UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA
DEPARTAMENTO DE COMPUTAÇÃO E AUTOMAÇÃO

# PROTOTIPAGEM E IMPLEMENTAÇÃO DE ESPECIFICAÇÕES LOTOS UTILIZANDO UM AMBIENTE PARA DESENVOLVIMENTO DE SISTEMAS DE TEMPO REAL



AUTOR: Eílson Barbosa Medeiros
ORIENTADOR: Dr. Maurício Ferreira Magalhães

Dissertação apresentada à Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de MESTRE em ENGENHARIA ELÉTRICA.

Campinas - SP
Abril de 1991

*Dedico este trabalho ao meu avô Pedro Barbosa, e aos meus sobrinhos Tiago, Marina e Rebeca.*

# AGRADECIMENTOS

À **Deus**, pela vida, pelo Seu amor e pelo Seu cuidado.

Aos meus pais, Manoel Marcelino e Edna, pelo amor, apoio e dedicação.

Aos meus irmãos Hilzanir, Marcos e Eudes, pelo amor, companherismo e amizade.

À minha namorada, Sara, pelo amor, carinho e paciência.

Ao meu orientador Maurício, pela orientação e acompanhamento no desenvolvimento deste trabalho.

Aos amigos Otacílio e Rogério, pela convivência e amizade.

À Lúcia pelos desenhos e ao Rosywan, Andrea e Cristina, pela digitação e composição.

À todos que com suas súplicas apresentaram-se a Deus em intercessão por mim.

À todos Colegas, Professores e Funcionários, que direta ou indiretamente contribuíram para realização deste trabalho.

# RESUMO

O processo de projeto e realização de sistemas é uma tarefa não trivial principalmente quando consideramos o desenvolvimento de sistemas complexos.

Uma proposta que tem sido recentemente discutida é a definição de uma metodologia baseada em técnicas de descrição formais, para suportar as diversas etapas do processo de projeto e realização de sistemas.

Segundo esta metodologia, inicialmente serão trabalhadas as características arquiteturais do sistema. Numa segunda fase, a partir da arquitetura definida na fase arquitetural, serão detalhados os aspectos voltados à realização do sistema. De acordo com esta metodologia uma vez tendo sido obtida uma arquitetura suficientemente expressiva do sistema, a especificação que descreve esta arquitetura deverá ser traduzida para uma especificação em um ambiente de implementação para que os aspectos relativos à realização possam então ser explorados.

Neste trabalho consideramos as questões envolvidas no procedimento de tradução. Uma metodologia de mapeamento é discutida e implementada. Utilizamos, de forma particular, LOTOS como linguagem de especificação na fase arquitetural e o ambiente de implementação de sistemas de tempo real - STER - como ambiente de desenvolvimento da fase de realização. Um exemplo é elaborado para avaliar a viabilidade da metodologia proposta.

# Índice

# Capítulo 1

# Introdução

# 1 - INTRODUÇÃO

O contínuo aumento da complexidade dos sistemas de software tem indicado para a importância da utilização de métodos formais, em substituição aos métodos informais[1], durante as diversas fases do ciclo de desenvolvimento de sistemas [COHEN 86],[TURNER 89]. Os métodos informais têm se mostrado insuficientes para a descrição de um sistema complexo sem a introdução de erros, ambiguidades e inconsistências [MEYER 85],[MENDES 88].

A principal vantagem da utilização de técnicas formais é que o rigor sintático e semântico em que são definidas favorece o desenvolvimento de especificações com propriedades raramente obtidas com as técnicas informais (p.ex.: concisão, precisão, clareza, ausência de ambiguidades). Além destas características, devido ao modelo matemático bem definido em que se baseiam as técnicas formais, a verificação e a análise (completude, consistência e conformidade) de uma especificação podem ser feitas de modo formal [VISSERS 83],[MEYER 85],[WING 90],[HALL 90].

Apesar das deficiências das técnicas informais, é importante ressaltar que a descrição formal de um sistema pode ser bastante enriquecida com a inclusão de textos informais na forma de comentários adicionais ou especificação de aspectos não abordados pela descrição formal [MEYER 85],[PIRES 89],[BOGAARDS 89].

