

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

UM SUPORTE PARA A CONFIGURAÇÃO ESTÁTICA DE SISTEMAS
DISTRIBUÍDOS UTILIZANDO ABORDAGEM POR LINGUAGEM: PROJETO E
IMPLEMENTAÇÃO

DISSERTAÇÃO SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA
CATARINA-PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA
ELÉTRICA

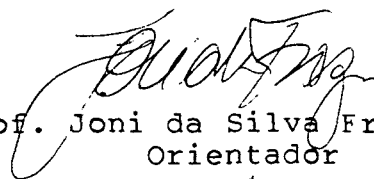
LUIZ EDIVAL DE SOUZA

FLORIANÓPOLIS - OUTUBRO/88

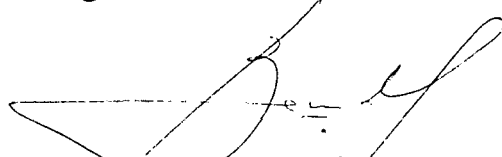
UM SUPORTE PARA A CONFIGURAÇÃO ESTÁTICA DE SISTEMAS DISTRIBUÍDOS
UTILIZANDO ABORDAGEM POR LINGUAGEM: PROJETO E IMPLEMENTAÇÃO

LUIZ EDIVAL DE SOUZA

ESTA DISSERTAÇÃO FOI JULGADA PARA A OBTENÇÃO DO TÍTULO DE MESTRE EM
ENGENHARIA - ESPECIALIDADE ENGENHARIA ELÉTRICA E APROVADA EM SUA
FORMA FINAL PELO CURSO DE PÓS-GRADUAÇÃO

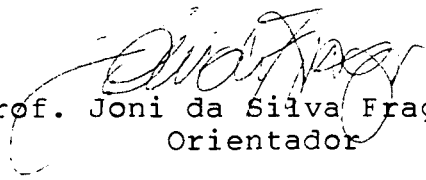


Prof. Joni da Silva Fraga, Dr.
Orientador



Prof. José Carlos Moreira Bermudez, Ph. D.
Coordenador do Curso de Pós-Graduação
em Engenharia Elétrica

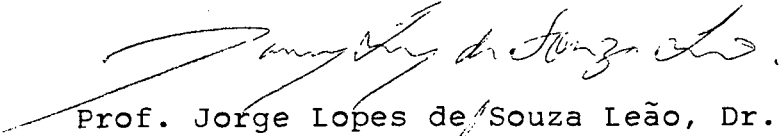
BANCA EXAMINADORA:



Prof. Joni da Silva Fraga, Dr.
Orientador



Prof. Jean-Marie Farinés, Dr. Ing.
Co-orientador



Prof. Jorge Lopes de Souza Leão, Dr. Ing.



Prof. Francisco E. C. Viola, Dr. Ing.

À minha esposa Rosy e as minhas filhas Alana e Thaís que em troca das minhas horas de ausência souberam me dar, com toda intensidade, o amor e a compreensão que me alimentaram na execução deste trabalho.

Agradecimento

Aos meus pais, meus primeiros professores.

Ao professor e amigo Joni da Silva Fraga, pelo empenho e dedicação nos trabalhos de orientação e correção.

Ao professor Jean-Marie Farines, pelo apoio e sugestões fornecidas.

À todos os professores e amigos do Laboratório de Controle e Microinformática e, em especial, aos amigos Eraldo Silveira e Silva e Luiz Nacamura Jr., pela amizade e contribuição para a realização deste trabalho. À Luiz Martins Oenning pela paciência e dedicação no trabalho de implementação.

Aos professores e amigos da EFEI que permitiram o início deste trabalho.

À EFEI e a CAPES pelo suporte financeiro.

À todos que direta ou indiretamente contribuíram para que este trabalho fosse concluído.

SUMÁRIO

RESUMO.....	x
ABSTRACT.....	xi
CAPÍTULO 1. INTRODUÇÃO.....	01
CAPÍTULO 2. PROGRAMAÇÃO EM LARGA ESCALA.....	04
2.1. Introdução.....	04
2.2. Abordagens de programação para aplicações distribuídas.....	04
2.3. Princípio de Decomposição Modular.....	05
2.3.1. Módulo.....	06
2.3.2. Relações de Dependências entre módulos.....	07
2.3.3. Programação em Pequena Escala.....	10
2.3.4. Programação em Larga Escala.....	10
2.4. Atividades Relacionadas com a Programação em Larga Escala.....	13
2.4.1. Gerenciamento de Configuração.....	14
2.4.2. Controle de Versão.....	15
2.4.3. Processo de Desenvolvimento de Software Distribuído.....	16
2.5. Trabalhos Relacionados.....	18
2.5.1. Sistema MESA	18

2.5.2. Sistema PRONET	20
2.5.3. Sistema MASCOT	21
2.5.4. Sistema CONIC	22
2.5.5. Linguagem de Programação ADA	23
2.6. Conclusão.....	24

CAPÍTULO 3. CONFIGURAÇÃO DE UM SISTEMA DISTRIBUÍDO

PARA APLICAÇÕES EM TEMPO REAL	26
3.1. Introdução.....	26
3.2. Definição do Sistema	27
3.2.1. Arquitetura do sistema.....	27
3.2.2. Paradigma de Programação.....	28
3.2.3. Linguagem de Implementação de Sistemas.....	31
3.2.4. Suporte de Tempo de Execução.....	32
3.3. Modelo de Configuração.....	34
3.3.1. Configuração Estática.....	35
3.3.2. Configuração Dinâmica.....	36
3.4. Linguagem de Configuração de Sistemas.....	38
3.4.1. Declarações LINCS	40
3.4.1.1. Notação Utilizada.....	42
3.4.1.2. Declaração de Constantes.....	42

3.4.1.3. Declaração de Famílias.....	43
3.4.1.4. Declaração de Portos.....	43
3.4.1.5. Declaração de Contexto.....	44
3.4.1.6. Declaração de Instanciação.....	45
3.4.1.7. Declaração de Conexão de Portos.....	47
3.4.1.8. Declarações de Especificação de Configuração.....	49
3.5. Aspectos do desenvolvimento do suporte para a configuração estática de sistemas.....	54
3.5.1. Análise de Contexto.....	54
3.5.2. Especificação Funcional.....	55
3.5.2.1. Processador LINC.S.....	55
3.5.2.2. Processo de Configuração Estática.....	56
3.6. Conclusão.....	61
CAPÍTULO 4. ASPECTOS DE IMPLEMENTAÇÃO DO CONFIGURADOR E RESULTADOS.....	62
4.1. Introdução.....	62
4.2. Processador LINC.S.....	63
4.2.1. Estrutura do Processador.....	63
4.2.2. Processamento de Especificação de Configuração.....	69

4.2.2.1. Processamento de Especificação	
SYSTEM	71
4.2.2.2. Processamento de Especificação	
GROUP MODULE	72
4.2.2.3. Processamento da Declaração de	
Constantes (CONST).....	73
4.2.2.4. Processamento da Declaração de	
Famílias (FAMILY).....	74
4.2.2.5. Processamento da Declaração de	
Portos (PORT).....	75
4.2.2.6. Processamento da Declaração de	
Contexto (USE).....	75
4.2.2.7. Processamento da Declaração de	
Instanciação (CREATE).....	76
4.2.2.8. Processamento da Declaração de	
Conexão de Portos (LINK).....	78
4.2.3. Estrutura da Base de Dados.....	80
4.2.4. Geração da Tabela de Configuração.....	82
4.3. Programa Construtor.....	84
4.3.1. Construção da Imagem de Carga.....	85
4.3.2. Carregamento.....	85
4.4. Ferramentas Utilizadas.....	87
4.5. Resultados.....	89

4.6. Conclusão.....	91
CAPÍTULO 5. PERSPECTIVAS FUTURAS E CONCLUSÃO FINAL.....	92
5.1. Introdução.....	92
5.2. Considerações à respeito do Módulo Gerenciador de Configuração.....	92
5.2.1. Aspectos relacionados com a Tradução e a Carga da imagem nas estações.....	93
5.2.2. Ativação da reconfiguração via aplicação.....	95
5.3. Ponto de Sincronização para Mudanças na Configuração.....	95
5.4. Atribuição de Prioridades via LINC.....	97
5.5. Conclusão Final.....	97
BIBLIOGRAFIA.....	99
APÊNDICE A: Estruturas utilizadas na implementação do processador LINC.....	103
APÊNDICE B: Especificação Funcional do Processo de Configuração de Sistemas Distribuídos Utilizando SADT.....	109
APÊNDICE C: Célula Flexível de Manufatura.....	119

RESUMO

Este trabalho visa apresentar uma Linguagem de Configuração de Sistemas (LINCS) e o suporte para a configuração estática de sistemas distribuídos. A linguagem LINCS é parte de um Ambiente de Desenvolvimento e Execução de Software Distribuído - ADES cujo objetivo principal é concentrar ferramentas que facilitem a programação distribuída para aplicações em tempo real. A abordagem adotada de concepção de software segue o princípio de decomposição modular e tem como objetivo o desenvolvimento de sistemas evolutivos.

A LINCS é parte das ferramentas relacionadas com as atividades de programação em larga escala. As declarações nesta linguagem foram definidas no sentido de facilitar a reutilização de software, a estruturação hierárquica de sistemas além de contribuir, por suas características, para a construção de softwares altamente flexíveis.

Neste trabalho são abordados os vários aspectos relacionados com a configuração sendo também discutido as necessidades de suporte e as soluções utilizadas para a construção inicial de sistemas distribuídos (configuração estática). O mecanismo de construção desenvolvido carrega os módulos aplicativos e o suporte de tempo de execução via rede local utilizando um protocolo a duas fases.

ABSTRACT

The objective of this work is to introduce the Systems Configuration Language (LINCS) and support of the static configuration for distributed systems. The language LINCS is part of Distributed Software Development and Execution Environment - ADES which has the main objective of integrating tools for distributed programming in real-time applications. The approach used follows the modular decomposition principle and allows the construction of evolve systems.

LINCS is part of a tool related with activities of large scale programming. The statements were defined to obtain software reutilization and hierarchy construction of flexible systems.

This work presents the aspects related with system's configuration, shows the approach used in initial construction of distributed systems (static configuration) and the features of the LINCS. The support of the static configuration for distributed system developed, loads the application's module and the run time support by down-line loading through local network, using a two phase protocol.

CAPÍTULO 1

INTRODUÇÃO

A crescente corrida tecnológica tem proporcionado mudanças significativas em ambientes industriais e de controle de processo. O mesmo impacto pode ser observado nos ambientes comerciais tais como, escritórios, bancos, reserva de passagens aéreas, etc. As vantagens experimentadas são decorrência da tecnologia da informação, cujas mudanças revolucionárias devem-se à evolução da tecnologia de integração - especialmente na forma VLSI, com a melhora surpreendente na capacidade de processamento, armazenamento e comunicação. Estas características aliadas ao baixo custo dos componentes tem fortemente influenciado a utilização em automação industrial e controle de processos de sistemas informáticos distribuídos.

Um sistema constituído por vários componentes de software alocados em computadores interconectados via uma rede de comunicação, mas que cooperam para atingir um objetivo comum é normalmente chamado de sistema distribuído. A interconexão dos computadores é dita fracamente acoplada no sentido de que a comunicação entre processos é realizada por passagem de mensagens e não por memória compartilhada.

As vantagens obtidas com a aplicação destes sistemas atendem plenamente as necessidades atuais no controle de

processo e automação da manufatura. A flexibilidade, a melhoria de desempenho e a disponibilidade são frequentemente citadas na literatura. A busca incansável de um aproveitamento eficiente destas características e principalmente as dificuldades de programação introduzidas com a distribuição fez com que surgissem novas abordagens de concepção de software, e é neste contexto que se enquadra esta dissertação.

Mesmo antes da aplicação do computador, o controle automático de processos contínuos já era verdadeiramente distribuído. A diferença fundamental é que este não tinha um funcionamento integrado, consistindo de um conjunto de malhas de controle independentes cujos elementos controladores eram os reguladores pneumáticos. A tendência atual é integrar todo o sistema, permitindo um controle hierarquizado através da introdução de níveis de controle. O processamento distribuído é uma tecnologia recente que facilita a introdução destes níveis através de novos conceitos em supervisão, operação e produção. A distribuição e integração do controle facilitam a implantação de sistemas hierárquicos do tipo Fabricação Integrada por Computador (Computer Integrated Manufacturing - CIM).

As vantagens esperadas ou desejadas com a aplicação de sistemas distribuídos não podem ser eficientemente exploradas se não houver um gerenciamento de serviços e ferramentas apropriadas. O objetivo desta dissertação é desenvolver uma ferramenta de implementação capaz de expressar a composição através de componentes modulares de um software distribuído, envolvendo também sua instalação.

A ferramenta a ser desenvolvida é parte de um contexto maior, Ambiente de Desenvolvimento e Execução de Software - ADES, cujo finalidade é fornecer facilidades para a investigação de problemas associados a programação de aplicações distribuídas em tempo real. A composição do sistema é realizada por uma Linguagem de Configuração de Sistemas - LINCS. Esta linguagem incorpora construções para a estruturação de softwares distribuídos, sendo que, em sua versão atual se destina à configuração estática de sistemas.

Esta dissertação está dividida em cinco capítulos. No capítulo 2 é apresentado um estudo sobre as características de softwares distribuídos e os atributos destes, de modo que se tenha sistemas evolutivos. O capítulo 3 introduz o objetivo principal desta dissertação que é um suporte para a configuração de sistemas distribuídos. Neste sentido, são abordadas as funcionalidades destas ferramentas e suas especificações.

O capítulo 4 é a descrição da implementação na sua forma mais sucinta, onde os aspectos mais relevantes são apontados. As estruturas das entidades envolvidas com o processo de configuração são descritas. Finalmente, baseados na integração das ferramentas de construção de sistemas e na implementação de uma aplicação das mesmas são apresentadas em forma de simulação de uma célula flexível da manufatura, algumas considerações finais. O capítulo 5 destaca alguns pontos considerados importantes para a continuidade deste trabalho.

CAPÍTULO 2

PROGRAMAÇÃO EM LARGA ESCALA

2.1. INTRODUÇÃO

As dificuldades encontradas na produção de softwares distribuídos conduziram a pesquisa no sentido de desenvolver novos modelos de programação. Estes modelos são implementados por ferramentas apropriadas e incorporados em ambientes para o compartilhamento de recursos e informações. O objetivo primordial destes modelos é explorar os atributos de um sistema distribuído tais como, desempenho, flexibilidade, alta disponibilidade e grande confiabilidade.

As características de modularidade para controlar a evolução do projeto e da implementação têm sido largamente suportadas por ambientes de desenvolvimento de software.

O objetivo deste capítulo é apresentar um estudo sobre as características do princípio de decomposição modular e suas atividades correlatas para a construção de sistemas distribuídos flexíveis. A seguir são apresentadas as atividades envolvidas com a programação em larga escala e, finalmente apresenta-se um resumo sucinto de trabalhos relacionados.

2.2. ABORDAGENS DE PROGRAMAÇÃO PARA APLICAÇÕES DISTRIBUÍDAS

Inicialmente a programação de sistemas distribuídos era realizada através de chamadas a serviços fornecidos pelos chamados Sistemas Operacionais de Rede (NOS). Nesta abordagem,

as aplicações distribuídas (coleção de programas sequenciais) se comunicavam usando as primitivas de comunicação disponíveis no NOS. As interfaces de comunicação apresentavam uma semântica diferente quando da sua utilização na comunicação local e remota. A disponibilidade de serviços de suporte para a configuração se restringia à construção inicial, dificultando o controle e a manutenção da configuração.

As dificuldades de programação utilizando a abordagem mencionada foram em parte contornadas através de novas propostas de modelos de programação. Nestes modelos as características de modularidade, concorrência, sincronização e a distribuição são incorporadas em uma única linguagem. Nesta abordagem de linguagem é muito importante a incorporação de conceitos de modularidade no sentido de dominar a complexidade do software distribuído e de obter atributos de flexibilidade. Nos próximos itens são levantados aspectos conceituais sobre a modularidade.

2.3. PRINCÍPIO DE DECOMPOSIÇÃO MODULAR

No caso de sistemas complexos a abordagem clássica para o projeto do software é o **princípio de decomposição modular** ("Programming-in-the-small vs Programming-in-the large") apresentado em (DeRemer, 76). Este princípio introduz o conceito de módulo como unidade básica para a obtenção de sistemas flexíveis; com isto a programação de um sistema se dá em dois níveis:

- **Programação em Pequena Escala:** Está relacionada com a construção de componentes de sistemas (módulos), devendo

portanto, tratar com atividades de projeto de estrutura de dados, algoritmos, etc.

- **Programação em Larga Escala:** Está envolvida com a composição do sistema a partir dos módulos.

Um arranjo de componentes (módulos), constituindo um sistema, é chamado normalmente na literatura de **configuração**. No contexto de programação distribuída, o princípio de decomposição modular permite a configuração do software através de módulos distribuídos nos diversos computadores interconectados por uma rede de comunicação de dados, fornecendo então, o cumprimento dos requisitos de flexibilidade. Nos itens subsequentes serão abordados aspectos que são geralmente associados ao módulo e à programação em larga escala.

2.3.1. Módulo

O **módulo** é classicamente apresentado como a associação de uma interface que especifica inteiramente os recursos fornecidos e utilizados por este, com um corpo que realiza concretamente os recursos do módulo (implementação). Interfaces e corpos se utilizam, segundo necessidades, de recursos fornecidos por outros módulos. Na prática estes recursos são constantes, tipos, variáveis e procedimentos.

A interface estabelece restrições de acesso aos clientes sobre recursos implementados no corpo de um módulo, ou seja, uma interface só torna acessível um subconjunto dos recursos do módulo (ocultamento de informação introduzido em (Parnas, 72)). As restrições de acesso são determinadas pelas relações de utilização definidas pela interface (ítem 2.3.2).

A separação entre interface e corpo permite que o projeto, implementação e testes de um módulo possam ser realizados de maneira independente de outros módulos do sistema. A abstração da implementação de um módulo favorece a definição de fronteiras para a propagação de modificações em componentes-módulos ou ainda no próprio sistema.

Em determinadas linguagens, as interfaces e corpos podem ser textualmente separados quando da programação em pequena escala. As interfaces são compiláveis em tabelas símbolos que são consultadas quando da ligação de módulos; isto decorre de que, do ponto de vista da programação em larga escala só é necessário o conhecimento da interface do módulo.

2.3.2. Relações de Dependência entre módulos

A modularidade introduz dependências entre módulos que deverão ser cuidadosamente observadas durante a manutenção e a própria evolução do sistema. Mudanças passíveis de ocorrer durante o ciclo de vida de um módulo provocam efeitos em outros módulos e no próprio sistema. Estes efeitos dependem das relações de dependência entre os módulos e estão associados aos possíveis tipos de mudanças: mudanças na interface e mudanças no corpo (Kamel, 87).

Em um sistema modular, a existência das relações de dependências se traduz nas formas seguintes (Krakowiac, 82):

1) Relações de Utilização que se manifestam segundo:

- **importações:** um módulo (cliente) utiliza recursos implementados por outro módulo.

- **exportações:** o módulo implementa recursos utilizados por outros.

As relações de utilização devem ser especificadas na interface através de declarações de importações e de exportações.

- 2) **Relações de realização** se manifestam através de operações de inclusão: um módulo inclui no seu corpo outros módulos. Estes módulos incluídos, chamados de módulos ou unidades de definição, se comportam como um modelo do qual são "tiradas várias cópias" pelos módulos que realizam a inclusão. Nas relações de utilização um módulo importado (ou recurso importado) é compartilhado por seus importadores existindo só um exemplar do mesmo.

As relações de dependência permitem portanto determinar as consequências de uma modificação sobre um componente do sistema:

- A modificação de uma interface implica na recompilação de todos os corpos que a importam;
- Modificações em um módulo incluído se propaga aos componentes que o incluem, exigindo a recompilação do mesmo; e
- As modificações de um corpo são invisíveis desde que respeitem as especificações funcionais e as relações de realização.

O problema de alterações pode ser resolvido através de documentação e comentários nos textos fontes o quê de certa forma pode se transformar em tarefa exaustiva e propensa a

erros. Porém a propagação de modificações pode ser facilitada com o uso de ferramentas automatizadas na manutenção do software. Provavelmente a mais conhecida é a ferramenta MAKE (Feldman, 79); esta requer um arquivo de entrada contendo as relações de dependências e as ações que devem ser executadas quando as condições especificadas forem satisfeitas.

A estabilidade de uma configuração está intimamente ligada à estabilidade dos módulos que a compõem. Se o efeito da retirada ou introdução de um módulo for tratado adequadamente nos módulos dependentes, a estabilidade destes garantirá uma propagação minimizada na configuração do sistema. As definições de (Schneidewind, 87) sobre estabilidade podem ser transportadas para o caso de sistemas evolutivos através das seguintes definições:

- **Estabilidade de Configuração:** Atributo de qualidade que indica a resistência aos efeitos potenciais na configuração, da retirada ou introdução de um módulo;
- **Estabilidade do Módulo:** Medida da resistência sobre efeitos potenciais de modificações de um módulo sobre outros módulos;

Em uma configuração, um módulo possuidor de uma estabilidade baixa, quando modificado, produzirá efeitos indesejáveis sobre outros módulos que o utilizam ou são chamados por este. Diante destes fatos, pode se concluir que a estabilidade de configuração depende da fase de projeto do sistema, onde alguns princípios básicos de modularidade devem ser respeitados.

2.3.3. Programação em Pequena Escala

Para a codificação do módulo é necessário uma linguagem de programação (ou linguagem de implementação de módulos) que apresente alguns atributos como o de permitir a separação entre corpo e interface, incentivando então o requisito de ocultamento de informação. Outro atributo importante e desejável para a linguagem é que este possibilite a definição de um tipo módulo, a partir do qual serão criadas instâncias, caracterizando a reutilização de software.

A linguagem de implementação de módulos pode especificar os recursos importados sem mencionar o nome dos módulos fornecedores. A responsabilidade de definir a unidade da qual os recursos são importados, bem como a verificação da consistência dos tipos nas interfaces estão vinculados a programação em larga escala. Esta característica torna a evolução de sistemas bem menos complexa.

2.3.4. Programação em Larga Escala

Os termos programação em larga escala e programação em pequena escala foram primeiramente introduzidos na literatura por (De Remer, 76). Este artigo está centrado nas idéias da distinção nas atividades de programação de grandes sistemas, através da divisão dos trabalhos em programação de componentes de software (módulos) e estruturação de sistemas a partir da interconexão destes componentes.

Dois aspectos podem ser distinguidos na programação em larga escala:

- a expressão da composição; e
- a construção propriamente dita do sistema.

A expressão clássica da composição corresponde a uma lista de comandos dirigida ao editor de ligações, especificando os arquivos componentes a serem ligados. A evolução destas listas fez com que surgissem as chamadas "linguagens de interconexão de módulos".

A linguagem de interconexão de módulos difere fundamentalmente das linguagens de programação convencionais nas unidades que são manipuladas. Esta linguagem é projetada para manipular unidades compiladas através de interfaces bem definidas com o objetivo de compor o sistema. Neste sentido, a linguagem deve fornecer mecanismos para descrever formalmente as dependências entre os componentes do sistema.

A composição de um sistema pode ser vista como um grafo orientado acíclico, onde as folhas são módulos e os nós internos são subsistemas (Narayanaswany, 87). Cada subsistema é realizado por uma configuração de seus nós sucessores que podem ser tanto módulos como subsistemas. Um subsistema é realizado pela composição de seus sucessores usando regras de construção (carregar, ligar e por vezes também compilar).

A primeira linguagem de interconexão de módulos, MIL-75 (DeRemer, 76), foi definida com o objetivo de descrever a estrutura do sistema através de três elementos básicos: recursos, módulos e sistemas. Um programa escrito em MIL-75, formaliza as interdependências entre os módulos através de uma lista de recursos fornecidos e requeridos em cada módulo. Um

sistema é obtido garantindo que para todos recursos importados por um módulo existem um ou mais módulos que os fornecem. A composição hierárquica é conseguida através da combinação de subsistemas de software existentes.

Na linguagem MIL-75 são introduzidos conceitos que permitem a descrição do sistema num sentido puramente estrutural; informações de como a estrutura será implementada fisicamente não são disponíveis através de construções de linguagem.

Em sistemas distribuídos para a composição do sistema é necessário que se leve em conta três aspectos importantes:

- **A estrutura lógica do sistema:** que envolve a definição dos componentes de software do sistema e suas interconexões;
- **A estrutura física do sistema:** descreve os recursos físicos disponíveis no sistema;
- **Mapeamento lógico - físico:** definindo como as estruturas lógicas podem ser mapeadas nos recursos físicos do sistema.

Neste sentido, uma linguagem de interconexão de módulos, em sistemas distribuídos deve evoluir para uma ferramenta básica que permita então, o mapeamento lógico-físico através de comandos possibilitando: a identificação de componentes (módulos e/ou subsistemas), a atribuição destes às unidades de execução, a criação de instâncias dos componentes e a interconexão das mesmas. É sempre importante para a utilização do software que existam construções nestas linguagens que permitam tanto a inclusão de componentes (módulos ou subsistemas) bem como a

especificação de subsistemas, hierarquizando a construção de sistemas.

A presença de construções que permitam a especificação de subsistemas e a definição de componentes (módulos e subsistemas) a partir de tipos (similar a tipos abstratos de dados), nestas linguagens, é sempre importante pois favorece a reutilização do software.

2.4. ATIVIDADES RELACIONADAS COM A PROGRAMAÇÃO EM LARGA ESCALA

Um objetivo sempre perseguido, em se tratando de softwares complexos, é a evolução de sistemas. Esta evolução não representa simplesmente uma evolução do código, mas deve se refletir sobre todas as atividades do processo de desenvolvimento do software. Uma decorrência da evolução do software é a existência de versões de módulos e também de sistemas; estas devido à existência de várias versões de módulos.

O problema de suportar a evolução é visto como o controle das dependências entre módulos (recursos requeridos ou fornecidos) e da proliferação de versões resultante da evolução. Neste sentido, a programação em larga escala se apresenta relacionada com diversas atividades das diferentes fases do ciclo de vida do software. Estas se apresentam na literatura, normalmente, concentradas no gerenciamento de configuração e controle de versão. Neste item são descritos de forma sucinta, algumas atividades ligadas a programação em larga escala.

2.4.1. Gerenciamento de configuração

O gerenciamento de configuração é uma disciplina emergente, cujo objetivo principal é gerenciar versões distintas do sistema que possam vir a ser produzidas a partir de componentes (módulos e subsistemas), permitindo desta maneira que a existência destas versões de configuração sejam metodicamente controladas.

