UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

METODOLOGIA E SUPORTE PARA PROGRAMAÇÃO DE APLICAÇÕES DISTRIBUÍDAS FLEXÍVEIS

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

WESLEY MASTERSON BELO DE ABREU

FLORIANÓPOLIS, 15 MARÇO DE 1991

METODOLOGIA E SUPORTE PARA PROGRAMAÇÃO DE APLICAÇÕES DISTRIBUÍDAS FLEXÍVEIS

WESLEY MASTERSON BELO DE ABREU

ESTA DISSERTAÇÃO FOI JULGADA PARA A OBTENÇÃO DO TÍTULO DE

MESTRE EM ENGENHARIA ELÉTRICA

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

Prof. Joni da Silva Fraga, Dr.
Orientador

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

BANCA EXAMINADORA:

Prof. Joni da Silva Fraga, Dr.
Orientador

Prof. Jean-Marie Farines, Dr. Ing.

Prof. Janette Cardoso , Dra.

RESUMO

O objetivo deste trabalho é propor métodos e ferramentas que facilitem programações flexíveis de sistemas distribuídos. Portanto, são abordados os vários aspectos relacionados com a programação destes sistemas, discutindo as necessidades de suporte e soluções para a configuração dinâmica dos mesmos. Neste sentido, são discutidas os aspectos relacionados com o desenvolvimento de um suporte de configuração dinâmica e os mecanismos, disponíveis a partir da Linguagem de Implementação de Sistemas (LIS), que auxiliam nas mudanças dinâmicas em programas distribuídos. A LIS faz parte de um ambiente para o desenvolvimento e execução de software distribuído de tempo real (ADES), cujo protótipo foi desenvolvido no LCMI-UFSC.

Foi preocupação neste trabalho propor também uma metodologia de programação que facilitasse a construção de aplicações distribuídas, e que evoluísse dinamicamente segundo os mecanismos propostos. A utilização desta metodologia está baseada no conceito de "zonas de dependência", que determinam de forma precisa as implicações de mudanças dinâmicas sobre os componentes de software de um sistema distribuído.

ABSTRACT

The objective of this work is to propose mechanisms and a methodology to facilitate the programming of flexible systens. The aspects related to the development of the support for the dynamic configuration and the solution to deal with changes in the distributed system are discussed.

These mechanisms were incorporated in a System Implementation Language (LIS). The LIS is part of ADES, an environment for development and execution of distributed software for real time systems, which has been developed at LCMI-UFSC.

The proposed methodology is based on the concept of dependency zones to determine the implications of dynamic changes on the software components in a distributed system.

À Rubia

Agradecimentos

Aos profs. do LCMI pela amizade e cooperação. Em particular ao prof. Joni da Silva Fraga, pela orientação e a dedicação durante a realização deste trabalho.

A minha mãe Maria das Vitórias Abreu, a meus irmãos que tanto souberam compreender e ajudar nas horas mais difíceis.

Aos membros da banca pelas críticas e sujestões a este trabalho.

A CAPES pela ajuda financeira.

SUMÁRIO

CAPÍTULO 1
INTRODUÇÃO

A proliferação do uso dos microcomputadores associada à evolução tecnológica no campo da comunicações tem feito com que redes de computadores e sistemas de computadores interconectados sejam uma solução com custo viável, abrindo as portas para novas e mais exigentes aplicações.

Na literatura são apresentados várias definições sobre o que é um sistema distribuído. Contudo, entre estas diversas definições, existe somente um ponto em comum: todas elas exigem a presença de múltiplos processadores. Em [Sloman, 87] é apresentada uma definição mais geral: "Um sistema de processamento distribuído é um conjunto de processadores autônomos e dispositivos de armazenamento que suportam processos e/ou bases de dados atuando em uma ordem para cooperarem e atingirem um objetivo comum. Os processos coordenam suas atividades pela troca de informações, usando uma rede de comunicação". Este trabalho se limita então a este tipo de sistema, normalmente citados como fracamente acoplados, por não apresentarem memória compartilhada.

As vantagens esperadas na aplicação de sistemas distribuídos não podem ser eficientemente exploradas se não houver ferramentas apropriadas e um gerenciamento integrando recursos distribuídos. E um dos maiores desafios é, através destas ferramentas e suportes integradores, construir sistemas flexíveis onde mudanças de configuração não impliquem na parada total do processamento referente a uma aplicação distribuída.

A introdução de mecanismos que tornam flexível um sistema distribuído, especialmente aqueles com aplicações para Controle e Automação, é direcionada por três motivos básicos. Em primeiro lugar, a própria aplicação pode por exigências funcionais, determinar que o sistema se altere dinamicamente (ex.centrais telefônicas). Num segundo ponto, a evolução tecnológica, o crescimento ou alterações da planta controlada (devido a expansão e/ou a manutenção desta),

fazem com que haja necessidades de reconfiguração do sistema de uma forma segura e rápida, preferencialmente com o sistema em curso de operação. Finalmente, a flexibilidade se impõe em sistemas confiáveis onde a presença de elementos faltosos não represente a ruptura de funções no sistema. Neste caso, as propriedades de flexibilidade são importantes para a definição e manuseio de redundâncias quando da implementação de técnicas de Tolerância a Faltas.

Embora em alguns sistemas operacionais sejam fornecidos recursos para configuração dinâmica, somente a abordagem de construção de sistemas distribuídos a partir de linguagens tem mostrado simplicidade e facilidade para a construção de softwares distribuídos flexíveis. Linguagem de programação distribuída, onde a separação dos aspectos de configuração (estruturais) daqueles referentes à programação dos componentes de uma aplicação, para fins de configuração dinâmica, tem sido consenso geral das pesquisas realizadas neste domínio [Kramer, 85].

Neste trabalho são discutidos os aspectos relacionados ao desenvolvimento de um suporte de configuração dinâmica e a introdução de mecanismos na Linguagem de Implementação de Sistemas (LIS) que auxiliem durante a configuração dinâmica em sistemas distribuídos. Também é proposta uma metodologia para o auxílio ao desenvolvimento de softwares distribuídos evolutivos. A metodologia está baseada na análise das possíveis dependências entre os componentes de software de um sistema distribuído.

A LIS faz parte de um ambiente para o desenvolvimento e execução de software distribuído de tempo real (ADES), cujo protótipo, concebido no LCMI-UFSC, é resultado de um trabalho de cinco anos.

Esta dissertação está dividida em seis capítulos. O capítulo 2 apresenta um estudo sobre as características principais da programação distribuída relacionadas com a programação em larga escala. O capítulo 3 apresenta os principais aspectos de um ambiente de desenvolvimento e execução de software distribuído (ADES), com o objetivo de introduzir o paradigma adotado para programação de sistemas distribuídos e o suporte para o gerenciamento da configuração dinâmica.

O capítulo 4 apresenta os aspectos mais relevantes da implementação do suporte da configuração dinâmica. O capítulo 5 apresenta uma metodologia de

concepção de programas distribuídos que oriente o projetista na construção de um sistema distribuído que suporte mudanças dinâmicas. A utilização desta metodologia é mostrada através de um exemplo. Neste capítulo são ainda avaliados os resultados obtidos durante este trabalho.

Finalmente as conclusões e perspectivas são apresentados no capítulo 6.

CAPÍTULO 2
PROGRAMAÇÃO EM SISTEMAS DISTRIBUÍDOS

2.1. Introdução

A complexidade na construção de programas distribuídos determina a necessidade de modelos de programação adaptados às características dos chamados Sistemas Distribuídos fracamente acoplados. Os modelos a princípio são propostos no sentido de atender requisitos, normalmente esperados nestes sistemas, como desempenho, flexibilidade, alta disponibilidade e grande confiabilidade. Ferramentas e suportes, integrados, incorporam estes modelos, permitindo uma programação distribuída estruturada segundo conceitos dos modelos, visando o compartilhamento de recursos e/ou de informações.

A característica de modularidade tem sido largamente difundida na construção de programas distribuídos. É notório a contribuição deste conceito na produção e na evolução de softwares complexos.

Neste capítulo é apresentado um estudo sobre as características principais da programação distribuída. O princípio de Decomposição Modular [DeRemer,76] é apresentado como base para a construção de sistemas distribuídos flexíveis. Neste sentido, as principais atividades envolvidas com a programação em larga escala são objeto de exame, visando a programação de sistemas evolutivos.

2.2. As Aplicações Distribuídas e os Meios de Programação

Aplicações distribuídas apresentam diferentes graus de descentralização. Algumas são estruturadas como uma coleção de serviços especializados, portanto mais eficientes. Cada serviço usa seu processador de forma dedicada. Existem aplicações que favorecem uma descentralização maior de suas funções. Neste caso, uma coleção de estações pode ser vista como um sistema computacional distribuído, onde as funções da aplicação se executam sobre o hardware distribuído.

As aplicações distribuídas apresentam granularidades diferentes de paralelismo. A granularidade é dada pela quantidade de processamento entre comunicações [Bal,89]. Os sistemas distribuídos fracamente acoplados, pelas características de baixa velocidade e de atrasos aleatórios de seus suportes de comunicação, se tornam ambientes adequados a largas granularidades de paralelismo, onde o programa distribuído despende grande parte de seu tempo em processamentos e eventualmente com a comunicação.

A construção de aplicações distribuídas em uma rede de computadores pode se dar sob duas abordagens diferentes. Na primeira, utilizando-se um Sistema Operacional de Rede (Networked Operating Systems, NOS), aplicações distribuídas, constituídas de uma coleção de programas seqüenciais, têm a comunicação remota implementada através de primitivas do NOS. Estas interfaces de comunicação remota são complexas, se diferenciando daquelas suportadas pelos executivos para a comunicação local. A disponibilidade de serviços de suporte para a configuração se restringe à construção inicial da aplicação, o quê dificulta o controle e a evolução da configuração. Estes aspectos tornam as aplicações distribuídas pouco flexíveis.

A outra abordagem, utilizada na construção de programas distribuídos é a partir de uma linguagem. Neste caso, é muito importante a incorporação de conceitos de modularidade, concorrência, comunicação e sincronismo numa mesma estrutura de linguagem. A verificação da compatibilidade de tipos e outros testes em tempo de compilação, neste caso, ficam disponíveis também aos aspectos ligados à configuração do sistema e à comunicação remota. Outro ponto importante é o fornecimento de uma visão homogênea e consistente em termos de

comunicação, sincronismo e identificação tanto para efeito local como remoto. A abordagem por linguagem torna mais segura e simples as aplicações construídas.

Numa abordagem por 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. No próximo item são levantados aspectos conceituais sobre a modularidade.

2.3. Princípio de Decomposição Modular

O princípio de Decomposição Modular ("Programming-in-the-Small vs Programming-in-the-Large") apresentado em [DeRemer,76] é uma técnica clássica para o projeto de softwares complexos. Este princípio prevê a construção de um sistema em dois níveis:

- **Programação em Pequena Escala**: neste nível são construídos os componentes do sistema (módulos), devendo portanto, tratar com aspectos algorítmicos, estrutura de dados, etc.

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

Normalmente, na literatura, é chamado de configuração um arranjo de componentes (módulos). No contexto da programação distribuída, o princípio de decomposição modular permite a configuração do software através de módulos distribuídos nos diversos elementos de processamento (hardware) interconectados por uma rede de comunicação de dados. Neste texto os termos configuração, sistema e programa distribuído serão usados sem distinção.

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 suas 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" introduzida em [Parnas,72]). As restrições de acesso são determinadas pelas relações de utilização definidas pela interface (relações de importação, exportação e inclusão).

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 ou ainda no próprio sistema.

As interfaces e os corpos podem se apresentar textualmente separados quando da programação em pequena escala. As interfaces, no caso, são compiláveis em tabelas símbolos que são consultadas quando da ligação dos módulos; isto implica que a programação em larga escala só necessita de informações sobre a interface do módulo.

2.3.1. Programação em Pequena Escala

Para a codificação do módulo é necessário uma **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 é que esta linguagem permita a definição de tipos módulos, facilitando então a reutilização de software, através de instanciações.

A linguagem de implementação de módulos pode especificar 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 à programação em larga escala. Esta característica torna a evolução de sistemas bem menos complexa.

2.3.2. Programação em Larga Escala

A programação em larga escala trata com os aspectos da expressão da composição e da construção propriamente dita do sistema. A expressão clássica da composição corresponde à uma lista de comandos dirigida ao editor de ligações, especificando os componentes a serem ligados. A evolução destas listas fez com que surgissem as **Linguagens de Interconexão de Módulos** (ou linguagens de configuração).

A linguagem de interconexão de módulos deve fornecer mecanismos para descrever formalmente as operações de composição 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 [Narayanaswamy,87]. Um subsistema é realizado pela composição de seus nós sucessores, usando regras ou operações de construção (carregar, ligar e por vezes também compilar componentes). As informações manipuladas por estas linguagens são resultantes da compilação das interfaces. Os testes executados durante a tradução da descrição de um sistema são feitos com base nestas informações de interface. Em sistemas distribuídos, para a composição de sistemas é necessário que se leve em conta três aspectos importantes:

- **A Estrutura Lógica do Sistema**: envolve a definição de componentes de software do sistema e suas interconexões;

- **A Estrutura Física do Sistema**: descreve os recursos físicos disponíveis no sistema; e

- **Mapeamento Lógico-Físico**: definição de como se dá o mapeamento das estruturas lógicas nos recursos físicos do sistema.

A primeira Linguagem de Interconexão de Módulos MIL-75, introduzida em [DeRemer,76], tratava apenas com a estrutura lógica. Para a aplicação em sistemas distribuídos, uma linguagem de interconexão de módulos deve evoluir para uma ferramenta básica que permita então a descrição do mapeamento lógico-físico. Estas linguagens no contexto dos sistemas distribuídos são denominadas **linguagens de configuração**. As expressões de composição de sistemas, realizadas a partir destas linguagens, serão chamadas neste texto de **especificações de configurações.**

A presença de construções nestas linguagens que permitam **especificações de subsistemas** (subconfigurações) e a **definição de componentes** (módulos ou subsistemas) a partir de **tipos** são sempre importantes pois favorecem a reutilização de software. As linguagens de configuração serão retomadas, no contexto deste trabalho, no capítulo 3.

Um objetivo sempre perseguido, em se tratando de softwares complexos, é a evolução de sistemas. Esta evolução não representa simplesmente uma evolução do código. O problema de suportar a evolução é visto como o controle das dependências entre módulos (recursos requeridos ou fornecidos) e da proliferação de versões resultantes da evolução. Neste sentido, a programação em larga escala se apresenta relacionada com atividades de diferentes fases do ciclo de vida do software. Estas atividades se apresentam, normalmente, concentradas no gerenciamento de configuração.

2.4. Gerenciamento de Configuração e Mudanças Dinâmicas no Sistema

Gerenciamento de Configuração

O gerenciamento de configuração está, portanto, relacionado com as atividades de programação em larga escala que tratam da implementação e manutenção da configuração do Sistema Distribuído. Neste contexto, pode se identificar alguns objetivos do gerenciamento de configuração:

- **Identificar a Configuração**: fornecer atributos que facilitem a identificação dos componentes do sistema;

- **Controlar a Configuração**: fornecer mecanismos para controlar a evolução do software e definir estratégias de mudanças; e

- **Manter a Integridade do Sistema**: garantir que os recursos requeridos por um módulo sejam fornecidos.

O gerenciamento da configuração, tratando com os efeitos da evolução, deve garantir a consistência entre a **especificação de configuração** (expressada, por exemplo, por meio de uma linguagem de configuração) e o estado de configuração atual do sistema. Para tanto, o gerenciamento deve manter registros de todos os recursos e as atividades no sentido da evolução do sistema. Neste sentido, o gerenciamento de configurações é visto na literatura como uma disciplina [Ramammoorthy,86], [Jino,89], cujo objetivo principal é gerenciar versões distintas do sistema que possam vir a ser produzidas a partir de componentes (módulos e subsistemas).

O gerenciamento, no contexto deste trabalho, envolve um conjunto de mecanismos e estratégias que permitem a evolução de um sistema, de uma configuração para outra, dinamicamente. Para uma melhor compreensão do problema é necessário que se faça a distinção entre configurações estática e dinâmica [kramer,85]:

> **Configuração Estática**: consiste na construção inicial do sistema segundo especificações de configuração. Em sistemas (ou linguagens) que só comportam mecanismos para configuração estática, qualquer modificação no sentido da evolução, implica que o sistema deve passar por uma nova construção.
>
> **Configuração dinâmica**: é o processo que permite introduzir mudanças em um sistema com este em operação, fazendo-o portanto assumir uma nova configuração sem passar por um processo de construção total do sistema.

O gerenciamento de configuração que junto com os gerenciamentos de faltas, de segurança ("security") e desempenho constitui o gerenciamento de sistema distribuído, é implementado em funções do sistema operacional distribuído ou em suportes de tempo de execução de linguagens distribuídas.

Controle de Mudanças Dinâmicas

Um dos problemas fundamentais da configuração dinâmica é a manutenção da consistência do sistema durante e após o processo de mudança.

Considerando que os módulos interagem no sistema através de **transações**, ou seja, aos pares de pedidos/respostas, o processamento distribuído pode ser expresso pelo sistema percorrendo estados, segundo a execução de transações. A partir deste modelo, são estabelecidos em [Kramer,88] alguns pontos necessários para que um sistema se mantenha consistente durante o processo de mudança dinâmica:

- as mudanças devem ser especificadas através de uma linguagem declarativa (linguagem de configuração). O suporte de configuração fica com a responsabilidade de estabelecer o momento adequado da execução das operações da mudança especificada;

- as especificações de mudanças de configuração devem ser independentes dos algoritmos e protocolos da aplicação. É básico, neste sentido, a separação total entre a programação em pequena escala e a programação em larga escala;

- as mudanças de configuração devem deixar o sistema num estado consistente de modo que este possa, após continuar normalmente o processamento. Isto permite evitar perdas de informação e portanto inconsistências. Operações de configuração só podem ser iniciadas sobre módulos que não tenham relação com transações em andamento; e

- o gerenciamento de configuração deve a partir da especificação de mudanças, determinar o conjunto de módulos afetados pelas mudanças. O restante do sistema deve continuar o processamento normalmente.

Em particular, o suporte de gerenciamento deve ser capaz de determinar o módulo ou conjunto de módulos que serão afetados por uma operação de configuração. Este mesmo suporte deve esperar que a aplicação atinja uma situação de processamento favorável, de modo que o sistema permaneça consistente durante a configuração dinâmica e após o término desta.

Neste trabalho, conclui-se pela necessidade de uma interface entre o gerenciamento e a aplicação de modo a possibilitar a detecção do momento mais apropriado da execução das modificações de configuração.

O momento apropriado para a execução de uma operação de configuração sobre um determinado módulo, envolve a espera da conclusão de transações iniciadas por este módulo e a não existência de processamento também neste módulo devido a transações iniciadas por outros módulos.

A **quiescência** de um módulo satisfaz estas exigências [Kramer,88]. Um módulo em quiescência não é iniciador e nem receptor de pedidos de processamento, portanto, pode ser submetido a operações de mudanças, não produzindo inconsistências nos outros módulos, mantendo o funcionamento correto do sistema.

A interface citada deve ser o mecanismo para que se atinja a quiescência de um módulo. Esta interface deve ser genérica o que a faz independente da aplicação. Os mecanismos de quiescência de um módulo serão retomados no capítulo 3, no sentido do trabalho desenvolvido.