Uma técnica de descrição formal pode ser usada em várias fases do ciclo de desenvolvimento de um sistema - desde que os elementos do sistema possam ser sintática e semanticamente descritos pela técnica. As técnicas de descrição formal favorecem ainda a construção de ferramentas de auxílio, tais como: editores dirigidos por sintaxe, analisadores de sintaxe e semântica estática, interpretadores, simuladores, etc. Devido, principalmente, a estas características, tem sido bastante discutida a definição de metodologias de projeto baseadas em técnicas formais [TURNER 89],[BOGAARDS 89],[HAEBERER 89],[KEMMERER 90],[PIRES 90].

---

[1] Por métodos informais refere-se tanto às técnicas informais quanto às técnicas semi-formais [MEYER 85],[COHEN 86],[MENDES 88].

L. Pires [PIRES 89] apresenta as diversas fases do projeto de sistemas segundo um modelo baseado em técnicas formais (fig. 1.1). O ciclo de desenvolvimento é dividido em três fases: fase arquitetural, fase de implementação e fase de realização[2].

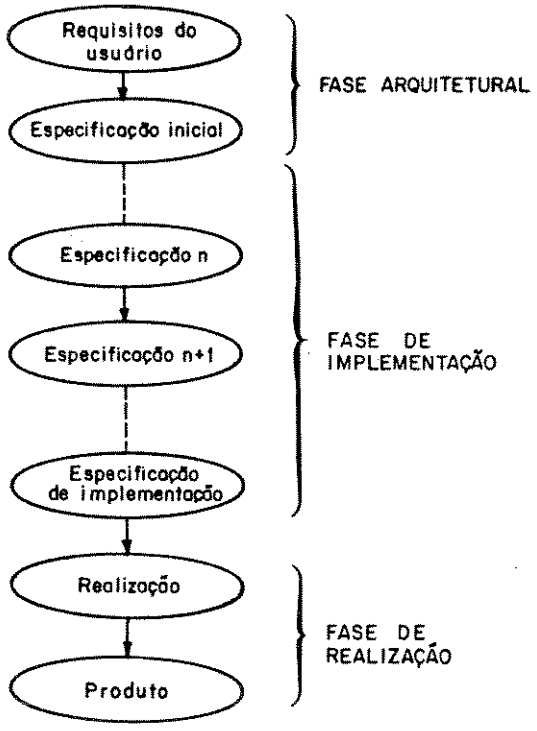



Figura 1.1 - Modelo para o projeto de sistemas

Na fase arquitetural, os requisitos do usuário [PRESMANN 87], na forma de idéias vagas, são agrupadas ordenadamente em um texto informal. Este texto será transformado em uma especificação inicial do sistema, descrito por uma técnica formal. Esta primeira especificação juntamente com os requisitos não formalizados (descritos informalmente) definem a arquitetura do sistema.

Na fase de implementação, a especificação inicial será refinada passo-a-passo (através de especificações intermediárias) até ser obtida uma especificação em um nível de abstração tal, que possa ser facilmente mapeada em uma realização (especi-

[2] Em [PIRES 90] é definida uma metodologia para o projeto de sistemas por refinamentos sucessivos utilizando LOTOS como técnica de descrição formal.

ficação de implementação). Em cada passo uma especificação (em um nível de abstração *n*) é transformada em uma especifiacação mais detalhada (em um nível de abstração *n+1*) pela inclusão de novos aspectos do sistema e/ou remoção, ou isolamento, de indeterminismos [PIRES 90].

A última fase será a de realização do sistema. Nesta fase a especificação de implementação será mapeada para um ambiente de implementação onde deverão ser tomadas várias decisões (técnicas ou de mercado) que definam as características finais do produto.

Uma das maiores lacunas que tem sido deixada em toda esta discussão da utilização de técnicas formais no processo de projeto de sistema, é a análise mais concistente da derivação de realizações finais do sistema a partir da sua especificação formal. Não se deve perder de vista que a utilização de uma técnica formal só terá sentido prático, se o objetivo final do processo de projeto for favorecido, ou seja, se as questões de realização forem convenientemente abordadas.