O gerenciamento de configuração envolve atividades responsáveis pelos mecanismos de controle necessários para que os seguintes objetivos sejam atingidos:

- **Identificar a configuração**: Definir os componentes integrantes do sistema e fornecer atributos para a identificação dos mesmos;
- **Controlar a configuração**: Fornecer mecanismos para controlar a evolução do software e definir a estratégia de mudanças;
- **Manter a integridade do sistema**: Garantir que os recursos requeridos por um módulo sejam fornecidos por um outro módulo.

Esta disciplina, tratando com os efeitos da modularidade nos sistemas evolutivos, deve garantir a consistência entre a especificação e o estado atual do sistema. Para tanto, o gerenciamento de configuração deve manter registros de todas as atividades exercidas durante a evolução do sistema.

As dificuldades encontradas no gerenciamento da configuração estão fortemente relacionadas com a complexidade do sistema. As características de interface do módulo e as

interdependências entre os mesmos são representativas quando se considera as dificuldades em manter a configuração ou realizar mudanças no sistema. Por exemplo, se um módulo apresenta na interface uma especificação de recursos fornecidos e requeridos esta dependência deve ser preservada na evolução do software.

2.4.2. Controle de versão

O controle de versão é uma atividade de programação em larga escala que requer uma atenção especial devido as inevitáveis mudanças durante o ciclo de vida do software. Para atender a estas mudanças pode ser necessário reescrever uma nova versão do código que satisfaça objetivos específicos. Embora os sistemas de controle de versão tenham sido tradicionalmente voltados para o código, os sistemas mais complexos estão forçando o reconhecimento da importância em manter um relacionamento entre as diferentes versões da especificação, documentação e implementação (Ramamoorthy, 87).

Para tratar dos problemas de controle de versão, técnicas e modelos para controlar e gerenciar múltiplas versões de módulos têm sido desenvolvidos. Ferramentas como Sistema de Controle de Código Fonte (SCCS - Source Code Control System) (Rochkind, 75) ou Sistema de Controle de Versão (RCS - Revision Control System) (Tichy, 85) apresentam mecanismos eficientes para armazenar o código fonte e suas versões sempre que ocorre alguma alteração no texto fonte. Estes mecanismos consistem basicamente em atribuir um número, correspondente a versão, e incrementá-lo toda vez que o código for alterado. Estas ferramentas simplesmente gravam as versões e coordenam o acesso às mesmas

ficando a critério do usuário decidir como usar as informações registradas.

2.4.3. Processo de Desenvolvimento de Software Distribuído

A aplicação do conceito de modularidade para reduzir o nível de complexidade dos grandes sistemas como visto, introduz outros problemas. A aplicação de técnicas e métodos convencionais quando aplicados nestes sistemas não apresenta a mesma eficiência. A complexidade das atividades tem direcionado os ambientes de desenvolvimento de software, no sentido da integração de ferramentas segundo metodologias que atuam nas diversas fases do ciclo de vida do software.

A estratégia normalmente adotada para o desenvolvimento de software está baseada no modelo do ciclo de vida que consiste no sequenciamento das fases de análise de requisitos e especificações, projeto, implementação, testes e manutenção (Yau, 86). As metodologias hoje conhecidas não cobrem a totalidade das fases de desenvolvimento e, por outro lado, muitas vezes são próprias para um determinado tipo de aplicação.

O princípio de decomposição modular ou ainda a programação modular (como metodologia da fase de implementação) se adapta perfeitamente a metodologias de especificação e projeto que seguem tanto a "orientação por processos", como as ditas "orientados por dados". As metodologias orientadas por processos compreendem as técnicas descendentes (decomposição "top-down") que enfatizam a decomposição e estruturação dos componentes, aplicando sucessivamente a técnica de divisão para se obter componentes mais simples. As metodologias orientadas por dados se utilizam da

composição de componentes sucessivamente mais complexos a partir de outros mais simples (composição "bottom-up") (Yay, 86).

Em (Weber, 86) é proposta uma metodologia descendente para sistemas modulares. A metodologia proposta define uma família de linguagens de especificação que pode ser aplicada em todas as fases do desenvolvimento. O emprego destas linguagens, em ordem sucessiva, divide em três fases o ciclo de desenvolvimento do software. A primeira linguagem suporta uma especificação, em linguagem natural, da estrutura inicial do software do sistema. Esta linguagem embora dita informal, apresenta uma sintaxe definida e quando de sua utilização os produtos resultantes são aproximações dos componentes modulares. A segunda linguagem completa a definição da sintaxe dos componentes modulares produzindo uma especificação "semiformal" que se caracteriza pela indefinição da semântica. Esta especificação semiformal pode então ser facilmente convertida em uma implementação em qualquer linguagem que suporte a modularidade. A última linguagem define um modelo formal que é utilizado na verificação da correção das especificações.

Metodologias de especificação ditas de "tipo abstrato de dados" que apresentam características de composição "bottom-up" são também adaptáveis a programação modular. Um exemplo destas metodologias é a metodologia orientada para objetos (Booch, 86), (Bruno, 86). Estas metodologias são consideradas como refletindo uma melhor correspondência entre abstrações e os objetos do mundo real. Mudanças, nestes casos, apresentam efeitos bem mais localizados do que em metodologias descendentes, que dividem já o sistema no mais alto nível de especificação.

O processo de desenvolvimento de um sistema devido a modularidade se apresenta dividido na programação em pequena escala cuja responsabilidade é a implementação dos módulos, e a programação em grande escala que deve estruturar o sistema, incorporando atividades de configuração, manutenção e controle de versão. Neste sentido, em (Ramamoorthy, 86) são classificadas técnicas e ferramentas utilizáveis na programação em larga escala. Estas técnicas e ferramentas estão divididas em dois grupos: técnicas independentes e técnicas dependentes de fase. Para atender a todas as necessidades na programação em larga escala, o ambiente de desenvolvimento deve então, combinar princípios da engenharia de software, de linguagem de programação, de base de dados e de inteligência artificial.

2.5. TRABALHOS RELACIONADOS

Muitas linguagens têm sido propostas, algumas já disponíveis comercialmente, com características voltadas para a programação em larga escala. Esta seção apresenta algumas linguagens sob o ponto de vista de facilidade de estruturação de software e flexibilidade frente a modificações durante a operação do sistema.

2.5.1. Sistema MESA

O ambiente de programação MESA (Sweet, 85) foi um dos primeiros sistemas a adotar o princípio de decomposição modular. O ambiente suporta uma programação modular com forte verificação de tipo em tempo de compilação.

As unidades de programação são os módulos objetos gerados pelo compilador MESA. Existem dois tipos de módulos: módulo de definição (interface) e módulo programa (corpo), sendo que ambos podem ser compilados separadamente. O primeiro define as interfaces e os tipos a serem compartilhados pelos módulos programas. O módulo programa é tipicamente constituído por uma coleção de declarações de inclusão e o código principal. A compilação de um módulo definição não gera códigos mas sim uma tabela de símbolos.

Os aspectos de configuração são resolvidos pelo carregador em tempo de execução ("Run-time Loader") e o Ligador ("Binder"). O carregador é capaz de ligar módulos carregados separadamente e é considerado tecnicamente como parte do núcleo do sistema operacional Pilot (Redell, 79) e não como uma ferramenta de programação. Este basicamente deve alocar memória para os módulos, carregá-los e resolver os itens da interface baseados nas informações de módulos já carregados que estão registradas na base de dados mantida pelo mesmo.

Em sistemas de maior porte é conveniente unir vários módulos em um único arquivo objeto; o Ligador é o programa responsável por esta operação. A entrada para o Ligador é uma descrição que contém uma lista de arquivos objetos para serem combinados num arquivo objeto maior.

A Linguagem de Configuração - C/Mesa (Mitchell, 79) é a ferramenta utilizada nesta descrição que serve de entrada às duas ferramentas. A separação entre a programação de módulos e a configuração não é completamente distinta, uma vez que, um

módulo pode conter em sua descrição a criação de uma instância em tempo de execução (Geschke,77).

A conexão entre as instâncias é estabelecida durante a criação das mesmas por parametrização dos itens da interface, comprometendo portanto a flexibilidade, exigindo que a alteração de uma conexão ocorra através de uma destruição da instância correspondente.

2.5.2. Sistema PRONET

A linguagem de programação distribuída, PRONET (Maccabe, 82), apresenta uma separação completa das atividades de programação e configuração de sistema através de duas linguagens complementares, ALSTEN e NETSLA. Um programa escrito em PRONET pode ser visto como uma "rede de comunicação lógica" onde os nós são os processos definidos pela linguagem de programação, ALSTEN, e os arcos entre os nós representa a ligação de comunicação lógica definidas por NETSLA. NETSLA é uma linguagem baseada em evento que descreve uma configuração e as modificações na rede de comunicação lógica.

A rede de comunicação lógica inicia a execução dos processos e descreve as atividades que devem ser realizados na ocorrência de um evento. Esta pode criar, terminar um processo, conectar/desconectar portos e manipular os eventos sinalizados pelo programa ALSTEN.

O processo é um componente de software que realiza operações em um espaço de endereçamento local e que se comunica através de portos caracterizando um mecanismo de comunicação totalmente independente da configuração do sistema. Estes portos

são visíveis à especificação de rede onde a troca de mensagens é efetivamente tratada. A execução de uma operação de envio causa a transmissão de um evento que é sinalizado à rede lógica com o objetivo de executar as atividades especificadas na mesma no sentido de transmissão (ou propagação) do mesmo.

As características apresentadas por PRONET permite que o sistema se adapte facilmente a configurações pré-planejadas, porém como apontado em (Maccabe, 82) o suporte em tempo de execução e o excesso de computação limitam a aplicação geral da linguagem em um ambiente distribuído.

2.5.3. Sistema MASCOT

A construção de software em MASCOT (Simpson, 79) consiste em combinar subsistemas através de facilidades de construção suportadas pelo núcleo. O subsistema consiste de uma ou mais atividades (processos) que interagem através de "área de dados para intercomunicação". As funções de uma atividade são definidas por um programa chamado "root procedure", cujos parâmetros formais especificam a área de comunicação (canal ou "pool") e o tipo (entrada ou saída). O canal é usado para transmissão de mensagem unidirecional entre atividades e o "pool" é uma área destinada a dados permanentes. O acesso às áreas de dados é obtida através de um procedimento de acesso - "access procedure".

As facilidades de construção do sistema são realizadas através de comandos convencionais descritos por COMPILE, LOAD e LINK adicionados de um comando especial - FORM que cria os

subsistemas. Alternativamente podem ser fornecidas facilidades para a remoção de subsistema em tempo de execução.

A flexibilidade a mudanças é limitada devido a não separação entre a função de instanciação e interconexão, sendo que ambas são realizadas através de um único comando (FORM). As alterações na estrutura do software são possíveis, porém exige uma nova versão do núcleo que apresenta um "overhead" consideravelmente maior do que o convencional.

2.5.4. Sistema CONIC

A implementação de um sistema CONIC (Kramer, 83) é realizada através de duas linguagens: Linguagem de Programação (Conic/P), usada para programar os módulos e uma Linguagem de Configuração (Conic/C), usada para interconectar instâncias dos tipos módulos.

A unidade básica para a estruturação do software é o módulo-tarefa que define um tipo a partir do qual instâncias podem ser criadas. A comunicação entre módulos é realizada pelo envio e recepção de mensagens a portos de saída e entrada respectivamente.

A descrição da configuração é realizada pela Linguagem de Configuração, cujas unidades manipuladas podem ser os módulos definidos por Conic/P ou subsistemas (grupos) compostos por outros módulos. Estes subsistemas são definidos usando a linguagem CONIC/C.

A Linguagem de Configuração também pode ser utilizada para especificar uma configuração de módulos em um nó lógico que é a

unidade básica de configuração dinâmica do sistema (Magge, 87). O nó lógico é composto por um conjunto de módulos e o suporte de tempo de execução. Assim sendo, um programa de aplicação distribuído, escrito em CONIC, consiste de um ou mais nós lógicos interconectados. Uma vez configurado, a estrutura do sistema pode ser modificada através de uma nova descrição de configuração. Este processo é executado por um gerenciador de configuração que traduz as especificações e executa as operações de reconfiguração.

A separação das atividades de programação e configuração é rigorosamente cumprida. As características apresentadas pelas linguagens permitem a concepção de estruturas de software flexíveis, atendendo a requisitos de alterações durante o funcionamento do sistema.

2.5.5. Linguagem de Programação ADA

A linguagem de programação ADA, (Mod 82), embora projetada para programar sistemas embutidos para aplicações em tempo real, tem sido largamente citada na literatura e indicada como modelo devido a um conjunto de atributos que possui.

O módulo em ADA pode ser representado por subprograma, pacote ou tarefa. Nas aplicações de ADA, o módulo é chamado de unidade e sua implementação é realizada através de duas partes separadas: a primeira se refere a especificação da unidade, contendo todos os itens exportados e a segunda ao corpo da unidade. A importação de entidades é obtida através de uma instrução especial que precede o corpo da unidade. A compilação separada pode ser realizada porém obedecendo certas restrições

entre as unidades: a compilação das especificações deve ser realizada antes da compilação dos corpos que as utilizam.

A comunicação entre as unidades é estabelecida através de um esquema de nomeação assimétrico, onde a unidade que inicia a comunicação necessita identificar a unidade de destino. Este mecanismo, conhecido como rendez-vous, compromete a separação da programação das unidades daquela da configuração do sistema. Conseqüentemente a estrutura do sistema escrito em ADA está embutida no texto a ser carregado e para o qual será criada a estrutura de dados como um todo. Neste caso, o processo de modificação de código na evolução, implica na recompilação e carregamento de todo o sistema.

Um certo grau de flexibilidade é obtido através da criação dinâmica de tarefas e sua conexão com as outras tarefas. Porém a limitação ocorre devido a criação de tarefas ser somente de tipos já existentes na configuração, o que diminui as características de flexibilidade.

2.6. CONCLUSÃO

O estudo bibliográfico e conceitual apresentado mostra que a necessidade em se adotar uma sistemática no desenvolvimento de sistemas complexos é imprescindível. A produção destes sistemas deve ser suportada por ferramentas que automatizem, tanto quanto possível, a parte não criativa do desenvolvimento.

O breve estudo sobre linguagens mostra claramente as facilidades e os suportes fornecidos por estas como respostas ao desejo de se obter um sistema flexível e evolutivo; embora

muitas vezes os resultados obtidos atendem somente a uma parcela das expectativas iniciais.

A adoção do princípio de decomposição modular como solução para a obtenção de sistemas evolutivos é largamente reconhecida, apesar de alguns sistemas não o cumprirem rigorosamente.

CAPÍTULO 3

CONFIGURAÇÃO DE UM SISTEMA DISTRIBUÍDO PARA APLICAÇÕES EM TEMPO REAL

3.1. INTRODUÇÃO

O objetivo deste capítulo é apresentar um ambiente de programação e descrever os aspectos de configuração de um sistema distribuído. A programação distribuída está centrada em uma Linguagem de Implementação de Sistemas (LIS) que encapsula todos os aspectos da arquitetura do software para aplicações em tempo real. Esta linguagem é parte de um conjunto de ferramentas a ser fornecido em um Ambiente de Desenvolvimento e Execução de Software - ADES (Fraga, 88) que está sendo desenvolvido no LCMI/UFSC. Este ambiente deve fornecer facilidades de programação e prover meios para a investigação de uma variedade de problemas relacionados com a construção e manutenção de software distribuído para aplicações em tempo real.

Os aspectos de configuração são descritos por uma sub-linguagem LINCS (Linguagem de Configuração de Sistemas) que incorpora as características descritas no capítulo anterior permitindo ao programador de sistema ativar o processo de configuração desejado. O desenvolvimento do processador da linguagem e os mecanismos de configuração são apresentados no sentido de atender as necessidades de uma configuração estática.

3.2. DEFINIÇÃO DO SISTEMA

O sistema considerado destina-se ao desenvolvimento de aplicações distribuídas encontradas nos ambientes de automação industrial e controle de processos. As necessidades destas aplicações devem ser atendidas a partir de facilidades de construção, orientadas para a obtenção de sistemas que cumpram então os requisitos de tempo real, flexibilidade, confiabilidade, etc.

A concepção e estruturação de programas distribuídos neste ambiente segue um paradigma que corresponde ao modelo de programação CONIC (Kramer, 85) modificado. Considerando a natureza das aplicações a que se destina (automação industrial e controle de processo), este modelo visa atender requisitos de flexibilidade, permitindo então, a concepção de sistemas evolutivos. Neste sentido é incluído um conjunto de ferramentas e serviços básicos de modo a permitir a implementação, execução e evolução do software distribuído, segundo o paradigma utilizado.

3.2.1. Arquitetura do Sistema

A arquitetura do sistema consiste de um conjunto de estações (compatíveis IBM-PC) interligadas via rede local. O conjunto de estações é composto de Estações de Execução e uma Estação de Trabalho, conforme ilustrado na Fig. 3.1.

O desenvolvimento do software é realizado na Estação de Trabalho onde se encontra um conjunto de ferramentas que atuam nas diferentes fases do ciclo de vida do software. Após implementado e testado, o software pode ser então carregado nas

Estações de Execução estabelecendo a configuração do sistema. Neste sentido, a organização da Estação de Trabalho, fortemente influenciada pelas atividades de desenvolvimento de software, deve conter um Sistema Operacional Hospedeiro fornecendo os serviços básicos (operações com arquivos, compilação, edição, etc.) e um Sistema Operacional Distribuído (SOD). A Fig. 3.2 ilustra a arquitetura de software da Estação de Trabalho.

O SOD é construído segundo a filosofia de composição de componentes e é formado por módulos gerenciadores e por um interfaceador com o Sistema Hospedeiro. Os gerenciadores são responsáveis pelas operações com módulos, ligação de portos, tratamentos de erros em tempo de execução, carregamento remoto, etc; enquanto que o módulo interfaceador torna sequencial as chamadas ao Sistema Operacional Hospedeiro.

A Estação de Execução apresenta uma estrutura similar a Estação de Trabalho, diferindo apenas nas funções operacionais de forma a executar os serviços específicos de cada estação. Os recursos disponíveis são mais modestos, dispensando a necessidade do Sistema Operacional Hospedeiro. O SOD neste caso incorpora uma versão mais simples dos gerenciadores, cujas funções se restringem a operações de controle de execução das instâncias, controle de erros e de acesso remoto às memórias.

3.2.2. Paradigma de Programação

O domínio da complexidade na programação distribuída depende em grande parte dos chamados paradigmas de programação. Estes fornecem uma disciplina para a concepção e a estruturação de softwares distribuídos.

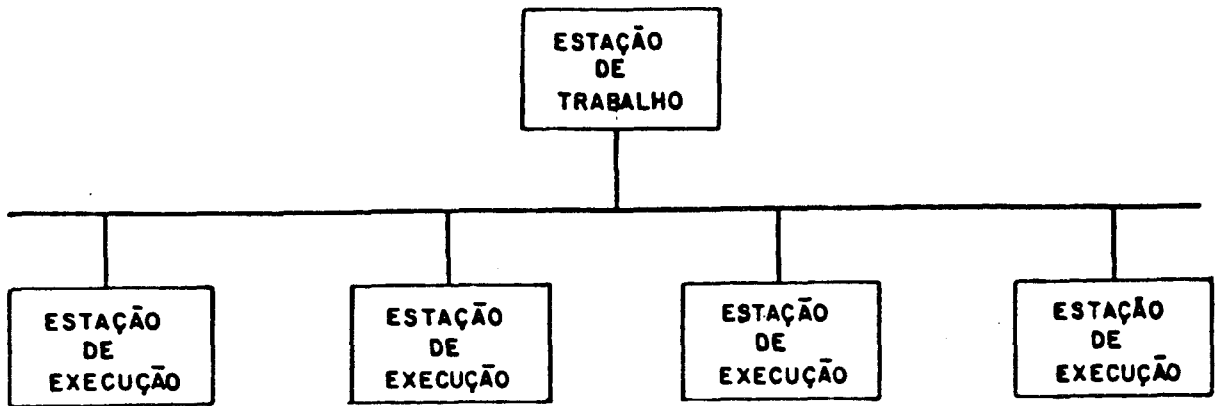


Fig. 3.1 - Arquitetura Física do Sistema

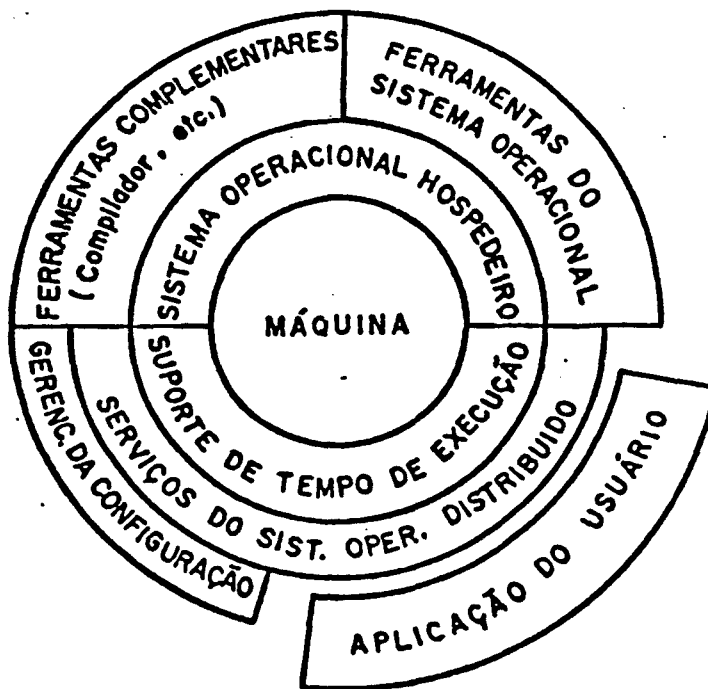


Fig. 3.2 - Arquitetura de Software da Estação de Trabalho

O desenvolvimento e concepção de sistemas complexos está baseado fundamentalmente nos conceitos introduzidos por (DeRemer, 76). O paradigma de programação proposto incorpora estes conceitos e permite a construção do sistema através de um conjunto de módulos. Para ser eficiente neste objetivo o paradigma deve estar contido na linguagem de implementação e suportado pelo ambiente de tempo de execução (Núcleo de Tempo Real e serviços do SOD).

O módulo é uma unidade encapsuladora de uma ou mais tarefas. Um tipo módulo é definido a partir da linguagem LINCE, permitindo então a criação de instâncias para a execução. A instanciação de um tipo módulo é o processo de atribuição de uma área de dados relativa ao módulo, estabelecendo um contexto de execução.

O módulo é considerado a menor unidade disponível para a construção do sistema. Em tempo de execução, a noção de modularidade é substituída pela noção de concorrência; sendo assim, o programa distribuído é constituído por um grupo de tarefas (unidade elementar de concorrência) que se executam e concorrem nos diversos processadores do sistema.

O endereçamento de tarefas e o armazenamento temporário de mensagens são resolvidos classicamente no modelo "troca de mensagens", pela noção de "porto". No modelo utilizado, tarefas enviam mensagens através de portos de saída e recebem via portos de entrada. Os portos de saída são conectados aos portos de entrada em tempo de configuração, formando então, canais de sincronização e comunicação entre tarefas. Neste modelo são

previstas comunicações síncronas e assíncronas com diferentes tipos de conexão (um-para-um, um-para-muitos, muitos-para-um).

Na Fig. 3.3, é apresentado um diagrama lógico de uma configuração de componentes para ilustrar o paradigma de programação adotado.

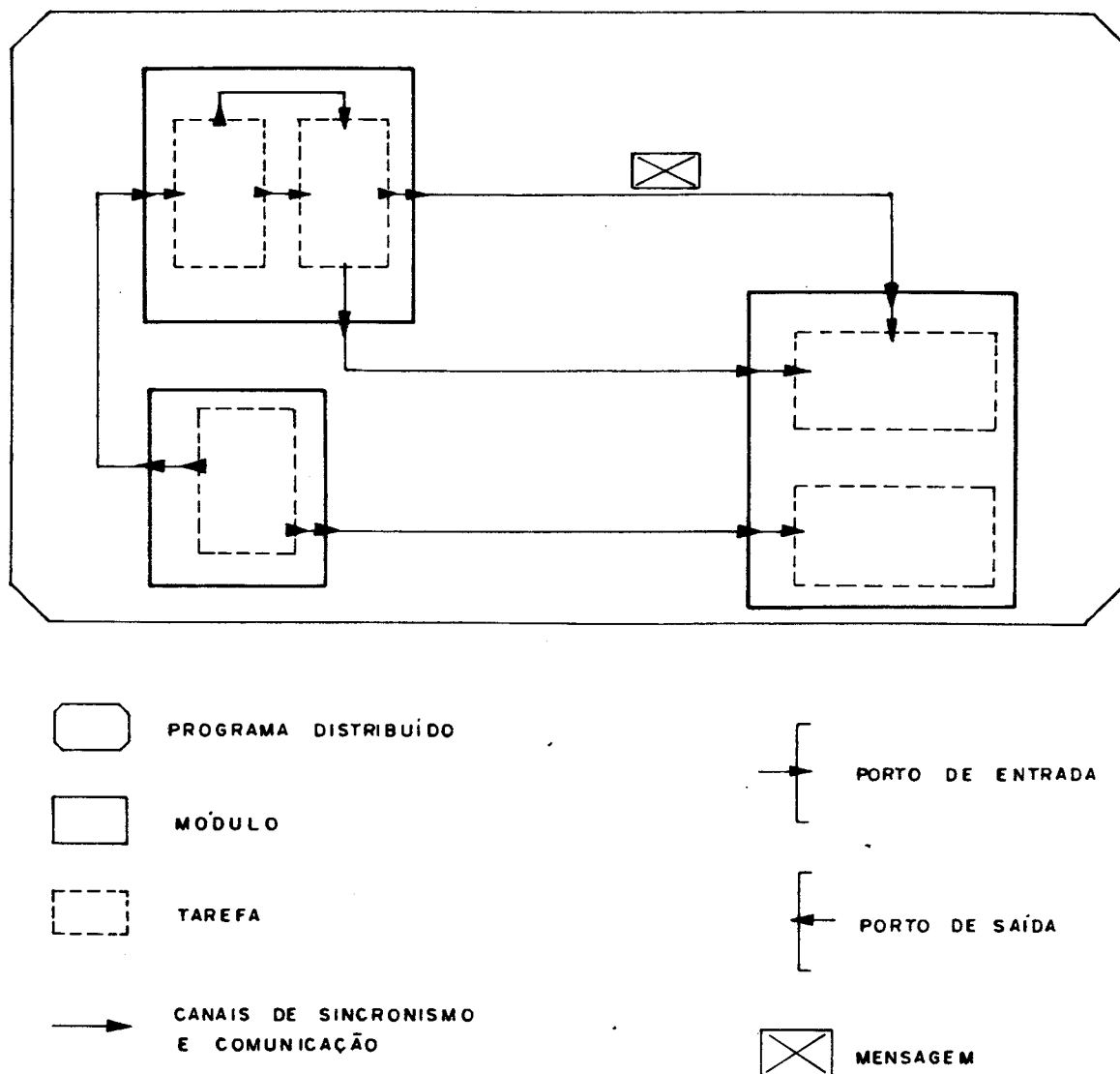


Fig. 3.3 - Paradigma de programação

3.2.3. Linguagem de Implementação de Sistemas

A Linguagem de Implementação de Sistemas (LIS) é o instrumento pelo qual o paradigma pode ser representado,

fornecendo facilidades ao usuário para estruturar e conceber sistemas. A fidelidade em representar o modelo adotado garante uma alta compreensibilidade e uma alta flexibilidade na estrutura do sistema. Neste sentido a Linguagem de Implementação de Sistemas é composta por uma Linguagem de Componentes Elementares - LINCE (Silva, 88) e uma Linguagem de Configuração de Sistemas - LINCS.

