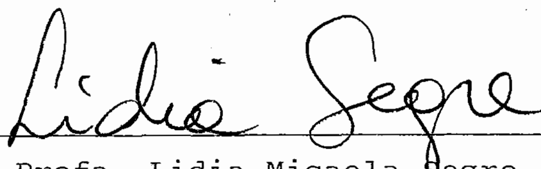
UM INTERPRETADOR DE UMA LINGUAGEM DE CONFIGURAÇÃO PARA UM AMBIENTE DE PROGRAMAÇÃO DISTRIBUÍDA BASEADO EM MODULA-2

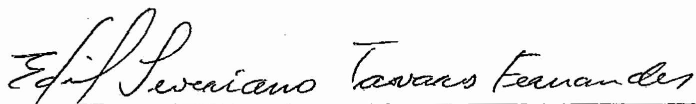
VANISE PARAÍSO VETROMILLE

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.Sc.) EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:



Profa. Lidia Micaela Segre
Presidente



Prof. Edil Severiano Tavares Fernandes

Prof. Daniel Alberto Menascé

RIO DE JANEIRO, RJ - BRASIL
ABRIL DE 1988

VETROMILLE, VANISE PARAÍSO

UM INTERPRETADOR DE UMA LINGUAGEM DE CONFIGURAÇÃO PARA UM AMBIENTE DE PROGRAMAÇÃO DISTRIBUÍDA BASEADO EM MODULA-2 (DIO DE JANEIRO), 1988.

xi, 206 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1988).

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Sistemas Distribuídos I. COPPE/UFRJ

II. Título (série)

Aos meus pais

AGRADECIMENTOS

Agradeço à Professora Lidia Micaela Segre pela sua valiosa e competente orientação e pelo seu apoio durante a realização desta tese.

Meus sinceros agradecimentos à Nanci Lages e ao Nelson da Silva pela participação e colaboração no estudo de alguns assuntos necessários para a execução deste trabalho.

Agradeço à minha família pelo incentivo e compreensão que me deu durante todo o período em que necessitei dedicar-me integralmente à tese, aproveitando para desculpar-me da ausência que precisei ter no papel de filha, irmã, neta, madrinha e tia, não podendo, muitas vezes, corresponder igualmente à dedicação e ao amor recebido.

Ao Ricardo Barz Sovat o meu agradecimento todo especial, tão especial quanto foi a demonstração de sua amizade, pois sem a sua ajuda, o seu incentivo, a sua compreensão e sobretudo, sem o seu carinho, talvez não tivesse conseguido chegar até o final.

A todos que, através de atitudes e palavras, incentivaram-me, principalmente ao Beto, à Cristina, à Angela, ao Reinaldo, ao Luiz Felipe, ao Ricardo Rocha, à Maria Cristina, à Vera e à Maria Fernanda, agradeço e deixo aqui registrado o meu eterno reconhecimento.

Agradeço à Yolanda da Fonseca pelo ótimo trabalho de datilografia e ao José Carlos de Carvalho Ferreira pela excelente elaboração dos desenhos.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de mestre de Ciências (M.Sc.).

UM INTERPRETADOR DE UMA LINGUAGEM DE CONFIGURAÇÃO PARA UM AMBIENTE DE PROGRAMAÇÃO DISTRIBUÍDA BASEADO EM MODULA-2.

VANISE PARAÍSO VETROMILLE

Abril, 1988

Orientador : Lidia Micaela Segre

Programa : Engenharia de Sistemas e Computação

O objetivo desta tese é implementar um interpretador para uma linguagem de configuração estática. A linguagem foi definida e acrescentada à linguagem Modula-2 para a construção de um ambiente de programação distribuída.

Para criar este ambiente baseado na linguagem Modula-2, foi incorporado nesta linguagem um mecanismo de comunicação e sincronização entre processos. O método escolhido foi a chamada remota de procedimento de forma transparente ao usuário.

O interpretador além de analisar o programa escrito na linguagem de configuração, provê também a interface necessária para execução de chamadas remotas a procedimentos. Sua implementação foi levada a cabo visando uma futura extensão para configuração dinâmica.

Com o objetivo de avaliar o potencial da linguagem que foi especificada, comparamos nossa proposta com outros projetos semelhantes. Concluímos o nosso trabalho com sugestões para tornar o interpretador mais flexível.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master Science (M.Sc.).

AN INTERPRETER OF A CONFIGURATION LANGUAGE FOR A DISTRIBUTED PROGRAMMING ENVIRONMENT, BASED IN THE MODULA-2 LANGUAGE

VANISE PARAÍSO VETROMILLE

April, 1988

Adiviser : Lidia Micaela Segre

Department: System and Computation Engineering

This thesis deals with the implementation of the interpreter for a static configuration language. The configuration language was defined and added to the Modula-2 language in order to construct a distributed programming environment.

In order to create this environment based on the Modula-2 language, it was necessary to incorporate a mechanism for interprocess communication and synchronization to this language. The remote procedure call (RPC) mechanism was chosen in a way which is transparent to the user.

The interpreter provides the analysis of the configuration program, and the necessary interface to execute the remote procedure call. The interpreter implementation was carried out bearing in mind future extensions for dynamic configuration.

A comparison between our work and a group of similar projects was carried out in order to evaluate the potential of our language. Some suggestions to make the interpreter more flexible are presented in the concluions.

ÍNDICE

CAPÍTULO I

INTRODUÇÃO 01

CAPÍTULO II

DESCRIÇÃO DETALHADO DO PROJETO 10

II.1 - Requisitos de um Ambiente de Programação Distribuída 10
II.2 - Extensões Introduzidas à Linguagem Modula-2 ... 13
II.3 - Núcleo de Multiprogramação 20
II.4 - Chamada REmota de Procedimento (RPC) 25
II.5 - Gerador de "Stubs" 27
II.5.1 - "Stub" do Cliente 32
II.5.2 - "Stub" do Servidor 32
II.6 - Exemplo Ilustrativo da Implementação de RPC ... 33

CAPÍTULO III

DEFINIÇÃO DE UMA LINGUAGEM DE CONFIGURAÇÃO 39

III.1 - Características das Linguagens de Configuração 39
III.2 - Requisitos Necsssários numa Linguagem de Configuração 41
III.3 - A Linguagem de Configuração Proposta 44

CAPÍTULO IV

COMPARAÇÃO ENTRE ALGUNAS LINGUAGENS DE CONFIGURAÇÃO .. 54

IV.1 - Conic e Conic/C 54
IV.2 - Mesa e c/Mesa 61
IV.3 - Método MASCOT3 73

CAPÍTULO V

A IMPLEMENTAÇÃO DE UM INTERPRETADOR PARA A NOSSA LINGUAGEM DE CONFIGURAÇÃO 92

V.1 - Ambiente de Programação Distribuída 92
V.2 - Estrutura Geral do Interpretador 93
V.3 - Estruturas de Dados Utilizados pelo Interpretador 95
V.3.1 - Tabela de Configurações 96
V.3.2 - Tabela de Tipos 97
V.3.3 - Tabela de Tipos de Configuração 98
V.3.4 - Tabela de Estações Lógicas 99
V.3.5 - Tabelas Pertencentes ao Módulo de Configuração 101

V.3.5.1 - Tabela de Importações 101
V.3.5.2 - Tabela de Exportações 102

V.3.6 - Observações 102

V.4 - Descrição do Interpretador 103

V.4.1 - Primeira Parte do Interpretador....... 103
V.4.2 - Segunda Parte do Interpretador (Módulo de Configuração) 112

V.5 - Interface do Módulo de Configuração com os "Stubs" e com o Núcleo de Programação 113

CAPÍTULO VI

EXPERIÊNCIA DO USO DO INTERPRETADOR 116

VI.1 - Exemplos Ilustrativos 116

VI.1.1 - Exemplo de Enfermaria 116
VI.1.2 - Exemplo de Propagação de Importações e Exportações 122
VI.1.3 - Exemplo de Uso de Instâncias de Tipos de Módulos e Sub-Configurações 128
VI.1.4 - Exemplo de Uso de Instâncias de Tipos Diferentes de Módulos e Sub-Configurações 133

VI.2 - Resultados Obtidos Aplicando-se o Interpretador no Exemplo da Seção VI.1.2 140

CAPÍTULO VII

CONCLUSÃO 143

REFERÊNCIAS 154

ANEXO I

LISTAGEM DA IMPLEMENTAÇÃO DO INTERPRETADOR 160

ÍNDICE DAS FIGURAS

CAPÍTULO II

II.1 - Módulo de Definição do NucleoHolt 22
II.2 - Mecanismo de uma Chamada Remota de Procedimento .. 26
II.3 - Interação entre os Diversos Componentes Envolvidos na Execução de uma Chamada Remota de Procedimento.. 31
II.4 - Módulo de Definição ServArq 33
II.5 - Módulo de Implementação de ServArq 34
II.6 - Módulo do Cliente 34
II.7 - "Stub" do Cliente 35
II.8 - "Stub" do Servidor 36
II.9 - Etapas na Execução de uma Chamada Remota de Procedimento 38

CAPÍTULO III

III.1 - Definição de Declaração de Configuração 53

CAPÍTULO IV

IV.1 - Exemplo da Utilização da Declaração IMPORTS num Programa Escrito em Mesa 63
IV.2 - Exemplo da Definição de Vários Registros de Interfaces para o Mesmo Tipo de Interface 64
IV.3 - Primeiro Exemplo de uma Definição de Configuração Escrito em C/Mesa 65
IV.4 - Exemplo de uma Configuração Auto-Contida, Escrita em C/Mesa 66
IV.5 - Exemplo de uma Definição de Configuração que utiliza a Declaração IMPORTS 67
IV.6 - Exemplo de uma Definição de Configuração que não utiliza Nenhuma Facilidade Implícita 69
IV.7 - Último Exemplo de Uma Definição de Configuração Utilizando C/Mesa 70
IV.8 - Exemplo do Sistema Sys 75
IV.9 - Representação do Subsistema s3 77
IV.10 - Representação do Subsistema s4 78
IV.11 - Definição das Interfaces send e fetch 80
IV.12 - Definição do Tipo Flow-Data 81
IV.13 - Definição do Canal chan-1 81

IV.14 - Definição das Atividades a-temp-1 e a-temp-2 83
IV.15 - Definição do Subsistema s4 83
IV.16 - Definição de um Servidor 85
IV.17 - Definição do Subsistema s3 85
IV.18 - Definição do Sistema Sys 86
IV.19 - Representação Gráfica da Atividade a-temp-1 87
IV.20 - Definição da Atividade a-temp-1 88
IV.21 - Declaração do Elemento Raiz main 89
IV.22 - Declaração da Sub-Raiz sub1 89

CAPÍTULO V

V.1 - Esquema de Todas as Etapas que Antecedem à Execução do Sistema 94
V.2 - Declaração dos Tipos Utilizados para Definir os Campos da Tabela de Tipos 98
V.3 - Declaração dos Tipos Utilizados para Definir os Campos da Tabela de Tipos de Configuração 99
V.4 - Entrada da Tabela de Estações Lógicas 100
V.5 - Tabela de Importações 101
V.6 - Tabela de Exportações 102
V.7 - Algoritmo Geral da Segunda Fase Pertencente à Primeira Parte do Interpretador 105
V.8 - Definição de Alguns Tipos Utilizados Pelos Campos da Tabela de Configurações 109

CAPÍTULO VI

VI.1 - Sistema de Controle de Pacientes 117
VI.2 - Declaração da Sub-Configuração PatientConf 120
VI.3 - Declaração da Sub-Configuração NurseConf 121
VI.4 - Declaração da Configuração WardConf 122
VI.5 - Esquema da Configuração do Exemplo de FloorConf com Controle Estatístico 124
VI.6 - Declaração da Sub-Configuração NurseConf 125
VI.7 - Declaração da Sub-Configuração ControlConf....... 126
VI.8 - Declaração da Sub-Configuração DoctorConf 126
VI.9 - Declaração da Sub-Configuração WardConf 127
VI.10 - Declaração da Configuração FloorConf 128

VI.11 - Esquema da Configuração do Exemplo de FloorConf1.. 129
VI.12 - Declaração da Sub-Configuração NurseConf 130
VI.13 - Declaração da Sub-Configuração WardConfT 131
VI.14 - Declaração da Configuração FloorConf1 132
VI.15 - Esquema da Configuração do Exemplo de FloorConf2.. 134
VI.16 - Declaração da Sub-Configuração PatientConf 136
VI.17 - Declaração da Sub-Configuração NurseConf1 137
VI.18 - Declaração da Sub-Configuração NurseConf2 137
VI.19 - Declaração da Sub-Configuração WardConf1 138
VI.20 - Declaração da Sub-Configuração WardConf2 138
VI.21 - Declaração da Sub-Configuração FloorConf2 139
VI.22 - Entrada Ø da Tabela de Configurações Referente à Sub-Configuração PatientConf 143
VI.23 - Entrada 1 da Tabela de Configurações Referente à Sub-Configuração NurseConf 144
VI.24 - Entrada 2 da Tabela de Configurações Referente à Sub-Configuração ControlConf 145
VI.25 - Entrada 3 da Tabela de Configurações Referente à Sub-Configuração DoctorConf 145
VI.26 - Entrada 4 da Tabela de Configurações Referente à Sub-Configuração WardConf 146
VI.27 - Tabela de Tipos 147
VI.28 - Entrada 5 da Tabela de Configurações Referente à Configuração FloorConf 148
VI.29 - Tabela de Importações da Estação Física nº 1 149
VI.30 - Tabela de Exportações da Estação Física nº 6 149
VI.31 - Tabela de Exportações da Estação Física nº 7 149

CAPÍTULO I

INTRODUÇÃO

Já se tornou fato indiscutível a importância da informática no desenvolvimento sócio-econômico de um país. Podemos apontar como causa principal deste acontecimento a rápida evolução nas tecnologias computacionais, que diminuiu, em muito, o gasto envolvido em componentes eletrônicos. Os custos mais acessíveis implicam num aumento significativo na produção de hardware, o que viabiliza o uso do computador numa gama enorme de aplicações.

Este avanço na área de hardware, entretanto, é acompanhado por um progresso também no que diz respeito ao software. Com o surgimento de novas aplicações, o desenvolvimento de software deve ser aprimorado para que este possa oferecer ferramentas que satisfaçam plenamente os requisitos necessários ao usuário.

No presente momento, o uso de computadores pessoais é, sem dúvida alguma, o ambiente de computação predominante. Hoje, os microcomputadores estão chegando à maturidade em termos de equipamentos, software e suporte, sendo utilizados cada vez mais ativamente, de forma independente ou integrados em sistemas distribuídos, onde podem estar presentes, até mesmo, máquinas mais potentes. Os computadores pessoais, que possuem uma grande importância na difusão da informática em todos os setores das atividades humanas, deixaram de ser para muitas pessoas e empresas uma simples curiosidade, e se transformaram num poderoso instrumento de trabalho, suplantando, em muitas situações, os tradicionais computadores de grande porte.

Este fato é decorrente das facilidades que este ambiente oferece, como por exemplo, a portabilidade do software desenvolvido para diversos computadores, e o não

compartilhamento de tempo de processamento entre os usuários.

Quando os computadores são conectados a outros computadores através de uma rede, além de se aumentar os recursos já existentes nestas máquinas, torna-se disponível o acesso às informações remotas, obtendo-se assim um potencial maior. O possível compartilhamento de periféricos poderosos e caros é mais um ponto favorável que podemos apontar quando utiliza-se uma rede.

No entanto, é necessário contar com um software de apoio que vai desde o sistema operacional, que facilita a operação e administração dos programas e arquivos, até pacotes de fácil utilização, que permitem seu emprego por pessoas mesmo sem especialização em informática.

Outro aspecto ao qual deve ser dada atenção é o ambiente de execução a ser utilizado. Um ambiente adequado que proporcione o desenvolvimento de sistemas de forma hierarquizada e modular, contribui muito, por exemplo, para a elaboração de sistemas distribuídos. Faz-se necessário portanto, criar para este caso uma metodologia de software que permita utilizar efetivamente o paralelismo físico existente em sistemas distribuídos, e que permita também minimizar os custos tanto no desenvolvimento quanto na manutenção do software, que estão, hoje em dia, entre os maiores gastos na computação.

No sentido de explorar ao máximo o potencial das arquiteturas distribuídas, pensou-se em desenvolver um ambiente de programação distribuída, baseado numa linguagem de alto nível moderna que fornecesse suporte ao programador para escrever aplicações eficientes, em particular, em sistemas de tempo real.

Analisaremos então, a evolução da linguagem Pascal, que conduziu à criação da linguagem Modula-2,

apresentando-se em seguida, os motivos que levaram à escolha desta última linguagem e seu ambiente de suporte de programação para construção de ambientes distribuídos.

Em 1971 N. Wirth criou a linguagem Pascal para o ensino de programação estruturada; no entanto, a sua difusão no meio acadêmico foi tão intensa que seus estudantes a conduziram para um ambiente profissional.

Pascal inovou na área de definição de tipos de dados e controle rígido do seu uso, o que permite identificar, em tempo de compilação, uma classe grande de erros semânticos. Apesar de pequena e elegante não faltaram críticas à linguagem Pascal, entre elas algumas apresentadas por KERNIGHAN |10|, tais como a inexistência de compilação separada, a inflexibilidade do modelo de entrada/saída, a inexistência de variáveis estáticas, as quais permitem a abstração de dados, e por fim, a impossibilidade de fugir do controle rígido dos tipos de dados, fazendo com que certas classes de programas precisem ser escritas numa linguagem diferente de Pascal.

Contudo, várias extensões de Pascal foram implementadas na tentativa de suprir as deficiências da linguagem; porém, a incompatibilidade existente entre elas não permitia a transportabilidade do software escrito nestas extensões, além de, no caso de haver compilação separada, quebrar o rígido controle de tipos de dados, uma das principais características de Pascal, a qual fornece uma grande segurança ao programador desta linguagem.

Foi observada a necessidade de incorporar aos projetos de linguagens certos conceitos tais como abstração de dados, modularidade e concorrência. Algumas soluções propostas foram Pascal Concorrente, Modula-1, Mesa, ADA e Modula-2. Entre estas linguagens, ADA e Modula-2 foram as que tiveram uma aceitação mais ampla. A seguir mostraremos sucintamente as características destas duas linguagens.

Com o propósito de implementar sistemas grandes, ADA |2| foi concebida no contexto de uma concorrência internacional, em resposta a uma especificação de requisitos ditada pelo governo norte americano. Uma das principais causas que motivaram a elaboração desta especificação, foi o desejo de substituir a coleção de linguagens diferentes, as quais eram usadas na programação de sistemas "embutidos", por uma única linguagem. Os requisitos determinavam que a linguagem vencedora possuísse facilidades de forma a atender a aplicações diversas. Os mecanismos de programação concorrente da linguagem ADA foram influenciados enormemente pelas propostas de Hoare e de Brinch Hansen. ADA foi a primeira linguagem totalmente definida para ser implementada seja num único processador, ou numa arquitetura distribuída com ou sem memória compartilhada |8|.

Modula-2, |43| de autoria de N. Wirth, o criador de Pascal, possui como conceito principal o módulo, uma ferramenta de estruturação de programas, considerado como unidade de compilação. O módulo delimita a visibilidade de um conjunto de declarações tais como constantes, tipos, variáveis e procedimentos. O grande uso do módulo está na abstração de dados, pois em Modula-2 é possível separar a definição de uma interface, que especifica como o módulo é usado, da implementação real do seu conceito. A essência da abstração reside exatamente nesta separação, e se presta imediatamente a uma metodologia de projeto de cima para baixo, onde as interfaces são projetadas antes de sua implementação.

A linguagem Pascal está orientada para projetos desenvolvidos por refinamentos sucessivos, nos quais as operações são especificadas em termos de suboperações, sendo todas elas executadas normalmente sobre dados globais. Ao contrário, Modula-2 apoia o desenvolvimento modular, que consiste da identificação das estruturas de dados usadas na solução de um problema, e do seu encapsulamento, num único módulo, juntamente com as operações definidas sobre estas

estruturas.

Wirth concentrou em Modula-2 todos os conceitos que achava interessante nas linguagens antecessoras: de Pascal manteve o conceito e os tipos de dados, de Modula-1 a sintaxe e os módulos e, por fim, de Mesa, extraiu a compilação separada. Além disto, Modula-2 permite que se definam co-rotinas, que são procedimentos que transferem explicitamente o controle entre si de maneira não hierarquizada. Desta forma, pode-se implementar vários modelos de programação concorrente utilizando-se diretamente Modula-2, dispensando-se o uso de uma linguagem de montagem. Na tentativa de abrir mão totalmente do auxílio de tais linguagens, Modula-2 introduziu mecanismos para manipulação de objetos de nível mais baixo, ou seja, mais próximos da máquina, permitindo assim, programar sistemas de entrada/saída, como também programar outros tipos de software que requeiram interação física com o hardware. Com estas características torna-se possível escrever todo o software de um sistema de computação na linguagem Modula-2.

Pelo fato da linguagem ADA ter sido projetada com o intuito de poder ser utilizada em, praticamente, todas as áreas de aplicação, em sistemas grandes, tal linguagem tornou-se relativamente complexa, apesar de poderosa, o que dificulta, primeiramente o seu uso, e, em segundo lugar, sua implementação em computadores de pequeno porte, fatores estes que desfavorecem a sua utilização.

Podemos apresentar como sendo outra desvantagem da linguagem ADA a sua elaboração por uma comissão, ao contrário de Modula-2, que pelo fato de ter sido projetada por uma única pessoa, obteve uma coerência marcante, além de mostrar uma sensibilidade para os problemas de implementação.

Mais um ponto favorável à Modula-2 é a sua disponibilidade para vários microcomputadores, inclusive o

PC da IBM.

Por todas as características e vantagens descritas anteriormente, além de ser simples e muito bem estruturada, fato que a torna elegante, Modula-2 foi entre todas as linguagens estudadas aquela que apresentou condições mais favoráveis para adaptar-se às necessidades do projeto descrito a seguir, sendo esta a razão que levou-nos à sua escolha para o desenvolvimento do sistema proposto.

Em 1984, os professores Lidia Micaela Segre, da COPPE/UFRJ e Michael Stanton, da PUC/RJ, apresentaram uma proposta de trabalho (|33|, |34|, |35|) cujo objetivo principal era o de conduzir uma pesquisa sobre a construção de software distribuído, incluindo também aplicações de tempo real, para executar em arquiteturas compostas por um conjunto de estações autônomas sem memória compartilhada e interligadas através de um sistema de comunicação. Para esta classe de sofware faz-se necessário propor formalismos para os conceitos de paralelismo, configuração e confiabilidade.

Como já foi mencionado, o hardware de um sistema distribuído inclui diversos processadores, e portanto, suporta naturalmente a execução em paralelo de vários programas em estações distintas. Além disto, o processador de uma estação poderá empregar técnicas de multiprogramação. O conceito de programação concorrente engloba tanto o paralelismo real como o aparente; logo, também devem ser incluídos mecanismos para comunicação e sincronização entre as atividades concorrentes. Tendo em mãos um software que possua uma estrutura modular, a prevenção de falhas torna-se uma tarefa menos árdua, uma vez que utilizando-se esta estrutura existe a possibilidade de desenvolvimento e verificação de módulos de forma independente.

Apesar de Modula-2 possuir muitas características, foi preciso acrescentar-lhe algumas extensões para atender os conceitos de paralelismo e confiabilidade, requisitos que

são necessários para montar o ambiente de programação distribuída.

Em relação à configuração, fez-se necessário apresentar uma proposta de linguagem para especificação de configuração estática para programas distribuídos.

Depois de alguns estudos realizados chegamos à seguinte conclusão, a qual contribuiu ainda mais para certificarmo-nos de que a escolha de Modula-2 foi a melhor opção:

- as extensões à linguagem, necessárias para implementar os conceitos mencionados anteriormente, poderiam ser realizadas através da definição de uma linguagem de configuração e de uma biblioteca de módulos escritos quase inteiramente em Modula-2, sem que fosse preciso alterar a linguagem. Os módulos contidos nesta biblioteca seriam responsáveis por implementar programação paralela, tanto dentro de uma estação como em estações distintas, além de dar suporte para comunicação, sincronização e para tratamento de falhas.

As várias etapas do projeto, não independentes nem tão pouco consecutivas, podem ser apresentadas da seguinte maneira:

(i) Implementação de Modula-2 na máquina alvo (micro de 8 bits) (|8|).

(ii) Definição e implementação das extensões necessárias à linguagem Modula-2 (mecanismos de comunicação e sincronização) (|15|, |34|).

(iii) Implementação do núcleo do ambiente de suporte de programação (|32|).

(iv) Implementação de suporte de comunicação entre nós fracamente acoplados num sistema distribuído |5|.

(v) Definição e implementação de ferramentas para o suporte de programação em Módula-2. Entre estas ferramentas encontra-se a linguagem de configuração estática |35|, necessária para definir tanto os componentes lógicos que constituem o sistema distribuído como as ligações existentes entre eles, e também necessária para definir o mapeamento destes componentes na arquitetura física.

Como consequência deste trabalho foram desencadeadas várias teses de mestrado (|14|, |5|, |28|) de alunos da COPPE/UFRJ e da PUC/RJ, incluindo esta, e uma tese de doutorado |30|, além de vários artigos apresentados em congressos (|31|, |32|, |34|, |35|, |40|).

Esta tese, apresentada em 7 capítulos, dá ênfase à implementação do interpretador da linguagem de configuração, o qual provê as interfaces necessárias para a execução de uma chamada de procedimento remoto.

O Capítulo II descreve detalhadamente o projeto como um todo, mostrando as extensões introduzidas à linguagem Modula-2, como também o núcleo de multiprogramação e o gerador automático de módulos denominados de "sbuts" (BIRRELL e Nelson|4|). O núcleo e o gerador de "stubs" foram implementados para dar suporte à chamada remota de procedimento.

No Capítulo III é apresentada a especificação da linguagem de configuração estática para a qual foi implementado um interpretador.

O Capítulo IV tem como objetivo comparar o nosso modelo com algumas outras propostas encontradas na literatura.

No Capítulo V encontramos a descrição da implementação do interpretador da linguagem de configuração, apresentando as estruturas de dados utilizadas pelo interpretador, uma visão geral das rotinas desenvolvidas, e as interfaces que o interpretador provê para a execução de uma chamada remota de procedimento.

O Capítulo VI ilustra uma série de exemplos mostrando a potencialidade da linguagem, tomando um destes exemplos como base para apresentar os resultados obtidos no final da interpretação.

Finalmente o Capítulo VII traça algumas conclusões.

No Anexo I encontramos a listagem do interpretador que foi implementado.

CAPÍTULO II

DESCRIÇÃO DETALHADA DO PROJETO

Neste capítulo apresentaremos os conceitos que precisavam ser introduzidos à linguagem Modula-2 para obter-se um ambiente de programação distribuída. A estrutura geral do projeto também será mostrada, descrevendo-se o núcleo que foi implementado para dar suporte à multiprogramação, o gerador automático de "stubs" que foi desenvolvido para implementar chamadas remotas de procedimentos de modo transparente, e o protocolo de comunicação necessário para execução destas chamadas.

II.1 REQUISITOS DE UM AMBIENTE DE PROGRAMAÇÃO DISTRIBUÍDA

Não existe uma definição precisa para "sistema distribuído". O termo distribuído pode significar a distribuição de funções, dados, controle, unidades de processamento, podendo também significar a distribuição de uma combinação de vários destes itens citados.

Porém, os problemas envolvidos num sistema distribuído são facilmente reconhecidos, como por exemplo: paralelismo, comunicação e sincronização dos programas, gerenciamento dos recursos compartilhados, tratamento de falhas, transparência, e dependência do tempo.

Estas características introduzem um novo domínio de programação, denominado programação distribuída, o qual difere dos demais domínios existentes, que são a programação sequencial e a programação concorrente.

Segundo LI |17|, a programação distribuída pode ser caracterizada por um alto custo de comunicação e pela impossibilidade do uso de variáveis compartilhadas como

ferramenta de sincronismo.

Quando a distribuição está relacionada com a arquitetura física, ou seja, existem várias unidades de processamento, obtemos uma velocidade de execução mais alta, sem falar na confiabilidade disponível nestas arquiteturas devido à redundância dos componentes. Algumas características presentes nestes sistemas são: a existência de partes de programas em diferentes máquinas e a comunicação remota entre estas partes de programas.

Um fator que deve ser ressaltado é que muitos dos métodos utilizados em sistemas centralizados para sincronização e comunicação não podem ser empregados quando as funções de um sistema são distribuídas entre vários processadores independentes. Logo, faz-se necessário introduzir novos conceitos às linguagens já existentes, de forma que estas consigam satisfazer os requisitos de um ambiente de programação distribuída.

Um ponto que deve ser considerado em sistemas distribuídos é a propriedade da <u>flexibilidade</u>, uma vez que estes sistemas estão propensos a mudanças pelo simples fato de poussírem, em média, um longo tempo de vida, além de serem sistemas grandes. Estas mudanças podem ocorrer devido a uma ampliação de recursos e/ou funções, como também devido a uma reestruturação por razões operacionais. Porém, nem sempre é viável introduzir tais alterações paralisando-se o sistema todo, tornando-se necessário portanto, que tais mudanças sejam realizadas dinamicamente.

Intimamente relacionada à flexibilidade está a característica da <u>modularidade</u>, que é mais um requisito indispensável à programação distribuída. Quanto mais modular o software se apresentar mais fácil torna-se toda e qualquer modificação que precise ser efetuada. Porém, estas alterações só devem afetar os módulos que necessitem ser modificados, mantendo-se a interface existente entre estes

módulos e os demais inalterada. Por esta razão, é aconselhável que os sistemas sejam compostos por módulos autônomos com suas interfaces bem definidas.

Podemos citar como sendo outra vantagem fornecida pela modularidade, a reutilização, em novas aplicações, de módulos que já foram desenvolvidos para outros sistemas. Esta reutilização poderá ser realizada através da criação de uma biblioteca, onde nela estarão armazenados os módulos que poderão ser, diversificadamente, combinados para serem utilizados por sistemas distintos. A compilação separada dos módulos, entre outras vantagens, é um requisito fundamental para conseguir-se montar a biblioteca de módulos. O conceito de módulo está sendo incorporado cada vez mais às linguagens de alto nível.

Nos sistemas distribuídos é necessário que haja um gerenciamento tanto dos módulos que os compõem como das ligações entre eles, e um gerenciamento também da localização física referente a estes módulos. Todo este gerenciamento é denominado de configuração. As primitivas que implementam o gerenciamento podem estar embutidas na própria linguagem de programação, ou então separadas numa outra linguagem, formando uma linguagem de configuração.

Um requisito importante que não deve deixar de estar presente em sistemas distribuídos, principalmente para utilização por aplicações em tempo real, é o poder de controle explícito da ordem de execução dos processos, através da definição e implementação de políticas de escalonamento de processos não embutidas na linguagem.

A flexibilidade de definir políticas de gerenciamento dos processos ligados a dispositivos de hardware é outro requisito também desejável em sistemas distribuídos.

Um problema sério encontrado em algumas linguagens

projetadas para sistemas distribuídos é a falta de ênfase na questão de <u>tratamento de falhas</u>, tornando os sistemas baseados em tais linguagens, não realísticos nem tão pouco confiáveis.

Em arquiteturas distribuídas, apesar de haver redundância nos componentes de hardware como também no software, não pode-se descartar a possibilidade de ocorrência de falhas, seja nos processadores, seja na comunicação entre eles ou, até mesmo, nos demais dispositivos que compõem o sistema. Não é permissível, particularmente quando se refere a aplicações em tempo real, que o sistema não consiga recuperar-se caso ocorra algum defeito. Por outro lado, a inclusão de técnicas de tratamento a falhas nas linguagens para sistemas distribuídos deve ser cautelosa e segura, de forma a não introduzir ainda, mais fatores propensos a erros, e sim a garantir uma maior confiabilidade do sistema.

Por último queremos destacar a <u>concorrência</u> presente em sistemas distribuídos, consequência do paralelismo característico destes sistemas. É importante portanto, existir ferramentas para o suporte de criação de processos concorrentes, seja ela realizada tanto de modo estático como dinâmico. No primeiro caso, os processos são criados durante a fase inicial e mantidos durante todo o tempo em que o sistema estiver sendo executado. Já no modo dinâmico, os processos são criados e ativados apenas quando necessários, sendo destruídos após a sua utilização.

II.2 EXTENSÕES INTRODUZIDAS À LINGUAGEM MODULA-2

Como já foi mencionado, Modula-2 foi escolhida pelo fato de apresentar inúmeras vantagens, atendendo a maioria dos requisitos apresentados na seção anterior.

O conceito de módulo pode ser apontado como uma das principais vantagens da linguagem Modula-2, proporcionando um alto grau de flexibilidade ao sistema. O módulo protege, através de um envólucro, um grupo de declarações entre tipos, variáveis e procedimentos, estabelecendo assim, um escopo para os identificadores. Esta proteção impede o acesso arbitrário aos objetos pertencentes ao módulo,ou seja, os objetos declarados dentro de um módulo não são visíveis fora dele, da mesma forma que os objetos declarados externamente ao módulo são invisíveis no seu interior. Existe porém, uma maneira seletiva de quebrar a impenetrabilidade do módulo, que vem ser a declaração das listas de identificadores: a lista de importações e a lista de exportações.

Modula-2, utilizando o conceito de módulo, oferece recursos tais como ocultação de informação e abstração de dados.

Ao projetarmos um módulo, devemos definir exatamente tanto o que o módulo se propõe a fazer, como também os objetos necessários a serem exportados para que os outros módulos utilizem os serviços por ele fornecidos.

A definição destes objetos, denominada de interface de módulo, está contida no módulo DEFINITION MODULE, enquanto que a implementação real da função está no módulo IMPLEMENTATION MODULE.

O módulo de implementação contém os detalhes da representação e das operações a serem utilizadas; no entanto, o usuário da interface correspondente não tem nenhum acesso a esta representação. Uma consequência deste fato é a possibilidade de alterar, a qualquer momento, a representação usada dentro de um módulo de implementação, sem contudo afetar a interface associada a este módulo de implementação,e nem tão pouco a programação de outros módulos que utilizam esta interface.

A separação entre estes dois módulos - definição e implementação - apresenta a seguinte vantagem: apesar da especificação das interfaces ser divulgada, as suas respectivas implementações ficam restritas exclusivamente aos seus implementadores.

Com o conceito de módulos de definição e implementação é possível portanto, realizar um desenvolvimento modular de um programa.

O conceito de módulo possibilita agrupar declarações de objetos e selecioná-los de forma a serem visíveis para outros módulos. Isto facilita a definição de tipos abstratos de dados.

A teoria de abstração de dados conduz a tipos abstratos de dados, onde podemos definir um tipo de dados através da especificação de sua representação e das operações definidas sobre este tipo. A partir deste tipo podemos criar várias instâncias, através de declarações de variáveis. A representação interna de uma variável declarada de tipo abstrato é, de um modo geral, automaticamente inicializada ao ser declarada, porém em Modula-2, onde a implementação de tipos abstratos de dados é realizada através do conceito de módulo, é necessário introduzir uma inicialização explícita de toda variável declarada de tipo abstrato.

No que diz respeito à compilação separada, esta torna-se perfeitamente viável em Modula-2 devido à separação entre os módulos de definição e implementação, e também por causa das declarações explícitas das listas de importação e exportação. Utilizando-se estes recursos, já é possível em tempo de compilação testar a consistência dos módulos de forma individual, verificando-se que a especificação de um módulo de implementação está de acordo com a interface contida no módulo de definição correspondente, e verificando-se também que as chamadas para

um determinado módulo estão coerentes com a interface deste módulo chamado.

Um único ponto a ressaltar é a ordem em que deve ser realizada a compilação. Em particular, podemos compilar um módulo de definição antes de escrever o seu módulo de implementação associado. A própria estrutura lógica do programa determina uma ordenação parcial de compilação, através das seguintes duas regras:

i) qualquer módulo que importa identificadores pode ser compilado somente depois dos módulos de definição que exportam estes identificadores.

ii) um módulo de implementação pode ser compilado somente depois do módulo de definição correspondente.

Pelo fato de Modula-2 prover facilidades para o manuseio direto dos mecanismos de interrupção e de entrada/saída, é possível implementar diferentes políticas de gerenciamento dos processos referentes à interface homem-máquina. Dispondo de ferramentas como acesso a endereços absolutos de memória, manuseio de interrupções e possibilidade de relaxamento da verificação de tipos, o usuário poderá com mais facilidade escolher e implementar a política que melhor adapte-se às necessidades da aplicação.

A seguir apresentaremos as extensões introduzidas à linguagem Modula-2 no que diz respeito a:

a) <u>Concorrência</u>

Modula-2 foi projetada inicialmente para ser implementada num computador convencional com um único processador. No entanto, devido às facilidades básicas presentes nesta linguagem, existe a possibilidade de especificação de processos quase concorrentes e de real concorrência no caso de periféricos.

O conceito de processo é implementado em Modula-2 através de co-rotina, que vem ser um procedimento que transfere explicitamente o controle para outros procedimentos. Assim sendo, é obtido o suporte de tempo real para algumas aplicações onde é desejável um comportamento determinístico.

Como em Modula-2 co-rotinas são consideradas facilidades de baixo nível, seu tipo associado e seus operadores fazem parte do pseudo-módulo denominado SYSTEM.

A co-rotina é criada pela chamada do procedimento NEWPROCESS do pseudo-módulo SYSTEM, porém ela só será ativada quando outra co-rotina lhe passar o controle de execução através da chamada do procedimento TRANSFER, que também pertence ao pseudo-módulo SYSTEM. Quando uma co-rotina chama este procedimento, sua execução é suspensa e o controle é transferido para a co-rotina que foi passada como parâmetro na chamada. A co-rotina que recebeu o controle começa então a ser executada a partir do último ponto que ela se encontrava antes de ser suspensa.

A linguagem Modula-2 permite programar quase toda a variedade de modelos de processos concorrentes, suas interações (monitor, troca de mensagens) e diferentes políticas de escalonamento, através da programação de um módulo denominado núcleo, o qual suporta os conceitos apropriados. Num sistema distribuído, o núcleo estará presente em cada estação que compõe o sistema, fornecendo o suporte para a execução concorrente de vários processos.

b) <u>Comunicação e Sincronização</u>

A concorrência está fortemente ligada aos mecanismos de comunicação e sincronização, uma vez que os processos necessitam de meios seguros para transmitir dados para outros processos e, em alguns casos, para direcionar uma determinada sequência de execução.

A sincronização está implícita na semântica das chamadas de co-rotinas que, como já foi mencionado, transferem explicitamente o controle para outros procedimentos.

Existem dois modelos básicos que são utilizados para combinar os conceitos de paralelismo, sincronização e comunicação, e que definem um estilo de programação: sistemas orientados por troca de mensagens e sistemas orientados por chamadas de procedimentos. SCOTT discute em |29| esta classificação, considerando-a enganosa por confundir as características a ela associadas.

No caso do sistema possuir alguma memória compartilhada, as variáveis para uso comum são protegidas por um monitor, ou por alguma outra construção responsável por fornecer mecanismos para exclusão mútua no acesso a estes dados. A comunicação entre os processos é realizada através de chamadas a procedimentos pertencentes ao monitor.

Já no caso dos sistemas que não possuem compartilhamento de memória, a comunicação entre os diferentes processadores é baseada na troca de mensagens.

Ambos os modelos foram inicialmente aplicados a sistemas compostos por conjuntos de processos residentes num único processador; no entanto, eles podem ser estendidos para sistemas com mais de um processador interligados através de algum meio físico, compartilhando ou não memória.

Os dois modelos citados possuem sob certas restrições uma dualidade, uma vez que um programa realizado segundo um modelo tem seu correspondente direto no outro (LAUER e NEEDHAM |15|).

Apesar do sistema proposto pelo projeto não possuir memória compartilhada, optou-se pelo uso de chamada de procedimentos como modelo para implementação da comunicação

entre os processos remotos. Um ponto que contribuiu bastante para esta opção foi o fato das interfaces dos módulos na linguagem Modula-2 serem procedurais, e a interrelação entre os módulos ser realizada através de chamadas de procedimentos importados.

Logo, foi necessário estender o modelo baseado em chamadas de procedimentos para atender a comunicação entre os diversos processadores. O acréscimo realizado foi o conceito de chamada remota de procedimento (RPC), transparente e embutida na linguagem, como o principal mecanismo de comunicação entre as diferentes estações |24|.

Para a implementação de RPC foi escolhido o modelo que usa "stubs" gerados de forma automática (BIRRELL e NELSON |4|).

O núcleo de programação concorrente, as chamadas remotas de procedimento e o gerador de "stubs", serão sucintamente analisados mais adiante, nas seções II.3, II.4 e II.5, respectivamente.

c) Configuração

Para que seja possível definir os componentes que formam o sistema, assim como estabelecer as ligações existentes entre eles, e também o mapeamento físico destes componentes, é necessário que existam primitivas que ofereçam tais facilidades.

Estas primitivas podem estar numa linguagem separada ou serem introduzidas à linguagem de programação.

Com o intuito de manter a modularidade, devido as inúmeras vantagens que este conceito oferece na área de programação, foi definida uma linguagem de configuração separada; contudo, seus conceitos estão baseados na linguagem Modula-2.

A linguagem de configuração proposta será apresentada com mais detalhes no capītulo III.

II.3 NÚCLEO DE MULTIPROGRAMAÇÃO

As linguagens mais recentes para multiprogramação, entre elas Modula-2, incluem um método fixo tanto para representar atividades concorrentes como para a comunicação entre estas atividades, e incluem também um método para tratamento de entrada/saída.

Os sistemas escritos nestas linguagens estão baseados num programa residente, denominado comumente de núcleo ("Kernel"), que implementa processos, comunicação, sincronização E/S. Este núcleo é, geralmente, a parte mais crucial do sistema oeperacional, e determina a sua eficiência, confiabilidade e desempenho.

Modula-2 não oferece certas características disponíveis em outras linguagens, tais como procedimento pre-definidos para entrada/saída, alocação de memória e escalonamento de processos, mas ela possui as ferramentas básicas necessárias para programar estas características, de forma que o programador possa escolher os mecanismos e as políticas mais apropriadas em cada caso.

De forma a acrescentar facilidades de multiprogramação, foi escolhido o núcleo de Holt para implementar o conceito de monitor, provendo as políticas necessárias para o escalonamento, comunicação e sincronização dos processos.

Este núcleo é caracterizado pelo fato dos processos executarem seus respectivos códigos sequencialmente, sendo que o sincronismo entre eles é realizado através do acesso a variáveis compartilhadas.

Para que um processo possa ter acesso às variáveis, certas condições lógicas sobre estes dados devem ser testadas e assinaladas quando forem satisfeitas. Logo, o núcleo implementa duas primitivas, Espera e Avisa, as quais atuam sobre variáveis do tipo condição.

No núcleo desenvolvido para o projeto |32|, cada processo possui um descritor de processo associado. Todos os processos estão encadeados em filas, e cada um contém um apontador para o descritor do próximo processo.

Existem as filas de processos, uma para cada condição, a fila de processos prontos, e a fila associada a cada periférico. Esta última fila é necessária para liberar os processos bloqueados à espera do término da execução de alguma operação de entrada e saída referente a um periférico.

O primeiro processo na fila de processos prontos é aquele que está sendo executado no momento.

Cada processo estará associado a uma co-rotina, pois, como já vimos anteriormente, este conceito que está embutido na linguagem Modula-2, permite a implementação da concorrência imposta pelos sistemas distribuídos.

O núcleo Holt foi implementado através da declaração de um módulo, onde a parte de definição contém a interface a ser utilizada por diversos programas (nome dos procedimentos e tipos implementados pelo núcleo que podem ser importados por outros módulos), e na parte de implementação está a especificação dos procedimentos.

