulo
- índice do módulo

4.4 Configuração do Sistema

Até o momento discutimos a especificação dos componentes elementares de um sistema distribuído: os módulos; que permitem a definição de tipos dos quais podem ser criadas instâncias. Agora, abordamos a construção do sistema a partir desses módulos escritos nas diversas linguagens. Recordando a figura 3.3 (item 3.4) percebemos que a partir do sucesso de uma especificação de sistema, o tradutor LINCS cria uma tabela de configuração para cada estação de um sistema distribuído, representando a estrutura lógica do software em cada entidade física (processador) do sistema. O construtor do sistema carrega em cada estação uma instância do suporte de tempo de execução, o **Gerenciador de Estação**, que a partir de um procedimento de iniciação monta o sistema, seguindo as informações da respectiva tabela de configuração (processo de configuração estática).

4.4.1 Processador LINCS

O processador LINCS de forma semelhante ao tradutor da DIM é composto por várias unidades funcionais destinadas a ler as especificações de configuração, processá-las e gerar estruturas de dados correspondentes ao tipo de especificação. O diagrama da figura 4.6 apresenta os aspectos funcionais do processador.

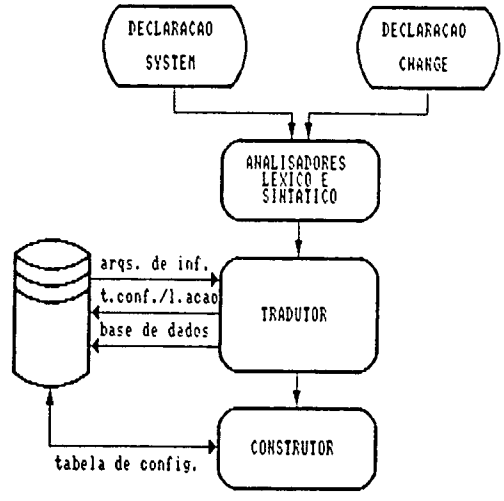

Figura 4.6 Processador LINCS

No sentido de atender a utilização em ambientes heterogêneos de programação, o processador LINCS teve que sofrer um conjunto de alterações em vários de seus aspectos originais. Por questões de simplicidade nos limitamos a certos detalhes ligados a estas alterações, principalmente devido ao fato de não haver mais os conceitos de módulo multi-tarefas, de grupo de módulos e família de portos. Aspectos mais completos sobre o processador LINCS e as estruturas de dados resultantes do processamento de uma especificação de configuração podem ser obtidos em [Souza 88] e [Abreu 91].

4.4.1.1 Alterações no Processamento de Declarações LINCS

Neste item serão citadas as declarações LINCS que apresentam modificações no seu processamento, no sentido de atender a programação em ambientes heterogêneos.

Processamento da Declaração de Contexto (USE)

Esta declaração tem por objetivo a determinação de tipos módulos e de unidades de definição que comporão a configuração do sistema. No seu processamento é verificado pelo processador LINCS a existência dos códigos

referentes aos tipos módulos no sistema de arquivos da estação de trabalho e em seguida, a cada módulo identificado nesta declaração é atribuído um índice que o identifica no contexto do sistema.

Processamento da Declaração de Instanciação (CREATE)

O processamento desta declaração cria a estrutura denominada "instância" e a insere em uma estrutura "estação", já existente na base de dados do suporte de configuração, correspondente à entidade física onde a instância será executada. Também são validados os parâmetros reais que devem ser fornecidos caso o tipo módulo contenha parâmetros de entrada e/ou saída. A instância é representada pela seguinte estrutura :

```
struct instancia {
        char    id_insta[], id_tipo[];
        int     ind_tip_mod;
        char    *param[];
        struct instancia *prox;
};
```

Com os dados obtidos no arquivo de dados do módulo, é realizada uma validação em relação à compatibilidade do tipo de processador existente na estação e o indicado nos atributos do módulo da DIM. A validação dos parâmetros reais é realizada através da leitura e comparação com os dados referentes aos tipos correspondentes, listados no atributo "parâmetros" da DIM correspondente.

Por fim, é atribuído um índice à instância que é único a nível de estação e que a identifica nas operações do sistema operacional distribuído. A nível de sistema a instância é identificada por uma hierarquia de campos com atributos de localização (concatenação dos índices da estação e da instância).

4.4.1.2 Base de Dados Representativa da Configuração Especificada

O sucesso da tradução e validação da especificação resulta em uma estrutura de dados global do sistema, composta de estruturas referentes a estações, tipos, instâncias, conexões, etc. A base de dados formada por esta estrutura hierárquica, uma vez que a configuração definida pela especificação correspondente esteja instalada no sistema, passa a ser o estado atual de configuração do sistema. Esta base de dados estará armazenada no sistema de forma a poder ser recuperada quando necessária.

4.4.1.3 Tabelas de Configuração

Obtendo as informações geradas na tradução da especificação de configuração (declaração System), são geradas as tabelas de configuração específicas para cada estação do sistema que contêm informações sobre :

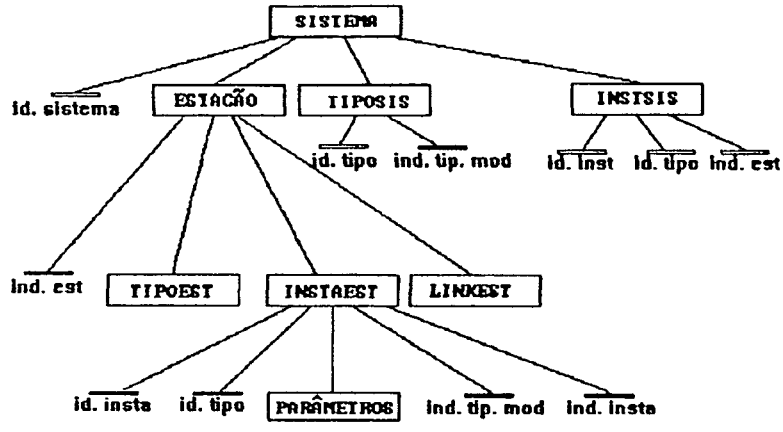

Fig. 4.7 Base de Dados Representativa do Estado de Configuração do Sistema

- O índice da estação
- Número de tipos módulos
- Informações de tipos módulos
- Número de instâncias
- Informações de instâncias
- Número de conexões
- Informações de conexões dos portos

Esses dados serão utilizados pelos procedimentos de iniciação das estações. Para construir o sistema, o procedimento de iniciação de cada estação através dos serviços do seu Gerenciador de Estação, utilizando as informações contidas na respectiva tabela de configuração, cria os blocos de controle dos tipos módulos, carrega os códigos, cria as instâncias e seus blocos de controle tornando-as aptas para a execução.

4.4.2 Programa Construtor

O procedimento de iniciação em cada estação, na sua execução precisa de três elementos: a tabela de configuração, o Gerenciador de Estação e os códigos aplicativos. Várias soluções são possíveis para criar as condições de execução do procedimento de iniciação, porém duas abordagens são sugeridas:

- A primeira utiliza um sistema de arquivos de rede (NFS) [Sun Microsystems 90] que permite o acesso transparente a arquivos no sistema distribuído, fornecendo independência de sistema operacional, de máquina e acesso transparente. Assim, para construir o sistema, a estação de trabalho cria em cada estação, uma instância do Gerenciador de Estação e executa o procedimento de iniciação que utiliza o serviço NFS para acessar os arquivos necessários.

- Outra proposta é a utilização de um processo servidor, criado durante o *boot* da máquina, e que seja responsável pelo recebimento dos códigos dos módulos e da tabela de configuração, através dos recursos de comunicação do sistema. Desta forma, o construtor na estação de trabalho, ativa esses servidores e envia os códigos às estações, cria instâncias do Gerenciador de Estação e são executados os programas de iniciação.

4.5 Gerenciamento em Tempo de Execução

O suporte de tempo de execução deve implementar todas as abstrações do modelo de programação adotado. No sentido de facilitar a implementação do suporte do SPML, foi definida como camada subjacente o sistema UNIX, possibilitando então que vários serviços e ferramentas disponíveis neste ambiente fossem utilizados nas implementações desejadas.

No caso das abstrações do paradigma de programação utilizado, no sistema original, estas eram implementadas através de um núcleo especialmente desenvolvido (NTR [Nacamura 88]). A implementação destas mesmas abstrações sobre o sistema UNIX, deve se utilizar de conceitos e serviços deste último. É o caso, por exemplo, de processos UNIX usados para implementar o conceito de *processos/instâncias* do módulo adotado.

O suporte de tempo de execução do SPML, deve manter praticamente a mesma interface de serviços do NTR. No sentido do gerenciamento das abstrações do modelo de programação, são também mantidas praticamente as mesmas estruturas

de dados presentes no NTR. Na implementação deste suporte sobre o sistema UNIX, a solução empregada foi a de centralizar todas as estruturas do suporte (e por conseqüência dos serviços) sobre um processo UNIX: O **Gerenciador de Estação**. As interfaces de serviços (primitivas do NTR) estão disponíveis em todos os processos que compõem a aplicação. Estas primitivas quando acionadas podem envolver um processamento local (Rotinas de Interface) antes de serem acionadas e ativarem os serviços correspondentes no processo Gerenciador de Estação. A passagem do pedido de serviço se dá através da camada inferior (sistema UNIX), usando os mecanismos de comunicação desta (serviços de *socket*). A figura 4.8 ilustra a interação dos processos/instâncias de módulos com o Gerenciador de Estação.

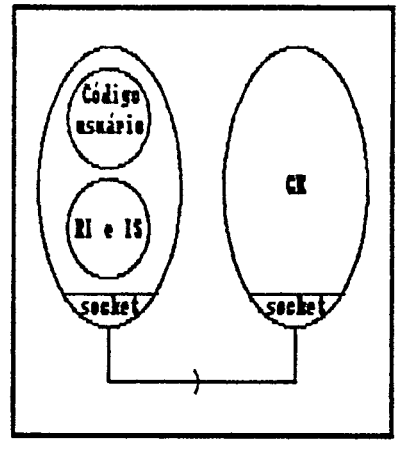

Figura 4.8 Solicitação de Serviços ao GE

4.5.1 Visão Global do Gerenciador de Estação

O Gerenciador de Estação (GE) corresponde a um processo do Sistema Operacional UNIX, onde são mantidas todas as informações sobre as entidades definidas no modelo de programação (tipo módulo, processo/instância de módulo, portos, etc.), presentes na estação. Diante disto, o Gerenciador de Estação centraliza aspectos como a configuração da estação. Na criação de instâncias dos módulos, o Gerenciador de Estação executa uma chamada *fork* do UNIX que cria um processo filho, em seguida uma chamada *exec* associa um arquivo de código executável (tipo módulo) ao processo/instância. Na seqüência da configuração os processos/instâncias devem ser "ligados" através de seus portos. Estas ligações devem passar pelo GE.

No momento da inicialização do GE um *socket* (mecanismo presente no UNIX que será usado tanto para comunicações locais como remotas entre processos) é

criado e habilitado a receber comunicações provenientes de sockets dos processos/instâncias. As instâncias devem logo no início chamar a rotina *start* da Interface de Serviços, que cria o seu *socket* de comunicação. Este procedimento habilita a instância a realizar uma solicitação de qualquer dos serviços do Gerenciador de Estação.

No escalonamento de processos, o NTR usa uma política de preempção a prioridades. A implementação desta política no suporte do SPML é feita com o Gerenciador de Estação se utilizando de facilidades UNIX de sincronização entre processos (*signals*), controlando com isto a execução de seus processos filhos (instâncias de módulos).