A programação dos componentes elementares (módulos) é realizada através da Linguagem de Componentes Elementares (LINCE), cujo objetivo é permitir a definição de tipos a partir dos quais podem ser criadas instâncias que serão interconectadas logicamente e mapeadas em entidades físicas pela Linguagem de Configuração. As características apresentadas pela LINCE favorecem a reutilização de software e a abstração da implementação interna do módulo, uma vez que não há qualquer referência à configuração embutida nos comandos da linguagem.

O compilador LINCE gera um arquivo de código realocável e um arquivo de dados contendo informações do tamanho da área de códigos, do tamanho da área de dados estáticos e dinâmicos, informações de portos e parâmetros do módulo.

A Linguagem de Configuração de Sistemas (LINCS) é a ferramenta destinada à construção de sistemas distribuídos a partir de tipos módulos escrito em LINCE. As características e a sintaxe desta linguagem são discutidas nas próximas seções.

3.2.4. Suporte de Tempo de Execução

O responsável pela implantação do ambiente multitarefas que executa as aplicações distribuídas e que fornece serviços

básicos para o gerenciamento da configuração do sistema é o Núcleo de Tempo Real. Este é constituído por um conjunto de primitivas acessadas em grande parte por tarefas em tempo de execução e que devem estar disponíveis ao programador da aplicação embutidas nos comandos da Linguagem de Implementação de Sistemas (LIS).

O Núcleo de Tempo Real fornece as seguintes funções (Nacamura, 88):

- **Suporte de Gerenciamento Local:** Fornece um conjunto de primitivas relacionadas com atividades de configuração. Entre elas podemos citar: carregar/remover tipo módulo, criar / destruir instância de módulo, ligar / desligar / religar portos. Estas operações são disponíveis aos gerenciadores locais que participam da configuração da aplicação.
- **Gerenciamento de Tarefas:** envolve primitivas para o manuseio de tarefas (criar/destruir/suspenso tarefas, modificar prioridade) e políticas de ocupação do processador. O escalonador de tarefas é "event-driven"- segue uma estratégia de pré-esvaziamento por prioridades.
- **Comunicação e Sincronização entre Tarefas (IPC):** fornece um conjunto de primitivas que suportam as diferentes operações de comunicação e sincronismo previstos no paradigma de programação. Fazem parte deste conjunto de primitivas, indicadores de exceções que permitem ao programador planejar estratégias de recuperação de erros de comunicação.

- **Gerenciamento de Memória:** supre as necessidades de memória através de dois gerenciadores: o primeiro atuando sobre a área de alocação dinâmica do núcleo e o segundo em área externa ao mesmo. Ambos seguem a abordagem "first-fit" e possuem a função de dealocação de memória.

Outros serviços presentes no núcleo são o de temporização e o de tratamento de interrupção que estão relacionados respectivamente, com o relógio de tempo real e o processamento de eventos aleatórios (ou externos).

O suporte de tempo de execução deve permitir a comunicação e sincronização entre tarefas de forma uniforme, independente da distribuição destas no sistema. A extensão do IPC (comunicação inter-processo) de forma transparente sobre a rede de computadores é feita usando tarefas "servidores de rede". Os requisitos colocados a um servidor de rede são: fornecer alguma forma de comunicação entre as estações e acomodar as semânticas das primitivas de IPC (definidas sobre bases locais) em comunicações remotas.

3.3. MODELO DE CONFIGURAÇÃO

O propósito de um modelo de configuração é definir o meio de expressão e a forma de configuração dos módulos de modo que, o responsável pela construção do sistema possa interagir com o ambiente, especificando o que deve ser feito. As idéias e a estrutura imaginada pelo programador devem ser organizadas em um texto fonte, denominado "especificação de configuração", utilizando-se de uma linguagem de configuração. A especificação de configuração é um conjunto de comandos da linguagem que

descreve a estrutura global do sistema. Esta deve identificar os módulos que constituem o sistema (definição de contexto); determinar a criação de instâncias dos tipos módulos; e interconectar estas instâncias formando canais de comunicação e sincronização entre tarefas.

Uma especificação de configuração completa deve descrever, além da estrutura lógica do software, o mapeamento lógico/físico que será executado durante o processo de configuração. Este último, por sua vez, consiste em traduzir a especificação de configuração e executar, de acordo com o tipo de configuração, os procedimentos envolvidos na construção do sistema.

O modelo de configuração, suportado pelo Sistema Operacional Distribuído-SOD, permite construir o sistema inicial através de um processo denominado "configuração estática" e introduzir mudanças na estrutura através de um processo dito "configuração dinâmica".

3.3.1. Configuração Estática

A configuração estática consiste na construção do sistema (configuração inicial) segundo especificações iniciais. Em sistemas que só comportam mecanismos para a configuração estática, qualquer modificação no sistema deve passar por uma nova construção (uma nova configuração total do sistema). Esta abordagem de construção tem sido adotada por muitos sistemas de programação distribuída (Mesa, Starmod, SR,..etc.). O processo de configuração estática é ilustrado na Fig. 3.4.

Nesta figura, o configurador é mostrado, a partir do sucesso na tradução da especificação inicial, construindo

"imagens de carga" (item 3.5.2.2) dos códigos aplicativos e de suporte, disponíveis no sistema de arquivos da Estação de Trabalho, para serem carregados nas diversas estações do sistema.

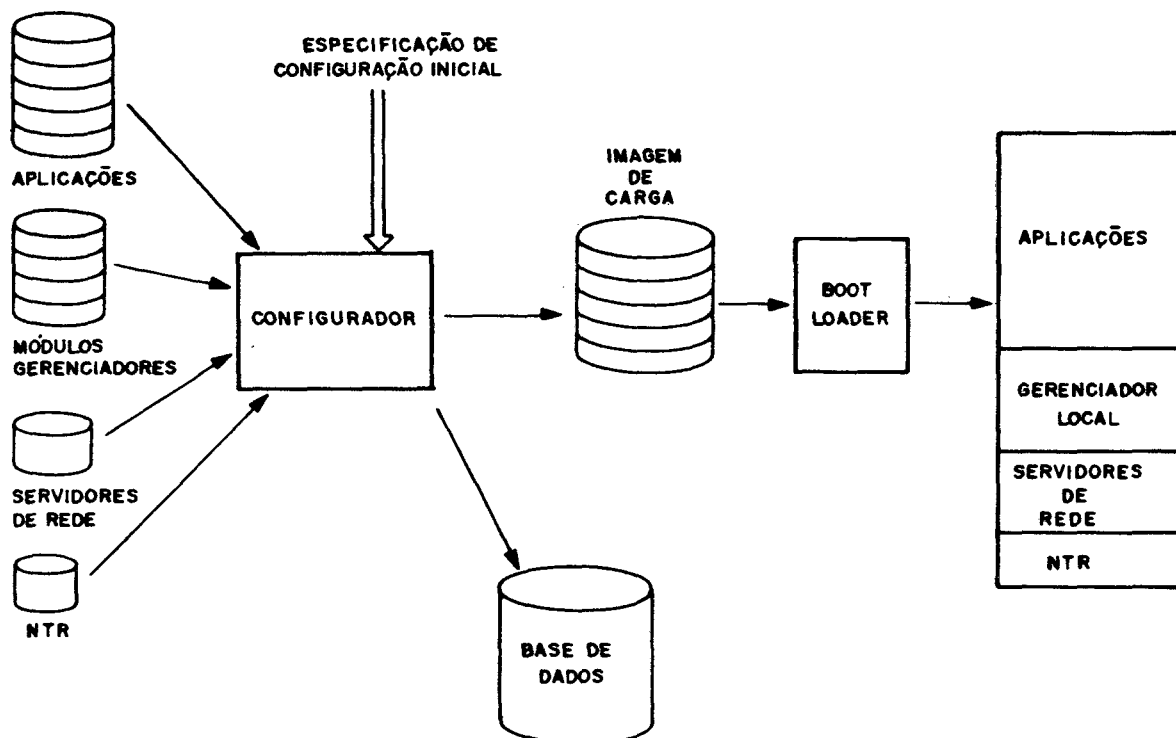


Fig. 3.4 - Processo de Configuração Estática

3.3.2. Configuração Dinâmica

A configuração dinâmica é o processo que permite introduzir mudanças "on-line" no sistema, sem a necessidade de passar por um processo de configuração total (Kramer, 85). Este processo é importante em aplicações de automação industrial onde não se pode interromper o processo controlado ou ainda diante de falhas parciais no sistema.

Para suportar a configuração dinâmica, o SOD deve ser constituído por módulos gerenciadores que executam operações sobre os tipos módulos, instâncias e nas interconexões de portos. A seguir é apresentado uma descrição suscinta dos módulos gerenciadores:

- **Gerenciador de Módulo:** É o responsável pelas operações com tipos módulos e instâncias. Os tipos módulos podem ser carregados ou removidos do contexto da estação; as instâncias podem ser criadas, destruídas e controladas em relação a execução. Todas estas operações são disponíveis a partir de primitivas do NTR;
- **Gerenciador de Link:** trata com a conexão e desconexão de portos de instâncias. Estas operações são realizadas sobre portos de saídas locais (estabelecendo conexões locais ou remotas) utilizando-se de primitivas no NTR;
- **Gerenciador de acesso remoto à memória:** este gerenciador deve permitir a escrita remota de blocos de memória. A utilidade deste gerenciador está ligado a operações de carregamento de tipos módulos quando do processo de configuração dinâmica.
- **Outros módulos gerenciadores:** Devido a concepção modular, outros módulos gerenciadores podem ser implementados e incluídos no SOD para atender uma operação específica do usuário.

A Fig. 3.5 ilustra um processo de configuração dinâmica em adição ao processo anterior (configuração estática). Este processo é ativado após a tradução de uma especificação de

mudanças. Como resultado desta tradução é gerada uma versão temporária da base de dados e uma lista de ações que é interpretada e executada pelo configurador através de mensagens enviadas a entidades de gerenciamento do SOD. O configurador consulta a base de dados, acessa o sistema de arquivos e define estratégias de mudanças afim de controlar a evolução do sistema.

Os novos módulos de aplicação a serem introduzidos devem ser acessados no sistema de arquivos, realocados e carregados nas Estações de Execução. O sucesso da operação implica na gravação da base de dados temporária que passa ser a nova base de dados do sistema.

Caso ocorra falha na configuração, as operações já realizadas devem ser desfeitas conduzindo o sistema ao estado original com o conseqüente abandono da base de dados temporária e da lista de ações. O processo de configuração deve em qualquer situação garantir uma consistência entre a representação descrita na base de dados e o estado final do sistema.

3.4. LINGUAGEM DE CONFIGURAÇÃO DE SISTEMAS (LINCS)

A LINCS é uma linguagem declarativa que deve descrever estrutura lógica do software distribuído e o mapeamento lógico/físico do sistema.

O modelo de estruturação do software suportado por LINCS permite a representação explícita dos aspectos de distribuição e facilidades para a construção hierárquica de sistema. Neste contexto, as características mais interessantes do modelo LINCS são:

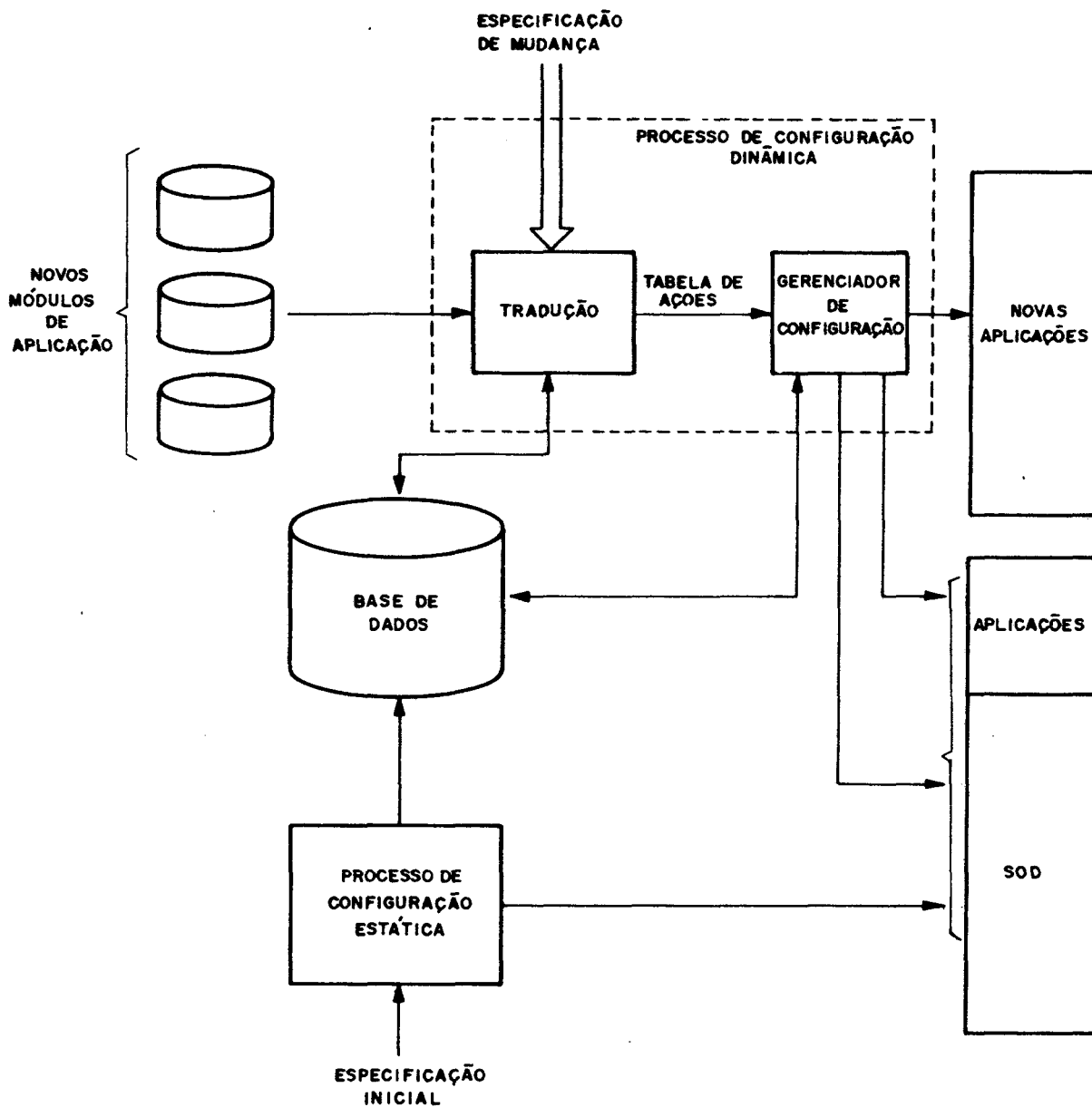


Fig. 3.5 - Processo de Configuração Dinâmica

- **Configuração hierárquica:** A linguagem permite definir subsistemas através de construções que definem módulos a partir de módulos mais elementares;
- **Reutilização de software:** Este conceito é apoiado pela instanciação de módulos e grupos (item 3.4.1.8), facilitado pelas características de parametrização formal;
- **Separação de Funções:** Declaração independente para cada operação necessária na configuração de um sistema;
- **Atribuição de módulos às unidades de execução :** Definição explícita da estações onde serão carregados os módulos;
- **Facilidades para a especificação de mudanças em configuração dinâmica:** esta característica está fundamentada na proibição de nomeação direta a entidades externas aos módulos.

Uma separação clara entre a programação de módulos e a configuração de sistemas, onde nos módulos as referências são todas locais, e a separação explícita na linguagem de configuração das operações envolvidas na construção de sistemas, permitem então que se obtenha sistemas evolutivos e passíveis de mudanças. A linguagem LINCS foi especificada para conter estes atributos.

3.4.1. Declarações LINCS

O conjunto de declarações da linguagem pode ser classificado em: auxiliares, básicas, inversas e declarações de tipo de especificação.

As declarações classificadas na categoria auxiliar consistem em declarar constantes (CONST), famílias (FAMILY) e portos (PORT). As duas primeiras declarações foram introduzidas com o objetivo de facilitar a edição da especificação de configuração, enquanto que a última foi necessária para permitir a abstração de configuração através da especificação de grupo, motivo pelo qual o seu uso se restringe a este tipo de especificação.

As declarações básicas descrevem a estrutura do software e consistem em definir o contexto (USE), criar instâncias (CREATE) e interconectar módulos através de ligações de portos (LINK). A separação explícita de funções pode ser observada neste conjunto através de declarações independentes para criar e interconectar instâncias.

As declarações inversas realizam as funções de destruir instâncias (DELETE), desligar portos (UNLINK) e remover tipos módulos do contexto (REMOVE). A aplicabilidade destas declarações se limita a especificação de mudanças na estrutura.

Finalmente, a definição do tipo de especificação é obtida através de declarações SYSTEM, GROUP MODULE e CHANGE correspondentes a especificação inicial, abstração de configuração e especificação de mudanças, respectivamente.

A seguir, neste item, são apresentados a sintaxe e os aspectos da semântica dos comandos LINCS.

3.4.1.1. Notação Utilizada

A apresentação da linguagem será baseada na definição da sintaxe, utilizando um formalismo derivado da notação BNF, e na descrição informal da semântica. A simbologia utilizada para a descrição formal é apresentada a seguir:

:=	definido por;
 	alternativa;
[]	opcional;
[x]*	0 ou mais ocorrências de x;
[x]+	1 ou mais ocorrências de x; e
(x y)	agrupamento.

As palavras reservadas aparecerão em letras maiúsculas e os símbolos terminais entre aspas.

3.4.1.2. Declaração de Constantes

Semântica:

A declaração de constante pode ser usada para introduzir identificadores representando um valor específico. Devido a aplicação que se destina este comando o valor a ser atribuído deve ser inteiro positivo.

Sintaxe:

```
decl_const := CONST def_const ";" [ def_const ";" ]*
def_const := id_const "=" valor
```

3.4.1.3. Declaração de Famílias

Semântica:

Esta declaração atribui a um identificador uma faixa limitada. O escopo do identificador é global à especificação em que foi declarado. Sempre que aparecer o identificador de família, este terá por valor inicial o limite inferior da faixa e será incrementado de uma unidade na ocorrência de um ";". Obviamente que o limite superior permite que se interrompa o processo de incrementar indefinidamente o valor do identificador.

Sintaxe:

```
decl_fam := FAMILY decl_faixa ";" [ decl_faixa ";" ]*
```

```
decl_faixa := id_fam ":" faixa
```

```
faixa := "[" limite "." "." limite "]"
```

```
limite := inteiro_positivo | id_const
```

3.4.1.4. Declaração de Portos

Semântica:

O objetivo desta declaração é definir um interface para o módulo grupo (item 3.4.1.8). Esta declaração só terá sentido em uma especificação de grupo sendo que seu uso em outro tipo de especificação não é permitido. Na especificação de grupo os portos criados terão a denominação de portos de saída de grupos e portos de entrada de grupo, porém este porto quando visto externamente será tratado como um porto comum de instâncias. A sintaxe da declaração segue a definida na linguagem LINCE, sendo

que os tipos de comunicação possíveis sobre estes portos foram definidos em (Silva, 88).

Sintaxe:

```

decl_porto := PORT [esp_porto]+
esp_porto := lista_porto ":" ( IN | OUT )
           "(" tipo_msg [REPLY tipo_msg] ")" [ext_porto] ";"
lista_porto := ident [ fam ] [ "(" prior ")" ]
           [ "," ident [ fam ] [ "(" prior ")" ] ]*
fam := ( "[" const "." const "]" | [ id_fam ] )
const := ( inteiro_positivo | ident )
prior := inteiro_positivo
ext_porto := "["int_positivo"]" | "("SB")" | "("SS")"
tipo_msg := ( real | integer | boolean |
           char | signaltype | ident )

```

3.4.1.5. Declaração de Contexto

Semântica:

Os tipos módulos que constituirão o sistema deverão ser incluídos em uma especificação definindo o contexto do sistema. A declaração identifica os tipos módulos para a criação de instâncias. A mesma declaração permite a inclusão de tipos externos que poderão ser usados na declaração de portos. Neste caso deve ser especificado a unidade de definição do tipo e o identificador do tipo para a verificação da compatibilidade entre os portos interconectados.

Sintaxe:

```

decl_contexto := USE contexto ";" [contexto ";"]*
contexto := ( ctexto_tipo_modulo | ctexto_tipo_dado )
ctexto_tipo_modulo := id_tipo_modulo
ctexto_tipo_dado := id_unidade_def ":" lista_dados ";"
lista_dados := ident_dado [ "," ident_dado ]*

```

3.4.1.6. Declaração de InstanciaçãoSemântica:

A declaração CREATE, a partir do contexto definido na especificação de configuração (tipos módulos), pode determinar o carregamento de tipos módulos e a criação de instância destes nas estações do sistema. Se o tipo módulo possui parâmetros formais, os parâmetros reais devem ser atribuídos durante a instanciação. Dependendo da situação, a parametrização pode ainda ser transferida para uma outra fase de instanciação (ver especificação de grupo); este caso deve ser caracterizado por uma parametrização real através de 'splics' na declaração de parâmetros.

Várias instâncias podem ser criadas a partir de um único tipo módulo; sendo que a linguagem fornece duas possibilidades de construção. A primeira define uma lista de instâncias com identificadores diferentes e a segunda define uma família de instâncias através de um identificador de instância e de uma variável de família previamente definida. Esta variável indexará a instância de acordo com a semântica definida na declaração FAMILY. Em ambos os casos se existir parâmetros formais no tipo todas instâncias serão criadas com os mesmos parâmetros reais.

Caso exista uma única instância de um tipo módulo o identificador de tipo pode ser usado como um identificador de instância. Caso se deseje criar instâncias em uma família de estações deve-se indexar as estações através de uma variável de família. Abaixo são apresentados exemplos de opções da declaração CREATE.

Sintaxe:

```

decl_instância := CREATE decl_insta
decl_insta := [ localiza ] lista_decl_insta
localiza := [ AT [ ":" ] ] STATION id_estação ":"
id_estação := "[" inteiro_positivo | id_fam "]"
lista_decl_insta := instância ";" [ instância ";" ]*
instância := [ lista_insta ] [ ":" ] def_tipo
def_tipo := id_tipo_mod [ lista_param ] [ atribui_indice ]
lista_insta := nome_instância [ "," nome_instância ]*
nome_instância := id_insta [ família ]
família := "[" id_fam "]"
lista_param := "(" parâmetros_reais ")"
parâmetros_reais := parâmetro [ "," parâmetro ]
parâmetro := valor | param_formal
valor := ( real | integer | boolean | strings )
param_formal := "'" id_param_formal "'"
atribui_indice := "(" inteiro_positivo ")"

```

Exemplo:

CREATE AT

STATION [0]:

instal [k]: tipol (2, 3, TRUE);

{Cria uma família de instância do tipo1 na estação 0}

tipo2; {Cria uma instância do tipo2 com o mesmo nome}

STATION [K]:

insta2 [k]: tipo3;

{Cria uma família de instâncias distribuídas na família de estações}.

3.4.1.7. Declaração de Conexões de Portos

Semântica:

A declaração LINK é a responsável pela conexão lógica de portos de instâncias de módulos e portos de grupo. A conexão lógica de instâncias de módulos é conseguida através da ligação dos portos de saída a portos de entrada. Neste caso o porto é identificado pelo nome da instância e pelo seu próprio identificador.

Os portos de grupo tem um tratamento especial e devem ser ligados a portos de instâncias declaradas na especificação de grupo, utilizando-se do mesmo comando LINK. Para tanto, esta declaração, na sua forma geral, é apresentada como responsável pela conexão entre portos de "origem" e portos de "destino", onde:

- **porto de origem:** pode ser um porto de uma instância de módulo ou um porto de entrada de grupo.
- **porto de destino:** é um porto de entrada de uma instância de módulo ou um porto de saída de grupo.

Em uma especificação grupo a conexão de um porto de entrada de grupo a um porto de instância, declarada internamente, torna este último visível externamente. O mesmo raciocínio pode ser aplicado a portos de saída de instâncias que serão visíveis através de portos de saída de grupo. Obviamente que a compatibilidade de tipos deve ser preservada nestas ligações.

Sintaxe:

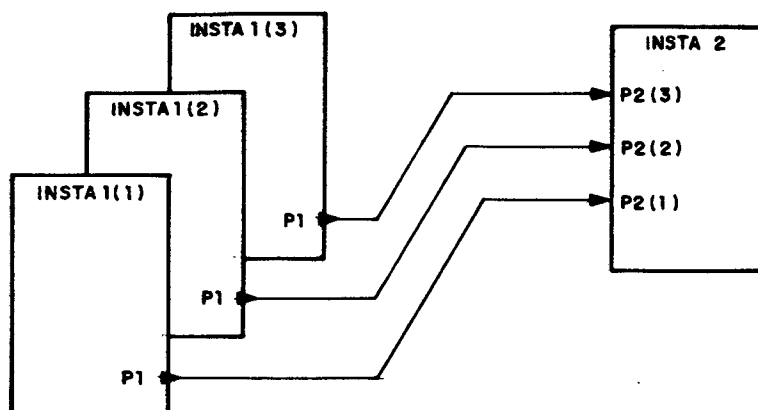
```

decl_link    := LINK lista_ligações
lista_ligações := ligação ";" [ ligação ";" ]*
ligação      := lista_portos_orig TO lista_portos_dest
lista_portos_orig := id_porto_orig [ "," id_porto_orig ]
id_porto_orig  := porto_entr_grupo | porto_sai_insta
porto_entr_grupo := nome_porto
nome_porto     := id_porto [ "[" id_fam "]" ]
porto_sai_insta := nome_instância "." nome_porto
lista_portos_dest := id_port_dest [ "," id_port_dest ]
id_porto_dest   := porto_sai_grupo | porto_entr_insta
porto_sai_grupo := nome_porto
porto_entr_insta := nome_instância "." nome_porto

```

O conceito de família pode ser aplicado de forma a facilitar a edição de comandos similares. Famílias de portos e/ou famílias de instâncias podem ser declaradas no comando LINK desde que as faixas sejam compatíveis. A seguir é apresentado um exemplo e representação gráfica correspondente.