2.5. Linguagens e Sistemas de Programação Distribuída

Muitas linguagens e ambientes de programação tem sido propostos, algumas já disponíveis comercialmente, com características voltados para a programação em larga escala em Sistemas Distribuídos. Dentre estas podemos citar os ambientes MESA [Sweet,85], PRONET [Maccabe,82], Mascot [Simpson,79], Argus [Liskov,83], etc. Embora apresentem linguagens de configuração ou meios de expressão da configuração estas propostas não fazem uma separação total da construção dos componentes (módulos) daquela referente à configuração do sistema. Neste aspecto, ficam limitadas as características de flexibilidade do sistema construído. Qualquer modificação (introduzindo ou retirando componentes) pode envolver recompilação de módulos e certamente resulta na reconstrução total do sistema distribuído (configuração estática).

Dois sistemas de programação nos interessam em particular: o ambiente CONIC [Magee,87] e a linguagem DURRA [Weinstock,89]. Estes ambientes são apresentados a seguir, no sentido de ressaltar as características que propiciam a construção de sistemas flexíveis e também a influência que tiveram no nosso trabalho.

2.5.1. CONIC

O ambiente CONIC fornece uma abordagem baseada em linguagem para a construção de sistemas distribuídos. Este fornece um conjunto de ferramentas para compilação, configuração, depuração e execução de programas distribuídos. A programação distribuída em CONIC está fundamentada no princípio de Decomposição Modular. Neste sentido, a linguagem trata com os aspectos ligados à construção de componentes (módulos). As interfaces destes módulos são definidas como portos (objetos locais).

Uma linguagem de configuração é usada para especificar a combinação de módulos formando o tipo **nó lógico.** Estes tipos como unidades de configuração, são instanciados nos nós físicos do sistema e interconectados, formando uma aplicação distribuída. Os nós lógicos além dos módulos apresentam um suporte para concorrência; o módulo é a unidade de concorrência.

Um sistema distribuído programado em CONIC pode sofrer mudanças incrementais, segundo especificações de configuração (descritas pela linguagem de configuração) implementados por um gerenciador de configuração. A consistência do sistema durante o processo de mudança é preservada através de um protocolo entre o gerenciamento de configuração e o nó lógico envolvido em operação de configuração.

Um grau de flexibilidade considerável é portanto obtido no sistema CONIC através do suporte de gerenciamento de configuração dinâmica. Porém existem limitações, num processo de mudança, só podem ser criadas instâncias de tipos nós lógicos já presentes na configuração inicial. A impossibilidade de instanciar dinamicamente módulos dentro de um nó lógico também diminuem as características de flexibilidade.

2.5.2. Linguagem DURRA

A linguagem Durra [Weinstock,89], foi projetada para suportar o desenvolvimento de programas distribuídos sobre máquinas heterogeneas. Esta abordagem de programação é uma tendência nova, notada na literatura como

programação por múltiplas linguagens, onde as unidades de concorrência, chamadas **tarefas**, são programadas em diferentes linguagens e se executam sobre uma variedade de sistemas operacionais.

A programação de uma aplicação se desenvolve em três partes: a primeira fase envolve a criação de uma biblioteca de tarefas e interfaces escritas em linguagem (C, ADA, Pascal, etc..) para as quais foram definidas interface DURRA. A segunda fase descreve o processo de configuração, que seleciona o conjunto de tarefas na biblioteca para comporem o sistema. A configuração descrita em DURRA, quando compilada, gera um conjunto de comandos de alocação de recursos e escalonamentos. Durante a última fase um executivo carrega as implementações de tarefas nos processadores de acordo com os comandos.

O ambiente de execução é composto pelas tarefas da aplicação e o executivo Durra. O executivo é responsável pela iniciação das tarefas e a manipulação das comunicações existentes entre as tarefas. Do ponto de vista das tarefas a comunicação é realizada através de primitivas, que são, por sua vez, determinadas por um conjunto de primitivas suportados pelo executivo. Pode-se citar o envio e o recebimento de mensagens e a sinalização para o inicio de reconfiguração da tarefa.

A declaração de reconfiguração pertence à parte da estrutura de descrição de uma tarefa. Esta descrição especifica as possíveis mudanças na estrutura da aplicação a serem executadas, em decorrência de algum evento previsto nesta descrição. Após a ocorrência do evento o executivo executa esta tarefa, provocando as mudanças.

Na linguagem DURRA, a preservação da consistência do sistema é dita similar a do sistema CONIC. Ou seja, é provocado a quiescência de tarefas afetadas pelo processo de mudanças. A tarefa sinaliza ao suporte através de uma primitiva **SAFE** que está pronta para a reconfiguração (**quiescente**).

A principal limitação dos mecanismos apresentados pela linguagem Durra está relacionada à pouca flexibilidade em modificações não previstas, devido ao fato que as especificações de mudança são descritas em tarefas quando da construção inicial. Qualquer configuração não prevista envolve a reconstrução total do sistema.

2.6. Conclusão

O princípio de decomposição modular ou ainda a programação modular (como metodologia em fase de implementação) se adapta perfeitamente a metodologias de especificação e projeto que seguem tanto a **orientação por processos** como os que seguem a **orientação por dados**. As metodologias **orientados por processos** compreendem as técnicas descendentes (decomposição "Top-down"). As metodologias orientadas a dados se utilizam do processo inverso, a composição ("Bottom-up").

O estudo bibliográfico e conceitual apresentado neste capítulo mostra a necessidade de ferramentas que simplifiquem a difícil tarefa de construir programas distribuídos. As características de flexibilidade são altamente desejáveis para aplicações distribuídas. A evolução destes sistemas deve ser facilitada, considerando o alto grau de complexidade envolvido nestes sistema.

CAPÍTULO 3
CONFIGURAÇÃO DINÂMICA DE SISTEMAS DISTRIBUÍDOS

3.1. Introdução

Este capítulo apresenta os principais aspectos do Ambiente de Desenvolvimento e Execução de Software - ADES, realizado com o intuito de conduzir o desenvolvimento de software em aplicações distribuídas em tempo real segundo uma metodologia baseada no princípio de decomposição modular [DeRemer,76]. Nesta metodologia, as atividades de programação em larga escala são separadas daquelas em pequena escala, proporcionando além das vantagens tradicionais da programação modular, as propriedades de configuração dinâmica do sistema.

Segundo a separação citada acima, a programação distribuída neste ambiente é feita através da Linguagem de Implementação de Sistemas (LIS) que encapsula todos os aspectos da arquitetura do software distribuído. Os aspectos de configuração são descritos por uma sub-linguagem LINCS (Linguagem de Configuração de Sistemas), que permite ao programador do sistema ativar o processo de configuração desejado. A linguagem LIS e o suporte de configuração do ambiente ADES, principalmente no referente à configuração dinâmica são tratados neste capítulo.

3.2. Descrição do Ambiente ADES

3.2.1. Objetivos do ADES

O objetivo primeiro do ADES é permitir uma ampla integração de um conjunto de ferramentas que atuam em diferentes fases do ciclo de vida do software, de modo a proporcionar:

- uma metodologia de concepção de software distribuído, baseada no princípio de decomposição modular, partindo da descrição formal em Redes de Petri.

- a validação das especificações formais através da análise e da simulação de maneira a obter um software, sempre que possível, livre de erros;

- A translação de forma semi-automática para uma Linguagem de Implementação de Sistemas (LIS);

- a execução do software de modo a cumprir com seus requisitos de tempo real (inicialmente com características de "soft real-time");

- a facilidade de configuração dinâmica do sistema de modo a permitir a manutenção e a evolução do sistema em curso de operação, bem como possibilitar a implementação de técnicas de Tolerância a Faltas;

- o teste de software a nível de componentes e do sistema global.

Na figura 3.1, é indicada a disposição de cada ferramenta envolvida nas diversas fases de desenvolvimento do software.

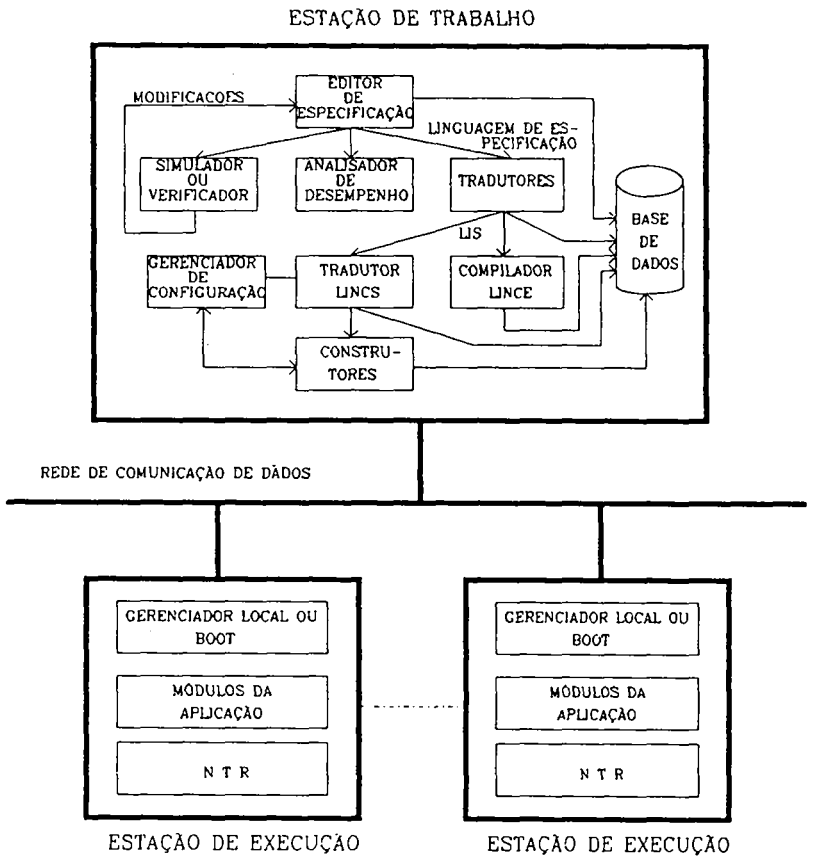

Figura 3.1 Representação Esquemática do Ambiente ADES

3.2.2. Organização do Sistema.

Este ambiente, na sua versão atual, foi construído para um conjunto de estações do tipo PC (figura 3.1). Uma destas estações concentra as ferramentas de desenvolvimento, assumindo o papel de Estação de Trabalho, enquanto as outras são Estações de Execução nas quais o software desenvolvido será carregado e executado.

Cada estação possui como software de base uma versão do Sistema Operacional Distribuído (SOD) que engloba funções de gerenciamento e alocação

de recursos de sistema distribuído. O SOD inclui também um suporte de tempo de execução que é constituído basicamente por um Núcleo de Tempo Real (NTR).

A organização da Estação de Trabalho é fortemente influenciada pelas atividades de desenvolvimento, carga e manutenção do software e deve conter ainda um sistema operacional hospedeiro fornecendo acesso à serviços básicos (operações com arquivos, compilação, edição, etc.).

O SOD é formado por módulos gerenciadores e por um módulo interfaceador com o sistema hospedeiro na Estação de Trabalho. Os gerenciadores são responsáveis pelas operações com módulos, ligação de portos, tratamento de erros, carregamento remoto etc. O módulo interfaceador por sua vez torna seqüencial as chamadas ao sistema hospedeiro. O SOD no caso das Estações de Execução incorpora uma configuração mais simples, cujas funções se restringem a operações de controle de instâncias de módulos, controle de erro e de acesso remoto às memórias.

O Núcleo Tempo Real é o responsável pela implementação do ambiente multi-tarefa distribuído que executa as aplicações distribuídas e que fornece serviços básicos para o gerenciamento da configuração do sistema.

3.3. Paradigma de Programação e a sua Implementação

3.3.1. O Paradigma Escolhido

O paradigma de programação adotado segue o princípio de decomposição modular, permitindo a separação total das atividades de programação em pequena escala da programação em larga escala. Um programa distribuído é visto como um conjunto de módulos interconectados entre si, durante a configuração do sistema.

Um módulo tem a sua visibilidade unicamente realizada através de portos, que são interfaces com características de recursos locais. Portos de saída são conectados a portos de entrada, formando canais de comunicação e sincronização.

Os canais de comunicação são definidos externamente aos módulos. Deste modo a configuração dos canais dentro do sistema é completamente independente da programação interna dos módulos. Não existe referências diretas a outros componentes (ou a portos pertencentes a outros módulos), proporcionando desta maneira facilidades para a reutilização do software, abstrações de implementações e modificações dinâmicas no sistema.

O módulo encapsula uma ou mais tarefas que por sua vez constituem as unidades básicas de concorrência. Da mesma maneira que módulos, tarefas se comunicam por troca de mensagens através de portos locais. Conceitualmente, as tarefas pertencem a um mesmo módulo quando apresentam uma forte coesão entre si. A noção de **módulo** é restrita à fase de desenvolvimento e de configuração do sistema, sendo substituída em tempo de execução pela noção de **tarefas**. A figura 3.2 representa o paradigma escolhido.

No modelo adotado são previstas comunicações síncronas e assíncronas com diferentes tipos de conexão (um para um, um para muitos e muitos para um).

Este modelo é uma variação do modelo CONIC [Sloman,86], permitindo, com a sua implementação, características de reutilização e de flexibilidade :

- uma configuração pode conter várias instâncias de um mesmo (tipo) módulo;

- a configuração de um sistema pode evoluir segundo alterações propostas através de mecanismos de programação em larga escala; estas mudanças podem ocorrer durante a operação do sistema, caracterizando a configuração dinâmica [Kramer,85].

- um módulo pode ser programado para alterar a sua configuração interna. Tarefas podem modificar em tempo de execução suas ligações e seus estados, provocando a reconfiguração do módulo interno a qual pertencem.

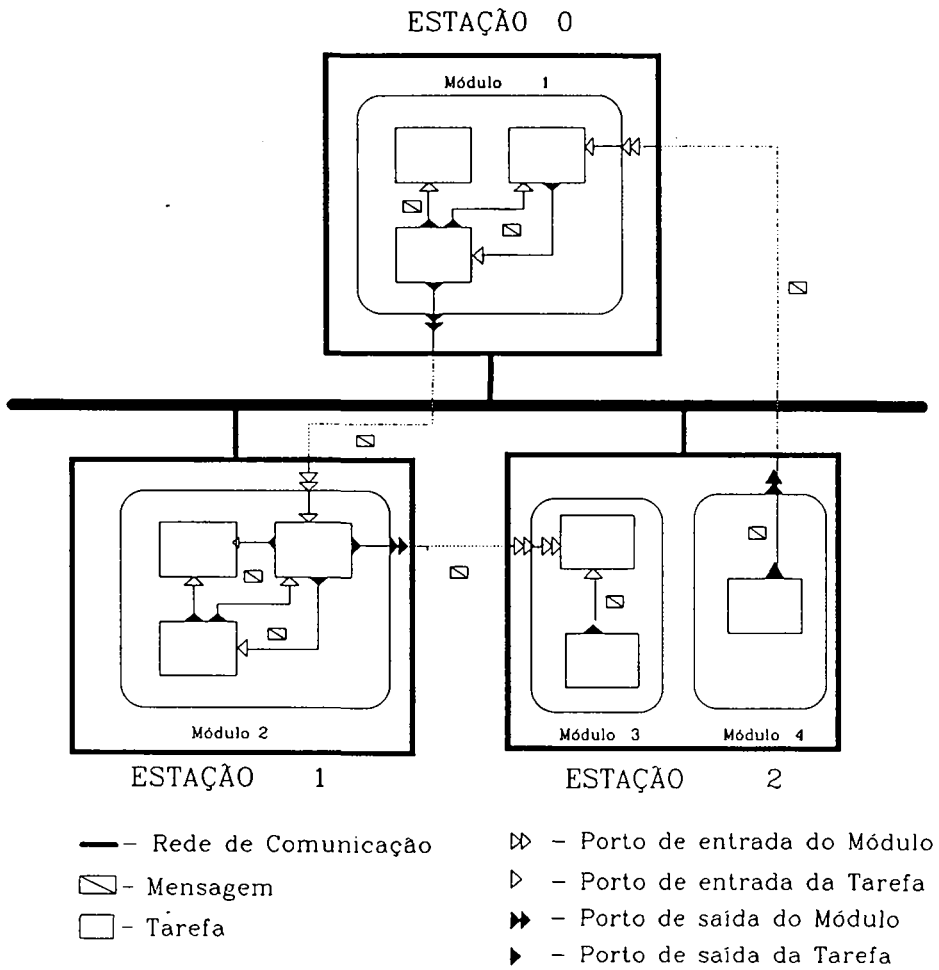

Figura 3.2 Paradigma de programação

3.3.2. LIS: Linguagem de Implementação de Sistemas

As abstrações definidas no modelo de programação são implementadas a partir da linguagem LIS, servindo desta maneira como instrumento ao usuário para estruturar e conceber sistemas distribuídos. A fidelidade ao modelo adotado garante uma boa compreensão e uma alta flexibilidade na estrutura do sistema. Esta fidelidade ao modelo conduz à subdivisão da linguagem LIS em :

- uma (sub)Linguagem de Componentes Elementares (LINCE) que permite a definição do módulo como tipo (tipo módulo).

- uma (sub)Linguagem de Configuração de Sistemas (LINCS) que possibilita a criação de instâncias do tipo módulo, que por sua vez, serão interconectadas e mapeadas em unidades de processamento formando um programa distribuído.

A linguagem LINCE será objeto do próximo item. Os aspectos relacionados com a linguagem LINCS e o suporte de configuração serão aprofundados no decorrer do capítulo.

3.3.3. LINCE: Linguagem de Componentes Elementares

Estrutura geral

O módulo, unidade básica de configuração, é programado através da LINCE. Esta, na sua versão atual, apresenta-se como uma linguagem procedural que contém as construções e declarações da linguagem Pascal, acrescidas de extensões de modo a possibilitar a expressão da estrutura do módulo.

A constituição multitarefas do modelo é definida em tempo de programação. A conexão entre portos de tarefas e a atribuição de portos de tarefas a portos de módulo são realizadas a partir da declaração de ligações internas. Estas ligações podem ser desfeitas a partir de tarefas do próprio módulo em tempo de execução.

A partir de um tipo módulo podem ser criadas várias instâncias que possuem espaços de memória disjuntos (áreas específicas de dados e de stack) quando localizados em uma mesma estação. Cada uma destas instâncias pode ser parametrizada, adaptando-se às condições específicas do seu ambiente de execução.

A independência de referências externas ao módulo faz com que este possa ser implementado, compilado e testado separadamente de outros módulos.

Comunicação

A declaração de porto envolve a definição de tipo: qualquer variável pode ser veiculada através de um porto, desde que apresente o mesmo tipo deste porto. Na declaração também se define o sincronismo associado à comunicação. Uma declaração de portos pode ser ilustrada na forma:

```
PORT
    alfa: IN(real REPLY signal); /* comunicação síncrona */
    beta: OUT(integer);          /* comunicação assíncrona*/
    gama: OUT(real REPLY real);<SS> /* comunicação semi-síncrona */
```

Combinações das primitivas SEND/WAIT/RECEIVE/REPLY na linguagem LINCE permitem a definição de diferentes possibilidades de sincronização nos canais de comunicação.

O compartilhamento de dados não é permitido diretamente no modelo. Entretanto, em módulos multitarefas, a troca de ponteiros entre tarefas é definida, e desta forma pode-se ter um acesso rápido a estruturas tais como listas, tabelas etc.

Características em Tempo Real

A linguagem LINCE apresenta construções importantes para aplicações em tempo real, em particular podemos citar os comandos: o comando "LOOP" temporizado que substitui o comando de retardo de tarefas, normalmente encontrado em sistemas similares; o comando "SELECT" para recepções seletivas dos canais de comunicação; e comandos que permitem acessos as primitivas do núcleo tais como espera de interrupção, relógio de tempo real, e outras. Os mecanismos para o tratamento de exceções são fornecidos para o uso quando de operações de comunicação.