As primitivas implementadas pelo módulo NucleoHolt podem ser resumidamente apresentadas pela Figura (II.1), que mostra o módulo de definição.

```
DEFINITION MODULE NucleoHolt;
   EXPORT QUALIFIED  Filaproc, CriaProcesso, Espera, Avisa,
            FilaVazia, Inicializa, FimProcesso;
   TYPE FilaProc;

   PROCEDURE CriaProcesso (CodigoProcesso:PROC;tamanhopilha:
        CARDINAL);
     (*cria um processo associado ao código do procedimento
      CodigoProcesso e com uma área de  trabalho  de tamanho
      tamanhopilha*)

   PROCEDURE Espera (VAR C: FilaProc);
     (*coloca o processo numa fila esperando que outro sina-
     lize a condição C*)
     (*ou*)

   PROCEDURE EsperaP (VAR C:FilaProc; Pr:CARDINAL);
     (*coloca o processo numa fila ordenada pela  prioridade
      Pr esperando que outro sinalize a condição C*)

   PROCEDURE Avisa (VAR C:FilaProc);
     (*ativa um processo que esteja esperando  pela condição
     C*)

   PROCEDURE FilaVazia (C:FilaProc):BOOLEAN;
     (*informa se a fila de condição tem algum elemento*)

   PROCEDURE Inicializa (VAR C:FilaProc);
     (*inicializa a fila correspondente à condição C*)

   PROCEDURE FimProcesso;
     (*acaba a execução do processo e desloca a área de tra-
     balho dele*)

END NucleoHolt.
```

FIGURA II.1 - Módulo de Definição do NucleoHolt

Com esta implementação do núcleo, cada monitor corresponde a um módulo, o qual encapsula as estruturas de dados compartilhadas e condições associadas, sobre as quais atuam as primitivas Espera e Avisa. As operações do monitor |7| são implementadas através de procedimentos que são exportados e que serão executados com exclusão mútua. Esta exclusão mútua é garantida através do mecanismo de prioridade da linguagem Modula-2. É importante frisar que as primitivas Espera e Avisa devem ser usadas somente dentro de monitores, pois assim é garantida a integridade das filas de condição.

Baseado no NucleoHolt foi desenvolvido um núcleo de multiprogramação para o ambiente de programação distribuída, utilizando como base a linguagem Modula-2.

Uma vez que o nosso modelo de comunicação e sincronização entre processos remotos baseia-se na utilização de chamada remota de procedimento, o núcleo foi especialmente projetado de forma a atender as necessidades do Protocolo de Chamada Remota de Procedimento.

Este núcleo, denominado NUCLEO, possui primitivas de criação e destruição de processos, primitivas de sinalização, como também primitivas para manipulação de erros e primitivas de apoio.

Todas as primitivas do núcleo são visíveis para os outros módulos poderem importá-las, sendo que a utilização destas primitivas não está amarrada à chamada remota.

As primitivas de criação e destruição de processos permitem que o usuário implemente a sua própria política de otimização de memória, através do manuseio do endereço da área de trabalho associado ao processo.

As primitivas de sinalização são análogas às do NucleoHolt. Porém, foi implementada também uma primitiva denominada Timeout, cujo objetivo é delimitar o tempo de espera por um sinal.

Entre as primitivas de apoio encontra-se a primitiva WaitIO, que permite ao processo suspender-se para esperar por uma interrupção de E/S, e também a primitiva Delay, que atrasa o processo corrente por um tempo determinado pelo parâmetro de entrada.

No módulo NUCLEO podemos identificar a existência do módulo local Dispatcher, análogo ao módulo FatiadeTempo do NucleoHolt, o qual implementa o conceito de fatia de tempo

para uso do processador de forma compartilhada entre os diferentes processos a serem executados. Em lugar de deixar que cada processo monopolize o uso do processador até cedê-lo voluntariamente, é atribuído a cada processo uma fatia de tempo fixa para o uso do processador. Uma diferença importante a ser ressaltada, é a existência, no núcleo NUCLEO, de um processo ocioso, o qual tomará o controle quando todos os demais processos estiverem bloqueados. Já no NucleoHolt esta situação indica bloqueio perpétuo, uma vez que não existe o processo ocioso.

Pelo fato do núcleo ter sido implementado para ser executado no microcomputador PC/XT ou compatível, em ambiente MS/PC DOS 2.X ou 3.X, existe a restrição de parte do seu código ser dependente desta arquitetura, principalmente aquele referente ao tratamento de interrupção.

Em cada estação do sistema estará presente uma cópia deste núcleo, fornecendo assim, o devido suporte para a execução multiprogramada dos vários processos existentes.

O mecanismo de chamada remota de procedimento é aquele que permite que uma chamada realizada numa máquina, a nível de linguagem, seja automaticamente transformada em uma chamada, do mesmo nível, em outra máquina. Tudo acontece como se o processo migrasse temporariamente para outra máquina, a fim de executar o procedimento remoto. Este mecanismo precisa de um protocolo de rede para dar suporte à transferência de seus argumentos e resultados.

Tanto o núcleo de multiprogramação NUCLEO quanto o protocolo de chamada remota de procedimento (PCRP) foram desenvolvidos por DA SILVA, e maiores detalhes sobre suas implementações são encontrados em |5|.

II.4 CHAMADA REMOTA DE PROCEDIMENTO

Uma chamada remota de procedimento (RPC) é uma transferência de controle síncrona, a nível de linguagem, entre programas que estão em espaços de endereçamento disjuntos.

A idéia principal é que as chamadas remotas de procedimentos entre programas distribuídos pareçam-se exatamente da mesma forma que às chamadas locais das programações convencionais.

No entanto, existem alguns pontos que devem ser levados em conta, como por exemplo, o fato da ocorrência de falhas e a diferença do desempenho. Pelo simples fato das chamadas remotas acontecerem entre programas autônomos, a ocorrência de uma falha em um programa não acarretará, necessariamente, uma falha em outro programa, podendo até mesmo ser detectada, no caso da chamada remota não voltar para o programa que realizou esta chamada.

Este isolamento e detecção de falhas são propriedades fundamentais em programas distribuídos, sendo isto o que difere estes programas daqueles que utilizam chamadas locais, onde uma falha causará uma paralisação completa, tanto no processo que chamou quanto no que foi chamado.

Outra diferença que pode ser mencionada é a impossibilidade de serem passados, nas chamadas remotas, ponteiros ou endereços em geral. Isto porque estes dados só têm validade no espaço de endereçamento de quem faz a chamada. Desta maneira, é necessário um tratamento especial para o uso de estruturas encadeadas, como também para os parâmetros que são atualizados pelo procedimento chamado.

Uma chamada remota pode ser, ou não, transparente ao programador do módulo que vai executar a chamada do

procedimento, como também ao programador do módulo que possui o procedimento chamado. Usualmente, denominamos o módulo que executa a chamada do procedimento de módulo cliente, enquanto que o módulo que implementa o procedimento chamado, denominamos de módulo servidor.

O programador não deve sair de seu ambiente usual de programação toda vez que seu programa for distribuído por mais de uma máquina, mesmo que para lidar com detalhes de projeto e implementação para um protocolo específico à aplicação.

Por todas as razões mencionadas, a implementação de uma chamada remota pode apresentar uma sobrecarga que não deve deixar de ser considerada.

Como já foi dito na seção II.2, foi escolhido o modelo que usa o conceito de "stub" como implementação de chamadas remotas de procedimentos. O gerador de "stubs" será apresentado com mais detalhes na próxima seção.

Ilustraremos através da Figura (II.2), uma chamada realizada por um módulo (cliente) a um procedimento contido num módulo remoto (servidor).

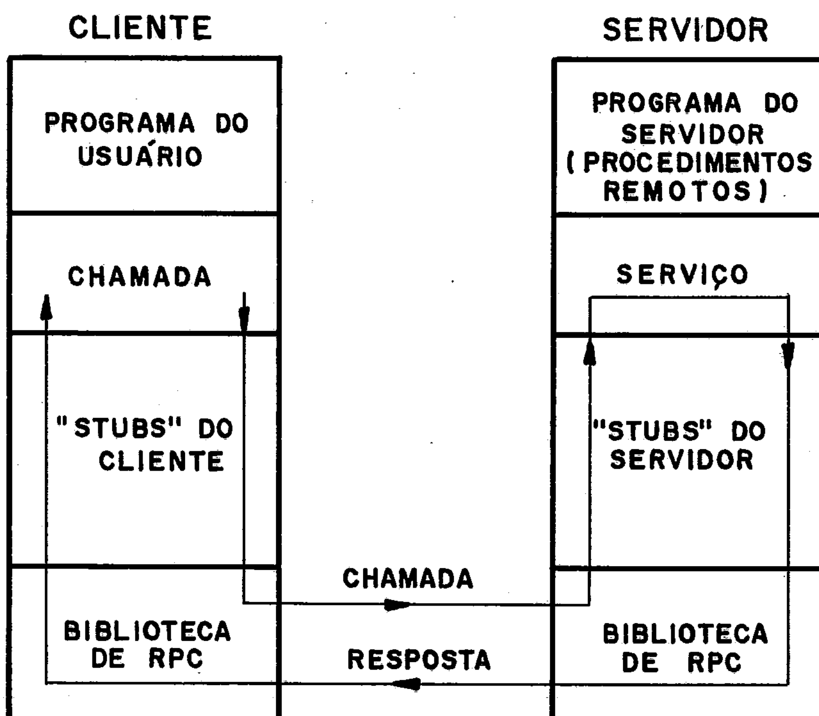


FIGURA II.2 - Mecanismo de uma Chamada Remota de Procedimento

II.5 GERADOR DE "STUBS"

Devido à escolha de usar uma RPC transparente embutida na linguagem, mantendo assim a mesma sintaxe utilizada nas chamadas locais, nenhuma palavra reservada foi acrescentada à linguagem Modula-2 e , consequentemente, não foi preciso modificar o compilador.

Para realizar a implementação de uma RPC foi adotado o método de geração automática de "stubs".

Este método consiste em criar "stubs" (um para o módulo cliente e outro para o módulo servidor) para cada módulo exportador.

Foram desenvolvidos, além dos "stubs", outros módulos que darão o necessário suporte ao programa na sua fase de execução, como por exemplo, o PCRP e um módulo de configuração, sendo este último utilizado pelo PCRP para ativar os procedimentos chamados.

O módulo de configuração juntamente com o interpretador da linguagem de configuração serão apresentados no capítulo V.

A seguir será descrito todo o procedimento realizado quando ocorre uma chamada remota.

Para que uma chamada remota pareça formalmente igual a uma chamada local, o controle da chamada é passado para o "stub" do cliente através de uma chamada local. Este, por sua vez, empacota numa mensagem o nome do procedimento que foi chamado, como também os seus argumentos.

Esta mensagem é enviada para a máquina que contém o procedimento servidor. O suporte de execução de RPC (chamado "RPC runtime") da estação cliente é o responsável pela transferência da mensagem.

O RPC runtime da estação servidora, ao receber a mensagem, ativa o "stub" do servidor através do módulo de configuração, passando todas as informações necessárias.

O "stub" do servidor tem como função identificar o conteúdo da mensagem e ativar o procedimento correspondente à chamada.

Quando a execução do procedimento acaba, o "stub" do servidor recebe os resultados, empacota-os e passa-os, através do módulo de configuração, para o RPC runtime da estação servidora, que por sua vez, passa a mensagem contendo os resultados para o RPC runtime da estação cliente que fez a chamada.

O stub do cliente que estava bloqueado à espera dos resultados, recebe a mensagem, desempacota-a e envia, em seguida, estas informações para o cliente. O cliente recebe os resultados dando continuidade à execução do seu código.

Resumidamente, podemos definir as funções dos "stubs" do cliente e do servidor, e também do RPC runtime, da seguinte forma:

i) <u>Funções do "Stub" do Cliente</u>

- Empacota o nome do procedimento chamado e seus argumentos.
- Ativa o RPC runtime da estação cliente.
- Desempacota os resultados contidos na mensagem de resposta.
- Envia os resultados para o cliente.

ii) <u>Funções do "Stub" do Servidor</u>

- Desempacota o nome do procedimento chamado e seus argumentos contidos na mensagem recebida.
- Identifica o servidor que contém o procedimento

chamado remotamente.

- Empacota os resultados na mensagem de resposta.
- Envia a mensagem com os resultados para o RPC runtime da estação servidora.

iii) <u>Funções do RPC Runtime</u>

a) Estação Cliente

- Recebe do "stub" do cliente a mensagem contendo o nome do procedimento e seus argumentos.
- Envia a mensagem de chamada para a estação servidora que contém o procedimento a ser ativado.
- Recebe da estação servidora a mensagem de resposta contendo os resultados.
- Envia a mensagem com os resultados para o "stub" do cliente.

b) Estação Servidora

- Recebe a mensagem com o nome do procedimento chamado e seus argumentos da estação cliente.
- Envia, através do módulo de configuração, a mensagem de chamada para o "stub" do servidor.
- Recebe, através do módulo de configuração, a mensagem de resposta proveniente do "stub" do servidor.
- Envia a mensagem contendo os resultados para a estação cliente.

Podemos observar que o RPC runtime da estação cliente possui funções análogas ao da estação servidora, com a única diferença que a comunicação entre este último e o "stub" do servidor é realizada através do módulo de configuração.

Queremos salientar que o suporte de execução de RPC precisa conter todas as funções descritas, tanto para

estação cliente quanto para a estação servidora, já que num mesmo nó físico podem estar localizados módulos clientes e módulos servidores. Além disto, um módulo não é caracterizado, especificamente, como sendo um módulo cliente ou um módulo servidor. Um único módulo pode exercer funções de cliente e servidor ao mesmo tempo, ou seja, ele pode chamar procedimentos de outros módulos (desempenhando a função de cliente), como pode também exportar procedimentos próprios para serem chamados por outros módulos (desempenhando a função de servidor).

Logo, será carregada uma cópia do suporte de execução de RPC em cada estação física que compõe a arquitetura do sistema.

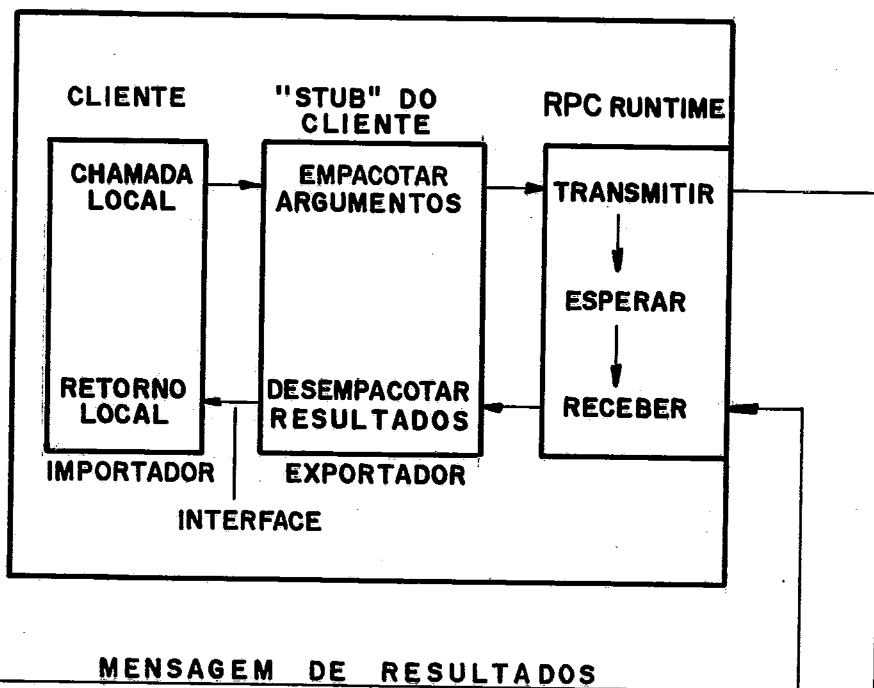
Através da Figura (II.3), podemos observar a interação que ocorre entre diversos componentes quando é executada uma chamada remota.

Os componentes envolvidos são: o cliente, o "stub" do cliente, o suporte de execução de RPC, o módulo de configuração, o "stub" do servidor, e o servidor.

Apesar do módulo de configuração estar presente em todas as estações, durante a execução de uma chamada remota ele só tem participação na estação servidora. Por esta razão o módulo de configuração não será ilustrado na estação cliente da Figura II.3.

Foi desenvolvido por LAGES |14| um gerador automático de "stubs" como parte de sua tese de mestrado.

Este gerador cria os "stubs" tanto para o exportador (servidor) quanto para o importador (cliente), sendo que os stubs são, nada mais nada menos, módulos escritos em Modula-2.



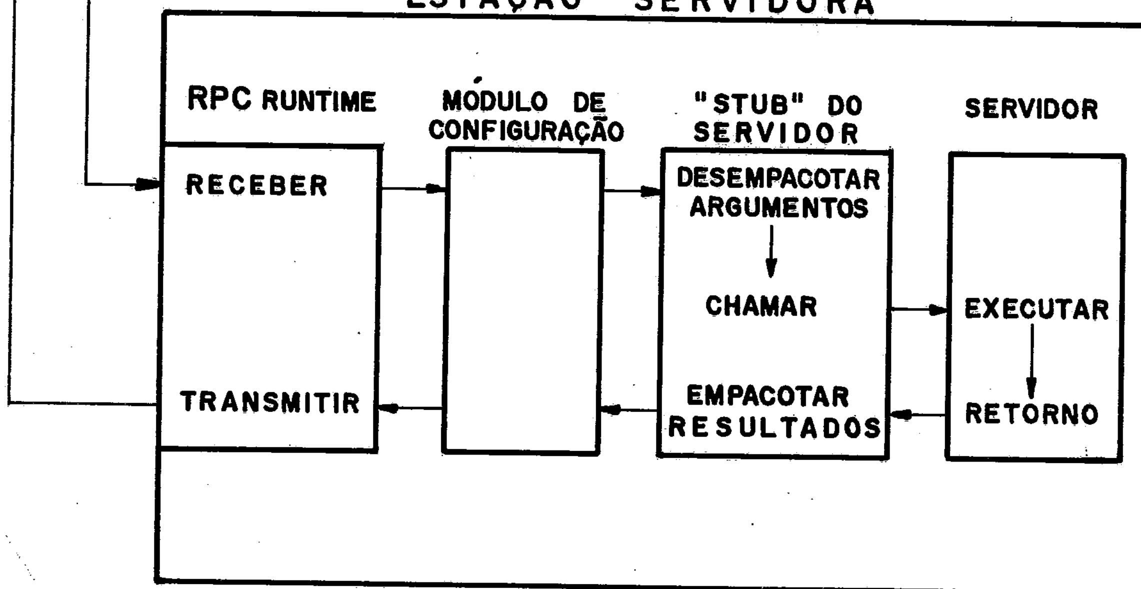


FIGURA II.3 - Interação entre os Diversos Componentes Envolvidos na Execução de uma Chamada Remota de Procedimento.

O gerador de "stubs" gera o par de "stubs" cada vez que é detectada uma interface exportadora, independentemente de saber se ela é remota ou não.

Se esta solução por um lado implica numa sobrecarga para o gerador, por outro falicita a reconfiguração, uma vez que no caso de chamadas locais se transformarem em remotas, consequência de uma relocalização dos módulos nas estações físicas, os "stubs" necessitarão ser somente carregados, e não gerados novamente.

II.5.1) "Stub" do Cliente

O "stub" do cliente é um módulo de implementação que simula o módulo exportado pelo servidor.

Para realizar esta simulação o "stub" do cliente implementa a mesma interface que o servidor que ele representa.

O "stub" do cliente precisa conter as entradas correspondentes a todos os procedimentos exportados, sendo que estes procedimentos são declarados da mesma forma que na implementação real.

II.5.2) "Stub" do Servidor

O "stub" do servidor é um módulo, que por não exportar nenhum objeto, não precisa estar separado em duas partes: definição e implementação, sendo apenas um módulo de programa.

O "stub" do servidor interage com o RPC runtime, o módulo de configuração e com o módulo exportador da interface, como foi mostrado na Figura (II.3).

O corpo do "stub" do servidor é constituído por um procedimento despachante que possui um comando CASE, cujas

alternativas correspondem aos procedimentos exportados, entre as quais será feita uma seleção para escolher o procedimento que foi chamado remotamente, fazendo com que ele agora seja ativado localmente.

Através do exemplo apresentado na próxima seção, os módulos que compõem tanto o "stub" do cliente quanto o "stub" do servidor para uma chamada remota, poderão ser mais facilmente visualizados.

II.6 EXEMPLO ILUSTRATIVO DA IMPLEMENTAÇÃO DE RPC

Vamos considerar um programa escrito em Modula-2 que represente um cliente que deseja importar o procedimento LePag, o qual faz a leitura de uma página de um determinado arquivo. Este procedimento é exportado por um módulo que contém um servidor de arquivo denominado ServArq, alocado em um nó físico diferente. O programa do cliente, neste caso, não exporta nenhum procedimento e portanto, será um módulo programa. A interface do módulo ServArq é ilustrada de forma resumida pela Figura (II.4).

```
DEFINITION MODULE ServArq:
EXPORT QUALIFIED LePag, LeCarac,.., PAGINA, BUFFER, ARQUIVO;
  TYPE     PAGINA =
           BUFFER =
           ARQUIVO =
  PROCEDURE LePag (pag:PAGINA; arq:ARQUIVO; VAR buf:BUFFER);
  PROCEDURE LeCarac (...
  .
  .
  .
END ServARq.
```

FIGURA II.4 - Modulo de Definição de ServArq

O módulo de implementação ServArq, que contém o procedimento LePag, é ilustrado de forma resumida pela Figura (II.5).

```
IMPLEMENTATION MODULE ServArq:
  .
  .
  .
PROCEDURE LePag (pag:PAGINA;arq:ARQUIVO;VAR buf:BUFFER);
  BEGIN
    .
    .      (*Implementação real de LePag*)
    .
  END LePag;

PROCEDURE LeCarac(.....):
  BEGIN
    .
    .
  END LeCarac;
  .
  .
  .
END ServArq.
```

FIGURA II.5 - Módulo de Implementação de ServArq

Não houve preocupação com a definição dos tipos PAGINA, BUFFER e ARQUIVO. O cliente deve importá-los porque são os tipos dos parâmetros do procedimento a ser chamado remotamente. Desta forma, o módulo do cliente terá, de forma resumida, a estrutura apresentada pela Figura (II.6). Foram utilizados, por uma questão de simplicidade, os mesmos nomes para os parâmetros.

```
MODULE Leitor:
FROM ServArq IMPORT LePag,PAGINA,ARQUIVO,BUFFER;
  .
  .
BEGIN
    .
    .
    LePag (pag,arq,buf);
      .
      .
END Leitor
```

FIGURA II.6 - Módulo do Cliente

A partir da interface do módulo ServArq, o gerador de "stubs" construirá os "stubs" do cliente e do servidor, os quais são ilustrados pelas Figuras (II.7 e II.8), respectivamente.

```
IMPLEMENTATION MODULE ServARq: (*STUB DO CLIENTE*)
  FROM SYSTEM IMPORT BYTE;
  FROM CONFIG IMPORT TipTablmp, Tiplmp, Tablmp;
  FROM NUCLEO IMPORT DWORD, ERROR;
  FROM   PRCP IMPORT  PACOTE,  RCall, ObjetoPacote, Alocapac

PROCEDURE LePag (pag:PAGINA;arq:ARQUIVO;VAR buf:BUFFER);
   BEGIN
     (*ordena parâmetros para variável par*)
     Despachante (par,nroproc);
     (*retorna para buf o resultado*);
END LePag;

PROCEDURE LeCarac(...);
  BEGIN
     .
     .
    Despachante (par,nroproc;
     .
     .
  END LeCarac;

PROCEDURE Despachante (VAR param:ARRAY(0..580) OF BYTE;
          np:CARDINAL);
                    LiberaPac;
VAR pac : PACOTE;
    nodo, indtab : CARDINAL;
 BEGIN
 Alocapac(pac);
  WITH paĉ DO
   dest:=nodo;
   mod:=indtab;
   ord:=nroproc;
   arg:=param
  END;
RCall (pac);
 IF ERROR () THEN ...
 param:=pac.res;
 LiberaPac (pac);
 END Despachante;

BEGIN
  (*consulta a tabela Tablmp para  obter valores das  variá-
   veis nodo e indtab*)
END ServArq
```

FIGURA II.7 - "Stub" do Cliente

```
MODULE ServArqServer: (*STUB DO SERVIDOR*)
  FROM CONFIG IMPORT TipTabExp, TipExp, TabExp;
  FROM PCRP IMPORT PACOTE, ObjetoPacote;
  FROM    ServArq  IMPORT   PAGINA, ARQUIVO, BUFFER,  LePag,
                LeCarac, ...;

  PROCEDURE Despachante(pac);
   VAR pac:PACOTE;
        PAR0:PAGINA
        PAR1:ARQUIVO,
        PAR2: BUFFER;
    BEGIN
      CASE pac.ord OF
        1: (*desordena os parâmetros do campo pac.arg)
           (para as variáveis PAR0,PAR1*)
           LePag (PAR0,PAR1,PAR2);
           pac.res:=PAR2;

        2: ;
           (LeCarac(...);
      END:  (*CASE*)
  END Despachante.

BEGIN
    (*O endereço do procedimento Despachante*)
    (*é colocado na tabela TabExp*)
END ServArq Server.
```

FIGURA II.8 - "Stub" do Servidor

Vamos agora descrever a execução de uma RPC na nossa implementação, mostrando a interação entre as diferentes interfaces já citadas através do exemplo da chamada do procedimento LePag pelo módulo Leitor. As setas numeradas da Figura (II.9) ilustram a sequência de passos que compõem a execução que listaremos a seguir.

A lista de passos é a seguinte:

1. transferência da chamada para o "stub" do cliente.
2. chamada ao PCRP do cliente passando o pacote com os argumentos do procedimento.
3. passagem do pacote do PCRP do cliente para o PCRP do servidor.
4. chamada do procedimento ligador do módulo de configuração do servidor, passando o pacote.
5. ativação do despachante correspondente à chamada re-

mota passando o pacote contendo os argumentos do procedimento.

6. execução real do procedimento ativado pelo "stub" do servidor (chamada local).
7. retorno do controle de execução para o "stub" do servidor com as variáveis atualizadas (resultados).
8. retorno do controle de execução para o procedimento ligador do módulo de configuração devolvendo o pacote com os resultados.
9. retorno do controle de execução para o PCRP do servidor devolvendo o pacote com os resultados.
10. passagem do pacote contendo os resultados para o PCRP do cliente.
11. retorno do controle de execução para o "stub" do cliente devolvendo o pacote com os resultados.
12. retorno do controle de execução para o cliente devolvendo os resultados da chamada remota.

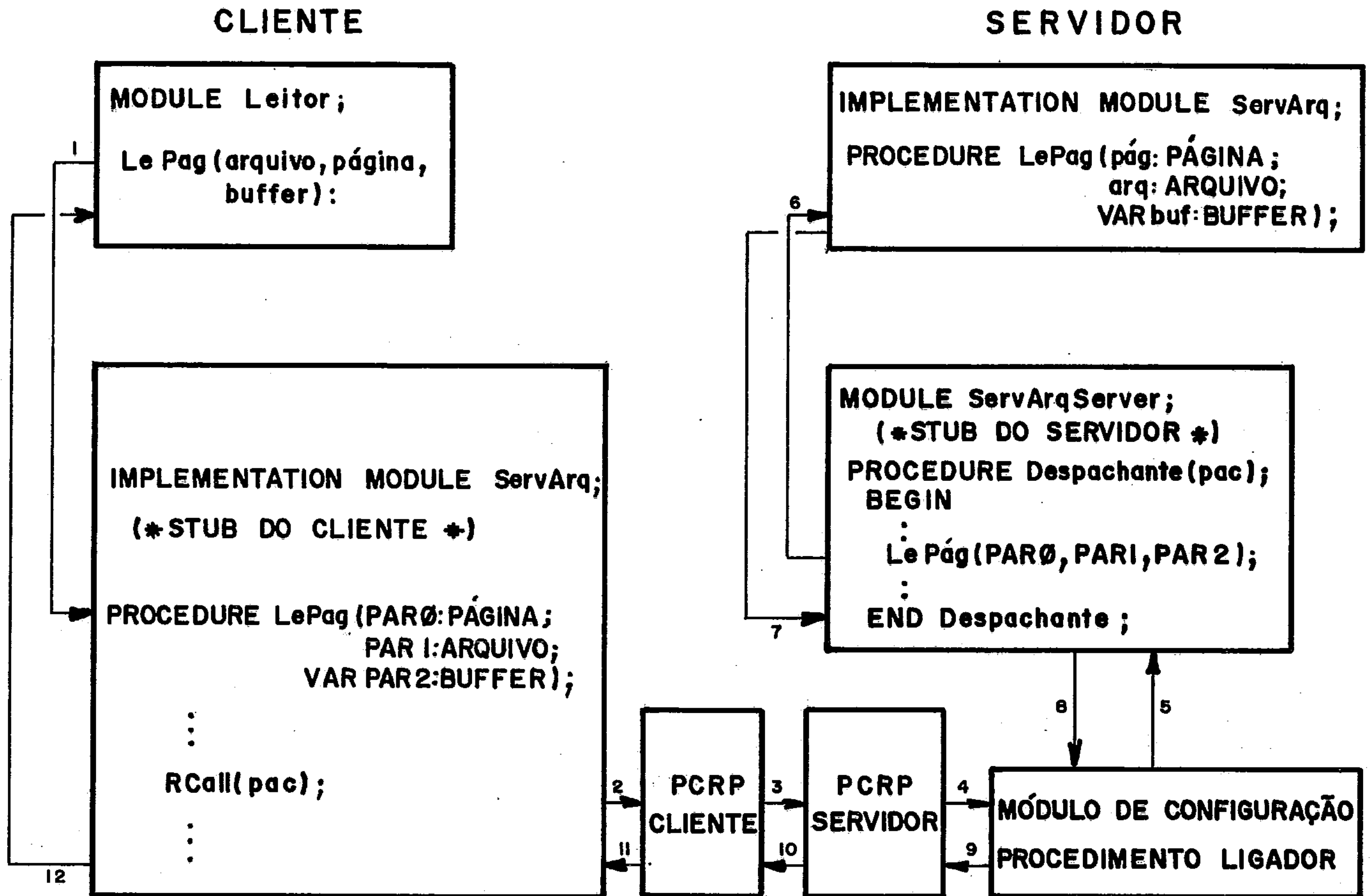


FIGURA II.9 - Etapas na Execução de uma Chamada Remota de Procedimento

CAPÍTULO III

DEFINIÇÃO DE UMA LINGUAGEM DE CONFIGURAÇÃO

Os principais conceitos sobre a necessidade de implementar uma linguagem de configuração para sistemas distribuídos serão apresentados neste capítulo, juntamente com a descrição da linguagem de configuração estática desenvolvida por SEGRE e STANTON |35|.

III.1 CARACTERÍSTICAS DA LINGUAGENS DE CONFIGURAÇÃO

Para configurar-se um sistema distribuído composto por um conjunto de módulos, faz-se necessário que a linguagem a ser utilizada ofereça declarações com as quais seja possível realizar o gerenciamento destes módulos. Em outras palavras, através das declarações serão especificados os módulos que vão compor o sistema, as ligações entre estes módulos, como também o mapeamento destes módulos na arquitetura física do sistema.

Foi apresentado por MAGEE |19| a divisão da especificação da configuração de um sistema distribuído em três partes apresentadas a seguir:

i) Estrutura Lógica

Nesta parte encontramos os tipos dos componentes de software a partir dos quais o sistema é construído, as instâncias destes componentes, e como estas instâncias estão interligadas, ou seja, qual instância se comunica com quais outras. Todas as informações sobre os requisitos que cada componente necessita, tais como memória, tipo de processador e periféricos, podem também estar incluídas na sua especificação.

ii) Estrutura Física

Descreve os componentes de hardware e a estrutura da sua interconexão física. Os componentes físicos são aqueles que processam os componentes lógicos. Associada a cada componente de hardware pode estar a especificação dos recursos por ele providos, tais como memória, processadores e periféricos.

iii) Mapeamento entre a Estrutura Lógica e a Física

Descreve a localização física dos componentes lógicos. Para obter-se flexibilidade deve-se permitir mapear um ou mais componentes lógicos num mesmo componente físico. A única restrição no mapeamento deve estar relacionada aos requisitos de recursos dos componentes lógicos, pois os recursos requisitados não podem exceder aqueles providos pelo componente físico associado a estes componentes lógicos.

Podemos identificar dois esquemas de descrição da especificação de um sistema: o estático e o dinâmico.

No estático supõe-se que dada a partida da execução do sistema, os detalhes da sua configuração permanecem inalterados, enquanto que, por outro lado, o esquema dinâmico permite que a configuração seja alterada mesmo depois que o sistema deu início à sua execução, sem ser necessário que esta seja paralisada.

Tratando-se de configuração dinâmica, a linguagem deve dispor de declarações de criação e destruição de processos, declarações para o desligamento das conexões entre os processos, enfim, a linguagem de configuração deve possuir declarações através das quais sejam elaboradas as mudanças desejáveis.

Apesar dos vários levantamentos realizados sobre as

vantagens e desvantagens, ainda não há uma definição sobre a necessidade da sepração entre a linguagem utilizada para descrever os componentes, denominada linguagem de programação em pequena escala (LPP), e a linguagem utilizada para integrar estes componentes formando assim o programa, denominada linguagem de programação em larga escala (LPL).

Modula-2 é uma linguagem adequada como LPP em sistemas centralizados, uma vez que ela permite o desenvolvimento e compilação em separado dos módulos, além de definir adequadamente as interfaces entre os diferentes módulos, através de mecanismos de importação e exportação de listas de declarações procedurais. Todos estes requisitos são fundamentais para poder configurar sistemas grandes e por esta razão torna-se fácil, através da definição de algumas extensões, utilizar Modula-2 como uma LPP também em sistemas distribuídos.

Na seção III.3 será apresentada a linguagem de configuração que foi desenvolvida baseada na linguagem Modula-2.

III.2 REQUISITOS NECESSÁRIOS NUMA LINGUAGEM DE CONFIGURAÇÃO

Como já foi dito anteriormente, um requisito fundamental que deve estar presente nas linguagens de programação é a <u>modularidade</u>, de forma que seja viável o desenvolvimento e a compilação em separado dos módulos, possibilitando também que a partir de módulos básicos sejam construídos facilmente módulos cada vez maiores, montando assim uma estrutura hierárquica.

Através da modularidade obtem-se o que denominamos de <u>independência de contexto</u>, já que os módulos não dependerão da configuração na qual o sistema será executado.

Já foi apresentada a importância dos módulos

possuirem uma <u>interface bem definida</u>. Com isto, o módulo só se comunica com o mundo externo através da sua interface, o que facilita, a nível da configuração, especificar e validar a interconexão entre os módulos.

Ao compilador, numa fase anterior à configuração do sistema, cabe a função de verificar a integridade dos tipos de dados utilizados individualmente por cada módulo. A nível da configuração, é necessário garantir que os módulos que são ligados entre si adotem a mesma definição de interface.

De forma a conseguir-se uma flexibilidade total de configuração, é importante que as declarações, oferecidas pela linguagem de programação distribuída para a comunicação entre os módulos, possuam a mesma sintaxe e semântica tanto para uma comunicação local (módulos que estão na mesma estação física) como para uma comunicação remota (entre estações físicas distintas). Logo, é desejável que a linguagem possua a propriedade de <u>transparência de comunicação</u>.

Para o caso de aplicações em tempo real, onde é indispensável um comportamento determinístico confiável, é interessante que, em tempo de compilação, sejam conhecidos os recursos requisitados por um módulo. Desta forma, quando o módulo for carregado na estação física será possível testar se todos os recursos necessários estão presentes.

Se a linguagem de programação possuir estas propriedades, torna-se fácil definir uma linguagem de configuração para os componentes escritos nesta linguagem de programação.

Logo é fundamental que uma linguagem de configuração possua declarações para poder especificar:

i) o conjunto dos tipos de módulos e dos tipos das sub-configurações, a partir do qual será definido o

contexto para construir o sistema.

ii) as interfaces de módulos importadas e as instâncias de módulos que serão exportadas.

iii) as instâncias de módulos e de sub-configurações que são criadas no sistema.

iv) as localizações lógicas que estão associadas às instâncias criadas.

v) as interconexões entre as instâncias de módulos e de sub-configurações.

Além de todos estes itens que acabaram de ser mencionados, a linguagem de configuração deverá mapear as estruturas lógicas, onde foram criadas as instâncias de módulos ou de sub-configurações, na estrutura física do sistema.

A especificação separada de cada uma das funções listadas anteriormente é outro fator, além da sintaxe das primitivas de comunicação presentes na linguagem de programação, que permite uma flexibilidade de configuração total.

Podemos destacar que existem alguns sistemas operacionais distribuídos que suportam configuração dinâmica, tais como UNIX (|11|, |27|), ROSCOE |39|, MEDUSA |26| e STAROS |9|. No entanto, se isto não está integrado com um sistema que contenha uma linguagem de configuração e de programação, não é possível validar as configurações testando a compatibilidade das interfaces dos componentes.

III.3 A LINGUAGEM DE CONFIGURAÇÃO PROPOSTA

Como em Modula-2 não existe o conceito de tipos de módulos, a primeira atitude providenciada foi a introdução deste conceito, pois as instâncias de módulos são obtidas a partir de tipos de módulos, de forma análoga às instâncias de variáveis.

A maneira alternativa que foi encontrada, sem contudo alterar a sintaxe da linguagem Modula-2, será descrita a seguir.

Em Modula-2 o nome de um módulo de implementação deve ser o mesmo que o do módulo de definição correspondente. Convencionalmente, o nome do arquivo que contém o módulo de implementação possui este mesmo nome também, porém é permitido que os nomes dos arquivos que possuem o código objeto de um módulo de implementação sejam diferentes. Então, temos dois nomes associados a um módulo de implementação: o primeiro que corresponde ao módulo de definição (sua interface), e o segundo que corresponde ao arquivo que contém o seu código objeto. Este último nome será utilizado como o nome do tipo de módulo para criar várias instâncias dele.

Existirá um segmento de dados próprio para cada instância de um tipo de módulo de implementação criada, porém, todas as instâncias de um tipo de módulo presentes na mesma estação física, poderão utilizar um único segmento de código.

Pelo fato de Modula-2, através de sua modularidade, permitir a separação entre a definição de uma interface e sua implementação, existe a possibilidade de se ter diferentes módulos de implementação para uma mesma interface. Desde que utilize-se o conceito anterior de tipo, é possível criar para uma mesma definição de interface de módulo, diferentes tipos de módulos de implementação, a

partir dos quais poderão ser criadas várias instâncias.

Passaremos agora a definir dois conceitos importantes utilizados na configuração.

i) estação lógica - ela é composta por um conjunto de módulos interligados, os quais formam uma unidade lógica que será carregada numa estação física.

ii) estação física - nela podem estar carregadas uma ou mais estações lógicas, porém cada estação lógica só pode ser carregada em apenas uma estação física.

As configurações definidas através da linguagem são compostas por módulos programados em Modula-2, ou por sub-configurações já definidas anteriormente, o que possibilita a formação de configurações hierárquicas. Tanto os módulos como as sub-configurações desempenham o papel de tipos, a partir dos quais várias instâncias poderão ser criadas. É importante destacar que as (sub-)configurações estão formadas unicamente de módulos ou de sub-configurações, não podendo-se misturar, numa mesma (sub-)configuração, módulos com sub-configurações. Logo, o primeiro nível de hierarquia da configuração é sempre composto por módulos programados em Modula-2, enquanto que os níveis subsequentes são compostos por sub-configurações.

A linguagem de configuração será interpretada por um programa escrito também em Modula-2, cuja implementação será descrita no capítulo V.

A partir de agora, apresentaremos a especificação da linguagem de configuração que foi desenvolvida para montar ambientes de programação distribuída baseados na linguagem Modula-2. O desenvolvimento desta linguagem de configuração

teve como base as linguagens CONIC e MESA.

A definição de uma configuração se inicia com a seguinte declaração:

CONFIGURATION: <Identificador do tipo da configuração>;

Com o intuito de mostrar o contexto da configuração, isto é, todos os tipos de módulos ou de sub-configurações, que serão utilizados dentro da (sub-)configuração para criar instâncias e para definir os tipos das interfaces importadas, usamos a seguinte declaração:

USE < lista de tipos de módulos de implementação ou sub-configurações e lista de interfaces de módulos >;

Cada (sub-) configuração é composta por módulos ou sub-configurações que pertencem a determinadas estações lógicas. Estas estações devem ser declaradas da seguinte forma:

STATIONS < lista de estações lógicas >;

Uma vez que uma configuração pode estar composta por módulos ou sub-configurações que pertencem a uma ou mais estações lógicas, a lista de estações lógicas pode conter mais do que um elemento.

Em Modula-2 os módulos interagem através de interfaces bem definidas, geralmente compostas de listas de procedimentos exportados. Estes procedimentos serão chamados por outros módulos, os quais precisam importar explicitamente as interfaces que contêm tais procedimentos.

Para ser possível realizar as ligações dentro das sub-configurações e entre diferentes sub-configurações, é necessário declarar explicitamente as importações de

interfaces de módulos e as exportações de instâncias de módulos de implementação. Estas declarações são opcionais, isto é, nem todas as configurações importam interfaces de módulos ou exportam instâncias de módulos de implementação, por exemplo, a configuração de mais alto nível, numa estrutura hierárquica, pode não importar interfaces de módulos nem exportar instâncias de módulos de implementação.

A sintaxe destas declarações é a seguinte:

IMPORT < lista de interfaces de módulos>;

A lista de interfaces de módulos tem a seguinte sintaxe, na qual as chaves indicam a possível repetição.

{ < nome local de interface de módulo > : < tipo da interface importada>; }

EXPORT < lista de instâncias de módulos de implementação>;

A lista de instâncias de módulos de implementação tem seguinte sintaxe:

{ < nome local do módulo de implementação>: < tipo do módulo de implementação>; }

A declaração indica, no caso de importação, o nome da interface do módulo e seu tipo, e, no caso de exportação, o nome do módulo de implementação e seu tipo. Neste último caso, o módulo de implementação corresponde à instância criada na configuração, a qual contém a declaração de exportação. Na importação, o tipo de interface de módulo corresponde a um módulo pertencente a outra configuração, e o nome usado tem as características de um parâmetro formal no módulo que declara a importação. Pode ser que os nomes dos tipos coincidam nas duas declarações, de importação e

exportação, mas em geral, não será assim, já que quando existir mais de um tipo de módulo de implementação para uma mesma interface, os nomes serão necessariamente diferentes.

No caso das sub-configurações que não pertencem ao primeiro nível de hierarquia, pode acontecer das importações e exportações não casarem dentro da sub-configuração. Neste caso é preciso que elas sejam propagadas para serem associadas às sub-configurações às quais elas pertencem. Esta propagação se faz repetindo-se as declarações de importação e exportação correspondentes. Para estes casos, existem regras de escopo para a visibilidade das interfaces de módulos importados e dos módulos de implementação exportados. De forma a não violar a visibilidade desejada, é necessário tomar cuidado de como repetir as declarações. Para isto, nas configurações aninhadas, na hora de fazer as ligações com os módulos exportados, estes serão qualificados unicamente pelo último nível no qual foi feita a declaração, não precisando explicitar a cadeia de sub-configurações à qual pertencem.

Quando uma mesma (sub-) configuração precisa declarar importações e exportações, esta ordem das declarações deve ser respeitada, já que em alguns casos, as instâncias dos módulos que são exportados podem utilizar alguma informação das interfaces de módulos importados, de forma análoga à importação e exportação de interfaces nos módulos de Modula-2.

Por exemplo, em configurações diferentes poderiam estar as duas declarações seguintes:

Numa configuração denominada NurseConf temos a declaração

```
EXPORT Selector : psel;
       ConsoleNurse : zterm;
```

enquanto que em outra configuração, denominada PatientConf, temos a declaração

```
IMPORT Select : psel ;
       ConsoleN : zterm;
```

Numa fase posterior poderemos então, ligar uma ou mais instâncias de PatientConf a uma instância de NurseConf, casando assim as listas de exportação e importação.