No caso específico de LOTOS, devido às suas características de alto nível de abstração, aparecem sérias limitações quanto à representação dos elementos mais concretos necessários para a descrição dos aspectos de realização do sistema.

Neste sentido vários trabalhos têm sido propostos para expandir LOTOS de modo a incluir aspectos desta natureza (p.ex.: tempo [HULZEN 89],[QUEMADA 89]). No entanto, esta solução ainda é parcial, pois certamente novas expansões se farão necessárias para inclusão de outras propriedades fundamentais para o projeto e realização de alguns sistemas (p.ex: prioridade). A inclusão destas propriedades na semântica de LOTOS aumentará em demasia a complexidade do seu modelo matemático.

Uma outra proposta é a utilização de LOTOS, na sua forma padrão, segundo uma metodologia definida (p.ex.: [PIRES 90]) durante as diversas fases do desenvolvimento do projeto, até um ponto em que os refinamentos sucessivos não estejam introduzindo características adicionais relevantes [PIRES 90]. Neste ponto a especificação LOTOS final (especificação de implementação - fig.1.2) será mapeada para um ambiente de implementação (PARLOG [GILBERT 87], MODULA 2 [FERNADEZ 88], C [MIGUEL 87], [NOMURA 90], [EIJK 90]), onde os demais aspectos relativos à realização do sistema possam então ser considerados (fig. 1.2).

É neste contexto que se insere este trabalho. Nele será proposta uma metodo-

logia de mapeamento de uma especificação LOTOS para uma aplicação em um ambiente de implementação de sistemas de tempo real (STER) - mapeamento, fig.1.2 -, onde questões de tempo e prioridade possam ser consideradas. Entretanto, neste trabalho será definida apenas a metodologia, sendo que uma análise mais consistente da consideração dos aspectos de tempo e prioridade deixados para trabalhos futuros.

O texto está organizado na seguinte forma: no capítulo 2 será apresentada, de forma geral, a técnica de descrição LOTOS; o capítulo 3 apresenta uma introdução ao ambiente STER; o capítulo 4 apresentará a metodologia definida para a tradução de uma especificação LOTOS em uma aplicação STER; no capítulo 5 serão descritos os mecanismos criados para permitir esta tradução; no capítulo 6 um exemplo de tradução será amplamente discutido; o capítulo 7 apresenta as conclusões do trabalho e algumas sujestões para trabalhos futuros.