Aspectos da Implementação

A linguagem de programação de componentes é implementada através de um pré-compilador que translada o código fonte em um programa Pascal, substituindo adequadamente os comandos-extensão por chamadas ao Núcleo Tempo Real.

Após a compilação é gerado um arquivo de código relocável, acompanhado de um código de iniciação que permite durante a instanciação do módulo, a criação de tarefas e portos. Em adição, é criado um arquivo de dados com informações sobre as necessidades de memória, os portos (interface) e parâmetros do módulo; este arquivo será utilizado para a realização da configuração do sistema. Maiores detalhes sobre a linguagem LINCE são encontradas em [Silva,88].

3.3.4. Suporte de Tempo de Execução

O Núcleo de Tempo Real [Nacamura,88] é o responsável pela implementação do ambiente multi-tarefas distribuído que oferece os serviços básicos para o gerenciamento do sistema. Os serviços fornecidos pelo Núcleo de Tempo Real são :

- **Suporte de Gerenciamento Local:** fornece um conjunto de primitivas relacionadas com atividades de configuração, envolvendo operações sobre tipos módulos, instâncias e estabelecimento de canais de comunicação. Estas operações são disponíveis a gerenciadores locais que participam da configuração da aplicação.

- **Gerenciamento de Tarefas:** envolve primitivas para o manuseio de tarefas (criar, destruir, suspender, iniciar, parar). O escalonamento das tarefas segue uma política de pré-esvaziamento por prioridades;

- **Comunicação e Sincronismo entre Tarefas (IPC):** o serviço de comunicação entre tarefas fornece um conjunto de primitivas que suportam as diferentes operações de comunicação e sincronização previstas no paradigma de programação. Fazem parte deste conjunto,

indicadores de exceções que permitem ao programador planejar estratégias de recuperação de erros de comunicações.

- **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 desalocação 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-processos) de forma transparente sobre a rede de computadores é feita usando módulos **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) sobre comunicações remotas.

3.4. Configuração no Ambiente ADES

O propósito de um suporte de configuração é definir o meio de expressão e a forma de configuração dos módulos no sistema. 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. Esta linguagem então permite descrever a estrutura global do sistema, identificando os módulos (tipos) que constituem o sistema (definição de contexto), determinando a criação de instâncias de tipos módulos e definindo as interconexões entre estas instâncias.

3.4.1. LINCS: Linguagem de Configuração de Sistemas

A LINCS é uma linguagem declarativa que permite a construção de um texto fonte. Este texto fonte é dado pela declaração SYSTEM, que descreve a estrutura lógica do software distribuído e o mapeamento lógico/físico do sistema. O modelo de estruturação do software suportado pela LINCS permite a representação explicita dos aspectos de distribuição e facilidades para a construção hierárquica do sistema.

As declarações da linguagem LINCS são classificadas em: básica, inversa e auxiliar. 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 de uma especificação de configuração. A declaração PORT é necessária para permitir a abstração de grupos (**GROUP MODULE**), restringindo o seu uso somente a este tipo de declaração de estruturação.

As declarações básicas usadas na descrição de configuração definem o contexto (**USE**), criam instâncias (**CREATE**) e interconectam módulos através de ligações de portos (**LINK**). As declarações inversas realizam funções ligadas a mudanças em configurações, envolvendo o desligamento de portos (**UNLINK**), a destruição de instâncias (**DELETE**) e a remoção de tipos módulo (**REMOVE**).

As declarações de estruturação SYSTEM, GROUP MODULE e CHANGE englobam especificações de configuração em vários níveis, correspondendo respectivamente à especificação inicial do sistema, definição de sub-especificações (ou sub-sistema) e especificação de mudanças. A sintaxe dos comandos LINCS podem ser encontrados no apêndice (A) e em [SOUZA,88].

3.4.2. Aspectos do Suporte para Configuração Estática

Processador LINCS

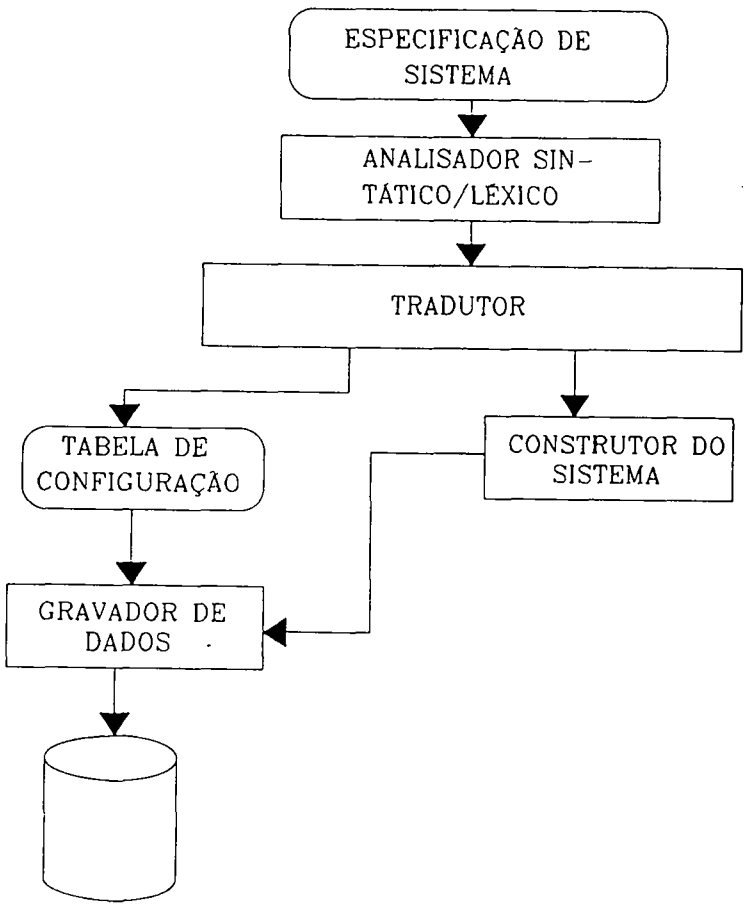

Figura 3.3 Processador LINCS - Configuração Estática

O processador LINCS na configuração estática (figura 3.3) realiza as funções de tradução da declaração SYSTEM e gera tabelas que serão usadas no processo de configuração do sistema. A entrada do processador é um arquivo contendo à especificação de configuração (declaração SYSTEM) escrita em LINCS, que deverá ser analisada quanto a integridade, compatibilidade e viabilidade da realização.

A função da tradução é composta de funções de análise léxica e sintática do texto fonte em LINCS com a sinalização de erros correspondentes.

A tradução envolve a validação da declaração SYSTEM, e é realizada no sentido de verificar : a compatibilidade dos tipos dos parâmetros na instanciação, a possibilidade de interconexão dos portos de saída aos de entrada dos módulos e a disponibilidade de recursos físicos nas estações. O resultado da tradução consiste numa tabela de configuração, contendo os identificadores dos arquivos de código dos tipos módulo para cada estação e numa base de dados representando a configuração especificada.

Processo de Configuração Estática

O Construtor (figura 3.4), a partir da Base de Dados, cria para cada estação do sistema, uma **imagem carga** constituída de tipos módulos, do núcleo de tempo real (NTR) e uma Tabela de Configuração . A configuração das estações é realizada através do carregamento das imagens carga através da rede seguindo um protocolo a duas fases:

- na primeira fase, cada estação de execução recebe a respectiva imagen-carga e interpreta a Tabela de Configuração. Uma vez concluída a configuração da Estação de Execução, esta deve enviar uma mensagem indicando o sucesso da operação à Estação de Trabalho.

- na segunda fase, uma vez de posse das indicações de sucesso da configuração de todas as estações, a Estação de Trabalho envia em difusão uma mensagem de confirmação de configuração às Estações de Execução. Estas retornam mensagens de reconhecimento que uma vez recebidas fazem da Base de Dados a representação do estado atual de configuração do sistema.

O insucesso em qualquer destas fases, implica no abandono dos resultados intermediários. A execução do protocolo a duas fases está a cargo do Construtor na Estação de Trabalho (figura 3.4). Nas Estações de Execução, a parte deste protocolo referente a transferência de arquivos é executada por entidades "boot"

enquanto a sincronização final está a cargo de um procedimento no NTR. Detalhes sobre o processo da tradução e a configuração estática podem ser encontrados em [Souza,88].

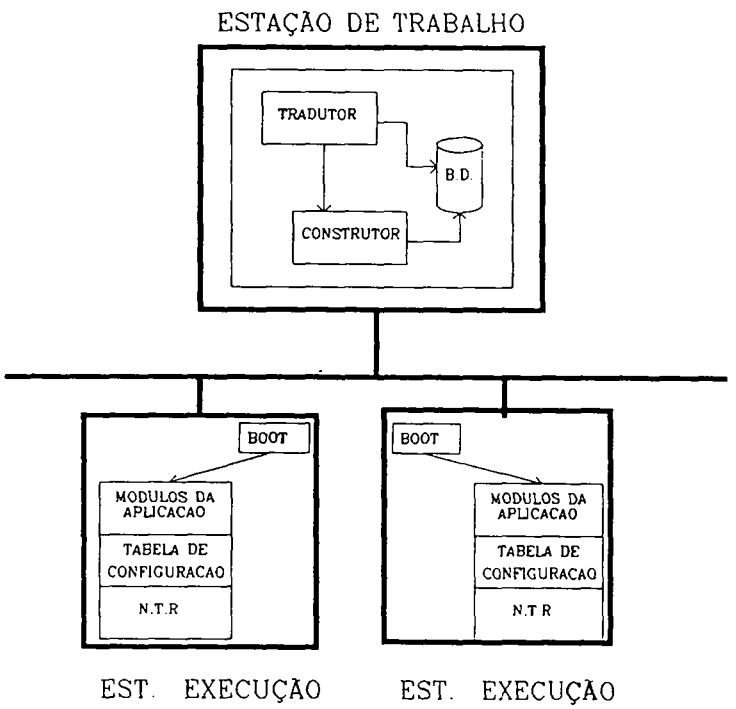

Figura 3.4 Processo de Configuração Estática

3.4.3. Suporte para Configuração Dinâmica

A estrutura modular e a hierarquização da aplicação segundo níveis de abstração são propriedades intrínsecas à linguagem LIS. Em adição, a impossibilidade de referências diretas de um módulo a outro módulo (ou a um grupo) faz com que o sistema possua as propriedades (ítem 3.3.1) que viabilizam a evolução e manutenção do sistema em execução. Isto se dá através de operações de conexão/desconexão de portos, criação/destruição de instâncias de módulos, parada/ativação de instâncias de módulos e carregamento/remoção de tipos módulos em uma estação. Estas operações caracterizam a configuração dinâmica e são suportadas por meio de uma declaração CHANGE.

3.4.3.1. Declaração Change

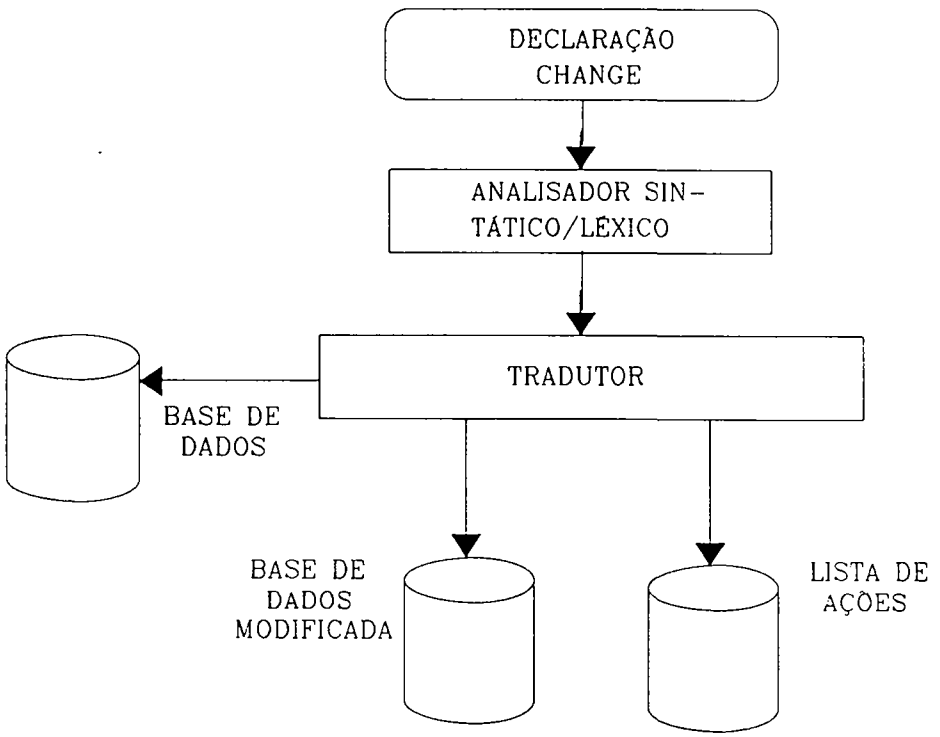

Figura 3.5 Processamento da Declaração CHANGE

Para permitir mudanças dinâmicas, a linguagem LINCS suporta uma declaração de configuração CHANGE (figura 3.5), que apresenta a mesma estrutura de uma declaração SYSTEM, acrescida de **operações inversas.** As operações inversas possibilitam desfazer ligações, destruir instâncias e remover tipos módulos do sistema distribuído. As mudanças são controladas pelo Módulo Gerenciador de Configuração do SOD, a partir destas declarações CHANGE.

3.4.3.2. Suporte de Gerenciamento de Configuração Dinâmica

A Configuração Dinâmica envolve a definição de módulos gerenciadores do sistema operacional distribuído (SOD), que concorrem com módulos da aplicação

nas estações. Estes gerenciadores são responsáveis pelas operações sobre tipos módulos, instâncias, interconexões entre portos e acesso remoto à memória, completando então o suporte necessário para configuração dinâmica (figura 3.6). Estes gerenciadores são divididos em gerenciador de configuração, gerenciadores locais (Gerente de Módulo, Gerente de Ligação, Gerente de Carregamento e Gerente de Acesso Remoto à Memória (GARM)) e gerenciadores auxiliares (Gerente de Arquivo, Gerente de Vídeo e Gerente de Teclado). O detalhamento da implicação destes gerenciadores no processo de configuração é apresentado no próximo capítulo.

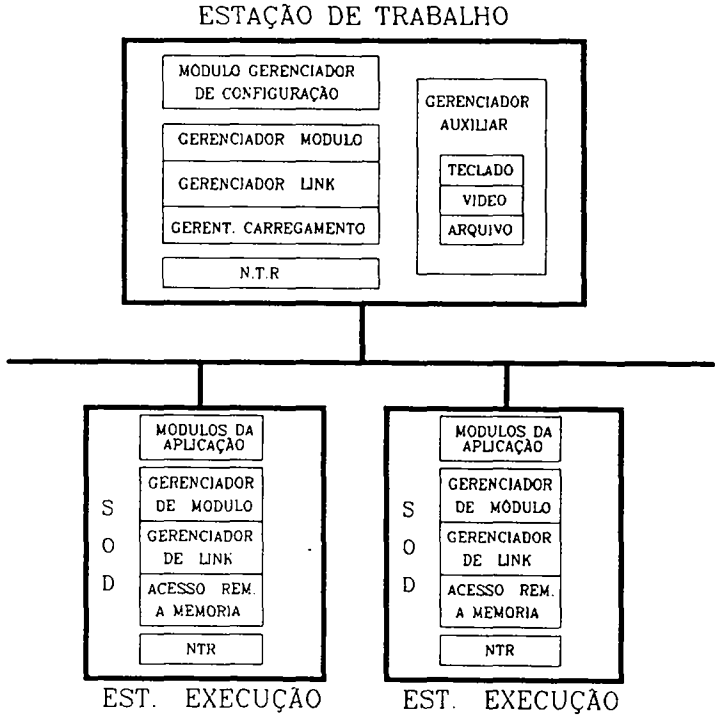

Figura 3.6 Suporte para Configuração Dinâmica

A tradução de uma declaração CHANGE, de modo semelhante ao SYSTEM, gera uma base de dados e uma lista de ações. A base de dados representa uma versão (temporária) do estado de configuração do sistema, incorporando as mudanças especificadas na declaração CHANGE. A lista de ações gerada será

interpretada pelo módulo gerenciador de configuração, localizado na estação de trabalho.

A lista de ações deve traduzir as necessidades de operações para provocar a mudança de configuração do sistema. Estas operações serão executadas a partir de mensagens enviadas pelo módulo gerenciador de configuração que controla as listas de ações, aos módulos gerenciadores locais das estações sujeitos ao processo de mudança.

Se todas as ações forem realizadas com sucesso a representação temporária do sistema (base de dados gerada a partir da declaração CHANGE) será efetivada como a versão atual do estado de configuração do sistema.

Na ocorrência de qualquer exceção que impeça a concretização da mudança, o sistema deve retornar a configuração anterior, mantendo a consistência com a versão atual da representação do estado de configuração (representação temporária é descartada). A execução da lista reversa de ações é o mecanismo necessário que permite este retorno ao estado anterior de configuração. Esta lista é executada a partir da operação de configuração que não se concretizou no sistema, e deve desfazer o que foi executado a partir da lista de ações antes da exceção.

O controle de execução da lista reversa é feito pelo gerenciador de configuração de maneira idêntica a um processo normal de configuração. Na impossibilidade de prosseguir no sentido de recuperar o antigo estado de configuração, devido a outra exceção, o gerenciador de configuração difunde uma mensagem de "parada" aos gerenciadores locais, de modo a que estes suspendam as tarefas dos módulos da aplicação.

3.4.3.3. Estratégia de Configuração

O gerenciamento da configuração dinâmica está diretamente associado ao modo de execução dos comandos da lista de ações.

A execução de uma lista de ações é seqüencial, onde cada operação só é iniciada após a confirmação do sucesso da execução da operação precedente, pelo gerenciador local envolvido. A execução seqüencial pode alongar o período de

mudanças, implicando portanto em possíveis inconsistências na aplicação. Porém a vantagem desta estratégia está na simplicidade do controle do processo de mudanças, e na segurança na recondução do sistema ao estado anterior da configuração, quando da impossibilidade de continuar a execução da declaração CHANGE.

A aplicação CHANGE tendo as operações de configuração descritas segundo a ordem :

LOAD > CREATE > STOP > UNLINK > LINK > START > DELETE > REMOVE

esta ordem tem a finalidade de facilitar a restauração do estado da aplicação, no caso da ocorrência de algum erro durante as operações de mudanças. Seguindo esta ordem na execução das operações de configuração, estaremos preservando os estados das instâncias e os códigos dos tipos módulos da configuração anterior, até o momento em que todas as outras operações definidas no CHANGE tenham sido realizadas com sucesso.

O protocolo entre o gerenciador de configuração e os gerenciadores locais, envolve trocas de pedido/resposta, onde a resposta define uma confirmação positiva ou negativa em relação à operação solicitada. Estas trocas são realizadas em comunicações síncronas (primitivas de IPC da LINCE). Neste caso cláusulas de TIMEOUT e FAILINK (ver item 3.4.4.1) devem tratam da falta de resposta em tempo hábil.

3.4.4. Mecanismos de Manutenção de Consistência

O problema de inconsistência está relacionado com as transações (trocas de pedidos/resposta) existentes entre os módulos que compõem o sistema. Um módulo pode iniciar transações (emissor do pedido) ou pode ser sensibilizado por transações (receptor). Quando um módulo é envolvido nas operações de mudanças, podem se produzir efeitos destas operações em módulos que iniciam transações

e/ou em módulos receptores. Estes efeitos estão diretamente ligados a produção de inconsistências no sistema. A LIS fornece alguns mecanismos, tanto para evitar como para tratar estas inconsistências.