A próxima declaração define as instâncias de módulos ou de sub-configurações que compõem a configuração, explicitando também a estrutura lógica associada.

A sintaxe da declaração CREATE é a seguinte:

CREATE <lista de criação de instâncias de módulos de implementação >;

ou

CREATE <lista de criação de instâncias de sub-configurações>;

A lista de criação de instâncias de módulos é composta por declarações com a seguinte sintaxe:

```
{ <um ou mais nomes      <tipo do módulo              <nome da estação }
{    de módulos > :       de implementação> ON          lógica> ;      }
```

A declaração CREATE cria várias instâncias de módulos de implementação de tipos definidos, indicando em todos os casos a sua localização lógica.

A lista de criação de instâncias de sub-configurações é composta por declarações com a seguinte sintaxe:

{ <um ou mais nomes de configurações >: <tipo da configuração >ON <lista de estações lógicas > }

A lista de estações lógicas deve possuir as estações numa ordem e número que correspondam às estações declaradas anteriormente, na declaração STATIONS que foi definida dentro da declaração do tipo da configuração.

Depois de ter definido as instâncias de módulos ou de sub-configurações que compõem a (sub-)configuração, é necessário estabelecer as ligações entre importadores e exportadores. Isto é realizado através da declaração LINK, que tem o seguinte formato:

LINK <lista de ligações de instâncias de módulos ;
ou
LINK <lista de ligações de instâncias de sub-configurações;

A lista de ligações de instâncias de módulos está composta por declarações com a seguinte sintaxe:

< um ou mais nomes de instâncias de módulos > WITH < um ou mais nomes de instâncias de módulos > ;

Esta declaração liga uma instância de cada um dos módulos importadores, posicionados à esquerda da palavra WITH, com as instâncias dos módulos exportadores, indicadas à direita da palavra WITH. Estas últimas são os exportadores das interfaces que são importadas pelas instâncias de módulos especificadas à esquerda da palavra WITH. É aqui então, que é realizada a associação entre a interface do módulo importada com a instância do tipo do módulo de implementação associada a esta interface, além de definir-se as ligações das instâncias.

A associação de exportadores com as interfaces importadas utiliza o modelo posicional, isto é, a lista de

nomes de instâncias de módulos à direita da palavra WITH deve corresponder em número, ordem e tipo à lista de importações de módulos declarada dentro do tipo de módulo que está à esquerda da palavra WITH.

Quando as instâncias de módulos à esquerda da palavra WITH possuirem listas de importações idênticas, estas instâncias poderão ser combinadas numa única declaração, colocando-se uma lista dos nomes das instâncias importadoras à esquerda da palavra WITH, e as listas de instâncias de módulos exportadores correspondentes às listas de importações, à direita da palavra WITH.

As instâncias dos módulos nomeadas depois da palavra WITH podem ou não estar na mesma estação lógica ou até na mesma configuração,que as instâncias dos módulos nomeados à esquerda da palavra WITH. Dependendo da localização física das instâncias dos módulos, estas ligações podem ser locais ou remotas. Isto será definido no final da configuração, quando o mapeamento entre a configuração lógica e a configuração física é especificado.

A consistência entre as declarações de importações e exportações, a nível dos módulos, já foi realizada durante a compilação dos módulos. O interpretador da linguagem de configuração precisa então, só conferir que nas instâncias nomeadas à esquerda da palavra WITH, as importações de interfaces de tipos de módulos correspondem aos tipos de módulos de implementação à direita da palavra WITH.

A lista de ligações de instâncias de sub-configurações é composta por declarações com a seguinte sintaxe:

{ <um ou mais nomes de instâncias de sub-configurações> WITH <um ou mais nomes de instâncias de módulos qualificados> ; }

<nome da instância do módulo qualificado> ::= nome da instância da sub-configuração . nome da instância do módulo de implementação

Esta declaração liga uma instância de cada uma das sub-configurações, que aparecem à esquerda da palavra WITH, a uma ou mais instâncias de tipos de módulos de implementação. Estas instâncias serão determinadas através da identificação da hierarquia de configurações à qual pertencem. Como já foi mencionado anteriormente, quando houver propagação das exportações, a instância do módulo será qualificada somente pelo último nível de sub-configuração que contém a sua declaração de exportação.

Cabe ao interpretador verificar que as instâncias dos módulos citados como exportadores nas declarações LINK, sejam realmente compatíveis com as interfaces importadas pelas instâncias de módulos e de sub-configurações importadoras.

Por causa de não poder combinar módulos e sub-configurações numa mesma (sub-) configuração, pode surgir a situação de precisar declarar uma sub-configuração contendo uma única instância de módulo, e neste caso não haverá declaração LINK; portanto esta declaração também é opcional.

Uma idéia global de como seria a declaração de uma configuração pode ser obtida através da Figura (III.1).

```
CONFIGURATION: <nome do tipo de configuração>;
   USE   <lista de tipos de módulos de implementação ou
         sub-configurações e lista de interfaces de módulos>;
   STATIONS  <lista de estações>;
   IMPORT    <lista de interfaces de módulos>;
   EXPORT    <lista de instâncias de módulos de
              implementação>;
   CREATE    <lista de criação de instâncias de módulos de
              implementação ou de instâncias de
              sub-configurações>;
   LINK      <lista de ligações de instâncias de módulos ou
              de instâncias de sub-configurações>;
END  <nome do tipo de configuração>.
```

FIGURA III.1 - Definição da Declaração de Configuração

O último passo para concluir a fase de configuração é especificar o mapeamento entre a configuração lógica e a configuração física, ou seja, definir a localização física dos módulos que compõem a configuração. Isto pode ser feito usando a seguinte diretiva:

```
LOAD  <nome da instância         :  <nome do tipo de
       de configuração>              configuração>

                 ON < lista de estações físicas>.
```

A lista de estações físicas pode conter nomes simbólicos ou endereços físicos, mas ela deve corresponder em número e ordem à lista de estações declaradas na definição da configuração.

Esta declaração será utilizada para o nível mais alto de hierarquia da configuração, isto é, depois de definida a configuração total do sistema distribuído. Queremos esclarecer que o nível mais alto será representado, na maioria dos casos, por um único tipo de configuração, englobando todas as partes que o compõem. Mas pode também estar formado por mais de um tipo de configuração, permitindo deixar algumas ligações soltas de maneira a serem combinadas depois com outras partes necessárias ao sistema.

CAPÍTULO IV

COMPARAÇÃO ENTRE ALGUMAS LINGUAGENS DE CONFIGURAÇÃO

Neste capítulo serão apresentadas algumas linguagens que dão suporte à construção de sistemas distribuídos. Será realizada uma comparação entre estas linguagens e a proposta que foi apresentada para programação distribuída com base na linguagem Modula-2. As vantagens e desvantagens de cada uma das linguagens serão ressaltadas de forma a fornecer condições de distinguir-se, entre elas, aquela que melhor se adpata às características de um determinado ambiente de programação distribuída. As linguagens escolhidas para comparação foram: Conic (|12|, |13|, |19|, |20|, |21|, |22|, |37|, |38|) e Conic/C (|6|), Mesa e C/Mesa (|16|, |23|) e Mascot3 (|13|, |36|).

IV.1 CONIC E CONIC/C

Conic é uma linguagem que foi definida como uma extensão à linguagem PASCAL, e projetada para dar suporte à construção e operação de software para sistemas distribuídos.

Esta linguagem foi concebida tendo em mente o modelo de máquinas hospedeira/alvo, provendo por um lado, um conjunto de ferramentas para compilação, ligação, depuração e execução dos programas na máquina hospedeira, e por outro lado, fornecendo nas máquina alvo suporte a operações distribuídas.

Em Conic define-se um módulo do tipo "task" como sendo a unidade de programação sequencial, ou seja, este tipo de módulo desempenha o papel de processo. As instâncias dos módulos são criadas a partir destes tipos.

O módulo "task" provê uma independência da

configuração na medida que todas as referências são realizadas a objetos locais, não existindo nenhuma nomeação direta de outros módulos ou entidades de comunicação. Isto significa que não existe nenhuma informação de configuração embutida na linguagem de programação e, consequentemente, nenhuma recompilação é necessária para realizar mudanças na configuração. De forma análoga a Modula-2, os módulos "task" da linguagem Conic são os componentes básicos, a partir dos quais todo o sistema distribuído é construído. Estes módulos podem ser reutilizados em muitas situações diferentes.

Outra unidade utilizada em Conic é aquela denominada "unidade de definição", cuja função é o encapsulamento das constantes, dos tipos e dos procedimentos, comuns aos módulos e, consequentemente, compartilhados pelos mesmos.

Estas unidades de definição podem ser compiladas separadamente podendo ser importadas por um módulo para que este defina o seu contexto.

As unidades de definição permitem introduzir extensões a linguagem, tais como definições de "string" e procedimentos de manipulação, sem que seja necessário modificar o compilador. As unidades de definição podem também definir dados e inicializar o código, provendo a mesma facilidade que os módulos proveêm na linguagem Modula-2.

As interfaces entre os módulos são definidas através de portas de entrada e portas de saída.

Uma porta de entrada pode ser vista como sendo um "buraco" através do qual as mensagens podem ser passadas para dentro do módulo.

Por outro lado, uma porta de saída pode ser considerada como um "buraco" pelo qual as mensagens são

enviadas para fora do módulo.

A ligação entre uma porta de saída e uma porta de entrada faz parte da especificação da configuração, e não é realizada dentro de um módulo "task" de programação.

As operações primitivas utilizadas nas portas, que são a de recepção e a de envio de mensagens, suportam dois tipos de comunicação: "request-reply" e "notify transaction".

O tipo "request-reply" provê troca de mensagem síncrona e bidirecional, enquanto que o tipo "notify transaction" é assíncrono e unidirecional.

No tipo "notify transaction" o transmissor da mensagem não fica bloqueado, embora o receptor possa estar bloqueado esperando a mensagem. Já no caso do tipo "request-reply", o transmissor é bloqueado até que uma resposta seja recebida do receptor, enquanto que este último pode ficar bloqueado a espera de um pedido.

Na linguagem Conic, o conceito de tipo foi estendido para as portas e mensagens, fazendo com isto que a comunicação das mensagens entre os módulos seja verificada a nível de seus tipos definidos, de forma similar ao uso de tipos com as variáveis.

Logo, nas declarações Entryport e Exitport, as quais definem, respectivamente, as portas de entrada e de saída, encontramos o tipo da mensagem e o tipo da comunicação (síncrona ou assíncrona) explicitamente definidos.

Em Conic, o nó lógico é considerado a unidade de configuração, podendo ser definido como um conjunto de tarefas, as quais são executadas, concorrentemente, dentro do mesmo espaço de endereçamento.

Os sistemas são construídos como conjuntos de um ou mais nós lógicos interconectados.

A comunicação entre as tarefas dentro de um nó lógico, e entre os diversos nós lógicos, é suportada através de troca de mensagens.

Foi definida uma linguagem de configuração separada, denominada Conic/C. Os motivos que levaram os autores a optarem pela separação entre a linguagem de programação e a linguagem de configuração, foram as vantagens obtidas com esta escolha, como por exemplo, a flexibilidade, a modularidade e a reutilização dos componentes de software em diferentes configurações.

A linguagem de configuração é empregada para especificar a configuração dos componentes de software, os nós lógicos, e o mapeamento destes nós lógicos na arquitetura física que compõe o sistema.

A especificação da configuração identifica os tipos de módulos através dos quais o sistema será construído, declara as instâncias destes tipos, como também estabelece as interconexões de instâncias através das ligações entre as suas portas de entrada e de saída. Estas três funções são denominadas, respectivamente, de definição do contexto, instanciação de módulos e interconexão de módulos.

Os padrões de interconexão suportados pela linguagem são: "um-para-um", "um-para-vários" e "vários-para-um".

Estruturas hierárquicas podem ser representadas através de especificações de configurações aninhadas.

Os tipos de módulos utilizados na descrição do sistema podem ser módulos do tipo "task" ou sub-sistemas definidos como um módulo do tipo "group". O tipo "group" é, por si próprio, uma especificação de configuração, e pode

estar formado por módulos do tipo "task" ou por outros módulos do tipo "group". Além disto, ele possui declarações de instâncias e interconexões. A interface de um módulo do tipo "group" também é definida em termos de portas de entrada e de saída. Desta forma, é impossível, a nível externo, distinguir entre um módulo do tipo "task" e um do tipo "group".

De forma a executar as possíveis mudanças necessárias em sistemas grandes, a linguagem Conic dá suporte à configuração dinâmica, isto é, ela oferece recursos que tornam viável realizar alterações no sistema enquanto ele está sendo executado.

Passaremos agora a apresentar algumas declarações da linguagem de configuração Conic/C:

O arquivo que possui os tipos comuns, os quais são compartilhados pelos diferentes módulos que contêm a sua importação, é indicado através da declaração DEFINE. Já os tipos de mensagens e os tipos de módulos, necessários para definir o contexto do sistema, são listados através da declaração USE. As instâncias dos tipos de módulos são criadas através da declaração CREATE. Pode ocorrer de algumas instâncias utilizarem parâmetros. As conexões entre as instâncias de módulos são realizadas utilizando-se a declaração LINK. As portas de saída são ligadas com as portas de entrada de diferentes módulos, sendo que elas precisam ter sido declaradas do mesmo tipo para poderem ser casadas. A compatibilidade tanto do tipo quanto da operação das portas são verificadas.

Apesar de ter havido uma preocupação, desde o início do projeto da linguagem Conic, em prover configuração dinâmica, as idéias foram evoluindo no decorrer do desenvolvimento do projeto.

Foram, no entanto, acrescentadas à linguagem de

configuração primitivas de destruição dinâmica de processos, de desligamento de portas, e comandos para elaborar as mudanças, que serão processadas por um gerenciador de configuração.

Estas primitivas são funções inversas daquelas que foram apresentadas acima: "unlink" (link), "delete" (create) e "remove" (use).

As declarações utilizadas para definir as mudanças de uma configuração devem estar após à declaração "change".

Por tudo que foi apresentado, podemos observar que o projeto Conic serviu como base para o desenvolvimento de nossos trabalhos. Porém, existem algumas diferenças as quais serão ressaltadas a seguir.

Enquanto que no nosso caso as interfaces são definidas através de importações e exportações de procedimentos, em Conic a comunicação é realizada através de portas de entrada e de saída.

Deve ser observado, no entanto, que tanto num caso como no outro, o programador projeta e implementa os tipos de módulos sem a preocupação de especificar, na comunicação, o nome do módulo transmissor/receptor. Do ponto de vista do programador, os nomes utilizados nas declarações são locais. A associação é realizada no estágio final da configuração. Isto facilita a modificação e a extensão do sistema através de uma reconfiguração.

Outra diferença observada, é o fato de se criar as instâncias, em Conic, sem mencionar as estações lógicas às quais elas estão associadas. O mapeamento físico das estações é realizado apenas no nível mais alto da hierarquia, através da indicação explícita do endereço físico do nó juntamente com a lista de módulos que o compõem.

Já na nossa linguagem, primeiramente definimos uma estrutura lógica, declarando-se explicitamente a localização lógica de cada instância criada, para depois então, realizar-se o mapeamento desta estrutura lógica na estrutura física. A especificação das estações lógicas e físicas resulta numa melhor identificação da configuração do sistema.

Outra vantagem que a nossa linguagem possui sobre a linguagem Conic/C é o fato de ser possível, em Modula-2, separar-se as interfaces, da parte de implementação. Desta forma pode-se criar diferentes módulos de implementação para uma mesma interface, o que já não ocorre em Conic, uma vez que as interfaces dos módulos são definidas implicitamente através das declarações locais de portas de entrada e de saída. Com isto, a nossa linguagem torna-se mais flexível e poderosa.

Por outro lado, no que se refere à ligação, em Conic as portas de entrada sempre são ligadas, de maneira explícita, com as portas de saída do mesmo nível hierárquico.

Isto já não acontece com a nossa linguagem, uma vez que as ligações do primeiro nível hierárquico são realizadas de forma diferente das ligações dos demais níveis. No primeiro nível de hierarquia, liga-se interfaces de tipos de módulos com instâncias de módulos de implementação, enquanto que nos demais níveis, no lugar dos nomes das interfaces de tipos de módulos, aparecem os nomes de sub-configurações, as quais são qualificadas pela instância de módulo de implementação.

Logo, em Conic existe uma coerência maior no que se refere às ligações, embora utilizando-se a nossa linguagem, seja possível chegar às interfaces de tipos de módulos, percorrendo-se todas as sub-configurações que compõem a sub-configuração nomeada na declaração LINK.

Não podemos deixar de destacar o empenho que os autores de Conic vêm dando à configuração dinâmica, muito embora, eles não tenham ainda conseguido resolver todos os problemas pertinentes a este assunto. Inclusive porque, tratando-se de configuração dinâmica, muitas vezes fica a cargo do programador controlar as condições necessárias para realizar a reconfiguração do sistema.

Tanto na nossa linguagem quanto em Conic, as mesmas primitivas são utilizadas para comunicações locais e remotas, trazendo como vantagem, a fácil portabilidade para diferentes arquiteturas.

IV.2 MESA E C/MESA

Mesa é uma linguagem de programação projetada para implementar sistemas, os quais são compostos por módulos.

O módulo é a unidade básica de compilação em Mesa, e existem dois tipos de módulos: os módulos de definição e os módulos denominados de programas.

Os módulos de definição servem para especificar como as partes do sistema serão ligadas. Quando eles são compilados provêem um conjunto de definições comuns, as quais podem ser referenciadas por outros módulos que estão sendo compilados.

O módulo do tipo programa contém os dados e o código executável. Quando o programa é compilado, obtem-se como saída um módulo objeto, o qual é um arquivo binário contendo o código objeto, a tabela de símbolos, e as estruturas de dados. Todas estas informações serão utilizadas para conectar este módulo com os demais.

Uma instância de um módulo gerencia uma coleção de objetos, provendo um conjunto de procedimentos para criar,

operar e destruir tais objetivos.

As interfaces declaradas no módulo de definição são utilizadas para importar e exportar facilidades de um para outros módulos.

A definição de uma interface pode ser particionada em duas partes, a parte estática e parte de operações, sendo que cada uma delas não precisa, necessariamente, estar preenchida.

Na parte estática estão contidos os tipos e constantes que serão compartilhados entre o módulo que importa e o que exporta a interface.

Já a parte de operações define as operações que estarão disponíveis para serem importadas. Em geral, as operações são definidas em termos de procedimentos e sinais.

O módulo que utiliza uma interface é denominada de importador enquanto que aquele que implementa a interface é denominado de exportador.

Após a compilação dos módulos do tipo programa, são gerados dois conjuntos: um de registros de interfaces importadas (interface records) e um de registros exportados (export records).

Para cada procedimento e/ou sinal definido na interface, existe, no módulo exportador, uma declaração deste procedimento e/ou sinal com o atributo PUBLIC.

O compilador certifica-se que os tipos no exportador são compatíveis com os tipos dos registros de interfaces e, consequentemente, com os tipos esperados pelos importadores da interface.

Caso um módulo B precise incluir o módulo A no seu

contexto, o acesso ao arquivo objeto de A deverá estar disponível ao compilador quando o módulo B for compilado.

Para isto, é utilizada a declaração DIRECTORY antes da declaração do módulo que vai incluir outros módulos. No caso do exemplo citado acima, aparecerá a declaração DIRECTORY antes da declaração do módulo B.

Além de listar os módulos que serão incluídos, a declaração DIRECTORY permite restringir o acesso a alguns símbolos. Isto é realizado através da cláusula USING.

Em Mesa, declara-se o nome dos registros de interfaces através da declaração IMPORTS.

Existe porém, uma diferença entre registros de interfaces e tipos de interfaces. Um módulo programa quando utiliza um tipo de interface, acessa somente elementos do tipo não interface. Porém, utilizando-se registros de interfaces, é possível acessar todos os elementos, tanto os do tipo interface quanto os do tipo não interface.

A sintaxe da declaração IMPORTS é a seguinte:

IMPORTS < nome de registros de interfaces > : < nome do tipo de interface >

A Figura (IV.1) ilustra um exemplo de utilização da declaração IMPORTS.

```
DIRECTORY Defs1 : FROM "defs1", Defs2 : FROM "defs2";
Prog : PROGRAM IMPORTS IRec : Defs1, I2Rec : Defs =
     BEGIN .. END
```

FIGURA IV.1 - Exemplo da Utilização da Declaração IMPORTS num Programa Escrito em Mesa

Dentro do corpo do programa Prog, uma referência do tipo Defs1.X, será válida somente se for a um elemento de Defs1, do tipo não interface. Entretanto, IRec.X pode se

referir a qualquer elemento de Defs1, seja ele do tipo interface ou do tipo não-interface. Esta distinção faz-se necessária porque uma chamada a uma rotina proc, definida em Defs1, deve referir-se ao descritor real do registro de interface IRec, em tempo de execução, e não somente a sua definição estabelecida em tempo de compilação.

Caso o nome do registro de interface seja omitido na declaração IMPORTS, significa que ele é igual ao nome do tipo da interface que aparece na declaração.

Quando for necessário acessar mais de uma instância de um registro de interface, define-se vários registros de interfaces para o mesmo tipo de interface. A Figura (IV.2) ilustra um exemplo deste fato.

```
DIRECTORY SymDefs : FROM "SymDefs"" :
PartofCompiler : PROGRAM IMPORTS mainSym : SymDefs,
                                auxSym : SymDefs =
BEGIN .. END
```

FIGURA IV.2 - Exemplo da Definição de Vários Registros de Interfaces para o Mesmo Tipo de Interface

Um módulo do tipo programa também pode ser importado de maneira similar à importação de registros de interfaces.

Em Mesa, é possível que a implementação de uma interface seja realizada por mais de um módulo, sendo necessário uma interação entre estes módulos. Neste caso, é comum que cada um dos módulos envolvidos na implementação use elementos de interfaces providos pelos outros módulos. Isto é realizado através da importação e da exportação da mesma interface.

Para ligar os vários módulos que formam o sistema, utiliza-se uma linguagem de configuração separada denominada C/MESA.

Na verdade, escrever um programa na linguagem de configuração C/MESA, nada mais é do que associar os registros de interfaces com as interfaces exportadas.

A linguagem de configuração C/MESA possui uma sintaxe similar à linguagem de programação MESA. O código fonte de um programa escrito em C/MESA é denominado de descrição de configuração (CD), e sua compilação resulta um arquivo denominado de descrição binária de configuração (BCD). Uma vez criado o arquivo BCD, ele pode ser carregado e executado.

A Figura (IV.3) ilustra um CD descrevendo um sistema composto por três módulos : Copier, IOPkg e Driver

```
MakeCopierSystem : CONFIGURATION
    Control Driver =
BEGIN
    Copier,
    IOPkg,
    Driver,
END
```

FIGURA IV.3 - Primeiro Exemplo de uma Definição de Configuração Escrita em C/MESA

A configuração ilustrada pela Figura (IV.3), especifica que os módulos objeto de Copier, IOPkg e Driver devem ser ligados. Em seguida será gerado o arquivo BCD correspondente a MakeCopierSystem.

O carregamento de um arquivo BCD é dividido em duas etapas. A primeira cria uma instância da configuração alocando uma área para cada módulo que compõe o BCD. Nesta área existe um espaço para as variáveis estáticas do módulo (aquelas declaradas no corpo principal do módulo), e também um espaço extra para as informações utilizadas pelo sistema. Os procedimentos importados e as variáveis são acessados através de encadeamentos. O espaço necessário para estes encadeamentos ou é alocado nesta área reservada para o

módulo, ou dentro do código do módulo. A segunda parte do carregamento completa a ligação, preenchendo os encadeamentos de cada instância de módulo pertencente a uma instância de configuração.

Uma vez carregada a configuração, cada instância de módulo tem todas as suas interfaces ligadas. Entretanto, até este momento nenhum código foi executado, sendo necessário, portanto, inicializar as variáveis globais e estáticas, como também dar início à execução do código principal das instâncias. Isto é realizado através da declaração START.

Nem sempre as configurações são auto-contidas, isto é, nem sempre elas possuem dentro de seu contexto, a declaração de todos os módulos que importam e exportam as interfaces necessárias para realizar as ligações.

A Figura (IV.4) ilustra uma configuração auto-contida.

As informações contidas dentro dos parênteses não fazem parte da configuração e servem apenas para facilitar a explicação que sucede a Figura (IV.4).

```
Config : CONFIGURATION
   CONTROL LexiconClient =

BEGIN
   Fsp     ; (exporta a interface SystemDefs)
   IOPkg   ; (exporta a interface IODefs)
   Shings  ; (exporta a interface StringsDefs)
   Lexicon ; (importa as interfaces SystemDefs,  IODefs e
             StringDefs)
   LexiconClient; (importa as interfaces IODefs e Lexicon
                   Defs)
END.
```

FIGURA IV.4 - Exemplo de uma Configuração Auto-Contida, Escrita em C/MESA

A configuração que acabou de ser descrita é completamente auto-contida, uma vez que para cada interface importada existe, dentro da própria configuração, um

componente exportando esta interface, como podemos observar pelos comentários contidos dentro dos parênteses.

Algumas configurações podem, entretanto, importar interfaces que foram exportadas por outros módulos ou por outras configurações. Por exemplo, as interfaces SystemDefs, IODefs e StringDefs utilizadas na configuração ilustrada pela Figura IV.4, poderiam ter sido implementadas por outra configuração. Neste caso, ao invés de incluir instâncias destas interfaces na configuração, bastaria importá-las. A Figura (IV.5) ilustra esta alternativa.

```
Config : CONFIGURATION
   IMPORTS SystemDefs, IODefs, StringDefs
   CONTROL LexiconClient =
BEGIN
   Lexicon ;
   LexiconClient ;
END
```

FIGURA IV.5 - Exemplo de uma Definição de Configuração que utiliza a Declaração IMPORTS

A declaração IMPORTS utilizada dentro de uma configuração satisfaz os mesmos propósitos de quando ela é utilizada dentro de um módulo programa. A regra para importação é a seguinte: se um dos componentes da configuração importa alguma interface e esta interface não é exportada por nenhum outro componente da configuração, então esta interface deve ser importada explicitamente.

Já no que diz respeito à exportação a regra é a seguinte: se um componente exporta uma interface, esta interface pode ou não ser exportada em outro nível da configuração. Esta importante facilidade que C/MESA oferece de se poder exportar ou não, possibilita um controle maior daquilo que se deseja esconder ou tornar visível para o ambiente externo.

Em C/MESA é possível nomear os registros de

interfaces que aparecem na declaração IMPORTS, da mesma forma que em MESA. Estes nomes serão utilizados para prover as interfaces necessárias às instâncias dos componentes da configuração.

Para que torne-se mais claro, vamos exemplificar. Suponha que quiséssemos criar três instâncias, uma de cada tipo de interface que aparece na declaração IMPORTS do exemplo da Figura (IV.5). Teríamos, então, que substituir a linha que contém a declaração IMPORTS, pela seguinte:

```
IMPORTS alloc : SystemDefs, io : IODefs, str : StringDEFS
```

Consideremos que é o módulo Lexicon que importa tais interfaces. A associação entre os tipos de interfaces importadas por Lexicon, e as instâncias dos tipos especificadas na declaração IMPORTS, pode ser realizada da seguinte maneira:

```
Lexicon [ alloc, io, str];
```

As interfaces, no entanto, devem corresponder em ordem e em tipo com as interfaces que foram declaradas no módulo de programa Lexicon através da declaração IMPORTS.

Pode-se também, dar um nome para cada instância de um componente da configuração. Este nome deve ser precedido por ":" e ele vem declarado antes da instância. Esta facilidade pode ser exemplificada da seguinte maneira:

```
alex : Lexicon [ alloc, io, str] ;
```

Com esta facilidade é possível distinguir várias instâncias de um mesmo tipo de componente.

Outra facilidade fornecida por C/MESA é a de poder-se nomear um registro de uma interfce exportada. Consideremos o seguinte exemplo:

```
lexRec : LexiconDefs ←— alex : Lexicon [ alloc, io, str ]
```

Esta declaração cria uma instância denominada lexRec do tipo de interface LexiconDefs, a qual é exportada pela instância alex do tipo Lexicon. As importações definidas pelo tipo Lexicon aparecem entre os colchetes, representadas pelas instâncias.

A Figura (IV.5) poderia ser substituída pela Figura (IV.6), onde nenhuma facilidade implícita ("default") na linguagem C/MESA é utilizada.

```
Config : CONFIGURATION
  IMPORTS alloc : SystemDefs, io : IODefs,str : StringDefs
  CONTROL LexiconClient =
BEGIN
  lexRec : LexiconDefs←alex : Lexicon [ alloc, io, str];
  lexclient : LexiconClient  [io, lexRec];
END.
```

FIGURA IV.6 - Exemplo de uma definição de Configuração que não utiliza Nenhuma Facilidade Implícita

Neste exemplo temos a definição de uma instância denominada lexClient do tipo LexiconClient.

O tipo LexiconClient importa uma interface do tipo IODefs e também o módulo Lexicon, sendo que as instâncias io e lexRec representam a associação entre os tipos de interfaces importadas e as instâncias que foram especificadas. A instância io do tipo IODefs foi especificada na declaração IMPORTS, enquanto que a instância alex do tipo Lexicon exporta uma interface denominada lexRec do tipo LexiconDefs.

Como podemos observar, esta configuração cria tanto instâncias de interfaces (io, alloc e str), quanto instâncias de módulos do tipo programa (alex, lexClient).

É possível, como já foi dito anteriormente, que a

implementação de uma interface seja realizada por mais de um módulo.

Analisaremos agora, a exportação de uma interface que não é implementada por um único módulo.

Vamos assumir que o módulo Lexicon é dividido em dois módulos, LexiconFA e LexiconP, com LexiconFA provendo os procedimentos FindString e AddString, enquanto que no módulo LexiconP está o procedimento PrintLexicon. Cada um destes dois módulos exporta a interface LexiconDefs, no entanto, nenhum deles implementa por completo esta interface:

Tomemos como exemplo a seguinte configuração definida pela Figura (IV.7).

```
Config : CONFIGURATION
   IMPORTS SystemDefs, IODefs, StringDefs
   CONTROL LexiconClient =
BEGIN
   lexRec : LexiconDefs ← LexiconFA [ ];
   lexRec :   LexiconP [ ];
   LexiconClient  [IODefs, lexRec];
END.
```

FIGURA IV.7 - Último Exemplo de Uma Definição de Configuração Utilizando C/MESA

A instância LexiconClient vê uma única interface. Primeiramente cria-se uma instância denominada lexRec do tipo LexiconDefs, atribuindo-se a ela a exportação realizada por LexiconFA. Os colchetes em branco indicam que a instância lexRec utiliza, por "default", as interfaces importadas por LexiconFA. Na linha seguinte, a interface exportada por LexiconP é atribuída à instância lexRec. Desta forma, realiza-se a união das interfaces exportadas pelos dois módulos responsáveis pela implementação.

Para finalizar, em C/MESA também é possível

definir-se configurações dentro de outras configurações, criando-se assim, diferentes níveis de hierarquia.

As configurações aninhadas podem ser utilizadas para esconder algumas interface exportadas pelos componentes da configuração.

Porém, no que refere-se à importação, as configurações aninhadas devem insetir na sua lista de importação (definida pela declaração IMPORTS) todas as interfaces que seus componentes importam, mesmo que elas não sejam casadas dentro desta configuração. Em outras palavras, as interfaces nunca são importadas automaticamente dentro de uma configuração aninhada e deve-se propagar as importações sempre que necessário.

As regras de escopo permitem que as configurações aninhadas acessam tanto interfaces como outras configurações (também aninhadas) definidas nos demais níveis de hierarquia.

Todos os exemplos até aqui apresentados, foram retirados de |23|, e maiores detalhes podem ser encontrados nesta referência.

Passaremos agora a descrever algumas diferenças entre a linguagem C/MESA e a nossa proposta.

A primeira diferença a ser ressaltada é o fato de tanto MESA quanto C/MESA não terem sido projetadas para programação distribuída, não possuindo com isto, uma definição de entidades físicas nem lógicas para a localização dos módulos.

Outra diferença encontrada é a maior flexibilidade fornecida por MESA no que se refere à implementação de uma interface. Em MESA, além de um módulo poder implementar mais

de uma interface, esta implementação pode ser realizada completamente ou parcialmente. Neste último caso, ela será implementada por mais de um módulo.

Em Modula-2, como não existe o conceito de tipo de módulo, foi necessário utilizar um artifício para ser possível criar vários módulos de implementação para um mesmo tipo de interface. Em MESA isto já não foi preciso, uma vez que o conceito de tipo de módulo está embutido na linguagem.

Na fase de configuração são realizadas as correspondências entre as interfaces e os módulos de implementação.

Em C/MESA porém, esta correspondência pode ser realizada de forma parametrizada. Com isto, no mesmo momento em que está sendo criada uma instância, está sendo definida, através dos parâmetros, suas ligações com outros módulos.

No nosso caso, a função de criação de uma instância é realizada através da declaração CREATE, enquanto que a função de ligação é definida pela declaração LINK.

Logo, neste sentido, a nossa linguagem torna-se mais flexível, fornecendo recursos que facilitam a sua extensão para dar suporte à configuração dinâmica, onde um dos requisitos desejáveis é a definição separada das funções de configuração.

Por outro lado, tanto a nossa linguagem quanto C/MESA, utilizam, de forma similar, declarações para definição de importações e exportações de interfaces.

A declaração USE definida na nossa linguagem pode ser considerada análoga à declaração DIRECTORY de C/MESA. Estas declarações são utilizadas para definir o contexto dos módulos.

A conclusão que chegamos é que apesar de serem poderosas, MESA e C/MESA são linguagens complexas, pois englobam muitos conceitos. O usuário destas linguagens, embora possua uma gama enorme de facilidades, tem que saber aplicá-las corretamente, uma vez que o uso inadequado destas ferramentas acarretará erros nem sempre fáceis de serem detectados. Por exemplo, é importante que o usuário conheça bem a diferença existente entre os registros de interfaces e os tipos de interfaces, pois caso contrário, alguns problemas poderão ocorrer quando existirem várias instâncias de componentes.

IV.3 MÉTODO MASCOT3

No início de 1970, uma experiência prática obtida no desenvolvimento de um sistema em tempo real, para processamento de dados distribuídos, originou um método denominado Mascot.

Este método aplica-se ao projeto, à construção e à execução de software para tempo real.

A versão original de Mascot, agora conhecida como Mascot1 foi rapidamente superada por uma outra versão denominada Mascot2.

No entanto, algumas críticas foram apresentadas, motivando o desenvolvimento de uma nova versão denominada Mascot 3.

Primeiramente vamos apresentar a característica básica do método Mascot, para depois então, citar os novos conceitos introduzidos por Mascot3.

Mascot é caracterizado por ser um método que utiliza uma rede gráfica de fluxo de dados como o meio de expressar a estrutura do software. Combinando esta rede gráfica com um método de desenvolvimento sistemático, é garantido que a

estrutura do software estará refletida com exatidão no software operacional resultante. Os diagramas de fluxo de dados provêm visibilidade do projeto, e oferecem uma visão do software, cuja validade é protegida por um esquema rígido de criação e destruição dinâmica de processos em tempo de execução.

Mascot3 foi desenvolvido para ser independente da linguagem de programação utilizada na implementação. O método provê duas notações para expressar um projeto: a notação gráfica e a notação textual (linguagem), sendo que elas podem ser derivadas uma a partir da outra.

A concorrência é uma das principais características de Mascot3. De uma forma geral, um projeto define, de maneira hierárquica, um conjunto de processos para serem executados concorrentemente.

Nos níveis mais altos de hierarquia, os caminhos paralelos de execução são agrupados, e juntos formam as unidades denominadas subsistemas. Através de expansões progressivas dos subsistemas, os caminhos de execução são divididos em grupos cada vez menores, até atingir os níveis mais baixos de hierarquia, onde são encontrados os caminhos individuais, denominados atividades. As atividades são executadas num ambiente padrão, provido por um conjunto de software, denominado software de contexto. A interface entre o software de contexto e a aplicação é denominada interface de contexto e é, geralmente, expressa de forma compatível com o estilo dos módulos de sofware da aplicação.

A natureza hierárquica permite que o projeto seja visto sob vários níveis de abstração. Cada nível de hierarquia é representado por uma rede, através da qual os dados são transmitidos de uma entidade ativa (subsistema ou atividade) para outra entidade.

Os dispositivos de hardware, que podem ser considerados como fontes de informação de nível mais baixo, não fazem parte do método Mascot3. Porém, faz-se necessário existir um meio de comunicação com tais dispositivos. Logo, Mascot3 definiu para este propósito elementos de software denominados servidores.

Para que os processos executados assincronamente possam trocar informações de uma maneira segura, faz-se necessário prover mecanismos que efetuem exclusão mútua no uso dos dados transferidos de ou para áreas de endereçamento comum. Uma falha nestes mecanismos pode conduzir a informações corruptas, ou até mesmo ao "deadlock".

Mascot3 definiu áreas de inter-comunicação de dados, conhecidas como IDA, através das quais os subsistemas sincronizam-se e comunicam-se.

Vamos agora, através de um exemplo mostrado em |3|, apresentar, primeiramente, a notação gráfica oferecida por Mascot3, para em seguida, representar este mesmo exemplo utilizando-se a notação textual.

A Figura (IV.8) representa o nível mais alto de hierarquia do sistema que será tomado como exemplo.

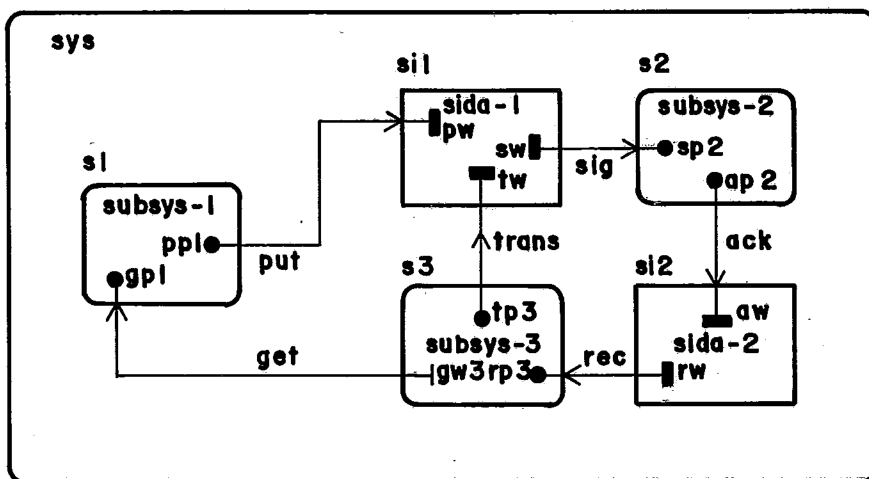


FIGURA IV.8 - Exemplo do Sistema Sys

O elemento do projeto ilustrado pela Figura (IV.8) é o sistema Sys, que está representado por um retângulo de bordas arredondadas que abrange todo o diagrama. Este sistema consiste de cinco elementos que comunicam-se entre si, dentre os quais, três, da mesma forma que o sistema, estão simbolizados por retângulos de bordas arredondadas, e representam os subsistemas. Logo, os três componentes s1, s2 e s3 são executados paralelamente.

Os outros dois elementos si1 e si2, representam as áreas de inter-comunicação de dados (IDA), as quais são, geralmente, simbolizadas através de retângulos.

Na notação gráfica do método Mascot3, está convencionado que bordas arredondadas indicam entidades ativas.

Podemos observar então, que existe uma separação nítida entre as atividades ativas e passivas.

As setas que ligam os subsistemas com as áreas de inter-comunicação de dados, representam o fluxo de dados. Estas linhas são chamadas de caminhos ("paths").

Na Figura (IV.8) podemos observar que os dados fluem do subsistema s2 para o subsistema s3, através de dois caminhos rotulados, respectivamente, de ack e rec. Os dados entram na IDA si2, pelo caminho ack, e saem dela pelo caminho rec.

Já na IDA si1, entram dados pelos caminhos put e trans, provenientes, respectivamente, dos subsistemas s1 e s2. Os dados que saem da IDA si1 pelo caminho seg, são direcionados para o subsistema s2.

A partir de agora vamos decompor os elementos do sistema sys.

A Figura (IV.9) ilustra com mais detalhes a composição interna do subsistema s3.

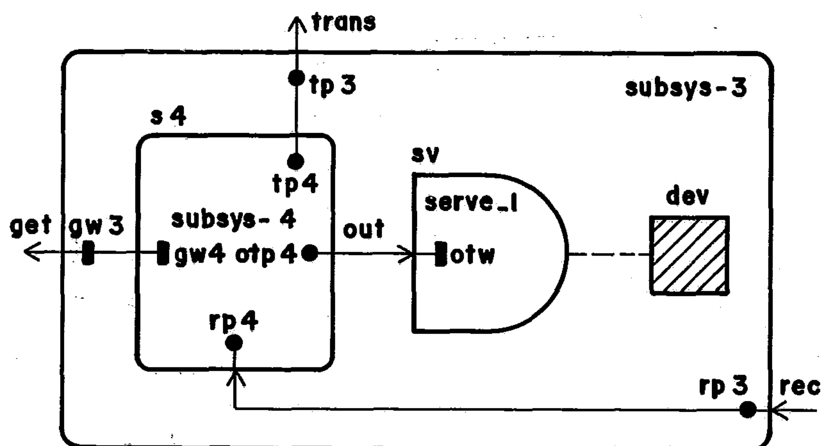


FIGURA IV.9 - Representação do Subsistema s3.

O subsistema s3 está composto por dois componentes, s4 e s5.

O componente s4 é um subsistema, no qual estão conectados os três caminhos que apareceram na ilustração do subsistema s3 (Figura IV.9): get, trans e rec, sendo os dois primeiros de fluxo de saída, enquanto o último é de entrada.

Já o componente sv, ilustrado por um símbolo na forma da letra D, representa um elemento servidor, que, como já foi mencionado anteriormente, é utilizado na comunicação com um dispositivo.

O dispositivo está representado na Figura (IV.9) por um pequeno retângulo hachurado, o qual está ligado ao servidor através de uma linha particionada.

Descendo agora mais um nível, podemos ilustrar o subsistema s4 com mais detalhes, através da Figura (IV.10).

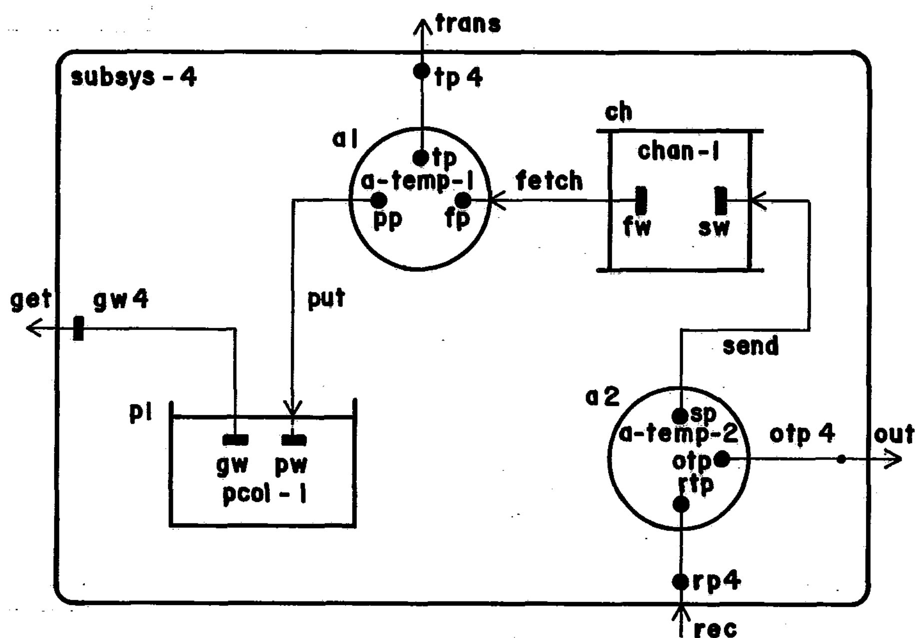


FIGURA IV.10 - Representação do Subsistema s4.