Também obedecendo os mecanismos de comunicação e sincronização definidos no ADES, as mensagens entre processos (módulos) devem passar pelo GE para depois seguirem seus destinos finais. Para implementar os serviços de comunicação definidos no modelo de programação do ADES, o suporte do SPML se utiliza dos serviços de socket da camada subjacente. O Gerenciador de Estação além da comunicação local, centraliza também toda a comunicação remota.

Detalhamos a seguir as funções do Gerenciador de Estação, que se resumem em oferecer serviços de criação/destruição de instâncias de módulos, escalonamento, temporização e comunicação. O conjunto de serviços do Gerenciador de Estação disponíveis aos processos de aplicação se encontra no apêndice A.

Criação dos Módulos

Para o cumprimento de todas as funções de gerenciamento em tempo de execução, o Gerenciador de Estação deve possuir uma estrutura que represente a configuração das instâncias dos módulos na sua estação. Essa estrutura é formada por descritores de módulos e instâncias chamados blocos de controle, e por descritores de portos e ligações, contendo os dados relevantes à efetivação da troca de mensagens. Esses descritores estão dispostos em filas como ilustra a figura 4.9.

Estas estruturas são criadas na execução do programa de iniciação da estação, durante a configuração estática da estação. Para cada instanciação de módulo é seguido o seguinte procedimento: primeiramente é criado o bloco de controle do tipo módulo, depois é chamada a função **habilita_tipo**, que verifica se o código do módulo já está carregado e em caso afirmativo, são criados então os blocos de controle e descritores das instâncias desejadas. A criação de uma

instância, além de dar origem às estruturas de controle da instância também cria os descritores dos portos associados à instância.

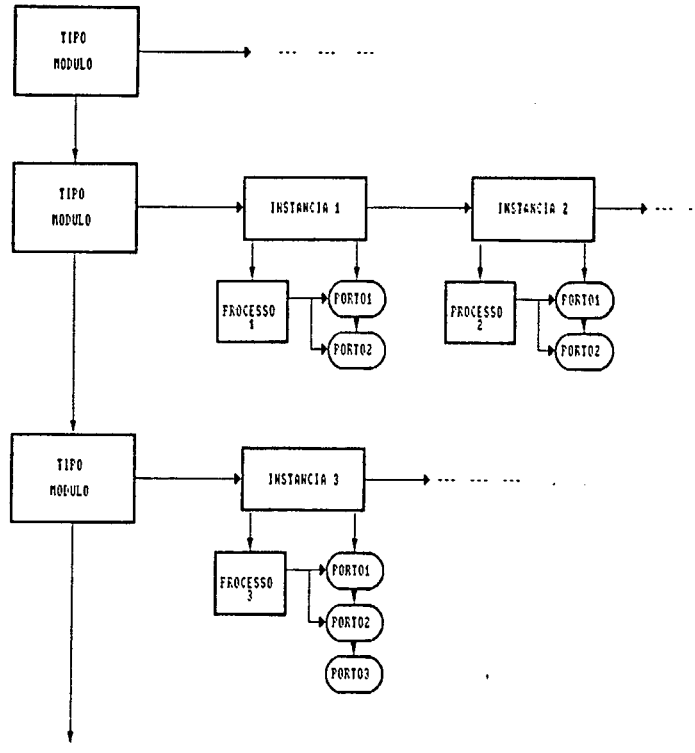

Figura 4.9 Filas de descritores do Gerenciador de Estação

Após a criação destas estruturas são realizadas as conexões dos portos através da função **liga_porto**. O estabelecimento desta ligação, consiste na criação de um descritor de ligação entre os portos relacionados.

Os serviços do Gerenciador de Estação utilizados pelo programa de inicialização para construir as estruturas mencionadas são: **cria_bc_tip_mod, habilita_tipo, cria_ins_mod, inicia_mod, cria_tarefa, cria_porto, liga_porto.**

O Gerenciador de Estação também possui serviços para a destruição de módulos, instâncias e de conexões de portos, que são utilizados principalmente na configuração dinâmica. Estes serviços são analisados no apêndice A.

Escalonamento

A execução dos processos segue uma política de preempção a prioridades controlada por um escalonador *event driven*. Nesta política os processos são

executados obedecendo a prioridade de ocupação do processador, determinada no momento da criação do processo e armazenada em seu respectivo descritor.

O escalonamento do tipo *event driven* é acionado através da função **escalonador()**, não acessível ao usuário, diante da alteração de estado do processo ativo que se dá a partir de eventos de comunicação, interrupção ou temporização. Maiores detalhes da política de escalonamento pode ser encontrados em [Siqueira 92].

Para implementar a política de escalonamento no sistema UNIX, os processos filhos do GE são interrompidos em seguida à sua criação, pela rotina *start*. O escalonador reativa o processo de estado "pronto" mais prioritário. Se este processo que ocupa o processador (processo ativo), muda de estado ou deixa de ser o mais prioritário, o escalonador o interrompe e o substitui pelo processo mais prioritário da fila de "prontos".

O mecanismo de interrupção e reativação de processos é conseguido utilizando os *signals* e a chamada *pause* fornecidos pelo UNIX. O sinal SIGUSR1 realiza a chamada *pause* e o SIGUSR2 reativa o processo enviando um signal através da chamada *kill*. O envio dos sinais SIGUSR1 e SIGUSR2 é realizado pela função **troca_contexto** do Gerenciador de Estação.

Temporização

Os serviços de temporização do GE têm por objetivo central fornecer ao usuário e ao sistema uma base de tempo local, através da atualização periódica do relógio tempo real. Os serviços são: **le_relogio, tempo, KrnReadAbsTime, KrnSetAbsTime.**

Comunicação

Pelo paradigma de programação adotado, as mensagens trocadas entre processos devem passar pelo Gerenciador de Estação, onde ficam armazenadas, se necessário, e por onde são enviadas quando possível, considerando os aspectos de concorrência na estação.

A comunicação inter-módulos é desenvolvida pelas Rotinas de Interface, que solicitam a execução de serviços ao GE através da Interface de Serviços. Desta

forma, quando um processo numa troca de mensagem ativa uma primitiva de envio da Interface de Serviços, através da Rotina de Interface correspondente, os dados a serem transmitidos são passados pela interface, usando os serviços de comunicação subjacente (mecanismo de *socket*) ao Gerenciador de Estação. Este analisa o tipo de envio a ser executado, verifica as ligações dos portos e se o porto está apto a receber a mensagem. A transferência ao processo receptor da mensagem também usa os serviços de *socket*. Para as mensagens que não podem ser transferidas imediatamente aos processos destinatários, por questões de escalonamento, o Gerenciador de Estação possui descritores de mensagens onde são armazenados a mensagem e todos os dados relevantes à efetivação do envio. No caso de comunicação síncrona a mensagem de resposta percorre o caminho inverso, usando as mesmas estruturas (descritores de porto).

Para tornar mais claro o modelo de implementação dos mecanismos de comunicação entre processos/instâncias no SPML, apresentamos a estratificação de serviços usados na implementação destes mecanismos tanto para comunicação local como remota. A figura 4.10 apresenta a estratificação de serviços para comunicação, o modelo OSI/ISO é colocado ao lado das camadas de comunicação remota no sentido de estabelecer comparações. A comunicação remota em nosso protótipo se utiliza de ferramentas e protocolos disponíveis em ambiente UNIX, que preenchem funcionalmente (em grande parte) as necessidades explicitadas no modelo OSI/ISO. Estes protocolos pela difusão do sistema UNIX, estão se tornando padrões de fato. Em nosso sistema os serviços de *socket* se executam sobre a camada Ethernet (serviços de enlace).

A comunicação local faz uso do XDR devido as diferenças de linguagens possíveis em uma estação. A interface de comunicação presente em cada processo/instância não diferencia aspectos de distribuição. Um processo se comunica com um outro processo remoto, usando as mesmas primitivas de comunicação que são utilizadas em comunicações locais, e portanto são encaminhados os dados relativos a comunicação também ao Gerenciador de Estação. É o GE que identifica o porto de entrada como remoto e neste caso utiliza os serviços de *socket* para transferir a mensagem ao GE remoto, no sentido que este complete o envio (figura 4.11).

Mecanismos de comunicação de Sist. Oper.	Comunicação remota no SPML		Comunicação local no SPML
APLICAÇÃO			
APRESENTAÇÃO	XDR		XDR
SESSÃO	SOCKET		SOCKET
TRANSPORTE	TCP	UDP	
REDE	IP (Internetwork)		
LINK	Ethernet	IEEE 802.2	
FÍSICA		IEEE 802.3	

Fig. 4.10 Estratificação de Serviços de Comunicação

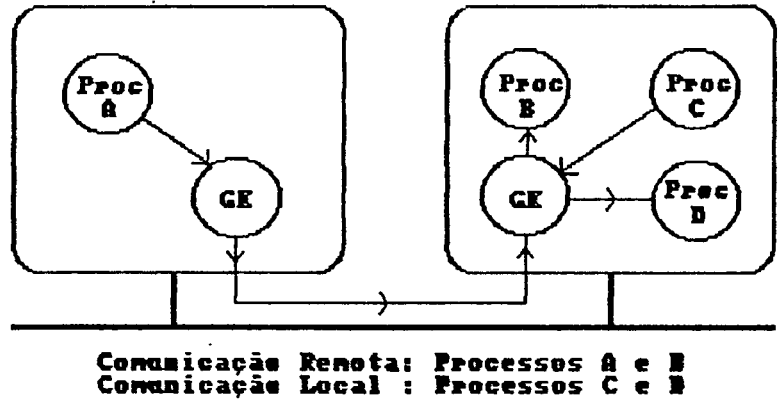

Fig. 4.11 Comunicação Inter-processos (Local e Remota)

4.5.2 Interface de Serviços

A Interface de Serviços está implementada em linguagem C e é ligada durante o processo de compilação, junto ao código fonte do usuário e às Rotinas de Interface do módulo. Assim o programador do módulo pode ter acesso a qualquer serviço do suporte de tempo de execução, desde que implemente a interface (se necessário) para chamar os serviços através da Interface de Serviços e tomar as devidas precauções quanto às conseqüências de suas chamadas. Os serviços do suporte de tempo de execução estão descritos no apêndice A. Para a comunicação inter-módulos, o programador deve utilizar a sintaxe genérica descrita no capítulo três, pois as Rotinas de Interface são responsáveis pela solicitação deste tipo de serviço à Interface de Serviços.

Ocorrendo uma solicitação de serviço, a Interface de Serviços identifica o serviço solicitado criando uma estrutura de dados do tipo solicita_servico (descrita no apêndice A) que especifica o serviço e os parâmetros necessários para a execução da solicitação. Esta estrutura é enviada ao Gerenciador de Estação para a execução do pedido. Se houver um valor de retorno, este é enviado pelo

Gerenciador de Estação à Interface de Serviços para o retorno ao programa chamador.