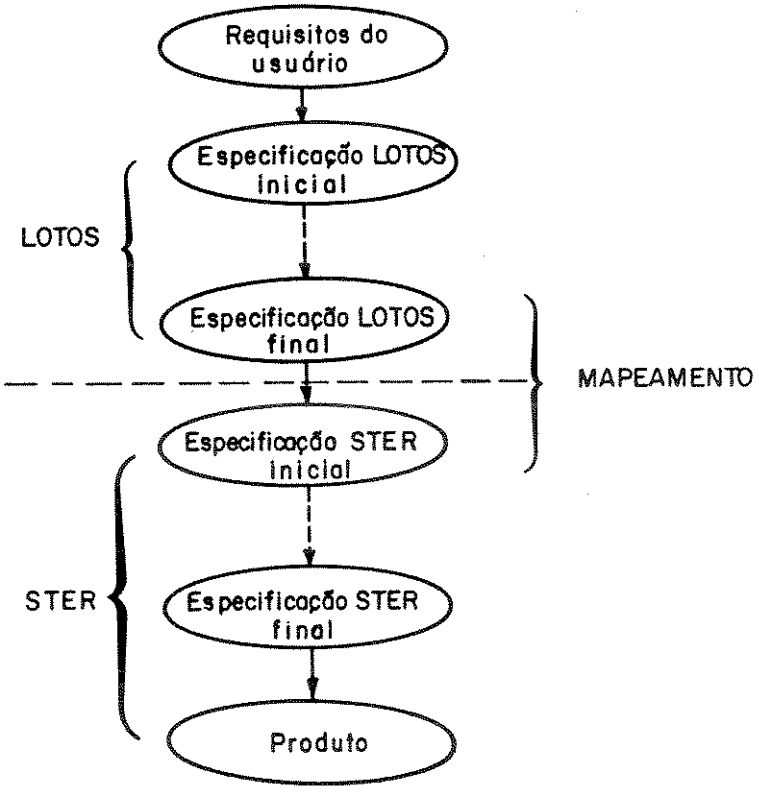



Figura 1.2 - Metodologia e realização de sistemas

# Capítulo 2

# Linguagem de especificação LOTOS

## INTRODUÇÃO

A linguagem de especificação LOTOS (Language Of Temporal Ordering Specification) é uma técnica de descrição formal desenvolvida pela ISO [ISO 8807] para a especificação formal de sistemas distribuídos e, em particular, aqueles relacionados ao modelo OSI.

Um sistema é especificado em LOTOS em duas componentes distintas. A primeira é relativa à descrição do comportamento do sistema, e baseia-se na idéia de que um sistema pode ser especificado pela definição da relação temporal entre as interações que descrevem o seu comportamento externo. Este modelo foi introduzido por Robin Milner em seu "Calculus of Communicating Systems" - CCS [MILNER 80],[WALKER 88]. A segunda componente trata da descrição da estrutura de dados e utiliza uma variante da linguagem ACT-ONE [EHRIG 85], que é uma técnica para descrição de tipos de dados abstratos (ADT) [LEDGARD 76],[GUTTAG 86].

As principais características da linguagem LOTOS são:

.**Expressividade** - Capacidade de expressar um grande número de propriedades inerentes aos sistemas distribuídos, principalmente no ambiente OSI;

.**Definição formal** - Permite, além da descrição não ambigua, a validação e derivação de testes formais relativos à especificação;

.**Abstração** - São considerados apenas os aspectos arquiteturais relevantes ao sistema, sendo desprezados detalhes relativos à implementação;

.**Estruturação** - Permite especificar os sistemas de acordo com diversas abordagens arquiteturais (estilos de especificação) [VISSERS 88].

Embora LOTOS tenha sido desenvolvida inicialmente para ser usada na especificação formal dos serviços [QUEMADA 86],[TURNER 86],[TOCHER 86],[ISO 88a],[ISO 88b],[BARRÉ 89] e protocolos [SCOLLO 85a],[ISO 88c],[ISO 89],[ISO 90] do modelo OSI/ISO, algumas experiências têm mostrado ser a sua aplicação extensiva a toda área de especificação de sistemas distribuídos. A primeira aplicação não associada ao ambiente OSI foi o projeto de um protocolo de aplicação para o controle de experimentos em física nuclear usando a rede de aquisição de dados MONDAN [SCOLLO 85b]. Outros projetos surgiram, como por exemplo: [CARCHIOLO 86] que apresenta a especificação LOTOS dos serviços do projeto PROWAY (sistema de comunicação padrão IEC para aplicação em controle de processos); [PUENTE 86] descreve a experiência na especificação formal de um

sistema de controle em tempo real para aplicação industrial; [BIEMANS 86] usa LOTOS para especificar a arquitetura de uma célula de manufatura flexível em um ambiente de manufatura integrada por computador (CIM); [FERNANDEZ 88] desenvolve arquiteturas de protocolos para um sistema de multi-processamento em tempo real (Projeto PRODAT).

Na seção 2.1 serão apresentados os conceitos fundamentais utilizados na definição de LOTOS.

A descrição de LOTOS será dividida em três partes: na seção 2.2 será apresentado LOTOS básico; na seção 2.3 será apresentado ACT-ONE e, finalmente, na seção 2.4 serão consideradas as duas partes (comportamento e estrutura de dados) na descrição de LOTOS completo.

Como o principal interesse que se terá nesse capítulo é a introdução de LOTOS, serão apresentadas apenas as carcterísticas básicas da linguagem, sendo que um estudo mais consistente deve ser reportado para [VISSERS 87a],[BOLOGNESI 87],[ ISO 8807].

## 2.1 - Conceitos Fundamentais.

Um **Sistema** é visto em LOTOS como um processo que pode ser composto por vários sub-processos. Um sub-processo nada mais é do que um processo definido a partir de um outro existente. Assim, um sistema é especificado em LOTOS pela definição hierárquica de Processos.