Atingimos então, o nível mais baixo de hierarquia do sistema, onde não é possível mais decompor os elementos.

O subsistema s4 está composto pelos elementos mais básicos, que são as atividades, as quais estão representadas por círculos denominados a1 e a2. As atividades, como já foi mencionado, são os elementos fundamentais de execução. Cada atividade implementa uma execução separada, para ser processada paralelamente com as demais atividades. Qualquer decomposição de uma atividade será realizada em termos de passos de execução sequencial, e não será mais uma decomposição a nível de rede.

Os outros elementos que aparecem na Figura (IV.10) são áreas de inter-comunicações de dados representando operações específicas. O retângulo que possui dois lados opostos sobressalentes é denominado canal ("CHANNEL"), enquanto o outro retângulo é denominado reservatório ("POOL").

O canal é caracterizado pela operação de leitura destrutiva; isto significa que os dados que passam por ele,

são acomodados temporariamente numa memória interna. Esta memória pode ficar completamente cheia, caso ocorram várias operações sucessivas de escrita, ou pode ficar vazia, como consequência de várias operações repetidas de leitura.

Já os reservatórios são caracterizados por operações de escrita destrutiva. Seu conteúdo consiste de uma coleção de variáveis, às quais são atribuídos valores iniciais quando o sistema começa a ser executado. Depois, estas variáveis podem ser consultadas e atualizadas.

Uma vez que já fornecemos uma visão geral da representação gráfica dos três níveis de hierarquia, vamos agora detalhar um pouco mais o modelo de comunicação definido pelo método Mascot3.

No subsistema s4 temos duas atividades, a1 e a2, que se comunicam através de um canal. Este caso pode, por exemplo, estar representando um produtor (a2) que esteja fornecendo informação a um consumidor (a1). O canal funciona como um depósito temporário ("buffer"), e pode conter dois procedimentos, um para adicionar um ítem no depósito e outro para retirar um item do depósito. Estes procedimentos, chamados de procedimentos de acesso, são encapsulados pela área de inter-comunicação de dados (IDA), e estão seletivamente disponíveis para as atividades através do conceito de janelas. As janelas da Figura (IV.10) e representadas por pequeninos retângulos negritados, estão rotuladas por fw e sw.

Uma janela de uma IDA exporta um conjunto de interações. A natureza das interações providas por uma determinada janela é definida pelo tipo do caminho que esta conectado nesta janela. Este tipo é indicado no diagrama por um identificador que rotula o caminho. Logo, as janelas <u>s</u>w e <u>f</u>w estão respectivamente conectadas a caminhos do tipo <u>s</u>end e <u>f</u>etch.

O tipo de um caminho é definido por um módulo denominado ACESS INTERFACE, que será explicado mais adiante.

Os caminhos ligam as janelas às portas, as quais são representadas, através de pequeninos círculos negritados e estão rotuladas na Figura (IV.10), por sp e fp.

Enquanto as janelas estão relacionadas a entidades passivas (canais e reservatórios), as portas estão associadas a entidades ativas (atividades).

Devemos observar que os dados podem fluir de uma janela para uma porta, como também no sentido inverso, ou seja, de uma porta para uma janela. Podemos ter também, um fluxo de dados num único caminho, de forma bidirecional. O sentido do fluxo de dados é indicado pela seta do caminho.

Agora representaremos este mesmo exemplo que foi mostrado na notação gráfica, na notação textual. Porém, partiremos do nível mais baixo para o mais alto, de forma inversa ao que foi feito na notação gráfica.

Começaremos pelas definições dos tipos de caminho: send e fetch. A Figura (IV.11) ilustra os dois módulos ACCESS INTERFACE.

```
ACCESS INTERFACE send;
   WITH flow-data;
   PROCEDURE insert (item:flow-data);
END.

ACCESS INTERFACE fetch;
   WITH flow-data;
   PROCEDURE extract (VAR item : flow-data);
END.
```

FIGURA IV.11 - Definição das Interfaces send e fetch

Os módulos ACCESS INTERFACE contêm informações suficientes para permitir que o conjunto de interações correspondentes sejam acessadas. Eles incluem o nome dos

procedimentos e definições indiretas dos tipos de dados que aparecem como parâmetros destes procedimentos. As definições indiretas são realizadas através da declaração WITH.

A declaração real de um tipo de dado é realizada através de uma especificação denominada DEFINITION. A Figura (IV.12) mostra a definição do tipo flow-data.

```
DEFINITION flow-data;
  TYPE
       flow-data = RECORD
                      .
                      .
                   END;
END.
```

FIGURA IV.12 - Definição do Tipo flow-data

Devemos ressaltar que as interfaces podem conter mais do que um procedimento. Neste exemplo específico, ilustrado pela Figura (IV.11), as interfaces possuem somente um procedimento, contudo, elas não tiveram seus nomes coincidindo com os nomes dos procedimentos, mesmo estes sendo os únicos elementos da interface.

A definição do canal chan-1 do subsistema s4 é apresentada pela Figura (IV.13).

```
CHANNEL chan-1;
     PROVIDES sw:send;
              fw:fetch;

ACCESS PROCEDURE insert (item:flow-data);
      .
      .
END.
ACCESS PROCEDURE remove (VAR item:flow-data);
      .
      .
END.

fw.extract = remove
END
```

FIGURA IV.13 - Definição do Canal chan-1

A declaração PROVIDES lista todas as janelas de uma IDA, fornecendo a cada uma delas, um nome e um tipo, o qual está relacionado com um módulo ACCESS INTERFACE.

Os procedimentos que implementam as interações especificadas pelas interfaces, são identificados pela declaração ACCESS. Como podemos observar através da Figura (IV.13), o nome do procedimento não tem que ser, necessariamente, igual ao nome da interface. Neste exemplo, a correspondência entre o procedimento insert e a sua interface é realizada implicitamente, uma vez que ambos possuem o mesmo nome. Já o procedimento remove tem que ser associado explicitamente a sua interface extract, através de uma declaração de equivalência definida no final do módulo.

As portas são os meios pelos quais uma atividade requisita uma interação especificada por uma interface.

Para que uma ligação estabelecida entre uma janela e uma porta seja válida, ambas devem referir-se a uma mesma interface.

Através da Figura (IV.14), apresentamos a definição das atividades que utilizam as interfaces que foram declaradas na Figura (IV.11).

Cada atividade lista, através da declaração REQUIRES, as portas utilizadas, especificando também as interfaces necessárias, cujos procedimentos associados serão chamados.

Uma vez definidas as interfaces, as atividades, os canais e os reservatórios, vamos passar para a definição do subsistema s4, ilustrado pela Figura (IV.15).

```
ACTIVITY a temp-2;
    REQUIRES sp:send; otp:out; rp:rec;
    VAR
        val:flow-data;
    BEGIN
      .
      .
    sp.insert(val);
    END.
END.

ACTIVITY a-temp-1;
    REQUIRES fp:fetch: tp:trans; pp:put;
    VAR
        next:flow-data;
    BEGIN
      .
      .
    fp.extract(next);
    END
END.
```

FIGURA IV.14 - Definição das Atividades a-temp-1 e a-temp-2

```
SUBSYSTEM subsys-4;
    PROVIDES gw4:get;
    REQUIRES rp4:rec;
             otp4:out;
             tp4:trans;
    USES pool-1, chan-1, a-temp-1, a-temp-2;
        POOL p1:pool-1;
        CHANNEL ch:chan-1;
        ACTIVITY a1:a-temp-1 (fp=ch.fw,
                              tp=tp4,
                              pp=p1.pw);
        ACTIVITY a2:a-temp-2 (sp=ch.sw,
                              otp=otp4,
                              rp=rp4);
   gw4=p1.gw
END.
```

FIGURA IV.15 - Definição do subsistema s4

A definição de um subsistema está dividido em duas partes: especificação e implementação. Na parte de especificação encontram-se as declarações de exportação (PROVIDES) e de importação (REQUIRES).

No início da parte de implementação, encontramos a

declaração USES, a qual se incube de definir todos os tipos de componentes que serão utilizados para criar as instâncias.

Logo em seguida vem a criação das instâncias, juntamente com as respectivas ligações. As ligações são listadas dentro dos parênteses, que sucedem o tipo da atividade, isto é, as ligações são realizadas de forma parametrizada.

Foram criadas uma instância de um reservatório, uma instância de canal e uma instância para cada tipo de atividade.

Os nomes das portas comportam-se da mesma forma que os parâmetros formais de um procedimento. Os parâmetros reais correspondentes especificam os pontos da rede com as quais cada porta será conectada.

No caso da atividade a1 temos as seguintes ligações:

a porta fp ligada com a janela fw do canal ch
a porta tp ligada com a porta tp4 do subsistema s4
a porta pp ligada com a janela pw do reservatório p1

A ligação de uma janela com outra janela é realizada através da declaração de equivalência. Logo, na última linha da definição do subsistema s4 encontramos a ligação da janela.

Passaremos então para o próximo nível de hierarquia, no qual está definido o subsistema s3, que é composto pelo subsistema s4 e por um servidor.

A definição do servidor é apresentada pela Figura (IV.16).

```
SERVES serve-1;
     PROVIDES otw:out;
END
```

FIGURA IV.16 - Definição de um Servidor

Um servidor parece-se muito com uma IDA, podendo-se apontar como principal diferença, a sua utilização para interação com periféricos, e não com atividades.

De forma análoga aos procedimentos de acesso (ACCESS PROCEDURE) definidos na IDA, existem procedimentos nos servidores, denominados "handlers". Estes procedimentos estão relacionados com as interrupções geradas pelo hardware (dispositivos). A função dos "handlers" é controlar tanto a transferência dos dados como a operação dos dispositivos. A transferência é especificada em termos de caminhos conectados às janelas que foram definidas no servidor. A definição do servidor, ilustrada na Figura (IV.16), especifica uma única janela, denominada otw, cujo tipo é out.

Já que definimos todos os componentes que formam o subsistema s3, podemos, através da Figura (IV.17), apresentar a sua definição completa.

```
SUBSYSTEM subsys-3;

   PROVIDES gw3:get;
   REQUIRES rp3:rec;
            tp3:trans;
   USES subsys-4,serve-1;
      SERVER sv:serve-1;
      SUBSYSTEM s4:subsys-4 (rp4 = rp3,
                             otp4 = sv.otw,
                             tp4 = tp3);
      gw3 = s4.gw4
  END.
```

FIGURA IV.17 - Definição do Subsistema s3

Observamos que na parte de especificação estão declaradas uma janela e duas portas, cujas respectivas

interfaces são: get, rec e trans.

Na parte de implentação foram criadas uma instância do servidor serve-1 e uma instância do subsistema s4.

As ligações definidas foram as seguintes:

a porta rp4 do subsistema s4 ligada com a porta rp3
a porta otp4 ligada com a janela otw do servidor sv
a porta tp4 do subsistema s4 ligada com a porta tp3

A ligação da janela gw4 do subsistema s4 com a janela gw3 está definida através da declaração de equivalência, que encontra-se no final da definição do subsistema s3.

Resta a definição do último nível de hierarquia, o sistema sys. Na Figura (IV.18) encontra-se a definição do sistema sys.

```
SYSTEM sys;
    USES subsys-1, subsys-2, subsys-3, sida-1, sida-2;
    IDA si1:sida-1;
    IDA si2:SIDA-2;

    SUBSYSTEM s1:subsys-1 (pp1=si1.pw,
                           gp1=si3.gw3);

    SUBSYSTEM s2:subsys-2 (sp2=si1.sw,
                           ap2=si2.aw);

    SUBSYSTEM s3:subsys-3 (tp3=si1.tw,
                           rp3=si2.rw);

END.
```

FIGURA IV.18 - Definição do Sistema sys

Podemos observar que existem alguns pontos que diferem a definição do sistema da definição dos subsistemas que o compõem.

A primeira diferença a ser ressaltada é que não existem dependências externas, ou seja, o sistema não especifica nem portas nem janelas. Outra diferença é que o sistema é definido pela palavra SYSTEM, e não existe mais a classificação das IDA em canais ("CHANNEL") e reservatórios ("POOL").

Vamos apresentar agora as representações gráfica e textual providas pelo método Mascot3 para a descrição de atividades.

Como já foi dito anteriormente, as atividades podem ser decompostas por sub-componentes sequenciais. Estes componentes comunicam-se, em tempo de execução, através de interfaces de procedimentos. A Figura (IV.19) ilustra a representação gráfica da atividade a-temp-1, enquanto que a Figura (IV.20) apresenta a representação textual.

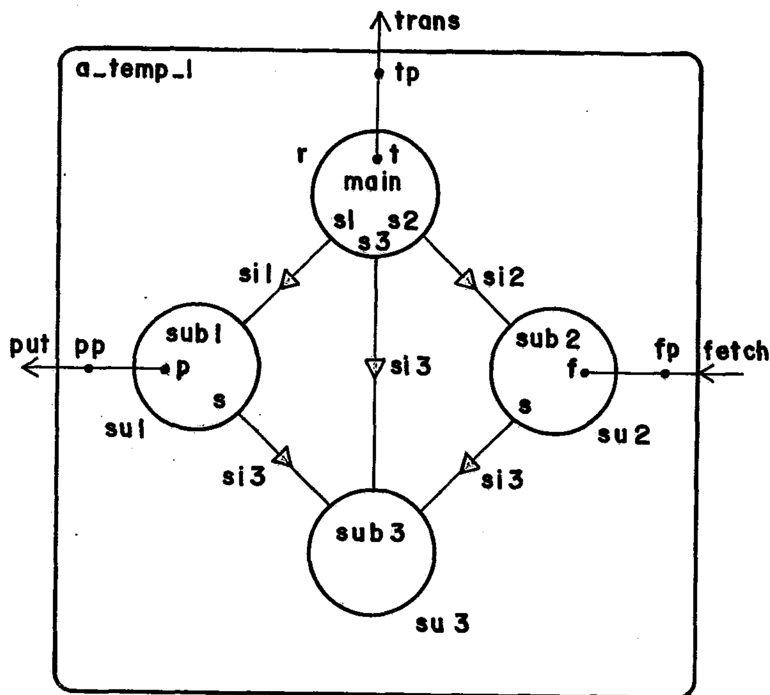


FIGURA IV.19 - Representação Gráfica da Atividade a-temp-1

```
ACTIVITY a-tem-1;
   REQUIRES fp:fetch;
            tp:trans;
            pp:put;

   USES main sub1, sub2, sub3;
      SUBROOT su3:sub3;
      SUBROOT su1;sub1 (p=pp,
                        s=su3);
      SUBROOT su2:sub2 (f=fp,
                        s=su3);
      ROOT   r:  main   (t=tp,
                       s1=su1,
                       s2=su2,
                       s3=su3;

END.
```

FIGURA IV.20 - Definição da Atividade a-temp-1

Podemos observar que existe um elemento principal denominado raiz (ROOT), enquanto que os demais componentes são denominados de sub-raiz (SUBROOT). Na raiz encontra-se o ponto inicial de execução.

A relação entre a raiz e as sub-raízes é representada graficamente através de setas não negritadas, como podemos observar na Figura (IV.19). Estas setas indicam que a raiz r chama diretamente procedimentos da sub-raiz su1 e da sub-raiz su2, e que estas últimas, por sua vez, chamam um procedimento da sub-raiz su3.

De forma análoga à definição de interfaces, realizada através da declaração ACCESS INTERFACE, os tipos de ligações que existem entre a raiz e as sub-raízes são especificados através da declaração SUBROOT INTERFACE. No conteúdo destes módulos encontramos a declaração dos procedimentos que a sub-raiz exporta para as demais sub-raízes e para a raiz principal.

Vamos passar agora para a definição do elemento raiz, o qual é estabelecido pela declaração ROOT, ilustrada pela Figura (IV.21).

```
ROOT main;
   REQUIRES t:trans;
   NEEDS s1:si1;
         s2:si2;
         s3:si3;

END
```

FIGURA IV.21 - Declaração do Elemento Raiz main.

A declaração NEEDS estabelece as ligações as quais são conectadas a componentes do tipo especificado. Estas ligações, como já foi mencionado, são realizadas em termos de interfaces das sub-raízes (SUBROOT INTERFACE).

A Figura (IV.22) apresenta a definição de uma das sub-raízes que compõem a atividade. Esta sub-raiz possui uma porta e duas ligações, sendo uma de entrada (si1) e a outra de saída (si3).

```
SUBROOT sub1;
   REQUIRES p:put;
   GIVES  si1;
   NEEDS s : si3;
END
```

FIGURA IV.22 - Declaração da Sub-Raiz sub1.

Através da declaração GIVES fica especificada a interface que uma sub-raiz está implementando.

Após termos apresentado como seriam as definições dos elementos que compõem um determinado sistema no método Mascot3, podemos concluir o seguinte: as ligações entre componentes distintos sempre são realizadas entre uma porta e uma janela, sendo permitido ligar várias portas a uma mesma janela, o que equivale dizer que várias entidades ativas podem usar a mesma interface de uma IDA. Outro ponto importante a ser destacado é que uma janela pode corresponder a vários procedimentos, e não necessariamente a um só. Neste caso, a declaração ACCESS INTERFACE conteria a

identificação de diversos procedimentos.

Queremos destacar ainda, que tanto subsistemas quanto IDAs podem ser ligados entre si diretamente. Para isto é necessário que os subsistemas possuam janelas para serem ligadas com portas do outro subsistema, e que a primeira IDA possua portas para serem ligadas com as janelas da segunda IDA.

Vamos passar agora a fazer algumas comparações entre o método Mascot3, que acabou de ser descrito, e a linguagem de configuração que foi proposta.

As principais diferenças que podem ser apontadas são: a representação gráfica disponível no método Mascot3 e a independência de uma linguagem de programação que este método apresenta.

O nosso projeto é totalmente dependente da linguagem Modula-2 e quanto à representação gráfica, esta pode ser vista como uma ferramenta auxiliar para a descrição de sistemas. No entanto, não é difícil definir uma representação gráfica para a nossa linguagem, caso isto seja considerado um requisito fundamental.

Analogamente à nossa proposta, Mascot3 possui uma estrutura hierárquica, porém deve ser ressaltado que este método emprega palavras chaves diferentes para cada nível, fato este que o torna mais complexo.

O método Mascot3 apresenta uma separação muito nítida entre os elementos ativos e passivos, dificultando um pouco a definição de tipos abstratos de dados. Na nossa linguagem, os módulos que implementam as interfaces de comunicação entre módulos, incluem tanto as estruturas de dados quanto as operações a serem executadas sobre elas.

Apesar da intercomunicação, no método Mascot3, ser também procedimental, igual a nossa linguagem, foram acrescentados os conceitos de portas e janelas. Com isto, a legibilidade diminuiu, uma vez que as ligações entre as portas e as janelas podem ter diferentes direções, e não representam a direção na qual a chamada de procedimento é realizada, tanto que é permitido definir caminhos até bidirecionais.

Outro ponto que deve ser ressaltado, é o fato do método Mascot3 não apresentar a separação entre a função de instanciação e de ligação. A função de ligação, por sua vez, é realizada de duas formas distintas: parametrizada e através de declarações explícitas de equivalência, o que dá margem a ocorrência de erros.

Não podemos deixar de reconhecer, no entanto, que o método Mascot3 apesar de complexo, apresenta um alto grau de flexibilidade.

CAPÍTULO V

A IMPLEMENTAÇÃO DE UM INTERPRETADOR PARA A NOSSA LINGUAGEM DE CONFIGURAÇÃO

A linguagem de configuração descrita no capítulo III servirá como base para o usuário estabelecer a configuração do sistema.

Entretanto, o programa escrito nesta linguagem precisa ser interpretado para que os erros ocorridos durante a sua elaboração sejam eliminados, e para obter-se também todas as informações necessárias para realizar o carregamento dos módulos nas estações físicas correspondentes.

O interpretador desenvolvido para a nossa linguagem de configuração foi implementado em Modula-2 (|18|, |25|), e será apresentado neste capítulo. As listagens do programa encontram-se no Anexo I.

V.1 AMBIENTE DE PROGRAMAÇÃO DISTRIBUÍDA

Como já foi dito anteriormente, o objetivo do desenvolvimento do projeto foi a construção de um ambiente de programação distribuída. A estrutura do ambiente poderá ser moldada de várias maneiras, dependendo da arquitetura física utilizada.

As configurações de sistemas distribuídos podem ser tratadas segundo dois modelos: de forma centralizada ou distribuída. No nosso caso específico ela será realizada de forma centralizada.

Num sistema de controle, por exemplo, geralmente os computadores utilizados são simples e sem memória auxiliar, não servindo para o desenvolvimento de software. Neste caso é necessário empregar o método que consiste em utilizar uma máquina hospedeira/alvo.

O software é desenvolvido na máquina hospedeira com o auxílio de um sistema operacional de propósito geral, o qual oferecerá o suporte necessário ao ambiente de programação, como por exemplo, editores, pacotes gráficos, banco de dados entre outras ferramentas. Depois de serem elaborados na máquina hospedeira, os programas são carregados nas respectivas máquinas alvo, podendo haver uma comunicação remota entre as máquinas alvo e a máquina hospedeira, para que esta última forneça algumas das facilidades contidas nela.

Alternativamente, podemos ter, de uma maneira menos centralizada, algumas funções específicas distribuídas entre as máquinas, de forma que estas funções possam ser compartilhadas pelas outras máquinas, ou então termos um conjunto mínimo de funções em cada máquina oferecendo uma maior autonomia a cada uma delas.

Através do esquema ilustrado pela Figura (V.1), são apresentadas as diversas etapas que antecedem à execução de um sistema no ambiente construído a partir da linguagem Modula-2 para programação distribuída.

V.2 ESTRUTURA GERAL DO INTERPRETADOR

O interpretador possui duas funções específicas: i) analisar a configuração, e a partir desta análise obter todos os dados que devem ser carregados nas estações físicas para ser realizada a execução do sistema; ii) montar as interfaces necessárias para os "stubs" do servidor e do cliente, como também para o núcleo de suporte de tempo de

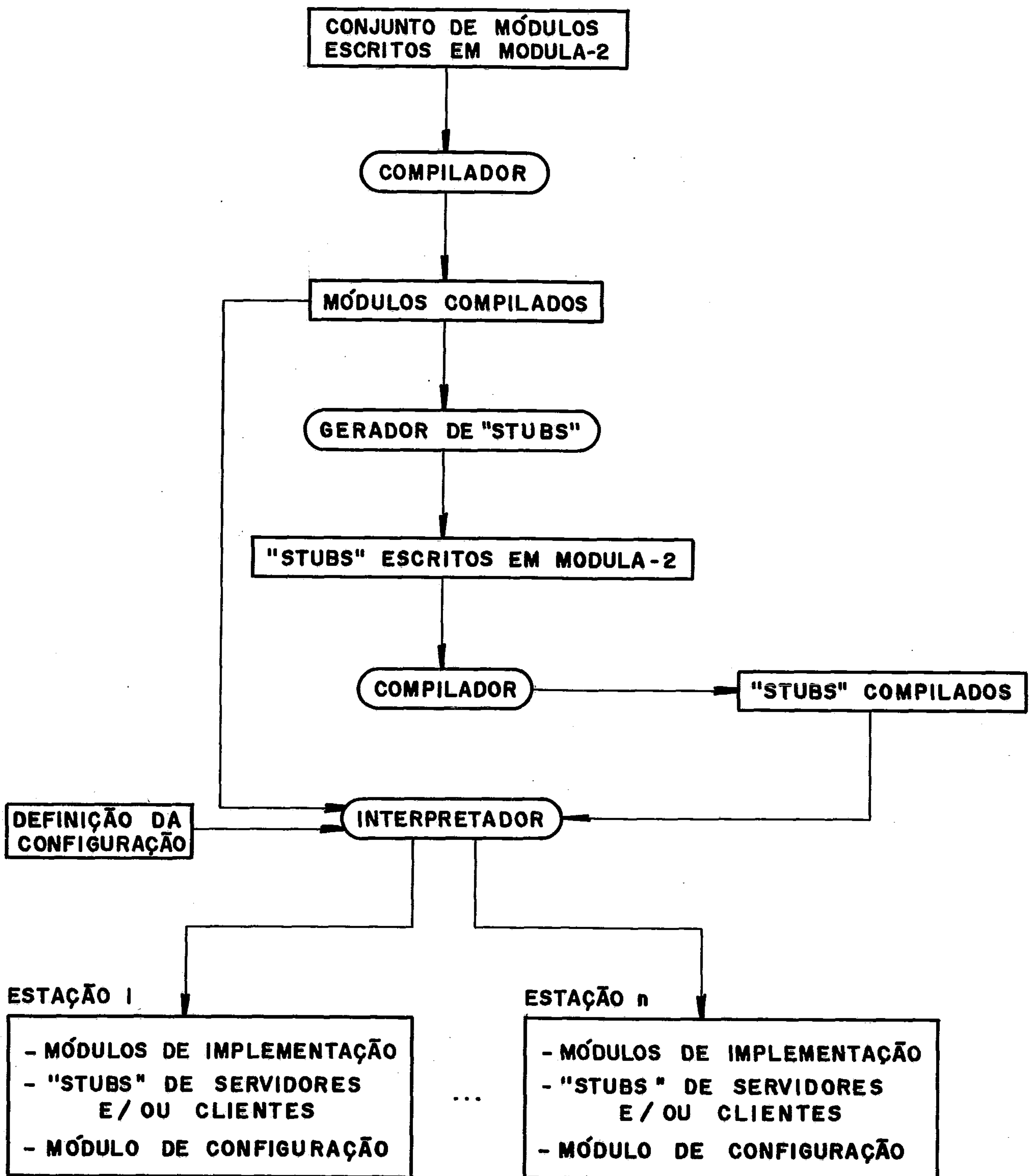


FIGURA V.1 - Esquema de Todas as Etapas que Antecedem à Execução do Sistema.

execução, utilizarem quando ocorrer uma chamada remota de procedimento.

Logo, como pode-se observar, o interpretador divide-se em duas partes. A primeira é processada antes da execução do sistema, numa máquina hospedeira, e a outra é para ser utilizada em tempo de execução, sendo que esta última é denominada módulo de configuração.

Faz-se necessário portanto, manter certas informações até o término da primeira parte (interpretação das declarações), e para isto são preenchidas algumas tabelas no decorrer da análise das declarações.

Somente ao final da interpretação de todas as declarações, e do carregamento nas estações físicas dos módulos de implementação, juntamente com os "stubs" de clientes e/ou servidores, como também uma cópia do módulo de configuração, é que o sistema se encontrará pronto para dar início a sua execução.

V.3 ESTRUTURAS DE DADOS UTILIZADAS PELO INTERPRETADOR

As estruturas de dados que serão apresentadas a seguir, são montadas à medida em que as declarações vão sendo analisadas. Estas estruturas servirão como base para identificar os módulos de implementação correspondentes a cada estação física.

Durante a interpretação são descobertas também quais são as ligações remotas, e por conseguinte obtem-se tanto a informação de quais são os "stubs" dos clientes e dos servidores que devem ser carregados, como também quais são as estações físicas as quais eles estão associados.

V.3.1 Tabela de Configurações

Cada entrada desta tabela corresponde a uma (sub-)configuração definida pelo usuário através da declaração CONFIGURATION.

Numa entrada estão contidas todas as informações correspondentes à (sub-) configuração. Estas informações são as seguintes:

1. o nome da (sub-) configuração.
2. os nomes das estações lógicas associadas às instâncias criadas.
3. os nomes das interfaces de tipos de módulos importadas.
4. os nomes das instâncias de implementação exportadas.
5. os nomes das instâncias de módulos de implementação ou de sub-configurações.
6. a quantidade total de estações lógicas que foram definidas.
7. a quantidade total de interfaces de tipos de módulos que são importadas.
8. a quantidade total de instâncias de implementação que são exportadas.

As cinco primeiras informações mencionadas são obtidas, respectivamente, no decorrer das declarações CONFIGURATION, STATIONS, IMPORT, EXPOR e CREATE.

Os dois últimos dados serão utilizados para simples verificação se a quantidade de interfaces de tipos de módulos importadas pela sub-configuração correspondente a esta entrada, casa com a soma total das instâncias de implementação que foram exportadas pelas sub-configurações que possuem alguma ligação com esta sub-configuração.

Deve ser ressaltado entretanto, que nem todas as instâncias de implementação que são exportadas por uma determinada sub-configuração, são necessariamente, importadas ao mesmo tempo por uma outra sub-configuração. Ou seja, cada uma das instâncias de implementação contidas na lista de exportação de uma sub-configuração, pode ser importada por uma sub-configuração diferente. Outra observação a ser feita é o fato de que a mesma instância de implementação pode ser importada por várias sub-configurações diferentes.

V.3.2 Tabela de Tipos

Esta tabela possui todos os tipos que foram utilizados por uma determinada (sub-) configuração para definir o seu contexto. Isto é, no caso de sub-configurações pertencentes ao primeiro nível de hierarquia, estarão declarados todos os tipos de módulos de implementação, e no caso das (sub-) configurações dos demais níveis, todos os tipos de sub-configurações.

Quando uma sub-configuração, pertencente a qualquer nível de hierarquia, declarar a importação de interfaces de tipos de módulos, os nomes destas interfaces também estarão contidos na tabela.

O campo que armazena os tipos citados acima, além de conter o nome dos tipos, possui também o nível de hierarquia, dentro do contexto geral da configuração, ao qual este tipo de (sub-) configuração pertence. Esta informação é obtida a partir da Tabela de Tipos de Configuração, que será descrita no próximo item.

Tratando-se de tipos de módulos de implementação, o campo referente ao nível de hierarquia será preenchido com o valor zero.

A Figura (V.2) ilustra os tipos que definem os campos da Tabela de Tipos.

```
TYPE PointerTipoConfig = POINTER TO TipoConfigType;

TYPE TipoConfigType = RECORD
                          NomeTipo : ArrayOfCharType ;
                          NivelHierarq : CARDINAL ;
                          ProxTipo : PointerTipoConfig ;
                        END (*record*);
```

FIGURA V.2 - Declaração dos Tipos Utilizados Para Definir os Campos da Tabela de Tipos.

V.3.3 Tabela de Tipos de Configuração

Esta tabela contém todos os tipos de (sub-)configuração criados com a declaração CONFIGURATION.

Através destes tipos são criadas as sub-configurações de maneira hierárquica.

Para cada tipo de (sub-) configuração é reservada uma entrada nesta tabela contendo as seguintes informações:

1. o nome da (sub-) configuração
2. o nível de hierarquia
3. lista das sub-configurações que compõem este tipo de (sub-) configuração

O segundo campo armazena o nível de hierarquia, dentro do contexto geral da configuração, ao qual a (sub-) configuração pertence. O nível de hierarquia é obtido a partir dos tipos utilizados pela (sub-) configuração os quais são mencionados através da declaração USE. Este campo tem como função distinguir as sub-configurações pertencentes ao primeiro nível de hierarquia, que pelo fato de serem compostas apenas por módulos de implementação, possuirão, durante a interpretação, um tratamento diferente daquele fornecido às (sub-) configurações pertencentes aos outros níveis de hierarquia.

Quando o campo referente ao nível de hierarquia contiver o maior valor, é o momento de preencher a Tabela de Estações Lógicas, que será descrita no próximo item.

O terceiro campo da tabela conterá uma lista com todas as sub-configurações que compõem a (sub-) configuração que está sendo analisada. Para cada uma das sub-configurações existirá uma outra lista, que por sua vez conterá as instâncias referentes a este tipo de sub-configuração. Estas instâncias vão sendo criadas através da declaração CREATE.

O tipo que define a Tabela de Tipos de Configuração está ilustrado pela Figura (V.3).

```
TYPE PointerInstancias = POINTER TO InstConfigType;
TYPE InstConfigType = RECORD
                        NomeInst : ARRAY [0..39] OF CHAR;
                        ProxInstConfig : PointerInstancias ;
                      END (*record*);

TYPE TabTipoConfigType=RECORD
                         NomeConfig : ARRAY [0..39] OF CHAR;
                         NivelHiearq : CARDINAL ;
                         InstanciasConfig:PointerInstancias;
                       END (*record*) ;
```

FIGURA V.3 - Declaração dos Tipos Utilizados para Definir os Campos da Tabela de Tipos de Configuração

V.3.4 Tabela de Estações Lógicas

A partir da interpretação da diretiva LOAD é possível realizar a associação existente entre as estações lógicas e as físicas.

Cada estação lógica será mapeada em apenas uma estação física, porém devemos ressaltar que podemos encontrar várias estações lógicas, associadas a uma mesma estação física. Uma (sub-) configuração pode possuir mais de

uma estação lógica, e estas por sua vez, não necessitam estar associadas a mesma estação física, logo, neste caso vamos ter partes (módulos de implementação) desta (sub-) configuração em estações físicas distintas.

Para que ao final da interpretação de todas as declarações seja possível carregar em cada estação física o conjunto dos módulos de implementação que lhe pertencem, utiliza-se uma tabela onde cada uma de suas entradas corresponde a uma estação lógica, e contém as seguintes informações:

- o nome da estação lógica
- o nome da estação física associada
- todas as instâncias das sub-configurações que são compostas ou por módulos de implementação ou por sub-configurações associadas à estação lógica referente a esta entrada.

A estrutura de cada entrada desta tabela pode ser melhor visualizada através da Figura (V.4).

1	nome da estação lógica
2	nome da estação física
3	apontador para o tipo I da tabela de configurações
4	lista das instâncias dos módulos de implementação
	⋮
3	apontador para o tipo i+n da tabela de configurações
4	lista das instâncias dos módulos de implementação

FIGURA V.4 - Entrada da Tabela de Estações Lógicas

Cada entrada desta tabela contém uma ou mais sub-configurações definidas pelo terceiro e quarto campos, os quais serão preenchidos somente quando for atingido o nível mais alto de configuração.

O campo referente ao nome da estação física (campo 2) será preenchido durante a interpretação da diretiva LOAD.

V.3.5 Tabelas Pertencentes ao Módulo de Configuração

As tabelas que serão apresentadas agora representam a interface com os "stubs", e sua utilização será mais bem detalhada na seção V.6.

V.3.5.1 Tabela de Importações

Esta tabela que será diferente para cada estação física, contém as informações ilustradas pela Figura (V.5) e será identificada pelo primeiro campo.

Existirá uma entrada nesta tabela para cada instância de módulo que importa algum procedimento de um módulo armazenado num outro nó físico, ou seja, o segundo, terceiro e quarto campos correspondem a uma entrada da tabela, e serão repetidos tantas vezes quantos forem os módulos importadores de procedimentos remotos contidos na estação física especificada.

1	identificador da estação física
2	nome da instância do módulo que importa procedimentos
3	identificação da estação física que contém o módulo que exporta os procedimentos
4	identificação do índice da tabela de exportação que contém o endereço do despachante

FIGURA V.5 - Tabela de Importações

V.3.5.2 Tabela de Exportações

Em cada estação física existirá uma Tabela de Exportações, ilustrada pela Figura (V.6). Esta tabela comtém uma entrada, composta pelo segundo e terceiro campos, para cada módulo exportador, associando a ele o endereço de seu despachante correspondente, isto é, o endereço do módulo que possui os procedimentos que são exportados pelo módulo exportador.

O segundo e terceiro campos serão repetidos tantas vezes quantos forem os módulos exportadores na estação física especificada através do primeiro campo da tabela.

As entradas desta tabela correspondem somente aos módulos cujos procedimentos exportados serão importados por módulos alocados em outras estações físicas, ou seja, para chamadas remotas de procedimento.

1	identificador da estação física
2	nome da instância do módulo exportador
3	endereço do despachante

FIGURA V.6 - Tabela de Exportações

V.3.6 Observações

A partir das Tabelas de Configurações e de Estações Logícas, são montadas as Tabelas de Importação correspondentes a cada estação física, e também são reservadas as entradas nas Tabelas de Exportações, cujo campo, que conterá o endereço do despachante referente ao módulo exportado, será preenchido posteriormente na fase de

inicialização dos "stubs" dos servidores.

Depois de montar-se as Tabelas de Importações e de Exportações, não são mais necessárias as informações das demais tabelas. Porém, isto só é válido para um ambiente de programação distribuída com configuração estática, porque no caso do ambiente dar suporte à configuração dinâmica, estas informações deverão ser mantidas para posteriores atualizações.

V.4 DESCRIÇÃO DO INTERPRETADOR

Como já foi mencionado, o interpretador está dividido em duas partes distintas, sendo que uma delas possui, entre outras funções, a responsabilidade de carregar e inicializar o sistema, enquanto que a outra será utilizada em tempo de execução. Nesta seção descreveremos estas duas partes contidas no interpretador, como também um resumo do procedimento realizado na análise de cada declaração.

V.4.1 Primeira Parte do Interpretador

Esta parte está subdividida em duas fases. Na primeira fase realiza-se a análise léxica e sintática das declarações. É nesta hora que verifica-se a correta utilização das declarações, isto é, se a lei de formação da linguagem (gramática) foi respeitada pelo usuário quando ele escreveu o programa para configurar o sistema.

A segunda fase desta primeira parte tem como objetivo interpretar a função de cada declaração, ou seja, realizar a análise semântica. Além disto, é nesta fase que ocorre a verificação da integridade da configuração, em outras palavras, todos os módulos que são ligados entre si devem estar coerentes com as exportações e importações previamente estabelecidas pelo programa de aplicação.

Após todo este procedimento de análise das declarações, descobre-se as ligações que são remotas e as que são locais. Juntando esta informação com os dados contidos nas tabelas descritas na seção anterior, é possível carregar nas estações físicas correspondentes os módulos de implementação, e também os "stubs" dos servidores e dos clientes referentes às ligações remotas que foram encontradas durante a interpretação da configuração.

Passaremos agora a descrever o procedimento realizado na análise de cada uma das declarações, porém antes será ilustrado pela Figura (V.7) o algoritmo geral da segunda fase.

Como podemos observar, o programa foi dividido em algumas rotinas as quais encarregam-se de realizar o tratamento específico de cada declaração.

Estas rotinas estão classificadas em dois grupos. O primeiro grupo encarrega-se de preencher as estruturas de dados, enquanto o segundo grupo utiliza as informações contidas nestas estruturas para analisar a consistência da configuração. Algumas rotinas do segundo grupo também preenchem campos das estruturas de dados.

As rotinas MontaConfig, AnalisaUse, MontaEstacoes, AnalisaImport, AnalisaExport, MontaModCriado, pertencem ao primeiro grupo. Já as rotinas AnalisaTipo, AnalisaEstacao, ModuloExportadoCriado, AnalisaLink, pertencem ao segundo grupo.

Algumas rotinas são autoexplicativas e portanto, não serão detalhadas aqui, como é o caso da rotina EliminaLinhaEmBranco.

Mesmo para as rotinas que serão apresentadas a seguir, não houve a preocupação de mostrar todos os detalhes da implementação, como por exemplo o tratamento de erros

```
INÍCIO
  EliminaLinhaEmBranco;
  SE não é fim de arquivo
  FAÇA
    (* Analisa a declaração CONFIGURATION *)
    Chama a rotina MontaConfig
    EliminaLinhaEmBranco;
    (* Analisa a declaração USE *)
    Chama a rotina AnalisaUse
    EliminaLinhaEmBranco;
    (* Analisa a declaração STATIONS *)
    Chama a rotina MontaEstàcoes
    EliminaLinhaEmBranco;
    SE existe importação
    FAÇA
       (* Analisa a declaração IMPORT *)
       Chama a rotina AnalisaImport
    FIM FAÇA
    EliminaLinhaEmBranco;
    SE existe exportação
    FAÇA
       (* Analisa a declaração EXPORT *)
       Chama a rotina AnalisaExport
    FIM FAÇA
    EliminaLinhaEmBranco;
    (* Analisa a declaração CREATE *)
    Chama a rotina MontaModCriados
    SE houve exportação
    FAÇA
       Verifica se as instâncias dos módulos de implementação
       que foram exportadas, foram criadas
    FIM FAÇA
    EliminaLinhaEmBranco;
    (* Analisa a declaração LINK *)
    Chama a rotina AnalisaLink
  FIM FAÇA
FIM.
```

Figura V.7 - Algoritmo Geral da Segunda Fase Pertencente à Primeira Parte do Interpretador

referentes a operações sobre os arquivos. A idéia principal é definir a função de cada uma das rotinas presentes no algoritmo da Figura (V.7), fornecendo uma visão global de como é realizada a interpretação de um programa escrito na nossa linguagem de configuração.

Maiores detalhes referentes à implementação podem ser encontrados consultando-se a listagem contida no Anexo I.

a) Rotina MontaConfig

Esta rotina tem como função analisar a declaração CONFIGURATION, alocando uma entrada, na Tabela de Configurações, para a (sub-) configuração que está sendo criada através desta declaração.

Além de reservar-se uma entrada na Tabela de Configurações, cria-se também uma entrada na Tabela de Tipos de Configuração.

Para ambas as tabelas é preenchido o campo que contém o nome da (sub-) configuração.

b) Rotina AnalisaUse

A função desta rotina é determinar o nível de hierarquia ao qual a sub-configuração pertence.

Para cada tipo definido pela declaração USE é verificado se ele está contido na Tabela de Tipos de Configuração. Se o tipo for encontrado na tabela, é sinal que a sub-configuração é composta por outra sub-configuração, e logo, ela não pertence ao primeiro nível de hierarquia, mas sim a um nível superior ao da sub-configuração que a compõe.

Este procedimento de pesquisar se o tipo declarado pertence ou não à Tabela de Tipos de Configuração será realizado para todos os tipos que foram declarados, chegando-se ao final à seguinte conclusão: Se não for encontrado, na Tabela de Tipos de Configuração, nem ao menos um entre todos os tipos declarados, a sub-configuração que está sendo analisada pertence ao primeiro nível de hierarquia, caso contrário, o nível de hierarquia será obtido acrescentando-se uma unidade ao maior nível de hierarquia entre todos os níveis de hierarquia encontrados durante a pesquisa.

A rotina preencherá também a Tabela de Tipos. Esta estrutura de dados será utilizada, posteriormente, pela rotina ModuloExportadoCriado.

c) Rotina MontaEstacoes

Esta rotina tem como função preencher o campo da Tabela de Configurações referente à lista de estações lógicas.

O campo que armazena a quantidade total de estações lógicas também é preenchido. Esta informação é utilizada para verificar-se, no caso de se fazer uso desta sub-configuração como um tipo na criação de instâncias, se estas instâncias estão sendo definidas em estações lógicas cuja quantidade total coincide com aquela contida nesta informação.

d) Rotina AnalisaImport

Com o intuito de preencher-se o campo da Tabela de Configurações reservado para guardar as interfaces de tipos de módulos importadas, a rotina AnalisaImport é chamada pelo programa principal.

Ao término da execução desta rotina, a informação

sobre a quantidade total de interfaces importadas também é obtida e armazenada na Tabela de Configurações.

e) Rotina AnalisaExport

Esta rotina possui como função armazenar, na Tabela de Configurações, a informação de quantos e quais são os módulos de implementação que estão sendo exportados pela sub-configuração analisada.

f) Rotina MontaModCriado

Quando as instâncias dos módulos de implementação ou de subconfiguração são criadas, a rotina MontaModCriado é chamada para preencher o campo da Tabela de Configurações destinado a guardar as instâncias.

A rotina AnalisaTipo será chamada dentro da rotina MontaModCriado para validar o tipo das instâncias. No próximo item a rotina AnalisaTipo será descrita com mais detalhes.

No caso da (sub-) configuração que está sendo analisada não pertencer ao primeiro nível de hierarquia, isto é, ser composta de instâncias de sub-configurações e não de módulos de implementação, a rotina MontaModCriado preenche o campo da Tabela de Tipos de Configuração reservado para armazenar as instâncias de um determinado tipo de configuração que forem criadas.

Em seguida, é necessário verificar se as estações lógicas associadas às instâncias criadas pertencem à lista de estações, a qual foi definida através da declaração STATIONS. Para isto, a rotina AnalisaEstacao, que será apresentada num item mais adiante, será chamada pela rotina MontaModCriado.