4.6 Exemplo

Para o exemplo de aplicação do capítulo 3 , temos Rotinas de Interface geradas pelo tradutor da DIM para os módulos codificados nas duas linguagens hospedeiro: C e Fortran. O tradutor da DIM gera além das Rotinas de Interface, códigos suplementares que são incluídos ou utilizados por estas. O primeiro arquivo gerado pelo tradutor referente a uma DIM de um módulo, contém a definição dos tipos (apresentados na DIM) em linguagem C. Este arquivo é incluído no arquivo que contém as rotinas de conversão XDR para os tipos especificados na DIM. Isto acontece, nesta primeira versão, para todas as DIM's independentemente das linguagens hospedeiros, já que as rotinas de conversão XDR estão escritas em C. A figura 4.13 ilustra o arquivo de definição dos tipos em C e as rotinas de conversão XDR. As Rotinas de Interface de cada módulo do exemplo encontram-se no apêndice C, juntamente com a especificação de linguagem necessária para geração de Rotinas de Interface para a linguagem C.

```
/* diretor.h */

#include <rpc/types.h>

typedef struc {
        u_int matel_len;
        float *matel_val;
}matel;
bool_t xdr_matel();

struct autoval {
        float preal[10];
        float pimag[10];
        int   ierr;
};
typedef struct autoval autoval;
bool_t xdr_autoval();
```

```
/* rc_xdr.c */
#include <rpc/rpc.h>
#include "diretor.h"
bool_t
xdr_matel(xdrs,objp)
  XDR *xdrs;
  matel *objp;
{
  if(!xdr_array(xdrs,(char **)objp->matel_val,
(u_int *)&objp->matel_len,100,sizeof(float),xdr_float)){
          return(FALSE);
  }
  return(TRUE);
}
bool_t
xdr_autoval(xdrs,objp)
...
}
```

Fig. 4.12 Códigos Auxiliares às Rotinas de Interface

Agora ilustramos a semântica da comunicação síncrona entre os processos do exemplo, mostrando a interação de cada parte do suporte na realização da comunicação. A figura 4.13 ilustra as camadas por onde a mensagem percorre até chegar ao processo receptor e o caminho inverso da resposta.

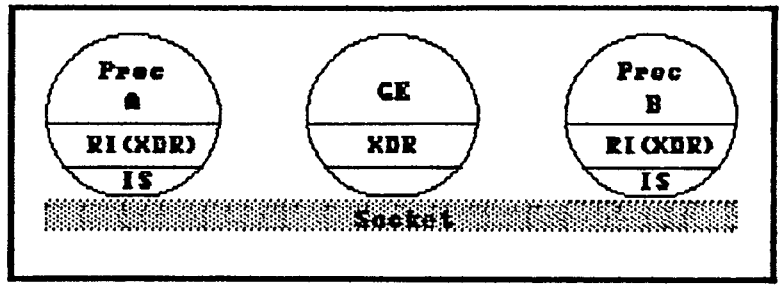

Fig. 4.13 Comunicação Síncrona

Verificamos primeiramente que o processo emissor ativa seu pedido de envio, através da operação **envsin** (Rotina de Interface) que ativa a conversão dos dados para a sintaxe de transferência. Após a conversão é solicitado o serviço à Interface de Serviços que estrutura o pedido contendo a mensagem e o envia ao Gerenciador de Estação. As figuras abaixo mostram as partes significativas dos códigos e as funções envolvidas nas interações do processo emissor de nosso exemplo.

a) Chamada da Rotina de Interface

```
/* diretor.c */
...
typedef float matel[100];
struct autoval {
        float preal[10];
        float pimag[10];
        int ierr;
};
typedef struct autoval
autoval;
...
envsin(ps2,val,...,apval,
...);
...
```

b) Conversão de Tipos e Ativação da Interface de Serviços

```
/* RI */
...
void
envsin(porto,msg,...,rsp,...)
...
{
...
xdr_matel(...);
...
envio(2,...,mem,...,mrsp);
...
xdr_autoval(...);
...
return;
}
```

c) Algoritmo da IS e a Utilização de Serviço de Socket

```
/* IS */
struct sockaddr_un dst,tar_sock;
struct solicita_servico serv;
...
envio(...);
...
{
...
/* monta a estrutura solicita_servico */
serv.ident_servico = ENVIO;
serv.param.msg_ind_porto = ind_porto;
...
/* chama o GE */
solicita_servico();
...
/* retorna resposta */
retorno = retorna_servico();
...
return(retorno);
}
```

Fig. 4.14 Códigos do Módulo C Envolvidos na Comunicação Síncrona

O Gerenciador de Estação recebe a solicitação do serviço, identifica-a como sendo de envio síncrono e realiza a semântica *SEND-WAIT*. São registrados os parâmetros de endereço da mensagem no descritor de porto de saída do processo emissor (DSC_PS), o descritor de porto de entrada do processo receptor (DSC_PE) é localizado, através da estrutura de ligação do DSC_PS. Se o processo receptor encontra-se aguardando a mensagem, esta é enviada via *socket*, através da chamada de uma função, o processo receptor é retirado da fila de espera e colocado na fila de pronto. O DSC_PS é colocado na fila de mensagens do DSC_PE de maneira que o processo receptor possa enviar a mensagem de resposta. O processo emissor é retirado do estado ativo e colocado em estado de espera; Se o processo receptor não se encontra aguardando mensagem neste porto, o DSC_PS é colocado na fila de mensagens do DSC_PE do processo receptor, permitindo que a mensagem seja recebida posteriormente. Assim nas duas condições o processo emissor é colocado na fila de espera, permanecendo até a chegada da resposta ou o esgotamento do *timeout*.

Quando o processo receptor recebe a mensagem, os dados da mensagem passam pela sequência de rotinas que os convertem e os transferem ao programa de cálculo de autovalores (de forma similar, porém inversa, ao processo emissor). Após o processamento necessário é produzida a resposta (ativada pela operação **resposta**) e os dados da mensagem de retorno são enviados ao GE e deste ao processo emissor (descrição que omitimos por ser semelhante ao envio descrito).

Assim o processo emissor recebe a resposta, através da função **retorna_serviço** (IS). Os dados são convertidos na RI (xdr_autoval) e finalmente o programa usuário continua seu processamento.

4.7 Conclusão

Neste capítulo foram apresentados os principais aspectos de implementação do SPML, segundo os trabalhos em curso no LCMI. As ferramentas de programação foram apresentadas, especificando suas estruturas e funcionalidades associadas. Em particular, foi salientado o processo de geração das Rotinas de Interface, apresentando os comandos de geração presentes nas especificações de linguagem.

O suporte de tempo de execução foi implementado tendo como base o sistema UNIX. Neste capítulo foram tratados vários pontos desta implementação, envolvendo

desde a disposição destes serviços até a estratificação envolvendo a plataforma UNIX.

Também ilustramos a ativação dos diversos mecanismos e algoritmos envolvidos no envio referente ao exemplo do capítulo 3, mostrando o encadeamento das diferentes camadas definidas na transferência de uma mensagem entre ambientes heterogênenos de linguagens.

CAPÍTULO 5

CONCLUSÃO E PERSPECTIVAS FUTURAS

O SPML foi originado da experiência inicial desenvolvida no LCMI com o ambiente distribuído ADES. Este sistema permitiu constatar, apesar das facilidades de desenvolvimento de aplicações proporcionadas pelo seu paradigma de programação, as dificuldades em utilizar software pré-existente escritos em outras linguagens. Em adição o ADES não possibilita nenhum tratamento específico da heterogeneidade entre máquinas.

O SPML vem neste sentido suprir uma lacuna em parte já preenchida por algumas linguagens propostas (Cap. 2), mas que no entanto diferem consideravelmente do modelo proposto. Tratamos então, neste trabalho do projeto de um suporte para programação distribuída em múltiplas linguagens, uma ferramenta útil e simples que permite a interação de componentes de software (subprogramas) escritos em diferentes linguagens.

Para atender estes objetivos, vários mecanismos foram concebidos e sucintamente detalhados em dois capítulos da presente dissertação. Podemos citar entre eles: a Linguagem de Descrição de Interface de Módulo (DIM), a sua tradução e a geração das Rotinas de Interface de Módulo.

Na continuidade do trabalho pretendemos:

- Implementar completamente este suporte, introduzindo abordagens dinâmicas de estruturação de dados nas unidades compiladoras. A implementação deve-se dar inicialmente em estações de trabalho Sun-4, devendo-se estender posteriormente para outras arquiteturas.

- Introduzir métodos eficazes de composição e decomposição de tipos sem correspondência com os suportados pela linguagem XDR;

- Incorporação de toda interface com as primitivas do ADES às Rotinas de Interface;

- Implementação de rotinas de conversão XDR nas demais linguagens suportadas pelo ambiente;

- Ampliar o número de linguagens suportadas pelo ambiente, introduzindo também linguagens concorrentes.

- Testes para a verificação do desempenho do sistema, através da verificação das velocidades de troca de mensagens entre módulos.

Por fim, realizar-se-áo desenvolvimento de aplicações no ambiente totalmente integrado, onde serão verificadas as vantagens e desvantagens desta abordagem, proporcionando a determinação de novos objetivos para desenvolvimentos futuros.

BIBLIOGRAFIA

[Abreu 91]: Abreu W. M. B. - "Metodologia e Suporte para Programação de Aplicações Distribuídas Flexíveis", Dissertação de Mestrado, CPGEEL, UFSC, Florianópolis, Março 1991.

[Aho 89]: Aho A.; Sethi R.; Ullman J. - "Compilateurs Principles, Techniques et Outils", Inter Editions, Paris, 1989.

[Andrews 82]: Andrews G. R. - "The Distributed Programming Language SR - Mechanisms, Design and Implementation", Software - Practice and Experience, vol 12, pp 719-753, March 1982.

[Bal 89]: Bal H. E.; Steiner J. G.; Tanenbaum A. S. - "Programming Languages for Distributed Computing Systems", ACM Computing Surveys: Special Issue on Programming Language Paradigms, vol 21, n 3, pp 261-322, September 1989.

[Barbacci 86]: Barbacci M. R.; Doubleday D. L.; Weinstock C. B. - "The Durra Runtime Environment", Technical Report CMU/SEI-88-TR-18 ESD-TR-88-19, Software Engineering Institut, Carnegie Mellon University, Pittsburg, July 1986.

[Barbacci 89]: Barbacci M. R.; Wing J. M. - "Durra: A Task-Level Description Language Reference Manual (Version 2)", Technical Report CMU/SEI-89-TR-34 ESD-TR-89-45, Software Engineering Institut, Carnegie Mellon University, Pittsburg, September 1989.

[Barnes 80]: Barnes J. G. P. - "An Overview of ADA", Software - Practice and Experience, vol 10, pp 851-887, July 1980.

[Birrel 84]: Birrel A. D.; Nelson B. J. - "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, vol 2, n 1, pp 39-59, February 1984.

[Black 86]: Black A.; Hutchinson N.; Jul E.; Levy H. - "Object Struture in the Emerald System", In Proceedings of Object-Oriented Programming Systems, Languages and Applications, SIG-PLAN not. (ACM) vol 21, n 11, pp 78-86, November 1986.

[Cohen 81]: Cohen D. - "On Holy Wars and a Plea for Peace", IEEE Computer, vol 14, n 10, pp 48-54, October 1981.

[Corbin 90]: Corbin J. R. - "The Art of Distributed Aplications", Sun Technical Reference Library, Springer-Verlag, 1990.

[Dannenberg 82]: Dannenberg R. B. - "Resource Sharing in a Network of Personal Computers", PhD Dissertation, Carnegie-Mellon University, CMU-CS-82-152, Pittsburgh, Pennsylvania, December 1982.

[DeRemer 76]: DeRemer F.; Kron H. H. - "Programming-in-the-Large versus Programming-in-the-Small", IEEE Transactions on Software Engineering, vol SE-2, n 2, pp 80-86, June 1976.

[Einarsson 84]: Einarsson B.; Gentleman W. M. - "Mixed Language Programming", Software - Practice and Experience, vol 14,n 4, pp 383-395, April 1984.