3.4.4.1. Tratadores de Exceção na Comunicação

As inconsistências podem ser reduzidas a partir das comunicações através das cláusulas de esgotamento de tempo e de falha de ligação de portos introduzidas na sub-linguagem LINCE (FAILTIME E FAILINK). As exceções causadas pela falta de resposta a uma transação iniciada pode ser detectada pelo esgotamento de tempo (TIMEOUT) o que permite o tratamento do "não envio" de mensagens. A cláusula de tempo permite a programação de um manuseador (handler) para exceções do tipo esgotamento de tempo em envios síncronos ou em operações de recepção em tarefas. Na segunda cláusula é permitido, na tarefa emissora, um manuseador de exceções de falta de ligações (FAILINK) (em envios síncronos e assíncronos). Estes mecanismos minimizam principalmente os efeitos no módulo iniciador da transação quando o outro participante, o módulo que recebe o pedido da transação, está envolvido no processo de mudanças.

3.4.4.2. Pontos de Sincronismo Estendido

A retirada na configuração de módulos iniciadores de transações, pode gerar processamentos órfãos nos receptores. Para tentar evitar o início de transações quando do envolvimento do emissor em mudanças, foi introduzida a noção de **pontos de sincronismo**. Estes pontos servem como meios de sinalização entre o suporte de configuração (isto é, o gerenciador de módulo que é parte do gerenciamento local) e os módulos da aplicação, de modo que operações de configuração nestes últimos só venham a ser executados entre transações, ou seja, antes de um módulo começar uma nova transação através do envio.

No sistema ADES, inicialmente os pontos de sincronismo são colocados em cada tarefa do módulo considerado, através da primitiva SINC definida na linguagem LINCE. A localização dos pontos de sincronismos é determinada após

um estudo das relações de dependência entre os módulos (ítem 3.4.5), e sua utilização é exemplificada no capítulo 5.

Os pontos de sincronismo, envolvem a suspensão implícita das tarefas de um módulo em processo de configuração. Esta suspensão se dá antes que as tarefas iniciem uma nova transação. Isto contribui para diminuir as possíveis inconsistências no sistema informático, mas não oferece ainda para o engenheiro do processo (ou programador) meios para minimizar os efeitos da parada, por exemplo de determinados módulos de uma planta controlada. Diante disto, o mecanismo ponto de sincronismo foi estendido no sistema ADES de modo a permitir que o programador, considerando a importância de determinados módulos para a aplicação, possa programar um conjunto de operações que devam ser executados pelas tarefas quando em estado de configuração, antes de se auto-suspenderem. A primitiva SINC toma então o seguinte aspecto:

```
decl_sinc ::= SINC
                  instrução [ ";" instrução ]*
              ENDSINC;
instrução ::= inst_ext|inst_pascal
```

O programador considerando as particularidades da aplicação, poderá então se valer deste mecanismo e preparar a aplicação para mudanças eventuais envolvendo determinados módulos. A colocação ou não da primitiva SINC é critério do programador do módulo. A não colocação implica na parada imediata do módulo através do gerenciador local.

O suporte de configuração do ADES foi projetado para admitir a execução da declaração CHANGE com ou sem esta sincronização SINC. A escolha fica a critério do programador de configuração. O processo de configuração sem esta sincronização é sempre utilizado quando da impossibilidade de continuar a execução de uma especificação CHANGE. Neste caso, o retorno ao estado de configuração anterior, é mais rápido que no caso da utilização do ponto de sincronismo.

3.4.5. Condições a serem Consideradas na Elaboração de uma Declaração CHANGE

A análise do estado global do sistema é importante no sentido de definir a estratégia da realização das modificações no mesmo. Em uma determinada configuração, podemos ter relações de dependência variadas entre os módulos do sistema.

Relações de Dependência entre Módulos

Sejam M_1 e M_2 módulos do sistema distribuído (SD). Definimos entre estes módulos a **Relação de Dependência**, notada "---->". Um módulo M_2 é dito dependente de M_1 (M_2---->M_1) quando M_2 é iniciador de uma transação da qual M_1 é respondedor.

As relações de dependência no sistema distribuído podem ser representadas por um grafo orientado G ("grafo de dependências"), onde os vértices correspondem aos módulos e os arcos às transações definidas para o sistema (figura 3.8).

Zonas de Dependência entre Módulos

"Zona de dependência M_k": formam a zona de dependência de M_k todos os vértices (módulos) do grafo G de SD que podem ser atingidos quando percorridos os arcos, a partir de M_k, no sentido contrário de suas orientações. As zonas de dependência identificadas em um grafo G podem ser classificadas em:

- **"Zona sem dependência"** : os efeitos das operações de mudança executadas sobre um determinado módulo, não se propagam sobre o restante do sistema na forma de inconsistências. Neste caso a zona de dependência se resume ao próprio módulo sujeito a operações de modificação. O módulo em questão é somente iniciador de transações. Na figura 3.7 o módulo M_2 tem uma zona de dependência zero.

- **"Zona de baixa dependência"**: um módulo M_k que apresenta uma zona de baixa dependência pode ser iniciador de transações mas é

obrigatoriamente respondedor.Na figura 3.7 fazem parte da zona de dependência de M_6 os módulos M_8 e M_9 que caracteriza a baixa dependência de M_6.

-**"Zona de alta dependência":** a zona que inclui praticamente todos os vértices de G, define uma alta dependência de M_k em relação a SD. Na figura 3.7 o módulo M_5 que tem uma zona de dependência que inclui os módulos M_6, M_7, M_8, M_9 e M_2, definindo uma alta dependência em SD.

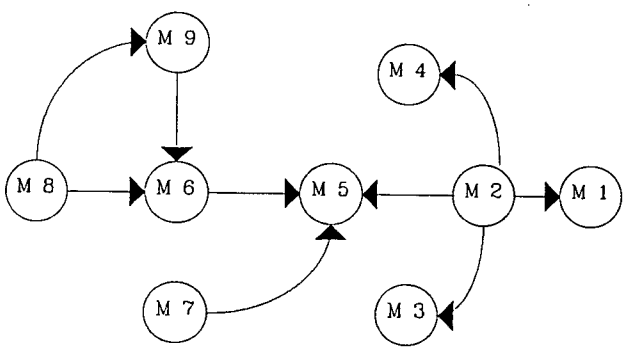

Figura 3.7 Grafo G de Dependência de um SD

Quiescência do Módulo

Um módulo para sofrer uma operação de mudança deve apresentar um estado no qual a execução desta operação não resulte em uma propagação de inconsistências pelo sistema. A quiescência de um módulo satisfaz estas exigências (ítem 2.4), é definida como um estado onde:

a- não existam transações pendentes com outros módulos;

b- não serão iniciadas novas transações a partir deste módulo;

c- este módulo não será sensibilizado por transações iniciadas por outros módulos.

Na elaboração de uma declaração CHANGE, o programador deve portanto levar em consideração a análise da zona de dependência entre os módulos. O conhecimento gerado desta análise adicionado às possibilidades do mecanismo "ponto de sincronismo" (a primitiva SINC da LIS) permite ao módulo alcançar o seu estado de quiescência, preservando desta maneira a consistência da aplicação durante as operações de mudança. A quiescência de um módulo está relacionada a sua zona de dependência da seguinte forma:

- uma declaração CHANGE que envolva operações de mudança sobre um módulo que possua uma "zona de alta dependência" deve executar operações que resultem na parada de toda a aplicação, antes de executar as mudanças sobre o módulo em questão. Desta forma estaremos garantindo a quiescência do módulo;

- uma zona de baixa dependência envolve operações que resultem na parada dos módulos pertencentes à zona de dependência do módulo alvo. Uma declaração CHANGE com este objetivo deve ser estabelecida, e ocasiona somente uma parada parcial, permitindo que o restante da aplicação continue a sua execução;

- um módulo que possua uma "zona sem dependência" atinge o seu estado de quiescência através da sua própria parada.

Após a definição da zona de dependência de um módulo é necessário determinar a ordem lógica da parada dos módulos pertencentes a esta zona (alta e baixa dependência). Esta ordem é determinada pelo sentido direto dos arcos da zona de dependência. Por exemplo um CHANGE que vai operar sobre M_5, na figura 3.8 deve parar na ordem indicada, os módulos: M_8-> M_9->M_6. Os módulos M_8, M_7 e M_2 não precisam ter uma ordem definida de parada pois são independentes entre sí.

A utilização dos mecanismos de configuração dinâmica descritos é apresentada em um exemplo no capítulo 5, no sentido de permitir um entendimento maior do suporte de configuração dinâmica da LIS.

3.5. Conclusão

Neste capítulo foram descritos o ambiente ADES e a linguagem LIS nos seus aspectos ligados às configurações estática e dinâmica de aplicações distribuídas.

A linguagem LINCS que permite declarar a configuração inicial do sistema, também é o meio para se especificar e controlar um processo de mudanças no sistema. Qualquer processo de mudanças gera efeitos sobre o sistema, podendo determinar o aparecimento de inconsistências no estado dos módulos da aplicação. Neste sentido este capítulo introduz alguns mecanismos incluídos na linguagem LIS que associados à determinação da zona de dependência têm como objetivo a minimização destas inconsistências.

CAPÍTULO 4
ASPECTOS DE IMPLEMENTAÇÃO

4.1. Introdução

O objetivo deste capítulo é apresentar os aspectos mais relevantes da implementação do suporte para configuração dinâmica, e as ferramentas utilizadas para o seu desenvolvimento. Este suporte foi dividido em duas partes que se interagem através de uma estrutura de dados comum (Lista de Ações).

A primeira envolve o processamento da declaração LINCS que descreve a mudança desejada, gerando a estrutura de dados correspondente a esta declaração. O processador desta linguagem deve também gerar Listas de Ações que especificam as operações que devem ser realizadas no sentido do sistema evoluir da configuração atual para a desejada.

A segunda parte descreve utilitários e módulos gerenciadores que atuam na mudança do sistema. Utilizando se das informações geradas no processamento da LINCS (lista de ações), estes módulos gerenciadores conduzem a modificação do sistema para a configuração desejada.

4.2. Processador LINCS

O processador LINCS é a ferramenta capaz de ler as especificações de configuração, processá-las e gerar as estruturas de dados correspondentes ao tipo de especificação. O diagrama da figura 4.1 apresenta os aspectos funcionais do processador. Neste ítem será apresentado alguns aspectos da estrutura funcional do

processador LINCS. Um detalhamento maior sobre o mesmo é encontrado em [Souza,88].

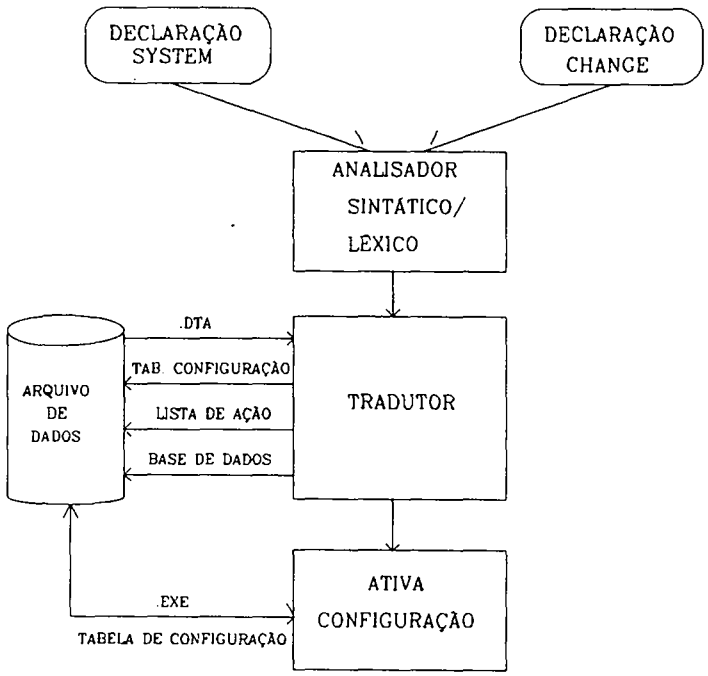

Figura 4.1 Processador LINCS

4.2.1 Aspectos Funcionais do Processador LINCS

O processador LINCS como todos os compiladores apresenta a ação combinada de várias unidades funcionais no sentido de realizar o processamento da linguagem. Estas unidades no referido processador podem ser resumidos em:

- **Analisador Léxico**: este realiza a leitura do arquivo contendo a declaração de configuração (SYSTEM ou CHANGE) e o reconhecimento da cadeia de caracteres denominados itens léxicos conforme definido na linguagem LINCS (CREATE, LOAD, REMOVE,etc). O analisador léxico tem a sua ação iniciada pelo analisador sintático e retorna um item léxico de entrada;

- **Analisador Sintático**: esta unidade com base em regras gramaticais definidas previamente pela sintaxe da linguagem determina se uma seqüência de itens léxicos válida. O reconhecimento da sintaxe permite a ativação do tradutor;

- **Tradutor**:o tradutor é caracterizado por um conjunto de ações determinando a semântica de cada instrução LINCS. Este conjunto de ações pode resolver:

  - **A tradução propriamente dita:** que consiste em transladar o código LINCS em uma estrutura de dados: a base de dados representativa da configuração desejada;

  - **Validação do Contexto:** verificar a existência dos arquivos dos tipos módulos declarados no **USE**. Estes arquivos são resultantes da compilação de códigos fontes em LINCE. O compilador produz dois arquivos: um arquivo executável (prefixo.exe) e um arquivo do tipo módulo (prefixo.dta) que contém gravadas e organizadas informações referentes às áreas de código e dados (tamanho,inicio, etc), aos portos e seus tipos e informações sobre parâmetros de tipo módulo;

  - **Validação de Parâmetro:** validar a parametrização real em relação aos parâmetros formais declarados nos tipos módulos;

  - **Validação de Compatibilidade de Tipos:** esta validação corresponde à verificação de compatibilidade dos tipos dos portos a serem conectados;

  - **Disponibilidade de Recursos:** esta função consiste basicamente em verificar a memória disponível em relação as necessidades para instalar a configuração desejada;

  - **Geração da Tabela de Configuração ou da Lista de Ações:** a partir da estrutura de dados traduzida do código LINCS, são criadas tabelas a serem utilizadas na configuração do sistema. No caso de uma declaração SYSTEM, são criadas tabelas de configuração que incluem primitivas do núcleo que serão executadas por códigos de iniciação nas respectivas estações de configuração. As listas de ações são geradas no fim do processo de tradução de uma declaração CHANGE.

Os analisadores sintático e léxico foram implementados com o uso de ferramentas como Yacc e Lex, respectivamente. Estes analisadores apresentam o comportamento de autômatos de estados finitos que evoluem no sentido de construir a árvore sintática. Os referidos analisadores são também responsáveis pela tarefa de reconhecimento de erros sintáticos e léxicos.

Outas unidades completam a estrutura funcional do processador LINCS que por razão de simplicidade não são apresentados neste documento. O processamento da declaração **SYSTEM** e das demais declarações LINCS ligadas à configuração estática são tratados em [Souza,88]. O processamento das declarações **CHANGE** e o suporte envolvido no gerenciamento da configuração dinâmica são abordados nos itens subseqüentes.

4.3. Processamento da Declaração CHANGE

O processamento da declaração CHANGE apresenta diferenças em relação à declaração SYSTEM. Uma destas diferenças é a existência prévia de uma base de dados representando o estado de configuração do sistema. Esta base de dados é o resultado do processamento de declarações anteriores (SYSTEM ou CHANGE). O processamento da declaração CHANGE, ao contrario da SYSTEM, deve envolver ações que representem operações inversas que desfaçam situações da configuração atual.

A figura 4.2 mostra uma estrutura simplificada da base de dados representando o estado atual da configuração do sistema. Durante o processamento da declaração CHANGE é criada uma versão temporária da base de dados a partir das informações de configuração da versão atual. As ações semânticas realizadas pelo tradutor são refletidas diretamente nesta base de dados temporária através da inserção de estruturas que representam novos tipos módulos , instâncias e conexões entre portos (USE, CREATE, LINK), e da retirada de estruturas que são resultantes das ações realizadas pelas operações inversas (REMOVE, DELETE, UNLINK).

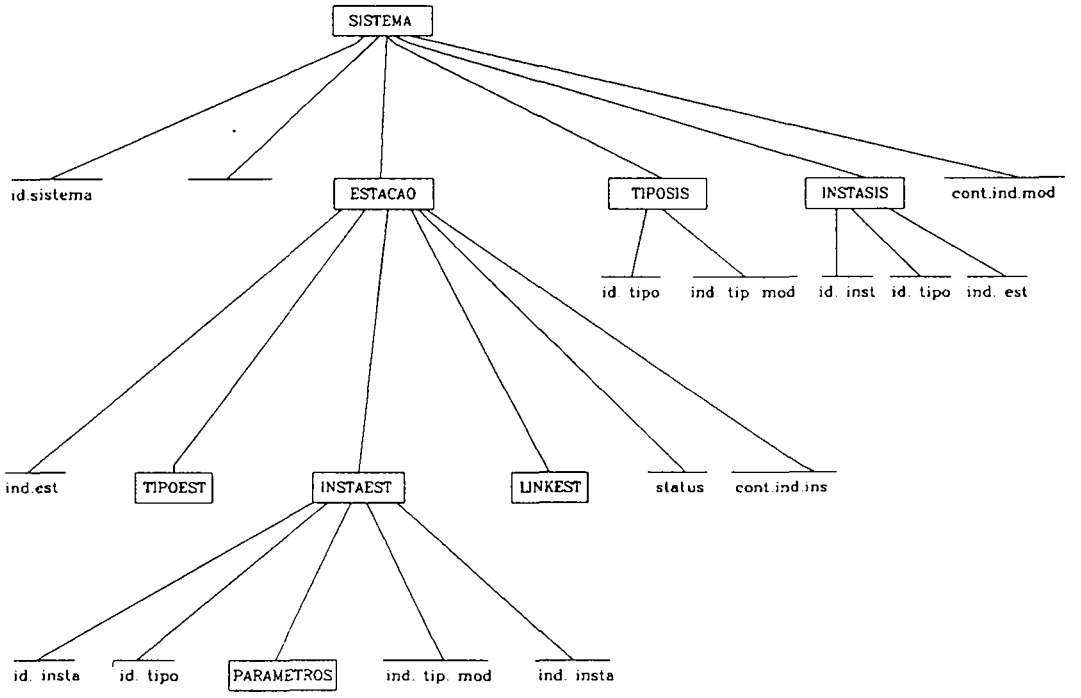

Figura 4.2 Base de Dados

É descrito a seguir o processamento de cada uma das declarações que são suportadas pela declaração CHANGE, as alterações provocadas na base de dados quando deste processamento e ainda o formato das operações geradas na lista de ações. O conteúdo das estruturas das operações da lista de ações são geradas a partir da própria base de dados temporária. A sintaxe das instruções LINCS são apresentadas no Apêndice A.

4.3.1. Processamento da Declaração de Contexto (USE)

A função desta declaração é a inclusão de novos tipos módulos que farão parte do contexto do sistema. As ações envolvidas na tradução desta declaração consistem em identificar os tipos módulos e verificar a existência dos arquivos correspondentes (prefixo.dta e prefixo.exe) na estação de trabalho. O tipo módulo

mesmo não fazendo parte da configuração do sistema, possui uma estrutura correspondente na base de dados temporária. O processamento do USE não tem como resultado a inclusão de operações na lista de ações.