O campo da Tabela de Configurações que armazena as instâncias, guarda, além do nome das instâncias, outras informações. A Figura (V.8) apresenta o tipo que foi definido para este campo.

```
TYPE PointerEstacoesLogicas = POINTER TO EstacaoLogicaType;
TYPE EstacaoLogicaType = RECORD
                             Estaçoes : ARRAY [0..39] OF CHAR;
                             ProxEstacao : PointerEstacoesLogi
                                           cas;
                           END (*record*);

TYPE PointerModulosCriados = POINTER to ModCriadoType;

TYPE ModCriadoType = RECORD
                         NomeModulo : ARRAY [0..39] OF CHAR ;
                         Tipo : ARRAY [0..39] OF CHAR;
                         IndTabConfig : CARDINAL ;
                         Estacao : PointerEstacoesLogicas;
                         NumEstacao : CARDINAL ;
                         ProxModCriado :PointerModulosCriados;
                     END (*record*);
```

FIGURA V.8 - Definição de Alguns Tipos Utilizados Pelos Campos da Tabela de Configurações

Como podemos observar através da Figura (V.8), para cada instância criada são armazenadas também as suas estações lógicas associadas, a quantidade de estações lógicas utilizadas, o tipo da instância criada, e o índice na Tabela de Configurações correspondente àquele tipo. Esta última informação só é preenchida quando se tratar de instâncias de sub-configurações.

g) Rotina AnalisaTipo

A principal função da rotina AnalisaTipo é verificar se o tipo que foi recebido como parâmetro é válido ou não.

Quando o nível de hierarquia da sub-configuração que está sendo analisada for igual a um, o tipo tem que ser, necessariamente, de um módulo de implementação. Este tipo foi criado utilizando-se o método descrito na seção III.3 do

capítulo III. Logo, o seu nome tem que pertencer ao diretório do sistema, uma vez que ele é o nome do arquivo que contém o código objeto referente ao módulo de implementação.

No caso de sub-configurações pertencentes aos demais níveis de hierarquia, o tipo deve estar declarado na Tabela de Tipos de Configuração.

h) Rotina AnalisaEstacao

Esta rotina só é chamada pela rotina MontaModCriado quando estiverem sendo criadas instâncias de sub-configurações.

O objetivo de se chamar esta rotina, é verificar se a quantidade de estações lógicas associadas às instâncias criadas, coincide com àquela que foi definida pela sub-configuração que está sendo utilizada como tipo destas instâncias.

i) Rotina ModuloExportadoCriado

Esta rotina foi definida como uma função do tipo "BOOLEAN", retornando ao programa que a chamou, o valor verdadeiro ou falso. O valor é armazenado numa variável que é utilizada na chamada da função.

Quando a sub-configuração que está sendo analisada tiver declarado algumas instâncias de implementação para serem exportadas, faz-se necessário verificar se estas instâncias foram criadas dentro desta sub-configuração, ou se elas estão sendo propagadas para serem utilizadas por outras (sub-) configurações pertencentes a um nível de hierarquia mais alto. A função ModuloExportadoCriado tem como objetivo realizar este teste.

Para cada uma das instâncias de implementação pertencentes à lista de exportações definida pela declaração EXPORT, o programa principal chamará a função ModuloExportadoCriado, passando como parâmetro o nome da instância, o nível de hierarquia da sub-configuração que está sendo analisada, e a entrada (índice) da Tabela de Configurações correspondente a esta sub-configuração.

Primeiramente, a função ModuloExportadoCriado pesquisa se a instância recebida como parâmetro faz parte da lista de instâncias criadas. Em caso negativo, é verificado se a sub-configuração pertence ao primeiro nível de hierarquia, pois assim sendo, terá ocorrido um erro. Se a sub-configuração não for do primeiro nível de hierarquia, pesquisa-se, na Tabela de Tipos, todas as sub-configurações que compõem a sub-configuração que está sendo analisada. É verificado então, se a instância foi criada em alguma destas sub-configurações que são utilizadas pela sub-configuração analisada. Logo, podemos obervar que a função ModuloExportadoCriado é utilizada recursivamente até certificar-se da criação da instância, ou então, assinalar a ocorrência de um erro referente à exportação de uma instância de implementação que não foi criada.

j) Rotina AnalisaLink

A rotina AnalisaLink tem como função verificar se as ligações entre os módulos estão sendo realizadas respeitando-se as exportações e importações previamente definidas.

Em primeiro lugar, a rotina AnalisaLink certifica se o módulo que está importando o procedimento, isto é, o módulo posicionado à esquerda da palavra WITH, foi criado através da declaração CREATE. Isto é realizado consultando-se a lista de módulos criados, na entrada da Tabela de Configurações referente à (sub-) configuração que

está sendo analisada. Caso o nome do módulo não esteja presente nesta lista, registra-se a ocorrência de um erro. Em seguida verifica-se se o módulo que contém o procedimento exportado, isto é, o módulo que está a direita da palavra WITH, também foi criado através da declaração CREATE. Porém, para este caso, o módulo pode ter sido importado, e não criado dentro da (sub-) configuração. Então, não encontrando-se o módulo na lista de módulos criados, pesquisa-se, também na Tabela de Configurações, a lista de interfaces de módulos importadas.

V.4.2 Segunda Parte do Interpretador (Módulo de Configuração)

A principal função da segunda parte do interpretador é prover a interface necessária aos "stubs" do servidor e do cliente, e também ao PCRP, de forma que o sistema seja capaz de atender perfeitamente uma chamada remota de procedimento. Todas estas interfaces, que são o procedimento ligador, a Tabela de Importações e a de Exportações, são utilizadas em tempo de execução.

A Tabela de Importações representa a interface com o "stub" do cliente, enquanto que a Tabela de Exportações é a interface com o "stub" do servidor, e o procedimento ligador com o núcleo de suporte de tempo de execução (PCRP).

Para poder executar uma RPC é preciso ter localizado anteriormente a estação servidora. O módulo de configuração é responsável, em tempo de execução, por realizar a ligação implícita entre a estação cliente e a estação servidora.

A rotina InitRCall, pertencente ao PCRP, é chamada pelo módulo de configuração durante a sua fase de inicialização. Esta rotina informa ao PCRP tanto a identificação da estação, como o endereço do procedimento ligador. Estes dois dados são passados como parâmetros na

chamada da rotina.

A identificação da estação é empregada pelo PCRP na ativação da rede, e o endereço do procedimento ligador é utilizado para desviar o controle de execução quando ocorrer uma chamada remota.

V.5 INTERFACE DO MÓDULO DE CONFIGURAÇÃO COM OS "STUBS" E COM O NÚCLEO DE PROGRAMAÇÃO

Diante de tudo que já foi mencionado até aqui, pode-se observar que existe uma grande interação entre o módulo de configuração, o PCRP, e os "stubs", durante a execução de uma chamada remota.

Passaremos a descrever agora os passos que ocorrem para que os componentes envolvidos numa chamada remota de procedimento obtenham todas as informações necessárias.

Como já foi ilustrado pela Figura V.4, na Tabela de Importações são catalogados todos os módulos importados, o endereço da estação servidora, e o índice da Tabela de Exportações, que identifica o endereço do despachante associado ao módulo que foi importado.

Na fase de inicialização do "stub" do cliente, existe uma consulta à Tabela de Importações, (interface do "stub" do cliente com o módulo de configuração), de forma que seja possível identificar a localização da interface importada, representada pelo "stub" do cliente em questão. Estas informações (o endereço da estação servidora e o índice da Tabela de Exportações) são guardadas para serem empregadas nas sucessivas chamadas remotas.

O "stub" do servidor, por sua vez, na fase de inicialização, encarrega-se de preencher o endereço do despachante na Tabela de Exportações do módulo de

configuração (interface do "stub" do servidor com o módulo de configuração).

A interface existente entre o módulo de configuração e os "stubs" do cliente e do servidor é realizada, respectivamente, através do compartilhamento das Tabelas de Importações e Exportações, as quais são estruturas de dados locais a cada estação física que compõe a arquitetura do sistema distribuído.

Logo, quando ocorrer uma chamada remota de procedimento, a execução será desviada para o "stub" do cliente, através de uma chamada local, onde o primeiro procedimento a ser realizado é a solicitação de um pacote ao PCRP. O "stub" do cliente deverá então, preencher alguns campos deste pacote com as informações que ele guardou durante a sua fase de inicialização.

De posse das informações adquiridas a partir do módulo de configuração, dos parâmetros referentes à chamada remota, e da quantidade total de bytes ocupados por estes parâmetros, o "stub" do cliente, após preencher os campos do pacote reservados para conter tais informações, passa o pacote para o PCRP. O PCRP se incumbe de colocar este pacote no nó destino, cuja identificação foi obtida através do módulo de configuração, e armazenada no pacote pelo "stub" do cliente.

Este pacote chegando ao nó destino é recebido pelo PCRP contido na estação servidora. O PRCP da estação servidora tem como função ativar o procedimento ligador, cujo endereço foi fornecido pelo módulo de configuração quando este estava sendo inicializado.

O procedimento ligador receberá o pacote que contém o índice da Tabela de Exportações. Este índice será acessado para obter-se o endereço do despachante referente ao

procedimento que foi chamado remotamente.

Ao término da execução do procedimento, o controle vai sendo retornado até que os resultados cheguem ao cliente que fez a chamada.

Logo, como podemos observar, quando dentro de um trecho de programa for executada uma chamada de procedimento remoto, que em geral passa parâmetros de entrada e de saída, sendo estes últimos utilizados para guardar os resultados da execução do procedimento chamado, ocorrerá uma transferência do fluxo de controle de execução. O controle de execução só voltará para o trecho de programa que fez a chamada depois do procedimento ter acabado a sua execução e de ter retornado os resultados.

Todos estes passos que acabamos de citar foram ilustrados na Figura (II.4), que apresenta um exemplo da implementação de RPC.

CAPÍTULO VI

EXPERIÊNCIA DO USO DO INTERPRETADOR

Os vários exemplos apresentados neste capítulo servirão para mostrar a potencialidade da linguagem de configuração que foi descrita no Capítulo III. Um destes exemplos é tomado como base para apresentar-se os resultados obtidos após a interpretação de um programa escrito na linguagem de configuração que foi proposta.

VI.1 EXEMPLOS ILUSTRATIVOS

VI.1.1 Exemplo da Enfermaria

O exemplo escolhido para servir de base foi aquele introduzido por STEVES et alii |41| que será descrito a seguir:

"Um hospital precisa de um sistema de controle de pacientes. Cada paciente é controlado por um dispositivo análogo que mede fatores tais como pulso, temperatura, pressão do sangue e resistência da pele. O sistema lê esses fatores periodicamente. Para cada paciente são especificados limites de variação de segurança para cada fator. Se um fator tomar um valor fora do limite de variação definido ou se um instrumento falhar, a estação da enfermaria é notificada".

Este exemplo já foi desenvolvido na linguagem CONIC (MAGEE |19|), e nós o reestruturamos para poder ser escrito na linguagem Modula-2, usando a linguagem de configuração proposta.

Este projeto decompõe a função do sistema em

sub-funções que podem ser associadas a módulos exportando e importando interfaces específicas.

O esquema físico do sistema de controle pode ser representado pela Figura (VI.1):

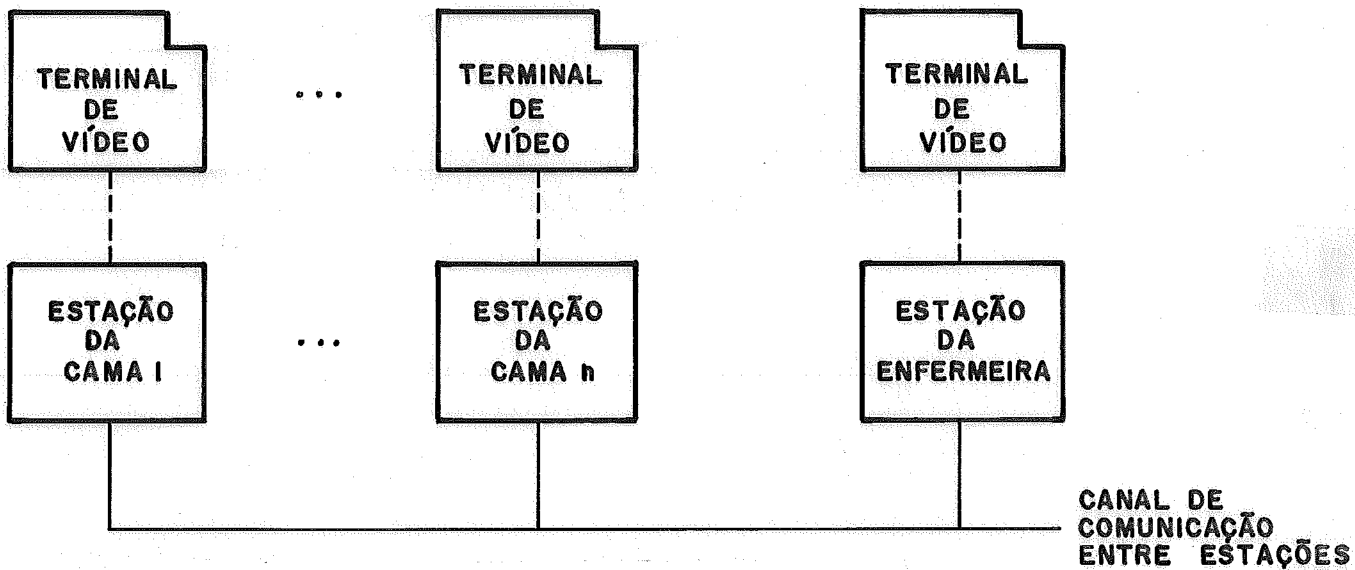


FIGURA VI.1 - Sistema de Controle de Pacientes

As estações das camas são todas iguais, contendo um terminal de vídeo onde são mostrados os valores dos fatores que estão sendo controlados, e através do qual são introduzidos os limites de variação de segurança. A estação da enfermaria emite alarmes, e pode querer repetir no seu terminal de vídeo os valores dos fatores que estão sendo controlados em cada paciente.

Cada estação lógica, que será carregada numa única estação física, é constituída por um conjunto de módulos ligados, que executam as funções da estação, e poderão pertencer a mais de uma configuração, e não necessariamente a uma só.

Para especificar este sistema definimos três tipos

de configurações, uma correspondente à Configuração do Paciente, da qual serão criadas tantas instâncias quantas camas existem, uma Configuração da Enfermeira, e uma Configuração da Enfermaria composta pelas instâncias criadas a partir das duas configurações anteriores.

A configuração da Enfermeira utiliza unicamente a estação lógica da Enfermeira. A Configuração do Paciente precisa de duas estações lógicas, a dele e a da enfermeira, já que algumas das funções são executadas perto da cama e outras são executadas na estação da enfermeira, tais como acionar o alarme e mostrar os dados do paciente no vídeo da enfermeira, quando estes forem requisitados. Estas funções precisam de módulos específicos para cada cama.

Vamos então mostrar a especificação da configuração do sistema usando nossa linguagem de configuração, sem nos preocupar em explicar os detalhes das funções de cada módulo que estão programados em Modula-2. A descrição completa do sistema seria muito grande, e para efeitos deste trabalho achamos suficiente a abordagem escolhida.

Descrevemos a seguir, sucintamente, as funções dos tipos de módulos utilizados para formar a configuração do exemplo apresentado:

psim: este módulo simula as entradas dos fatores do paciente a serem controlados e importa a interface de pmonit.

pmonit: este módulo compara as leituras recebidas com os limites de variação e provoca um alarme quando o valor estiver fora do limite. Armazena todas as informações do paciente para poder fornecê-las quando forem solicitadas. Ele exporta uma interface para psim e importa as interfaces de palarm, de psel

e de pdisp.

pdisp: este módulo recebe informações de um cliente para formatá-las e encaminhá-las para serem exebidas. Ele importa a interface de kwind e exporta uma interface para o cliente.

pcom: este módulo permite que sejam introduzidas novas informações do paciente através do console da cama do paciente. Ele importa as interfaces de pmonit e de zterm.

palarm: este módulo aceita as mensagens de alarme enviadas a partir das camas e as formata para que sejam exibidas. Ele exporta uma interface para pmonit e importa a interface de kwind.

kwind: este módulo formata a tela do console. Ele exporta uma interface para pdisp e zlogw e importa a interface de zterm.

zlogw: este módulo formata os erros para serem exibidos na tela. Ele importa a interface de kwind.

psel: este módulo recebe informações das camas e as armazena para tê-las disponíveis para pedidos da enfermeira. Ele exporta uma interface para nursecom e pmonit e importa a interface de pdisp.

nursecom: este módulo executa as funções de interpretação dos comandos que permitem à enfermeira escoher uma cama para monitorar. Ele importa as interfaces de zterm e de psel.

zterm: este módulo corresponde ao gerenciador de console e contém quatro tarefas internas que gerenciam respectivamente a leitura de linha, escrita de linha, a tela e o teclado.

Este exemplo se caracteriza por ter um único tipo de módulo de implementação para cada interface de módulo, e por possuir dois níveis de hierarquia. No primeiro nível são definidas as sub-configurações PatientConf e NurseConf, a serem combinadas no segundo nível para formar a configuração WardConf, na qual casam as importações de interfaces de módulos com as exportações de módulos de implementação. Como neste exemplo só tem um módulo de implementação para cada módulo de interface, usamos o mesmo nome para os dois nas configurações definidas a seguir.

A configuração do paciente, considerando os módulos principais que a constituem, é ilustrada na Figura (VI.2):

```
CONFIGURATION : PatientConf;
   USE zterm,pmonit,pcom,kwind,pdisp,psim,zlogw,palarm,psel;
   STATIONS Bed, Nurse;
   IMPORT Select : psel, ConsoleN:zterm;
   CREATE
      Console:zterm ON Bed;
      Monitor:pmonit ON Bed;
      Command:pcom ON Bed;
      ErrorWindow, Window:kwind ON Bed;
      Display: pdisp ON Bed;
      Scanner:psim ON Bed;
      ErrorFormat:zlowg ON Bed;
      AlarmN:palarm ON Nurse;
      WindowN:kwind ON Nurse;
   LINK
      WindowN WITH ConsoleN;
      AlarmN WITH WindowN;
      Monitor WITH Select,AlarmN,Display;
      Command WITH Monitor,Console;
      ErrorWindow,Window WITH Console;
      Display WITH Window;
      Scanner WITH Monitor;
      ErrorFormat WITH ErrorWindow;
END PatientConf.
```

FIGURA VI.2 - Declaração da Sub-Configuração PatientConf

As ligações dentro desta configuração ligam instâncias de módulos locais entre si, e instâncias de módulos de implementação desta configuração com interfaces

de módulos de outras configurações, explicitadas na lista de importações. Os nomes das interfaces dos módulos importadas são locais a este tipo de configuração, e serão substituídos por instâncias de módulos reais na hora de definir as ligações das configurações, como será visto mais adiante na Figura(VI.4).

Convém destacar que, já que as estações lógicas Bed e Nurse serão mapeadas em estações físicas diferentes, a terceira declaração de LINK estabelece uma ligação remota, já que o módulo AlarmN pertence à estação Nurse, enquanto Monitor pertence à estação Bed. Podemos observar também, que a ligação de Monitor com Select é potencialmente remota, mas só poderemos confirmá-lo quando for ligada esta sub-configuração com outra, no próximo nível de hierarquia. A primeira declaração de ligação é local porque os dois módulos pertencem a mesma estação Nurse.

A configuração da enfermeira, considerando os módulos principais que a constituem, é ilustrada na Figura (VI.3):

```
CONFIGURATION: NurseConf;
   USE nursecom, pdisp, psel, zterm, zlogw, kwind;
   STATIONS Nurse;
   EXPORT Selector:psel,ConsoleNurse:zterm;
   CREATE
      Command:nursecom ON Nurse;
      Display:pdisp ON Nurse;
      Selector:psel ON Nurse;
      ConsoleNurse:zterm ON Nurse;
      ErrorFormat:zlogw ON Nurse;
      ErrorWindow,Window:kwind ON Nurse;

   LINK
     Window,ErrorWindow WITH ConsoleNurse;
     ErrorFormat  WITH ErrorWindow;
     Display WITH Window;
     Command WITH Selector,consoleNurse;
     Selector WITH Display;
END NurseConf.
```

FIGURA VI.3 - Declaração da Sub-Configuração NurseConf

Passaremos agora a definir a configuração da enfermaria ilustrada pela Figura(VI.4),que é composta por instâncias das Configurações já declaradas.

```
CONFIGURATION: WardConf;
   USE NurseConf, PatientConf;
   STATIONS Bed1, Bed2, Bed3, Bed4, Bed5, Nurse;
   CREATE
      Maria:NurseConf ON Nurse;
      Pat1:PatientConf ON Bed1,Nurse;
      Pat2:PatientConf ON Bed2,Nurse;
      Pat3:PatientConf ON Bed3,Nurse;
      Pat4:PatientConf ON Bed4,Nurse;
      Pat5:PatientConf ON Bed5,Nurse

   LINK
     Pat1,Pat2,Pat3,Pat4,Pat5
     WITH Maria.Selector,Maria.ConsoleNurse;
END WardConf.
```

FIGURA VI.4 - Declaração da Configuração WardConf

O sistema aqui descrito é composto por uma enfermeira controlando cinco camas de pacientes. Podemos ressaltar que a configuração de mais alto nível não exporta nem importa nada. É nesta configuração que são feitas as ligações entre as Configurações de PatientConf e NurseConf através da associação entre as interfaces dos módulos importadas pelos pacientes e as instâncias dos módulos de implementação exportadas pela enfermeira.

Para completar este exemplo, podemos explicitar o mapeamento da configuração lógica na configuração física através de:

```
LOAD Enfermeira:WardConf ON  Station1,Station2,Station3,
                             Station4,Station5,Station6.
```

VI.1.2 Exemplo de Propagação de Importações e Exportações

Este exemplo tem como objetivo mostrar a propagação

de importações de interfaces de módulos e de exportações de módulos de implementação. Para isto foi estendido o exemplo da seção anterior, acrescentando dois tipos de sub-configurações no primeiro nível de hierarquia, chamados ControlConf e DoctorConf. Foi acrescentado também um terceiro nível de hierarquia chamado FloorConf, formado a partir das sub-configurações WardConf, ControlConf e DoctorConf.

A sub-configuração ControlConf tem como função fazer algum cálculo estatístico sobre os dados dos pacientes, sendo que para isto obtém informações através da Nurseconf. A sub-configuração DoctorConf desempenha o papel do doutor que será consultado em determinadas situações pela sub-configuração NurseConf. A configuração de terceiro nível FloorConf representa um andar de um hospital, no qual estão alocados um doutor, uma enfermeira e cinco pacientes. É neste último nível que vão casar todas as importações de interfaces de módulos com as exportações de módulos de implementação.

Para melhor entendimento do exemplo apresentado ilustramos resumidamente a sua estrutura através da Figura (VI.5).

lo. nível de hierarquia

Tipos de sub-configuração

PatientConf
    Importa de NurseConf

NurseConf
    Importa de DoctorConf
    Exporta para PatientConf e ControlConf

ControlConf
    Importa de NurseConf

DoctorConf
    Exporta para NurseConf

2o. nível de hierarquia

Tipo de sub-configuração

WardConf
    Importa de DoctorConf
    Exporta para ControlConf
 1 instância do tipo Nurseconf
 5 instâncias do tipo PatientConf

3o. nível de hierarquia

Tipo de configuração

FloorConf
    1 instância do tipo WardConf
    1 instância do tipo ControlConf
    1 instância do tipo DoctorConf

FIGURA VI.5 - Esquema da Configuração do Exemplo de FloorConf com Controle Estatístico

Queremos destacar que na sub-configuração WardConf do segundo nível de hierarquia, casam as importações de interfaces de módulos de PatientConf com as exportações de módulos de implementação de NurseConf. É neste nível que mostramos a propagação tanto da importação declarada em DoctorConf quanto da exportação declarada em NurseConf, que só serão casadas no terceiro nível de hierarquia, na configuração FloorConf.

Apresentaremos a seguir, com mais detalhes, a definição das sub-configurações e da configuração mencionadas.

A definição da sub-configuração PatientConf permanece inalterada em relação ao exemplo anterior e é ilustrada na Figura (VI.2).

Para este exemplo é necessário acrescentar à definição anterior da sub-configuração NurseConf a importação da interface do módulo do tipo consDoctor, contido na sub-configuração DoctorConf, e a exportação do

módulo de implementação correspondente à interface do módulo importada na sub-configuração ControlConf, do tipo xtipo.

```
CONFIGURATION: NurseConf;
   USE nursecom, pdisp, psel, zterm, zlogw, kwind, xtipo,
            ytipo, consDoctor;
   STATIONS Nurse;
   IMPORT ConsultDoctor : consDoctor;
   EXPORT Selector:psel,ConsoleNurse:zterm,X:xtipo;
   CREATE
      Command:nursecom ON Nurse;
      Display:pdisp ON Nurse;
      Selector:psel ON Nurse;
      ConsoleNurse:zterm ON Nurse;
      X: xtipo ON Nurse;
      Y: ytipo ON Nurse;
      ErrorFormat :zlogw ON Nurse;
      ErrorWindow,Window:kwind ON Nurse;

   LINK
      Window,ErrorWindow WITH ConsoleNurse;
      ErrorFormat WITH ErrorWindow;
      Display WITH Window;
      Command WITH Selector,ConsoleNurse;
      Selector WITH Display;
      X WITH Selector;
      Y WITH ConsultDoctor;
      X WITH Y ;
END NurseConf.
```

FIGURA VI.6 - Declaração da Sub-Configuração NurseConf

Os módulos X e Y foram acrescentados para fazer consultas ao doutor em relação às informações que precisam ser utilizadas para os cálculos estatísticos, a partir dos dados recolhidos dos pacientes.

A definição da sub-configuração de ControlConf é ilustrada pela Figura (VI.7).

```
CONFIGURATION: ControlConf;
   USE wtipo,ztipo,xtipo;
   STATIONS Nurse;
   IMPORT X: xtipo;
   CREATE
      W: wtipo ON Nurse;
      Z: ztipo ON Nurse;

   LINK
     W WITH Z, X;
END ControlConf.
```

FIGURA VI.7 - Declaração da Sub-Configuração ControlConf

Esta sub-configuração faz uns cálculos estatísticos utilizando os módulos W e Z, e importa, através de X, dados fornecidos pela enfermeira. Esta importação será casada num nível superior de hierarquia.

A última sub-configuração neste primeiro nível de hierarquia é a DoctorConf, cuja definição é ilustrada pela Figura (VI.8).

```
CONFIGURATION: DoctorConf;
    USE consDoctor;
    STATIONS Doctor;
    EXPORT CD:consDoctor;
    CREATE
        CD:consDoctor ON Doctor;
END DoctorConf.
```

FIGURA VI.8 - Declaração da Sub-Configuração DoctorConf

Esta sub-configuração consiste num único módulo de implementação, cuja interface será importada pela enfermeira, e portanto não contém nenhuma ligação.

No segundo nível de hierarquia, este exemplo define uma sub-configuração chamada WardConf, que consiste de uma instância de enfermeira e cinco instâncias de pacientes, analogamente ao exemplo anterior.

Por causa das modificações acrescentadas à sub-configuração NurseConf, a definição ilustrada pela Figura (VI.9) mostra algumas alterações em relação à versão anterior da Figura (VI.4).

```
CONFIGURATION: WardConf;
   USE NurseConf,PatientConf,xtipo,consDoctor;
   STATIONS Bed1,Bed2,Bed3,Bed4,Bed5,Nurse;
   IMPORT ConsultDoctor:consDoctor;
   EXPORT X:xtipo;
   CREATE
      Maria:NurseConf ON Nurse;
      Pat1:PatientConf ON Bed1,Nurse;
      Pat2:PatientConf ON Bed2,Nurse;
      Pat3:PatientConf ON Bed3,Nurse;
      Pat4:PatientConf ON Bed4,Nurse;
      Pat5:PatientConf ON Bed5,Nurse;

   LINK
      Pat1,Pat2,Pat3,Pat4,Pat5
      WITH Maria.Selector,Maria.ConsoleNurse;
END WardConf.
```

FIGURA VI.9 - Declaração da Sub-Configuração WardConf

Como já mencionamos anteriormente, neste segundo nível da hierarquia aparecem as propagações, por um lado, da importação da interface do módulo ConsultDoctor, utilizada pela enfermeira, e, por outro, da exportação do módulo de implementação X contido na enfermeira.

Para finalizar este exemplo, mostraremos a definição do terceiro e último nível de configuração chamado FloorConf, ilustrada pela Figura(VI.10),que consiste de uma instância de cada uma das três sub-configurações WardConf, ControlConf e DoctorConf.

```
CONFIGURATION: FloorConf;
   USE WardConf, ControlConf, DoctorConf;
   STATIONS Bed1,Bed2,Bed3,Bed4,Bed5,Nurse,Doctor;
   CREATE
      Enfermaria:WardConf ON Bed1,Bed2,Bed3,Bed4,Bed5,Nurse;
      ControleEstatistico:ControlConf ON Nurse;
      Jose:DoctorConf ON Doctor;

   LINK
      ControleEstatistico WITH Enfermaria.X;
      Enfermaria WITH Jose.CD;
END FloorConf.
```

FIGURA VI.10 - Declaração da Configuração FloorConf

Podemos notar que neste último nível casam a importação e a exportação propagadas, que tinham ficado pendentes. Queremos salientar também, que a importação de ConsultDoctor é feita através da instância Enfermaria da sub-configuração do tipo WardConf, mas ela só é visível para a instância Maria da sub-configuração do tipo NurseConf, e não é visível para nenhuma das instâncias das sub-configurações do tipo PatienConf contidas nesta instância Enfermaria.

VI.1.3 Exemplo de Uso de Instâncias de Tipos de Módulos e Sub-Configurações Iguais

No exemplo anterior vimos o uso de várias instâncias de uma mesma sub-configuração (PatientConf), que em particular continha declarações de importação. Neste exemplo queremos mostrar o uso de várias instâncias de uma mesma sub-configuração, tanto no caso de sub-configurações que importam, quanto das que exportam, já que estas últimas apresentam alguns problemas adicionais.

Neste caso desenvolveremos o exemplo do andar do hospital descrito antes, mas com a ressalva de que tiramos a sub-configuração do cálculo estatístico do tipo ControlConf por razões que explicaremos mais adiante. O exemplo vai

conter então o mesmo doutor, mas uma enfermaria diferente, com duas enfermeiras do mesmo tipo, e aparecerá somente a propagação da importação já mostrada no exemplo anterior.

A estrutura do exemplo pode ser ilustrada resumidamente pela Figura (VI.11).

```
1o. nível da hierarquia

Tipos de sub-configurações

PatientConf
   Importa de NurseConf

NurseConf
   Importa de DoctorConf
   Exporta para PatientConf

DoctorConf
   Exporta para Nurseconf

2o. nível da hierarquia

Tipo de sub-configuração

WardConfT
   Importa de DoctorConf
 2 instâncias do tipo NurseConf
 5 instâncias do tipo PatientConf

3o. nível da hierarquia

Tipo de configuração

FloorConf1
   1 instância do tipo WardConf
   1 instância do tipo DoctorConf
```

FIGURA VI.11 - Esquema da Configuração do Exemplo de FloorConf1

Neste caso, tendo duas enfermeiras iguais para atender aos cinco pacientes, precisamos ligar explicitamente cada paciente a uma determinada enfermeira. Isto é consequência do fato que, quando o paciente faz a chamada remota do procedimento contido na interface exportada pela enfermeira, não é possível precisar a que instância corresponde essa chamada. Esta correspondência é feita de

forma fixa na declaração LINK. Por causa disto, é que tivemos que retirar a sub-configuração ControlConf, que importava uma interface da sub-configuração NurseConf. Tendo duas enfermeiras não poderíamos diferenciar, na hora de ControlConf fazer a chamada a NurseConf, qual delas estaria chamando, mesmo fazendo as duas ligações de ControlConf com elas.

As sub-configurações do primeiro nível da hierarquia são então, as três mencionadas na Figura (VI.11), sendo que a única diferença em relação ao exemplo anterior é que a sub-configuração do tipo NurseConf agora não exporta mais para ControlConf, que foi retirado neste exemplo. As declarações de PatientConf e DoctorConf, correspondem às Figuras (VI.2 e VI.8). A declaração de NurseConf será ilustrada pela Figura (VI.12).

```
CONFIGURATION: NurseConf;
   USE nursecom,pdisp,psel,zterm,zlogw,kwind,
            ytipo,consDoctor;
   STATIONS Nurse;
   IMPORT ConsultDoctor:consDoctor;
   EXPORT Selector:psel,ConsoleNurse:zterm;
   CREATE
      Command:nursecom ON Nurse;
      Display:pdisp ON Nurse;
      Selector:psel ON Nurse;
      ConsoleNurse:zterm ON Nurse;
      Y:ytipo ON Nurse;
      ErrorFormat:zlogw ON Nurse;
      ErrorWindow,Window:kwind ON Nurse;

   LINK
     Window,ErrorWindow WITH ConsoleNurse;
     ErrorFormat WITH ErrorWindow;
     Display WITH Window;
     Command WITH Selector,ConsoleNurse;
     Selector WITH Display;
     Y WITH ConsultDoctor;
END Nurseconf.
```

Figura VI.12 - Declaração da Sub-Configuração NurseConf

Foram retiradas da sub-configuração NurseConf anterior, ilustrada na Figura (VI.6), a criação e a ligação do módulo X de tipo xtipo, que tínhamos declarado para interagir com a sub-configuração ControlConf.

No segundo nível criamos uma nova sub-configuração do tipo WardConfT, a partir do primeiro exemplo (seção VI.3), no qual acrescentamos uma outra instância da sub-configuração NurseConf. A nova sub-configuração WardConfT estará formada então por duas instâncias de enfermeiras e cinco instâncias de pacientes, e a sua definição é ilustrada na Figura (VI.13).

```
CONFIGURATION: WardConfT;
   USE NurseConf,PatientConf,ConsultDoctor;
   STATIONS Bed1,Bed2,Bed3,Bed4,Bed5,Nurse1,Nurse2;
   IMPORT ConsultDoctor:consDoctor;
   CREATE
      Maria:NurseConf ON Nurse1;
      Ana:NurseConf ON Nurse2;
      Pat1:PatientConf ON Bed1,Nurse1;
      Pat2:PatientConf ON Bed2,Nurse1;
      Pat3:PatientConf ON Bed3,Nurse1;
      Pat4:PatientConf ON BVed4,Nurse2;
      Pat5:PatientConf ON Bed5,Nurse2;

   LINK
     Pat1,Pat2,Pat3 WITH Maria.Selector,Maria.ConsoleNurse;
     Pat4,Pat5 WITH Ana.Selector,Ana.ConsoleNurse;

END WardConfT.
```

FIGURA VI.13 - Declaração da Sub-Configuração WardConfT

Queremos ressaltar, que nesta declaração de WardConfT temos os três primeiros pacientes ligados com a enfermeira Maria, e os outros dois com a enfermeira Ana. As duas enfermeiras foram associadas a estações lógicas diferentes, mas poderiam também estar associadas à mesma estação lógica, e neste caso a declaração teria sido:

```
CREATE
   Maria,Ana:NurseConf ON Nurse;
```

e os pacientes teriam sido declarados todos na mesma estação Nurse.

No terceiro nível temos então a definição da configuração do tipo FloorConf1 ilustrada pela Figura (VI.14).

```
CONFIGURATION: FloorConf1;
   USE WardConfT,DoctorConf;
   STATIONS Bed1,Bed2,Bed3,Bed4,Bed5,Nurse1,Nurse2,Doctor;
   CREATE
      EnfermariaT:WardConfT ON Bed1,Bed2,Bed3,Bed4,Bed5,
                  Nurse1,Nurse2;
      Jose:DoctorConf ON Doctor;

   LINK
      EnfermariaT WITH Jose.CD;
END FllorConf1.
```

FIGURA VI.14 - Declaração da Configuração FloorConf1

Podemos notar que neste último nível da hierarquia se casa a importação programada pela sub-configuração WardConfT, a interface de ConsultDoctor, com a exportação da correspondente, propagada pela configuração DoctorConf. Mas o fato mais importante a ser ressaltado é que a ligação declarada implica em duas ligações implícitas que o interpretador da linguagem de configuração precisa identificar, já que na sub-configuração WardConfT existem duas instâncias, Maria e Ana, da sub-configuração NurseConf que importam a interface de ConsultDoctor.

Este exemplo podia ter sido estruturado, separando a sub-configuração WardConfT em duas, WardConf1 e WardConf2, contendo, respectivamente, a enfermeira Maria com seus três pacientes, e a enfermeira Ana com seus dois pacientes. Esta estrutura não teria introduzido nenhuma diferença conceitual para a configuração final: a única mudança seria em relação às duas ligações explícitas.

Enfermarial WITH Jose.CD;
Enfermaria2 WITH Jose.CD;

supondo que Enfermarial fosse do tipo Wardconfl e Enfermaria2 do tipo WardConf2.

Voltamos a ressaltar aqui, que quando existem mais de uma instância de uma sub-configuração, sejam elas do mesmo tipo ou de tipos diferentes, não podemos diferenciá-las no caso delas exportarem alguma interface que seja importada por outra sub-configuração. Esta limitação é consequência da propriedade de transparência na chamada remota de procedimento, que quisemos implementar para estender Modula-2 para o seu uso em sistemas distribuídos. Queremos deixar claro que uma sub-configuração no primeiro nível da hierarquia está constituída por módulos escritos em Modula-2, e, quando falamos de instâncias de sub-configurações de tipos diferentes, isto corresponde a sub-configurações contendo módulos de implementação de tipos diferentes, mas com a mesma interface. Isto ocorre tanto em relação a módulos locais à sub-configuração quanto em relação a módulos que são visíveis para as outras sub-configurações, por terem sido exportados. Trataremos deste caso no próximo exemplo.

VI.1.4 Exemplo de Uso de Instâncias de Tipos Diferentes de Módulos e Sub-Configurações

Até agora temos tratado exemplos, nos quais para cada interface de módulo existia somente um tipo de módulo de implementação, mesmo com várias instâncias iguais, como no caso anterior. Queremos agora apresentar um exemplo no qual, para uma mesma interface de módulo, existem diferentes tipos de módulos de implementação programados em Modula-2. Para isto escolhemos o caso de ter dentro da sub-configuração do tipo NurseConf tipos de módulos de implementação diferentes para a interface psel, que em particular são exportados. Poderíamos ter escolhido um tipo de módulo local à NurseConf

que não fosse exportado, mas o caso seria análogo só que mais simples.

1o. nível da hierarquia

Tipos de sub-configuração

PatientConf
   Importa de NurseConf1 ou de NurseConf2

NurseConf1
   Importa de DoctorConf
   Exporta para PatientConf

NurseConf2
   Importa de DoctorConf
   Exporta para PatientConf

DoctorConf
   Exporta para NurseConf1 e NurseConf2

2o. nível da hierarquia

Tipos de sub-configuração

WardConf1
   Importa de DoctorConf
   1 instância do tipo NurseConf1
   3 instâncias do tipo PatientConf

WardConf2
   Importa de DoctorConf
   1 instância do tipo NurseConf2
   2 instâncias do tipo PatientConf

3o. nível da hierarquia

Tipo de configuração

FloorConf2
   1 instância do tipo WardConf1
   1 instância do tipo WardConf2
   1 instância do tipo DoctorConf

FIGURA VI.15 - Esquema da Configuração do Exemplo de FloorConf2

Vamos então definir aqui os nomes diferentes para as interfaces dos módulos e para os nomes dos módulos de implementação correspondentes. Chamaremos a interfade do módulo psel, pselint, e os dois tipos de módulos de implementação psel1 e psel2 respectivamente. Vamos supor também que para o módulo do tipo zterm existem dois nomes diferentes, um para a interface do módulo, chamada ztermint, e outro para o módulo de implementação, chamado zterm, mesmo tendo um único tipo de módulo de implementação.

Para desenvolver o exemplo análogo ao anterior, precisamos agora definir dois tipos diferentes de sub-configurações para as duas enfermeiras. Para facilitar a compreensão do exemplo de FloorConf2, ilustraremos resumidamente a sua estrutura com a Figura (VI.15) acima.

Com a definição de nomes diferentes para as interfaces dos módulos e para os tipos dos módulos de implementação, analisemos agora as modificações a serem introduzidas em relação às definições das subconfigurações apresentadas anteriormente.

No primeiro nível da hierarquia vejamos agora as declarações dos tipos de sub-configurações. A declaração do tipo de sub-configuração PatientConf será agora ilustrado pela Figura (VI.16).

Podemos destacar que as únicas diferenças aparecem nas declarações USE e IMPORT, onde foram acrescentados os nomes das interfaces de módulos importadas, pselint e ztermint. Na declaração CREATE é criada uma instância do tipo zterm, que continua igual à anterior, já que estamos supondo que existe um único tipo de módulo correspondente à interface ztermint.

```
CONFIGURATION: PatientConf;
   USE zterm,pmonit,pcom,kwind,pdisp,psim,zlogw,palarm,
             pselint,ztermint;
   STATIONS Bed,Nurse;
   IMPORT Select : pselint,ConsoleN:ztermint;
   CREATE
   Console:zterm ON Bed;
   Monitor:pmonit ON Bed;
   Command:pcom ON Bed;
   ErrorWindow, Window:kwind ON Bed;
   Display:pdisp ON Bed;
   Scanner:psim ON Bed;
   ErrorFormat:zlogw ON Bed;
   AlarmN:palarm ON Nurse;
   WindowN:kwind ON Nurse;

   LINK
     WindowN WITH ConsoleN;
     AlarmN WITH WindowN;
     Monitor WITH Select,AlarmN,Display;
     Command WITH Monitor,Console;
     ErrorWindow,Window WITH Console;
     Display WITH Window;
     Scanner WITH Monitor;
     ErrorFormat WITH ErrorWindow;
END Patientconf.
```

FIGURA VI.16 - Declaração da Sub-Configuração PatientConf

Neste exemplo precisamos definir duas sub-configurações e tipos diferentes para NurseConf, que serão chamadas respectivamente NurseConf1 e NurseConf2, contendo tipos de módulos de implementação diferentes, psel1 e psel2, para o módulo psel usado anteriormente.

A sub-configuração do tipo NurseConf1 é ilustrada pela Figura (VI.17).

A única modificação introduzida na Figura (VI.17) em relação à Figura (VI.12) é a substituição de psel por psel1.

```
CONFIGURATION: NurseConf1;
   USE nursecom,pdisp,psel1,zterm,zlogw,kwind,
            ytipo,ConsDoctor;
   STATIONS Nurse;
   IMPORT  ConsultDoctor:consDoctor;
   EXPORT  Selector:psel1,ConsoleNurse:zterm;
   CREATE
      Command:nursecom ON Nurse;
      Display:pdisp ON Nurse;
      Selector:psel1 ON Nurse;
      ConsoleNurse:zterm ON Nurse;
      Y:ytipo ON Nurse;
      ErrorFormat:zlogw ON Nurse;
      ErrorWindow,Window:kwind ON Nurse;
   LINK
      Window,ErrorWindow WITH ConsoleNurse;
      ErrorFormat WITH ErrorWindow;
      Display WITH Window;
      Command WITH Selector,ConsoleNurse;
      Selector WITH Display;
      Y WITH ConsultDoctor;
END NurseConf.
```

FIGURA VI.17 - Declaração da Sub-Configuração NurseConf1

Analogamente, a sub-configuração do tipo NurseConf2 é ilustrada pela Figura (VI.18) com a única modificação causada pela substituição de psel por psel2.

```
CONFIGURATION: NurseConf2;
   USE nursecom,pdisp,psel2,zterm,zlogw,kwind,
            ytipo,consDoctor;
   STATIONS Nurse;
   IMPORT  ConsultDoctor:consDoctor;
   EXPORT  Selector:psel2,ConsoleNurse:zterm;
   CREATE
      Command:nursecom ON Nurse;
      Display:pdisp ON Nurse;
      Selector:psel2 ON Nurse;
      ConsoleNurse:zterm ON Nurse;
      Y:ytipo ON Nurse;
      ErrorFormat:zlogw ON Nurse;
      ErrorWindow,Window:kwind ON Nurse;

   LINK
      Window,ErrorWindow WITH ConsoleNurse;
      ErrorFormat WITH ErrorWindow;
      Display WITH Window;
      Command WITH Selector,ConsoleNurse;
      Selector WITH Display;
      Y WITH ConsultDoctor;
END NurseConf.
```

FIGURA VI.18 - Declaração da Sub-Configuração NurseConf2

No segundo nível da hierarquia são definidas duas sub-configurações do tipo WardConf1 e WardConf2. A sub-configuração do tipo WardConf1 é definida pela Figura (VI.19) seguinte:

```
CONFIGURATION: WardConf1;
   USE PatientConf,NurseConf1;
   STATIONS Bed1,Bed2,Bed3,Nurse1;
   IMPORT  ConsulDoctor:consDoctor;
   CREATE
      Maria:NurseConf1  ON Nurse1;
      Pat1:PatientConf  ON Bed1,Nurse1;
      Pat2:PatientConf  ON Bed2,Nurse1;
      Pat3:PatientConf  ON Bed3,Nurse1;

   LINK
      Pat1,Pat2,Pat3 WITH Maria.Selector,Maria.ConsoleNurse;
END WardConf1.
```

FIGURA VI.19 - Declaração da Sub-Configuração WardConf1

Esta sub-configuração está composta por uma instância da sub-configuração NurseConf1 e três instâncias da sub-configuração PatientConf. É neste nível que na declaração do LINK são ligadas de forma implícita as interfaces pselint, chamadas localmente de Select, dos três pacientes com o módulo de implementação de tipo psel1 de NurseConf1, chamado localmente de Selector.