[Fraga 89]: Fraga J. S.; Farines J. M.; Abreu W. M. B.; Nacamura Jr. L.; Coelho F. O. - "ADES: Ambiente de Desenvolvimento e Execução de Software Distribuído", Actes du Sèminaire Franco-Brésilien sur les Systèmes Informatiques Répartis, Florianópolis, SC, Brasil, September 1989.

[Frave 89]: Frave J. M.; Estublier J.; Equipe ADELE, "Structuring Large Versioned Software Products", Published in Proceeding 13th Annual International Computer Software and Applications Conference, Orlando (Florida), September 1989, pp 404-411, IEEE Computer Society Press.

[Gibbons 87]: Gibbons P. B. - "Stub Generator for Multilanguage RPC in Heterogeneous Environments", IEEE Transactions on Software Engineering, vol SE-13, n 1, pp 77-87, January 1987.

[Hansen 78]: Hansen P. B. - "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, vol 21, n 11, pp 934-941, November 1978.

[Hayes 86]: Hayes R.; Schlichting R. D. - "Facilitating Mixed Language Programming in Distributed System", Technical Report 85-11a, Department of Computer Science, University of Arizona, March 1986.

[Hayes 88]: Hayes R.; Manweiler S. W.; Schlichting R. D. - "A Simple System for Constructing Distributed, Mixed Language Programs", Technical Report 85-11a, Department of Computer Science, University of Arizona, March 1986.

[Hayes 89]: Hayes R.; Hutchinson N. C.; Schlichting R. D. - "Integrating Emerald into a System for Mixed-Language Programming", Comput. Language, vol 15, pp 95-108, 1990.

[ISO 87a]: ISO IS 8824 - "Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)", 1987.

[ISO 87b]: ISO IS 8825 - "Information Processing - Open Systems Interconnection - Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)", 1987.

[Johnson 78]: Johnson S.C. - "YACC: Another Compiler Compiler", Bell Laboratories, July 1978.

[Jones 85]: Jones M. B.; Rashid R. F.; Thompson M. R. - "Matchmaker: An Interface Specification Language for Distributed Processing", Published in Proceedings of 12th ACM Symposium of Principles of Programming Language, January 1985.

[Jul 88]: Jul E., Levy H.; Hutchinson N.; Black A. - "Fine-Grained Mobility in Emerald System", ACM Transactions on Computer Systems, vol 6, n 1, pp 109-133, February 1988.

[Kowaltowski 83]: Kowaltowski T. - "Implementação de Linguagens de Programação", Editora Guanabara, Rio de Janeiro 1983.

[Kramer 83]: Kramer J.; Magee J.; Sloman M.; Lister A. - "Conic: An Integrated Approach to Distributed Computer Systems", IEE Proceedings, vol 130, Pt_E, n 1, pp 1-10, January 1983.

[Kramer 85]: Kramer J.; Magee J. - "Dynamic Configuration of the Distributed Systems", IEEE Transactions on Software Engineering, SE-11, vol 4, pp 425-436, April 1985.

[Lesk 82]: Lesk M. E.; Schimidt E. - "LEX - A Lexical Analyser Generator", Bell Laboratories, August 1982.

[Liskov 87]: Liskov B.; Curtis D.; Johnson P.; Scheifler R. - "Implementation of Argus", In Proceedings of the 11th Symposium on Operating Systems Principles, ACM-SIGOPS, pp 111-122, New York, November 1987.

[Maccabe 82]: Maccabe A. B. - "Language Features for Fully Distributed Processing Systems", Georgia Institute of Technology, G/T - ICS - 82/12, Aug 1982.

[Magee 87]: Magee J. N.; Kramer J.; Sloman M. - "Construction Distributed Systems in CONIC", Research Report Doc. 87/4, Department of Computing, Imperial College, London, March 1987.

[McGuffin 88]: McGuffin L. J. et al. - "MAP/TOP in CIM Distributed Computing", IEEE Network, vol 2, n 3, May 1988.

[McQuillan 77]: McQuillan J. M.; Walden D. C. - "The ARPA Network Design Decisions", Comput. Networks, vol 1, pp 243-289, August 1977.

[Nacamura 88]: Nacamura Jr. L. - "Projeto e Implementação de um Núcleo de Sistema Operacional Distribuído com Mecanismos para Tempo Real", Dissertação de Mestrado, CPGEEL, UFSC, Florianópolis, Julho 1988.

[Nelson 81]: Nelson B. J. - "Remote Procedure Call", PhD. Dissertation, Dep. Comput. Science, Carnegie-Mellon University, Pittsburgh, PA, Tech. Report CMU-CS-81-119, 1981.

[Rashid 81]: Rashid R. F.; Robertson G. G. - "Accent: A Communication Oriented Network Operating System Kernel", 8th Symposium on Operating Systems Principles, Pacifc Grove, California, pp 64-75, December 1981.

[Rifflet 90]: Rifflet J. M. - "La programmation sous UNIX". McGraw-Hill, 1990.

[Setzer 85]: Setzer V. W.; Melo I. S. H. - "A Construção de um Compilador", Editora Campus, Rio de Janeiro 1985.

[Silva 88]: Silva E. S. - "Uma Linguagem de Programação de Componentes Elementares para Aplicações Distribuídas em Tempo Real: Projeto e Implementação", Dissertação de Mestrado, CPGEEL, UFSC, Florianópolis, Agosto 1988.

[Siqueira 92]: Siqueira F. A.; Silva E. S. - "O Gerenciador de Estação ADES sobre o Ambiente UNIX", Dep. de Eng. Elétrica-LCMI-UFSC, Nota Interna, Março 1992.

[Souza 88]: Souza L. E. - "Um Suporte para Configuração Estática de Sistemas Distribuídos Utilizando Abordagem por Linguagem: Projeto e Implementação", Dissertação de Mestrado, CPGEEL, UFSC, Florianópolis, Dezembro 1988.

[Sun Microsystems 90]: Sun Microsystems Inc., Manuais de Referência (13), Part 800-1177-01 Revision A, Mountain View, CA, March 1990.

[Sweet 85]: Sweet R. E. - "The MESA Programming Environment", SIGPLAN Notices, 1985.

[Weinstock 89]: Weinstock C. B. - "Fault Tolerance in Durra", Software Engineering Institut, Carnegie Mellon University, Pittsburg, 1989.

[Wirth 82]: Wirth N. - "Programming in Modula-2", Springer-Verlag, 1982.

APÊNDICE A

A interface de serviços, também chamada Krnlib, oferece entrada para trinta e três serviços do Gerenciador de Estação, os quais fornecem ao usuário ferramentas para programação em sistemas distribuídos. Eles se dividem em serviços de comunicação, de temporização e de configuração dinâmica. Os dois primeiros são mostrados em detalhe, os serviços de configuração dinâmica são listados e descritos ligeiramente, mas podem ser analisados em "O Núcleo de Tempo Real" - Nota técnica LCMI - Apêndice : Interfaces do Núcleo, porque a implementação dos serviços oferecidos pelo Gerenciador de Estação segue o padrão dos serviços oferecidos pelo Núcleo de Tempo Real do sistema ADES [Nacamura, 88].

Serviços de Comunicação

1) envio (ind_porto, msg_tam, base_msg, ofs_msg, timeout, base_rsp, ofs_rsp)
```
int    ind_porto, timeout, msg_tam;
char   *base_msg, *ofs_msg, *base_rsp, *ofs_rsp;
```

Função : Realiza o envio de mensagem entre módulos através de portos.

Parâmetros de Entrada :

ind_porto	:	índice do porto de saída.
timeout	:	tempo máximo de espera do processo.
base_msg, base_rsp	:	base do segmento em que se encontram as mensagens de envio e resposta.
ofs_msg, ofs_rsp	:	indicam o *offset* das mensagens de envio e resposta, respectivamente.

Parâmetros de Retorno : Não há.

2) falha (ind_envio)
```
int    *ind_envio;
```

Função : Permite a um processo, após o envio, detectar a razão pela qual retornou ao estado de pronto.

Parâmetros de Entrada :

ind_envio : índice do envio; retorna atualizado indicando:
- 0 - timeout esgotado
- 1 - envio normal;
- 2 - PS não ligado
- 3 - *buffer* do PE assíncrono esgotado

Parâmetros de Retorno :
Status(int) indicando :
0 - timeout esgotado
1 - envio normal;
2 - PS não ligado

3) inicia_recepcao (ind_porto, base_msg, ofs_msg, ind_for, clausula)
int ind_porto, ind_for, clausula;
char *base_msg, *ofs_msg;

Função : Habilita o processo para a recepção da mensagem.

Parâmetros de Entrada :
ind_porto : índice do porto de entrada.
base_msg : base do segmento da mensagem a ser recebida.
ofs_msg : *offset* da mensagem de envio.
ind_for : sem uso atualmente.
clausula : identifica o PE para um conjunto de recepções.

Parâmetros de Retorno : Não há.

4) recepcao (timeout)
int timeout;

Função : Possibilita a um processo receber mensagem.

Parâmetros de Entrada :
timeout : tempo que o processo permanecerá em espera.

Parâmetros de Retorno : Não há.

5) condicao_recepcao (ind_for_ptr, clausula_ptr)
int *ind_for)ptr, *clausula_ptr;

Função : Indica a razão pela qual o processo retornou ao estado de pronto.

Parâmetros de Entrada :
ind_for_ptr : sem uso atualmente.
clausula_ptr : endereço da variável na qual é retornada a cláusula do PE em que o processo recebeu mensagem (0 se o timeout se esgotou).

Parâmetros de Retorno : Não há.

6) responde (ind_porto, tam_rsp, base_rsp, ofs_rsp)
int ind_porto, tam_rsp;
char *base_rsp, *ofs_rsp;

Função : Permite o envio de uma mensagem de resposta.

Parâmetros de Entrada :

ind_porto	:	índice do porto de entrada.
tam_base	:	tamanho da mensagem de resposta.
base_rsp	:	base do segmento da mensagem de resposta.
ofs_rsp	:	*offset* da mensagem de resposta.

Parâmetros de Retorno : Não há.

Serviços de Temporização

1) le_relogio (relogio)
unsigned int *relogio;

Função : Retorna o valor atual do relógio tempo real.

Parâmetros de Entrada :
relogio : tem seu valor atualizado com o valor do relógio tempo real.

Parâmetros de Retorno : Não há.

2) tempo()

Função : Incrementa o relógio tempo real e verifica se o tempo de espera do processo atual na fila de espera não se esgotou.

Parâmetros de Entrada : Não há.

Parâmetros de Retorno : Não há.

3) KrnReadAbsTime (AbsTime)
long int *AbsTime;

Função : Lê o tempo global no instante de sua chamada.

Parâmetros de Entrada :
AbsTime : retorna atualizado com o valor do tempo global no instante da chamada da função.

Parâmetros de Retorno : Não há.

4) KrnWaitAbsTime (AbsTime)
long int *AbsTime;

Função : Coloca o processo solicitante em espera por um tempo global na *Time Table* e a transfere da fila de prontas para a fila de espera.

Parâmetros de Entrada :
AbsTime : tempo absoluto de espera.

Parâmetros de Retorno : Não há.

Outros Serviços

1) muda_prioridade
Função : Muda a prioridade do módulo.

2) suspende_modulo
Função : Suspende o processo de uma instância de módulo.

3) cria_porto
Função : Cria um descritor de porto para um módulo instanciado.

4) liga_porto
Função : Liga um porto de entrada a um porto de saída.

5) espera_temporizada
Função : Coloca um processo em espera.

6)inicia_interrupcao
Função : Inicia uma interrupção.

7) espera_interrupcao
Função : Verifica a ocorrência de uma interrupção.

8) cria_bc_tip_mod
Função : Cria o bloco de controle do tipo módulo.