Pode-se imaginar um **Processo** como uma caixa preta (fig 2.1.1) que é capaz de interagir com outros processos - que formam o seu ambiente - através de pontos de interação (portas), ou de realizar ações internas que não são observadas pelo ambiente. Uma interação é também chamada de **evento** ou, simplesmente, **ação**. O elemento básico do comportamento de um processo é o evento, que representa a sincronização entre processos (dois ou mais).

Um **Evento** implica em sincronização porque os processos que interagem participam dele ao mesmo tempo. Uma sincronização pode ser com ou sem troca de dados. Um evento é atômico, pois ocorre instantaneamente (sem interrupção). Um processo participa no máximo de um evento em um dado instante.

O **Ambiente** de um processo é composto dos demais processos pertencentes ao sistema com os quais interage, mais um processo observador - não especificado - sempre pronto a participar de algum evento observável do sistema. Uma observação, do ponto de vista dos conceitos básicos de LOTOS, nada mais é do que uma interação que ocorre entre o sistema e um observador.

Um processo LOTOS é definido na forma:

```
Process nome_do_processo [g1,...,gn](V1:t1,...,Vn:tn):=
        <Expressão de comportamento>
 where
        <Definição de processos>
endproc
```

A componente essencial na definição de um processo é a **expressão de comportamento** (EC). Uma EC é construída compondo-se duas outras EC's por meio de um operador LOTOS. Uma EC pode incluir a criação de instâncias de outros processos, sendo que a definição destes processos deve aparecer na cláusula **where**. Diz-se que um processo, ou uma EC, está na **forma normal**, ou **monolítica**, quando é definida por uma sequência de eventos ou pela escolha de uma, entre várias, destas sequências.

Na fig. 2.1.1 é apresentada na forma simbólica de caixas-pretas, a definição de dois processos: $P_1$ e $P_2$.

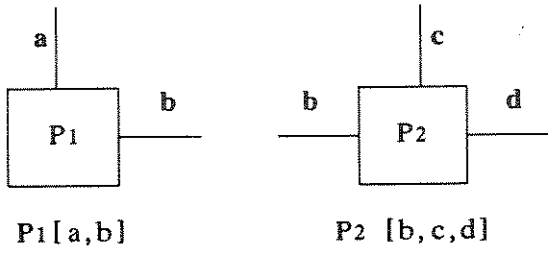



Figura 2.1.1 - Representação na forma de caixas-pretas dos processsos P1 E P2.

$P_1$ é um processo capaz de realizar ações observáveis através de ofertas de eventos nas portas **a** e **b**. $P_2$, de modo idêntico, apresenta ofertas ao seu ambiente nas portas **b**, **c** e **d**.

A composição dos (sub-) processos $P_1$ e $P_2$ por meio de um operador LOTOS, **op**, na forma:

$$P0 := P1\ [a,b]\ op\ P2\ [b,c,d],$$

define um novo processo, $P_0$ (fig. 2.1.2). $P_0$ pode sincronizar com seu ambiente através das portas **a**, **b**, **c**, ou **d**.

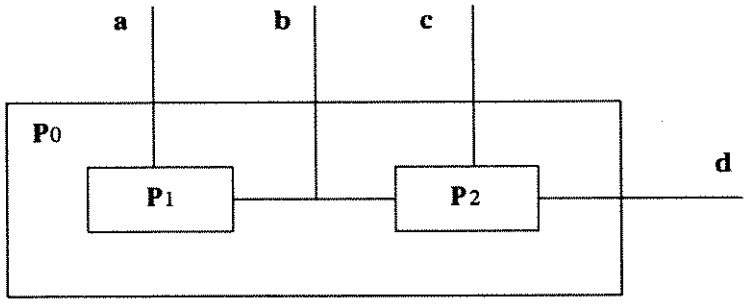



Figura 2.1.2 - Representação do processo Po
(Po := P1[a,b] op P2[b,c,d]).

O comportamento de um processo pode evoluir por meio de um evento em uma porta não definida externamente. Isto implica que o ambiente não pode participar deste evento. Este tipo de evento é chamado de evento **interno**.