Analogamente, a sub-configuração do tipo WardConf2 é definida pela Figura (VI.20) seguinte:

```
CONFIGURATION: WardConf2;
    USE PatientConf,NurseConf2;
    STATIONS Bed4,Bed5,Nurse2;
    IMPORT ConsultDoctor:consDoctor;
    CREATE
        Ana:NurseConf2 ON Nurse2;
        Pat4:PatientConf  ON Bed4,Nurse2;
        Pat5:PatientConf  ON Bed5,Nurse2;

   LINK
      Pat4,Pat5 WITH Ana.Selector,Ana.ConsolNurse;
END WardConf2.
```

FIGURA VI.20 - Declaração da Sub-Configuração WardConf2

Nesta sub-configuração, que contém uma instância da sub-configuração NurseConf2 e duas instâncias de PatientConf, são feitas implicitamente as ligações entre as interfaces pselint, chamadas localmente de Select, dos pacientes com o módulo de implementação de tipo psel2 de NurseConf2, chamado localmente de Selector.

No terceiro nível definimos a configuração do tipo FloorConf2 constituída por uma instância de cada um dos três tipos de sub-configurações WardConf1, WardConf2 e DoctorConf, ilustrada pela Figura (VI.21).

```
CONFIGURATION: FloorConf2;
   USE WardConf1, WardConf2,DoctorConf;
   STATIONS Bed1,Bed2,Bed3,Bed4,Bed5,Nurse1,Nurse2,Doctor;
   CREATE
       Enfermaria1:WardConf1 ON Bed1,Bed2,Bed3,Nurse1;
       Enfermaria2:WardConf2 ON Bed4,Bed5,Nurse2;
       José:DoctorConf ON Doctor;
   LINK
       Enfermaria1,Enfermaria2 WITH José.CD;
END FloorConf2.
```

FIGURA VI.21 - Declaração da Sub-Configuração FloorConf2

Os comentários feitos no exemplo anterior são válidos aqui também. Poderíamos ter definido uma única sub-configuração do tipo WardConfT, contendo uma instância de cada tipo NurseConf1 e NurseConf2, e ligando os respectivos pacientes a uma e outra enfermeiras. Essa modificação não teria causado, neste nível, nenhuma mudança conceitual. Teria aparecido uma única ligação neste terceiro nível, que, de forma implícita, seria equivalente as duas ligações explícitas da Figura (VI.21).

O mais importante a ser destacado neste exemplo é a forma simples na qual é feita a ligação entre a inferface do módulo e a implementação do módulo desejado. Cada sub-configuração é programada sem precisar se preocupar com esta associação, que será feita na declaração LINK, quando

forem combinadas as sub-configurações que contêm as respectivas importações e exportações. O programador da configuração só precisa estar ciente dos nomes das interfaces dos módulos, e dos diferentes tipos de módulos de implementação correspondentes às interfaces. É importante salientar aqui a separação de funções entre a criação de instâncias de módulos e sub-configurações, e as suas ligações. Isto facilita, na reconfiguração, a possibilidade de modificar ligações, já que para isto será necessário somente destruir as ligações existentes e fazer ligações novas, nos casos existentes. Também no caso de modificar os módulos de implementação, esta separação facilita as mudanças a serem introduzidas, que serão minimizadas desta forma.

VI.2 RESULTADOS OBTIDOS APLICANDO-SE O INTERPRETADOR NO EXEMPLO DA SEÇÃO VI.1.2

Usando o exemplo apresentado na seção VI.1.2, vamos mostrar como ficariam preenchidas as estruturas de dados utilizadas pelo interpretador, as quais foram descritas no item V.3 do Capítulo V.

Ao término da interpretação de todas as sub-configurações do primeiro nível de hierarquia, a Tabela de Configurações conterá as informações ilustradas pelas Figuras (VI.22, VI.23, VI.24 e VI.25).

Quando o interpretador analisar a sub-configuração WardConf, pertencente ao segundo nível de hierarquia, ele preencherá mais uma entrada na Tabela de Configurações com as informações ilustradas pela Figura (VI.26).

O interpretador ao analisar esta sub-configuração, perceberá que está ocorrendo uma propagação de exportação, uma vez que o módulo de implementação X do tipo xtipo não foi criado dentro desta sub-configuração. O interpretador

verifica então, se o módulo de implementação foi criado em alguma das sub-configurações que compõem esta sub-configuração. Logo, a Tabela de Tipos que está ilustrada pela Figura (VI.27), será pesquisada até encontrar-se os tipos referentes a sub-configurações, ou seja, os tipos cujo o campo que indica o nível de hierarquia esteja com o valor diferente de zero. Podemos observar através da Figura (VI.27) que o primeiro tipo encontrado, NurseConf, é um tipo de configuração. O interpretador então, consultando a Tabela de Configurações na entrada referente a este tipo, encontra entre as instâncias de módulos criados, o módulo X do tipo xtipo (vide Figura VI.23).

Continuando a interpretação do programa, o nível mais alto de hierarquia é atingido. Depois da interpretação das declarações CONFIGURATION, STATIONS, IMPORT, EXPORT e CREATE, a Tabela de Configurações conterá mais uma entrada com as informações referentes à configuração FloorConf, e que estão ilustradas na Figura (VI.28).

Neste último nível, o interpretador então verifica que a importação do módulo ConsultDoctor do tipo consDoctor, propagada pela sub-configuração WardConf, casa com a exportação realizada pela sub-configuração DoctorConf. É neste nível também, que a exportação X do xtipo, propagada pela sub-configuração WardConf, casa com a importação realizada pela sub-configuração ControlConf.

Consultando a Tabela de Configurações, o interpretador descobrirá que a estação lógica associada ao módulo X é a mesma que está associada ao módulo W pertencente à sub-configuração ControlConf, indicando neste caso que a ligação entre estes dois módulos é uma ligação local e não remóta. Já a ligação entre o módulo Y pertencente à sub-configuração NurseConf, e o módulo CD pertencente à sub-configuração DoctorConf, é uma ligação remota, uma vez que o primeiro módulo está associado a uma

estação lógica (Nurse) diferente daquela a qual o segundo módulo está associado (Doctor).

Devemos ressaltar que as estações lógicas não precisam aparecer sempre com os mesmos nomes em todas as sub-configurações, para significar que está se tratando da mesma estação lógica. O interpretador descobrirá se as estações lógicas são as mesmas ou não, considerando, na hora de realizar o relacionamento entre as estações lógicas, a posição na qual elas apareceram nas declarações STATIONS e CREATE.

Depois de descobrir todos os módulos pertencentes a cada estação física, e quais são as ligações remotas e as locais, o interpretador monta as Tabelas de Importações e de Exportações, que deverão ser carregadas nas estações físicas correspondentes. As Figuras (VI.29, VI.30 VI.31) ilustram estas tabelas, sendo que existirá uma tabela idêntica a da Figura (VI.29) nas estações físicas nº 2, nº 3, nº 4 e nº 5, as quais representam os pacientes PAT2, PAT3, PAT4, e PAT5, respectivamente. O primeiro campo da Tabela de Importações conterá os valores STATION 2, STATION 3, STATION 4 e STATION 5 nas estações físicas nº 2, nº 3,nº 4 e nº 5,respectivamente.

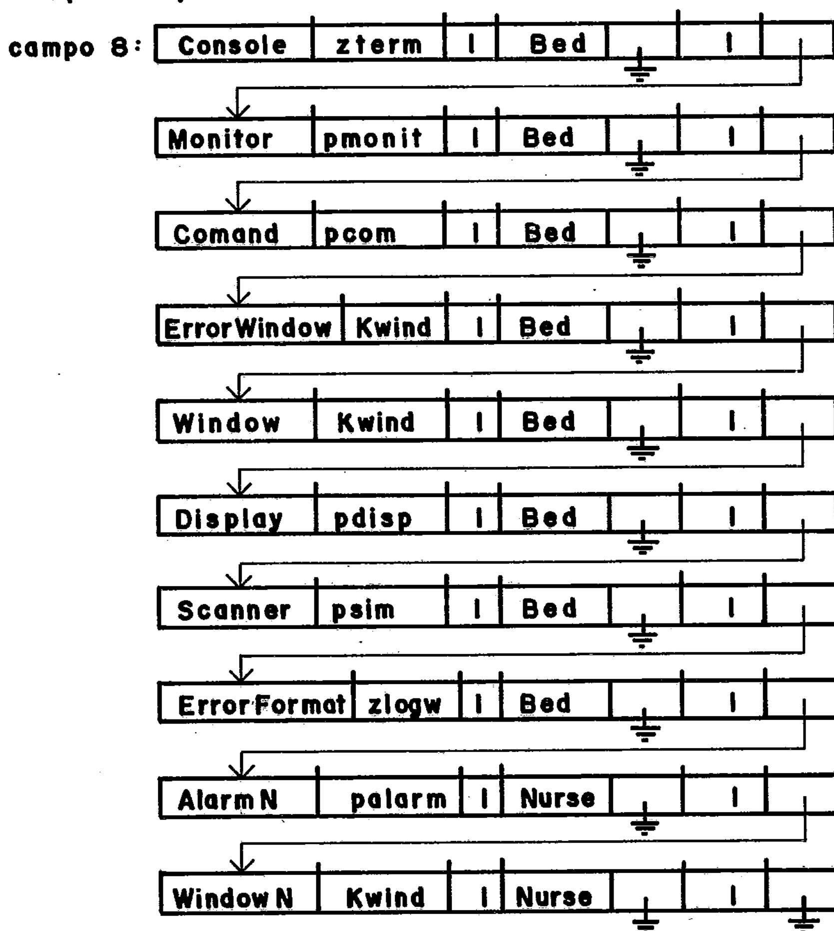


Figura V.22 - Entrada ∅ da Tabela de Configurações Referente à Sub-Configuração PatientConf.

campo 1: NurseConf

campo 2: Nurse | ⏚

campo 3: ConsultDoctor | consDoctor | ⏚

campo 4: Selector | psel | →

ConsoleNurse | zterm | →

X | xtipo | ⏚

campo 5: 1

campo 6: 1

campo 7: 2

campo 8: Command | nursecom | 1 | Nurse | ⏚ | 1 | →

Display | pdisp | 1 | Nurse | ⏚ | 1 | →

Selector | psel | 1 | Nurse | ⏚ | 1 | →

ConsoleNurse | zterm | 1 | Nurse | ⏚ | 1 | →

X | xtipo | 1 | Nurse | ⏚ | 1 | →

Y | ytipo | 1 | Nurse | ⏚ | 1 | →

ErrorFormat | zlogw | 1 | Nurse | ⏚ | 1 | →

Errorwindow | Kwind | 1 | Nurse | ⏚ | 1 | →

Window | Kwind | 1 | Nurse | ⏚ | 1 | ⏚

Figura VI.23 - Entrada 1 da Tabela de Configurações Referente à Sub-Configuração NurseConf.

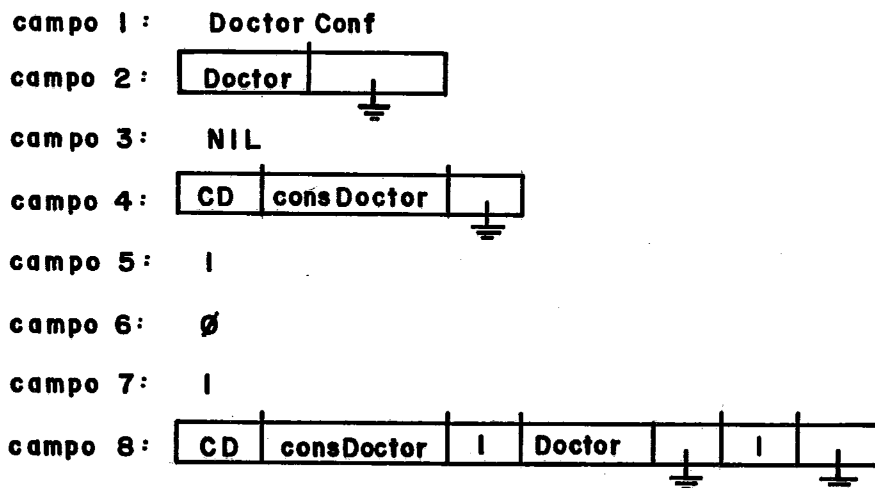


Figura VI.24 - Entrada 2 da Tabela de Configurações Referente à Sub-Configuração DoctorConf.

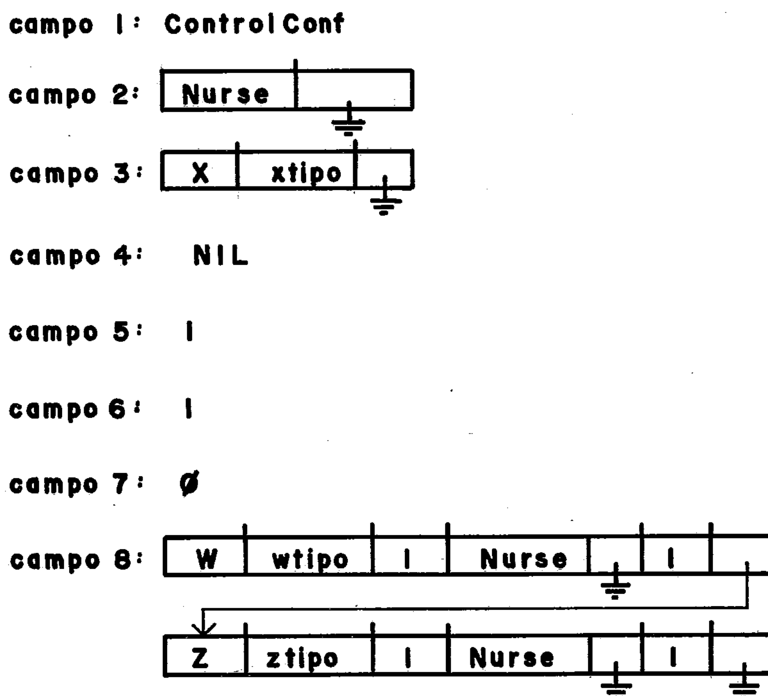
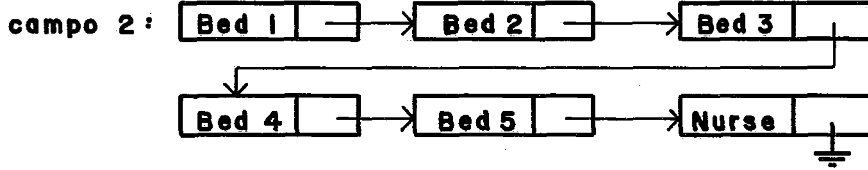


Figura VI.25 - Entrada 3 da Tabela de Configurações Referente à Sub-Configuração ControlConf.

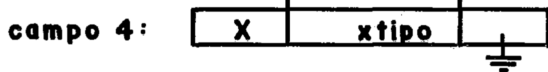
campo 1: WardConf

campo 2:



campo 3:

campo 4:



campo 5: 6

campo 6: 1

campo 7: 1

campo 8:

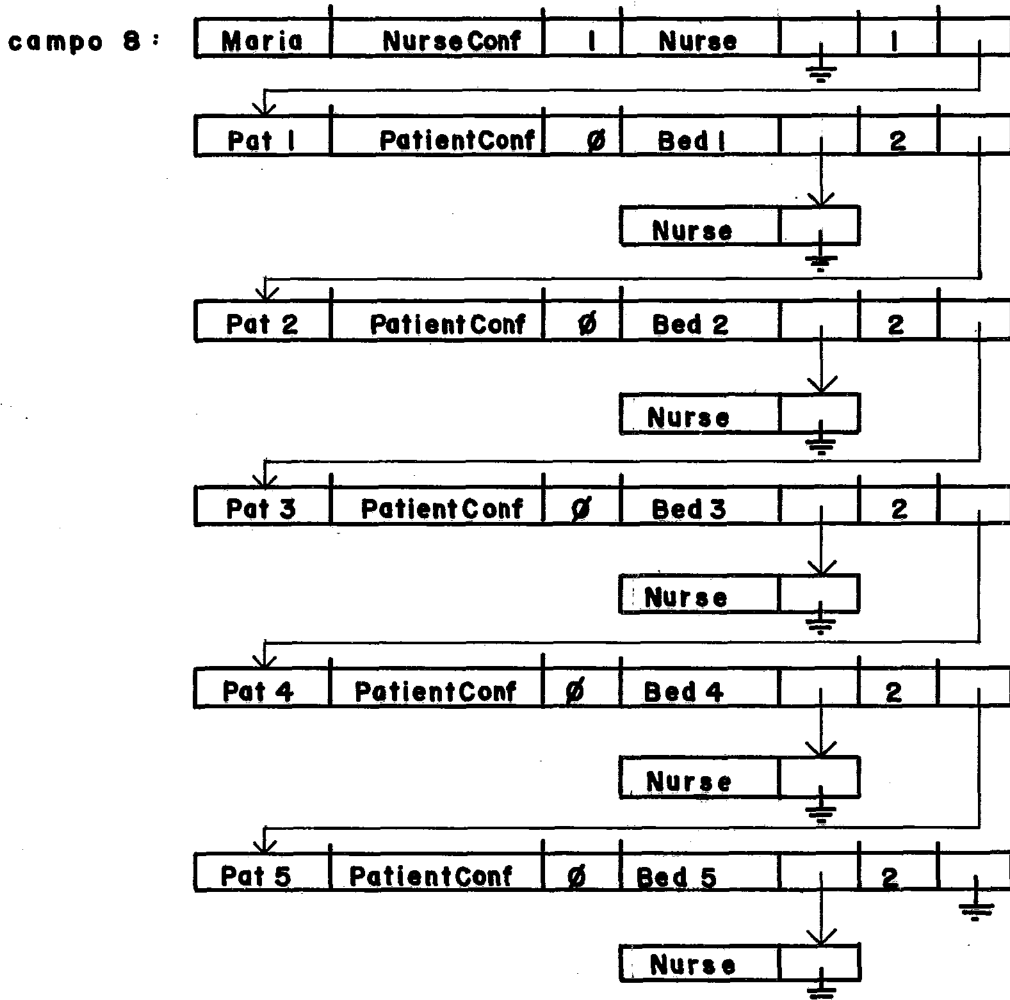


Figura VI.26 - Entrada 4 da Tabela de Configurações Referente à Sub-Configuração WardConf.

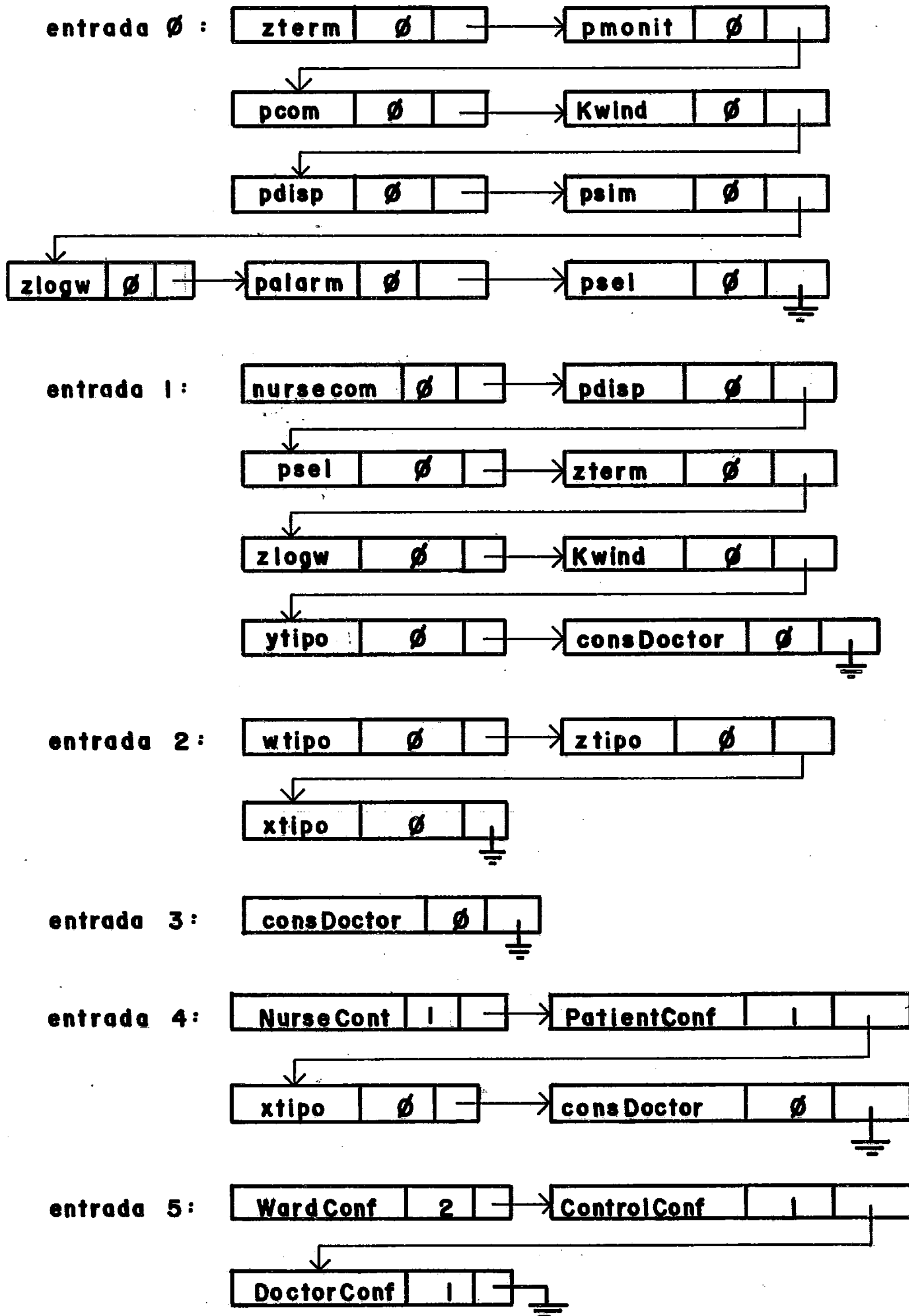
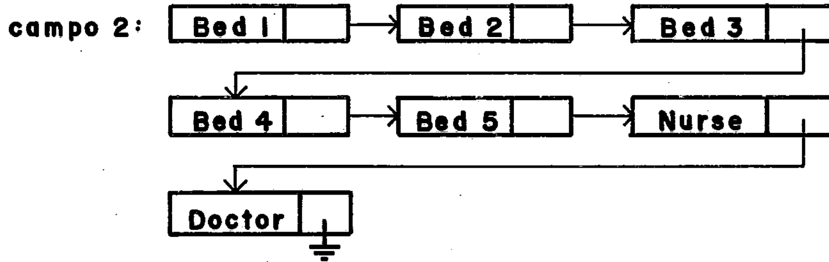


Figura VI.27 - Tabela de Tipos

campo 1: FloorConf



campo 3: NIL

campo 4: NIL

campo 5: 7

campo 6: Ø

campo 7: Ø

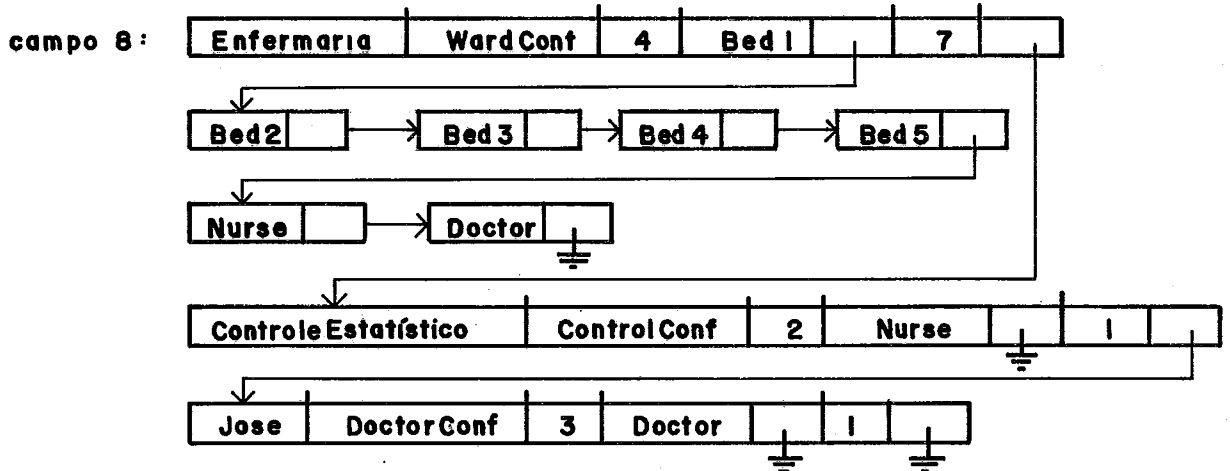


Figura VI.28 - Entrada 5 da Tabela de Configurações Referente à Configuração FloorConf

campo 1: STATION 1

campo 2:

entrada ∅ :	Monitor	Station 6	∅
entrada 1 :	Monitor	Station 6	1

Figura VI.29 - Tabela de Importações da Estação Física nº 1.

campo 1: STATION 6

campo 2:

entrada ∅ :	Selector	
entrada 1:	AlarmN	

Estes campos serão preenchidos pelos "stubs" do servidor com o endereço do despachante referente ao módulo exportado.

Figura VI.30 - Tabela de Exportações da Estação Física nº 6.

campo 1: STATION 7

campo 2:

entrada ∅ :	CD	

obs: É válida a mesma observação feita na figura VI.30

Figura VI.31 - Tabela de Exportações da Estação Física nº 7.

CAPÍTULO VII

CONCLUSÃO

Esta tese teve como objetivo implementar um interpretador para uma linguagem de configuração para um ambiente de programação distribuída. Modula-2 foi a linguagem base para o desenvolvimento dos trabalhos. No entanto, foi necessário acrescentar-lhe algumas extensões de forma que ela atendesse os requisitos impostos às linguagens de programação que se propõem a ser utilizadas em sistemas distribuídos.

Primeiramente foram apresentadas as extensões introduzidas em Modula-2, entre elas o mecanismo de comuinicação e sincronização entre processos denominado chamada remota de procedimento, na forma transparente ao usuário. O núcleo de multiprogramação que foi implementado, e que fornece o suporte necessário à execução de uma chamada remota de procedimento também foi sucintamente apresentado.

Em seguida mostramos a definição da linguagem de configuração para a qual foi implementado o interpretador. Esta linguagem é utilizada para especificar a configuração estática de um sistema distribuído, satisfazendo os seguintes requisitos: a linguagem é declarativa, possui uma unidade de configuração bem definida, permite a definição clara do contexto, possui funções totalmente separadas (definição de tipos, criação de instâncias, ligação de interfaces), permite estruturação hierárquica e finalmente inclui funções de mapeamento e carregamento.

Vários exemplos utilizando esta linguagem de configuração foram apresentados de forma a mostrar as características da linguagem, como também o seu potencial.

Uma comparação extensa foi realizada entre a nossa proposta e outros projetos utilizados para desenvolver sempre todos os requisitos necessários para montar-se um ambiente de programação distribuída estão presentes nas linguagens, sendo que na maioria dos casos apenas alguns deles são considerados.

Apesar da nossa proposta não ter abordado a questão referente à configuração dinâmica, houve uma preocupação, tanto na definição da linguagem de configuração quanto na implementação do interpretador, de realizar-se uma fácil adaptação quando desejar-se estender a linguagem de configuração estática para uma dinâmica. As funções de criação de instâncias e de ligação entre estas instâncias já foram definidas separadamente para facilitar uma configuração dinâmica. Para este caso seria necessário então, incluir na nossa linguagem novas funções, as quais são funções inversas daquelas que foram definidas, ou seja, uma função de destruição de instâncias e outra de desligamento de conexões.

Vamos mencionar agora as pendências que ficaram, indicando também algumas melhorias que podem ser realizadas de forma a tornar o interpretador mais flexível.

Primeiramente devemos ressaltar que não demos muita ênfase no que diz respeito à análise dos comandos. Isto se deve ao fato de existirem métodos bem reconhecidos na literatura, e o nosso objetivo principal ter sido focalizar a parte mais original do trabalho, que é referente ao preenchimento das estruturas de dados que dão suporte, em tempo de execução, ao processamento de uma chamada remota de procedimento.

Uma sugestão para realizar esta análise seria empregar o método denominado LR |1|, o qual utiliza uma

tabela montada a partir da gramática que gerou a linguagem, e a própria gramática. Este método consiste em percorrer os elementos que são recebidos como entrada, da esquerda para a direita, sendo esta a razão que originou o nome do método (Left to Right). Consultando-se a tabela montada e a gramática, são realizadas várias reduções até atingir-se um elemento terminal pertencente à gramática.

Entre as melhorias que poderiam ser realizadas, uma seria a criação de um arquivo de saída contendo todos os erros encontrados durante a interpretação, distinguindo-se cada um deles através de um código acompanhado de uma mensagem. Outra alternativa que mostra-se mais interessante ainda, é implementar o fornecimento destas mensagens de erro de forma interativa com o usuário, já posicionando na linha em que ocorreu o erro, permitindo, desta maneira, que o usuário corrija-o imediatamente. Este método está sendo utilizado amplamente pelo que existe de mais moderno em editores de texto. Dentro dos próprios editores é oferecida a opção de compilação, e na medida em que os erros são encontrados , o controle é retornado automaticamente para a edição, de forma que os erros possam ser consertados instantaneamente.

Pela indisponibilidade de um ambiente distribuído para realização dos testes, não foi possível implementar a interpretação da diretiva referente ao carregamento (LOAD) nas diferentes estações físicas.

No que se refere ao tamanho das tabelas de dados utilizadas pelo interpretador, estas foram dimensionadas com o valor 255, mas nada impede que estas sejam transformadas em listas encadeadas, limitadas apenas pela capacidade de memória da máquina hospedeira onde será executada a interpretação da configuração.

Outro ponto que devemos levantar é que não foi realizado um teste geral, abrangendo todos os componentes

que interagem na execução de uma chamada remota de procedimento, isto porque a parte referente à geração automática de "stubs" está em andamento, não estando ainda concluída. Logo, é possível que alguns erros ocorram quando todas as partes forem integradas, sendo necessário portanto, alterar os módulos que apresentarem problemas.

Pelo fato da linguagem Modula-2 estar sendo mais utilizada no meio acadêmico, e não ter ainda recebido muita divulgação, aqui no Brasil, no ambiente profissional, o nosso trabalho pode ser considerado, de certa forma, como mais um teste de Modula-2 como linguagem de desenvolvimento, além de avaliar realmente a sua viabilidade. O mecanismo de compilação em separado, essencial no desenvolvimento de um sistema do porte deste projeto, mostra-se de uso eficiente, uma vez que várias fontes de erro são eliminadas por partes, permitindo que a fase final, onde todos os módulos são integrados, seja efetuada em um período de tempo bastante reduzido, e com um processo de depuração simplificado.

Concluímos então, que para a nossa proposta foi muito produtivo termos utilizado uma linguagem de programação moderna e elegante, como é o caso de Modula-2, e desde que tenhamos sempre respeito às limitações intrínsecas da mente humana, conseguiremos obter bons resultados.

REFERÊNCIAS

|1| AHO, A.V. & ULLMAN, J.D. - "Principles of Compiler Design". Addison-Wesley, Reading, Massachusetts, 1977, 591 p.

|2| BARNES, J. G. P. - "An Overview of ADA". SOFTWARE-PRACTICE AND EXPERIENCE, 10(1): 851-887, Jul. 1980.

|3| BATE, G. - "Mascot 3: An Informal Introduction Tutorial".Software Engineering Journal, 1(3):93-102, Mai. 1986.

|4| BIRRELL, A.D. & NELSON, B.J. - "Implementing Remote Procedure Calls". ACM Transaction on Computer Systems, 2(1): 39-59, Fev. 1984.

|5| DA SILVA FILHO, N.A. - "Protocolo de Chamada Remota de Procedimento". Dissertação de Mestrado, Departamento de Informática, PUC/RJ, Ago. 1987, 79 p.

|6| DULAY, N. et alii - "The CONIC Configuration Language". Version 1.3. Imperial College Research Report DOC 84/26, Nov. 1984. 12 p.

|7| HOARE, C.A.R. - "Monitors: An Operating System Structuring Concept". Communications of the ACM, 17(10): 549-557, Out. 1974.

|8| ICHBIAH, J.D. et alii - "Rationale for Design of the ADA Programming Language". ACM SIGPLAN Notices, 14(6) : Part B, Jun. 1979.

|9| JONES, A.K. et alii - "StarOs, A Multiprocessor Operating System for the Support of Task Forces".

In: Proceedings of the 7th Symposium on Operating Systems Principles, California, p.117-127, Dez. 1979.

|10| KERNIGHAN, B.W. - "Why Pascal is Not My Favorite Programming Language". Bell Laboratories, Jul. 1981.

|11| KERNIGHAN, B.W. & MASHEY, J.R. - "The Unix Programming Environment". Software-Practice and Experience, 9(1): 1-15, 1979.

|12| KRAMER, J. et alii - "The CONIC Programming Language". Version 2.4. Imperial College Research Report DOC 84/19, Out. 1984, 21 p.

|13| KRAMER, J. et alii - "CONIC: An Integrated Approach to Distributed Computer Control Systems". IEE Proc., 130(1): 1-10, Jan. 1983.

|14| LAGES, N. dos S. - "Um Gerador Automático de "Stubs" para Chamada Remota de Procedimentos num Ambiente de Programação Distribuída Baseado em Modula-2". Tese de Mestrado, em andamento, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ.

|15| LAUER, H.C. & NEEDHAM, R.M. - "On the Duality of Operating Systems Structures". In: Proc. of 2nd Int. Symposium on Operating Systems. IRIA, out. 1978, reimpresso: ACM Operating System Review, 13(2): 3-19, Abr. 1979.

|16| LAUER, H.C. & SATTERTHWAITE, E.H. - "The Impact of MESA on System Design", IEEE Ch1479-5/79/0000-0174, p. 174-182, 1979.

|17| LI, C. & C. LIU, M. - "Dislang: A Distributed Programming Language/System". IEEE 162-172, 1981.

|18| LOGITECH, - "Modula-2/86 User's Manual". Logitech, Inc., USA, 1985.

|19| MAGEE, J.N. - "Provision of Flexibility in Distributed Systems". Ph.D. Thesis, Department of Computing, Imperial College, London, Abr. 1984, 135 p.

|20| MAGEE, J., KRAMER, J. & SLOMAN, M. - "The CONIC Support Environment for Distributed Systems". NATO Advanced Study Institute, Distributed Operating Systems: Theory and Practice, Izmir, Turkey, Ago. 1986, 19 p.

|21| MAGEE, J., KRAMER, J. & SLOMAN, M. - "Constructing Distributed Systems in CONIC". Imperial College, Department of Computing, Research Report DOC 97/4, Dept. of Computing, Mar. 1987, 26 p.

|22| MAGEE, J. N. & KRAMER, J. - "DYNAMIC SYSTEMS". CONFIGURATION FOR DISTRIBUTED RAL-TIME SYSTEMS". Department of Computing, Imperial College, London, Mar. 1983, 18 p.

|23| MITCHELL, J.G. et alii - "Mesa Language Manual". Report CSL - 78-1, Xerox Parc, Palo Alto, California, 1978, 189p.

|24| NELSON, B.J. - "Remote Procedure Call". Ph. D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, Mai. 1981, 201 p. (Tech.Rep.CMU-CS-81-119).

|25| OGILVIE, J.W. L. - "Modula-2 Programming". Mcgraw-Hill Inc., USA, 1985, 304 p.

|26| OUSTERHOUT, J.K., SCELZA, D.A. & SINDHU, P.S. - "Medusa: An Experiment in Distributed Operating Structure". Communications of the ACM, 23(2): 92-

104, Fev. 1980.

|27| RITCHIE, D.M. & THOMPSON, K. - "The Unix Time-Sharing System". Communications of the ACM, 17(7): 365-375, 1974.

|28| RODRIGUEZ, N. - "Um Sistema Operacional Dedicado a Modula-2 para um Microcomputador de 8 bits". Tese de Mestrado, Departamento de Informática, PUC/RJ, Abr. 1986.

|29| SCOTT, M.L. - "Messages vs. Remote Procedures is a False Dichomoty". ACM Sigplan Notices, 18(5): 57-61, Mai. 1983.

|30| SEGRE, L.M. - "Um Estudo de Ambientes e Programação Distribuída: Proposta de Extensão para Modula-2". Tese de Doutorado, Programa de Engenharia de Sistemas Computação, COPPE/UFRJ, Dez. 1987, 338 p.

|31| SEGRE, L. & STANTON, M. - "Análise de Programação Concorrente no Contexto de Sistemas Distribuídos". In: Anais do II Congresso Latino-Iberoamericano de Investigacion Operativa e Ingenieria de Sistemas, Buenos Aires, Argentina, Ago. 1984, p. 150-164.

|32| SEGRE, L. & STANTON, M. - "Modula-2: Suporte para o Desenvolvimento de Software Concorrente". In: Anais do XVII Congresso Nacional de Informática, Nov. 1984.

|33| SEGRE, L. & STANTON, M. - "A Linguagem Modula-2 e seu Ambiente de Suporte de Programação". In: Anais do XVII Congresso Nacional de Informática, Nov. 1984.

|34| SEGRE, L.M. & STANTON., M.A. - "A Construção de Software Distribuído Usando Modula-2: Paralelismo,

Comunicação e Confiabilidade". Artigo apresentado no III Congresso Latino-Iberoamericano e Investigacion Operativa de Ingenieria de Sistemas, Santiago de Chile, Ago. 1986, 8 p.

|35| SEGRE, L.M. & STANTON, M.A. - "A Construção de Software Distribuído Usando Modula-2: Linguagem de Configuração". Artigo apresentado no III Congresso Latino-Iberoamericano de Investigacion Operativa e Ingenieria de Sistemas, Santiago de Chile, Ago. 1986, 8 p.

|36| SIMPSON, H.R. - "The Mascot Method". Software Engineering Journal, 1(3): 103-120, Mai. 1986.

|37| SLOMAN, M., MAGEE, J. & KRAMER, J. - "Building Flexible Distributed Systems in CONIC". In: Proc. SERC Distributed Computing 84 Conf., University of Sussex, Brighton, Set. 1984, 21 p.

|38| SLOMAN, M.; KRAMER, J. & MAGEE, J. - "THE CONIC TOOLKIT FOR BUILDING DISTRIBUTED SYSTEMS". 6th IFAC Distributed Computer Control System Workshop, Monterey, California, USA, Maio 1985, 10 p.

|39| SOLOMON, M.H. & FINKEL, R.A. - "The Roscoe Distributed Operating System". In: ACM Sigops Proceedings of the 7th Symposium on Operating Systems Principles, California, p. 108-114, Dez. 1979.

|40| STANTON, M.A. et alii - "Uma Experiência na Montagem de um Ambiente de Suporte de Programação em Modula-2". Anais do XIII SEMISH e VI Congresso da Sociedade Brasileira de Computação, Recife, Jul. 1986, p. 454-464.

|41| STEVENS, W.P., MYERS, G.F. & CONSTANTINE, L.C.

- "Structured Design". IBM Systems Journal, 13(2): 115-139, 1974.

|42| WIRTH, N. - "The Module: A System Structuring Facility in High-Level Programming Languages". In: Proceedings of the Symposium on Language Design and Programming Methodology. Lectures Notes in Computer Science, no. 79, Springer Verlag, 1980.

|43| WIRTH, N. "Programming in Modula-2". Berlim, Springer-Verlag, 1982, 176 p.