4.3.2. Processamento da Declaração de Instanciação (CREATE)

Esta declaração possibilita quando necessário o carregamento de tipos módulos em estações alvo e a correspondente criação de instâncias. A tradução desta declaração representa a criação da estrutura "instaest" na estrutura "estação" correspondente na base de dados temporária. Como conseqüência desta operação são gerados as seguintes operações na lista de ações:

- **"LOAD":** esta operação é gerada caso o tipo módulo ainda não tenha sido carregado na estação. O formato desta operação na lista de ações envolve informações como o número da estação, tamanho do segmento de códigos, dados e de stack:

```
est_load = record
                    nome          : LOAD;
                    n_est         : integer;
                    nome_tmd              : string (20);
                    n_tmd,
                    cs_tam,
                    ds_tam,
                    ss_tam,
                    end_inicio : word;
end;
```

- **"CREATE":** a estrutura CREATE da lista de ações contém informações necessárias à operação correspondente: número da estação, número do tipo módulo, número da instância, etc :

```
est_create = record
                    nome          :Create;
```

```
                                n_est          : integer;
                                n_tmd,
                                n_insta        :integer
                                ind_dta,
                                tam_dados              :word;
                                n_parâmetro:integer;
                                parâmetro              :array [1..10] of Param;
end;
```

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

A ligação entre um porto de saída e um porto de entrada é possível através da utilização da declaração LINK. Para gerar na lista de ação uma operação de ligação (LINK), é necessário primeiramente verificar se as instâncias dos tipos módulos foram criadas e se existe uma compatibilidade entre os portos a serem interconectados. Se estas condições forem satisfeitas a base de dados temporária é atualizada com a introdução de uma nova representação de ligação entre um porto de saída e um porto de entrada. A operação LINK contendo no seu formato informações referentes aos portos a serem ligados (número da estação, número dos portos, número de instâncias, etc) é incluída na lista de ações com a seguinte estrutura:

```
est_link = record
                                nome           : LINK;
                                n_est          : integer;
                                n_insta_ps,
                                n_port_ps,
                                n_est_pe,
                                n_insta_pe,
                                n_port_pe              : integer;
end;
```

4.3.4. Processamento da Declaração de Destruição de uma Instância (DELETE)

Esta declaração é uma operação inversa à CREATE; possibilitando a destruição de uma instância em uma determinada estação. Na validação desta declaração, o tradutor verifica na base de dados se a instância a ser destruída faz parte da configuração inicial e se os portos da instância estão desligados O processamento desta declaração envolve a retirada da representação da instância da base de dados temporária e a inclusão da estrutura representando a operação (DELETE) na lista de ações; com informações referentes ao número da estação e o número da instância:

```
est_delete = record
                        nome        : DELETE;
                        n_est,
                        n_insta     : integer;
end;
```

4.3.5. Processamento da Declaração de Destruição da Conexão entre Portos (UNLINK)

A declaração UNLINK é outro tipo de declaração inversa. É usada para desfazer as ligações lógicas existentes entre portos de instâncias no sistema. Para sua validação é necessário uma verificação da existência da ligação através da análise das estruturas de ligação das instâncias envolvidas. O resultado da tradução desta declaração é a introdução da operação UNLINK na lista de ações , que contém no seu formato informações do LINK, diferenciando-se apenas do identificador do campo "nome".

4.3.6. Processamento da Declaração de Retirada de Tipos Módulos (REMOVE)

A declaração REMOVE provoca a retirada do segmento de código de um tipo módulo e a destruição do descritor de tipo módulo em uma determinada estação. Esta declaração é válida se não existir nenhuma instância do tipo módulo na estação. Uma vez confirmada esta situação, a representação do tipo módulo é retirada da estação correspondente na base de dados temporária. A operação REMOVE, no formato abaixo, é introduzida na lista de ações:

```
est_remove = record
                        nome    : REMOVE,
                        n_est,
                        n_tmd   : integer;
end;
```

4.3.7. Processamento da Declaração de Parada de uma Instância (STOP)

A parada de uma instância de um módulo implica na parada de todas as tarefas pertencentes a esta. Esta operação pode ser **assíncrona**, quando independente do estado das tarefas, ou **síncrona**, onde a parada está condicionada a que as tarefas tenham atingido determinados pontos de seus processamentos. No caso da parada síncrona é necessário a utilização de pontos de sincronismo. Os **pontos de sincronismos** são então os mecanismos que determinam o momento em que cada tarefa deve estar para que se efetue a parada síncrona de uma instância; a localização destes pontos é programada em cada tarefa pelo projetista do módulo. O formato desta operação é incluída na lista de ações e apresenta informações do número da estação, número da instância e o tipo da parada:

```
est_stop = record
                        nome            : STOP;
                        n_est,
                        n_insta         : integer;
```

```
                                síncrono        : boolean;
end;
```

4.3.8. Processamento da Declaração de Iniciação de uma Instância (START)

A declaração START retira as tarefas de uma instância parada (submetida anteriormente a uma operação STOP) da fila de tarefas suspensas deixando as no estado de prontas, competindo pelo processador. O processamento da declaração START incluí o formato abaixo na lista de ações:

```
est_start = record
                                nome            : START;
                                n_est,
                                n_insta         : integer;
end;
```

4.3.9. Considerações Gerais

Ao contrário do processamento de uma declaração SYSTEM, que ativa o processo de configuração do sistema após a validação de suas declarações, no final do processamento da declaração CHANGE o tradutor não ativa o processo de configuração dinâmica. O efeito da tradução envolve a organização e a geração da base de dados temporária (não efetivada ainda como versão atual do estado da configuração) e a lista de ações em um arquivo (prefixo.lta) na estação de trabalho. O processo de ativação da configuração dinâmica é iniciado quando o sistema já está em curso de operação. Isto é feito pelo operador do sistema através de uma interface conversacional, suportada pelos gerenciadores de configuração que será objeto do próximo ítem..

4.4. Suporte para a Configuração Dinâmica: Gerenciamento da Configuração

A configuração dinâmica é executada por um conjunto de módulos gerenciadores que fazem parte do SOD. Neste item é realizada uma descrição sucinta dos módulos gerenciadores, bem como da relação existente entre estes. A figura 4.3 representa o sistema operacional distribuído.

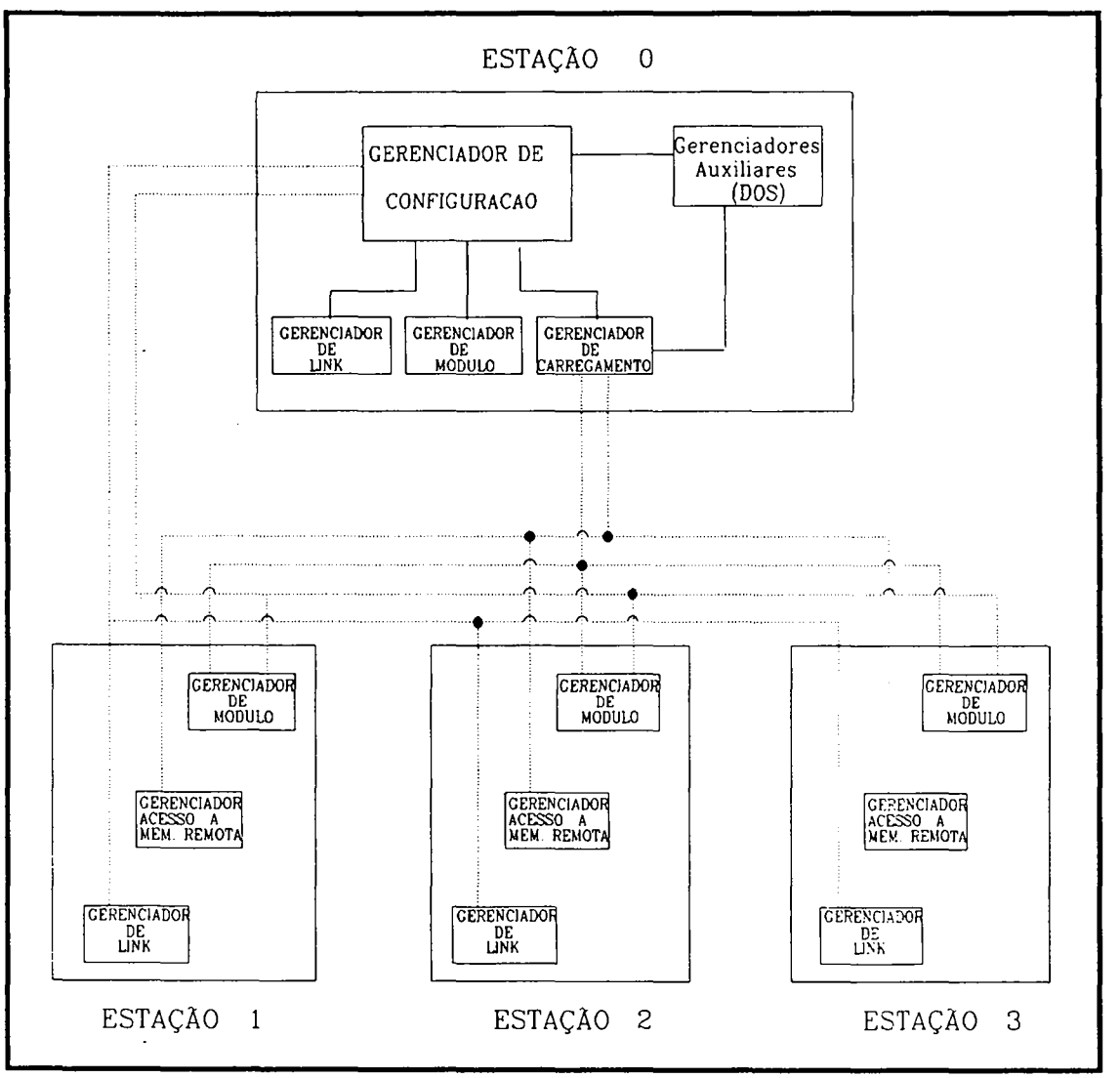

Figura 4.3 Suporte de Configuração Dinâmica

4.4.1. Gerenciador de Configuração

O módulo gerenciador de configuração é peça fundamental no tratamento da evolução do sistema. Algumas atribuições deste módulo são distinguíveis:

- ler o arquivo contendo a lista de ações e interpretar as operações desta lista, estabelecendo as iterações necessárias no sentido de executar as mudanças propostas;

- manter um controle centralizado da execução das operações explicitadas na lista de ações, executadas pelos gerentes locais;

- gerenciar e executar a lista reversa na impossibilidade de evolução na execução de listas de ações; e

- efetivar a base de dados temporária como versão atual da representação do estado de configuração do sistema. Esta efetivação se dá quando da confirmação do processo de mudanças no sistema.

4.4.1.1. Processamento da Lista de Ações

O gerenciador de configuração através de troca de mensagens com os mecanismos do sistema operacional hospedeiro (gerenciadores auxiliares) carrega na memória principal o arquivo contendo a lista de ações. A lista de ações se apresenta na forma de lista ligada, e é constituída das estruturas descritas no item anterior. A lista ligada apresentada favorece, diante de uma exceção, a restauração do estado de configuração anterior (execução da lista reversa).

Segundo a política de execução da lista de ações apresentadas no capítulo 3, o gerenciador de configuração interpreta cada formato das estruturas na lista, enviando mensagens síncronas aos gerenciadores locais responsáveis pelas execuções das operações correspondentes. Cada mensagem contém as informações necessárias para as execuções. Como resultado da execução de uma operação, o gerenciador local completa a comunicação enviando uma resposta indicando o resultado da execução pedida. Se todas as respostas resultantes da execução das operações solicitadas forem satisfatórias a configuração dinâmica é considerada

terminada. Neste caso a base de dados temporária é efetivada como versão atual do estado de configuração do sistema, descartando a anterior.

4.4.1.2. Execução da Lista Reversa de Ações

A lista reversa é o mecanismo que permite colocar o sistema no estado de configuração imediatamente anterior ao inicio da execução da lista de ações. Como pode ser visto na figura 4.4 a lista de ações apresenta apontadores indicando a próxima estrutura e a anterior na lista. Desta forma, durante a execução normal, a lista é percorrida no sentido da esquerda para a direita. Caso ocorra uma exceção durante a execução de uma operação, o processo normal é interrompido, e a lista passa a ser percorrida no sentido da direita para a esquerda, do ponto onde foi gerada a exceção, executando com semântica inversa as operações.

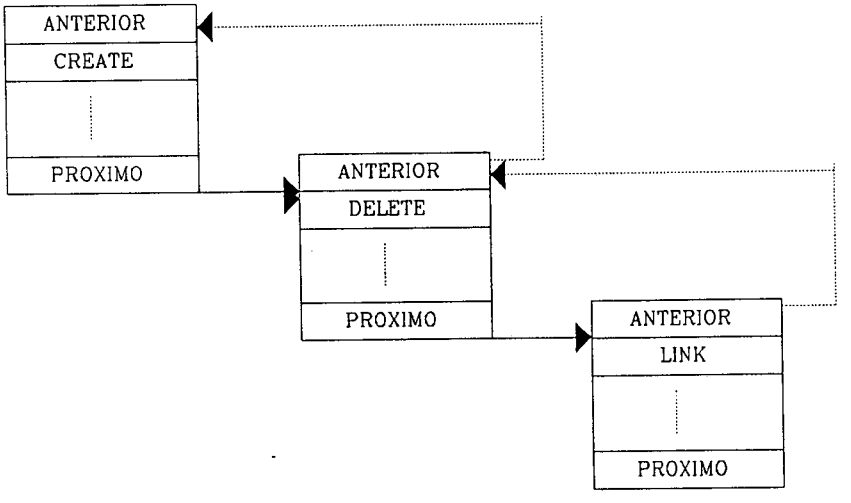

Figura 4.4 Lista de Ações

4.4.2. Gerenciadores Locais

Os gerenciadores locais são os módulos responsáveis pela execução das operações da lista de ações . Estes módulos são instanciados na estação de trabalho e/ou estações remotas dependendo da aplicação e das possibilidades de mudança no sistema. A seguir são apresentados os gerenciadores locais, descrevendo as suas principais funções e apresentando um cabeçalho em LINCE de suas interfaces.

4.4.2.1. Gerenciador de Ligação

O gerenciador de ligação é o módulo gerenciador responsável pela criação e a destruição de canais de comunicação entre portos de módulos. As operações de conexão/desconexão dos portos são executadas usando primitivas correspondentes do suporte de tempo de execução (NTR). A interface do gerenciador de Ligação é mostrada abaixo:

```
Module gen_link;

    Use config.def;

    PORT PGLINK : IN (est_link REPLY integer);
```

Todos os pedidos de operações relacionados à criação e destruição de canais de comunicação durante a configuração dinâmica são enviadas através de mensagens (est_link) ao porto de entrada PGLINK do gerente de ligação.

4.4.2.2. Gerenciador de Módulo

O gerenciador de módulo possui um conjunto de tarefas que executam operações sobre os tipos módulos e instâncias de módulos na estação. Estas operações envolvem:

- remover o tipo módulo da memória;

- alocação de área de memória;

- criar instâncias;

- destruir instâncias; e

- parar e dar partida a instâncias.

A interface do gerenciador de Módulo envolve um conjunto de portos tipados conforme a funcionalidade associada:

```
Module gen_mod;

   Use config.def;

   PORT PGMOD_CREATE : IN (est_create REPLY integer);

   PORT PGMOD_START  : IN (est_start  REPLY integer);

   PORT PGMOD_DELETE : IN (est_delete REPLY integer);

   PORT PGMOD_LOAD   : IN (est_load   REPLY segmento);

   PORT PGMOD_REMOVE : IN (est_remove REPLY integer);
```

A seguir são descritas as operações executadas pelo gerenciador de módulo quando solicitado por uma mensagem correspondente, nos portos apropriados.

Criar Instância (CREATE): a mensagem "est_create" contém os identificadores da instância e o do tipo módulo e os parâmetros a serem usados na instanciação. O gerenciador de módulo utiliza primitivas do NTR na criação de um Bloco de Controle de Instância (BCI) dentro da área de dados do NTR e a criação de estruturas que representem as tarefas (BCT) e os portos (BCP) de comunicação.

A mensagem resposta contém um código que indica o sucesso da instanciação ou um código de erro. No caso de perda da resposta, o gerente de configuração pode pedir a execução da operação novamente, pois é previsto um código de resposta indicando que a instância já existe.

Destruir Instância (DELETE): o gerenciador de módulo ao receber a mensagem (est_delete) contendo o identificador da instância a ser destruída, inicia um processo de destruição dos blocos de controle que representam a instância (ou seja, BCI, BCT e BCP) e a liberação do segmento de dados pertencentes à instância. A destruição da instância é realizada através de chamadas correspondentes às primitivas do NTR, isto se os portos do módulo já estão desconectados.

Aloca Memória: ao receber esta mensagem "est_load" é identificado o tipo do módulo e o tamanho de memória a ser alocada. Através de chamadas às primitivas do NTR é alocado o segmento de memória, criado um bloco de controle de tipo módulo (BCTM) e habilitada a criação de instâncias. A mensagem resposta contém o endereço do segmento de memória alocado ou um valor zero indicando que não existe espaço disponível em memória.

Remove Módulo (REMOVE): a remoção de um tipo módulo implica na liberação da área de memória ocupada pelo segmento de código do tipo módulo e a destruição do BCTM. O gerenciador de módulo recebe a mensagem "est_remove" e executa a chamadas ao NTR no sentido de remover o tipo módulo. Esta remoção somente é efetivada se não existirem na estação instâncias do tipo módulo.

Parar e Reiniciar a Instância (STOP-START): a mensagem "est_stop e est_start" contém o identificador de uma instância e um atributo indicando se a operação a ser executada na instância é de parada (STOP) ou de reiniciação (START). Se a mensagem contém uma operação START, o gerenciador de módulos reinicia as tarefas "suspensas" da instância parada, efetivando portanto a ativação da instância. Se a mensagem corresponde a um STOP, o gerenciador de módulo vai executar uma das seguintes operações:

- **Parada assíncrona:** na execução de uma parada assíncrona pelo gerenciador, todas as tarefas deixam de concorrer ao processador imediatamente, e são colocadas na fila de "tarefas suspensas",

caracterizando a parada da instância. Neste caso o gerenciador de módulo não é suspenso; e

- **Parada síncrona:** nesta operação o gerenciador de módulo sinaliza o bloco de controle de instância indicando que esta instância está em estado de configuração e se suspende na espera da parada da instância. As tarefas da instância através de uma interface com o NTR verificam periodicamente se a instância está em estado de configuração, caso esteja, a tarefa se suspende imediatamente ou executa algum procedimento antes de sua suspensão. A última tarefa a se suspender ativa o gerenciador de módulo que responde ao gerenciador de configurador o sucesso da operação.

Este mecanismo de parada síncrona contém um mecanismo de "TIMEOUT" que pode ser parametrizado pelo gerenciador de módulo. A parada de todas as tarefas da instância está condicionada ao valor deste parâmetro. Caso o tempo para parada da instância ultrapasse este valor, o mecanismo ativa o gerenciador de módulo. Ao ganhar novamente o processador o gerenciador executa uma primitiva do núcleo, que determina o número de tarefas ainda não suspensas. Sendo este valor diferente de zero, o gerenciador produz um código de erro para o gerenciador de configuração e as tarefas da instância que foram suspensas voltam a ser ativadas. O mecanismo de interação entre tarefa e NTR está embutido dentro da primitiva SINC da linguagem LINCE discutida no capítulo 3.

4.4.2.3. Gerenciador de Carregamento (LOADER)

O gerenciador de Carregamento apresenta como interface os seguintes portos tipados:

```
Module gen_loader;

  Use config.def;

  PORT PGLOAD_PGMOD: OUT (est_load REPLY segmento);
```

```
PORT PGLOAD_PGMOR: OUT (est_load REPLY segmento);

PORT PGLOAD_PGARM: OUT (controle REPLY integer );

PORT PGC_PGLOAD : IN (est_load REPLY integer );

PORT PGC_PGARQ : OUT (est_arq REPLY integer );
```

Este gerenciador tem por objetivo o carregamento de códigos executáveis em uma estação do sistema. Esta operação é ativada com a recepção de uma mensagem "est-load" vinda do gerenciador de configuração. A mensagem contém informações referentes ao tipo módulo, e dá estação onde o carregamento deve ser realizado.