Na nova definição do processo $P_0$ (fig. 2.1.3), o ambiente só observa eventos ocorridos em **a**, **c** ou **d**. Os eventos em **b** passam a ser internos a $P_0$. Deste modo, o comportamento de $P_0$ evoluirá através de um evento interno ou de uma interação com o ambiente em uma das portas **a**, **c** ou **d**.

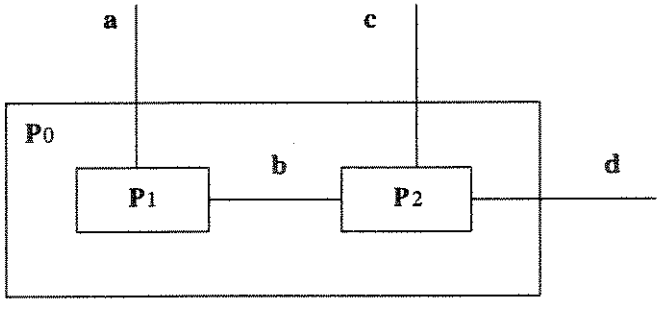



Figura 2.1.3 - Definição de processo com
restrição de porta.

Em LOTOS, as expressões de comportamento podem ser decompostas em expressões mais simples através de **transições rotuladas**.

Ex.: Suponhamos que o processo **P**, após participar de um evento **e**, transforma-se em P'. Este fato pode ser mostrado pela transição:

$$P \text{ —— } e \longrightarrow P'$$

A semântica formal dos operadores LOTOS é apresentada na forma de axiomas e de regras de inferência.

**Axiomas** são premissas consideradas como válidas na avaliação realizada por um raciocínio. **Inferência** é o raciocínio em sí.

Ex.:

$$\left.\begin{array}{l} x = y \text{ e } y = z \end{array}\right\} \longrightarrow \text{Axiomas ou Premissas}$$

$$\left.\begin{array}{ll} \text{Se} & x = y \text{ e } y = z \\ \text{Então} & x = z \end{array}\right\} \longrightarrow \text{Inferência ou Raciocínio}$$

Em LOTOS, as regras de inferência são escritas na forma:

$$\frac{P_1, \ldots, P_n}{C} \quad ,$$

e significa que se as premissas ($P_1,\ldots,P_n$) são válidas, então C - que é a conclusão do raciocínio - é também válida, ou seja, diz-se que C é inferida a partir de $P_1,\ldots,P_n$.

Ex.: Seja a regra

$$\frac{B1 \text{ — } a \longrightarrow B1'}{B \text{ — } a \longrightarrow B1'}$$

A interpretação para a regra é: se B1 transformar-se em B1' pela ocorrência do evento a (axioma), então B se transformará em B1' pela ocorrência de a (inferência).

## 2.2 - LOTOS BÁSICO.

Em LOTOS básico os eventos são identificados apenas pelo nome da porta onde são ofertados. As interações são tratadas como simples sincronizações de processos, sem que seja realizado algum tipo de negociação de valores entre eles.

A seguir será apresentado cada um dos operadores LOTOS (tabela 2.1). Na definição das regras semânticas dos operadores serão usadas as seguintes definições:

.B, B', B1, B1',... $\longrightarrow$ Define expressões de comportamento;

.G $\longrightarrow$ Denota o universo de portas que podem ser usadas na definição das expressões de comportamento (portas úteis);

.A $\longrightarrow$ Um sub-conjunto qualquer de G;

.g,g1,...,gn $\longrightarrow$ São portas definidas em G;

.i $\longrightarrow$ Descreve uma ação não observável (evento interno);

.ACT $\longrightarrow$ Conjunto de eventos definidos em G $\cup$ {i};

.$\mu$ $\longrightarrow$ Um evento qualquer em ACT;

.$\delta$ $\longrightarrow$ Pseudo evento que marca a terminação de um processo;

.$G^+$ $\longrightarrow$ G $\cup$ {$\delta$};

.$g^+$ $\longrightarrow$ Um evento qualquer em $G^+$;

.$ACT^+$ $\longrightarrow$ ACT $\cup$ {$\delta$};