ANEXO I


```
 1  0040  IMPLEMENTATION MODULE  INTERPRET;
 2  0040
 3  0040  (*
 4  0040  Title    : INTERPRET - Interpretador
 5  0040  LastEdit: 0 - 10/11/87
 6  0040  Author   : Vanise Paraiso Vetromille
 7  0040  System   : LOGITECH MODULA-2/86
 8  0040  *)
 9  0040
10  0040  FROM SYSTEM IMPORT  ADDRESS, WORD, BYTE,
11  0040                                    ADR ;
12  0040
13  0040  FROM Storage IMPORT  ALLOCATE, DEALLOCATE ;
14  0040
15  0040  FROM Strings IMPORT  CompareStr ;
16  0040
17  0040  FROM Terminal  IMPORT  Read ;
18  0040
19  0040  (* FROM FileNames IMPORT *)
20  0040
21  0040  FROM FileSystem  IMPORT  File, Response,
                          Command, Flag, FlagSet,
22  0040                                MediumType,
23  0040
24  0040    (* procedures de operacoes em arquivos *)
25  0040              Create, Close, Lookup, ReadChar,
26  0040        SetRead, SetOpen, SetPos, GetPos, Again ;
27  0040
28  0040  CONST EOL = 36C ;
29  0040
30  0040  TYPE ArrayOfCharType = ARRAY [0..39] OF CHAR ;
31  0040
32  0040  TYPE  PointerModuloImport  =  POINTER TO
                                     ModImportType ;
33  0040
34  0040  TYPE  ModImportType  =  RECORD
35  0040                             ModImport :
                                      ArrayOfCharType ;
36  0040                             TipoInterf :
                                      ArrayOfCharType ;
37  0040                             ProxModImport :
                                   PointerModuloImport ;
38  0040                          END (* record *) ;
39  0040
40  0040
41  0040
42  0040  TYPE PointerModuloExport  =  POINTER TO
                                     ModExportType  ;
43  0040
```

```
44  0040  TYPE ModExportType  =  RECORD
45  0040                            ModExport :
                                       ArrayOfCharType ;
46  0040                            TipoImpl :
                                       ArrayOfCharType ;
47  0040                            ProxModExport :
                                    PointerModuloExport ;
48  0040                         END (* record *) ;
49  0040
50  0040
51  0040
52  0040  TYPE PointerEstacoesLogicas  =  POINTER TO
                                    EstacaoLogicaType ;
53  0040
```

```
54  0040  TYPE EstacaoLogicaType  =  RECORD
55  0040                               Estacoes :
                                       ArrayOfCharType ;
56  0040                               ProxEstacao :
                                 PointerEstacoesLogicas ;
57  0040                             END (* record *) ;
58  0040
59  0040
60  0040
61  0040  TYPE PointerModulosCriados  =  POINTER TO
                                        ModCriadoType ;
62  0040
63  0040  TYPE ModCriadoType  =  RECORD
64  0040                            NomeModulo :
                                      ArrayOfCharType ;
65  0040                            Tipo :
                                      ArrayOfCharType ;
66  0040                            IndTabConfig :
                                            CARDINAL ;
67  0040                            Estacao :
                                PointerEstacoesLogicas ;
68  0040                            NumEstacao :
                                            CARDINAL ;
69  0040                            ProxModCriado :
                                 PointerModulosCriados ;
70  0040                         END (* record *) ;
71  0040
72  0040
73  0040
74  0040  TYPE PointerInstancias  =  POINTER TO
                                          Instancias ;
75  0040
76  0040  TYPE Instancias  =  RECORD
77  0040                        NomeInst :
                                       ArrayOfCharType ;
78  0040                        ProxInst :
                                     PointerInstancias ;
```

```
 79  0040                         END (* record *) ;
 80  0040
 81  0040
 82  0040
 83  0040  TYPE PointerInstanciasConfigType  =  POINTER TO
                                   InstanciasConfigType ;
 84  0040
 85  0040  TYPE InstanciasConfigType  =  RECORD
 86  0040                                  TipoInstConfig :
                                         ArrayOfCharType ;
 87  0040                                  Instancias :
                                       PointerInstancias ;
 88  0040                                  ProxInstConfig :
                              PointerInstanciasConfigType ;
 89  0040                               END (* record *) ;
 90  0040
 91  0040
 92  0040
 93  0040  TYPE PointerInstConfig  =  POINTER TO
                                         InstConfigType ;
 94  0040
 95  0040  TYPE InstConfigType  =  RECORD
 96  0040                            NomeInstConfig :
                                         ArrayOfCharType ;
 97  0040                            InstModImpl :
                                       PointerInstancias ;
 98  0040                            ProxInstConfig :
                                       PointerInstConfig ;
 99  0040                         END (* record *) ;
100  0040
101  0040
102  0040
103  0040  TYPE PointerListaTipoConfig  =  POINTER TO
                                   ListaTipoConfigType ;
104  0040
105  0040  TYPE ListaTipoConfigType  =  RECORD
106  0040                               NomeConfig :
                                        ArrayOfCharType ;
107  0040                               InstanciasConfig :
                                         InstConfigType ;
108  0040                               ProxTipoConfig :
                                  PointerListaTipoConfig ;
109  0040                               END (* record *) ;
```



```
110  0040
111  0040
112  0040
113  0040  TYPE TabConfigType  =  RECORD
114  0040                          NomeConfig :
                                         ArrayOfCharType ;
115  0040                          EstacoesLogicas :
                                 PointerEstacoesLogicas ;
```

```
116  0040                                   ModulosImportados :
                                            PointerModuloImport ;
117  0040                                   ModulosExportados :
                                            PointerModuloExport ;
118  0040                                   NumEstacao    :
                                                       CARDINAL ;
119  0040                                   NumModImport :
                                                       CARDINAL ;
120  0040                                   NumModExport :
                                                       CARDINAL ;
121  0040                                   ModulosCriados :
                                          PointerModulosCriados ;
122  0040                                 END (* record *) ;
123  0040
124  0040
125  0040
126  0040  TYPE TabTipoConfigType  =  RECORD
127  0040                                   NomeConfig :
                                             ArrayOfCharType ;
128  0040                                   NivelHierarq :
                                                       CARDINAL ;
129  0040                                   InstanciasConfig :
                                  PointerInstanciasConfigType ;
130  0040                                   END (* record *) ;
131  0040
132  0040
133  0040
134  0040  TYPE TabEstLogicaType  =  RECORD
135  0040                                   NomeEstLogica :
                                             ArrayOfCharType ;
136  0040                                   NomeEstFisica :
                                             ArrayOfCharType ;
137  0040                                   SubConfig :
                                          ListaTipoConfigType ;
138  0040                                 END (* record *) ;
139  0040
140  0040
141  0040
142  0040  TYPE PointerTipoConfig  =  POINTER TO
                                              TipoConfigType ;
143  0040
144  0040  TYPE TipoConfigType  =  RECORD
145  0040                                   NomeTipo :
                                             ArrayOfCharType ;
146  0040                                   NivelHierarq :
                                                       CARDINAL ;
147  0040                                   ProxTipo :
                                            PointerTipoConfig ;
148  0040                                 END ;
149  0040
150  0040          (* Decalaracao das variaveis internas *)
                   (* ao programa                        *)
151  0040
152  0040  VAR  TabelaConfig : ARRAY [0..255] OF
                                                TabConfigType ;
153  0040
154  0040  VAR  TabelaTiposConfig : ARRAY [0..255] OF
```

```
                                                   TabTipoConfigType ;
155  0040
156  0040  VAR  TabelaEstacoes : ARRAY [0..39] OF
                                                    TabEstLogicaType ;
157  0040
```

```
158  0040  VAR  TabTipos : ARRAY [0..255] OF
                                                     TipoConfigType ;
159  0040
160  0040  VAR  arq : File ;
161  0040
162  0040  VAR  Erro : BOOLEAN ;
163  0040
164  0040  VAR  Carac : CHAR ;
165  0040
166  0040  VAR  I : CARDINAL ;
167  0040
168  0040  VAR  Nome : ArrayOfCharType ;
169  0040
170  0040  VAR  Igual : INTEGER ;
171  0040
172  0040  VAR  NaoImporta : BOOLEAN ;
173  0040
174  0040  VAR  NaoAchouLnk : BOOLEAN ;
175  0040
176  0040  VAR  PtrMod0 : PointerModulosCriados ;
177  0040
178  0040  VAR  PtrMod1 : PointerModulosCriados ;
179  0040
180  0040  VAR  PtrExp0 : PointerModuloExport ;
181  0040
182  0040  VAR  PtrExp1 : PointerModuloExport ;
183  0040
184  0040  VAR  InstCriada : BOOLEAN ;
185  0040
186  0040  VAR  NaoExporta : BOOLEAN ;
187  0040
188  0040  VAR  ParteAlta : CARDINAL ;
189  0040
190  0040  VAR  ParteBaixa : CARDINAL ;
191  0040
192  0040  VAR  AindaExisteEstacao : BOOLEAN ;
193  0040
194  0040  VAR  NivelHierarq : CARDINAL ;
195  0040
196  0040  VAR  IndTipoConfig : CARDINAL ;
197  0040
198  0040  VAR  IndConfig : CARDINAL ;
199  0040
200  0040
```

```
201  0040        (* Declaracoes das procedures internas *)
                 (* ao programa                         *)
202  0040
203  0040
204  0040   PROCEDURE  MontaConfig ;
205  0040
206  0040     VAR  Carac : CHAR ;
207  0040
208  0040     VAR  NomeConfig : ArrayOfCharType ;
209  0040
210  0040     VAR  I : CARDINAL ;
211  0040
212  0040
```



```
213  0040   BEGIN
214  004D
215  004D      REPEAT (* Percorre a palavra CONFIGURATION
                         como tambem os brancos que
216  004D                a sucedem ate' encontrar ':'    *)
217  004D         ReadChar (arq, Carac) ;
218  0062      UNTIL  Carac = ':'  ;
219  0062
220  0065      REPEAT
221  006B         ReadChar (arq, Carac) ;
222  0080      UNTIL  Carac <> ' '  ;
223  0080
224  0089      I := 0 ;
225  0089
226  0089      REPEAT (* Le o nome da configuracao *)
227  0091         I := I + 1 ;
228  009A         ReadChar (arq, Carac) ;
229  00AD      UNTIL ((Carac = ' ')  OR  (Carac = ':')  OR
                     (Carac = EOL)) ;
230  00C5
231  00C5      TabelaConfig [IndConfig]. NomeConfig  :=
                                             NomeConfig ;
232  00F4
233  00F4      TabelaTiposConfig [IndTipoConfig].
                                 NomeConfig := NomeConfig ;
234  0120
235  0120      TabelaTiposConfig [IndTipoConfig].
                                 InstanciasConfig := NIL ;
236  0151
237  0151      TabelaTiposConfig [IndTipoConfig].
                          InstanciasConfig^. Instancias := NIL ;
238  0191
239  0191   END MontaConfig ;
240  019D
241  019D
242  019D   PROCEDURE  EliminaLinhaEmBranco ;
243  019D
244  019D   BEGIN
```

```
245  01A7
246  01A7     REPEAT
247  01A7        REPEAT (* Elimina os brancos ate'
                           encontrar um caractere *)
248  01A7           ReadChar (arq, Carac) ;
249  01BB        UNTIL Carac <> ' ' ;
250  01BB
251  01C9     UNTIL  Carac <> EOL  ;  (* Se  o   caractere
                                         for EOL pula esta
                                         linha           *)
252  01C9
253  01CC     Again (arq) ; (* Coloca  o  caractere,  que
                                 nao  e' nem igual a branco
254  01D7                        nem EOL,   de   volta   no
                                 arquivo para  que ele seja
255  01D7                        lido novamente           *)
256  01D7
257  01D7  END EliminaLinhaEmBranco ;
258  01E0
259  01E0
260  01E0  PROCEDURE  AnalisaUse (VAR TabTipos : ARRAY OF
                                        TipoConfigType ;
261  01E0                          VAR NivelHierarq :
                                              CARDINAL) ;
262  01E0
263  01E0     VAR  I : CARDINAL ;
264  01E0
```



```
265  01E0     VAR  Tipo : ArrayOfCharType ;
266  01E0
267  01E0     VAR  NaoAchou : BOOLEAN ;
268  01E0
269  01E0     VAR  PtrTipo0 : PointerTipoConfig ;
270  01E0
271  01E0     VAR  PtrTipo1 : PointerTipoConfig ;
272  01E0
273  01E0  BEGIN
274  01ED
275  01ED     ReadChar (arq, Carac) ; (* Le o caractere
                                        'U' da palavra
                                        USE             *)
276  01FC
277  01FC     ReadChar (arq, Carac) ; (* Le o caractere
                                        'S' da palavra
                                        USE             *)
278  0210
279  0210     ReadChar (arq, Carac) ; (* Le o caractere
                                        'E' da palavra
                                        USE             *)
280  0224
281  0224     REPEAT  (* Elimina os brancos que sucedem a
                         palavra USE                      *)
```

```
 282  0229         ReadChar (arq, Carac) ;
 283  023D      UNTIL  Carac <> ' ' ;
 284  023D
 285  024B      IF Carac = EOL
 286  024B      THEN
 287  0253          EliminaLinhaEmBranco ;
 288  0257          ReadChar (arq, Carac) ;
 289  026A      END ;
 290  026F
 291  0273      FOR I := 0 TO 39
 292  0273      DO
 293  0273        Tipo [I] := ' ' ;
 294  0290      END ;
 295  0295
 296  029B      I := 0 ;
 297  029B
 298  029B      REPEAT  (* Le o nome do tipo *)
 299  02A0         Tipo [I] := Carac ;
 300  02C1         I := I + 1 ;
 301  02CA         ReadChar (arq, Carac) ;
 302  02D7      UNTIL  ((Carac = ' ')  OR  (Carac = ',')  OR
 303  02FE            (Carac = ';')  OR  (Carac = EOL)) ;
 304  030A
 305  030E      IF Carac = ' '   (* Elimina os brancos que
                                   sucedem o nome do tipo *)
 306  030E      THEN
 307  0316        REPEAT
 308  031A           ReadChar (arq, Carac) ;
 309  032E        UNTIL  Carac <> ' ' ;
 310  0336      END ;
 311  033C
 312  033C      IF Carac = EOL
 313  033C      THEN
 314  0344          EliminaLinhaEmBranco ;
 315  0348          ReadChar (arq, Carac) ;
 316  035B      END ;
 317  0360
```



```
 318  0360      NivelHierarq := 1 ;
 319  0360
 320  0368      NaoAchou := TRUE ;
 321  0368
 322  036C      I := 0 ;
 323  036C
 324  036C      WHILE  ((I <= 255)  OR  NaoAchou) DO
 325  03B3        Igual := CompareStr (Tipo,
                          TabelaTiposConfig [I]. NomeConfig) ;
 326  03B3
 327  03C1        IF  Igual  =  0
 328  03C1        THEN
 329  03CE            NaoAchou := FALSE ;
 330  03CE
```

```
331   03CE            IF    NivelHierarq < TabelaTiposConfig
                                                [I]. NivelHierarq
332   03FC            THEN
333   0422                NivelHierarq := TabelaTiposConfig
                                             [I]. NivelHierarq ;
334   0426            END ;
335   042C        ELSE
336   042F            NivelHierarq := 0 ;
337   042F        END ;
338   0437
339   043A        I := I + 1 ;
340   043A
341   0443      END (* while *) ;
342   044E
343   044E      TabTipos [IndConfig]. NomeTipo := Tipo ;
344   047C
345   047C      TabTipos [IndConfig]. NivelHierarq :=
                                                 NivelHierarq ;
346   04B2
347   04B2      NEW (PtrTipo0) ;
348   04C0
349   04C0      TabTipos [IndConfig]. ProxTipo := PtrTipo0 ;
350   04EB
351   04F1      WHILE  Carac <> ';'  DO
352   04F1
353   04F9        REPEAT  (* Elimina os brancos que sucedem
                             a virgula                      *)
354   04FD           ReadChar (arq, Carac) ;
355   0511        UNTIL  Carac <> ' ' ;
356   0511
357   051F        IF Carac = EOL
358   051F        THEN
359   0527            EliminaLinhaEmBranco ;
360   052B            ReadChar (arq, Carac) ;
361   053E        END ;
362   0543
363   0547        FOR I := 0 TO 39
364   0547        DO
365   0547           Tipo [I] := ' ' ;
366   0564        END ;
367   0569
368   056F        I := 0 ;
369   056F
370   056F        REPEAT  (* Le o nome do tipo *)
371   0574           Tipo [I] := Carac ;
372   0595           I := I + 1 ;
```



```
373   059E           ReadChar (arq, Carac) ;
374   05AB        UNTIL  ((Carac = ' ')  OR  (Carac = ',')
                          OR  (Carac = ';')) ;
375   05D2
```

```
376  05D6        IF Carac = ' '(* Elimina os brancos que
                                   sucedem o nome do tipo *)
377  05D6        THEN
378  05DE          REPEAT
379  05E2             ReadChar (arq, Carac) ;
380  05F6          UNTIL  Carac <> ' ' ;
381  05FE        END ;
382  0604
383  0604        NivelHierarq := 1 ;
384  0604
385  060C        NaoAchou := TRUE ;
386  060C
387  0610        I := 0 ;
388  0610
389  0610        WHILE  ((I <= 255)  OR  NaoAchou) DO
390  0657          Igual := CompareStr (Tipo,
                       TabelaTiposConfig [I]. NomeConfig) ;
391  0657
392  0665          IF  Igual  =  0
393  0665          THEN
394  0672              NaoAchou := FALSE ;
395  0672
396  0672              IF NivelHierarq  <
                        TabelaTiposConfig [I]. NivelHierarq
397  06A0              THEN
398  06C6                  NivelHierarq :=
                       TabelaTiposConfig [I]. NivelHierarq;
399  06CA              END ;
400  06D0          END ;
401  06D0
402  06D3          I := I + 1 ;
403  06D3
404  06DC        END (* while *) ;
405  06E7
406  06E7        NEW (PtrTipo1) ;
407  06F1
408  06F1        PtrTipo1^. NomeTipo := Tipo ;
409  070D
410  070D        PtrTipo1^. NivelHierarq := NivelHierarq ;
411. 0726
412  0726        PtrTipo1^. ProxTipo := NIL ;
413  073D
414  073D        PtrTipo0^. ProxTipo := PtrTipo1 ;
415  0755
416  075B        PtrTipo0 := PtrTipo1 ;
417  075B
418  075E     END (* while *) ;
419  076C
420  076C  END AnalisaUse ;
421  0772
422  0772
```

```
423  0772  PROCEDURE  MontaEstacoes ;
424  0772
425  0772    VAR  NomeEstacao : ArrayOfCharType ;
426  0772
427  0772    VAR  Carac : CHAR ;
428  0772
429  0772    VAR  PtrEstacao1 : PointerEstacoesLogicas ;
430  0772
431  0772    VAR  PtrEstacao2 : PointerEstacoesLogicas ;
432  0772
433  0772    VAR  ContaEstacao : CARDINAL ;
434  0772
435  0772
436  0772  BEGIN
437  077F
438  077F     REPEAT  (* Percorre a palavra STATIONS *)
439  077F        ReadChar (arq, Carac) ;
440  0794     UNTIL Carac = ' ' ;
441  0794
442  0797     REPEAT (* Elimina os brancos que sucedem a
                          palavra STATIONS                  *)
443  079D        ReadChar (arq, Carac) ;
444  07B2     UNTIL Carac <> ' ' ;
445  07B2
446  07BB     IF Carac = EOL
447  07BB     THEN
448  07BE         EliminaLinhaEmBranco ;
449  07C2         ReadChar (arq, Carac) ;
450  07D6     END ;
451  07DB
452  07DB     I := 0 ;
453  07DB
454  07E9     FOR I:= 0 TO 39
455  07E9     DO
456  07E9       NomeEstacao [I] := ' ' ;
457  0809     END ;
458  080E
459  0813     I := 0 ;
460  0813
461  0813     REPEAT  (* Le o nome da estacao *)
462  081E        NomeEstacao [I] := Carac ;
463  0841        I := I + 1 ;
464  084A        ReadChar (arq, Carac) ;
465  0859     UNTIL  ((Carac = ',')  OR  (Carac = ';')  OR
466  0871            (Carac = ' ')  OR  (Carac = EOL)) ;
467  0878
468  087C     IF  Carac = ' '  (* Elimina os brancos que
                                 sucedem o nome da estacao *)
469  087C     THEN
470  087F       REPEAT
471  0883          ReadChar (arq, Carac) ;
```

```
472  0898      UNTIL Carac <> ' ' ;
473  089B    END ;
474  08A1
475  08A1    IF Carac = EOL
476  08A1    THEN
477  08A4        EliminaLinhaEmBranco ;
478  08A8        ReadChar (arq, Carac) ;
479  08BC    END ;
480  08C1
```

```
481  08C1    NEW (PtrEstacao1) ;
482  08CB
483  08CB    PtrEstacao1^. Estacoes := NomeEstacao ;
484  08E7
485  08E7    PtrEstacao1^. ProxEstacao := NIL ;
486  0900
487  0900    TabelaConfig [IndConfig]. EstacoesLogicas :=
                                                PtrEstacao1 ;
488  092E
489  0934    ContaEstacao := 1 ;
490  0934
491  0939    WHILE  Carac <> ':' DO  (* Existem   mais
                                          estacoes logicas *)
492  0939
493  0943      ContaEstacao := ContaEstacao + 1 ;
494  0943
495  094C      REPEAT  (* Elimina os brancos que sucedem
                          a virgula *)
496  094F         ReadChar (arq, Carac) ;
497  0964      UNTIL Carac <> ' ' ;
498  0964
499  096D      IF Carac = EOL
500  096D      THEN
501  0970         EliminaLinhaEmBranco ;
502  0974         ReadChar (arq, Carac) ;
503  0988      END ;
504  098D
505  0995      FOR I:= 0 TO 39
506  0995      DO
507  0995        NomeEstacao [I] := ' ' ;
508  09B5      END ;
509  09BA
510  09BF      I := 0 ;
511  09BF
512  09BF      REPEAT (* Le o nome da proxima estacao *)
513  09CA         NomeEstacao [I] := Carac ;
514  09ED         I := I + 1 ;
515  09F6         ReadChar (arq, Carac) ;
516  0A05      UNTIL  ((Carac = ',')  OR  (Carac = ';')
                        OR  (Carac = ' ')
517  0A1D               OR  (Carac = EOL)) ;
518  0A24
```

```
519  0A28       IF  Carac = ' '   (* Elimina os brancos que
                                     sucedem  o   nome   da
                                     estacao              *)
520  0A28       THEN
521  0A2B          REPEAT
522  0A2F             ReadChar (arq, Carac) ;
523  0A44          UNTIL Carac <> ' ' ;
524  0A47       END ;
525  0A4D
526  0A4D       IF Carac = EOL
527  0A4D       THEN
528  0A50          EliminaLinhaEmBranco ;
529  0A54          ReadChar (arq, Carac) ;
530  0A68       END ;
531  0A6D
532  0A6D       NEW (PtrEstacao2) ;   (* Insere  a   nova
                                         estacao na lista
                                         de estacoes      *)
533  0A77
534  0A77       PtrEstacao2^. Estacoes := NomeEstacao ;
```