Exemplo: LINK instal [k] . p1 TO insta2 . p2 [k];



A semântica neste caso produz conexões, uma a uma, caso existam famílias definidas nos portos de origem e destino.

3.4.1.8. Declarações de Especificação de Configuração

A seguir são descritas as especificações que identificam o tipo de configuração suportadas pela LINCS.

Especificação GROUP MODULE

A declaração GROUP MODULE permite a definição de sub-especificações de configuração (subsistemas), proporcionando a possibilidade de construções hierárquicas de sistemas. Esta facilidade de construção é obtida através de um encapsulamento de um conjunto de módulos, e da atribuição ao conjunto de uma interface de comunicação (portos). O conjunto definido por esta declaração é tratado de forma idêntica a um módulo definido pelo compilador LINCE, não existindo a nível de especificação de configuração, qualquer diferença entre os mesmos. Consequentemente um módulo grupo pode ser importado, instanciado e interconectado a outros módulos do sistema (inclusive módulo

grupo). Devido a esta semelhança não se fará diferença entre um tipo módulo grupo e um tipo módulo simples sendo que ambos serão referenciados apenas por tipo módulo. A estrutura e a sintaxe para a especificação são ilustradas a seguir:

```
decl_grupo := GROUP MODULE id_grupo [ param_formais ] ";"
           corpo_grupo
           END "."
```

```
corpo_grupo := [ decl_const ]* |
               [ decl_fam ]* |
               [ decl_ctexto ]+ |
               [ decl_porto ]+ |
               [ decl_instância ]+ |
               [ decl_link ]+
```

```
param_formais := "(" param_formal [ ";" param_formal ] ")"
param_formal := lista_ident ":" tipo_param
lista_ident := ident [ ";" ident ]
tipo_param := ( real | integer | boolean | char | ident )
```

Para proporcionar uma maior flexibilidade de reutilização do grupo, a parametrização real das instâncias declaradas dentro do grupo podem ser definidas através dos parâmetros formais listados no cabeçalho da declaração GROUP MODULE. A aplicabilidade deste recurso pode ser vista quando de sua utilização em um nível superior de abstração (ver Fig. 3.6), durante a instanciação deste grupo, onde a parametrização real de valores implica na atribuição destes às instâncias cujos parâmetros ficaram pendentes na especificação de grupo.

```

SYSTEM exemplo;
USE grupo 1; ...
  ⋮
CREATE AT STATION [1]:
  insta : grupo 1 ( 22 , 3.14);
  ⋮
END

```

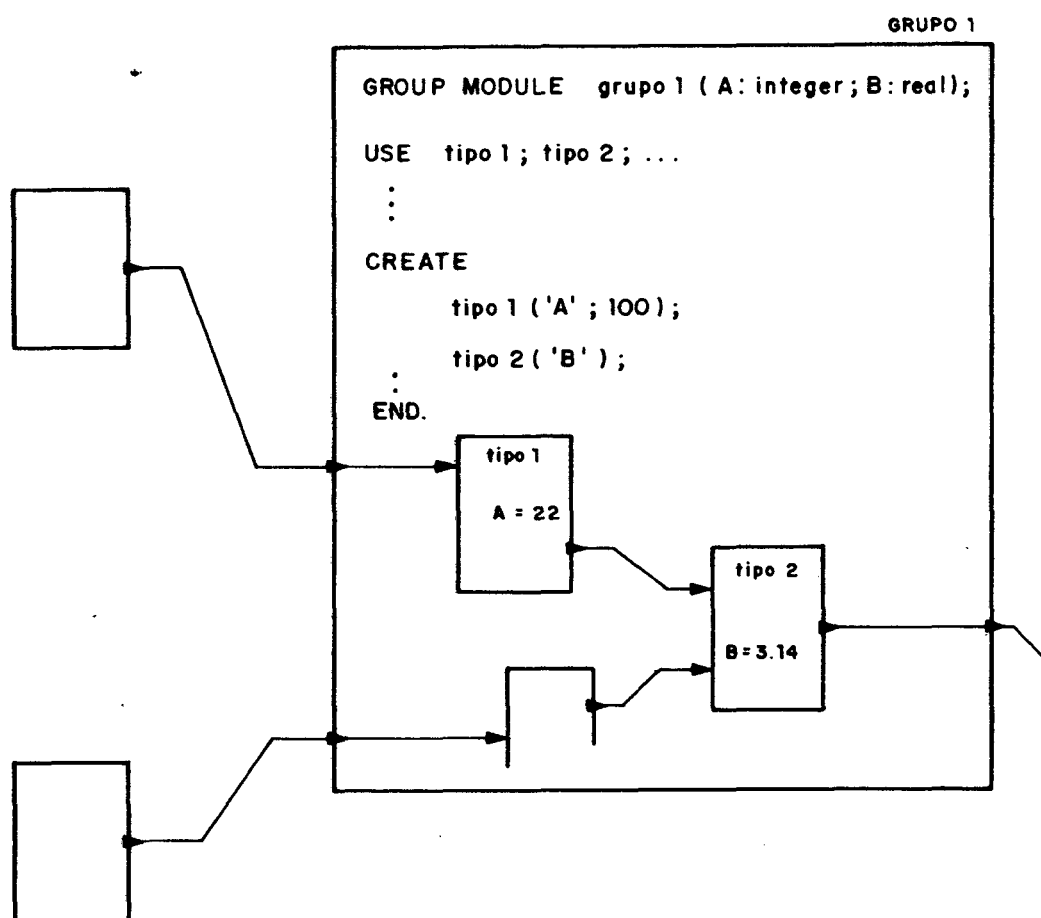


Fig. 3.6 - Ilustração do uso de parametrização em grupo

A declaração de contexto (USE) pode ser usada no sentido de incluir tipos externos a serem utilizados na declaração de portos. Caso os tipos de mensagens sejam do tipo padrão a inclusão de dados externos pode ser dispensada.

Especificação SYSTEM

A especificação SYSTEM define a configuração inicial do sistema utilizando-se dos comandos básicos da linguagem. A estrutura da especificação apresenta a seguinte sintaxe:

```
decl_sistema := SYSTEM id_sistema ";"
                corpo_esp
                END "."
```

```
corpo_esp := [ decl_const ]* |
              [ decl_fam ]* |
              [ decl_ctexto ]+ |
              [ decl_instância ]+ |
              [ decl_link ]+
```

Uma especificação completa deve identificar o contexto, criar as instâncias e interconectá-las. O resultado do processamento desta especificação é uma base de dados que será utilizada durante o processo de construção do sistema.

A declaração SYSTEM é a maior unidade encapsuladora na especificação de configuração de um sistema, não existindo portanto abstrações de maior nível; com isto são determinadas restrições em relação as declarações USE e CREATE. A primeira só é utilizada para a inclusão de tipos módulos; uma vez que não há sentido, neste nível, a inclusão de dados externos. A segunda é limitada à parametrização por valores; isto garante que todas as

pendências de parametrização deixadas na especificação de grupo (quando utilizadas) sejam resolvidas.

Especificação CHANGE

A especificação de mudanças define um conjunto de declarações com o objetivo de alterar a configuração do sistema em tempo de execução. Para tanto, nestas especificações são necessárias declarações de operações inversas ao LINK, ao CREATE e ao USE, ou seja, operações que desfaçam conexão, destruam instâncias e removam tipo módulo. Estas declarações permitirão ao usuário especificar as mudanças que deverão ser realizadas dinamicamente no sistema, controladas pelo configurador. Estão sendo realizados atualmente estudos no sentido de definir a semântica das ações para estas declarações. Porém, a estrutura e a sintaxe da especificação deverão seguir o padrão atual e podem ser descritas como:

```
decl_mud := CHANGE id_sistema ";"
          corpo_esp
          END "."
```

```
corpo_esp := [decl_const]* |
             [decl_fam]* |
             [decl_ctexto]* |
             [decl_instância]* |
             [decl_link]* |
             [decl_remove]* |
             [decl_destroi]* |
             [decl_unlink]*
```


Esta especificação deverá incorporar as alterações na base de dados corrente e gerar uma lista de ações que serão executadas em um processo de configuração dinâmica.

3.5. ASPECTOS DO DESENVOLVIMENTO DO SUPORTE PARA A CONFIGURAÇÃO ESTÁTICA DE SISTEMAS

O desenvolvimento das ferramentas e utilitários necessários para a configuração de um sistema, na fase de análise de requisitos e especificação, foi auxiliado pela Técnica de Projeto e Análise Estruturada - SADT (Ross, 77a) (Ross, 77b). Esta metodologia permitiu uma especificação inicial que ponto de vista funcional correspondeu fortemente com o nível de abstração necessário para visualizar e definir o problema. A seguir são apresentados alguns aspectos do processo de desenvolvimento.

3.5.1. Análise de Contexto

A construção de um sistema distribuído está ligada com a necessidade de ferramentas e utilitários que processem as especificações de configuração e carreguem os componentes modulares nas estações. Uma destas ferramentas é o processador LINCS. Este surgiu como uma necessidade de apoiar o princípio de decomposição modular, substituindo de uma forma mais eficiente os tradicionais editores de ligações. O processador LINCS é parte de um processo de configuração e juntamente com o carregador estático forma o configurador de sistemas.

3.5.2. Especificação Funcional

Neste item são descritas as principais atividades envolvidas com o processo de configuração de sistemas. A formalização desta especificação se encontra no apêndice B, através dos diagramas SADT.

3.5.2.1. O Processador LINCS

O processador LINCS é um programa que realiza as funções de tradução da especificação de configuração e ativa o processo de configuração na ausência de erro. A entrada do processador é um arquivo contendo a especificação de configuração, escrita em LINCS, que deverá ser analisada quanto a integridade, compatibilidade e viabilidade de realização.

A função de tradução é decomposta em funções de análise sintática, validação da especificação e geração de estrutura de dados. A primeira consiste em analisar as partes sintática e léxica da especificação de configuração com respeito a sintaxe da Linguagem de Configuração, sinalizando e indicando a ocorrência de erros. A presença de erros sintáticos deverá inibir a geração de dados.

A validação da especificação deve passar pela verificação a consistência de dados, da disponibilidade de recursos físicos nas estações e da compatibilidade dos portos de módulos indicados nas declarações de ligação. Para a validação da especificação é necessário que estejam disponíveis, nos sistema de arquivos da Estação de Trabalho, todos os arquivos de dados gerados pelo compilador LINCE para os tipos módulos envolvidos

na configuração. A validação da especificação corresponde então a uma análise de contexto da configuração.

A geração de uma estrutura de dados pelo processador LINCS é a função básica para a criação da base de dados com informações sobre o sistema; à medida que o processamento das especificações vai se realizando, as estruturas vão sendo criadas na memória e posteriormente, deverão ser gravadas em disco dependendo do sucesso ou não da implantação da configuração especificada.

3.5.2.2. Processo de Configuração Estática

De acordo com o modelo de configuração apresentado, a configuração pode ser realizada estática e dinamicamente. Muito embora alguns aspectos relacionados com mudanças tenham sido incluídas na linguagem LINCS, o objetivo deste trabalho é o processo de configuração estática. Neste sentido, as utilidades e ferramentas desenvolvidas devem atender às necessidades da configuração inicial do sistema. Para tanto, neste processo de configuração, é necessário selecionar os arquivos a serem carregados em cada estação, formando as chamadas imagens de carga. Uma vez carregadas e inicializadas todas as estações, a estrutura de dados gerada pelo tradutor pode ser gravada em disco tornando-se a base de dados efetiva do sistema.

A construção do sistema está portanto, relacionado com o carregamento e inicialização de todas as Estações de Execução. A configuração das estações é realizada através do carregamento das imagens carga através da rede ("down-line loading"); exigindo para tanto, a definição de mecanismos capazes de

suportar a transferência de arquivos e controlar a inicialização destas estações.

A configuração do sistema se dá segundo um protocolo a duas fases (Gray, 78):

- Na **primeira fase** as Estações de Execução recebem as respectivas imagens de cargas e começam a executar o programa de iniciação. Uma vez concluída a configuração da Estação de Execução este deve enviar uma mensagem indicando o sucesso ou não à Estação de Trabalho;
- Na **segunda fase**, uma vez de posse das indicações de sucesso na construção de todas as estações, a Estação de Trabalho envia em difusão uma mensagem de confirmação da configuração às Estações de Execução. Estas retornam mensagens de reconhecimento que uma vez recebidas fazem da estrutura de dados, a nova base de dados do sistema.

O insucesso em qualquer das fases, implica no abandono dos resultados intermediários e no reinício do processo de configuração.

A construção da configuração inicial de cada estação é realizada por meio de um programa de iniciação, que se executa com referências a uma Tabela de Configuração específica para cada estação. Estas tabelas serão apresentadas no próximo capítulo e são geradas automaticamente na tradução da especificação SYSTEM.

A execução do protocolo a duas fases na Estação de Trabalho está a cargo de uma entidade chamada Construtor que após a geração das Tabelas de Configuração é ativado, formando, por sua

vez, as imagens de carga e dando início aos processos de transferências destas para as estações correspondentes.

Programa de Iniciação

O programa de iniciação é responsável pela construção das estações remotas e deve informar ao Construtor na Estação de Trabalho o sucesso ou não de suas atividades.

A construção das estações consiste em iniciar as estruturas de dados do Núcleo de Tempo Real, criar as instâncias dos tipos módulos carregados e interconectá-las. Após a execução destas funções uma mensagem de controle é enviada a Estação de Trabalho sinalizando o estado das operações. Neste ponto o programa de iniciação é bloqueado, aguardando uma confirmação da configuração do sistema. Dependendo desta confirmação o processo de construção local poderá ser interrompido ou seguirá a sequência normal: enviando o reconhecimento à mensagem de confirmação, ativando as instâncias da estação e por fim passando o controle ao escalonador, finalizando então o processo de configuração da estação local.

Imagens de Carga

A imagem de carga em cada estação consiste de tipos módulos do SOD (Sistema Operacional Distribuído), da aplicação e mais os códigos do NTR, todos com os seus itens realocáveis devidamente resolvidos. No caso particular da configuração estática é adicionado à imagem a Tabela de Configuração.

Programa "Boot" nas Estações de Execução

Para permitir o carregamento nas Estações de Execução é necessário a existência de um programa "Boot". As funções principais deste programa estão relacionadas com o controle de carregamento dos arquivos de tipos na memória e com a ativação do programa de iniciação local.

A Fig. 3.7 ilustra o procedimento de configuração estática onde pode ser visto a interação entre o Tradutor e o programa Construtor.

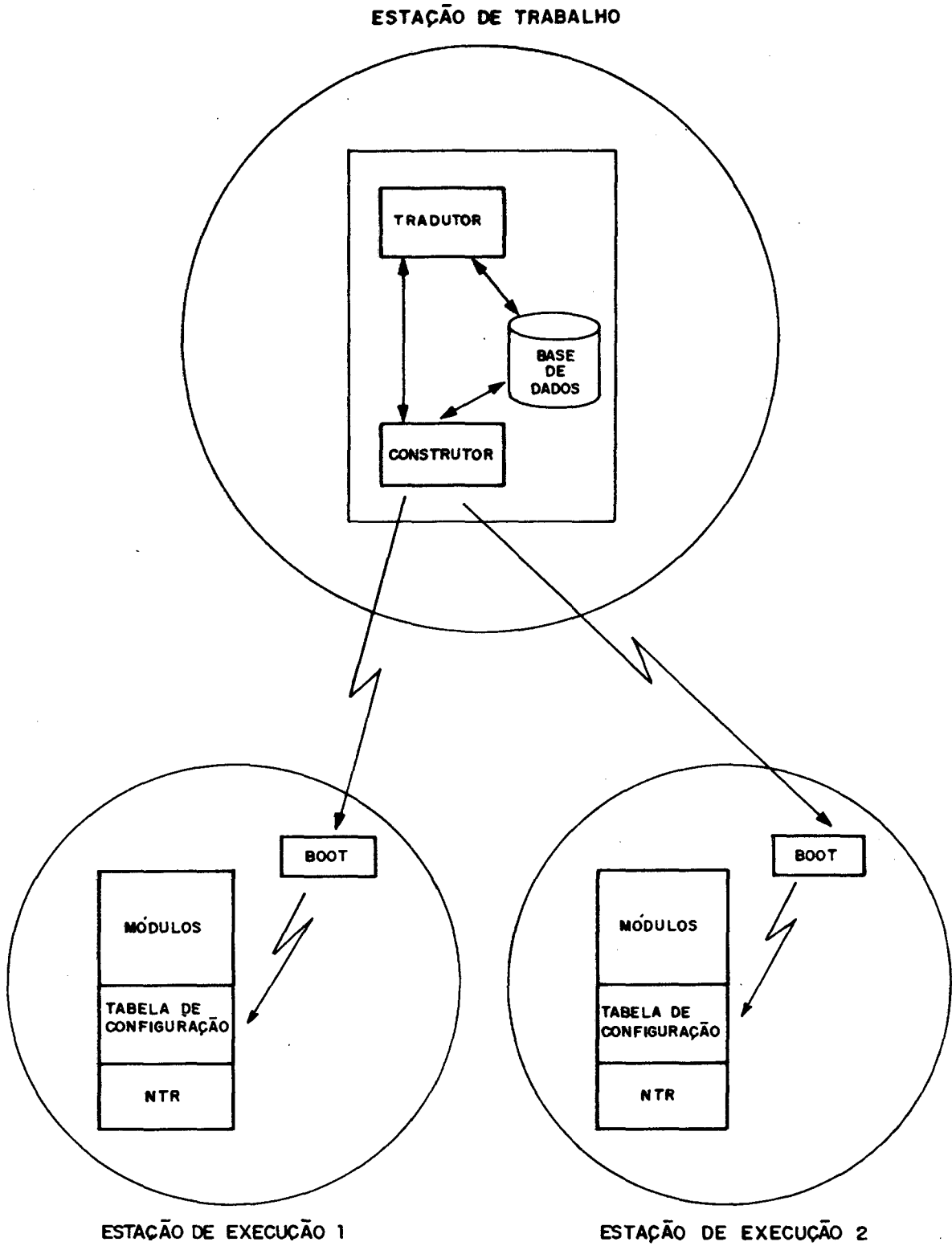


Fig. 3.7 - Configuração Estática

3.6. CONCLUSÃO

Neste capítulo foi apresentado a Linguagem de Configuração de Sistemas - LINCS e um conjunto de ferramentas para atender os aspectos de configuração estática de aplicações distribuídas.

A linguagem LINCS permite declarar a configuração inicial resultando em um carregamento estático dos módulos do SOD e dos módulos de aplicação. O suporte para a configuração estática descrita neste capítulo deve fornecer subsídios para a construção do Sistema Operacional Distribuído que fornecerá então o suporte para alteração na estrutura do software em tempo de execução (configuração dinâmica).

O uso da ferramenta SADT para a representação do problema no início do desenvolvimento foi de grande auxílio. O nível de abstração atingido foi considerado satisfatório, dispensando um maior refinamento para o desenvolvimento total do modelo.

CAPÍTULO 4

ASPECTOS DE IMPLEMENTAÇÃO DO SUPORTE PARA A CONFIGURAÇÃO ESTÁTICA E RESULTADOS

4.1. INTRODUÇÃO

Neste capítulo são apresentados os aspectos mais relevantes da implementação do suporte para a configuração estática. Este suporte foi dividido em duas partes que interagem através de uma estrutura de dados comum :

- A primeira é a responsável pelo processamento da LINCS, gerando a estrutura de dados correspondente a uma especificação de configuração. A implementação do processador desta linguagem foi realizada pelo uso de ferramentas especiais as quais são citadas na sua importância, neste capítulo.
- A segunda parte envolve utilitários e mecanismos envolvidos com a construção do sistema. Utilizando-se de informações geradas na tradução, estes utilitários e mecanismos conduzem a construção das estações permitindo a configuração do sistema.

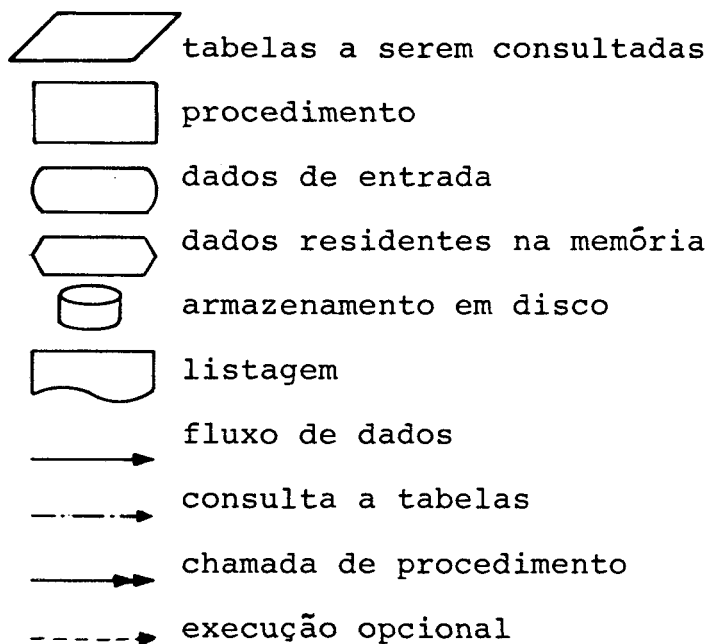
Também é objeto deste capítulo apresentar os resultados obtidos a partir da aplicação da LIS (Linguagem de Implementação de Sistemas) na simulação de uma célula flexível de manufatura.

4.2. PROCESSADOR LINCS

A implementação do processador LINCS consiste em escrever um programa capaz de ler as especificações de configuração, processá-las e gerar uma estrutura de dados de acordo com o tipo de especificação. Neste item será apresentado a estrutura funcional detalhada do processador, para depois então serem levantados os aspectos relevantes do processamento de especificação de configuração.

4.2.1. Estrutura do Processador

A estrutura do processador pode ser vista pelo diagrama da Fig. 4.1. Neste diagrama as seguintes convenções são utilizadas:



O processador LINCS apresenta algumas características típicas de compiladores. Estas se devem ao fato do processador ser dirigido para uma Linguagem de Configuração. Considerando que o processamento se dará em função de uma linguagem é necessário que esteja presente, na estrutura do processador, um elemento capaz de construir uma árvore sintática a partir das

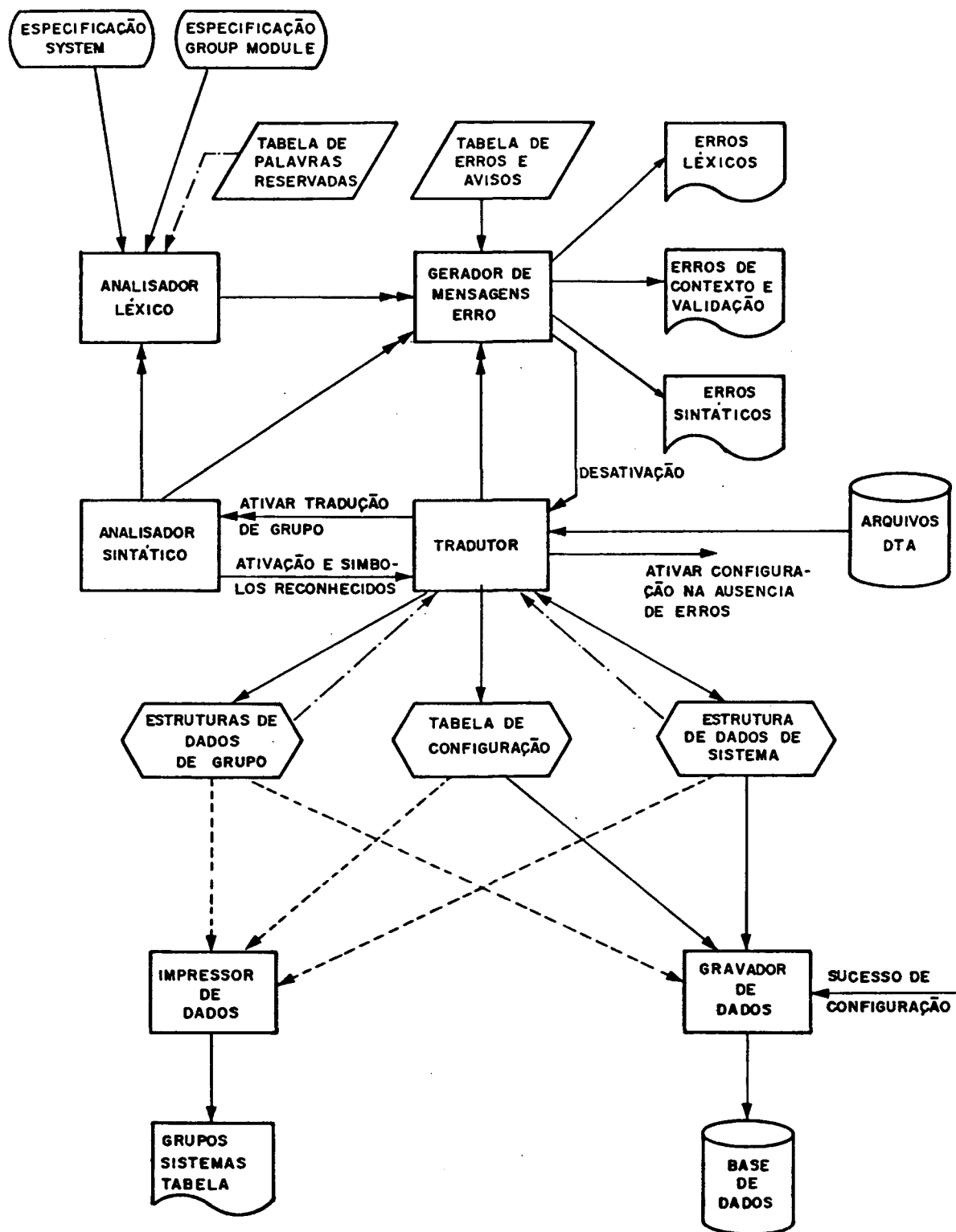


Fig. 4.1 - Estrutura Funcional do Processador LINC

regras de sintaxe da linguagem. Este elemento é o analisador sintático.

O analisador sintático opera com uma sequência de unidades sintáticas, agrupando-as e formando a árvore sintática. Cada nó da árvore representa uma unidade sintática; por exemplo, CREATE, SYSTEM, LINK, ";", etc... são unidades sintáticas. Os exemplos citados são unidades sintáticas elementares da LINCS que estão no primeiro nível da árvore, à estas unidades dá-se o nome de itens léxicos. O reconhecimento destes itens é realizado pelo analisador léxico, que se constitui no segundo elemento da estrutura. Um item léxico é então reconhecido e classificado pelo analisador léxico sempre que este for solicitado pelo analisador sintático. A ação combinada destes dois elementos permite implementar o processamento da LINCS.