.$\mu^+$ $\longrightarrow$ Um evento qualquer em $ACT^+$.

| NOME | SINTAXE |
|---|---|
| inação | STOP |
| sequência de ações | |
| . interna (não observável) | i ; B |
| . observável | g ; B |
| escolha | B1 [ ] B2 |
| paralelismo | |
| . com sincronização restrita | B1 \|[ g1,...,gn]\| B2 |
| . sem sincronização (interleaving) | B1 \|\|\| B2 |
| . com sincronização completa | B1 \|\| B2 |
| escondimento de portas | hide g1,...,gn in B |
| instanciação de processos | P[g1,...,gn] |
| terminação com sucesso | EXIT |
| composição sequencial | B1 >> B2 |
| desabilitação | B1 [> B2 |

Tabela 2.1 - Sintaxe de expressões de comportamento em LOTOS básico.

. **Inação** $\longrightarrow$ B := **stop**

O processo B é incapaz de participar de um evento com seu ambiente ou de realizar uma transição não observável (evento interno).

Não existem axiomas nem regras de inferência associados a este operador.

. **Sequência de ações** $\longrightarrow$ B := g;B1 ou B := i;B1

O processo B oferta evento na porta **g**. Se este evento ocorrer, B passa a ser equivalente a B1. B poderá evoluir para um novo estado (B1) por meio de uma transição interna.

A regra de inferência para este operador é:

$$\mu;B \xrightarrow{\mu} B$$

. **Escolha** $\longrightarrow$ B := B1 [] B2

Dependendo do evento oferecido pelo ambiente, B se comportará como B1 ou B2. Se o ambiente oferece um evento de B1 (B2), B poderá ser equivalente a B1 (B2). No caso em que o ambiente oferta indistintamente um evento de B1 ou B2, a escolha será não determinística.

Este comportamento é definido por duas regras de inferência, que são:

$$\frac{B1 \xrightarrow{\mu^+} B1'}{B1 \;[]\; B2 \xrightarrow{\mu^+} B1'}$$

$$\frac{B2 \xrightarrow{\mu^+} B2'}{B1 \;[]\; B2 \xrightarrow{\mu^+} B2'}$$

. **Composição Paralela**

.. **Paralelismo com Sincronização Restrita** $\longrightarrow$ B := B1 |[A]| B2

$B_1$ e $B_2$ interagem independentemente com o ambiente, exceto para eventos em uma porta pertencente a **A**. Neste caso, B só participa de um evento com o ambiente se $B_1$ e $B_2$ participarem.

As seguintes regras de inferência definem a semântica deste operador:

$$\frac{B_1 \xrightarrow{\mu} B_1'}{B_1 |[A]| B_2 \xrightarrow{\mu} B_1' |[A]| B_2} \quad , \mu \notin A$$

$$\frac{B_2 \xrightarrow{\mu} B_2'}{B_1 |[A]| B_2 \xrightarrow{\mu} B_1 |[A]| B_2'} \quad , \mu \notin A$$

$$\frac{B_1 \xrightarrow{g^+} B_1',\ B_2 \xrightarrow{g^+} B_2'}{B_1 |[A]| B_2 \xrightarrow{g^+} B_1' |[A]| B_2'} \quad , g^+ \in A \cup \{ \delta \}$$

**.. Paralelismo sem Sincronização** $\longrightarrow B := B1|||B2$

Este caso é equivalente ao anterior para um conjunto vazio de portas de sincronização ($|||$ ≡ $|\ []\ |$).

As regras de inferência para este operador são:

$$\frac{B_1 \xrightarrow{\mu} B_1'}{B_1 \ ||| \ B_2 \xrightarrow{\mu} B_1' \ ||| \ B_2}$$

$$\frac{B_2 \xrightarrow{\mu} B_2'}{B_1 \ ||| \ B_2 \xrightarrow{\mu} B_1 \ ||| \ B_2'}$$

**.. Paralelismo com Sincronização Completa** $\longrightarrow B := B1||B2$

Considerando, para o caso de paralelismo com sincronização restrita, o conjunto de portas de sincronização como sendo igual ao conjunto de portas úteis ($A = G \Rightarrow |[A]| = ||$), as EC's devem sincronizar-se em todos os eventos.