Modula-2/86   INTERPRE.MOD   Page   11

```
535  0A93
536  0A93       PtrEstacao2^. ProxEstacao := NIL ;
537  0AAC
538  0AAC       PtrEstacao1^. ProxEstacao := PtrEstacao2 ;
539  0AC4
540  0ACA       PtrEstacao1 := PtrEstacao2 ;
541  0ACA
542  0ACD     END (* while *) ;
543  0ADB
544  0ADB     TabelaConfig [IndConfig]. NumEstacao :=
                                            ContaEstacao ;
545  0B03
546  0B03  END MontaEstacoes ;
547  0B0A
548  0B0A
549  0B0A  PROCEDURE  AnalisaImport ;
550  0B0A
551  0B0A     VAR  ModImport : ArrayOfCharType ;
552  0B0A
553  0B0A     VAR  Tipo : ArrayOfCharType ;
554  0B0A
555  0B0A     VAR  PtrImp1 : PointerModuloImport ;
556  0B0A
557  0B0A     VAR  PtrImp2 : PointerModuloImport ;
558  0B0A
559  0B0A     VAR  ContaImp : CARDINAL ;
560  0B0A
561  0B0A  BEGIN
562  0B17
563  0B17     REPEAT   (* Elimina os brancos que sucedem a
                          palavra IMPORT                  *)
```

```
564  0B17        ReadChar (arq, Carac) ;
565  0B2B     UNTIL Carac <> ' ' ;
566  0B2B
567  0B39     IF Carac = EOL
568  0B39     THEN
569  0B41         EliminaLinhaEmBranco ;
570  0B45         ReadChar (arq, Carac) ;
571  0B58     END ;
572  0B5D
573  0B5D     I := 0 ;
574  0B5D
575  0B68     ContaImp := 1 ;
576  0B68
577  0B68     REPEAT
578  0B6D        ModImport [I] := Carac ;
579  0B90        I := I + 1 ;
580  0B99        ReadChar (arq, Carac) ;
581  0BA7     UNTIL  ((Carac = ' ')  OR  (Carac = ':')  OR
                        (Carac = EOL)) ;
582  0BCE
583  0BD2     IF Carac = ' '      (* Elimina os brancos que
                                      sucedem   o  nome da
                                      interface de  modulo *)
584  0BD2     THEN                (* que    esta'   sendo
                                      importada        ate'
                                      encontrar ':' ou EOL *)
585  0BDA         REPEAT
586  0BDE            ReadChar (arq, Carac) ;
587  0BF2         UNTIL  Carac <> ' ' ;
```



```
588  0BFA     END ;
589  0C00
590  0C00     REPEAT (* Elimina os brancos que
                           sucedem ':'             *)
591  0C00        ReadChar (arq, Carac) ;
592  0C14     UNTIL  Carac <> ' ' ;
593  0C14
594  0C22     IF Carac = EOL
595  0C22     THEN
596  0C2A         EliminaLinhaEmBranco ;
597  0C2E         ReadChar (arq, Carac) ;
598  0C41     END ;
599  0C46
600  0C46     I := 0 ;
601  0C46
602  0C46     REPEAT (* Le o nome da interface de
                           modulo *)
603  0C51        Tipo [I] := Carac ;
604  0C74        I := I + 1 ;
605  0C7D        ReadChar (arq, Carac) ;
606  0C8B     UNTIL  ((Carac = ',')  OR  (Carac = ';')  OR
607  0CB2              (Carac = ' ') OR  (Carac = EOL)) ;
```

```
608  0CBE
609  0CC2      IF Carac = ' '   (* Elimina os brancos que
                                   sucedem  o  tipo   da
                                   interface              *)
610  0CC2      THEN
611  0CCA          REPEAT
612  0CCE              ReadChar (arq, Carac) ;
613  0CE2          UNTIL  Carac <> ' ' ;
614  0CEA      END ;
615  0CF0
616  0CF0      REPEAT    (* Elimina os brancos que
                             sucedem ':'          *)
617  0CF0         ReadChar (arq, Carac) ;
618  0D04      UNTIL  Carac <> ' ' ;
619  0D04
620  0D12      IF Carac = EOL
621  0D12      THEN
622  0D1A          EliminaLinhaEmBranco ;
623  0D1E          ReadChar (arq, Carac) ;
624  0D31      END ;
625  0D36
626  0D36      NEW (PtrImp1) ;
627  0D40
628  0D40      PtrImp1^. ModImport := ModImport ;
629  0D5C
630  0D5C      PtrImp1^. TipoInterf := Tipo ;
631  0D79
632  0D79      PtrImp1^. ProxModImport := NIL ;
633  0D95
634  0D95      TabelaConfig [IndConfig]. ModulosImportados
                                              := PtrImp1 ;
635  0DC3
636  0DC9      WHILE  Carac <> ';'  DO (* Existem  mais
                                          interfaces de
                                          modulos
                                          importadas    *)
637  0DC9
638  0DD8        ContaImp :=  ContaImp + 1 ;
639  0DD8
640  0DE1        REPEAT    (* Elimina os brancos que sucedem
                              a virgula *)
641  0DE4           ReadChar (arq, Carac) ;
642  0DF8        UNTIL Carac <> ' ' ;
```

```
643  0DF8
644  0E06        IF Carac = EOL
645  0E06        THEN
646  0E0E            EliminaLinhaEmBranco ;
647  0E12            ReadChar (arq, Carac) ;
648  0E25        END ;
649  0E2A
```

```
650   0E32      FOR I:= 0 TO 39
651   0E32      DO
652   0E32        ModImport [I] := ' ' ;
653   0E52      END ;
654   0E57
655   0E5C      I := 0 ;
656   0E5C
657   0E5C      REPEAT  (* Le o nome da proxima
                           interface                 *)
658   0E67         ModImport [I] := Carac ;
659   0E8A         I := I + 1 ;
660   0E93         ReadChar (arq, Carac) ;
661   0EA1      UNTIL  ((Carac = ':')  OR  (Carac = ' ')
                        OR  (Carac = EOL)) ;

662   0EC8
663   0ECC      IF Carac = ' '   (* Elimina os brancos que
                                    sucedem  o   nome   da
                                    interface de  modulo *)
664   0ECC      THEN             (* ate' encontrar ':'  ou
                                    EOL                  *)
665   0ED4          REPEAT
666   0ED8             ReadChar (arq, Carac) ;
667   0EEC          UNTIL  Carac <> ' ' ;
668   0EF4      END ;
669   0EFA
670   0EFA      REPEAT   (* Elimina os brancos que
                            sucedem ':'              *)
671   0EFA         ReadChar (arq, Carac) ;
672   0F0E      UNTIL  Carac <> ' '  ;
673   0F0E
674   0F1C      IF Carac = EOL
675   0F1C      THEN
676   0F24          EliminaLinhaEmBranco ;
677   0F28          ReadChar (arq, Carac) ;
678   0F38      END ;
679   0F40
680   0F40      I := 0 ;
681   0F40
682   0F40      REPEAT  (* Le o nome do tipo da interface
                           de modulo importada           *)
683   0F4B         Tipo [I] := Carac ;
684   0F6E         I := I + 1 ;
685   0F77         ReadChar (arq, Carac) ;
686   0F85      UNTIL  ((Carac = ',')  OR  (Carac = ';')
                        OR  (Carac = ' ')
687   0FAC              OR  (Carac = EOL)) ;
688   0FB8
689   0FBC      IF Carac = ' '   (* Elimina os brancos que
                                    sucedem  o   tipo   da
                                    interface            *)
690   0FBC      THEN
691   0FC4          REPEAT
692   0FC8             ReadChar (arq, Carac) ;
693   0FDC          UNTIL  Carac <> ' ' ;
694   0FE4      END ;
```

```
695  0FEA
696  0FEA        REPEAT   (* Elimina os brancos que
                                 sucedem ':'              *)
697  0FEA           ReadChar (arq, Carac) ;
698  0FFE        UNTIL  Carac <> ' '  ;
699  0FFE
700  100C        IF Carac = EOL
701  100C        THEN
702  1014            EliminaLinhaEmBranco ;
703  1018            ReadChar (arq, Carac) ;
704  102B        END ;
705  1030
706  1030        NEW (PtrImp2) ;
707  103A
708  103A        PtrImp2^. ModImport := ModImport ;
709  1056
710  1056        PtrImp2^. TipoInterf := Tipo ;
711  1073
712  1073        PtrImp2^. ProxModImport := NIL ;
713  108F
714  108F        PtrImp1^. ProxModImport := PtrImp2 ;
715  10A7
716  10AD        PtrImp1 := PtrImp2 ;
717  10AD
718  10B0      END (* while *) ;
719  10BE
720  10BE      TabelaConfig [IndConfig]. NumModImport :=
                                                     ContaImp ;
721  10E6
722  10E6   END AnalisaImport ;
723  10ED
724  10ED
725  10ED   PROCEDURE  AnalisaExport ;
726  10ED
727  10ED      VAR  ModExport : ArrayOfCharType ;
728  10ED
729  10ED      VAR  Tipo : ArrayOfCharType ;
730  10ED
731  10ED      VAR  PtrExp1 : PointerModuloExport ;
732  10ED
733  10ED      VAR  PtrExp2 : PointerModuloExport ;
734  10ED
735  10ED      VAR  ContaExp : CARDINAL ;
736  10ED
737  10ED   BEGIN
738  10FA
739  10FA      REPEAT   (* Elimina os brancos que sucedem a
                             palavra EXPORT                       *)
740  10FA         ReadChar (arq, Carac) ;
741  110E      UNTIL Carac <> ' ' ;
742  110E
```

```
743  111C       IF Carac = EOL
744  111C       THEN
745  1124           EliminaLinhaEmBranco ;
746  1128           ReadChar (arq, Carac) ;
747  113B       END ;
748  1140
749  1140       I := 0 ;
```

```
750  1140
751  114B       ContaExp := 1 ;
752  114B
753  114B       REPEAT
754  1150          ModExport [I] := Carac ;
755  1173          I := I + 1 ;
756  117C          ReadChar (arq, Carac) ;
757  118A       UNTIL  ((Carac = ' ')  OR  (Carac = ':')  OR
                        (Carac = EOL)) ;
758  11B1
759  11B5       IF Carac = ' '(* Elimina os  brancos que
                                 sucedem  o    nome   da
                                 instancia   de   modulo  *)
760  11B5       THEN            (* de  implementacao  que
                                 esta' sendo   exportada  *)
761  11BD           REPEAT      (* ate'encontrar ':' ou EOL *)
762  11C1              ReadChar (arq, Carac) ;
763  11D5           UNTIL  Carac <> ' ' ;
764  11DD       END ;
765  11E3
766  11E3       IF Carac = EOL
767  11E3       THEN
768  11EB           EliminaLinhaEmBranco ;
769  11EF           ReadChar (arq, Carac) ;
770  1202       END ;
771  1207
772  1207       I := 0 ;
773  1207
774  1207       REPEAT  (* Le o nome da instancia de modulo
                          de implementacao exportada      *)
775  1212          Tipo [I] := Carac ;
776  1235          I := I + 1 ;
777  123E          ReadChar (arq, Carac) ;
778  124C       UNTIL  ((Carac = ',')  OR  (Carac = ';')
                        OR (Carac = ' ')
779  1273               OR (Carac = EOL)) ;
780  127F
781  1283       IF Carac = ' ' (* Elimina os brancos que
                                  sucedem  o   tipo   da
                                  instancia               *)
782  1283       THEN
783  128B           REPEAT
784  128F              ReadChar (arq, Carac) ;
785  12A3           UNTIL  Carac <> ' ' ;
```

```
786  12AB     END ;
787  12B1
788  12B1     IF Carac = EOL
789  12B1     THEN
790  12B9         EliminaLinhaEmBranco ;
791  12BD         ReadChar (arq, Carac) ;
792  12D0     END ;
793  12D5
794  12D5     NEW (PtrExp1) ;
795  12DF
796  12DF     PtrExp1^. ModExport := ModExport ;
797  12FB
798  12FB     PtrExp1^. TipoImpl := Tipo ;
799  1318
800  1318     PtrExp1^. ProxModExport := NIL ;
801  1334
```

```
802  1334     TabelaConfig [IndConfig]. ModulosExportados
                                              := PtrExp1 ;
803  1362
804  1368     WHILE  Carac <> ';'  DO  (* Existem  mais
                                         instancias de
                                         modulos        *)
805  1368
806  1377       ContaExp := ContaExp + 1 ;
807  1380       REPEAT   (* Elimina os brancos que sucedem
                            a virgula                   *)
808  1383         ReadChar (arq, Carac) ;
809  1397       UNTIL Carac <> ' ' ;
810  1397
811  13A5       IF Carac = EOL
812  13A5       THEN
813  13AD           EliminaLinhaEmBranco ;
814  13B1           ReadChar (arq, Carac) ;
815  13C4       END ;
816  13C9
817  13D1       FOR I:= 0 TO 39
818  13D1       DO
819  13D1         ModExport [I] := ' ' ;
820  13F1       END ;
821  13F6
822  13FB       I := 0 ;
823  13FB
824  13FB       REPEAT  (* Le o nome da proxima instancia
                           de modulo de implementacao   *)
825  1406         ModExport [I] := Carac ;
826  1429         I := I + 1 ;
827  1432         ReadChar (arq, Carac) ;
828  1440       UNTIL  ((Carac = ':')  OR  (Carac = ' ')
                        OR  (Carac = EOL)) ;

829  1467
```

```
 830   146B      IF Carac = ' '  (* Elimina os brancos que
                                    sucedem  o  nome  da
                                    instancia de modulo  *)
 831   146B      THEN            (* de  implementacao  ate'
                                    encontrar ':' ou EOL  *)
 832   1473          REPEAT
 833   1477              ReadChar (arq, Carac) ;
 834   148B          UNTIL  Carac <> ' ' ;
 835   1493      END ;
 836   1499
 837   1499      IF Carac = EOL
 838   1499      THEN
 839   14A1          EliminaLinhaEmBranco ;
 840   14A5          ReadChar (arq, Carac) ;
 841   14B8      END ;
 842   14BD
 843   14BD      I := 0 ;
 844   14BD
 845   14BD      REPEAT  (* Le o nome do tipo da instancia
                            de modulo de implementacao   *)
 846   14C8          Tipo [I] := Carac ;
 847   14EB          I := I + 1 ;
 848   14F4          ReadChar (arq, Carac) ;
 849   1502      UNTIL  ((Carac = ',')  OR  (Carac = ';')
                         OR  (Carac = ' ')
 850   1529              OR  (Carac = EOL)) ;
 851   1535
```

```
 852   1539      IF Carac = ' '  (* Elimina os brancos que
                                    sucedem  o  tipo  da
                                    instancia            *)
 853   1539      THEN
 854   1541          REPEAT
 855   1545              ReadChar (arq, Carac) ;
 856   1559          UNTIL  Carac <> ' ' ;
 857   1561      END ;
 858   1567
 859   1567      IF Carac = EOL
 860   1567      THEN
 861   156F          EliminaLinhaEmBranco ;
 862   1573          ReadChar (arq, Carac) ;
 863   1586      END ;
 864   158B
 865   158B      NEW (PtrExp2) ;
 866   1595
 867   1595      PtrExp2^. ModExport := ModExport ;
 868   15B1
 869   15B1      PtrExp2^. TipoImpl := Tipo ;
 870   15CE
 871   15CE      PtrExp2^. ProxModExport := NIL ;
 872   15EA
 873   15EA      PtrExp1^. ProxModExport := PtrExp2 ;
```

```
874  1602
875  1608          PtrExp1 := PtrExp2 ;
876  1608
877  160B        END (* while *) ;
878  1619
879  1619        TabelaConfig [IndConfig]. NumModExport :=
                                                      ContaExp ;
880  1641
881  1641     END AnalisaExport ;
882  1648
883  1648
884  1648     PROCEDURE  AnalisaTipo (VAR Erro : BOOLEAN ;
                               VAR Tipo : ArrayOfCharType ;
885  1648                          VAR SubConfig : BOOLEAN ;
                                   NivelHierarq : CARDINAL ;
886  1648                     VAR IndTabConfig : CARDINAL) ;
887  1648
888  1648        VAR  I : CARDINAL ;
889  1648
890  1648        VAR  NaoEncontrou : BOOLEAN ;
891  1648
892  1648        VAR  NaoExisteConfig : BOOLEAN ;
893  1648
894  1648        VAR  PtrInst0 : PointerInstanciasConfigType;
895  1648
896  1648        VAR  PtrInst1 : PointerInstanciasConfigType;
897  1648
898  1648        VAR  arqtemp  : File ;
899  1648
900  1648     BEGIN
901  1655
902  1655        NaoEncontrou := TRUE ;
903  1655
904  1659        I := 0 ;
905  1659
```



```
906  165E        IF NivelHierarq = 1
907  165E        THEN
908  1661            Lookup (arqtemp, Tipo, FALSE) ;
909  167D            IF arqtemp.res = notdone
910  167D            THEN
911  1684                Erro := TRUE ;
912  1684            END ;
913  168B        ELSE
914  168E            WHILE ((I <= IndTipoConfig)  OR
                               NaoEncontrou)  DO
915  16D5              Igual := CompareStr (Tipo,
                          TabelaTiposConfig [I]. NomeConfig) ;
916  16D5
917  16E3              IF Igual = 0  (* Monta  a   tabela
                                         TabelaTiposConfig  *)
918  16E3              THEN
```

```
919   16F0                NaoEncontrou := FALSE ;
920   16F0
921   16F4                IndTabConfig := I ;
922   16F4
923   16FD                SubConfig := TRUE ;
924   16FD
925   1724                PtrInst0 := TabelaTiposConfig
                                      [IndTipoConfig].
                                      InstanciasConfig ;
926   1724
927   172E                IF PtrInst0 <> NIL
928   172E                THEN
929   173C                    NaoExisteConfig := TRUE ;
930   173C
931   173C                    WHILE  ((PtrInst0 <> NIL)  OR
                                     (NaoExisteConfig))  DO
932   1770                      Igual :=  CompareStr
                                (PtrInst0^. TipoInstConfig,
                                                     Tipo) ;
933   1770
934   177E                      IF Igual = 0
935   177E                      THEN
936   178B                          NaoExisteConfig :=
                                                    FALSE ;
937   178B                      ELSE
938   179F                          PtrInst1 := PtrInst0^.
                                           ProxInstConfig ;
939   17AB                          PtrInst0 := PtrInst1 ;
940   17AB                      END ;
941   17B1                    END  (* while *) ;
942   17B3
943   17B3                    IF NaoExisteConfig
944   17B3                    THEN
945   17B3                         NEW (PtrInst1) ;
946   17C3                         PtrInst1^.
                                   TipoInstConfig := Tipo ;
947   17D8                         PtrInst1^.
                                   ProxInstConfig := NIL ;
948   17F9                         PtrInst0 := PtrInst1 ;
949   17FC                    END ;
950   1802                ELSE
951   180A                     NEW (PtrInst1) ;
952   1814                     PtrInst1^. TipoInstConfig :=
                                                     Tipo ;
953   1829                     PtrInst1^. ProxInstConfig :=
                                                      NIL ;
954   1842                     TabelaTiposConfig [I].
                               InstanciasConfig :=
                                                 PtrInst1 ;
955   186F                END ;
956   1877
957   1877            END ;
```

```
  958  187C
  959  187F              I := I + 1 ;
  960  187F
  961  1888           END  (* while *) ;
  962  1893
  963  1893       END ;
  964  1893
  965  1893       IF NaoEncontrou
  966  1893       THEN
  967  1899            Erro := TRUE ;
  968  1899       END ;
  969  18A0
  970  18A0   END AnalisaTipo ;
  971  18A6
  972  18A6
  973  18A6   PROCEDURE  AnalisaEstacao (VAR Erro : BOOLEAN ;
                                       Tipo : ArrayOfCharType ;
  974  18A6                            PtrMod0 :
                                       PointerModulosCriados) ;
  975  18A6
  976  18A6       VAR   NaoEncontrou : BOOLEAN ;
  977  18A6
  978  18A6   BEGIN
  979  18C0
  980  18C0       NaoEncontrou := TRUE ;
  981  18C0
  982  18C4       I := 0 ;
  983  18C4
  984  18C4       WHILE  ((I <= IndConfig)  OR  NaoEncontrou)
                  DO
  985  1917         Igual := CompareStr (Tipo,
                         TabelaTiposConfig [I]. NomeConfig) ;
  986  1917
  987  1925         IF Igual = 0
  988  1925         THEN
  989  1932             NaoEncontrou := FALSE ;
  990  1941         ELSE I := I + 1 ;
  991  194A         END ;
  992  194D
  993  194D       END (* while *) ;
  994  1955
  995  1955       IF PtrMod0^. NumEstacao
  996  1962                   <>
  997  1962       TabelaConfig [I]. NumEstacao
  998  198B       THEN
  999  1990           Erro := TRUE ;
 1000  1990       END ;
 1001  1997
 1002  1997   END AnalisaEstacao ;
 1003  199D
 1004  199D
```

```
1005  199D  PROCEDURE  MontaModCriados (VAR  Carac : CHAR ;
1006  199D                             VAR  PtrMod1 :
                                    PointerModulosCriados ;
1007  199D                             VAR  NomeModulo :
                                          ArrayOfCharType ;
1008  199D                             VAR  Erro :
                                             BOOLEAN)  ;
1009  199D
1010  199D      VAR  PtrMod0 : PointerModulosCriados ;
1011  199D
```



```
1012  199D      VAR  PtrMod2 : PointerModulosCriados ;
1013  199D
1014  199D      VAR  PtrMod3 : PointerModulosCriados ;
1015  199D
1016  199D      VAR  PtrEstacao1 : PointerEstacoesLogicas ;
1017  199D
1018  199D      VAR  PtrEstacao2 : PointerEstacoesLogicas ;
1019  199D
1020  199D      VAR  PtrEstacao3 : PointerEstacoesLogicas ;
1021  199D
1022  199D      VAR  ContaInst : CARDINAL ;
1023  199D
1024  199D      VAR  I : CARDINAL ;
1025  199D
1026  199D      VAR  Estacao : ArrayOfCharType ;
1027  199D
1028  199D      VAR  Tipo : ArrayOfCharType ;
1029  199D
1030  199D      VAR  ContaEstacao : CARDINAL ;
1031  199D
1032  199D      VAR  SubConfig : BOOLEAN ;
1033  199D
1034  199D      VAR  IndTabConfig : CARDINAL ;
1035  199D
1036  199D      VAR PtrInst0 : PointerInstanciasConfigType ;
1037  199D
1038  199D      VAR PtrInst1 : PointerInstanciasConfigType ;
1039  199D
1040  199D      VAR  Ptr : PointerInstancias ;
1041  199D
1042  199D      VAR  Ptr1 : PointerInstancias ;
1043  199D
1044  199D      VAR  NaoAchouConfig : BOOLEAN ;
1045  199D
1046  199D      VAR  NaoAchou : BOOLEAN ;
1047  199D
1048  199D  BEGIN
1049  19AB
1050  19AB      PtrMod0 := PtrMod1 ;
1051  19AB
1052  19B9      ContaInst := 1 ;
```

```
1053  19B9
1054  19B9       WHILE  (Carac <> ':')  DO  (* Existem mais
                                                 modulos      *)
1055  19C6
1056  19CB         ContaInst := ContaInst + 1 ;
1057  19CB
1058  19D4         REPEAT  (* Elimina os brancos que sucedem
                                a virgula                   *)
1059  19D7           ReadChar (arq, Carac) ;
1060  19EB         UNTIL  Carac <> ' ' ;
1061  19EB
1062  19F7         IF Carac = EOL
1063  19F7         THEN
1064  19FD             EliminaLinhaEmBranco ;
1065  1A01             ReadChar (arq, Carac) ;
1066  1A14         END ;
1067  1A19
1068  1A1D         FOR I:= 0 TO 39
```



```
1069  1A1D         DO
1070  1A1D           NomeModulo [I] := ' ' ;
1071  1A3B         END ;
1072  1A3F
1073  1A45         I := 0 ;
1074  1A45
1075  1A45         REPEAT  (* Le o nome do modulo *)
1076  1A4A            NomeModulo [I] := Carac ;
1077  1A69            I := I + 1 ;
1078  1A72            ReadChar (arq, Carac) ;
1079  1A84         UNTIL  ((Carac = ' ')  OR  (Carac = ':')
                           OR  (Carac = ',')
1080  1AA5                 OR  (Carac = EOL)) ;
1081  1AAF
1082  1AB3         IF Carac = ' '
1083  1AB3         THEN
1084  1AB9             REPEAT  (* Elimina  os  brancos que
                                  sucedem o nome do modulo *)
1085  1ABD                ReadChar (arq, Carac) ;
1086  1AD1             UNTIL  Carac <> ' ' ;
1087  1AD7         END ;
1088  1ADD
1089  1ADD         IF Carac = EOL
1090  1ADD         THEN
1091  1AE3              EliminaLinhaEmBranco ;
1092  1AE7              ReadChar (arq, Carac) ;
1093  1AFA         END ;
1094  1AFF
1095  1AFF         NEW (PtrMod2) ; (* Insere o novo modulo
                                      na lista  de modulos *)
1096  1B09
1097  1B09         PtrMod2^. NomeModulo := NomeModulo ;
1098  1B1E
```

```
1099  1B1E      PtrMod2^. ProxModCriado := NIL ;
1100  1B37
1101  1B37      PtrMod1^. ProxModCriado := PtrMod2 ;
1102  1B51
1103  1B57      PtrMod1 := PtrMod2 ;
1104  1B57
1105  1B5A    END (* while *) ;
1106  1B6C
1107  1B6C    REPEAT  (* Elimina os brancos que
                         sucedem  ':'          *) ;
1108  1B6C       ReadChar (arq, Carac) ;
1109  1B80    UNTIL  Carac <> ' ' ;
1110  1B80
1111  1B8C    IF Carac = EOL
1112  1B8C    THEN
1113  1B92        EliminaLinhaEmBranco ;
1114  1B96        ReadChar (arq, Carac) ;
1115  1BA9    END ;
1116  1BAE
1117  1BAE    I := 0 ;
1118  1BAE
1119  1BAE    REPEAT  (* Le o tipo do modulo *)
1120  1BB3       Tipo [I] :=  Carac ;
1121  1BD2       I := I + 1 ;
1122  1BDB       ReadChar (arq, Carac) ;
```



```
1123  1BF2    UNTIL  Carac = ' ' ;
1124  1BF2
1125  1BF8    PtrMod0^. Tipo := Tipo ;
1126  1C16
1127  1C1F    SubConfig := FALSE ;
1128  1C1F
1129  1C23    Erro := FALSE ;
1130  1C23
1131  1C23    AnalisaTipo (Erro, Tipo, SubConfig,
                           NivelHierarq, IndTabConfig) ;
1132  1C4A
1133  1C53    IF SubConfig  AND NOT Erro
1134  1C53    THEN
1135  1C5C        PtrMod3 := PtrMod0 ;
1136  1C5C
1137  1C86        PtrInst0 := TabelaTiposConfig
                       [IndTipoConfig]. InstanciasConfig ;
1138  1C86
1139  1C90        NaoAchouConfig := TRUE ;
1140  1C90
1141  1C94        WHILE  NaoAchouConfig  DO
1142  1CB7           Igual := CompareStr (PtrInst0^.
                                  TipoInstConfig, Tipo) ;
1143  1CB7
1144  1CC5           IF Igual = 0
1145  1CC5           THEN
```

```
 1146  1CDF                 Ptr := PtrInst0^. Instancias ;
 1147  1CDF
 1148  1CE8                 IF Ptr <> NIL
 1149  1CE8                 THEN
 1150  1CE8
 1151  1CF6                     WHILE Ptr <> NIL  DO
 1152  1D11                       Ptr1 := Ptr^. ProxInst ;
 1153  1D1A                       Ptr := Ptr1 ;
 1154  1D1D                     END (* while *) ;
 1155  1D25
 1156  1D25                 ELSE
 1157  1D28                     NEW (Ptr1) ;
 1158  1D32                     Ptr1^. NomeInst := PtrMod3^.
                                                  NomeModulo ;
 1159  1D57                     PtrMod2 := PtrMod3 ;
 1160  1D6D                     PtrMod3 := PtrMod2^.
                                               ProxModCriado ;
 1161  1D70                     Ptr1^. ProxInst := NIL ;
 1162  1D89                     PtrInst0^.Instancias := Ptr1 ;
 1163  1DA1
 1164  1DA1                 END ;
 1165  1DA7
 1166  1DA7                 WHILE PtrMod3 <> NIL  DO
 1167  1DA7                   NEW (Ptr1) ;
 1168  1DBF                   Ptr1^. NomeInst := PtrMod3^.
                                                  NomeModulo ;
 1169  1DE4                   PtrMod2 := PtrMod3 ;
 1170  1DFA                   PtrMod3 := PtrMod2^.
                                               ProxModCriado ;
 1171  1DFD                   Ptr1^. ProxInst := NIL ;
 1172  1E16                   Ptr^. ProxInst := Ptr1 ;
 1173  1E34                   Ptr := Ptr1 ;
 1174  1E37                 END  (* while *) ;
 1175  1E45
 1176  1E45                 NaoAchouConfig := FALSE ;
 1177  1E45
```

```
 1178  1E45             ELSE
 1179  1E5E                 PtrInst1 := PtrInst0^.
                                          ProxInstConfig ;
 1180  1E67                 PtrInst0 := PtrInst1 ;
 1181  1E6A             END ;
 1182  1E70
 1183  1E70           END (* while *) ;
 1184  1E78
 1185  1E78       END ;
 1186  1E7D
 1187  1E7D       REPEAT (* Elimina os brancos que sucedem o
                          tipo  do  modulo  ate' encontrar
 1188  1E7D               o  caractere   'O'   da    palavra
                          ON                                *)
 1189  1E7D         ReadChar (arq, Carac) ;
```

```
1190  1E91     UNTIL  Carac <> ' ' ;
1191  1E91
1192  1E97     ReadChar (arq, Carac) ;  (* Le o caractere
                                          'N' da palavra
                                          ON           *)

1193  1EAC
1194  1EAC     REPEAT  (* Elimina os brancos que sucedem a
                          palavra ON                    *)
1195  1EB1        ReadChar (arq, Carac) ;
1196  1EC5     UNTIL  Carac <> ' ' ;
1197  1EC5
1198  1ED1     IF Carac = EOL
1199  1ED1     THEN
1200  1ED7         EliminaLinhaEmBranco ;
1201  1EDB         ReadChar (arq, Carac) ;
1202  1EEE     END ;
1203  1EF3
1204  1EF3     I := 0 ;
1205  1EF3
1206  1EFC     FOR I:= 0 TO 39
1207  1EFC     DO
1208  1EFC       Nome [I] := ' ' ;
1209  1F1C     END ;
1210  1F21
1211  1F21     REPEAT  (* Le o nome da estacao *)
1212  1F27        Estacao [I] := Carac ;
1213  1F46        I := I + 1 ;
1214  1F4F        ReadChar (arq, Carac) ;
1215  1F61     UNTIL  ((Carac = ' ')  OR  (Carac = ';')
                       OR (Carac = ',')
1216  1F82             OR  (Carac = EOL)) ;
1217  1F8C
1218  1F90     IF Carac = ' ' (* Elimina os brancos que
                                 sucedem  o  nome  da
                                 estacao              *)
1219  1F90     THEN
1220  1F96         REPEAT
1221  1F9A            ReadChar (arq, Carac) ;
1222  1FAE         UNTIL  Carac <> ' ' ;
1223  1FB4     END ;
1224  1FBA
1225  1FBA     IF Carac = EOL
1226  1FBA     THEN
1227  1FC0         EliminaLinhaEmBranco ;
1228  1FC4         ReadChar (arq, Carac) ;
1229  1FD7     END ;
1230  1FDC
```



```
1231  1FDC     (* Procura se o  nome desta estacao pertence
                  a lista de estacoes  da  configuracao,
1232  1FDC        lista esta que foi definida pelo comando
```

```
                       STATIONS                                          *)
1233  1FDC
1234  1FDC      I := 0 ;
1235  1FDC
1236  1FE1      NaoAchou := TRUE ;
1237  1FE1
1238  2005      PtrEstacao1 := TabelaConfig [IndConfig].
                                                   EstacoesLogicas ;
1239  2005
1240  2008      WHILE (NaoAchou  OR  (PtrEstacao1 <> NIL))
                DO ;
1241  203F        Igual := CompareStr (Estacao,
                                       PtrEstacao1^. Estacoes) ;
1242  203F
1243  204D        IF Igual = 0
1244  204D        THEN
1245  205A            NaoAchou := TRUE ;
1246  205A        END ;
1247  205E
1248  206B        PtrEstacao2 := PtrEstacao1^. ProxEstacao ;
1249  206B
1250  2074        PtrEstacao1 := PtrEstacao2 ;
1251  2074
1252  2077      END (* while *) ;
1253  207F
1254  207F      IF NaoAchou
1255  207F      THEN
1256  2085          Erro := TRUE ;
1257  2085      END ;
1258  208C
1259  208C      NEW (PtrEstacao1) ;
1260  2096
1261  2096      PtrEstacao1^. Estacoes := Estacao ;
1262  20B2
1263  20B2      PtrEstacao1^. ProxEstacao := NIL ;
1264  20CB
1265  20CB      PtrMod0^. Estacao := PtrEstacao1 ;
1266  20E4
1267  20EA      ContaEstacao := 1 ;
1268  20EA
1269  20EA      WHILE  (Carac <> ';') DO      (* Existem mais
                                                      estacoes  *)
1270  20F7
1271  20FC        ContaEstacao := ContaEstacao + 1 ;
1272  20FC
1273  2105        REPEAT  (* Elimina os brancos que sucedem
                             a virgula *)
1274  2108           ReadChar (arq, Carac) ;
1275  211C        UNTIL  Carac <> ' ' ;
1276  211C
1277  2128        IF Carac = EOL
1278  2128        THEN
1279  212E            EliminaLinhaEmBranco ;
1280  2132            ReadChar (arq, Carac) ;
1281  2145        END ;
1282  214A
```

```
1283  214E      FOR I:= 0 TO 39
1284  214E      DO
1285  214E        Estacao [I] := ' ' ;
```



```
1286  216B      END ;
1287  2170
1288  2176      I := 0 ;
1289  2176
1290  2176      REPEAT  (* Le o nome da proxima estacao *)
1291  217B         Estacao [I] := Carac ;
1292  219A         I := I + 1 ;
1293  21A3         ReadChar (arq, Carac) ;
1294  21B5      UNTIL  ((Carac = ',')  OR  (Carac = ' ')
                        OR  (Carac = ';')
1295  21D6              OR  (Carac = EOL)) ;
1296  21E0
1297  21E4      IF Carac = ' '   (* Elimina os brancos que
                                    sucedem   o   nome  da
                                    estacao              *)
1298  21E4      THEN
1299  21EA          REPEAT
1300  21EE             ReadChar (arq, Carac) ;
1301  2202          UNTIL  Carac <> ' ' ;
1302  2208      END ;
1303  220E
1304  220E      IF Carac = EOL
1305  220E      THEN
1306  2214          EliminaLinhaEmBranco ;
1307  2218          ReadChar (arq, Carac) ;
1308  222B      END ;
1309  2230
1310  2230      (* Procura se  o  nome  desta  estacao
                   pertence a  lista  de  estacoes  da
1311  2230         configuracao,  lista  esta  que foi
                   definida pela declaracao STATIONS   *)
1312  2230
1313  2230      I := 0 ;
1314  2230
1315  2235      NaoAchou := TRUE ;
1316  2235
1317  2259      PtrEstacao3 := TabelaConfig [IndConfig].
                                EstacoesLogicas ;
1318  2259
1319  225C      WHILE (NaoAchou  OR  (PtrEstacao3 <> NIL))
                DO ;
1320  2293        Igual := CompareStr (Estacao,
                                 PtrEstacao3^. Estacoes) ;
1321  2293
1322  22A1        IF Igual = 0
1323  22A1        THEN
1324  22AE            NaoAchou := TRUE ;
1325  22AE        END ;
```

```
1326  22B2
1327  22BF          PtrEstacao2 := PtrEstacao3^.ProxEstacao;
1328  22BF
1329  22C8          PtrEstacao3 := PtrEstacao2 ;
1330  22C8
1331  22CB        END (* while *) ;
1332  22D3
1333  22D3        IF NaoAchou
1334  22D3        THEN
1335  22D9            Erro := TRUE ;
1336  22D9        END ;
1337  22E0
```

```
1338  22E0        NEW (PtrEstacao2) ;   (* Insere  a   nova
                                           estacao na lista
                                           de estacoes      *)
1339  22EA
1340  22EA        PtrEstacao2^. Estacoes :=  Estacao ;
1341  2306
1342  2306        PtrEstacao2^. ProxEstacao := NIL ;
1343  231F
1344  231F        PtrEstacao1^. ProxEstacao := PtrEstacao2 ;
1345  2337
1346  233D        PtrEstacao1 := PtrEstacao2 ;
1347  233D
1348  2340      END (* while *) ;
1349  234E
1350  234E      PtrMod0^. NumEstacao := ContaEstacao ;
1351  235F
1352  235F      IF ((NivelHierarq <> 1)  AND  (NOT Erro))
1353  2377      THEN
1354  2377          AnalisaEstacao (Erro, Tipo, PtrMod0) ;
1355  238A      END ;
1356  238D
1357  238D      IF ContaInst > 1
1358  238D      THEN
1359  23A3          PtrMod3 := PtrMod0^. ProxModCriado ;
1360  23A6          PtrMod3^. Tipo := Tipo ;
1361  23C3          PtrMod3^. Estacao := PtrMod0^. Estacao ;
1362  23EA          PtrMod3^. NumEstacao := ContaEstacao ;
1363  2400          WHILE  (PtrMod3^. ProxModCriado <> NIL)
                    DO
1364  242D            PtrMod2 := PtrMod3^. ProxModCriado ;
1365  2436            PtrMod3 := PtrMod2 ;
1366  2439            PtrMod3^. Tipo := Tipo ;
1367  2456            PtrMod3^. Estacao := PtrMod0^.Estacao;
1368  247D            PtrMod3^.IndTabConfig := IndTabConfig;
1369  2493            PtrMod3^. NumEstacao := ContaEstacao ;
1370  24A6          END (* while *) ;
1371  24B1      END ;
1372  24B6
1373  24B6   END MontaModCriados ;
```

```
1374  24BC
1375  24BC  PROCEDURE  ModuloExportadoCriado (IndTabConfig:
                                                   CARDINAL ;
1376  24BC                                          Hierarq :
                                                   CARDINAL ;
1377  24BC                                           PtrExp :
                                      PointerModuloExport) :
                                                    BOOLEAN ;
1378  24BC
1379  24BC     VAR  PtrMod0 : PointerModulosCriados ;
1380  24BC
1381  24BC     VAR  PtrMod1 : PointerModulosCriados ;
1382  24BC
1383  24BC     VAR  NaoAchou : BOOLEAN ;
1384  24BC
1385  24BC     VAR  ExisteSubConfig : BOOLEAN ;
1386  24BC
1387  24BC     VAR  Achou : BOOLEAN ;
1388  24BC
1389  24BC     VAR  NaoEncontrou : BOOLEAN ;
1390  24BC
1391  24BC     VAR  K : CARDINAL ;
1392  24BC
```



```
1393  24BC     VAR  I : CARDINAL ;
1394  24BC
1395  24BC     VAR  Erro : BOOLEAN ;
1396  24BC
1397  24BC
1398  24BC  BEGIN
1399  24C9
1400  24C9     NaoAchou := TRUE ;
1401  24C9
1402  24ED     PtrMod0 := TabelaConfig [IndTabConfig].
                                         ModulosCriados ;
1403  24ED
1404  24F0     WHILE  (NaoAchou  OR  (PtrMod0 <> NIL))  DO
1405  2530       Igual := CompareStr (PtrExp^. ModExport,
                                   PtrMod0^. NomeModulo);
1406  2530
1407  253E       IF Igual = 0
1408  253E       THEN
1409  254B           NaoAchou := FALSE ;
1410  254B
1411  257D           Igual := CompareStr (PtrMod0^. Tipo,
                                      PtrExp^. TipoImpl) ;
1412  257D
1413  258B           IF Igual <> 0
1414  258B           THEN
1415  2598              Erro := TRUE ;
1416  2598           END ;
1417  259C       ELSE
```

```
1418  25AC              PtrMod1 := PtrMod0^. ProxModCriado ;
1419  25B5              PtrMod0 := PtrMod1 ;
1420  25B8          END ;
1421  25BE
1422  25BE        END (* while *);
1423  25C6
1424  25C6        IF NaoAchou
1425  25C6        THEN
1426  25CC             IF Hierarq <> 1
1427  25CC             THEN
1428  25D4                  Achou := FALSE ;
1429  25D4
1430  25D8                  I := 0 ;
1431  25D8
1432  25DD                  ExisteSubConfig := TRUE ;
1433  25DD
1434  25E1                  WHILE  ExisteSubConfig  DO
1435  25E1
1436  2607                    IF TabTipos [I].NivelHierarq = 1
1437  2607                    THEN
1438  2613                        I := I + 1 ;
1439  2613
1440  261F                        IF I = 255
1441  261F                        THEN
1442  2627                            ExisteSubConfig:= FALSE;
1443  262B                            Erro := TRUE ;
1444  262B                        END ;
1445  262F                    ELSE
1446  2632                        NaoEncontrou := TRUE ;
1447  2632
```

```
1448  2636                        K := 0 ;
1449  2636
1450  2636                        WHILE  (NaoEncontrou  OR
                                         (K < 255))  DO
1451  2647                        (* Pesquisa o indice da
                                     subconfiguracao   na
1452  2647                           TabelaTiposConfig       *)
1453  2647
1454  2698                          Igual := CompareStr
                                      (TabTipos [I]. NomeTipo,
1455  2698                             TabelaTiposConfig
                                       [K].NomeConfig) ;
1456  2698
1457  26A6                          IF Igual = 0
1458  26A6                          THEN
1459  26B3                             NaoEncontrou := FALSE;
1460  26B3                          ELSE
1461  26BD                             K := K + 1 ;
1462  26C6                          END ;
1463  26C9
1464  26C9                        END (* while *) ;
```

```
1465  26D1
1466  26D1                        IF NaoEncontrou
1467  26D1                        THEN
1468  26D7                            Erro := TRUE ;
1469  26D7                        ELSE
1470  270E                            Achou :=
                                         ModuloExportadoCriado
1471  270E                               (K, TabTipos[I].
                                          NivelHierarq,
                                          PtrExp) ;
1472  2711                        END ;
1473  2714
1474  2714                    END ;
1475  2714
1476  2714                    IF ExisteSubConfig
1477  2714                    THEN
1478  271A                        IF Achou
1479  271A                        THEN
1480  2720                            NaoAchou := FALSE ;
1481  2724                            ExisteSubConfig:= FALSE;
1482  272B                        ELSIF I = 255 (* Nao  foi
                                                  criado
                                                  este  modulo
                                                  em nenhuma
                                                  das        *)
1483  272B                              THEN    (* subconfigu-
                                                  racoes que
                                                  compoem a
                                                  configura-
                                                  racao      *)
1484  2733                                  ExisteSubConfig :=
                                                        FALSE ;
1485  2733
1486  2737                                  Erro := TRUE ;
1487  2737                              ELSE
1488  2741                                  I := I + 1 ;
1489  274A                        END ;
1490  274D                    ELSE
1491  2750                        Erro := TRUE ;
1492  2750                    END ;
1493  2754
1494  2754                END (* while *);
1495  275C           END ;
1496  2761
1497  2761       END ;
1498  2766
```



```
1499  2766       RETURN (NOT NaoAchou) ;
1500  276A
1501  276A   END ModuloExportadoCriado ;
1502  277F
```

```
1503  277F  PROCEDURE  AnalisaLnk ;
1504  277F
1505  277F     VAR  PtrModE0 : PointerModulosCriados ;
1506  277F
1507  277F     VAR  PtrModE1 : PointerModulosCriados ;
1508  277F
1509  277F     VAR  NaoAchou : BOOLEAN ;
1510  277F
1511  277F     VAR  NaoEncontrouWITH : BOOLEAN ;
1512  277F
1513  277F     VAR  NaoAcabouLnk : BOOLEAN ;
1514  277F
1515  277F     VAR  TemLigacoes : BOOLEAN ;
1516  277F
1517  277F     VAR  Procura : BOOLEAN ;
1518  277F
1519  277F     VAR  PtrExp0 : PointerModuloExport ;
1520  277F
1521  277F     VAR  PtrExp1 : PointerModuloExport ;
1522  277F
1523  277F     VAR  PtrImp0 : PointerModuloImport ;
1524  277F
1525  277F     VAR  PtrImp1 : PointerModuloImport ;
1526  277F
1527  277F     VAR  PtrModD0 : PointerModulosCriados ;
1528  277F
1529  277F     VAR  PtrModD1 : PointerModulosCriados ;
1530  277F
1531  277F     VAR  ParteAlta : CARDINAL ;
1532  277F
1533  277F     VAR  ParteBaixa : CARDINAL ;
1534  277F
1535  277F     VAR  ParteAlta1 : CARDINAL ;
1536  277F
1537  277F     VAR  ParteBaixa1 : CARDINAL ;
1538  277F
1539  277F     VAR  Nome  : ArrayOfCharType ;
1540  277F
1541  277F     VAR  Nome1 : ArrayOfCharType ;
1542  277F
1543  277F     VAR  NovaListaExp : BOOLEAN ;
1544  277F
1545  277F  BEGIN
1546  278C
1547  278C    REPEAT (* Elimina os brancos que sucedem a
                        palavra LINK                    *)
1548  278C       ReadChar (arq, Carac) ;
1549  27A0    UNTIL  Carac <> " " ;
1550  27A0
1551  27AE    IF Carac = EOL (* Pula as linhas em branco *)
1552  27AE    THEN
1553  27B6        EliminaLinhaEmBranco ;
1554  27BA    END ;
1555  27BE
```

```
  1556  27BE      NaoAcabouLnk := TRUE ;
  1557  27BE
  1558  27BE      ReadChar (arq, Carac) ;
  1559  27D1
  1560  27D6      WHILE  NaoAcabouLnk DO
  1561  27D6
  1562  27E4        FOR I:= 0 TO 39
  1563  27E4        DO
  1564  27E4          Nome [I] := " " ;
  1565  2804        END ;
  1566  2809
  1567  280E        I := 0 ;
  1568  280E
  1569  280E        REPEAT
  1570  2819           Nome [I] := Carac ;
  1571  283C           I := I + 1 ;
  1572  2845           ReadChar (arq, Carac) ;
  1573  2853        UNTIL ((Carac = " ")  OR  (Carac = ",")  OR
                          (Carac = EOL)) ;
  1574  287A
  1575  287E        IF Carac  =  ' '
  1576  287E        THEN
  1577  2886             REPEAT
  1578  288A                ReadChar (arq, Carac) ;
  1579  289E             UNTIL Carac <> ' ' ;
  1580  28A6        END ;
  1581  28AC
  1582  28AC        IF Carac  = EOL
  1583  28AC        THEN
  1584  28B4            EliminaLinhaEmBranco ;
  1585  28B8        END ;
  1586  28BC
  1587  28BC        ReadChar (arq, Carac) ;
  1588  28CB
  1589  28CB        GetPos (arq, ParteAlta, ParteBaixa) ;
  1590  28E6
  1591  28EB        NaoAchou := TRUE ;
  1592  28EB
  1593  290F        PtrModEO := TabelaConfig [IndConfig].
                                               ModulosCriados ;
  1594  290F
  1595  290F        (* Procura se o modulo que esta' sendo
                       ligado foi criado                      *)
  1596  290F
  1597  2912        WHILE (NaoAchou  OR  (PtrModEO^.
                          ProxModCriado  <>  NIL)) DO
  1598  2958          Igual := CompareStr (PtrModEO^.
                                  NomeModulo, Nome) ;
  1599  2958
  1600  2966          IF Igual <> 0
  1601  2966          THEN
```

```
1602  2980            PtrModE1 := PtrModE0^.ProxModCriado ;
1603  2989            PtrModE0 := PtrModE1 ;
1604  298C         ELSE     (* O modulo foi criado *)
1605  2995            NaoAchou := FALSE ;
1606  2995
1607  29C7            PtrImp0 := TabelaConfig [PtrModE0^.
                       IndTabConfig]. ModulosImportados ;
1608  29C7
1609  29D0            NaoEncontrouWITH := TRUE ;
1610  29D0
1611  29D4            WHILE  NaoEncontrouWITH DO
```

```
1612  29D4
1613  29DA               IF Carac = " " (* Elimina      os
                                           brancos    que
                                           sucedem o nome
                                           do modulo      *)
1614  29DA               THEN
1615  29E2                  REPEAT
1616  29E6                     ReadChar (arq, Carac) ;
1617  29FA                  UNTIL  Carac <> " " ;
1618  2A02               END ;
1619  2A08
1620  2A08               REPEAT  (* Elimina os brancos que
                                    sucedem a virgula      *)

1621  2A08                 ReadChar (arq, Carac) ;
1622  2A1C               UNTIL  Carac <> " " ;
1623  2A1C
1624  2A2D               FOR I:= 0 TO 39
1625  2A2D               DO
1626  2A2D                 Nome [I] := " " ;
1627  2A4D               END ;
1628  2A52
1629  2A57               I := 0 ;
1630  2A57
1631  2A57               REPEAT
1632  2A62                 Nome [I] := Carac ;
1633  2A85                 I := I + 1 ;
1634  2A8E                 ReadChar (arq, Carac) ;
1635  2A9C               UNTIL  ((Carac = " ")  OR
                                 (Carac = ",")) ;
1636  2AB7
1637  2ACE               Igual := CompareStr (Nome, "WITH");
1638  2ACE
1639  2ADC               IF Igual = 0
1640  2ADC               THEN
1641  2AE9                  NaoEncontrouWITH := FALSE ;
1642  2AE9               END ;
1643  2AED
1644  2AED           END (* while *) ;
1645  2AF5
```

```
1646   2AF5          REPEAT (* Elimina os brancos que
                               sucedem a palavra WITH *)
1647   2AF5             ReadChar (arq, Carac) ;
1648   2B09          UNTIL  Carac <> " " ;
1649   2B09
1650   2B32          PtrModDO := TabelaConfig [IndConfig].
                                               ModulosCriados ;
1651   2B32
1652   2B3E          FOR I:= 0 TO 39
1653   2B3E          DO
1654   2B3E            Nome1 [I] := " " ;
1655   2B5E          END ;
1656   2B63
1657   2B68          I := 0 ;
1658   2B68
1659   2B68          REPEAT
1660   2B73             Nome1 [I] := Carac ;
1661   2B96             I := I + 1 ;
1662   2B9F             ReadChar (arq, Carac) ;
1663   2BB2          UNTIL  Carac = " " ;
1664   2BB2
1665   2BC0          TemLigacoes := TRUE ;
```



```
1666   2BC0
1667   2BC4          Procura := TRUE ;
1668   2BC4
1669   2BC8          NovaListaExp := TRUE ;
1670   2BC8
1671   2BC8          WHILE  (Procura  OR  (PtrModDO^.
                          ProxModCriado  <>  NIL)) DO
1672   2C0C              Igual := CompareStr (PtrModDO^.
                                        NomeModulo, Nome1) ;
1673   2C0C
1674   2C1A              IF Igual <> 0
1675   2C1A              THEN
1676   2C3A                  PtrModD1 := PtrMod0^.
                                           ProxModCriado ;
1677   2C43                  PtrModDO := PtrMod1 ;
1678   2C4C              END ;
1679   2C52
1680   2C52              IF Procura
1681   2C52              THEN
1682   2C58                  Erro := TRUE ;
1683   2C58              ELSE
1684   2C65                  IF NovaListaExp
1685   2C65                  THEN
1686   2C99                      PtrExp0 := TabelaConfig
                                   [PtrModDO^. IndTabConfig].
                                           ModulosExportados ;
1687   2C99
1688   2C99
1689   2C9C                      ReadChar (arq, Carac) ;
1690   2CAC
```

```
1691  2CB9                    FOR I:= 0 TO 39
1692  2CB9                    DO
1693  2CB9                      Nome [I] := " " ;
1694  2CD9                    END ;
1695  2CDE
1696  2CDE                    REPEAT
1697  2CE3                        Nome [I] := Carac ;
1698  2D06                        I := I + 1 ;
1699  2D0F                        ReadChar (arq, Carac) ;
1700  2D22                    UNTIL  Carac = " " ;
1701  2D22
1702  2D22                    (* Verifica se o modulo foi
                                 realmente exportado *)
1703  2D22
1704  2D4D                    Igual := CompareStr (Nome,
                                   PtrExpO^. ModExport);
1705  2D4D
1706  2D5B                    IF Igual <> 0
1707  2D5B                    THEN
1708  2D68                         Erro := TRUE ;
1709  2D68                    ELSE
1710  2DA3                         Igual := CompareStr
                                    (PtrImpO^. TipoInterf,
1711  2DA3                          PtrExpO^.TipoImpl);
1712  2DB1                         IF Igual <> 0
1713  2DB1                         THEN
1714  2DBE                             Erro := TRUE ;
1715  2DBE                         END ;
```



```
1716  2DC8                    END ;
1717  2DC8
1718  2DC8                    IF NOT Erro
1719  2DC8                    THEN
1720  2DC8                        REPEAT (* Elimina  os
                                           brancos que
                                           sucedem   o
                                           nome      do
1721  2DD4                                 modulo
                                           exportado *)
1722  2DD4                           ReadChar (arq,
                                              Carac) ;
1723  2DE8                        UNTIL  Carac <> " " ;
1724  2DF0                    END ;
1725  2DF6
1726  2DF6                    IF Carac = ","
1727  2DF6                    THEN
1728  2DFE                        REPEAT
1729  2E02                           ReadChar (arq,
                                              Carac) ;
1730  2E16                        UNTIL  Carac <> " " ;
1731  2E1E                    END ;
1732  2E24
```

```
1733  2E24                    IF Carac = EOL
1734  2E24                    THEN
1735  2E2C                       GetPos (arq,
                                         ParteAlta1,
                                         ParteBaixa1) ;
1736  2E46                       EliminaLinhaEmBranco ;
1737  2E4B                       REPEAT (* Elimina os
                                           brancos que
                                           antecedem
                                           a proxima
1738  2E4F                                 palavra   *)
1739  2E4F                         ReadChar (arq,
                                             Carac) ;
1740  2E63                       UNTIL  Carac <> " " ;
1741  2E63
1742  2E74                       FOR I:= 0 TO 39
1743  2E74                       DO
1744  2E74                         Nome [I] := " " ;
1745  2E94                       END ;
1746  2E99
1747  2E9E                       I := 0 ;
1748  2E9E
1749  2E9E                       REPEAT
1750  2EA9                          Nome [I] := Carac ;
1751  2ECC                          I := I + 1 ;
1752  2ED5                          ReadChar (arq,
                                              Carac) ;
1753  2EE3                       UNTIL ((Carac = " ")
                                    OR  (Carac = ".")
1754  2EFE                          OR  (Carac = ",")
                                    OR  (Carac = ";"));
1755  2F16
1756  2F1A                       IF Carac = " "
1757  2F1A                       THEN  (* Elimina  os
                                          brancos que
                                          estao  apos
                                          a palavra   *)
1758  2F22                         REPEAT
1759  2F26                            ReadChar (arq,
                                                Carac);
1760  2F3A                         UNTIL Carac <> " ";
1761  2F42                       END ;
1762  2F48
1763  2F48                       IF ((Carac = ",")  OR
                                    (Carac = ";"))
1764  2F5E                       THEN
1765  2F6D                          IF PtrExpO^.
                                    ProxModExport = NIL
```



```
1766  2F6D                          THEN
1767  2F7D                             TemLigacoes :=
                                                FALSE ;
```

```
1768  2F81                                         NaoAcabouLnk :=
                                                         FALSE ;
1769  2F81                                     ELSE
1770  2F88                                         Erro := TRUE ;
1771  2F88                                     END ;
1772  2F92                                 ELSE
1773  2FA2                                     PtrImp1 :=
                                                   PtrImp0^.
                                                   ProxModImport ;
1774  2FA2
1775  2FAB                                     PtrImp0 := PtrImp1;
1776  2FAB
1777  2FC8                                     Igual := CompareStr
                                                    (Nome, Nome1) ;
1778  2FC8
1779  2FD6                                     IF Igual = 0
1780  2FD6                                     THEN
1781  2FF0                                         PtrExp1 :=
                                                   PtrExp0^.
                                                   ProxModExport ;
1782  2FF0
1783  2FF9                                         PtrExp0 :=
                                                         PtrExp1 ;
1784  2FF9
1785  3002                                         Procura :=
                                                         FALSE ;
1786  3002
1787  3006                                         NovaListaExp :=
                                                         FALSE ;
1788  3006                                     ELSE
1789  300D                                         NovaListaExp :=
                                                          TRUE ;
1790  300D
1791  3011                                         Procura :=
                                                          TRUE ;
1792  3011
1793  301D                                         FOR I:= 0 TO 39
1794  301D                                         DO
1795  301D                                           Nome1 [I] :=
                                                          " " ;
1796  303D                                         END ;
1797  3042
1798  3047                                         Nome1 := Nome ;
1799  3051                                     END ;
1800  305F                                 END ;
1801  305F                             END ;
1802  3064                         END ;
1803  3069                     END ;
1804  3069                   END (* while *) ;
1805  3071               END ;
1806  3071              END (* while *) ;
1807  3079            END (* while *) ;
1808  3081            END AnalisaLnk ;
1809  3085
1810  3085                  (* Corpo do programa principal *)
1811  3085
1812  3085        BEGIN
```

```
 1813  308F
 1814  308F       (* Le o nome do arquivo que contem a
                     configuracao                          *)
 1815  308F
 1816  308F       SetOpen (arq) ;
 1817  309E
 1818  30A3       IF arq.res <> done
 1819  30A3       THEN
 1820  30AF          Erro := TRUE ;
 1821  30AF       ELSE
```



```
 1822  30BC          SetRead (arq) ;
 1823  30C6
 1824  30C6          REPEAT          (* Elimina as linhas em
                                        branco iniciais      *)
 1825  30CB             REPEAT
 1826  30CB                ReadChar (arq, Carac) ;
 1827  30DF             UNTIL  Carac <> ' ' ;
 1828  30DF
 1829  30E7          UNTIL  ((arq.eof)  OR (Carac <> EOL)) ;
 1830  30FE
 1831  3102          IndTipoConfig := 0 ;
 1832  3102
 1833  310D          IndConfig := 0 ;
 1834  310D
 1835  310D          IF NOT arq.eof   (* Nao e' fim de
                                         arquivo        *)
 1836  3113          THEN
 1837  3113               Again (arq) ;   (* Posiciona no
                                            ultimo caractere
                                            lido             *)
 1838  3129
 1839  3129               MontaConfig ;
 1840  312E
 1841  312E               EliminaLinhaEmBranco ;
 1842  3131
 1843  3131               AnalisaUse (TabTipos,
                                      NivelHierarq) ;
 1844  3148
 1845  3148               EliminaLinhaEmBranco ;
 1846  314B
 1847  314B               MontaEstacoes ;
 1848  314F
 1849  314F               EliminaLinhaEmBranco ;
 1850  3152
 1851  3152               REPEAT
 1852  3156                  ReadChar (arq, Carac) ;
 1853  316A               UNTIL  Carac <> ' ' ;
 1854  316A
 1855  3178               I := 0 ;
 1856  3178
```

```
1857  3178          REPEAT
1858  317E             Nome [I] := Carac ;
1859  319F             I := I + 1 ;
1860  31A8             ReadChar (arq, Carac) ;
1861  31BB          UNTIL  Carac = ' ' ;
1862  31BB
1863  31C9          NaoExporta := TRUE ;
1864  31C9
1865  31CE          NaoImporta := TRUE ;
1866  31CE
1867  31E5          Igual := CompareStr (Nome,
                                         'IMPORT') ;
1868  31E5
1869  31F3          IF Igual = 0
1870  31F3          THEN
1871  31FB               AnalisaImport ;
1872  3200
1873  3200               EliminaLinhaEmBranco ;
1874  3203
1875  3203               REPEAT    (* Elimina os brancos
                                      que antecedem a
1876  3207                            proxima palavra *)
```



```
1877  3207                  ReadChar (arq, Carac) ;
1878  321B               UNTIL  Carac <> ' ' ;
1879  321B
1880  3229               NaoImporta := FALSE ;
1881  3229
1882  322E               I := 0 ;
1883  322E
1884  322E               REPEAT
1885  3234                  Nome [I] := Carac  ;
1886  3255                  I := I + 1 ;
1887  325E                  ReadChar (arq, Carac) ;
1888  3271               UNTIL  Carac = ' ' ;
1889  3271
1890  3279          ELSE TabelaConfig [IndConfig].
                                   NumModImport := 0 ;
1891  32A7          END ;
1892  32AC
1893  32C3          Igual := CompareStr (Nome,
                                         'EXPORT') ;
1894  32C3
1895  32D1          IF Igual <> 0
1896  32D1          THEN
1897  32D9               TabelaConfig [IndConfig].
                                   NumModExport := 0 ;
1898  32FE     (*            IF NaoImporta   *)
1899  32FE     (*            THEN  Esta' na hora de
                                   montar  a tabela
                                   de      estacoes
1900  32FE                         porque esta e' a
```

```
                                           configuracao  de
1901  32FE                                 mais  alto nivel
1902  32FE                           END ;                        *)
1903  32FE                   ELSE
1904  3306                        AnalisaExport ;
1905  3306
1906  3309                        NaoExporta := FALSE ;
1907  3309                   END ;
1908  3313
1909  3313                   EliminaLinhaEmBranco ;
1910  3313
1911  3313                   ReadChar (arq, Carac) ;
1912  3326
1913  3326                   REPEAT    (* Percorre os caracteres
                                          que  formam a  palavra
1914  332B                                CREATE                 *)
1915  332B                      ReadChar (arq, Carac) ;
1916  333F                   UNTIL  Carac = ' ' ;
1917  333F
1918  3347                   REPEAT    (* Percorre os brancos que
                                          sucedem    a    palavra
1919  334D                                CREATE                 *)
1920  334D                      ReadChar (arq, Carac) ;
1921  3361                   UNTIL  Carac <> ' ' ;
1922  3361
1923  336F                   IF Carac = EOL
1924  336F                   THEN
1925  3377                      EliminaLinhaEmBranco ;
```



```
1926  337B                   END ;
1927  337F
1928  337F                   ReadChar (arq, Carac) ;
1929  338E
1930  3393                   I := 0 ;
1931  3393
1932  3393                   REPEAT    (* Le o nome da primeira
                                          instancia criada      *)
1933  339E                      Nome [I] := Carac  ;
1934  33BF                      I := I + 1 ;
1935  33C8                      ReadChar (arq, Carac) ;
1936  33D6                   UNTIL  ((Carac = ' ')   OR
                                     (Carac = ';')   OR
                                     (Carac = ',')) ;
1937  33FD
1938  3401                   IF Carac = ' '
1939  3409                   THEN REPEAT  (* Elimina os brancos
                                              que sucedem o nome
1940  340D                                    instancia que esta'
                                              sendo criada      *)
1941  340D                          ReadChar (arq, Carac) ;
1942  3421                       UNTIL  Carac <> ' ' ;
1943  3429                   END ;
```

```
1944  342F
1945  342F                Again (arq) ;
1946  3439
1947  3439                NEW (PtrMod0) ;
1948  344C
1949  344C                PtrMod0^. NomeModulo := Nome ;
1950  3467
1951  3467                PtrMod0^. ProxModCriado := NIL ;
1952  3481
1953  3481                TabelaConfig [IndConfig].
                                   ModulosCriados := PtrMod0 ;
1954  34AB
1955   34AB                 MontaModCriados (Carac, PtrMod0,
                                  PtrMod0^.NomeModulo, Erro) ;
1956  34D0
1957  34D3                NaoAchouLnk := TRUE ;
1958  34D3
1959  34DD                WHILE  NaoAchouLnk  DO
1960  34DD
1961  34DD                  EliminaLinhaEmBranco ;
1962  34E9
1963  34E9                  ReadChar (arq, Carac) ;
1964  34FC
1965  3501                  I := 0 ;
1966  3501
1967  3501                  REPEAT
1968  350C                     Nome [I] := Carac ;
1969  352D                     I := I + 1 ;
1970  3536                     ReadChar (arq, Carac) ;
1971  3544                  UNTIL  ((Carac = ' ')  OR
                                   (Carac = ',')  OR
                                   (Carac = ';')) ;
1972  356B
1973  3586                  Igual := CompareStr (Nome,
                                              'LINK') ;
1974  3586
1975  3594                  IF Igual = 0
1976  3594                  THEN
1977  35A1                       NaoAchouLnk := FALSE ;
```

```
1978  35A1                  ELSE
1979  35AE                       NEW (PtrMod1) ;
1980  35BC
1981  35BC                       PtrMod1^.NomeModulo := Nome;
1982  35D7
1983  35D7                       PtrMod1^. ProxModCriado :=
                                                      NIL ;
1984  35F1
1985  35F1                       PtrMod0^. ProxModCriado :=
                                                  PtrMod1 ;
1986  360B
1987  3611                       PtrMod0 := PtrMod1 ;
```

```
1988  3611
1989  3622                     IF Carac = ' '
1990  3622                     THEN
1991  362A                        REPEAT
1992  362E                           ReadChar (arq,
                                               Carac) ;
1993  3642                        UNTIL  Carac <> ' ' ;
1994  364A                     END ;
1995  3650
1996  3650                     MontaModCriados
                               (Carac, PtrMod0, PtrMod0^.
                                NomeModulo, Erro) ;
1997  3674                  END ;
1998  3677
1999  3677              END (* while *) ;
2000  367F
2001  367F              IF NOT NaoExporta   (* No caso de
                                               ter havido
                                               exportacao,
                                               verifica   *)
2002  367F              THEN                (* se    as
                                               instancias
                                               dos modulos
                                               de implemen-
2003  367F                                     tacao    que
                                               foram  expor-
                                               tados, foram
                                               criados    *)
2004  36AB                 PtrExp0 := TabelaConfig
                                      [IndConfig].
                                      ModulosExportados ;
2005  36AB
2006  36AB
2007  36C9                 WHILE  PtrExp0^. ProxModExport
                                  <> NIL  DO
2008  36E8                   InstCriada :=
                                   ModuloExportadoCriado
                                   (IndConfig,
                                    NivelHierarq,
2009  36E8                          PtrExp0) ;
2010  36E8
2011  36F4                   IF NOT InstCriada
2012  36F4                   THEN
2013  3700                       Erro := TRUE ;
2014  3700                   END ;
2015  370A
2016  371D                   PtrExp1 := PtrExp0^.
                                             ProxModExport ;
2017  371D
2018  372D                   PtrExp0 := PtrExp1 ;
2019  372D
2020  3731                 END (* while *) ;
2021  3741              END ;
2022  3746
2023  3746         (*   IF NOT Erro
2024  3746              THEN
2025  3746                 IF NivelHierarq <> 1
```

```
  2026  3746                      THEN
  2027  3746                            AnalisaLnk ;   *)
  2028  3746              END ;
```

```
  2029  374B        END ;
  2030  374B
  2031  374B    END INTERPRET.
```

```
Modula-2/86 Compiler Version V 2.00

==>      0 Error(s) found

    source file: C:interpre.mod
    module name: INTERPRET
    module key :  B057H  037EH  078AH

    The interactive setting of the options was:
            emulator    (E): on
            stacktest   (S): on
            rangetest   (R): on
            indextest   (T): on

    No code for 8087 Emulator generated

    Codesize:   14157 bytes
    Datasize:   46409 bytes
```