Com base nestas considerações iniciais, pode-se descrever as unidades funcionais do processador LINCS que se encontra dividido em:

- **Analisador Léxico:** É o responsável pela leitura dos arquivos fontes, contendo a especificação de configuração, agrupando os caracteres em itens léxicos, definidos na LINCS, e determinando sua classe (palavra reservada, número, etc..). Por exemplo, os caracteres 'L', 'I', 'N' e 'K' são agrupados e reconhecidos como uma palavra reservada da LINCS. O analisador léxico é chamado pelo analisador sintático e retorna um valor indicando o item léxico de entrada.

- **Analisador Sintático:** Analisa se uma sequência de cadeia de caracteres (unidades sintáticas) corresponde a uma

regra gramatical previamente definida na gramática da Linguagem (sintaxe da LINCS). O sucesso da operação permite ativar o tradutor para executar uma determinada ação correspondente à sequência de entrada.

- **Tradutor:** É constituído por um conjunto de ações que caracterizam a semântica de cada instrução LINCS. Realiza a análise de contexto e validação, executando as funções de verificação de tipos, compatibilidade de conexões e disponibilidade de recursos nas estações.
- **Gerador de Mensagens de Erros:** É o responsável pela formatação e impressão das mensagens de erro durante o processamento.
- **Gravador de Dados:** É o responsável pela gravação da estrutura de dados gerada no processamento da especificação SYSTEM bem como da Tabela de Configuração. Opcionalmente pode ser utilizado para gravar as estruturas de dados resultante de um processamento de grupo.

Os analisadores sintático e léxico foram implementados por ferramentas geradoras de programas conhecidas como YACC e LEX, respectivamente; estas ferramentas serão apresentadas no item 4.4. A estrutura de funcionamento destes analisadores pode ser representada por máquinas abstratas, chamadas de Autômatos de Estados Finitos, que são capazes de reconhecer cadeias de caracteres. A estrutura do analisador léxico é mais simples, uma vez que este deve apenas reconhecer unidades sintáticas elementares. A estrutura do analisador sintático é um pouco mais complexa, necessitando de pilhas para armazenamento de valores

(retornado pelo analisador l xico) e estados; isto se deve   composi o de unidades sint ticas para que a  rvore sint tica possa ser constru da.

Estes analisadores tamb m s o respons veis pela tarefa de reconhecimento de erros sint ticos e l xicos durante o processamento. A ocorr ncia deste erros   sinalizada ao Gerador de Mensagens de Erros (atrav s de uma chamada de procedimento) que pesquisa na sua tabela a mensagem correspondente e produz a sa da ao usu rio. Uma vez detetado um erro, o processamento restante se resume simplesmente em novas an lise de erros inibindo a gera o de dados e por consequencia, a configura o do sistema.

O Tradutor   um programa dirigido por sintaxe. Este deve recuperar os valores armazenados na pilha do analisador sint tico e criar estruturas na mem ria de acordo com a instru o LINCS reconhecida (ver Fig. 4.1). Quando o analisador sint tico reconhece uma gram tica da linguagem, o Tradutor   ativado para executar uma determinada a o. Esta a o consiste basicamente em recuperar s mbolos armazenados na pilha de valores, consultar informa oes em estruturas j  formadas, criar novas estruturas e finalmente ativar o processo de configura o do sistema no final da tradu o da especifica o SYSTEM. O conjunto de a oes executadas pelo Tradutor podem ser resumidos nas seguintes fun oes:

- **Tradu o propriamente dita:** Consiste em transladar o c digo fonte para uma estrutura de dados, atribuindo   identificadores l gicos de inst ncias, tipos m dulos e esta oes atributos de localiza o ( ndice);

- **Validação de Contexto:** Verificar a existência dos arquivos correspondentes a tipos módulos declarados no contexto (USE);
- **Validação de Parâmetros:** Validar a parametrização real em relação aos parâmetros formais declarados nos tipos módulos durante a instanciação;
- **Validação de Compatibilidade de Tipos:** Os portos identificados na declaração LINK estão sujeitos a uma série de validações. Estas consistem em : verificar existência das instâncias correspondentes, verificar existência dos portos, e principalmente verificar a compatibilidade de tipos dos portos a serem conectados, etc;
- **Disponibilidade de Recursos:** Após a definição de todo o contexto da especificação, as estruturas formadas são organizadas em direção a uma forma final para gravação. Neste processo é realizado a validação de disponibilidade de recursos nas Estações de Execução. Esta função consiste basicamente em verificar a memória disponível em relação a necessária para carregar os tipos módulos e criar as instâncias;
- **Gerar a Tabela de Configuração:** À partir da estrutura final de dados, as informações de cada estação são acessadas e organizadas em forma de tabelas para a configuração das estações (ver item 4.2.4).

Após a operação de tradução, o processo de construção do sistema é ativado que em caso de sucesso habilita o Gravador de

Dados a registrar a estrutura correspondente como a base de dados representando o sistema. Opcionalmente podem ser solicitados impressões das estruturas formadas através do Impressor de Dados.

4.2.2. Processamento de Especificação de Configuração

Este processamento consiste em traduzir a especificação em uma estrutura de dados e a seguir realizar uma validação correspondente a instrução fornecida. O tradutor ao executar o processamento pressupõe a existência de arquivos descritores correspondentes aos tipos módulos definidos para a configuração e ainda de um arquivo descritor das características físicas das estações.

Arquivo Descritor de Tipo Módulo:

Este arquivo (prefixo.dta) é criado quando da compilação LINCE e tem gravadas e organizadas as seguintes informações:

Dados inerentes ao código e a área de dados

- Tamanho da área de código;
- Tamanho da área de dados estáticos;
- Tamanho do stack;
- Tamanho da área de dados dinâmicos; e
- início do código de inicialização do módulo.

Informações de portos do módulos

- Nome do porto;
- Tipo de mensagem;
- Módulo de definição do tipo de mensagem;

- Tipo da mensagem de resposta;
- Módulo de definição da mensagem de resposta;
- Prioridade;
- Tipo do porto;
- Índice do porto;
- Número de buffers; e
- Início e fim de família.

Informações de parâmetros formais do módulo

- Nome do parâmetro;
- Tipo do parâmetro; e
- Endereço.

Arquivo Descritor da Configuração Física:

As informações de estações são obtidas de um arquivo (cnfísica.est) que apresenta a seguinte estrutura:

Dados físicos da estação

- Índice da estação;
- Endereço de início de carga;
- Área de memória disponível;
- Endereço da placa de comunicação; e
- Nível de interrupção utilizado pela placa.

O projetista de sistemas tem acesso a este arquivo através de facilidades que o permitem gerar uma configuração física e também introduzir modificações.

Durante a descrição do processamento das declarações LINCS serão apresentados, sempre que necessário, estruturas que

ilustrarão o texto. As estruturas que forem referenciadas sem a sua devida ilustração, estão apresentadas no apêndice A.

4.2.2.1. Processamento de Especificação SYSTEM

O processamento desta especificação resulta em uma estrutura de dados contendo informações globais do sistema. O reconhecimento desta especificação cria uma estrutura inicial, conforme ilustrado abaixo, que será atualizada no decorrer do processamento.

```

struct sistema
    { char          id_sistema [MAX_C];
      struct estação *pest;
      struct tipomodulo *ptipo;
      struct instância *pinstta;
      int          cont_ind_mod;
    };

```

O campo identificado por "pest" consiste basicamente de um ponteiro para uma lista de estruturas "estação" que são definidas a partir da especificação. Uma estrutura estação contém informações de tipos módulos simples, de instâncias, de conexões e de recursos disponíveis na estação correspondente. Todas as informações nesta estrutura são relativas a tipo módulo simples; referências a grupo em uma especificação são processadas e registradas segundo os tipos módulos simples que constituem o grupo.

Os campos seguintes (ptipo e pinstta) são apontadores para estruturas que permitem identificar o contexto do sistema e as instâncias criadas. O último campo é usado para controlar a

translação de identificador lógico dos tipos módulos para um identificador global com atributos de localização no sistema (endereço de sistema). A cada identificador lógico é associado um número inteiro (índice) que o identificará nas operações com o Sistema Operacional Distribuído (SOD).

4.2.2.2. Processamento de Especificação GROUP MODULE

O processamento de especificação grupo tem por objetivo gerar uma estrutura de dados que pode ser utilizada posteriormente ou por uma especificação de sistema ou ainda por outra especificação grupo. Esta estrutura será posteriormente incorporada à estrutura gerada pela especificação SYSTEM. O mecanismo de processamento de grupo define um conjunto de registros e campos de dados cujo tamanho pode crescer arbitrariamente resultando em uma estrutura de complexidade arbitrária. O aninhamento e a reutilização de grupo podem criar vários caminhos e as necessidades de memória podem se tornar consideráveis.

A reutilização de uma especificação grupo não exige um novo processamento, mas sim a criação de uma estrutura (instagrupo) com as informações de interface do grupo (portos) atualizadas. Este tratamento implica na perda de algumas informações da estrutura interna do grupo que do ponto de vista de construção de sistema são irrelevantes, porém isto proporciona uma otimização na memória e no tempo.

A estrutura resultante do processamento do grupo apresenta a seguinte forma:

```

struct grupo
{
    char                id_grupo [Max_C];
    struct tipoform     *ptifo;
    struct tipodado     *ptida;
    struct tipomodulo   *ptimo;
    struct portgrupo    *pporten,
                        *pporsai;
    struct instância    *pisnta;
    struct instagrupo  *pisntagr;
    struct link         *plink;
    struct const        *pconst;
    struct fam          *pfam;
    struct grupo        *prox;
} ;

```

Os campos deste registro são ponteiros para outras estruturas que descrevem toda a especificação grupo. Estas estruturas estão ilustradas no apêndice A.

As especificações SYSTEM e GROUP MODULE são elementos encapsuladores na linguagem LINCS. A ocorrência destas define algumas alterações no processamento das demais declarações. A seguir são apresentados os processamentos das declarações restantes que a menos de observações explícitas se referem a ambas especificações.

4.2.2.3. Processamento da Declaração de Constantes (CONST)

Este processamento consiste em criar uma estrutura da forma:

```
struct constante
```

```
    { char          id_const [Max_C];
      int           valor;
      struct constante *prox;
    } ;
```

Os dois primeiros campos deste registro são atualizados de acordo com os símbolos recuperados da pilha do analisador sintático. A seguir esta deve ser introduzida em uma lista ligada, utilizando-se do terceiro campo. Desta maneira, são fornecidas duas funções responsáveis pela inserção e consulta à lista já existente. Não foi definida uma tabela de identificadores, e qualquer conflito em relação a igualdade com outros identificadores é solucionado pelo contexto da declaração.

4.2.2.4. Processamento da Declaração de Família (FAMILY)

As ações envolvidas com o processamento desta declaração criam uma estrutura da forma ilustrada abaixo e a inserem na lista ligada.

```
struct fam
```

```
    { char          id_fam [Max_C];
      int           min,
                  max;
      struct fam    *pfam;
    } ;
```

Adicionalmente é implementada uma função que consulta a lista de famílias e verifica a compatibilidade nos limites da faixa.

4.2.2.5. Processamento da Declaração de Portos (PORT)

Esta declaração é restrita a especificação de grupo e o seu processamento consiste em criar uma estrutura "portgrupo" referente ao porto e atualizá-la de acordo com os dados da declaração. A estrutura é inserida na lista correspondente e posteriormente será utilizada na ligação de portos. Esta estrutura apresenta a seguinte forma:

```

struct portgrupo
    { char          id_porto [Max_C];
      int          tipo,
                pmax,
                pmin,
                indice,
                buf_prior;
      struct msg   req,
                reply,
      struct portmodulo *pmod;
      struct portgrupo *prox;
    };

```

4.2.2.6 Processamento da Declaração de Contexto (USE)

A função desta declaração é a inclusão de tipos módulos e tipos dados. A inclusão de tipos módulos é comum em ambas especificações (GROUP e SYSTEM) e a de tipos dados é exclusiva para a especificação de grupo. A ocorrência desta última em uma especificação SYSTEM é ignorada e um aviso é gerado pelo processador.

As operações sobre tipos módulos consistem em identificá-los, se são módulos grupo ou módulos simples, e verificar a existência dos arquivos correspondentes no sistema de arquivos da Estação de Trabalho. Caso seja um tipo módulo simples é atribuído ao mesmo um índice, identificando-o no contexto do sistema. A identificação de um módulo grupo implica no seu processamento, exigindo portanto um redirecionamento para o arquivo fonte correspondente. Uma vez concluído o processamento do arquivo da especificação grupo, o retorno à declaração de contexto causadora do desvio é provocado.

As operações sobre tipos dados consiste em criar uma estrutura que permita identificar, posteriormente, o tipo e a unidade de definição correspondente.

4.2.2.7 Processamento da Declaração de Instanciação (CREATE)

O processamento desta declaração cria uma estrutura "instância" e a insere na estrutura "estação" correspondente.

Além da função de tradução, é realizada uma validação nos parâmetros reais que devem ser fornecidos caso o tipo módulo contenha parâmetros formais. Os aspectos de distribuição também são resolvidos identificando a estação e atualizando a estrutura "estação" correspondente. A estrutura instância é definida da seguinte forma:

```

struct instância
    { char                id_insta [Max_C];
                                id_tipo  [Max_C];
      int                ind_tip_mod,
                                imin,
                                imax,
                                base;
      struct parâmetros  *pparam;
      struct instância  *prox;
    } ;

```

A ocorrência desta declaração dentro da especificação SYSTEM deve ser completa no sentido de identificar a estação de carga. Numa especificação grupo não é permitido na declaração CREATE a atribuição de tipos módulos à estações; é assumida por "default", para todos os tipos módulos definidos estação em que o grupo for instanciado.

A validação dos parâmetros é realizada através da leitura e comparação com os dados obtidos do arquivo descritor do tipo módulo. Na especificação grupo existe uma parametrização especial realizada através de um parâmetro formal declarado no cabeçário do grupo. Nesta parametrização ocorre uma validação de tipos, considerando as informações dadas no cabeçário.

A todo identificador lógico de instâncias é atribuído um índice que é único a nível de estação e que a identifica nas operações de Sistema Operacional Distribuído. A nível de sistema, a instância é identificada por uma hierarquia de campos com atributos de localização (concatenação dos índice da estação

e da instância). Este número é chamado de endereço de sistema da instância.

4.2.2.8. Processamento da Declaração de Conexão de Portos (LINK)

A implementação das ações semânticas desta declaração envolve uma série de operações de validação nos tipos dos portos, nos tipos das mensagens e na compatibilidade entre famílias. Estas validações consistem em verificar:

- a existência da instância;
- a existência do porto;
- a compatibilidade dos tipos dos portos;
- a compatibilidade dos tipos de conexão (um-para-um, um-para-vários, vários-para-um) com os tipos dos portos;
- a compatibilidade dos tipos de mensagens (caso seja diferente do tipo padrão é feita uma verificação se as unidades de definição das mesmas são as mesmas); e
- a compatibilidade entre os limites da faixa de família, caso existam.

O sucesso das validações gera uma estrutura que representa a conexão de portos sendo posteriormente inserida na estrutura "estação" correspondente ao porto de saída. Esta estrutura pode ser descrita por:

```

struct link
{
    char          id_insta_ps  [Max_C],
                id_porto_sai [Max_C],
                id_insta_pe  [Max_C],
                id_porto_pe  [Max_C];

    int          ind_ins_mod_ps,
                ind_ps,
                ind_ins_mod_pe,
                ind_ps,
                ind_est_pe;

    struct link  *prox;
};

```

A partir dos campos desta estrutura pode ser formado o endereço de sistema de porto concatenando o índice da estação, o índice da instância e o índice do porto.

A implementação das ações é dependente do tipo de conexão a ser realizada. A tabela 4.1 resume as ações correspondentes as conexões possíveis.

Porto de Orig	Porto de Dest	Ação
Port_ent_grp	Port_sai_grp	não permitida
Port_ent_grp	Port_ent_insta	atualizar Port_ent_grp
Port_sai_insta	Port_sai_grp	atualizar Port_sai_grp
Port_sai_insta	Port_ent_insta	gerar dados referente a conexão e inserção na estação onde a instância do porto de saída foi criada.

Tabela 4.1 - Tipos de conexões possíveis

Port_ent_grp e Port_sai_grp se referem a portos de grupos definidos pela declaração PORT dentro de uma especificação grupo.

Considerações Gerais

Ao término do processamento de uma especificação SYSTEM ocorre uma validação com respeito a disponibilidade de memória nas estações de execução. Somente na ausência de erros de sintaxe e/ou erros de contexto, o processador ativa o processo de carga e construção das estações. O índice da Estação de Trabalho é assumido por "default" igual 0, porém este pode ser alterado pelo usuário através da linha de comando no início do processamento. Após a configuração das Estações de Execução a Estação de Trabalho é automaticamente configurada se o programador assim desejar.

4.2.3. Estrutura da Base de Dados

Durante o processamento são criadas na memória principal, estruturas referentes a estações, tipos, instâncias, etc., que convenientemente organizadas descrevem a estrutura global do sistema. Esta estrutura é gravada, de forma a poder ser recuperada quando necessário, definindo a base de dados representativa do sistema. As informações mais relevantes da base de dados podem ser vistas na Fig. 4.2.

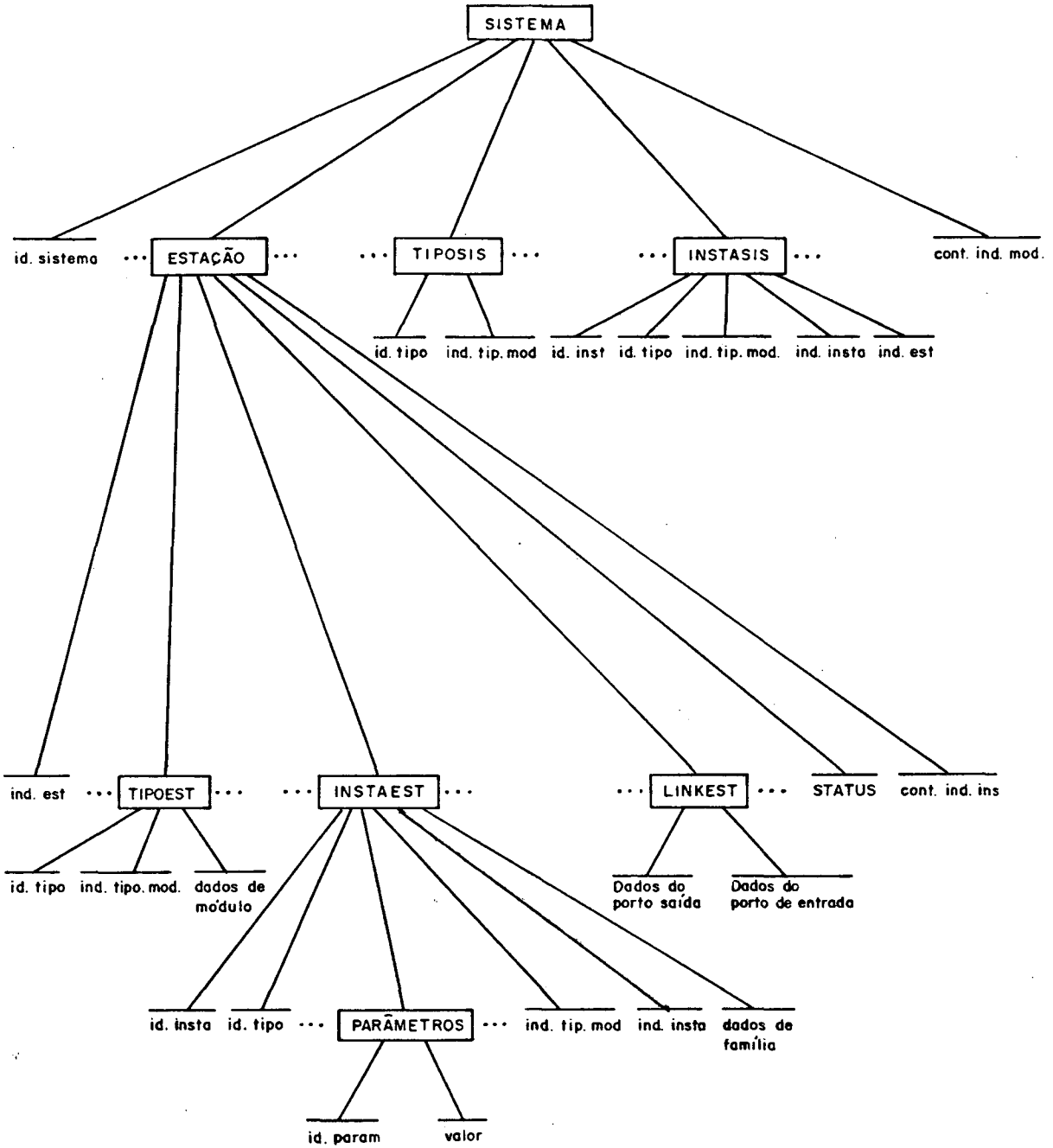


Fig. 4.2 - Estrutura da Base de Dados

Gravação e Leitura da Base de Dados

Devido a forma de implementação da estrutura na memória (lista ligada) é necessário definir e implementar funções que grave a lista e a recupere, posteriormente, para atualização. Para implementar a gravação da base de dados, devido às várias ramificações que a mesma pode ter, foi utilizado um algoritmo baseado no princípio de desenvolvimento da árvore em largura ("breadth first") que organizou e facilitou a procura das estruturas a serem gravadas.

4.2.4. Geração da Tabela de Configuração

O sucesso da tradução e validação da especificação de sistema permite a geração de uma Tabela de Configuração específica para cada estação do sistema. Estas tabelas serão utilizadas pelo programa de iniciação na construção destas estações. A tabela contém informações sobre (Ver Fig. 4.3):

- dados referentes a organização da memória de trabalho do núcleo;
- a criação de blocos de controle de tipos módulos;
- criação de instâncias; e
- dados para as conexões de portos.

Programa de Iniciação

O programa de iniciação implementado se destina a iniciar as Estações de Execução e a Estação de Trabalho. A configuração da Estação de Trabalho difere da Estação de Execução com respeito a organização de memória e a própria construção. Assim

Índice da estação
nível de interrupção
tam_mem_ocupada
num_tipo_modulos
Prim_descr_tipo
Seg_descr_tipo
:
Enes_descr_tipo
num_instância
Prim_descr_insta
Seg_descr_insta
:
Enes_descr_insta
num_conexões
Prim_descr_conex
Seg_descr_conex
:
Enes_descr_conex

Fig. 4.3 - Estrutura da Tabela de Configuração

sendo, o programa deve identificar a estação e executar as funções correspondentes.

A configuração da Estação de Trabalho implica na convivência do Sistema Operacional Hospedeiro - SOH com a imagem de carga desta e portanto a área de trabalho do NTR deve ser conhecida do SOH e vice-versa. A organização de memória para o núcleo consiste em iniciar as áreas correspondentes ao gerenciamento de memória (item 3.2.4). Após organizada a

memória, os tipos são carregados, as instâncias são criadas, os portos são conectados, os servidores de rede iniciados e finalmente o controle é passado ao escalonador para a execução das tarefas.

A construção da Estação de Execução pode ser organizada de forma mais simples uma vez que não existe SOH e o NTR é o único elemento gerenciador de memória. Após o carregamento de todos os módulos, o programa de configuração é ativado e se encarrega de realizar as seguintes operações:

- iniciar as estruturas internas do núcleo;
- criar os blocos de controle dos tipos módulos e registrar os endereços de carga;
- criar as instâncias dos tipos módulos;
- estabelecer as conexões de portos;
- iniciar os servidores de rede;

Após a conclusão destas operações, o programa de iniciação deve dar sequência ao protocolo à duas fases (item 3.5.2.2. do capítulo anterior), executado junto com o programa CONSTRUTOR na Estação de Trabalho, com o objetivo de garantir a consistência da configuração.

O programa de iniciação tem uma estrutura fixa e é compilado e "linkado" junto com o Núcleo de Tempo Real.

4.3. PROGRAMA CONSTRUTOR

O programa Construtor é implementado com o objetivo de carregar as imagens de cargas e controlar a execução da

configuração nas Estações de Execução. Os aspectos mais relevantes da construção estática são descritos a seguir.

4.3.1. Construção da Imagem de Carga

Os arquivos de códigos que serão carregados, devem ser preparados para a execução na estação alvo; isto consiste em obter do serviço de arquivos da Estação de Trabalho os arquivos de código dos tipos módulos, carregá-los na memória principal e resolver as grandezas realocáveis com base no endereço de início de carga em cada estação alvo. Estes endereços de carga estão disponíveis na representação da configuração física na base de dados. Para completar a imagem de carga é necessário adicionar aos tipos módulos executáveis a Tabela de Configuração e o Núcleo de Tempo Real.

4.3.2. Carregamento

O processo de carga é realizado através de um software da rede local utilizada (driver de rede). Este software encapsula os aspectos de hardware da comunicação. Assim sendo, a construção do carregador se resumiu na implementação de um protocolo de comunicação para garantir a integridade dos arquivos transferidos (controle de erro e controle de fluxo na comunicação).

Devido as limitações impostas pelo driver de rede, deve-se particionar a imagem de carga em blocos (512 bytes) e transmiti-los separadamente. Na transmissão sobre a rede é utilizado um mecanismo de "timeout" sendo que o recobrimento se dá por meio da retransmissão de bloco. Os blocos são constituídos por

cabeçalhos que permitem suas identificações. A estrutura básica deste cabeçalho pode ser vista na Fig. 4.4.

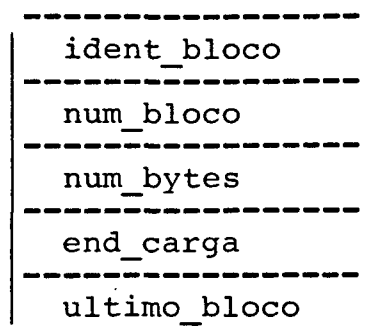


Fig. 4.4 - Estrutura do cabeçalho de bloco

Programa "Boot" nas Estações de Execução

O programa "Boot" recebe os blocos e os identifica para executar as ações necessárias. Basicamente este programa deve colocar os blocos recebidos na memória e ativar o programa de iniciação. No cabeçalho de bloco é obtido o endereço de início de carga. Para cada bloco recebido o "Boot" envia uma mensagem confirmando a recepção do bloco. Uma vez encerrada a transferência da imagem de carga, a área de memória ocupada pelo "Boot" é liberada e o contexto de execução passa a ser definido pelo programa de iniciação.

Driver de Rede

O driver de rede é um programa gerenciador de recursos da rede Cetus (Cetus, 87) que fornece um conjunto de primitivas básicas para a comunicação com a rede. O interface com o driver é realizado através de uma estrutura de dados, denominada bloco de controle de rede, onde o usuário passa informações de formatação e os endereços dos blocos de comunicação.