O carregamento de um tipo módulo em uma estação alvo depende primeiramente da alocação de uma área de memória correspondente ao tamanho do tipo módulo. Neste sentido, o gerenciador de carregamento envia uma mensagem ao gerenciador de módulo pedindo a alocação de um segmento de memória e a criação dos BCTM. O retorno da mensagem resposta deve informar o endereço de inicio da área de memória para o carregamento, ou um valor zero que significa a não existência de espaço disponível de memória para o carregamento do tipo módulo. Neste último caso um código de erro é retornado ao gerenciador de configuração.

O sucesso da alocação inicia o processo de carregamento do tipo módulo. O carregamento na estação de trabalho é executada através de mecanismos do sistema operacional hospedeiro que estão disponíveis nos gerenciadores auxiliares (item 4.4.2.5). O gerenciador de carregamento envia uma mensagem "est_arq" sinalizando ao gerenciador auxiliar para que execute o carregamento. O sucesso desta operação é retornado ao gerenciador de configuração indicando que o tipo módulo foi carregado.

O carregamento de um tipo módulo em uma estação de execução pode ser verificado através da figura 4.6 (b). O gerenciador de configuração inicialmente realiza um pedido de ligação dinâmica entre o gerenciador de carregamento na estação de trabalho aos gerenciadores de módulo e de acesso remoto à memória

(item 4.4.2.4), localizados na estação de execução. O gerenciador de módulo da estação de execução receberá um pedido de alocação de memória e retornará o endereço do inicio da área alocada. O gerente de acesso remoto à memória executará um protocolo com o gerente de carregamento envolvendo a transferência do tipo módulo e posterior carregamento na memória alocada. Este protocolo envolve a decomposição do arquivo tipo módulo em blocos de 256 bytes e o carregamento destes na memória, no endereço previamente determinado. O sucesso deste protocolo é sinalizado ao gerenciador de configuração.

4.4.2.4. Gerenciador de Acesso Remoto à Memória (GARM)

O GARM possibilita o acesso à memória de uma estação de execução para carregamento de códigos executáveis de tipos módulos. Seus serviços são utilizados pelo gerenciador de carregamento, conforme o item anterior. A interface do GARM toma a seguinte forma:

```
Module gen_GARM;

    Use config.def;

    PORT PGARM_LOAD : IN  (controle REPLY integer );
```

Os arquivos transferidos de tipos módulos devem conter juntamente com o código executável, uma área de cabeçalho e uma área para "offsets" dos endereços relocáveis. No final do carregamento do último bloco o GARM executa um procedimento de definição das quantidades relocáveis em relação ao endereço de carga do tipo módulo. O sucesso do carregamento é sinalizado ao gerente de carregamento.

4.4.2.5. Gerenciadores Auxiliares

Os Gerenciadores Auxiliares são mecanismos que possibilitam encapsular as facilidades do sistema operacional hospedeiro (SOH) no sentido de garantir a iteração do sistema operacional distribuído (gerenciadores) e a própria aplicação com facilidades do SOH como serviços de arquivos, "drivers" de teclado, monitor, etc. Os gerenciadores auxiliares apresentam como interface básica, a indicada abaixo:

```
Module gen_aux;

    Use config.def

    PORT PGAUX_TECLADO : OUT (integer);

    PORT PGAUX_VIDEO   : IN (est_video REPLY integer);

    PORT PGAUX_ARQ     : IN (est_arq REPLY integer);
```

Os gerenciadores Auxiliares são módulos portanto que encapsulam funções que normalmente são atribuídas ao SOH. Estas funções são acessadas através de módulos servidores que visam a ordenação na utilização dos recursos gerenciados fornecidos pelo sistema operacional hospedeiro (SOH) na gerência das interfaces com o usuário. A seqüencialização nas facilidades do SOH é realizada através de um módulo gerenciador de sinalização que recebe e envia mensagens através das interfaces mostradas acima e ativa as funções do SOH, pelo envio subseqüente aos módulos correspondentes. A figura 4.5 ilustra os gerenciadores Auxiliares.

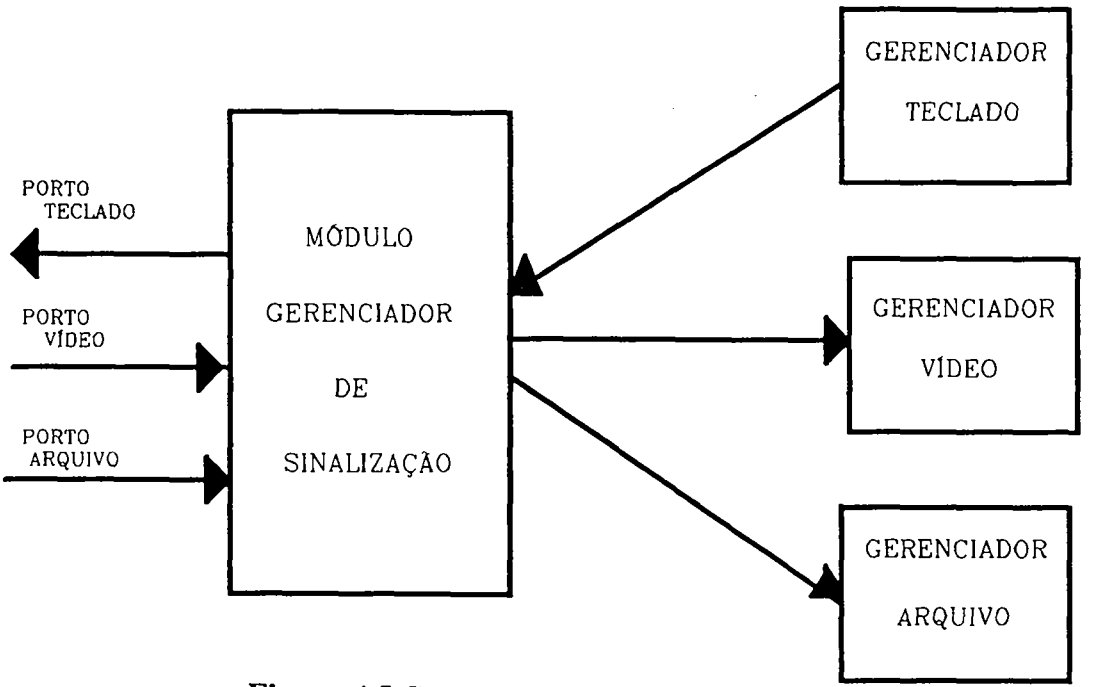

Figura 4.5 Gerenciadores Auxiliares

O gerenciador auxiliar implementado apresenta um módulo servidor de teclado que permite a obtenção de caracteres ASCII através de um "driver" de teclado, quando uma tecla é pressionada. O caracter correspondente à tecla pressionada é enviada ao gerenciador de sinalização que por sua vez retransmite esta informação aos módulos conectados ao porto de saída (PAUX_TECLADO).

O módulo servidor de arquivo prevê um conjunto de funções que possibilitam a manipulação de arquivos em unidades de discos, tais como: criar arquivo, abrir arquivo, escrever em arquivo, e fechar arquivo. Estas funções são determinadas através de mensagens "est_arq" que são enviadas ao seu porto de entrada (PAUX_ARQ).

O módulo servidor de vídeo possui um conjunto de funções que possibilitam a manipulação de caracteres no vídeo. O módulo recebe mensagens contendo pedidos de execução destas funções através do seu porto de entrada (PAUX_VIDEO).

4.4.2.6. Iteração dos Gerenciadores Locais e de Configuração

O gerenciador de configuração após interpretação de uma operação da lista de ações, prepara uma mensagem para o gerenciador local que a executará. A comunicação entre o gerenciador de configuração e o gerenciador local ocorre através de um protocolo do tipo pedido-resposta, onde o pedido contém informações sobre a operação a ser executada e a resposta indica o sucesso da operação. A figura 4.6 (a) apresenta um diagrama de tempo ilustrando o protocolo entre o gerenciador de configuração e os gerenciadores locais na execução de operações como LINK, UNLINK e LOAD.

Quando a operação deve ser executada por um gerenciador local em uma estação remota, o protocolo estabelecido a partir do gerenciador de configuração é mais complexo (figura 4.6 (b)). Isto ocorre devido ao fato do gerenciador de configuração não estar estaticamente conectado aos gerentes locais das estações de execução (figura 4.3). Para execução da operação, neste caso, o protocolo estabelece a ligação dinâmica entre o gerenciador de configuração ao gerenciador local da estação de execução. A figura 4.6 (b) representa a criação de uma instância de um tipo módulo em uma estação remota.

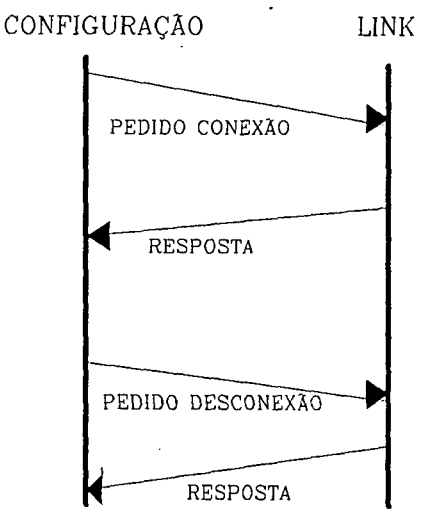

Figura 4.6 (a) Protocolo Pedido-Resposta Local

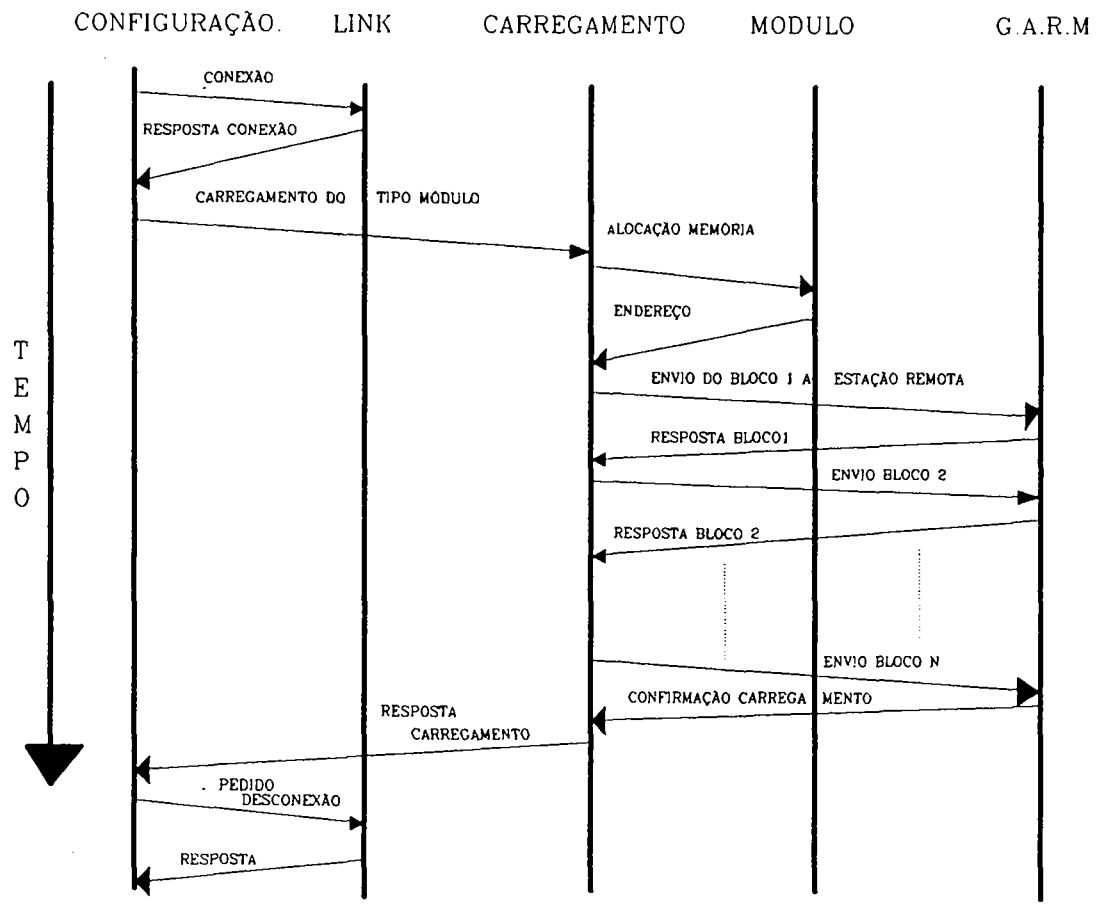

Figura 4.6 (b) Protocolo Pedido-Resposta Remoto

A iteração entre gerenciadores ocorre a partir do gerenciador de configuração que centraliza o controle da execução das listas de ações. A comunicação entre o gerenciador de configuração e os gerenciadores locais ocorre através de canais de comunicação do tipo síncrono. Aos envios de mensagens nestes canais estão associados mecanismos do tipo TIMEOUT, com o objetivo de possibilitar recobrimentos de erros que possam ocorrer durante a comunicação. Inicialmente, caso o TIMEOUT estoure, é gerado um código de erro que é manipulado por um manuseador (handler) de exceções. O "handler" associado a este erro possibilita que novas tentativas da execução da operação de configuração

sejam realizadas, permitindo só em último caso a interrupção da configuração dinâmica.

4.5. Ferramentas Utilizadas

No desenvolvimento do suporte de configuração dinâmica foram utilizadas as ferramentas Yacc [Johnson,78] e Lex [Lesk,82].

A ferramenta YACC é destinada à geração de analisadores sintáticos guiados por especificações de alto nível. As especificações descrevem um conjunto de regras gramaticais que definem a sintaxe da linguagem a ser analisada. O usuário especifica a gramática da linguagem e a ação que deve ser executada quando esta for reconhecida.

A ferramenta LEX é dirigida por uma especificação de alto nível com o objetivo de gerar um analisador léxico, onde a entrada para esta ferramenta é um arquivo contendo uma tabela de expressões regulares e fragmentos de programas. As expressões definem as unidades sintáticas elementares que o analisador gerado será capaz de reconhecer e classificar. Os fragmentos de programas, associados com as expressões regulares, são as ações que o analisador léxico deve executar quando as expressões regulares forem satisfeitas.

O programa gerado pela ferramenta YACC (parser) se encarrega de chamar automaticamente o analisador gerado pelo Lex para o reconhecimento de itens léxicos. Desta maneira foi possível implementar um tradutor dirigido por sintaxe. Uma ação semântica é executada para cada seqüência de itens léxicos de entrada validada.

Na implementação do suporte de configuração dinâmica foram utilizadas as linguagens C no caso do tradutor e LINCE para a implementação do Suporte para Gerenciamento de Configuração. Esta última escolha possibilita a disponibilidade dos gerenciadores locais serem incluídos ou não, conforme as particularidades da estação de destino.

4.6. Conclusão

Os aspectos de implementação de um suporte para configuração de um sistema distribuído no sentido de configuração dinâmica foram apresentados. Neste capítulo, o uso das ferramentas citadas permitiu uma implementação rápida, possibilitando a polarização de todos os esforços nos aspectos conceituais dos mecanismos envolvidos com a configuração dinâmica. As implementações foram realizadas no sentido de obedecer técnicas de engenharia de software proporcionando então características, como a modularidade, que permite mudanças na sintaxe e possivelmente na semântica da linguagem LINCS.

Os resultados obtidos pela configuração dinâmica serão avaliadas através de um exemplo que é apresentado no capítulo 5.

CAPÍTULO 5
PROGRAMAÇÃO EM LIS DE SISTEMAS DISTRIBUÍDOS EVOLUTIVOS

5.1. Introdução

Os mecanismos descritos nos capítulos anteriores são ferramentas importantes na minimização de inconsistências que se produziriam quando das mudanças dinâmicas efetuadas sobre um sistema distribuído. No entanto, a correta utilização de muitos destes mecanismos depende do conhecimento do comportamento da aplicação por parte do programador.

O objetivo deste capítulo é apresentar uma metodologia de concepção de programas distribuídos, que oriente o projetista na construção de um sistema distribuído suportando configuração dinâmica. Esta metodologia leva em consideração as necessidades da programação de mecanismos para preservar a aplicação em mudanças dinâmicas de configuração. Em seguida a utilização desta metodologia e a eficiência dos mecanismos para configuração dinâmica da linguagem LIS, são ilustrados a partir de um exemplo de uma célula flexível de manufatura simulada no LCMI.

5.2. Metodologia para a Concepção de Sistemas Evolutivos Programados em LIS

Na concepção de programas distribuídos que por exemplo, fazem parte do controle de uma planta industrial, o programador da aplicação deverá considerar as

possíveis evoluções da configuração, tanto de ordem tecnológica como operacional. Estas antecipações são importantes pois, permitem a programação de mecanismos LIS, possibilitando que os processos de mudanças dinâmicas de configuração não se reflitam de maneira danosa sobre a aplicação controlada.

A metodologia introduzida neste item tem o sentido de orientar o projetista na programação distribuída, com perspectivas de configuração dinâmica. Esta metodologia é aplicada nas fases de especificação e de projeto detalhado em modelos clássicos de engenharia de software. Os passos que regram a programação segundo esta metodologia são descritos a seguir.

Definição dos Módulos no Sistema Distribuído

Esta etapa corresponde à clássica decomposição funcional. O projetista conhecendo a aplicação deve definir os módulos que comporão o programa distribuído. O uso de técnicas formais (SADT, Redes de Petri, etc) e de ferramentas (analisadores, simuladores) é útil no sentido de facilitar estas definições, bem como valida-las.

Identificação de Módulos Passíveis de Operações de Configuração

O programador deve antecipar situações potenciais onde necessita-se de mudanças na configuração de sistemas. Estas antecipações podem corresponder a mudanças atendendo aspectos operacionais, como troca de algoritmos, por questões de eficiência ou de demanda da aplicação, tratamento de faltas, etc.

A evolução da planta e o desenvolvimento tecnológico são outros fatores que podem provocar mudanças dinâmicas. Os módulos LIS que interfaceiam com equipamentos (CLP, CNC, Robôs, placas de aquisição de dados, etc.) são altamente dependentes da evolução tecnológica.

Dentro deste contexto o programador conhecendo a aplicação pode antecipar situações de mudanças de configuração nas quais o sistema poderá estar sujeito, podendo desta maneira identificar os módulos no sistema, passíveis de operações de configuração.

Identificação das Zonas de Dependências

Uma vez identificado os módulos que podem sofrer operações de configuração (stop, delete, remove, etc), o passo seguinte é dimensionar as respectiva zonas de dependência (3.4.5), ou seja, identificar os módulos que seriam afetados pela execução de uma declaração CHANGE sobre um determinado módulo.

Na definição das zonas de dependência o projetista do sistema reúne condições para estabelecer estratégias de configuração que vão caracterizar as declarações CHANGE previstas no sistema:

- Uma zona de dependência define a priori o grau de dificuldade na execução de uma determinada operação de configuração. Se a zona pode ser classificada como de alta dependência, o número de módulos afetados por uma operação de configuração é muito grande. Neste caso, a declaração CHANGE deve ser construída para parar todos os módulos da aplicação, antes da execução da mudança da configuração.

- Se uma zona for baixa dependência, teremos operações de efeitos limitados a poucos módulos. A execução de uma operação de configuração exige somente a quiescência dos módulos envolvidos (3.4.5); neste caso teremos uma parte do sistema em estado de configuração e o restante do sistema em operação. A declaração CHANGE envolvida deve provocar a quiescência do módulo, parando os módulos que iniciam transações com este último .