9) cria_ins_mod
Função : Cria instância de um módulo.

10) KrnParaModuloAsy
Função : Interrompe o processo de um módulo assíncrono.

11) KrnParaModuloSinc
Função : Interrompe o processo de um módulo síncrono.

12) destroi_ins_mod
Função : Destrói uma instância de um módulo.

13) destroi_bc_tip_mod
Função : Destrói um tipo módulo.

14) inicia_mod
Função : Inicia um módulo.

15) habilita_tipo
Função : Indica no bloco de controle de um tipo que o módulo pode ser instanciado porque seu código já foi carregado.

16) KrnReIniciamodulo
Função : Coloca na fila de prontos todos os processos suspensos por estado de configuração.

17) KrnCheckTaskSusp
Função : Verifica o número de processos ainda não suspensos.

18) KrnVerificaConfig
Função : Verifica o estado de configuração da instância a qual pertence o processo ativo naquele instante.

19) KrnWaitSuspTasks
Função : Suspende o processo configurador por um tempo determinado indicado como parâmetro na chamada da função.

20) religa_porto
Função : Religa um porto de saída a um porto de entrada.

21) desliga_porto
Função : Desliga um porto de saída de um porto de entrada.

22) KrnSetAbsTime
Função : Seta o tempo global para o valor indicado como parâmetro.

Para identificar o serviço e criar a estrutura de dados correspondente aos parâmetros do serviço, a Krnlib possui uma estrutura em union denominada solicita_servico que está implementada em C da seguinte maneira:

```
typedef struct solicita_servico
   {
        int ident_sevico;
        union {
                struct mod {
                   int  ind_tip_mod, ind_ins_mod, timeout;
                   unsigned int  cs_tam, ds_tam, ss_tam, param_tam, db_base, cs_base, task_susp;
                   char  iniciacao[10], base_param[10], ofs_param[10];
                }mod;

                struct tar {
                   char  tar_inicio[100], tar_seg[10];
                   int  prioridade, timeout;
                   unsigned int  ss_tam;
                   BOOLEAN  estado;
                }tar;

                struct porto_l {
                   int  ind_ins_mod_pe, ind_ins_mod_ps, ind_pe, ind_ps, ind_estacao_pe, ind_estacao_ps;
                }porto_l;

                struct porto_c {
                   int  identificador, ind_porto, num_max_msg, prioridade;
                }porto_c;

                struct msg {
                   int  ind_porto, timeout, msg_tam, ind_for, clausula, tam_rsp;
                   int  ind_envio, ind_for_ptr, clausula_ptr;
                   char  base_rsp[10], ofs_rsp[10], base_msg[10], ofs_msg[10];
                }msg;

                struct intr {
                   unsigned char  ind_vet_int;
                   int  ind_int, nivel;
                }intr;

                struct time {
                   long int  AbsTime;
                   long int  DeadTime;
                }time;
           }param;
```

APÊNDICE B

Este apêndice especifica o protocolo XDR que tem sido usado para transferir dados entre diferentes arquiteturas de computadores. Ele foi projetado para o uso entre diferentes linguagens, sistemas operacionais e arquiteturas de máquinas. XDR é levemente análogo à proposta para X.409 da ISO , o ASN-1 (*Abstract Syntax Notation*), a maior diferença entre as duas abordagens é que XDR usa tipagem implícita, enquanto o X.409 utiliza tipagem explícita.

Inicialmente mostramos a especificação do protocolo XDR e depois descrevemos um pequeno tutorial das rotinas em C, da biblioteca XDR. Maiores detalhes podem ser obtidos em *Network Programming Guide* [SunMicrosystem 90].

Sintaxe da linguagem XDR

A sintaxe na notação XDR é bastante similar à linguagem C. Em XDR existem quatro tipos básicos de declarações: a declaração simples, a de *array* de comprimento fixo, a de *array* de comprimento variável e a declaração de apontador. Também existem declarações especiais para *string*, *opaque data* (semelhantes à declaração de *array*) e para o tipo *void*, que serão apresentadas posteriomente.

As declarações simples e de *array* de comprimento fixo são idênticas às utilizadas pela linguagem C:

```
declaração ::= tipo  ident_variável
declaração ::= tipo  ident_variável"[" valor "]"
```

As declarações de *arrays* de comprimento variável não tem sintaxe explícita em C, então o XDR utiliza parênteses angulares < e >, entre os quais indica-se o tamanho máximo do *array*, se o valor for omitido o *array* é de qualquer tamanho:

```
declaração ::= tipo ident_variável"<" [valor] ">"
```

As declarações de apontadores são feitas em XDR exatamente como em C. Não se pode utilizar apontadores em sistemas distribuídos, mas pode-se usá-los para enviar tipos de dados recursivos tais como listas e árvores. O tipo é chamado de *opcional data*:

declaração ::= tipo "*" ident_variável

onde:

```
tipo ::=    ["unsigned"] "int"
            ¦["unsigned"] "hyper"
            ¦"float"
            ¦"double"
            ¦"bool"
            ¦t_enum
            ¦t_struct
            ¦t_union
```

B 1. Declaração de Inteiro

tipo ::=["unsigned"] "int" ¦["unsigned"] "hyper"

Um inteiro com sinal em XDR é um dado com 32 bits que codifica um inteiro entre os valores -2147483648 e 2147483647. O inteiro sem sinal também com 32 bits codifica um inteiro positivo entre 0 e 4294967295. XDR também suporta o chamado "*hyper int*" definido com 64 bits que é uma extensão do inteiro com e sem sinal.

B 2. Declaração de Real

tipo ::="float" ¦"double"

Para representar valores de ponto flutuante XDR possui o tipo "*float*" de precisão simples que contém três campos, o *bit* de sinal, o expoente do número, em base 2 com 8 *bits* e a parte fracionária da mantissa do número, em base 2 com 23 *bits*. XDR usa o "*IEEE standard*" para codificar reais de precisão simples.

Seguindo também a normalização do IEEE o tipo "*double*" também possui três campos, com 64 *bits*, reservando 1 *bit* para o sinal, 11 para o expoente em base 2 e 52 para a parte fracionária da mantissa do número.

B 3. Declaração de Booleano

tipo ::= "bool"

O tipo pré-definido booleano é um tipo enumerado que tem dois valores possíveis: *TRUE* e *FALSE*. É resultado de expressões booleanas que geralmente apresentam operadores lógicos.

B 4. Declaração de Tipo Enumerado

```
tipo      ::= t_enum
t_enum    ::= "enum" [etiqueta] c_enum
c_enum    ::= "{"
              (identificador "=" valor)
              ("," identificador "=" valor) *
              "}"
etiqueta  ::= identificador
```

O tipo enumerado tem a mesma representação que inteiros com sinal. São muito úteis para descrever subconjuntos de inteiros.

Pode ser incluída uma etiqueta para definir um novo nome representando o tipo enumerado. O conjunto de identificadores denotando os valores do tipo podem ser usados como constantes, favorecendo a compreensão.

Exemplo:

```
enum cores { AZUL =2, AMARELO = 3, VERMELHO = 5 } flag;
```

B 5. Declaração de Estrutura

```
tipo      ::= t_struct
t_struct::= "struct" [etiqueta] c_struct
c_struct::= "{"
              (declaração ";")+
              "}"
etiqueta::= identificador
```

A semântica é a mesma do "*struct*" do C e do "*record*" do Pascal. Compreendendo um conjunto de componentes, que podem ser de tipos diferentes.

A etiqueta opcional atribui um nome de tipo à estrutura definida, sendo usada posteriormente como uma abreviação da descrição detalhada.

Exemplo:

```
struct data {
        int dia;
        string nome_mes[4];
        int ano;
};
data d;          /* define a variável d como sendo
                 uma estrutura do tipo data */
```

B 6. Declaração de Tipo União Discriminada

```
tipo     ::=t_union
t_union ::="union" [etiqueta] c_union
c_union ::="switch" "("t_enum")" "{"
           ( "case" valor ":" declaração";")+
           [ "default"     ":" declaração ";"]
           "}"
etiqueta::=identificador
```

Semelhante ao *record* variante do Pascal, o tipo união discriminada é uma união C e um valor enumerado que seleciona um "braço" da união. Cada tipo componente, ou braço da união, é precedido pelo valor do discriminante que implica em sua execução. A declaração do discriminante é feita especificando um tipo enumerado seguido pelo seu identificador, ou seja, a variável que conterá o valor do discriminante.

Somente um tipo componente é selecionado. O braço *default* é opcional e executado se nenhum dos outros casos forem satisfeitos. Para uma codificação válida da união, não havendo braço *default*, o valor do discriminante deve selecionar um dos casos.

Como nas declaraçãoes dos tipos "*struct*" e "enum" também na declaração de "*union*" a etiqueta atribui um novo nome de tipo à união definida.

Exemplo:

```
enum filekind { TEXTO = 0, EXEC = 1};
union tipoarq switch (filekind opcao) {
              case TEXTO : void;
                       case EXEC  : string interp<255>;
          }
```

B 7. Declaração de Array

```
declaração::= tipo  identificador "[" valor "]"
             |tipo  identificador "<" [valor] ">"
```

Arrays consistem de componentes que são do mesmo tipo; os elementos são numerados de 0 a valor-1, onde 'valor' é um inteiro sem sinal. Em 'tipo' supõem-se um tipo XDR. Quando o *array* tem comprimento variável e o tamanho máximo de elementos não é especificado, assume-se o valor de (2** 32)-1. O identificador é a variável do tipo *array* especificado.

Exemplo:

```
float dados<50>;
```

B 8. Declaração de String

declaração::="string" identificador"<" [valor] ">"

Um tipo *string* em XDR é uma sequência de caracteres ASCII, terminada com um *byte* nulo que não é considerado no cálculo do comprimento da *string*. O identificador é a variável do tipo *string* especificado. O comprimento máximo de caracteres é especificado com um inteiro sem sinal entre os parênteses angulares (sem contar o byte nulo), em caso de omissão é assumido (2**32)-1 como comprimento máximo.

B 9. Declaração do Tipo Opaque-data

declaração::= "opaque" identificador "[" valor "]"
¦"opaque" identificador "<" [valor] ">"

Às vezes, dados não interpretados precisam ser passados entre máquinas. Este dado é chamado "opaco" e é uma sequência de *bytes* arbitrários. Seu comprimento é declarado com um inteiro sem sinal em 'valor',e se omitido, no caso de comprimento variável, assume-se (2**32)-1 como comprimento máximo. O identificador é a variável do tipo opaco especificado.

B 10. Declaração do Tipo opcional-data

declaração ::= tipo "*" identificador

O tipo opcional é um tipo de *union* que ocorre com frequência e ao qual foi dada uma sintaxe especial para declará-lo, utilizando o terminal "*".

```
        nome_do_tipo * identificador;
```

Que é equivalente à seguinte declaração union :

```
    union switch (bool opted) {
    case TRUE :
    nome_do_tipo elemento;
    case FALSE :
    void;
} identificador;
```

A sintaxe do tipo opcional é a mesma da utilizada para apontadores na linguagem C. Uma vez que não se utiliza apontadores na comunicação de dados remota, ela não aparece como uma declaração única, mas dentro de uma estrutura referenciando-se a outra.

Exemplo: Definindo o tipo "stringlist" que codifica uma lista de *strings* de comprimentos arbitrários.

```
struct * stringlist {
                string item<>;
                stringlist next;
};
```

B 11. Declaração do Tipo void

declaração ::= "void"

A declaração é simplesmente "*void*". Um *void* em XDR não tem comprimento (0-*byte*). Eles são muito úties em "*unions*", onde alguns braços podem conter dados e outros não, e para descrever operações que não levam nenhum dado como entrada ou saída.