O driver limita o tamanho máximo do bloco em 576 bytes e é ativado por interrupção. O tipo de transmissão pode ser estabelecido pelo usuário através de campos apropriados no bloco de controle. As primitivas básicas utilizadas são o envio e recepção de pacotes, sendo que a comunicação é do tipo sem conexão ("datagrama").

4.4. FERRAMENTAS UTILIZADAS

No desenvolvimento do suporte para a configuração estática foram utilizadas as seguintes ferramentas:

Sistema YACC (Johnson, 78):

Esta ferramenta é destinada a geração de analisadores sintáticos guiado por especificações de alto nível. As especificações descrevem um conjunto de regras gramaticais que definem a sintaxe da linguagem a ser analisada (ou processada). O usuário especifica a estrutura da entrada (gramática da linguagem) e a ação que deve ser executada quando esta estrutura for reconhecida (ação semântica). Em adição à especificação de entrada, o usuário deve fornecer uma rotina (analisador léxico) capaz de reconhecer as unidades sintáticas elementares (item léxico). O programa gerado pela ferramenta se encarrega de chamar esta função sempre que necessário.

Sistema Lex (Lesk, 82):

O sistema Lex é uma ferramenta dirigida por uma especificação de alto nível com o objetivo de gerar um analisador léxico. A entrada para esta ferramenta é um arquivo

contendo uma tabela de expressões regulares e fragmentos de programas. Estas expressões regulares definem as unidades sintáticas elementares que o analisador gerado será capaz de reconhecer e classificar. Os fragmentos de programas, associado com a expressão regular, são as ações que o analisador léxico deve executar quando a expressão regular for satisfeita.

Estas ferramentas podem ser vistas como geradoras automáticas de sistemas, isto é, geram um programa a partir de sua especificação. O usuário deve definir a linguagem de entrada - estrutura léxica e sintática - e as ações para cada entrada.

Uma aplicação normalmente encontrada e, facilmente disponível nestas ferramentas, é a combinação do analisador gerado pelo YACC ("parser") com o gerado pelo LEX. Neste caso o parser chama automaticamente o analisador léxico para o reconhecimento de itens léxicos (palavras reservadas, operadores, números, etc..). Desta maneira, é possível implementar um tradutor dirigido por sintaxe, isto é, uma ação semântica é executada para cada sequência de itens léxicos de entrada.

Sistema Depurador:

Devido a implementação do suporte para a configuração estática ter sido realizada utilizando a linguagem C (Kerningham, 78), a depuração das funções envolvidas foram auxiliadas pela ferramenta Codeview. Esta ferramenta permite controlar a execução e a depuração do programa.

4.5. RESULTADOS

O estágio atual da implementação permite o processamento completo de uma especificação SYSTEM, utilizando-se de sub-especificações de sistema (especificação grupo), e a construção inicial do sistema (configuração estática). A integração do ambiente, a partir dos resultados obtidos de (Nacamura, 88), (Silva, 88) e do trabalho aqui descrito foi realizada e testada na forma de simulação de uma célula flexível de manufatura que se encontra no apêndice C. Com base neste exemplo pode ser constatado a eficiência do NTR e a facilidade de programação utilizando a linguagem LIS. A partir dos resultados obtidos durante o uso das ferramentas na implementação da célula pôde-se avaliar o desempenho das mesmas e inclusive sanar alguns erros que não foram detetados isoladamente. No que diz respeito a este trabalho pode-se destacar as seguintes considerações:

Considerações Gerais sobre o Processamento

O processador LINCS passou por depurações iniciais e ainda continua sob testes exaustivos visando alcançar um certo grau de confiabilidade satisfatório. O processamento de declarações simples, praticamente se apresenta livre de erros. O esforço de teste atual continua nas declarações que envolvem família e grupo, principalmente, na combinação de ambas. A complexidade da semântica e a variedade de opções, neste caso, implica numa grande diversidade de testes.

Considerações Gerais sobre a Configuração

A primeira implementação da célula flexível foi configurada a partir de uma versão inicial da configuração que não oferecia muita flexibilidade em face de mudanças nos dados do módulo, exigindo uma nova compilação e ligação do programa de configuração com o NTR. Esta desvantagem foi completamente sanada com o modelo de configuração descrito neste trabalho. Na ocorrência de uma nova compilação do módulo, implicando em mudança de dados, é gerada uma nova Tabela de Configuração porém sem alteração no programa de configuração, uma vez que o mesmo é fixo. Apesar da complexidade do processo de configuração, o desempenho em termos de tempo de reconfiguração foi sensivelmente melhorado.

Considerações Gerais sobre a Base de dados

A base de dados apresenta uma estrutura relativamente simples, apesar de conter todas as informações relevantes para a construção do sistema e para informações ao usuário. Todos os identificadores lógicos foram registrados possibilitando futuramente, a implementação de aplicativos que ilustrem graficamente a estrutura global do sistema.

O tamanho em número de bytes da base de dados, pode ser avaliado. Uma especificação SYSTEM definindo uma estação, duas instâncias e uma conexão resultou em uma base de dados de 460 bytes. A base de dados resultante da implementação da célula flexível tem um tamanho 1916 bytes. Embora estes números não sejam significativos, considerando a capacidade de memória dos microprocessadores atuais uma revisão na estrutura da base de

dados pode ser realizada afim de eliminar informações irrelevantes.

4.6. CONCLUSÃO

Os aspectos de implementação de um protótipo de configurador para sistemas distribuídos atendendo a configuração estática foram apresentados. O uso das ferramentas citadas permitiu uma implementação rápida, polarizando todos os esforços intelectuais na construção das funções do Tradutor e do Construtor, além de proporcionar uma flexibilidade maior caso se queira realizar mudanças na sintaxe e possivelmente na semântica da linguagem.

A fase bem sucedida de integração do ambiente incentiva a continuação do desenvolvimento partindo para a implementação dos módulos gerenciadores afim de pesquisar os problemas com a configuração dinâmica. A flexibilidade obtida com o modelo de configuração estática, permitindo uma nova reconfiguração de uma forma relativamente rápida, facilitará os trabalhos iniciais de implementação da configuração dinâmica.

CAPÍTULO 5

PERPECTIVAS FUTURAS E CONCLUSÃO FINAL

5.1. INTRODUÇÃO

O objetivo deste capítulo é traçar algumas considerações futuras para auxiliar a continuidade do trabalho aqui descrito. Estas considerações são frutos da experiência adquirida no projeto e implementação do configurador estático de sistemas. Obviamente, não são soluções definitivas mas sim propostas que irão contribuir para a melhora das características do sistema. Estas propostas vão no sentido de auxiliar a implementação inicial da estrutura da Estação de Trabalho e de mudanças nas características da linguagem LIS, que por vez exigirá mudanças inclusive no Núcleo de Tempo Real.

5.2. CONSIDERAÇÕES À RESPEITO DO MÓDULO GERENCIADOR DE CONFIGURAÇÃO

A seguir são apresentados alguns aspectos relacionados com a implementação do gerenciador de configuração que poderão ser úteis nos trabalhos iniciais de implantação da estrutura da Estação de Trabalho.

5.2.1. Aspectos relativos à Tradução e à Carga da imagem nas estações

Durante o processo de configuração estática as funções de tradução e carga das estações se realizam, de forma sequencial, no ambiente DOS. Assim sendo, é relativamente simples ativar este processo considerando que existe apenas este ambiente na Estação de Trabalho. Quando se considera a existência do ambiente ADES nesta estação e portanto, a existência de informações em tempo real, a organização e a funcionalidade desta estação se tornam menos simples.

Na configuração dinâmica, o processo de tradução e carga pode ser realizado separadamente sob o controle do gerenciador de configuração. Neste caso, é necessário implementar um módulo carregador e manter a função de tradução para ser executada no ambiente DOS, a qual será ativada via módulo interfaceador. A Fig. 5.1 ilustra este processo que é descrito a seguir.

O gerenciador recebe uma mensagem de reconfiguração do sistema (esta pode ser proveniente da aplicação (ver item 5.3.3) ou do operador), cujo conteúdo identifica o arquivo fonte contendo a especificação de mudança. O gerenciador ativa o tradutor, via mensagem (porto P1, Fig. 5.1), a realizar o processamento da especificação correspondente. Após a tradução, o resultado é informado ao gerenciador que em caso de sucesso solicita a Tabela de Ações (via porto P2). Esta tabela é interpretada pelo gerenciador que a executa através das entidades do SOD. No caso específico do carregamento, o módulo carregador é ativado (via porto P3). Este deve acessar o arquivo código, no sistema de arquivos, formar a imagem de carga e

ESTAÇÃO DE TRABALHO

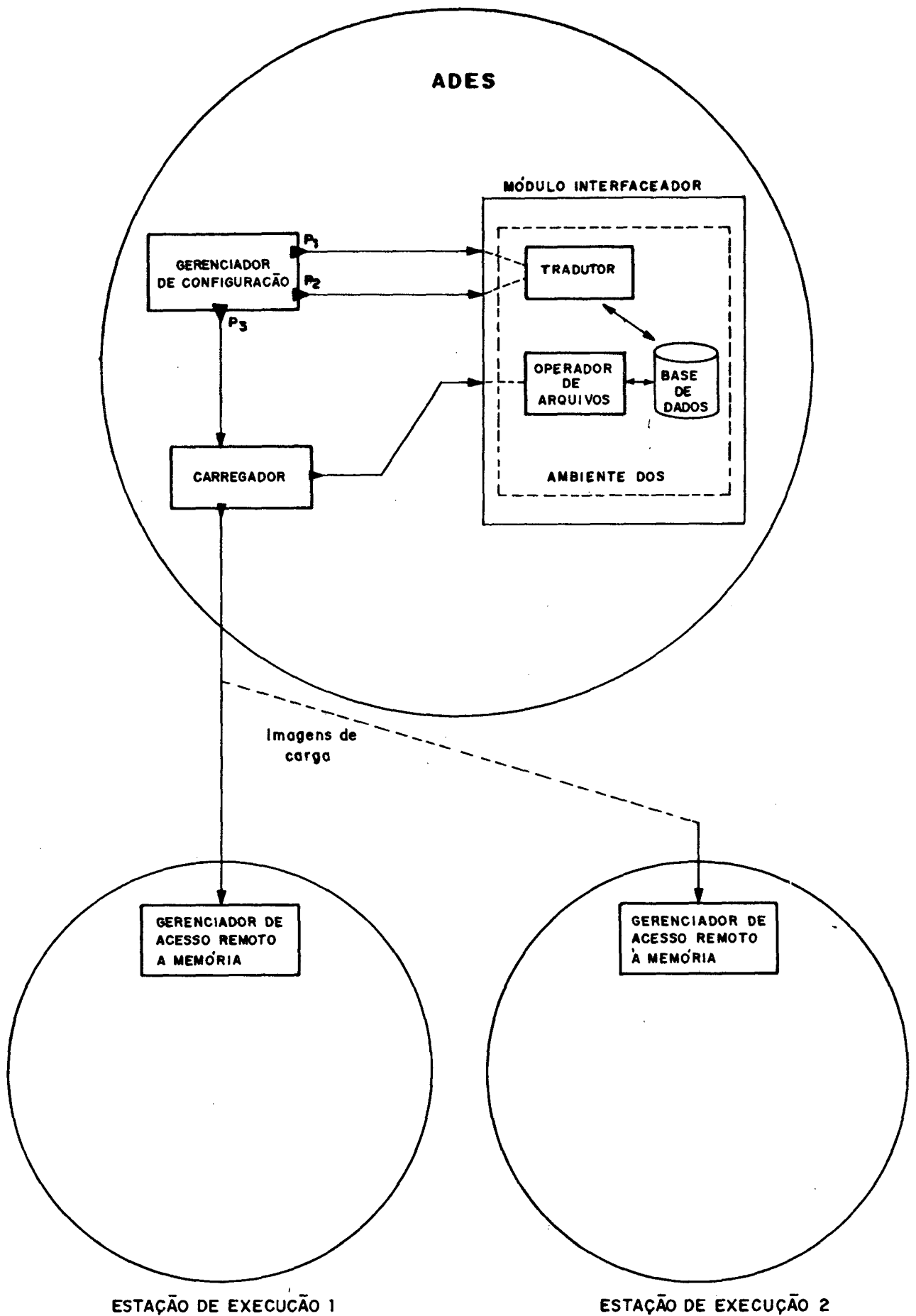


Fig. 5.1 - Processo de Carga na Configuração Dinâmica

carregá-la via servidor de rede. Os blocos de mensagens são enviados ao gerenciador de acesso remoto à memória na Estação de Execução que efetiva o carregamento.

5.2.2. Ativação do Processo de Configuração via Aplicação

Esta característica possibilita ativar uma configuração dinâmica. O objetivo é realizar uma reconfiguração pré-planejada à partir de um evento iniciado por um módulo de aplicação e complementado pelo gerenciador de configuração. Através de uma declaração especial na Linguagem de Componentes Elementares, denominada por ALIVE, o gerenciador de configuração recebe um pedido de configuração dinâmica, via um porto padrão específico, e inicia o processo. Várias estratégias podem ser utilizadas afim de otimizar este processo, por exemplo, ter sempre disponível a(s) especificação(ões) já validada(s) com respeito ao estado atual do sistema. Caso ocorra uma mudança de estado da configuração do sistema, as especificações de mudanças, previamente escritas, devem ser validadas com relação a este novo estado.

5.3. Pontos de Sincronização para Mudanças na Configuração

A configuração dinâmica em sistemas distribuídos, como definida em (Kramer, 85) envolve um conjunto de problemas que merecem uma reflexão cuidadosa. A sua realização exige uma análise ampla, envolvendo o estado do processo controlado (aplicação) e o mecanismo de configuração. É necessário que as consequências da alteração não se reflitam negativamente sobre o processo controlado.

No sentido de minimizar as possíveis inconsistências produzidas no sistema informático, foi introduzida a noção de **pontos de sincronização** em (Magee, 87), visando determinar instantes mais apropriados para mudanças. Esta noção é prevista no nosso modelo de configuração dinâmica; para tanto, deve levarem consideração a característica multi-tarefas do módulo no paradigma adotado.

A idéia central destes pontos de sincronização é permitir que módulos envolvidos com mudanças estejam inativos quando da realização destas, ou seja, o módulo está para iniciar ou já terminou uma operação importante. Os pontos de sincronização devem ser definidos pelo projetista e resultam na suspensão de todas as tarefas do módulo através de uma função ativada nestes pontos sob determinadas circunstâncias. Este processo não é muito simples, devido principalmente à característica multi-tarefa dos módulos, porém uma sugestão para sua implementação é uma operação à duas fases do gerenciador de módulos:

- Na **primeira**, o gerenciador de módulo recebe uma mensagem de que um determinado módulo está sujeito à reconfiguração; imediatamente este estado é registrado no bloco de controle da instância. Após esta operação, o gerenciador vai para a fila de espera.
- Na **segunda**, o gerenciador ativado pela suspensão da última tarefa pertencente ao módulo efetiva as operações. Isto acontecerá quando as tarefas chegarem no ponto de sincronização e se auto-suspenderem, sendo que é de responsabilidade da última tarefa a ativação do gerenciador.

5.4. ATRIBUIÇÃO DE PRIORIDADES VIA LINCS

Na implementação atual a atribuição de prioridades as tarefas pertencentes ao módulo é realizada na linguagem LINCE. Este tratamento poderá ser inconveniente considerando que um módulo ao ser transferido de uma estação para outra, necessitará que as prioridades atribuídas sejam adequadas ao novo contexto de execução. A solução atual é submeter o módulo a um novo processo de compilação o quê de certa forma não é a mais indicada.

Uma solução mais apropriada é a atribuição de prioridade durante a instanciação do módulo, implicando em alterações na LINCE que deve permitir a atribuição de prioridade por parametrização. Considerando que a LINCS suporta a parametrização por valores, a atribuição de prioridade pode ser então realizada em tempo de configuração.

5.5. CONCLUSÃO FINAL

A presente dissertação tratou do projeto e da implementação de um configurador de sistemas distribuídos. Para atender aos objetivos de configuração de aplicativos distribuídos foi desenvolvida uma linguagem de Configuração de Sistemas - LINCS que incorporou fielmente as propriedades do paradigma de programação adotado.

Para dotar a linguagem de construções poderosas para a expressão da composição do sistema foram implementadas características auxiliares (grupo e família) que consumiu muito esforço intelectual no desenvolvimento do protótipo.

O desenvolvimento do suporte para a configuração estática se sucedeu de uma forma relativamente rápida, permitindo inclusive uma melhora sensível na primeira versão implementada. A configuração estática atual atende plenamente aos objetivos propostos, satisfazendo todos os requisitos iniciais, não se tem no momento nenhuma ponderação a ser feita. O esforço de pesquisa em relação à configuração deve agora ser concentrado na configuração dinâmica.

BIBLIOGRAFIA

- Booch G. - "Object-Oriented Development", IEEE Trans. Software Engineering, Vol. SE-12, N.2, 211-221, Feb 1986.
- Bruno G., and Balsamo, A. - "Petri Net-Based Object Oriented Modelling of Distributed Systems", OOPLSA'86 Conference, 284-293, Sep 1986.
- Cetus - Manual de Utilização do driver de rede versão 1.04, Agosto 87.
- DeRemer F., and Kron, H.H. - "Programming-in-the-Large Versus Programming-in-the-Small", IEEE Trans. Software Engineering, Vol. SE-2, N.2, 80-86, June 1976.
- Feldman S.I. - "MAKE - A program for maintaining computer programs", Software - Practice and Experience, Vol.9, 255-265, 1979.
- Fraga J. S., Silva E. S., Farines J. M., Souza, L. E., Nacamura Jr. L. - "A Software Environment for Distributed Applications", 15. IFAC/IFIC, Valencia, Spain, 1988.
- Geschke C.M., Morris D.H. and Satterthwaite E.H. - "Early Experience with Mesa", Communications of the ACM, Vol. 20. N.8, 540-553, Aug 1977.

Kamel, Ragui F. - "Effect of Modularity on System Evolution",
IEEE Software, 48-54, Jan. 1987.

Kerningham, B.W., and Ritchie D.M. - "The C Programming Language", Englewood Cliffs, N.J.: Prentice-Hall, 1978.

Krakowiak, S. - "Systèmes Intégrés de Production de Logiciel: Concepts et réalisations", T.S.I., Vol. 1, N. 3, 187-200, 82.

Kramer J., Magge J., Sloman M., Lister A. - "Conic: An Integrated Approach to Distributed Computer Control Systems", IEE Proc., Vol. 130, Pt_E, N. 1, 1-10, Jan. 1983.

Kramer J., Magge J. - "Dynamic Configuration of the Distributed Systems", IEEE Trans. Software Engineering, SE-11, Vol. 4, 425-436, April 1985.

Maccabe A.B. - "Language Features for Fully Distributed Processing Systems", Georgia Institute of Technology, GIT-ICS-82/12, Aug 1985.

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

Mitchell J.G., Maybury W. and Sweet R.E. - Mesa Language Manual, Xerox Palo Alto Research Center Report CSL-79/3, April 1979.

Mod - Ministère Americain de la Defense. Manuel de Référence du Langage de Programmation ADA, 82..

Nacamura Jr. L. - "Projeto e Implementação de um Núcleo de Sistema Operacional Distribuído com Mecanismos para Tempo Real", Dissertação de Mestrado, CPGEEL, UFSC, Jul 1988.

Narayanaswamy, K., et all. - "Maintaining Configurations of Evolving Software Systems", IEEE Trans. Software Engineering, Vol. SE-13, N.3, 324-334, March 1987.

Parnas, D. L. - "On the Criteria To Be Used in Decomposing Systems Into Modules", Communications of the ACM, 1053-1058, Dec. 1972.

Ramammoorthy, C. V. - "Programming in the Large", IEEE Trans. Software Engineering, Vol. SE-12, N.7, 769-783, July 1986.

Redell, D.D., et all - "Pilot: An Operating System for a Personal Computer", Communications of the ACM, Vol. 23, N.2, 81-92, Feb. 1980.

Rochkind, M.J. - "The Source Code Control System", IEEE Trans. Software Engineering, Vol. SE-1, N.1, 364-370, Dec 1975.

Ross D. T. - "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Trans. Software Engineering, Vol. SE-3, N. 1, 16-34, Jan 1977.

- Ross D. T., Schoman Jr. K. E. - "Structured Analysis for Requirements Definition", IEEE Trans. Software Engineering, Vol. SE-3, N. 1, 69-84, Jan 1977.
- Schneidewind, N. F. - "The State of Software Maintenance", IEEE Trans. Software Engineering, Vol. SE-13, N.3, 303-310, March 1987.
- Silva E. S. - "Uma Linguagem de Programação de Componentes Elementares para Aplicações Distribuídas em Tempo Real: Projeto e Implementação", Dissertação de Mestrado, CPGEEL, UFSC, Ago 1988.
- Simpson H.R., Jackson K. - "Process Synchronization in Mascot", The Computer Journal, Vol.22, N.4, 332-345, Nov. 79.
- Sweet R.E. - "The Mesa Programming Environment", SIGPLAN NOTICES, 85.
- Tichy W.F., "RCS - A System for Version Control", Software Practice and Experience, Vol.15, N.7, 637-654, 1985.
- Weber, H., et all. - "Specification of Modular Systems", IEEE Trans. Software Engineering, Vol. SE-12, N.7, 784 - 798, July 1986.
- Yau S.S., Tsai J. - "A survey of Software Design Techniques", IEEE Trans. Software Engineering, Vol. SE-12, N.6, 713-721, Jun 1986.

APÊNDICE A

ESTRUTURAS UTILIZADAS NA IMPLEMENTAÇÃO DO PROCESSADOR LINCS

O processamento das declarações LINCS envolve a criação de várias estruturas; abaixo estão listadas as mais importantes e que não foram apresentadas no texto. Estas estruturas são utilizadas para processamentos intermediários e para formar a base de dados do sistema.

Estrutura referente à estação

```
struct ESTAÇÃO
{
    int                ind_est, con_ind_inst;
    unsigned char      end_placa, niv_int,
                      status;
    unsigned long int  tam_mem_disp,
                      tam_mem_ocupada;
    unsigned int       end_atual;
    struct TIPOEST     *ptipoest;
    struct LINKEST     *plinkest;
    struct INSTANCIA   *pinstaest;
    struct INDLIVRE    *pind_inst;
    struct ESTAÇÃO     *prox;
};
```

Estrutura referente à tipos módulos de estação

```
struct TIPOEST
{
    char            id_timo [MAX_C];
    int             num_ins, ind_tip_mod;
    unsigned int    tam_cod,
                   tam_dta,
                   tam_stack,
                   tam_heap,
                   ind_dta,
                   end_tipo,
                   inicial;
    struct TIPOEST *prox;
};
```

Estrutura referente a instâncias no sistema

```
struct INSTASIS
{
    char            id_insta [Max_C],
                   id_tipo [Max_C];
    int             ind_tip_mod,
                   imin, imax,
                   emin, emax;
    struct INSTASIS *prox;
};
```

Estrutura referente a conexões em estação

```

struct LINKEST
{
    char          id_insta_ps  [Max_C],
                 id_porto_sai [Max_C],
                 id_insta_pe  [Max_C],
                 id_porto_pe  [Max_C];

    int          ind_ins_mod_ps,
                 ind_ps,
                 ind_ins_mod_pe,
                 ind_pe,
                 ind_est_pe;

    struct LINKEST *prox;
};

```

Estrutura referente à instância

```

struct INSTÂNCIA
{
    char          id_insta [Max_C];
                 id_tipo  [Max_C];

    int          ind_tip_mod,
                 imin, imax, base;

    struct PARÂMETROS *pparam;
    struct INSTÂNCIA *prox;
};

```

Estrutura referente à parâmetros

```
struct PARÂMETROS
{
    char          id_param [Max_C];
    int           tipo,
                vint;
    float         vreal;
    struct PARÂMETROS *prox;
} ;
```

Estruturas utilizadas no processamento de grupo

```
struct TIPOFORM
{
    char          id_const [Max_C];
                id_tipo [Max_C],
                pachar [Max_C];
    float         pareal;
    int           paint;
    struct TIPOFORM *prox;
} ;
```

```
struct TIPODADO
{
    char          id_tida [Max_C],
                id_unidef [Max_C];
    struct TIPODADO *prox;
} ;
```

```
struct TIPOMODULO
```

```
{  
    char          id_timo [Max_C];  
    int           ind_tip_mod;  
    struct TIPOMODULO *prox;  
} ;
```

```
struct INSTAGRUPPO
```

```
{  
    char          id_insta [Max_C], id_tipo [Max_C];  
    int           indice;  
    struct PORTOS *pportos;  
    struct INSTAGRUPPO *prox;  
} ;
```

```
struct PORTOS
```

```
{  
    char          id_porto [Max_C];  
    struct PORTMODULO *pmod;  
    struct PORTOS *prox;  
} ;
```

```
struct PORTMODULO
```

```
{  
    char          id_porto [Max_C], id_insta [Max_C],  
                id_tipo [Max_C];  
    int           imin, imax,  
                pmin, pmax,  
                base, num, tipo;  
    struct PORTMODULO *prox;  
} ;
```

```
struct LINKSAIDA
{
    char            id_isnta [Max_C],
                  id_porto [Max_C];
    int             imin,
                  imax,
                  base,
                  pmin,
                  pmax,
                  num,
                  tipo;
    struct LINKENTR *plinken;
    struct LINKSAIDA *prox;
} ;
```

```
struct LINKENTR
{
    char            id_isnta [Max_C],
                  id_porto [Max_C];
    int             imin, imax,
                  pmin, pmax,
                  base,
                  num,
                  tipo;
    struct LINKENTR *prox;
};
```

APÊNDICE B

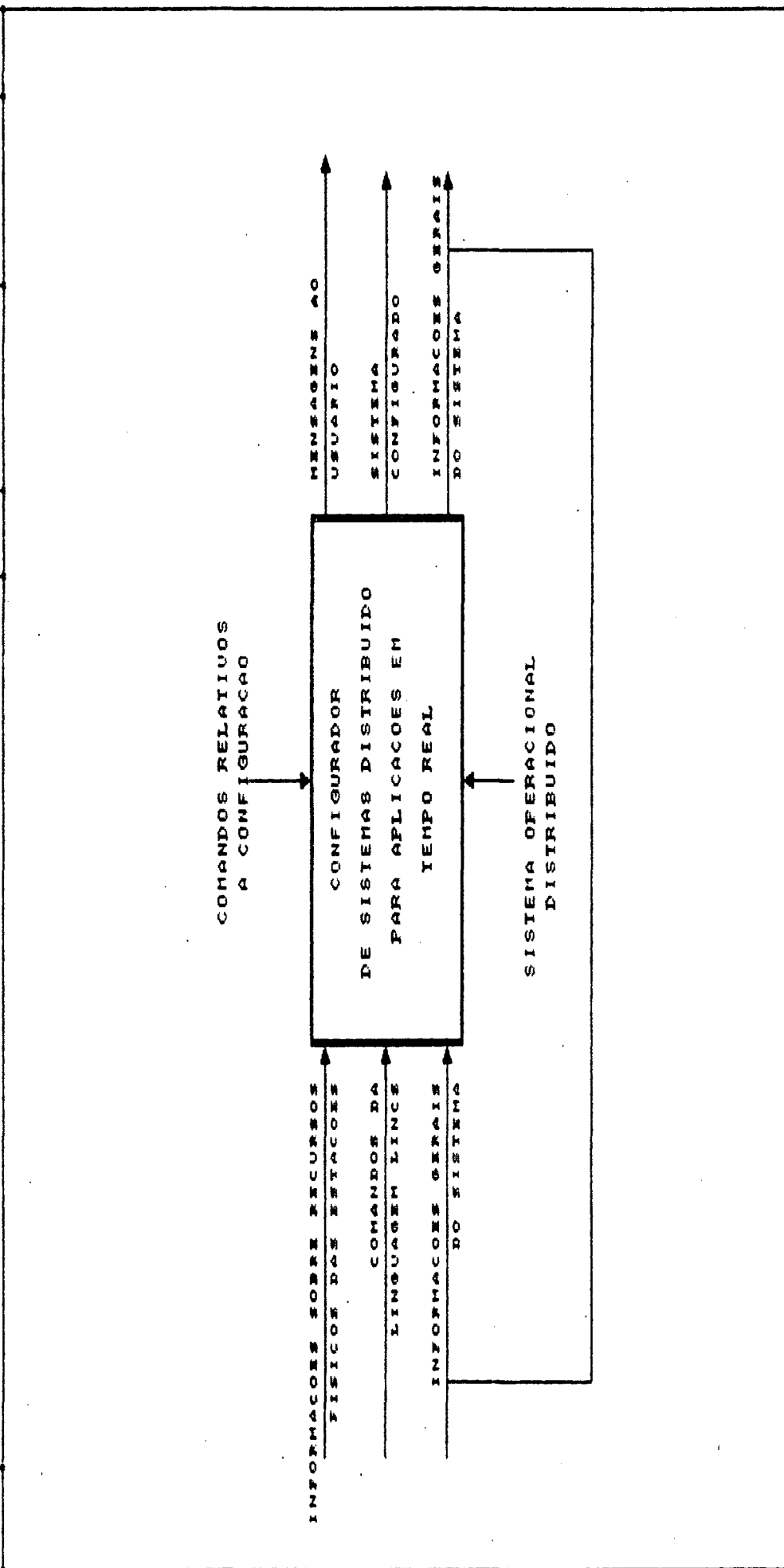
ESPECIFICAÇÃO FUNCIONAL DO PROCESSO DE CONFIGURAÇÃO DE SISTEMAS DISTRIBUÍDOS UTILIZANDO SADT.