Programação dos Mecanismos de Manutenção de Consistência

Durante a execução das mudanças, o sistema é submetido a um período de perturbações decorrentes da parada de módulos, desconexão de portos, destruição de instâncias, etc. É importante que durante as mudanças, informações não sejam perdidas, e a aplicação controlada seja preparada para o transitório do sistema. Como citado anteriormente, desejamos deixar a aplicação em um estado consistente. Para conseguir estes objetivos devem ser inseridos nos módulos envolvidos na declaração CHANGE mecanismos próprios para as possíveis exceções geradas pela configuração dinâmica. O mecanismo ponto de sincronismo

pertencente a LIS, quando bem utilizado pode resolver a maioria dos casos. Este mecanismo foi discutido no capítulo 3.

5.3. Exemplo Ilustrativo da Utilização do Suporte de Configuração Dinâmica

O exemplo de uma célula flexível de manufatura foi desenvolvido com dois objetivos principais.O primeiro, no sentido de testar os mecanismos propostos na linguagem Lis para a configuração dinâmica, e o segundo no sentido de servir de veículo para a aplicação da metodologia introduzida neste capítulo. No item subseqüente é descrita a célula flexível do exemplo.

5.3.1. Descrição de uma Célula Flexível de Manufatura

A célula exemplo tem como componentes uma esteira, uma câmera, um depósito intermediário (buffer), dois robôs (Rp e Ra), um torno e um depósito para peças torneadas (figura 5.1) .

A esteira transporta peças de diversos tipos. A distribuição de peças pela esteira é aleatória. Esta esteira possui duas posições: P1 e P2. Quando uma peça atinge P1, o movimento é interrompido para que seja feita a visualização da peça pela câmera. Uma vez esgotado o tempo de visualização da peça pela câmera, a esteira retorna ao seu movimento. As informações obtidas na visualização são classificadas segundo parâmetros pré-estabelecidos. Com base nestes parâmetros, peças podem ser eventualmente rejeitadas.

A peça na posição P2 provoca também a interrupção do movimento da esteira. Neste caso, a interrupção é para que a peça possa ser retirada pelo robô Rp. Peças rejeitadas pela visualização não são pegas pelo robô Rp, continuando portanto na esteira. A esteira parada na posição P2 deve continuar o seu movimento após o tempo de retirada de peça (pelo robô Rp). Várias peças podem se encontrar entre P1 e P2.

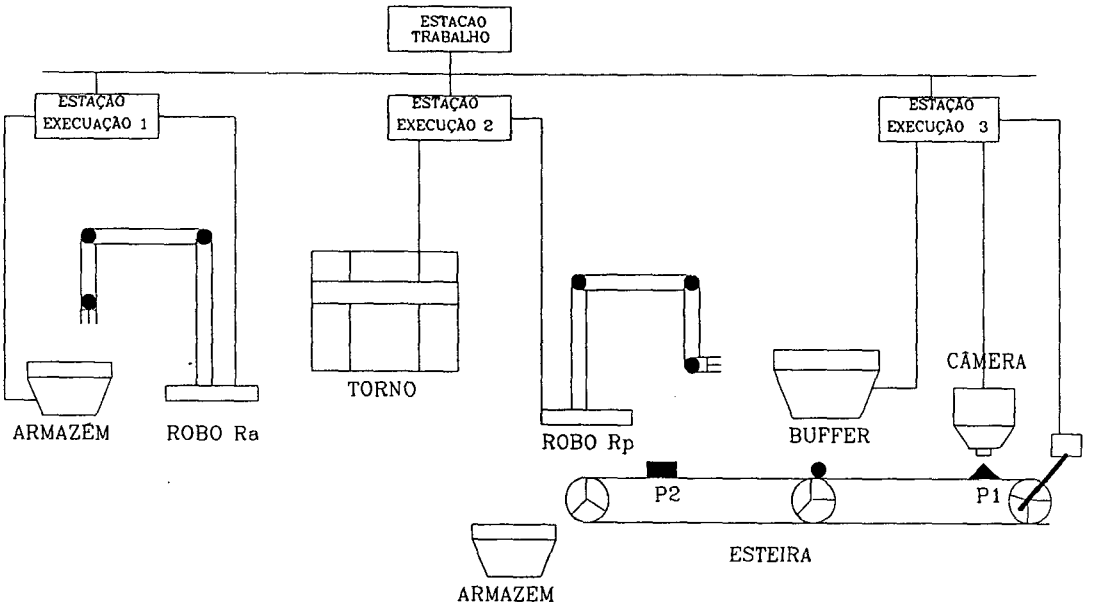

Figura 5.1 Célula Flexível

Para que uma peça possa ser colocada no torno é necessário que este esteja disponível (sem peça) e ainda que o robô Ra não se encontre nas proximidades do torno. Caso o torno esteja ocupado, peças que ocupem a posição P2 serão colocadas no depósito intermediário (buffer). Ao final da usinagem da peça no torno, esta é retirada pelo robô Ra. O torno volta a ser ocupado pela ação do robô Rp que busca uma peça no buffer.

O buffer, neste exemplo, tem o objetivo de introduzir um maior grau de paralelismo entre as atividades realizadas na célula. Com o torno ocupado, a existência de uma peça em P2 não implica na parada da esteira e de todo o sistema de transporte até o fim da usinagem da peça que está no torno; o robô Rp ao retirar a peça da posição P2 e colocá-la no buffer, provoca o movimento novamente na esteira.

5.3.2. Programação da Célula Exemplo

A programação em Lis da célula flexível segue a sistemática introduzida no item 5.2. Como primeiro passo, foi feita a decomposição da aplicação em módulos controladores, simuladores e visualização:

- **Módulos Controladores:** são os módulos responsáveis pela execução dos algoritmos de controle que comandam a operação dos componentes da célula. Na decomposição realizada existe um módulo controlador para cada componente (robô, esteira, câmera, etc.);

- **Módulos Simuladores:** estes módulos simulam os dispositivos físicos da célula exemplo. Os módulos simuladores são ativados através de sinais de controle recebidos dos módulos controladores. Após a simulação do evento estes módulos respondem, enviando informações de estado dos dispositivos, que será usado na visualização.

- **Módulos de Visualização:** representarão em um terminal de vídeo as simulações dos movimentos realizados pelos dispositivos físicos. São controlados pelos módulos simuladores que indicarão aos módulos de visualização quais os movimentos que devem ser realizados.

A estrutura lógica do programa Lis da célula flexível é representada na figura 5.2 (a). O grafo G de dependências potenciais do sistema célula flexível é ilustrado em 5.2 (b)

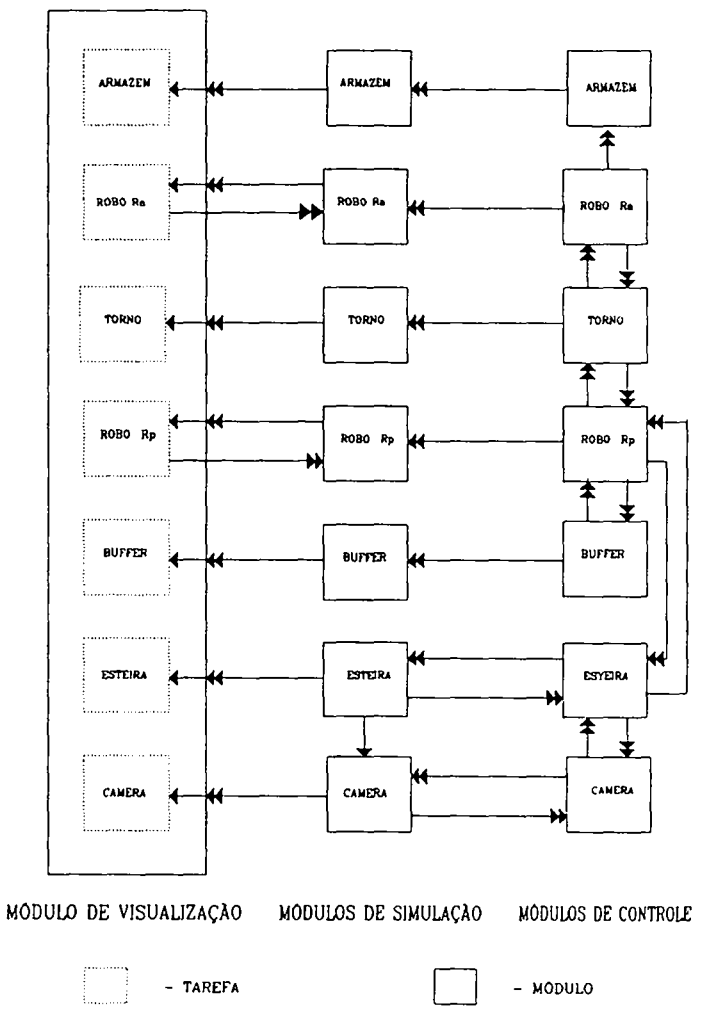

Figura 5.2 (a) Estrutura Lógica da Célula Flexível

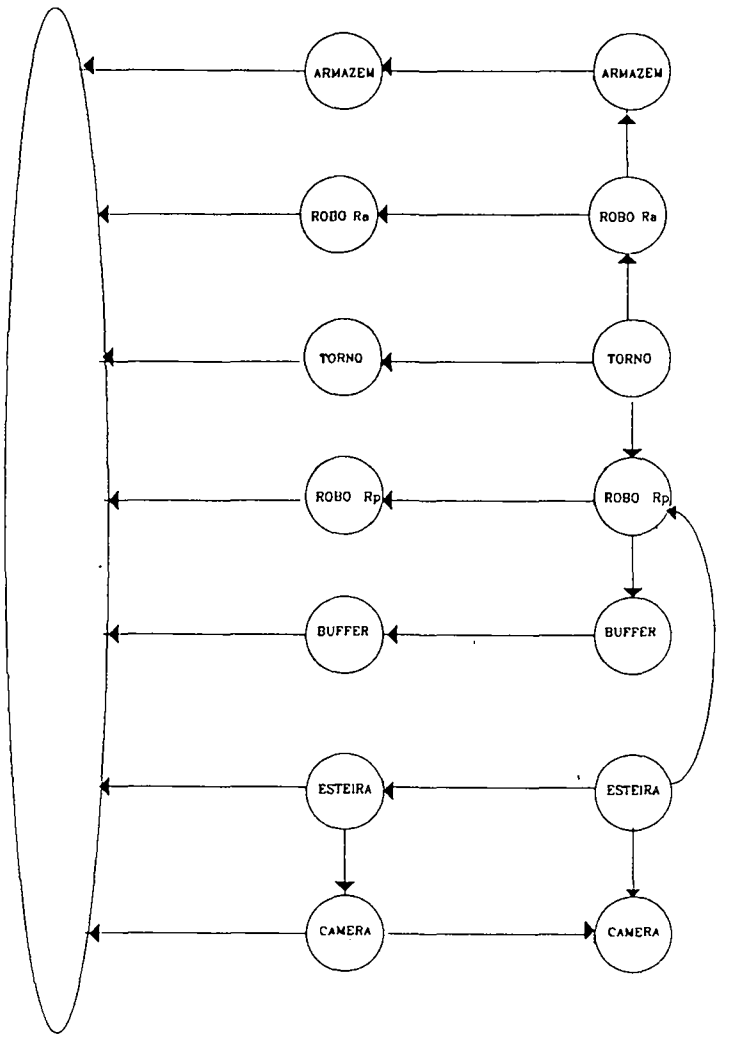

Figura 5.2 (b) Grafo de Dependências da Célula flexível

Identificação de Módulos Passíveis de Operações de Configuração

Neste exemplo podemos ter um conjunto de módulos que poderiam estar sujeitos à operações de declarações da CHANGE. As razões poderiam, por exemplo, ser operacionais ou ainda no sentido de previsão de expansão da célula. Esta célula foi assumida como programável para diferentes tipos de peças. Neste

caso, podemos prever mudanças ligadas a aspectos operacionais, reprogramando a célula, ou seja, se esta célula está usinando peças do tipo A, a programação para usinagem de peças do tipo B pode ser feita através de uma declaração CHANGE.

Estas mudanças de reprogramação envolvem seguramente o módulo controlador torno (algoritmos diferentes para peças diferentes) e ainda, os módulos controladores robôs Rp e Ra em situações que os tipos de peças diferem em dimensão ou na rigidez dos materiais.

Outro motivo de mudança operacional previsível é a retirada de um dos robôs da célula para a manutenção ou devido a uma falha. Nesta situação, os serviços de uma célula podem ser mantidos, com o robô restante assumindo as funções do outro. Declarações CHANGE podem ser previstas tanto para retirada dos robôs, como para o retorno dos mesmos à célula.

Neste exemplo portanto, identificaremos os módulos controladores de torno, Robôs Rp e Ra como os que podem ser submetidos a operações de configuração.

Identificação das Zonas de Dependência

- **Módulo Controlador de Torno:** a zona de dependência do módulo controlador do torno se restringe ao próprio módulo (zona de dependência zero) devido ao fato que este módulo é somente iniciador de transações (figura 5.3 (a)).

- **Módulo Controlador do Robô Ra:** uma zona de baixa dependência pode ser caracterizada sobre este módulo, pois somente o módulo controlador de torno inicia transações sobre o controlador do robô Ra. Operações de modificação sobre o módulo controlador do robô Ra só serão sentidos no controlador do torno (figura 5.3 (b)).

- **Módulo Controlador do Robô Rp:** o módulo controlador do robô Rp é um módulo centralizador das ações da célula. Uma operação de mudança sobre este módulo implica em reflexos sobre quase toda a célula. Isto pode ser verificado através do grafo da figura 5.3 (c), onde os módulos controladores de torno e esteira são iniciadores de transação sobre o controlador de Rp. Uma declaração de mudanças que envolvesse uma modificação no módulo controlador do Robô Rp implicaria na parada destes outros dois módulos. A parada destes módulos

implicaria na não Iniciação de novas transações com os outros módulos do sistema, parando por conseqüência toda a aplicação.

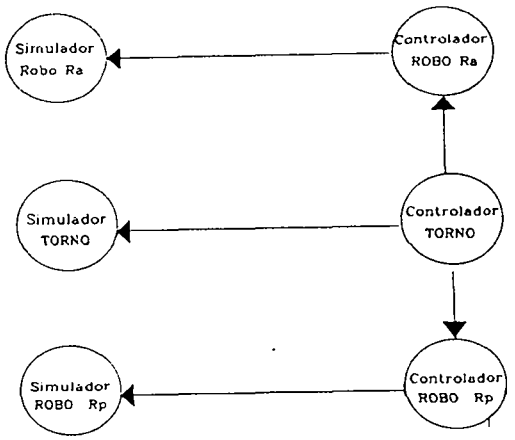

Figura 5.3 (a) TORNO

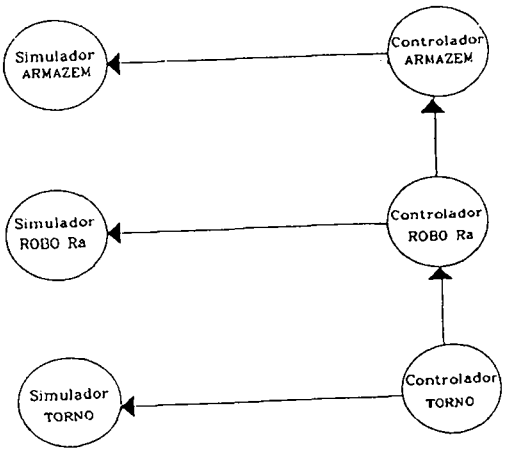

Figura 5.3 (b) ROBO Ra

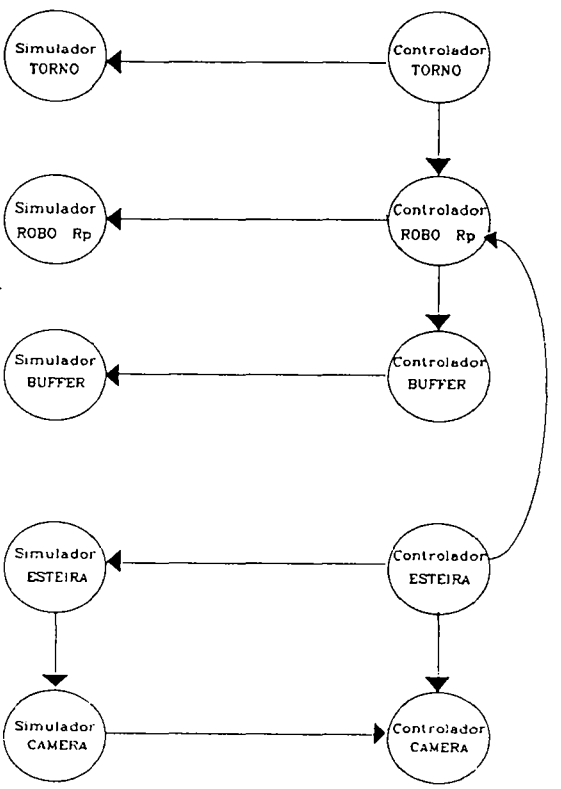

Figura 5.3 (c) ROBO Rp

Figura 5.3 Zonas de Dependência

Programação dos Mecanismos de Manutenção de Consistência

Identificado a zona de dependência de cada módulo sujeito a operações de modificação deve-se determinar as operações necessárias para tornar este módulo quiescente (item 3.4.5). O módulo para atingir este estado não deve responder a transações e nem iniciar novas transações. Além do aspecto das zonas de dependência é necessário introduzir "pontos de sincronismo" nos códigos LINCE de todos os módulos iniciadores de transação que serão "parados" numa especificação CHANGE.

A utilização de uma primitiva SINC colocada estrategicamente pelo programador possibilita a parada do módulo em um instante que não existam transações iniciadas pelo mesmo. Na figura 5.4 é mostrada um segmento de código onde aparece o mecanismo de sincronização no módulo controle do torno.

A partir desta etapa os projetos detalhados dos módulos LINCE estarão prontos. A programação da célula dependerá da construção da declaração SYSTEM.

A quiescência do **módulo controlador de robô Rp** é garantida através da parada de todos os módulos que fazem parte da sua zona de dependência.

```
Module controle_torno;
use interf;
PORT
                    {declaracao dos portos do modulo}
TASK cont_torno <2048>,10;
    VAR { declaracao das variaveis locais }
    PORT { declaracao dos portos das tarefas}
    BEGIN
        SEND sinal TO TCT5; { indica torno livre }
        LOOP
            RECEIVE peca FROM TCT1; {robo RP -> peca torno}
            SEND peca TO TCT2 WAIT sinal; {simula usinagem}
            SEND peca TO TCT4; {torno -> peca Robo Ra}
            RECEIVE sinal FROM TCT3; {robo pegou peca}
            SINC
                {procedimento para desligar torno}
            END; {sinc}
            SEND sinal TO TCT5; {indica torno livre}
        END; {Loop}
    END; {BEGIN}
LINK
    {ligacao entre os portos do modulo e tarefa}
ENDMODULE.
```

Figura 5.4 Ponto de Sincronismo do Torno

5.3.3 Operações de Mudança

Neste ítem são abordados as operações de mudança necessárias para evolução dos componentes Torno, Robô Ra.

Operações de Mudanças no Torno

As operações de mudança no módulo controlador do torno tem como objetivo possibilitar a troca do tipo de usinagem. Para a reprogramação da usinagem, a seguinte declaração CHANGE deve ser executada:

```
CHANGE CÉLULA;

    USE     cont_novo_torno;
```

```
CREATE AT STATION [3] :

                    cont_novo_torno;

STOP                cont_torno;

UNLINK              cont_torno;

LINK                cont_novo_torno;

START               cont_novo_torno;

DELETE              cont_torno;

REMOVE              cont_torno;

END.
```

Nesta especificação a declaração CREATE implica no carregamento de um novo tipo módulo (cont_novo_torno) e a instanciação do mesmo. A operação de parada resultante da declaração STOP possibilita que as outras operações de mudança sejam realizadas com segurança. Neste ponto o módulo está em quiescência. Um aspecto importante a ser observado é a seqüência nas declarações de mudança.