B 12. Declaração de Constantes

def_constante ::= "const" identificador "=" cte ";"

"const" é usada para definir um nome simbólico para uma constante inteira. Pode ser usada onde qualquer constante regular possa ser utilizada.

Exemplo:

```
const DOZEN = 12;
```

B 13. Declaração de Definição de Tipo

```
def_tipo    ::= "typedef" declaração ";"
              ¦"enum"   etiqueta c_enum ";"
              ¦"struct" etiqueta c_struct ";"
              ¦"union"  etiqueta c_union  ";"
```

A definição de tipo não cria tipo algum, somente serve para criar novos nomes de tipos de dados já existentes. Tem a mesma sintaxe que as definições de

tipo em C. O nome de um tipo é o nome da variável na parte da declaração de uma "typedef". Porém difere da linguagem C nas outras formas de definição de nomes de tipos quando são usadas etiquetas para representar o tipo definido, como já mostrado nos itens B (5,6,7).

Sinopse das Rotinas da Biblioteca XDR

A rotina xdrmem_create() inicializa um stream XDR em um local na memória:

```
void
xdrmem_create(xdrs, addr, len, x_op)
        XDR             *xdrs;
        char            *addr;
        u_int   len;
        enum    xdr_op x_op;
```

Filtros primitivos XDR:

```
#define bool_t int
#define TRUE   1
#define FALSE  2

bool_t xdr_char(xdrs, cp)
        XDR             *xdrs;
        char            *cp;


bool_t xdr_u_char(xdrs, ucp)
        XDR             *xdrs;
        unsigned char   *ucp;


bool_t xdr_int(xdrs, ip)
        XDR             *xdrs;
        int             *ip;

bool_t xdr_u_int(xdrs, up)
        XDR             *xdrs;
        unsigned        *up;


bool_t xdr_long(xdrs, lip)
        XDR             *xdrs;
        long            *lip;


bool_t xdr_u_long(xdrs, lup)
        XDR             *xdrs;
        u_long  *lup;


bool_t xdr_short(xdrs, sip)
        XDR             *xdrs;
        short   *sip;


bool_t xdr_u_short(xdrs, sup)
        XDR             *xdrs;
        u_short *sup;


bool_t xdr_float(xdrs, fp)
        XDR             *xdrs;
        float   *fp;
```

```
bool_t xdr_double(xdrs, dp)
        XDR             *xdrs;
        double  *dp;
```

Filtros Enumerados:

```
#define enum_t  int


bool_t xdr_enum(xdrs, ep)
        XDR             *xdrs;
        enum_t  *ep;


bool_t xdr_bool(xdrs, bp)
        XDR             *xdrs;
        bool_t  *bp;
```

Filtro para o tipo void:

```
bool_t xdr_void()       /* Sempre retorna TRUE */
```

Filtros para tipos construídos:
Filtro para string:

```
bool_t xdr_string(xdrs, sp, maxlength)
        XDR             *xdrs;
        char            **sp;
        u_int   maxlength;
```

Filtro para array de bytes:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
        XDR             *xdrs;
        char            **bpp;
        u_int   *lp;
        u_int   maxlength;
```

Filtro para array fixo:

```
bool_t xdr_vector(xdrs, ap, size, elementsiz, xdr_element)
        XDR             *xdrs;
        char            **ap;
        u_int   size;
        u_int   elementsiz;
        bool_t  (*xdr_element) ();
```

Filtro para array variável:

```
bool_t xdr_array(xdrs, ap, lp, maxlength, elementsiz, xdr_element)
        XDR             *xdrs;
        char            **ap;
        u_int   *lp;
        u_int   maxlength;
        u_int   elementsiz;
        bool_t  (*xdr_element) ();
```

Filtro para opaque data:

```
bool_t xdr_opaque(xdrs, p, len)
        XDR             *xdrs;
        char            *p;
        u_int   len;
```

Filtro para união discriminada:

```
struct xdr_discrim {
        enum_t value;
        bool_t (*proc)();
};

bool_t xdr_union(xdrs, dscmp, unp, arms, defaultarm)
        XDR             *xdrs;
        enum_t *dscmp;
        char    *unp;
        struct xdr_discrim  *arms;
        bool_t  (*defaultarm) ();
```

Filtro para apontadores:

```
bool_t xdr_reference(xdrs, pp, size, proc)
        XDR             *xdrs;
        char            **pp;
        u_int   size;
        bool_t  (*proc) ();
```

Outras primitivas XDR que não são filtros:

```
u_int xdr_getpos(xdrs)
        XDR             *xdrs;

bool_t xdr_setpos(xdrs, pos)
        XDR             *xdrs;
        u_int   pos;

xdr_destroy(xdrs)
        XDR             *xdrs;
```

Implementação do stream XDR:

```
enum xdr_op {XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2};

typedef struct {
        enum xdr_op x_op;
        struct xdr_ops {
                bool_t  (*x_getlong)();
                bool_t  (*x_putlong)();
                bool_t  (*x_getbytes)();
                u_int   (*x_putbytes)();
                bool_t  (*x_getpostn)();
                bool_t  (*x_setpostn)();
                caddr_t (*x_inline)();
                VOID    (*x_destroy)();
        } *x_ops;
        caddr_t x_public;
        caddr_t x_private;
        caddr_t x_base;
        int         x_handy;
} XDR;
```

APÊNDICE C

Neste apêndice mostramos os códigos de implementação em C e Fortran referentes ao exemplo dos capítulos 3 e 4, juntamente com os códigos que devem ser gerados pelo Gerador de Rotinas de Interface (GRI). Também descrevemos a especificação de linguagem (não completa) para a linguagem hospedeiro C, dividida em geração de: Cabeçalho, Rotinas de Conversão XDR e Rotinas de Interface. E finalmente a especificação de máquina utilizada.

Primeiramente mostramos os códigos de implementação do módulo C (rotina cliente) e do módulo Fortran (rotina servidor). Depois o cabeçalho gerado, contendo as definições em C dos tipos definidos nas DIM's, este cabeçalho é incluído no arquivo de Rotinas de Conversão XDR, também gerado pelo GRI. Outro código de conversão de dados auxiliar para o módulo Fortran é apresentado (xdr_converte). Os dois últimos códigos gerados pelo GRI são as rotinas de interface propriamente ditas.

Finalizando apresentamos os códigos da especificação de linguagem, onde salientamos as partes da especificação envolvidas no exemplo e a especificação de máquina.

C.1 Código de Implementação dos Módulos

Código de Implementação do Módulo C:

```
#include "math.h"
typedef float matel[100];
struct autoval {
float preal[10];
float pimag[10];
int ierr;
};
typedef struct autoval autoval;

void
calc_autoval(d,val,apval)
int d;
matel val;
autoval *apval;
{
        int timeout,falha,n,i;
        ps1 = "ps1";
        ps2 = "ps2";
        falha = 0;
        envass (ps1, &d, sizeof(ndim), &falha);
        if (falha == 2) return;   /* falha no envio */
        n = pow(d,2);
```

```
        timeout = -1;
        envsin(ps2,val,sizeof(matel),timeout,apval,sizeof(autoval),falha);
        switch(falha) {
                case 0: fprintf(stderr, "esgotamento de timeout");
                        break;
                case 2: fprintf(stderr, "falha de ligacao");
                        break;
        }
}
```

Código de Implementação do Módulo Fortran:

```
IMPLICIT  REAL*8 (A-H,O-Z)
DIMENSION A(10,10)
DIMENSION WR(10), WI(10), FV1(10)
INTEGER   IV1(10), IERR, DIM, TIMEOUT
CHARACTER*10 PORTO
REAL*8 MATEL(100)

TIMEOUT = -1
PORTO = 'PEN1'
DIM = 0
CALL RECEBE (PORTO, DIM, TIMEOUT)
L = 0
DO 10 I = 1, DIM
   DO J = 1, DIM
   L  = L + 1
   A(I,J) = MATEL[L]
   END DO
END DO
CALL RG (N, A, WR, WI,IV1, FV1, IERR)
CALL RESPOSTA (PORTO, WR, WI, IERR)
STOP
END
```

C.2 Rotinas de Interface Geradas

Cabeçalho:

```
#include <rpc/types.h>

typedef struct {
        u_int matel_len;
        float *matel_val;
} matel;
bool_t xdr_matel();

struct autoval {
        float preal[10];
        float pimag[10];
        int ierr;
};
typedef struct autoval autoval;
bool_t xdr_autoval();
```

Rotinas de Conversão XDR:

```
/*
 * Please do not edit this file.
 * It was generated using rigen.
```

```
 */

#include <rpc/rpc.h>
#include "diretor.h"

bool_t
xdr_matel(xdrs, objp)
        XDR *xdrs;
        matel *objp;
{
        if (!xdr_array(xdrs, (char **)&objp->matel_val, (u_int *)&objp->matel_len, 100, sizeof(float),
xdr_float)) {
                return (FALSE);
        }
        return (TRUE);
}

bool_t
xdr_autoval(xdrs, objp)
        XDR *xdrs;
        autoval *objp;
{
        if (!xdr_vector(xdrs, (char *)objp->preal, 10, sizeof(float), xdr_float)) {
                return (FALSE);
        }
        if (!xdr_vector(xdrs, (char *)objp->pimag, 10, sizeof(float), xdr_float)) {
                return (FALSE);
        }
        if (!xdr_int(xdrs, &objp->ierr)) {
                return (FALSE);
        }
        return (TRUE);
}
```

Rotina de Conversão Auxiliar:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "diretor.h"

int
    xdr_converte(porto, lengthport, endmsg, opxdr, mem)
        char    porto[20], *endmsg, *mem;
        int     *lengthport;
        u_int   opxdr;

{
        XDR     xdr_handle;
        u_int   size, sizemsg;
        int     v, p;

        v = strcmp(porto,"pen1");
        if (v == 0) p = 1;
        else {
                v = strcmp(porto, "pen2");
                if (v == 0) p = 2;
                else {
                        fprintf(stderr,"O porto especificado na funcao xdr_converte nao esta correto!!");
                        return;
                }
        }
        switch (p) {
                case 1:
                        size = RNDUP(sizeof(int));
                        if (mem == NULL) {
                                fprintf(stderr,"Inexistencia de dados para conversao na recepcao do
porto %s", porto);
                                return;
                       }
                        /* Criacao do stream_xdr */
                        xdrmem_create(&xdr_handle, mem, size, opxdr);
                        /* Conversao dos dados de entrada*/
                        xdr_int(&xdr_handle, (int *)endmsg);
                        sizemsg = xdr_getpos(&xdr_handle);
                        return(sizemsg);
                        break;
                case 2:
                        if (opxdr == 1) {
```

```
                        size = RNDUP(sizeof(matel));
                        if (mem == NULL) {
                              fprintf(stderr,"Inexistencia de dados para conversao na recepcao do
porto %s", porto);
                              return;
                        }
                        /* Criacao do stream_xdr */
                        xdrmem_create(&xdr_handle, mem, size, opxdr);
                        /* Conversao dos dados de entrada*/
                        xdr_matel(&xdr_handle, (matel *)endmsg);
                        sizemsg = xdr_getpos(&xdr_handle);
                        return(sizemsg);
                      }
                      else {
                        size = RNDUP(sizeof(autoval));
                        if (mem == NULL) {
                              fprintf(stderr,"Erro na alocacao de memoria para o stream_xdr na
resposta do porto %s", porto);
                              return;
                        }
                        /* Criacao do stream_xdr */
                        xdrmem_create(&xdr_handle, mem, size, opxdr);
                        /* Conversao dos dados de entrada*/
                        xdr_autoval(&xdr_handle, (autoval *)endmsg);
                        sizemsg = xdr_getpos(&xdr_handle);
                        return(sizemsg);
                      }
                      break;
        }
}
```