O desenvolvimento de sistemas é constituído por uma sucessão de problemas que devem ser entendidos e solucionados pelo projetista. Qualquer solução adotada deve ser suficientemente flexível para acomodar as mudanças que naturalmente irão aparecer no decorrer do projeto. A necessidade em ser adotada uma metodologia que incorpore todos os aspectos do desenvolvimento do sistema antes do projeto surgiu naturalmente devido a complexidade do problema. Neste sentido, o início do desenvolvimento do configurador foi auxiliado por uma ferramenta afim de decompor o problema e apresentá-lo em uma forma organizada. A Técnica de Projeto e Análise Estruturada - SADT (Ross, 77a) permitiu à uma especificação inicial que correspondeu fortemente com o nível de abstração necessário para visualizar e definir o problema. O uso dessa ferramenta auxiliou positivamente na organização das idéias e na definição das funções que deveriam ser realizadas.

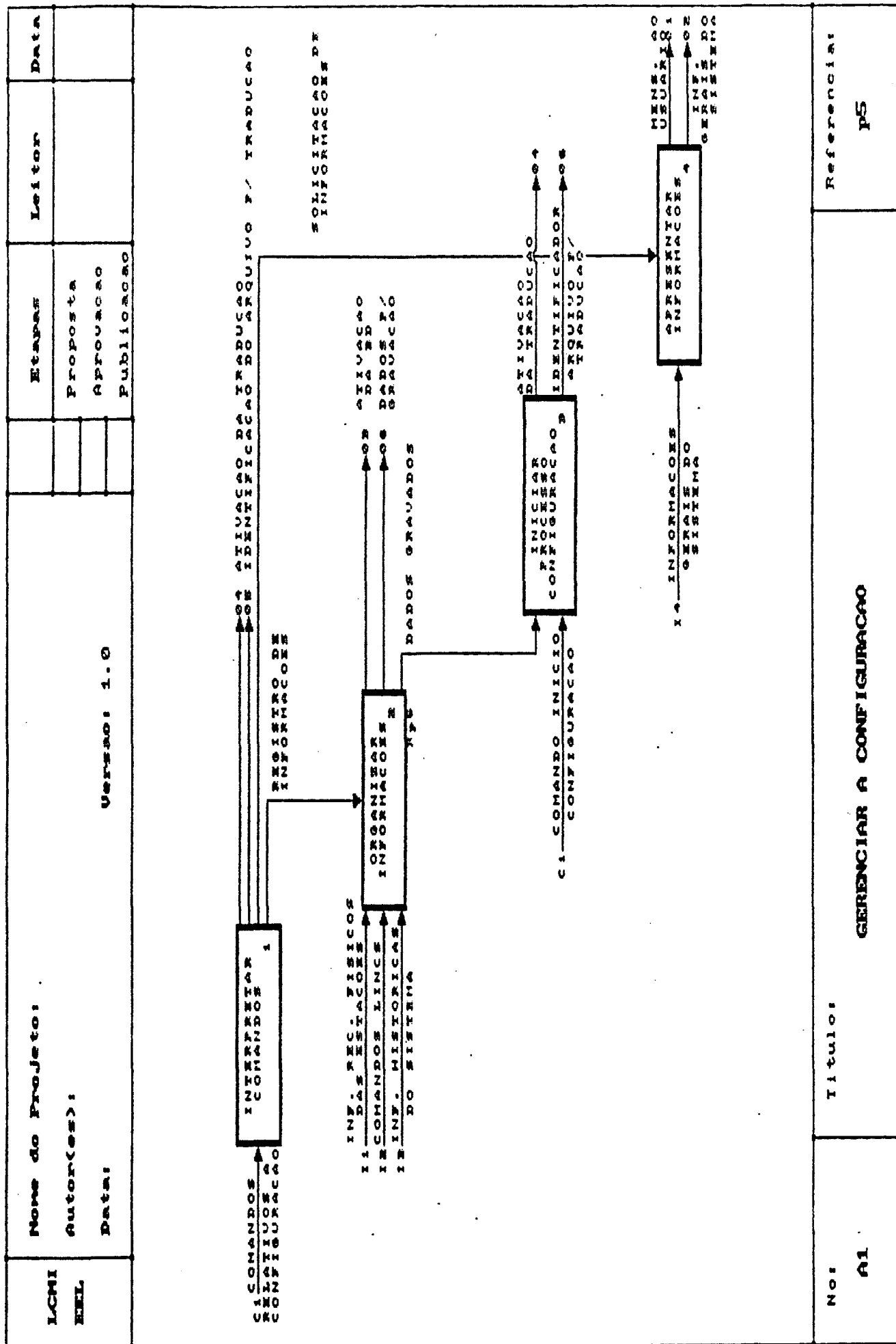
Embora tenham sido desenvolvidos mais diagramas do que o aqui apresentados, optou-se por eliminá-los uma vez que o mesmo não prejudica a finalidade deste apêndice, ou seja, fornecer através de uma formalização uma visão geral dos aspectos de configuração.

<p>LCMI EEL</p>	<p>Nome do Projeto: Configurador Autor(es): Luiz Edival de Souza Versao: 1.1</p>			<p>Estapas</p>	<p>Leitor</p>	<p>Data</p>
<p>Data:</p>			<p>Proposta Aprovacao Publicacao</p>	<p> </p>		<p> </p>
<pre> graph TD Root(()) --- A0((A0)) Root --- A1((A1)) Root --- A2((A2)) Root --- A3((A3)) A1 --- A12((A12)) A2 --- A31((A31)) A3 --- A4((A4)) A3 --- A32((A32)) A32 --- A322((A322)) A32 --- A323((A323)) </pre>						
<p>No:</p>	<p> </p>					<p>Referencia:</p>

<p>LCMI EEL</p>	<p>Nome do Projeto: Configurador Autor(es): Luiz Edival de Souza Data: Versao: 1.1</p>			<p>Etapas Proposta Aprovacao Publicacao</p>	<p>Leitor</p>	<p>Data</p>
---------------------	--	--	--	---	---------------	-------------



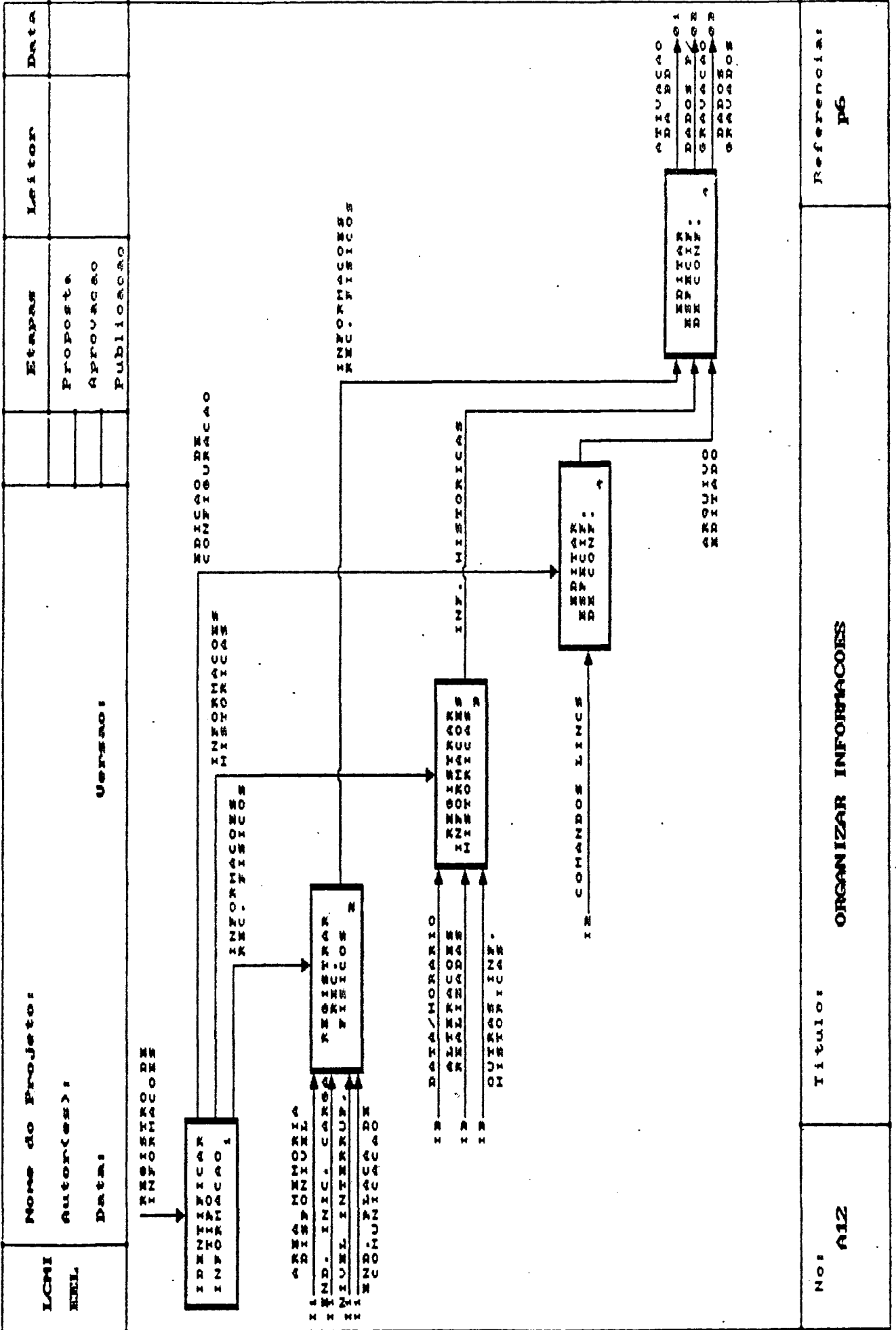
<p>Not AB</p>	<p>Titulor</p>	<p>Referencia: p3</p>
-------------------	----------------	---------------------------



Referencia: P5

Título: GERENCIAR A CONFIGURACAO

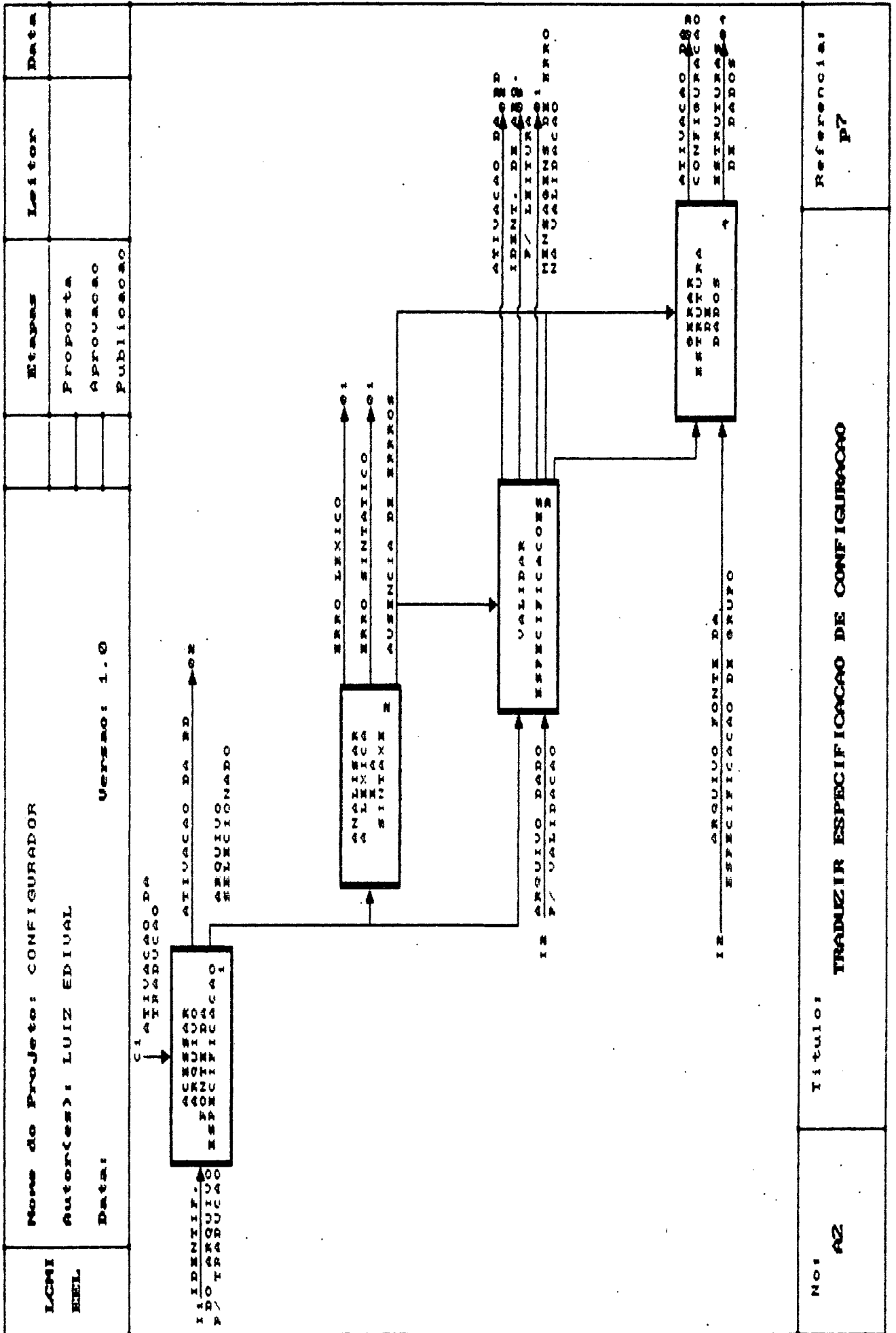
No: A1



№: A12

Título: ORGANIZAR INFORMACOES

Referencia: p6



Nome do Projeto: CONFIGURADOR
 Autor(es): LUIZ EDIVAL
 Data: Versao: 1.0

Referencia:

Titulo: TRADUZIR ESPECIFICACAO DE CONFIGURACAO

Referencia:

P7

Nome do Projeto: CONFIGURADOR
 Autor(es): LUIZ EDIVAL
 Data: Versao: 1.0

Referencia:

Titulo: TRADUZIR ESPECIFICACAO DE CONFIGURACAO

Referencia:

P7

Nome do Projeto: CONFIGURADOR
 Autor(es): LUIZ EDIVAL
 Data: Versao: 1.0

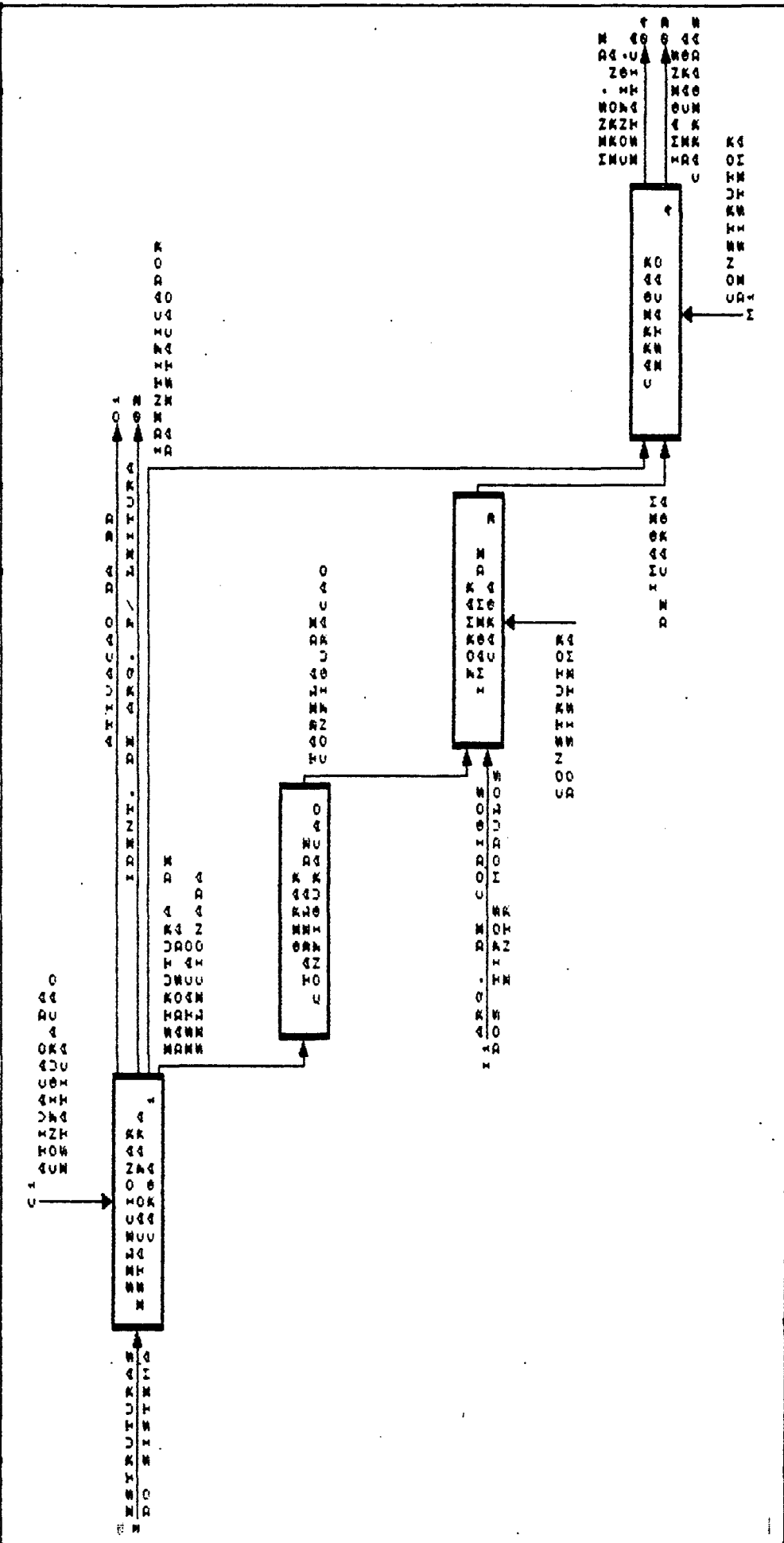
Referencia:

Titulo: TRADUZIR ESPECIFICACAO DE CONFIGURACAO

Referencia:

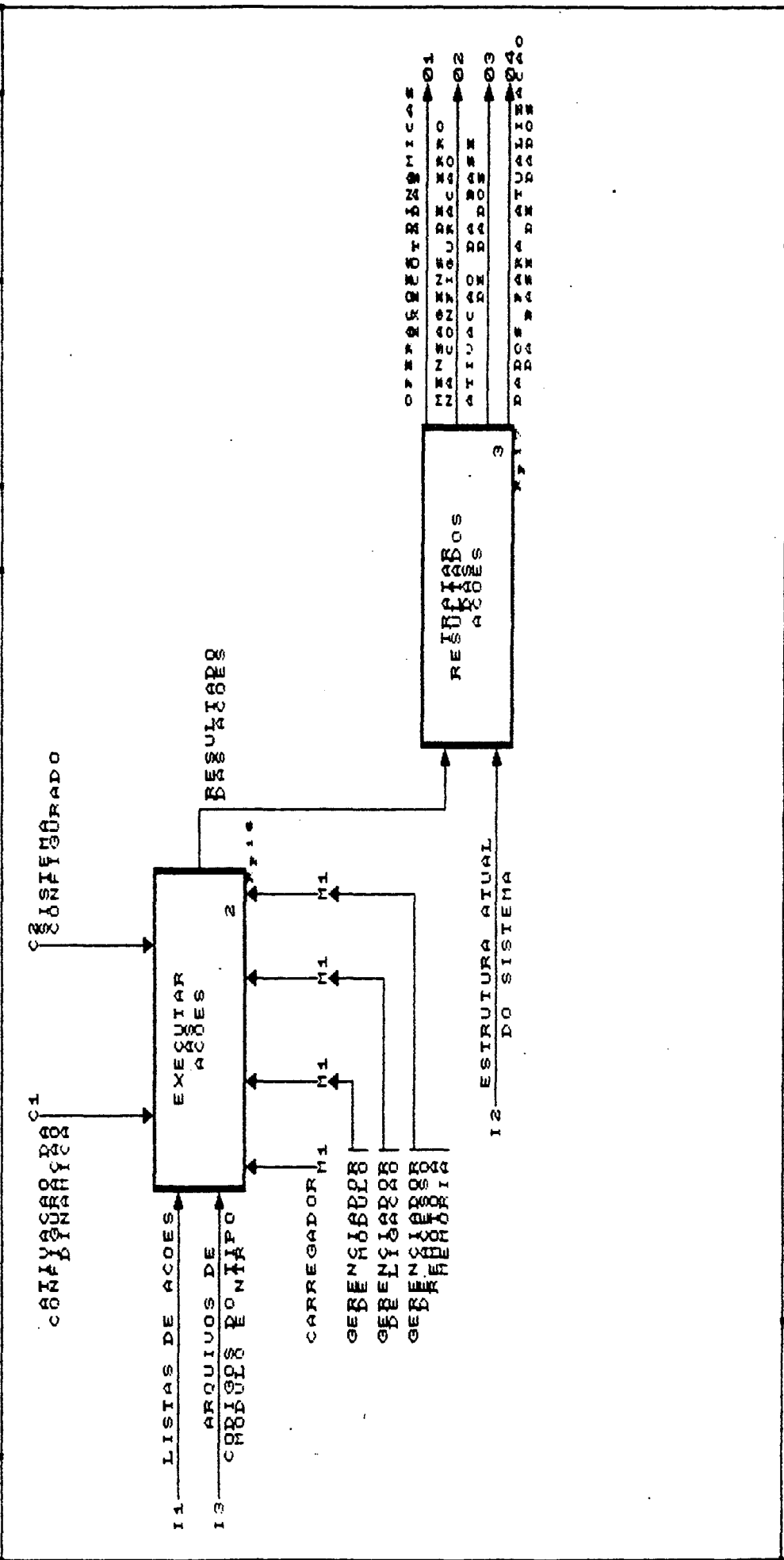
P7

L. L. CMI E. FEL.	Nome do Projeto: CONFIGURADOR		Leitor	Data
	Autor(es): LUIZ EDIVAL DE SOUZA			
	Data: Versao: 1.0			
Etapas		Proposta		
		Aprovacao		
		Publicacao		



Not	Referencia:
A31	P11
Titulo: CONFIGURAR ESTATICAMENTE	

LCMI EHEL	Nome do Projeto: CONFIGURADOR PARA SDCD			Leitor	Data
	Autor(es): LUIZ EDIVAL				
	Data: 13/07/87				
	Versão: 1.0				
Etapas					
Proposta					
Aprovacao					
Publicacao					



No: A32	Titulo: CONFIGURAR DINAMICAMENTE	Referencia: p15
---------	----------------------------------	-----------------

APÊNDICE C

EXEMPLO - CÉLULA FLEXÍVEL DE MANUFATURA

1.1. Descrição do Exemplo

De forma a verificar a integração do ambiente utilizando-se das ferramentas LINCE e LINC3 foi realizado a simulação de uma célula flexível de manufatura (ver Fig. 1) constituída pelos seguintes componentes:

- uma esteira;
- uma câmera;
- dois robôs: Rp e Ra;
- um torno; e
- um armazém.

A esteira possui duas posições de parada:

- P1: para que a peça possa ser visualizada pela câmera; e
- P2: para que a peça possa ser retirada da esteira pelo robô Rp.