As operações devem ser executadas em tal ordem pois, se ocorrer algum problema na execução das operações intermediárias (CREATE, STOP, LINK, UNLINK) sempre será possível reconstruir o sistema, recuperando os estados das instâncias na configuração anterior. Isto se explica pela execução por último das declarações DELETE e REMOVE que destroem os estados e os segmentos de dados das instâncias a serem retiradas.

Operações de mudança no Robô Ra

Operações de mudança no Robô Ra são resultantes da seguinte declaração CHANGE:

```
CHANGE CÉLULA;

        USE                     cont_novo_Ra

                CREATE AT STATION [5]:

                                cont_novo_Ra;

                STOP            cont_torno;

                STOP            cont_Ra;

                UNLINK          cont_Ra;

                LINK            cont_novo_Ra;

                START           cont_novo_Ra;

                START           cont_torno;

                DELETE          cont_Ra;

                REMOVE          cont_Ra;

END.
```

Neste caso, a declaração CHANGE envolve um critério um pouco mais crítico na sua elaboração. O módulo possui uma zona de baixa dependência, necessitando então da parada de outros módulos para atingir o seu estado de quiescência. Como no exemplo anterior, inicialmente são realizadas operações de carregamento do novo tipo módulo (novo_cont_Ra) e a criação da instância. A quiescência do módulo controlador do robô Ra é conseguida através da parada do módulo controlador de torno. Após atingido o estado de quiescência, as operações de mudança UNLINK e LINK são executadas, ocasionando as desconexões e conexões necessárias. As operações de iniciação resultantes das declarações START devem ativar os controladores na ordem inversa de suas dependências.

As declarações SYSTEM e CHANGE citadas neste capítulo referentes a programação da célula flexível estão apresentados no apêndice B.

5.4 Conclusão

A utilização da metodologia para concepção de programas distribuídos apresentada neste capítulo, tem como principal objetivo auxiliar o projetista no desenvolvimento de programas evolutivos.

A determinação da zonas de dependência através de grafos, mostrou-se uma ferramenta útil no estabelecimento de estratégias nos CHANGEs. Outro aspecto importante, está relacionado com a correta especificação das declarações CHANGE. A ordenação das operações de parada na zona de dependência deve seguir o sentido de orientação dos arcos nestas zonas. As operações de partida ao contrário, devem seguir uma ordem de execução no sentido inverso das orientações dos arcos.

A utilização de uma metodologia que estabeleça a priori as zonas de dependência dos módulos e a programação de mecanismos LINCE para manutenção de consistência, não se limita à programação de CHANGEs pré-planejados. Qualquer mudança futura pode ser definida através de novos CHANGEs. A correta utilização desta metodologia torna o sistema mais robusto, diminuindo a probabilidade de geração de inconsistências.

CAPÍTULO 6
CONCLUSÃO

A presente dissertação tratou do projeto e da implementação de um suporte de configuração dinâmica para o ambiente ADES. Para atender os objetivos de mudanças dinâmicas em aplicativos distribuídos, vários mecanismos foram então concebidos. Pode se citar entre eles, um conjunto de gerenciadores que controlam em tempo de execução a evolução da configuração do sistema.

A preocupação com a consistência da aplicação distribuída levou à definição de estratégias e a introdução de novos mecanismos na linguagem LIS. Os pontos de sincronização são exemplos destes mecanismos.

O modo de utilização do suporte de configuração foi também uma das preocupações neste trabalho. Uma metodologia foi introduzida no sentido de orientar o projetista no desenvolvimento de programas flexíveis para aplicações distribuídas.

A utilização da metodologia e mecanismos em aplicações experimentais tem comprovado a eficiência destas propostas. Os objetivos traçados inicialmente para a configuração dinâmica foram em grande parte alcançados principalmente devido a adoção do modelo básico de programação da linguagem LIS.

Um aspecto que não foi explorado no decorrer deste documento é o desempenho dos mecanismos implementados. Pode-se afirmar que o tempo de execução de uma declaração CHANGE é limitado pela carga dos tipos módulos envolvidos, contudo, em alguns casos poderia-se carregar previamente os tipos módulos nas estações de execução. Mudanças que não tratam este tipo de operação são executados rapidamente.

A nível de continuidade deste trabalho prevê-se que a metodologia proposta no capítulo 5 pode ser automatizada. Uma vez determinada pelo programador os

módulos e as relações de dependência da aplicação distribuída (geração do grafo de dependências), a inspeção destes grafos para determinação das zonas de dependência e módulos que devem ter pontos de sincronismo (SINC) incluídos em seus códigos podem ser executados de forma automatizada.

Também estão sendo previstos a utilização de uma ferramenta automatizada, que auxilie o programador na geração da declaração CHANGE, a partir da análise do grafo de dependências.

O ambiente ADES se encontra atualmente na forma de um protótipo operacional para um conjunto de estações PCxt, interconectados via uma rede local. Nesta implementação as ferramentas atendem os aspectos de configuração estática e dinâmica. Uma revisão desta primeira versão está em curso, visando principalmente o transporte do ambiente ADES para uma estação de trabalho que disponha do sistema UNIX. Desta forma será acrescida no ambiente ADES um conjunto de ferramentas UNIX que permitirão um melhor desenvolvimento e teste do software aplicativo.

BIBLIOGRAFIA

[**Anderson,81**]: Anderson, W ; Lee, P. "Fault Tolerance Principles and Practice". Prentice International, 81.

[**Bal,89**]: Bal, E. H; Steiner G. J.; Tanenbaum S. A. "Programming Languages for Distributed Computing Systems", ACM Computing Surveys, vol. 21, No. 3, September 1989.

[**DeRemer,76**]: DeRemer F.; Kron, H.H. "Programming-in-the-Large Versus Programming-in-the-Small", IEEE Trans. Software Engineering, vol. SE-2, N.2, pp 80-86, June 1976.

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

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

[**Gien,89**] : Gien, M. "Architecture des Rystèmes Répartis - Une Nouvelle Génération d'UNIX", Actes Du Séminaire Franco-Brésilien Sur Les Systèmes Informatiques Répartis, Florianópolis, SC, Brasil, September 89.

[**Jino,89**]: Jino, M.; Carvalho, M. B.; Traina, C. Jr. "Automação do Desenvolvimento de Software", SBA: Controle de Automação, vol 1, No. 2, pp 99-109, 89.

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

[**Kim,88**]: Kim, K.H. ; Yoon, J.C. "Approaches to Implementation of a Repairable Distributed Recovery Block Scheme". 18th International Symposium On Fault Tolerant Computing, Pittsburg, Juin 1988.

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

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

[**Kramer,88**]: Kramer, J.; Magee J.; "A Model for Change Management", IEEE, Distributed Computing Systems in the '90s, Hong Kong, September 1988.

[**Lesk,82**]: Lesk, M. E. ; Schimidt, E., LEX - A Lexical Analyzer Generator. Bell Laboratories, Aug. 1982.

[**Liskov,83**]: Liskov, B.; Herlihy, M. "Issue in Process and Communication Structure for Distributed Programs", III Symp. IEEE on Realibility in Distributed Software and Data Base System, pp. 123-132, 1983.

[**Maccabe,85**]: Maccabe, A.B. - "Language Feature for Fully Distributed Processing Systems", Georgia Institute of Techonology, GIT-ICS-82/12, Aug 1985.

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

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

[**Narayanaswany,87**]: Narayanaswany, K. et all. "Maintainig Configurations of Envolving Software Systems", IEEE Trans. on Software Engineering, vol S.E. 13, N. 3, pp 324-334, March 87.

[**Parnas,72**]: Parnas, D. L. - "On the Criteria To Be Used in Decomposing Systems Into Modules", Communications of the ACM, pp 1053-1058, Dec. 1972.

[**Ramammoorthy,86**] Ramammoorthy, C. V. "Programming in the Large", IEEE Trans. Software Engineering, vol SE-12, N.7, pp 769-783, July 1986.

[**Rodrigues,89**]: Rodrigues, V. "Modelos para Programação Distribuída Tolerante à Faltas". Nota interna LCMI-UFSC, Florianópolis, Agosto 89.

[**Schneidewind,87**]: Schneidewind, N. F. "The State of Software Maintence", IEEE Trans. Software Engineering, vol SE-13, N.3, pp 303-310, March 1987.

[**Simpson,79**]: Simpson, H.R.; Jackson K. "Process Synchronization in Mascot", The Computer Journal, vol. 22, N.4, pp 332-345, Nov. 79.

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

[**Sloman,86**]: Sloman, M.; Kramer, J.; Magee, J. "The CONIC Toolkit for Building Distributed Systems", 4.SBRC, Recife, Março 1986.

[**Sloman,86**]: Sloman, M.; Kramer, J. "Distributed System and Computer Networks", Printice - Hall International (UK) Ltd, 1987.

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

[Sweet,85]: Sweet, R.E. "The Mesa Programming Enviroment", SIGPLAN NOTICES, 85.

[Weinstock,89]: Weinstock, C. B. "Fault Tolerance in Durva", Software Engineering Institute, Cornegie Mellon University, Pittsburg, pp 15213, 89

APÊNDICE A
SINTAXE DA LINGUAGEM

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.

Declaração de Constantes

Sintaxe:

decl_const := CONST def_const ";" [def_const ";"]*

def_const := id_const "=" valor

Declaração de Famílias

Sintaxe:

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

decl_faixa : = id_fam ":" faixa

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

limite : = inteiro_positivo | id_const

Declaração de Portos

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 : = interior_positivo

ext_porto : = "["int_positivo"]" | "("SB")" | "("SS")"

tipo_msg : = (real | integer | boolean | char | signaltype | ident)

Declaração de Contexto

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]*

Declaração de Retirada de Contexto

Sintaxe:

decl_contexto : = REMOVE decl_tipo

decl_tipo := [LOCALIZA] decl_contexto

LOCALIZA := [FROM [":"]] STATION id_estação ":"

id_estação := "[" inteiro_positivo "]"

decl_contexto := contexto ";" [contexto ";"]*

contexto := id_tipo_mod

Declaração de Instanciação

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 ")"

Declaração de Destruição de Instanciação

Sintaxe:

decl_instância := DELETE decl_insta

decl_insta := lista_decl_insta

lista_decl_insta := instãcia ";" [instância ";"]*

instância := [lista_insta]

lista_insta := nome_instância [";" nome_instância]*

nome_instância : = id_insta [família]

família : = "[" id_fam "]"

Declaração de Conexões de Portos

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_port_dest : = porto_sai_grupo | porto_entr_insta

porto_sai_grupo : = nome_porto

porto_entr_insta : = nome_instância "." nome_porto

Declaração de Desconexões de Portos

Sintaxe:

decl_link : = UNLINK 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_port_dest : = porto_sai_grupo | porto_entr_insta

porto_sai_grupo : = nome_porto

porto_entr_insta : = nome_instância "." nome_porto

Declaração de Iniciação de Instâncias

Sintaxe:

```
decl_inicio        := START decl_insta
decl_insta         := nome_instância ";" [nome_instância]*
nome_instância     := id_insta [ família ]
família            := "[" id_família "]"
```

Declaração de Parada de Instâncias

Sintaxe:

```
decl_inicio        := STOP decl_insta
decl_insta         := nome_instância ";" [nome_instância]*
nome_instância     := id_insta [ família ] [sincronismo]
sincronismo        := "(" S | N ")"
família            := "[" id_família "]"
```

Especificação GROUP MODULE

Sintaxe:

```
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)
```

Especificação SYSTEM

Sintaxe:

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

corpo_esp : =   [ decl_const ]* |

                                    [ decl_fam ]* |
                                    [ decl_ctexto ]+ |
                                    [ decl_instância ]+ |
                                    [ decl_link ]+
```

Especificação CHANGE

Sintaxe:

```
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 ]*
```

APÊNDICE B
DECLARAÇÃO SYSTEM E CHANGE DO EXEMPLO

Declaração SYSTEM

SYSTEM CÉLULA_FLEXÍVEL;

```
USE

        OCIOSA;
        GEREN;GENLINK;GENMOD;JAN;ERROCON;GAMR;{suporte configuracao dinamica}
        INTVIDEO;INTARQ;INTECLA; {interface c/ S.O.}
        CONT_EST; CONT_CAM; CONT_Rp; CONT_BUF;
        CONT_EST; CONT_Ra;CONT_TRN; {modulos controladores}
        SIMULA1; {modulo de simulacao}
        visual; { modulo de visualizacao}
        REDARQ;VISAUX;{modulos servidores de arquivo remoto}
        GSERV1; {grupo servidor de comunicacao remoto}

CREATE AT STATION [0]: serv0 : gserv1(0);

        ocio0:ociosa;
        geren(true)<1>;
        genlink1:genlink<2>;
        genmod1:genmod<4>;
        genload;
        JAN;
        errocom;
        redarq;
```

```
        intarq;
        intecla;
        intvideo;

CREATE AT STATION [3]: serv3 : gserv1(3);

        ocio3:ociosa;
        visual;
        visaux;
        cont_trn;
        gamr3   :gamr;
        genlink3:genlink<2>;
        genmod3 :genmod<4>;

CREATE AT STATION [5]: serv5 : gserv1(5);

        ocio5:ociosa;
        cont_est;
        cont_Rp;
        cont_buf;
        simula1;
        cont_Ra;
        cont_arm;
        cont_cam;
        gamr5   :gamr;
        genlink5:genlink<2>;
        genmod5 :genmod<4>;

LINK

        intecla.PTECLA  TO  cont_est.MCE1;
        cont_est.MCE2   TO  simula1.MSE3;
        cont_est.MCE8   TO  simula1.MSE3;
        simula1.MSE1    TO  cont_est.MCE3;
        simula1.MSE2    TO  cont_est.MCE6;
        cont_est.MCE4   TO  cont_cam.MCC1;
        cont_cam.MCC4   TO  cont_est.MCE5;
```

```
cont_cam.MCC2  TO  simula1.MSC1;
simula1.MSC3   TO  cont_cam.MCC3;
cont_est.MCE13 TO  cont_Rp.MCR6;
cont_Rp.MCR7   TO  cont_est.MCE12;
cont_trn.MCT5  TO  cont_Rp.MCR5;
cont_Rp.MCR4   TO  cont_trn.MCT1;
cont_Rp.MCR1   TO  cont_buf.MCB1;
cont_Rp.MCR2   TO  cont_buf.MCB2;
cont_Rp.MCR3   TO  simula1.MSA1;
cont_trn.MCT4  TO  cont_Ra.MCB1;
cont_Ra.MCB4   TO  cont_trn.MCT3;
cont_trn.MCT2  TO  simula1.MST1;
cont_Ra.MCB2   TO  simula1.MSB1;
cont_Ra.MCB3   TO  cont_arm.MCAR1;
cont_arm.MCAR2 TO  simula1.MSAR1;

visual.MABRE_ARQ    TO visaux.MABRE_ARQ;
visual.MLE_ARQ      TO visaux.MLE_ARQ;
visual.MFECHA_ARQ   TO visaux.MFECHA_ARQ;
visaux.MABRE_REM    TO redarq.MABRE_REM;
visaux.MFECHA_REM   TO redarq.MFECHA_REM;
visaux.MLE_BUFFER   TO redarq.MLE_BUFFER;
redarq.PABRE_ARQUIVO TO intarq.PABRE_ARQUIVO;
redarq.PLE_ARQUIVO  TO intarq.PLE_ARQUIVO;
redarq.PFECHA_ARQUIVO TO intarq.PFECHA_ARQUIVO;
redarq.PVISAUX      TO visaux.PVISAUX;

simula1.MSA2  TO visual.MMOVR1;
visual.MMOVR1A TO simula1.MSA3;
simula1.MSB2  TO visual.MMOVR2;
visual.MMOVR2B TO simula1.MSB3;
simula1.MSE6  TO visual.MGPECA;
simula1.MSC4  TO visual.MGCAM;
simula1.MST2  TO visual.MGTOR;

geren.PVIDEO          TO  intvideo.PINTER_VIDEO;
```

```
        geren.MNOME_ARQ      TO   jan.MNOME_ARQ;
        geren.PINTER         TO   intarq.PINT_ARQ;
        geren.PGLINK         TO   genlink1.PGLINK;
        geren.PGMOD_CREATE   TO   genmod1.PGMOD_CREATE;
        geren.PGMOD_DELETE   TO   genmod1.PGMOD_DELETE;
        geren.PGMOD_START    TO   genmod1.PGMOD_START;
        geren.PLOADER        TO   genload.PLOADER;
        geren.PERRO          TO   errocon.PERRO;
        genload.PINT_LOAD    TO   intarq.PINT_LOAD;
        genload.PABRE_ARQUIVO TO  intarq.PABRE_ARQUIVO;
        genload.PFECHA_ARQUIVO TO intarq.PFECHA_ARQUIVO;
        genload.PLE_ARQUIVO  TO   intarq.PLE_ARQUIVO;
        genload.PG_LOAD      TO   genmod1.PGMOD_LOAD;
        jan.PVIDEO           TO   intvideo.PINTER_VIDEO;
        intecla.PTECLA       TO   geren.PTECLADO;
        intecla.PTECLA       TO   jan.MTECLA;
        errocon.PVIDEO       TO   intvideo.PINTER_VIDEO;
END.
```

Declaração CHANGE

Declaração de mudanças no Torno

```
CHANGE CELULA;    {Operacao de mudanca no Torno}

USE novo_trn;

CREATE AT STATION [3]:

novo_trn;

STOP cont_trn(S); { parada Sincrona na instancia}

UNLINK

        cont_Ra.MCR4   FROM cont_trn.MCT1;
        cont_trn.MCT5  FROM cont_Ra.MCR5;
```

```
        cont_trn.MCT4  FROM  cont_Rp.MCB1;
        cont_Rp.MCB4   FROM  cont_trn.MCT3;
        cont_trn.MCT2  FROM  simula1.MST1;

LINK

        cont_Ra.MCR4   TO  novo_trn.MCT1;
        novo_trn.MCT5  TO  cont_Ra.MCR5;
        novo_trn.MCT4  TO  cont_Rp.MCB1;
        cont_Rp.MCB4   TO  novo_trn.MCT3;
        novo_trn.MCT2  TO  simula1.MST1;

START  novo_trn;

DELETE  AT STATION [3]: cont_trn;

REMOVE  FROM STATION [3]: cont_trn;

END.
```

Declaração de mudanças no Robô Rp

```
CHANGE  CELULA;    {Operacao de mudanca no Robo_Rp}

USE novo_Rp;

STOP novo_trn(S),cont_Rp(N);

CREATE AT STATION [5]:

        novo_Rp;

UNLINK

        novo_trn.MCT4  FROM  cont_Rp.MCB1;
        cont_Rp.MCB4   FROM  novo_trn.MCT3;
        cont_Rp.MCB2   FROM  simula1.MSB1;
        cont_Rp.MCB3   FROM  cont_arm.MCAR1;
```

```
LINK

        novo_trn.MCT4  TO  novo_Rp.MCB1;
        novo_Rp.MCB4   TO  novo_trn.MCT3;
        novo_Rp.MCB2   TO  simula1.MSB1;
        novo_Rp.MCB3   TO  cont_arm.MCAR1;

START  novo_Rp,

        novo_trn;

DELETE AT STATION  [5] : cont_Rp;

REMOVE FROM STATION [5] : cont_Rp;

END.
```