Rotina de Interface Módulo C:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "diretor.h"

void
     enviass (porto, msg, tamsg, apfail)
        char    porto[20], *msg;
        int     tamsg, *apfail;

{
        XDR     xdr_handle;
        char    *mem;
        u_int   size, sizemsg;


        size = RNDUP(tamsg);
        mem  = malloc (size);
        if (mem == NULL) {
                fprintf(stderr,"Erro na alocacao de memoria para o stream_xdr do porto %s", porto);
                return;
        }
/* Criacao do stream_xdr */
        xdrmem_create(&xdr_handle, mem, size, XDR_ENCODE);
/* Conversao dos dados de saida */
        xdr_int(&xdr_handle, (int *)msg);
        sizemsg = xdr_getpos(&xdr_handle);
/* Envio da mensagem */
        envio(1,sizemsg,0,mem,-1,0,0);
        *apfail = falha();
}


void
     envsin (porto, msg, tamsg, timeout, rsp, tamrsp, fail)
        char    porto[20], *msg, *rsp;
        int     timeout, *fail;
        u_int   tamsg, tamrsp;

{
        XDR     xdr_handle, xdrs;
        char    *mem, *mrsp;
        u_int   size, sizemsg;


        size = RNDUP(tamsg);
```

```
        mem  = malloc (size);
        if (mem == NULL) {
                fprintf(stderr,"Erro na alocacao de memoria para o stream_xdr do porto %s", porto);
                return;
        }
/* Criacao do stream_xdr */
        xdrmem_create(&xdr_handle, mem, size, XDR_ENCODE);
/* Conversao dos dados de saida */
        xdr_matel(&xdr_handle, (matel *)msg);
        sizemsg = xdr_getpos(&xdr_handle);
/* Envio da mensagem */
        envio(2,sizemsg,0,mem, timeout,0, mrsp);
        *fail = falha();
/* Se o envio nao obteve sucesso indica  o valor de "falha" */
        if(*fail != 1) {
                fprintf(stderr," Problema no envio da mensagem, falha = ", *fail);
        return;
        }
/* Conversao dos dados da resposta */
        else    xdr_autoval(xdrs, (autoval *) rsp);
}
```

Rotina de Interface do Módulo Fortran:

```
C-------------------------------------------------------
C     ROTINAS DE INTERFACE DO MODULO CALCULA
C-------------------------------------------------------
C                   SUBROTINA RECEBE
C
      SUBROUTINE RECEBE (PORTO, ENDMSG, TIMEOUT, SEL, CLAUSULA)
      CHARACTER*20    PORTO
      POINTER(ENDMSG, A)
      EXTERNAL CHARACTER            A, C
      INTEGER         TIMEOUT, SEL, CLAUSULA

      CHARACTER*20    PORTOMSG
      POINTER(MEM, C)
      INTEGER         PUSH, TEMPO, XDR_DECODE
      PARAMETER(PUSH=0, XDR_DECODE=1)
      EXTERNAL INICIA_RECEPCAO !$PRAGMA C (INICIA_RECEPCAO)
      EXTERNAL RECEPCAO !$PRAGMA C (RECEPCAO)
      EXTERNAL XDR_CONVERTE !$PRAGMA C (XDR_CONVERTE)

      IF (PORTO.EQ.'PEN1') THEN
            MEM = MALLOC(4)
            CALL INICIA_RECEPCAO(1,0,MEM,0,CLAUSULA)
            IF (SEL.EQ.0) THEN
                  CALL RECEPCAO(TIMEOUT,PORTOMSG)
                  IF (PORTOMSG.NE.PORTO) THEN
C  *ESGOTAMENTO DO TIMEOUT OU INEXISTENCIA DE MENSAGEM
                        CLAUSULA = 0
                        RETURN 1
                   ELSE
                        CALL XDR_CONVERTE(PORTO,ENDMSG,XDR_DECODE,
&       MEM)
                        RETURN
                   END IF
            ELSE
                  TEMPO = TEMPO_DE_ESPERA(TIMEOUT,PUSH)
                  CALL GUARDA_ENDS(ENDMSG,MEM)
                  RETURN
            END IF
       ELSE IF (PORTO.EQ.'PEN2') THEN
                   MEM = MALLOC(800)
                   CALL INICIA_RECEPCAO(2,0,MEM,0,CLAUSULA)
                   IF (SEL.EQ.0) THEN
                         CALL RECEPCAO(TIMEOUT, PORTOMSG)
                         IF (PORTOMSG.NE.PORTO) THEN
C  *ESGOTAMENTO DO TIMEOUT OU INEXISTENCIA DE MENSAGEM
                               CLAUSULA = 0
                               RETURN 1
                         ELSE
                               CALL XDR_CONVERTE(PORTO,ENDMSG,XDR_DEC
&        ODE,MEM)
                         END IF
                   ELSE
                         TEMPO = TEMPO_DE_ESPERA(TIMEOUT,PUSH)
                         CALL GUARDA_ENDS(ENDMSG,MEM)
```

```
                    RETURN
                END IF
      ELSE
C  *O PORTO ESPECIFICADO NA FUNCAO RECEBE NAO ESTA CORRETO
            RETURN 2
      END IF
      RETURN
      END
C
C
C-----                SUBROTINA RECEPCAO_SEL                -----
C
      SUBROUTINE RECEPCAO_SEL()
      CHARACTER*20    PORTOMSG
      INTEGER         TIMEOUT,POP, XDR_DECODE
      PARAMETER(POP = 1, XDR_DECODE=1)
      POINTER(ENDMSG, A)
      EXTERNAL CHARACTER            A, C
      POINTER(MEM, C)
      EXTERNAL RECEPCAO !$PRAGMA C (RECEPCAO)
      EXTERNAL XDR_CONVERTE !$PRAGMA C (XDR_CONVERTE)
      TIMEOUT = TEMPO_DE_ESPERA(0,POP)
      CALL RECEPCAO(TIMEOUT,PORTOMSG)
      IF(PORTO.EQ.'')THEN
C  *ESGOTAMENTO DO TIMEOUT OU INEXISTENCIA DE MENSAGEM
            CLAUSULA = 0
            RETURN 1
      ELSE
            CALL RETIRA_ENDS(ENDMSG,MEM)
            CALL XDR_CONVERTE(PORTO,ENDMSG,XDR_DECODE,MEM)
      END IF
      RETURN
      END
C
C
C-----                  SUROTINA RESPOSTA                   -----
C
      SUBROUTINE RESPOSTA(PORTO, ENDMSG)
      CHARACTER*20    PORTO
      POINTER(ENDMSG, A)
      EXTERNAL CHARACTER            A, C
      INTEGER         XDR_ENCODE, TAMSG
      PARAMETER(XDR_ENCODE=0)
      POINTER(MEM, C)
      EXTERNAL XDR_CONVERTE !$PRAGMA C (XDR_CONVERTE)
      EXTERNAL RESPONDE !$PRAGMA C (RESPONDE)
      MEM = MALLOC(204)
C     *CONVERTE OS DADOS DA RESPOSTA
      TAMSG = XDR_CONVERTE(PORTO, ENDMSG, XDR_ENCODE, MEM)
C     *ENVIA A RESPOSTA
      CALL RESPONDE(2, TAMSG, MEM, 0)
      RETURN
      END
```

C.3 Especificação de linguagem para C

Parte da Especificação de Definição de tipos (XDR -> C):

```
; Emitindo codigo equivalente em C para o array variavel

name_type:      array
type-decl:
WRITE           struct $type_name {
WRITE                   u_int  $type_name_len;
WRITE                   $base_type_name  $type_name_val;
WRITE           };
type-def:
WRITE  typedef struct $type_name $type_name;

; Emitindo a chamada para a rotina de conversao do tipo especificado

type_xdr:
```

```
WRITE  bool_t  xdr_$type_name();


;Emitindo codigo equivalente em C para registro

name_type:      record
type_decl:
WRITE  struct $type_name {
REPEAT (n_fields)
WRITE  $field_type_name $field_name( [$value]);
NEXT
WRITE  };
type_def:
WRITE  typedef struct $type_name $type_name;

; Emitindo a chamada para a rotina de conversao do tipo especificado

type_xdr:
WRITE  bool_t  xdr_$type_name();
```

Parte da Especificação das Rotinas de Conversão XDR:

```
; Emitindo codigo para conversao de array variavel

conversion:     array
WRITE  bool_t
MARSHAL                 xdr_$type_name(xdrs, objp)
WRITE                   XDR             *xdrs;
WRITE                   $type_name      *objp;
WRITE  {
WRITE           if(!xdr_array(xdrs,(char **)&objp->$type_name_val,(u_int *)&objp->$type_name_len,
$value, sizeof($base_type_name), xdr_$base_tape_name)){
WRITE                   return(FALSE);
WRITE           }
WRITE           return(TRUE);
WRITE  }


; Emitindo codigo para conversao da estrutura definida

conversion:     record
WRITE  bool_t
MARSHAL                 xdr_$type_name(xdrs, objp)
WRITE                   XDR             *xdrs;
WRITE                   $type_name      *objp;
WRITE  {
REPEAT (n_fields)
WRITE           if(!
M_MEMBER                xdr_$type_fiiden) {
WRITE                   return(FALSE);
WRITE           }
NEXT
WRITE           return(TRUE);
WRITE  }
```

Especificação das Rotinas de Interface:

```
WRITE  /*Rotinas de interface para $module */
; Emitindo o cabecalho das rotinas de interface
WRITE  #include <stdio.h>
WRITE  #include <rpc/rpc.h>
WRITE  #include <$module.h>
WRITE  #define   PUSH = 0
WRITE  #define   POP = 1


;**** Emitindo o codigo da funcao ENVIASS, se existir ****
WRITE  void
WRITE           enviass(porto, msg, tamsg, apfail)
WRITE                   char    porto[20], *msg;
WRITE                   int     tamsg, *apfail;
```

```
WRITE  {
WRITE           XDR     xdr_handle;
WRITE           char    *mem;
WRITE           u_int   size, sizemsg;
; Verifica se existe mais de um porto de saida asssincrono
; Se existe passa para o switch
; Senao perfaz o programa de envio assincrono

;Codigo para mais de um porto
WRITE           int     v,p;
REPEAT (n_port_saida_ass)
WRITE           v = strcmp(porto, "$port_name");
WRITE           if(v == 0) p = $tabport[n]->num;
WRITE           else {
NEXT
WRITE           fprintf(stderr,"0 porto especificado na funcao enviass nao esta correto!!");
WRITE             return;
REPEAT (n_port_saida_ass)
WRITE           }
NEXT
WRITE           switch (p) {
REPEAT (n_port_saida_ass)
WRITE                   case $tabport[n]->num:
M_MEMBER                                        programa_ass
WRITE                                           break;
NEXT
WRITE           }

; Codigo do Programa assincrono
WRITE           size = RNDUP(sizeof($port_msg_type));
WRITE           mem = malloc(size);
WRITE           if (mem == NULL) {
WRITE                   fprintf(stderr,"Erro na alocacao de memoria para o stream_xdr de envio no
porto %s", porto);
WRITE                   return;
WRITE           }
WRITE           /* Criacao do stream_xdr */
WRITE           xdrmem_create(&xdr_handle, mem, size, XDR_ENCODE);
WRITE           /* Conversao dos dados de saida*/
WRITE           xdr_$port_msg_type(&xdr_handle, ($port_msg_type *)msg);
WRITE           sizemsg =xdr_getpos(&xdr_handle);
WRITE           /* Envio da mensagem */
WRITE           envio($tabport[n]->num, sizemsg, 0, mem, 0, 0, 0);
WRITE           *apfail = falha();