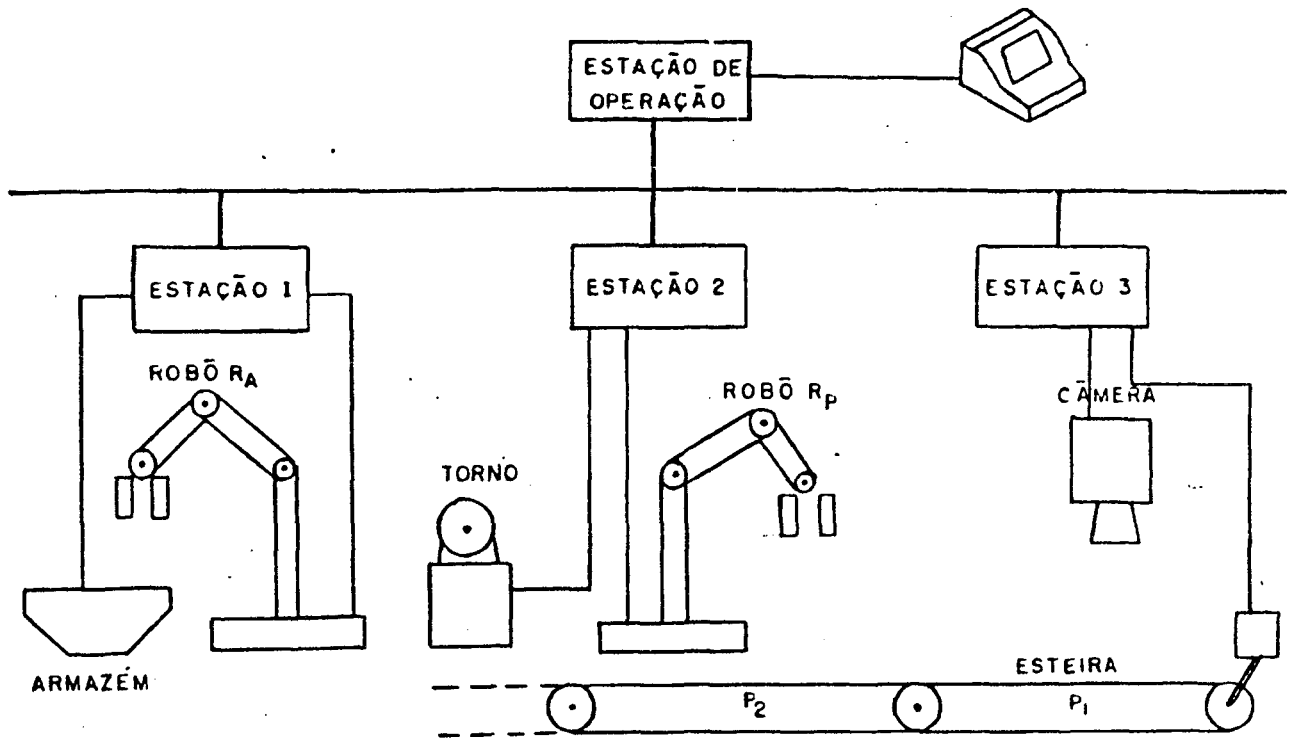


Fig. 1 - Célula Flexível de manufatura

As posições S_{Rp} corresponde a posição de repouso do robô R_p , sendo que a posição S_{Ra} é posição de repouso do robô R_a .

O sistema consiste de uma esteira que transporta peças de diversos tipos. A distribuição de peças sobre a esteira é aleatória. Quando a esteira encontra-se na posição P_1 , o movimento da esteira é interrompido para que seja feita a visualização da peça, pela camera. Uma vez terminada a visualização a esteira é novamente colocada em movimento. A informação obtida por esta visualização é então classificada através de parâmetros pré-estabelecidos. Se as características visualizadas não se enquadra em nenhum destes parâmetros a peça é rejeitada. Quando a peça encontra-se na posição P_2 , o movimento da esteira é novamente interrompido para que a peça seja retirada pelo robô R_p . Várias peças podem se encontrar entre P_1 e P_2 . Quando a esteira encontra-se em repouso, para que o robô R_p retire a peça, é necessário que a informação da

localização (direita, esquerda e centro) da peça tenha sido recebida. Entretanto se a peça foi rejeitada pela visualização esta continua na esteira (não é pega pelo robô). Um vez retirada de P2 (peça válida), a esteira é posta em movimento.

Para que a peça possa ser colocada no torno é necessário que o torno esteja disponível e o robô Ra não se encontre nas proximidades do torno. Após a colocação das peça no torno o robô Rp retorna a posição de repouso, estando assim disponível para retirar outra peça da esteira. Ao final do torneamento a peça pode ser retirada do torno pelo robô Ra, desde que o robô Rp não se encontre nas proximidades do torno. Uma vez retirada a peça o robô Ra coloca a peça no armazém. No final da colocação o robô Ra retorna a posição de repouso.

1.2. Decomposição do Sistema em Módulos

Baseado na descrição do item anterior a aplicação (célula flexível) foi decomposta em vários módulos, sendo que cada um destes representa operações executadas por componentes ou em componentes do sistema (robô, esteira, etc). De forma a simular funcionamento real de cada componente foram incorporadas também na aplicação módulos simuladores. A definição do corpo e da interface dos módulos são apresentados na sequência onde poderão ser identificados os tipos de portos e as tarefas correspondentes a cada módulo.

```

MODULE vis ;
USE grafo1, grafo2, def1;
PORT  MVIS1 , MVIS2 , MVIS5 , MVIS7 , MVIS8 , MVIS10 : IN ( char ) ;
      MVIS3 : IN ( t2_est ) ;
      MVIS4 : IN ( t_est ) ;
      MVIS9 : IN ( integer ) ;
TASK inicia_tela <1000>, 3 ;
  BEGIN
    ( Desenha na tela o grafico estatico simulando a
      celula flexivel. A tarefa e' suspensa e a tarefa inicia )
  END;
TASK visu <4096>, 5 ;
  CONST  em_reposo = '0' ;
         visualizando = '1' ;
         classificando = '1' ;
         apanhando = '1' ;
         colocando = '2' ;
         esperando = '0' ;
         torneando = '1' ;
         retirando = '1' ;
         armazenando = '2' ;
         em_troca = '0' ;
         disponivel = '1' ;
  VAR    estado : char ;
         contador : t2_est ;
         v : t_est ;
         cont_aux : t3_est ;
         n_armazena : integer ;
         i : integer ;
  PORT  TVIS1 , TVIS2 , TVIS5 , TVIS7 , TVIS8 , TVIS10 : IN ( char ) ;
      TVIS3 : IN ( t2_est ) ;
      TVIS4 : IN ( t_est ) ;
      TVIS9 : IN ( integer ) ;
  BEGIN
    LOOP
      ( Recebe atraves de um "select" em varios portos as mensagens
        que darao o movimento dos componentes da celula. Os portos
        deste modulo tem uma correspondencia com o movimento de cada
        componente )
    END ;
  END ;
LINK MVIS1 TO TVIS1;
     MVIS2 TO TVIS2;
     MVIS3 TO TVIS3;
     MVIS4 TO TVIS4;
     MVIS5 TO TVIS5;
     MVIS7 TO TVIS7;
     MVIS8 TO TVIS8;
     MVIS9 TO TVIS9;
     MVIS10 TO TVIS10;
ENDMODULE.

```

```

MODULE simu ;
USE GRAFO1, GRAFO2;
CONST f1 = 59;
    em_repouso = '0' ;
    apanhando = '1' ;
    colocando = '2' ;
    t_apanhar = 10 ;
    t_colocar = 15 ;
    t_repouso = 5 ;
    retirando = '1' ;
    t_tirar = 10 ;
    t_armazenar = 15 ;
    t_trocar = 30 ;
    armazenando = '2' ;
    em_troca = '0' ;
    disponivel = '1' ;
    torneando = '1' ;
    t_tipo1 = 10 ;
    t_tipo2 = 20 ;
    t_tipo3 = 30 ;
    esperando = '0' ;
    t_visualizacao = 20 ;
    visualizando = '1' ;
    max = 32767 ;
    i1 = 6 ;
    i2 = 18 ;

PORT MSE2 : IN ( signaltype ) ;
MSE3 : OUT ( real REPLY signaltype ) ;
MSE4 : OUT ( char REPLY signaltype ) ;
MSE5 : OUT ( t_est ) ;
MSRP1 : IN ( char REPLY signaltype ) ;
MSRP3, MSRA1, MSTO2, MSTR1, MSC2 : OUT ( char ) ;
MSRP2, MSRA2, MSRA3, MSTR2 : IN ( signaltype REPLY signaltype ) ;
MSTO1 : IN ( char REPLY signaltype ) ;
MSC1 : IN ( signaltype REPLY integer ) ;

TASK simu_esteira <1024>, 10;
VAR sinal : signaltype ;
    guarda, i : integer;
    v : t_est;
    p : t1_est;
    x : integer ;
    p2 : char;
PORT TSE1 : IN ( signaltype ) <SB> ;
TSE2 : IN ( signaltype ) ;
TSE3 : OUT ( real REPLY signaltype ) ;
TSE4 : OUT ( char REPLY signaltype ) ;
TSE5 : OUT ( t_est ) ;
BEGIN
    LOOP
        ( Realiza a distribuicao aleatoria de pecas na esteira e envia
          mensagens para o modulo vis informando o estado atual da esteira )
    END;
END;

TASK inicio_simu <300>, 7;
VAR tecla: integer ;
    sinal: signaltype ;
PORT TIS1: OUT(signaltype) <SB>;
BEGIN
    LOOP
        ( Espera a ocorrencia de uma autorizacao de inicio de simulacao
          identificada pela tecla F1, envia mensagem ao modulo simu_est
          sinalizando o inicio de operacao )
    END;
END;

```

```

TASK simula_Rp <800>,9;
VAR estado : char ;
    posicao : char ;
    sinal : signaltype ;
PORT TSRP1 : IN ( char REPLY signaltype ) ;
    TSRP2 : IN ( signaltype REPLY signaltype ) ;
    TSRP3 : OUT ( char ) ;

BEGIN
  LOOP
    ( Simula o movimento do robo Rp atraves de tres estados: em repouso,
      apanhando e colocando. Envia estado correspondente ao modulo de
      simulacao (vis) )
  END;
END ;

TASK simula_Ra <800>,6;
VAR estado : char ;
    sinal : signaltype ;
PORT TSRA1 : OUT ( char ) ;
    TSRA2 : IN ( signaltype REPLY signaltype ) ;
    TSRA3 : IN ( signaltype REPLY signaltype ) ;

BEGIN
  LOOP
    ( Simula o movimento do robo Ra atraves de tres estados: em repouso,
      retirando e armazenando. Envia estado correspondente ao modulo de
      simulacao )
  END;
END ;

TASK simu_troca <1024> ,6;
VAR sinal : signaltype ;
    estado : char ;
PORT TSTR1 : OUT ( char ) ;
    TSTR2 : IN ( signaltype REPLY signaltype ) ;

BEGIN
  LOOP
    ( Simula a troca de armazem quando a contagem de pecas torneadas
      for igual a 20 )
  END ;
END ;

TASK simu_torno <800> ,5;
VAR tipo, estado : char ;
    sinal : signaltype;

PORT TSTO1 : IN ( char REPLY signaltype ) ;
    TSTO2 : OUT ( char ) ;

BEGIN
  LOOP
    ( Simula o torno que pode estar em dois estados: em operacao e
      esperando )
  END ;
END ;

TASK simu_camera <1024>,9;
VAR l : integer ;
    sinal : signaltype ;
    estado : char ;
PORT TSC1 : IN ( signaltype REPLY Integer ) ;
    TSC2 : OUT ( char ) ;

BEGIN
  LOOP
    ( Simula a camera que pode estar em dois estados: visualizado ou
      em repouso )
  END ;
END ;

LINK TIS1 TO TSE1 ;
MSE2 TO TSE2 ;
MSE3 TO TSE3 ;
MSE4 TO TSE4 ;
MSE5 TO TSE5 ;
MSRP1 TO TSRP1 ;
MSRP3 TO TSRP3 ;
MSRA1 TO TSRA1;
MSTO2 TO TSTO2 ;
MSTR1 TO TSTR1 ;
MSC2 TO TSC2 ;
MSRP2 TO TSRP2 ;
MSRA2 TO TSRA2 ;
MSRA3 TO TSRA3 ;
MSTR2 TO TSTR2 ;
MSTO1 TO TSTO1 ;
MSC1 TO TSC1 ;

ENDMODULE.

```

```

MODULE camera;
USE def1;
PORT    MCAM1 : IN (real REPLY signaltype);
        MCAM2 : OUT (signaltype REPLY integer);
        MCAM3: OUT (t_info);
TASK cam <1024>,5;
    VAR    p: real;
           loc1: integer;
           info: t_info;
           sinal: signaltype;
    PORT    TCAM1 : IN (real REPLY signaltype);
           TCAM2 : OUT (signaltype REPLY integer);
           TCAM3 : OUT (t_info);
    BEGIN
        LOOP
            ( Ativa o modulo de simulacao da camera e permite
              a classificacao da peca enviando um sinal para o
              modulo classificar )
        END;
    END;
TASK ociosa <8192>,13;
    BEGIN
        LOOP
            ( Nao faz nada. E responsavel pela ocupacao do processador
              enquanto as tarefas nao estiverem prontas para execucao )
        END;
    END;
LINK    MCAM1 TO TCAM1 ;
        MCAM2 TO TCAM2 ;
        MCAM3 TO TCAM3 ;
ENDMODULE.

```

```

MODULE transp;
PORT    MTR1: OUT (real REPLY signaltype);
        MTR2: IN (real REPLY signaltype);
        MTR3: IN (char REPLY signaltype);
        MTR4: OUT (signaltype REPLY signaltype);
TASK transportar <1024>,4;
    VAR    p1: real;
           p2: char;
           sinal: signaltype;
    PORT    TTR1: OUT (real REPLY signaltype);
           TTR2: IN (real REPLY signaltype);
           TTR3: IN (char REPLY signaltype);
           TTR4: OUT (signaltype REPLY signaltype);
    BEGIN
        LOOP
            SELECT
                ( Recebe mensagem em MTR2 ou MTR3 e sinaliza a operacao
                  de apanhar peca do torno ou visualizar peca atraves do
                  modulo camera )
            END;
        END;
    END;
LINK    MTR1 TO TTR1;
        MTR2 TO TTR2;
        MTR3 TO TTR3;
        MTR4 TO TTR4;
ENDMODULE.

```



```
MODULE armaze;
```

```
PORT MTP1: OUT ( signaltype REPLY signaltype );
      MTP2: IN  ( signaltype REPLY signaltype );
      MTP3: OUT ( signaltype REPLY signaltype );
      MTP4: OUT ( signaltype ) <SB>;
      MTP5: OUT ( char );
      MAZ2: OUT (signaltype REPLY signaltype);
      MTZ1: OUT (signaltype REPLY signaltype);
      MTZ3: OUT (integer);
```

```
TASK tirar_peca <1024>,8;
```

```
VAR sinal : signaltype;
```

```
PORT TTP1: OUT ( signaltype REPLY signaltype );
      TTP2: IN  ( signaltype REPLY signaltype );
      TTP3: OUT ( signaltype REPLY signaltype );
      TTP4: OUT ( signaltype ) <SB>;
      TTP5: OUT ( char );
```

```
BEGIN
```

```
  LOOP
```

```
    ( Recebe sinal para tirar peca, envia mensagens para o nova operacao
      com o torno, para o modulo vis e para o processo de armazenamento
      ativando o robo Ra )
```

```
  END;
```

```
END;
```

```
TASK armazenar <1024>,10;
```

```
VAR sinal: signaltype;
```

```
PORT TAZ1: IN  (signaltype) <SB>;
      TAZ2: OUT (signaltype REPLY signaltype);
      TAZ3: IN  (signaltype REPLY signaltype);
      TAZ4: OUT (char);
```

```
BEGIN
```

```
  LOOP
```

```
    ( Realiza a operacao de armazenamento atraves do robo Ra. E' sincronizado
      com a operacao troca de armazenam de forma a impedir uma armazenamento
      durante a troca )
```

```
  END;
```

```
END;
```

```
TASK trocar_armazem <1024>, 13;
```

```
CONST nmax=20;
```

```
VAR sinal: signaltype;
    n: integer;
```

```
PORT TTZ1: OUT (signaltype REPLY signaltype);
      TTZ2: OUT (signaltype REPLY signaltype);
      TTZ3: OUT (integer);
      TTZ4: OUT (char);
```

```
BEGIN
```

```
  n := 0;
```

```
  LOOP
```

```
    ( Realiza a contagem de pecas estocadas e executa a troca de
      armazem quando o numero de peca chegar a 20 )
```

```
  END;
```

```
END;
```

```
LINK MTP1 TO TTP1;
      MTP2 TO TTP2;
      MTP3 TO TTP3;
      MTP4 TO TTP4;
      MTP5 TO TTP5;
      MAZ2 TO TAZ2;
      MTZ3 TO TTZ3;
      MTZ1 TO TTZ1;
      TTP2 TO TAZ3;
      TTP4 TO TAZ1;
```

```
ENDMODULE.
```

MODULE torno ;

USE def1;

PORT

```
MAP1: OUT ( signaltype REPLY classe ) ;
MAP2: IN ( signaltype REPLY signaltype ) ;
MAP3: OUT ( char REPLY signaltype ) ;
MAP5: OUT ( char ) ;
MCT2: OUT ( signaltype REPLY signaltype ) ;
MCT3: OUT ( signaltype ) <SB> ;
MCT4: IN ( signaltype REPLY signaltype ) ;
MCT5: OUT ( char ) ;
MTN1: OUT ( signaltype REPLY char ) ;
MTN4: OUT ( char REPLY signaltype ) ;
MTN5: OUT ( signaltype REPLY signaltype ) ;
```

TASK apanhar <2048>, 5;

```
VAR sinal      : signaltype ;
    posicao1    : char ;
    classifica : classe ;
```

PORT

```
TAP1: OUT ( signaltype REPLY classe ) ;
TAP2: IN ( signaltype REPLY signaltype ) ;
TAP3: OUT ( char REPLY signaltype ) ;
TAP4: OUT ( signaltype REPLY signaltype ) ;
TAP5: OUT ( char ) ;
```

BEGIN

LOOP

(Testa a existencia de uma peca aprovada na posicao e envia mensagens para permitir o torneamento da peca)

END;

END;

TASK colocar_torno <2048>, 5;

```
VAR sinal: signaltype;
```

```
PORT TCT1: IN ( signaltype REPLY signaltype ) ;
      TCT2: OUT ( signaltype REPLY signaltype ) ;
      TCT3: OUT ( signaltype ) <SB> ;
      TCT4: IN ( signaltype REPLY signaltype ) ;
      TCT5: OUT ( char ) ;
```

BEGIN

LOOP

BEGIN

(Recebe sinal de apanhar peca, envia mensagem para o modulo vis e permite o torno a operar)

END;

END;

END;

TASK tornear <2048>, 5;

```
VAR tipo : char;
    sinal : signaltype;
```

```
PORT TTN1: OUT ( signaltype REPLY char ) ;
      TTN2: IN ( signaltype ) <SB>;
      TTN4: OUT ( char REPLY signaltype ) ;
      TTN5: OUT ( signaltype REPLY signaltype ) ;
      TTN6: OUT ( char ) ;
```

BEGIN

LOOP

(Realiza as operacoes envolvidas com o torneamento, enviando mensagem para o modulo vis e quando ocorre o fim de torneamento da peca. O tempo de torneamento depende da peca a ser trabalhada)

END;

END;

```
LINK TAP4 TO TCT1;
      TCT3 TO TTN2;
      MAP1 TO TAP1;
      MAP2 TO TAP2;
      MAP3 TO TAP3;
      MAP5 TO TAP5;
      MCT2 TO TCT2;
      MCT3 TO TCT3;
      MCT4 TO TCT4;
      MCT5 TO TCT5;
      MTN1 TO TTN1;
      MTN4 TO TTN4;
      MTN5 TO TTN5;
```

ENDMODULE.

```

MODULE classi;
  USE def1;
  CONST
    classificando = '1';
    em_repouso    = '0';
    max           = 32767;
    esquerda     = 'e';
    direita      = 'd';
    centro       = 'c';

  PORT
    MCLAS1      : IN (t_info)[30];
    MCLAS2      : OUT (char);
    MCLAS3      : OUT (t2_est);
    MCLAS5      : IN (signaltype REPLY classe);
    MCLAS6      : IN (signaltype REPLY char);

  TASK classificar <1024>,4;
    VAR
      info                : t_info;
      estado,tipo,validade1,
      posicao1             : char;
      L                   : real;
      classificacao       : classe;
      sinal               : signaltype;
      tipoA,tipoB,tipoC,
      rejeitadas         : integer;
      contador            : t2_est;

    PORT
      TCLAS1      : IN (t_info)[30];
      TCLAS2      : OUT (char);
      TCLAS3      : OUT (t2_est);
      TCLAS5      : IN (signaltype REPLY classe);
      TCLAS6      : IN (signaltype REPLY char);

    BEGIN
      LOOP
        ( Realiza a classificacao das pecas atraves de contagem das pecas
          rejeitadas e das pecas boas. Envia sinal ao modulo simulacao para
          que este mostre a contagem global de pecas )

      END;
    END;

  LINK
    MCLAS1 TO TCLAS1 ;
    MCLAS2 TO TCLAS2 ;
    MCLAS3 TO TCLAS3 ;
    MCLAS5 TO TCLAS5 ;
    MCLAS6 TO TCLAS6 ;

ENDMODULE.

```

1.3. Configuração do Sistema

A partir da decomposição do sistema em módulos pode-se realizar de forma independente, da implementação dos mesmos, a configuração do sistema. Inicialmente implementou-se uma configuração em uma única estação. O sucesso desta operação permitiu a distribuição dos módulos em duas estações e posteriormente em três estações. A seguir é apresentado a descrição de configuração considerando três estações.

SYSTEM celula;

USE vis; simu; transp; camera; torno; armaze; classi;

CREATE at station [0]:

vis;
simu;
armaze;
camera;
transp;
classi;

at station [1]:

transp;
classi;

at station [2]:

torno;

```
LINK  simu      . mse5  TO  vis      . mvis4;
      simu      . msc2  TO  vis      . mvis1;
      camera    . mcam2 TO  simu      . msc1;
      simu      . mse4  TO  transp   . mtr3;
      simu      . msc3  TO  transp   . mtr2;
      transp    . mtr1  TO  camera    . mcam1;
      transp    . mtr4  TO  torno     . map2;
      torno     . map3  TO  simu      . msrp1;
      torno     . mct2  TO  simu      . msrp2;
      torno     . mtn4  TO  simu      . msto1;
      simu      . msto2 TO  vis      . mvis7;
      simu      . msrp3 TO  vis      . mvis5;
      torno     . mtn5  TO  armaze1   . mtp2;
      armaze1   . mtp1  TO  torno     . mct4;
      armaze1   . mtp3  TO  simu      . msra2;
      simu      . msra1 TO  vis      . mvis8;
      armaze1   . maz2  TO  simu      . msra3;
      armaze1   . mtz3  TO  vis      . mvis9;
      armaze1   . mtz1  TO  simu      . mstr2;
      simu      . mstr1 TO  vis      . mvis10;
      camera    . mcam3 TO  classi   . mclas1;
      classi    . mclas2 TO  vis      . mvis2;
      classi    . mclas3 TO  vis      . mvis3;
      torno     . map1  TO  classi   . mclas5;
      torno     . mtn1  TO  classi   . mclas6;
```

END.

Não existe nenhuma restrição com respeito a alocação de módulos em uma determinada estação, uma vez que todas as estações apresentam os mesmos recursos físicos. A Fig. 3 ilustra a interconexão dos módulos de acordo com a decomposição realizada.

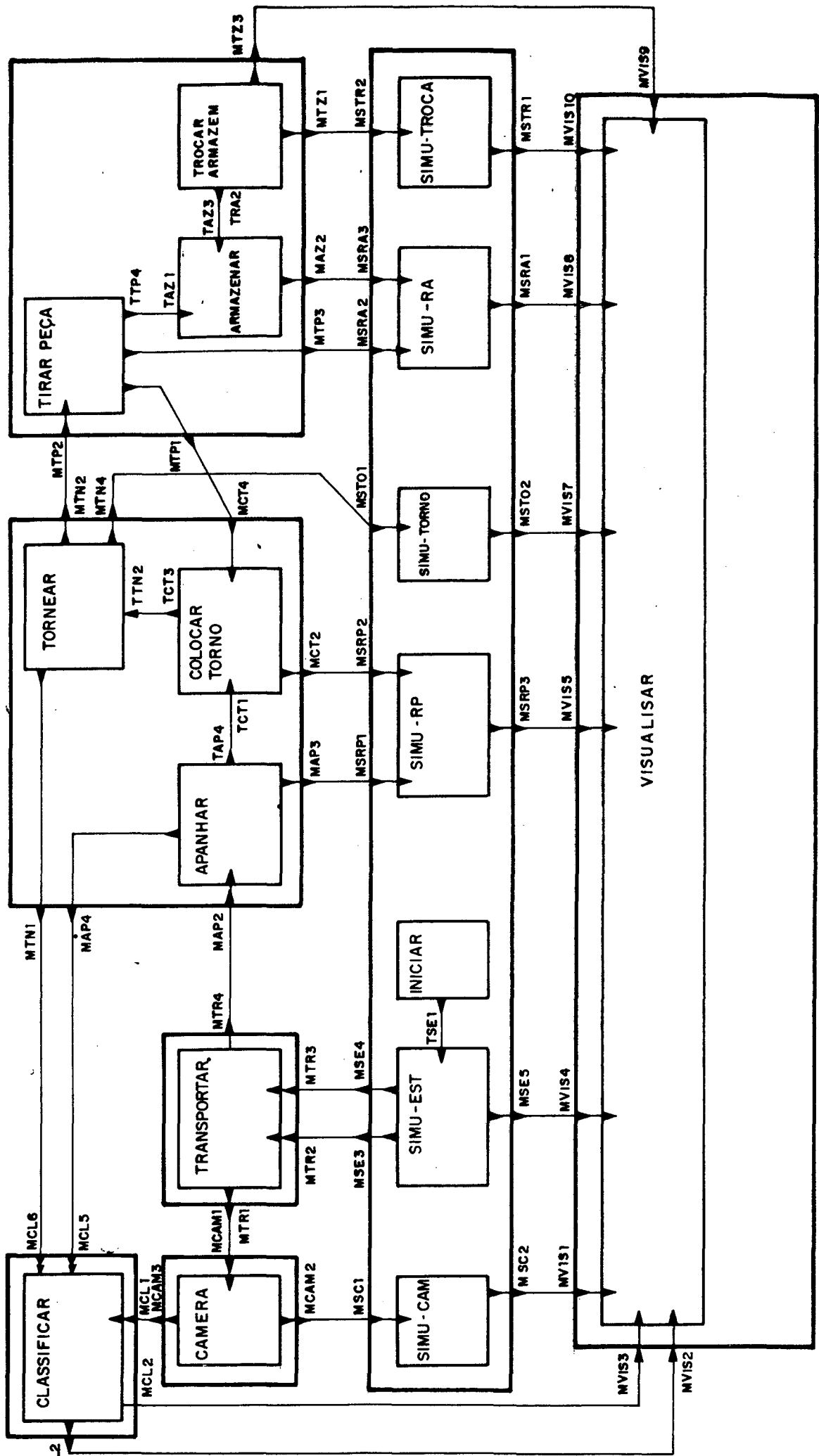


Fig. 3 - Interconexão dos módulos