; Emitindo o final da funcao enviass
WRITE  }



; **** Emitindo o codigo da funcao ENVSIN, se existir ****
WRITE  void
WRITE           envsin(porto, msg, tamsg, timeout, rsp, tamrsp, apfail)
WRITE                   char    porto[20], *msg, *rsp;
WRITE                   int     timeout, *apfail;
WRITE                   u_int   tamsg, tamrsp;
WRITE  {
WRITE           XDR     xdr_handle, xdrs;
WRITE           char    *mem, *mrsp;
WRITE           u_int   size, sizemsg, sizersp;
; Verifica se existe mais de um porto de saida sincrono
; Se existe passa para o switch
; Senao perfaz o programa de envio sincrono

;Codigo para mais de um porto
WRITE           int     v,p;
REPEAT (n_port_saida_sin)
WRITE           v = strcmp(porto, "$port_name");
WRITE           if(v == 0) p = $tabport[n]->num;
WRITE           else {
NEXT
WRITE           fprintf(stderr,"0 porto especificado na funcao envsin nao esta correto!!");
WRITE             return;
REPEAT (n_porto_saida_sin)
WRITE           }
NEXT
WRITE           switch (p) {
REPEAT (n_port_saida_sin)
WRITE                   case $tabport[n]->num:
M_MEMBER                                        programa_sin
WRITE                                           break;
NEXT
```

```
WRITE           }

; Codigo do Programa Sincrono
WRITE           size = RNDUP(sizeof($port_msg_type));
WRITE           mem = malloc(size);
WRITE           if (mem == NULL) {
WRITE                   fprintf(stderr,"Erro na alocacao de memoria para o stream_xdr de envio no
porto %s", porto);
WRITE                   return;
WRITE           }
WRITE           /* Criacao do stream_xdr */
WRITE           xdrmem_create(&xdr_handle, mem, size, XDR_ENCODE);
WRITE           /* Conversao dos dados de saida*/
WRITE           xdr_$port_msg_type(&xdr_handle, ($port_msg_type *)msg);
WRITE           sizemsg = xdr_getpos(&xdr_handle);
WRITE           /* Criacao do stream_xdr para a resposta */
WRITE           sizersp = RNDUP(sizeof($port_reply_type));
WRITE           mrsp= malloc(sizersp);
WRITE           if (mrsp == NULL) {
WRITE                   fprintf(stderr,"Erro na alocacao de memoria para o stream_xdr de resposta no
porto %s", porto);
WRITE                   return;
WRITE           }
WRITE           xdrmem_create(&xdrs, mrsp, sizersp, XDR_DECODE);
WRITE           /* Envio da mensagem */
WRITE           envio($tabport[n]->num, sizemsg, 0, mem, timeout, 0, mrsp);
WRITE           *apfail = falha();
WRITE           if((*apfail == 0) || (*apfail == 1)) {
WRITE                   fprintf(stderr,"Envio do porto %s sem sucesso", porto);
WRITE                   return;
WRITE           }
WRITE           /* Conversao da resposta*/
WRITE           xdr_$port_reply_type(&xdrs, ($port_reply_type *)rsp);


; Emitindo o final da funcao envsin
WRITE   }


; **** Emitindo o codigo da funcao RECEBE, se existir ****
WRITE   void
WRITE           recebe(porto, endmsg, timeout, sel, clausula)
WRITE                   char    porto[20], *endmsg;
WRITE                   int     timeout;
WRITE                   u_int   sel, clausula;
WRITE   {
WRITE           char    *mem;
WRITE           int     tempo;
WRITE           u_int   size;
WRITE           char    portomsg[20];
; Verifica se existe mais de um porto de entrada
; Se existe passa para o switch
; Senao perfaz o programa de recepcao

;Codigo para mais de um porto
WRITE           int     v,p;
REPEAT (n_port_ent)
WRITE           v = strcmp(porto, "$port_name");
WRITE           if(v == 0) p = $tabport[n]->num;
WRITE           else {
NEXT
WRITE           fprintf(stderr,"O porto especificado na funcao recebe nao esta correto!!");
WRITE             return;
REPEAT (n_port_ent)
WRITE           }
NEXT
WRITE           switch (p) {
REPEAT (n_port_ent)
WRITE                   case $tabport[n]->num:
M_MEMBER                                        programa_recepcao
WRITE                                           break;
NEXT
WRITE           }


; Codigo do programa de recepcao
WRITE           size = RNDUP(sizeof($port_msg_type));
WRITE           mem = malloc(size);
WRITE           if (mem == NULL) {
WRITE                   fprintf(stderr,"Erro na alocacao de memoria para o stream_xdr de recepcao no
porto %s", porto);
WRITE                   return;
WRITE           }
WRITE           /* Habilita o porto para a recepcao */
```

```
WRITE           inicia_recepcao($tabport[n]->num, 0, mem, 0, clausula);
WRITE           if (sel == 0) {
WRITE           /* Recepcao simples */
WRITE                   recepcao(timeout,portomsg);
WRITE                   if(portomsg != porto) {
WRITE                           fprintf(stderr,"Esgotamento de tempo ou inexistencia de mensagem no
porto %s", porto);
WRITE                           return;
WRITE                   }
WRITE                   else {
WRITE                           /* Conversao dos dados de entrada */
WRITE                           xdr_converte(porto, endmsg, XDR_DECODE, mem);
WRITE                   }
WRITE           }
WRITE           else {
WRITE                   /* Recepcao seletiva */
WRITE                   tempo = tempo_de_espera(timeout, PUSH);
WRITE                   guarda_ends(porto, endmsg, mem);
WRITE           }

; Emitindo o final da funcao recebe
WRITE   }


; **** Emitindo o codigo da funcao RECEPCAO_SEL, se existir ****
WRITE   void
WRITE           recepcao_sel();
WRITE   {
WRITE           char    portomsg[20] = "", *endmsg, *mem;
WRITE           int     timeout;
WRITE           \n
WRITE           /* Escolha do menor timeout */
WRITE           timeout = tempo_de_espera(0, POP);
WRITE           /* Recepcao  */
WRITE           recepcao(timeout,portomsg);
WRITE           if((strcmp(portomsg,"") == 0) {
WRITE                   fprintf(stderr,"Esgotamento de tempo ou inexistencia de mensagem na recepcao
seletiva");
WRITE                   return;
WRITE           }
WRITE           else {
WRITE           /* Recupera os enderecos de "portmsg" */
WRITE           retira_ends(portomsg, endmsg, mem);
WRITE           /* Conversao dos dados de entrada */
WRITE           xdr_converte(portomsg, endmsg, XDR_DECODE, mem);
WRITE           }
WRITE   }


; **** Emitindo o codigo da funcao RESPOSTA, se existir ****
WRITE   void
WRITE           resposta(porto, endmsg);
WRITE           char    porto[20], *endmsg;
WRITE   {
WRITE           u_int   tamsg, size;
WRITE           char    *mem;
WRITE           \n
WRITE           size = RNDUP(sizeof($port_reply_type));
WRITE           mem = malloc(size);
WRITE           if (mem == NULL) {
WRITE                   fprintf(stderr,"Erro na alocacao de memoria para o stream_xdr de resposta no
porto %s", porto);
WRITE                   return;
WRITE           }
WRITE           /* Conversao dos dados da resposta */
WRITE           tamsg = xdr_converte(porto,endmsg,XDR_ENCODE, mem);
WRITE           /* Envio da resposta */
WRITE           responde($tabport[n]->num,tamsg,0,mem);
WRITE   }


; **** Emitindo o codigo da funcao XDR_CONVERTE, se existir ****
WRITE   int
WRITE           xdr_converte(porto, lengthport, endmsg, opxdr, mem)
WRITE           char    porto[20], *endmsg, *mem;
WRITE           int     *lengthport;
WRITE           u_int   opxdr;
WRITE   {
WRITE           XDR     xdr_handle;
WRITE           u_int   size, sizemsg;
; Verifica se existe mais de um porto de entrada
; Se existe passa para o switch
```

```
; Senao perfaz o programa de conversao

;Codigo para mais de um porto
WRITE           int     v,p;
REPEAT (n_port_ent)
WRITE           v = strcmp(porto, "$port_name");
WRITE           if(v == 0) p = $tabport[n]->num;
WRITE           else {
NEXT
WRITE           fprintf(stderr,"O porto especificado na funcao xdr_converte nao esta correto!!");
WRITE             return;
REPEAT (n_port_ent)
WRITE           }
NEXT
WRITE           switch (p) {
REPEAT (n_port_ent)
WRITE                   case $tabport[n]->num:
M_MEMBER                                        programa_conversao
WRITE                                           break;
NEXT
WRITE           }

; Emitindo o codigo do programa de conversao
WRITE           if (opxdr == 1) {
WRITE                   size = RNDUP(sizeof($port_msg_type));
WRITE                   if (mem == NULL) {
WRITE                           fprintf(stderr,"Erro na alocacao de memoria para a recepcao do porto
%s", porto);
WRITE                           return;
WRITE                   }
WRITE                   /* Criacao do stream_xdr */
WRITE                   xdrmem_create(&xdr_handle, mem, size, opxdr);
WRITE                   /* Conversao dos dados de entrada*/
WRITE                   xdr_$port_msg_type(&xdr_handle, ($port_msg_type *)endmsg);
WRITE                   sizemsg = xdr_getpos(&xdr_handle);
WRITE                   return(sizemsg);
WRITE           }
; Emite o restante do codigo somente se o porto de entrada for sincrono
WRITE           else {
WRITE           /* opxdr deve ser 0, indicando que se trata de resposta */
WRITE                   size = RNDUP(sizeof($port_reply_type));
WRITE                   if (mem == NULL) {
WRITE                           fprintf(stderr,"Erro na alocacao de memoria para o stream_xdr na
resposta do porto %s", porto);
WRITE                           return;
WRITE                   }
WRITE                   /* Criacao do stream_xdr */
WRITE                   xdrmem_create(&xdr_handle, mem, size, opxdr);
WRITE                   /* Conversao dos dados de entrada*/
WRITE                   xdr_$port_reply_type(&xdr_handle, ($port_reply_type *)endmsg);
WRITE                   sizemsg = xdr_getpos(&xdr_handle);
WRITE                   return(sizemsg);
WRITE           }

; Emitindo o final da funcao xdr_converte
WRITE   }


; **** Emitindo o codigo da funcao TEMPO_DE_ESPERA ****
WRITE   int
WRITE           tempo_de_espera(timeout, operacao)
WRITE           int     timeout, operacao;
WRITE   {...
WRITE   }

; **** Emitindo o codigo da funcao GUARDA_ENDS ****
WRITE   void
WRITE           guarda_ends(porto, endmsg, mem)
WRITE           char    porto[20], *endmsg, *mem;
WRITE   {...
WRITE   }

; **** Emitindo o codigo da funcao RETIRA_ENDS ****
WRITE   void
WRITE           retira_ends(porto, endmsg, mem)
WRITE           char    porto[20], *endmsg, *mem;
WRITE   {...
WRITE   }
```

C.4 Especificação de Máquina

```
machine:                SUN 4
machine-byte_order:     big-endian

data_name:              integer
size:                   32
representation:         2 complement

data_name:              double
size:                   64
representation:         IEEE standard
```