A regra de inferência para este caso é:

$$\frac{B1 \text{ —} g^{+} \longrightarrow B1',\ B2 \text{ —} g^{+} \longrightarrow B2'}{B1 \ ||\ B2 \text{ —} g^{+} \longrightarrow B1' \ ||\ B2'} \quad , g^{+} \in A \cup \{ \delta \}$$

. **Escondimento de portas** $\longrightarrow$ hide A in B

O operador **hide** permite que ações observáveis de um processo sejam transformadas em eventos internos. Isto implica que o ambiente não poderá participar das interações ocorridas no conjunto de portas escondidas.

As regras de inferência para este operador são:

$$\frac{B \text{ —} \mu^{+} \longrightarrow B'}{\textbf{hide } g1,...,gn \textbf{ in } B \text{ —} \mu^{+} \longrightarrow \textbf{hide } g1,...,gn \textbf{ in } B'}, \ \mu^{+} \notin \{g1,...,gn\}$$

$$\frac{B \text{ —} g \longrightarrow B'}{\textbf{hide } g1,...,gn \textbf{ in } B \text{ —} i \longrightarrow \textbf{hide } g1,...,gn \textbf{ in } B'}, \ g \in \{g1,...,gn\}$$

. **Instanciação de Processos**

A instanciação de um processo LOTOS assemelha-se à invocação de um procedimento em uma linguagem de programação, (p.ex.: PASCAL). A instanciação implica na criação de instância de um processo que **deve** ser definido em algum lugar na especificação. Na instância criada, a lista de portas formais é substituída por uma lista real.

O mecanismo de substituição é definido formalmente por meio de um operador auxiliar - chamado renomeação - que não faz parte da **sintaxe** LOTOS, tendo no entanto significado na **semântica dinâmica.**

A renomeação é uma operação unária pós-fixa que consiste na lista de pares de porta [g1/g1',..., gn/gn'] e é interpretada por:

a porta gi' é renomeada para gi, i := 1,...,n em todo evento ofertado em gi'.

As regras de inferência que definem a semântica do operador de renomeação de portas são:

$$\text{i)}\ \frac{B \xrightarrow{\ g\ } B'}{B\phi \xrightarrow{\ g\ } B'\phi}\ ,\quad \phi = [g_1/g_1', \ldots, g_n/g_n']\ \text{ e } g/g' \in \phi$$

$$\text{ii)}\ \frac{B \xrightarrow{\ \mu\ } B'}{B\phi \xrightarrow{\ \mu^+\ } B'\phi}\ ,\ \mu^+ \notin \{g1' \ldots, gn'\}$$

É importante notar de (ii) que eventos internos e a terminação com sucesso não são afetados pela renomeação.

A regra de inferência para a instânciação de processos é:

Se **Process** P $[g_1', \ldots, g_n']$ := $B_p$ **endproc** é uma definição de processo, então:

$$\frac{B_p\ [g_1/g_1', \ldots,\ g_n/g_n'] \xrightarrow{\ \mu^+\ } B'}{P\ [g_1, \ldots,\ g_n] \xrightarrow{\ \mu^+\ } B'}$$

Comportamentos recursivos são definidos em LOTOS através da instanciação de processos, onde um processo cria diretamente uma instância sua ou, indiretamente, onde um processo instanciado cria uma instância do processo que o instanciou. A recursão é usada para se criar comportamentos infinitos.

. **Composição sequencial e terminação com sucesso** $\longrightarrow$ B := B1 >> B2

Com o objetivo de expressar mais claramente a estrutura do sistema na especificação, foi definido o operador de composição sequencial. A idéia deste operador é que um segundo processo é habilitado somente quando o primeiro processo terminar com sucesso - **exit** -.

Considerando a natureza recursiva inerente à definição de um processo LOTOS, em que um processo transforma-se em um outro processo após participar de um evento - notar que o operador **stop** pode ser visto como um processo incapaz de realizar alguma ação -, pode-se dizer que o operador **exit** é um processo cujo propósito é somente de realizar o evento de terminação com sucesso ($\delta$), transformando-se depois em um

processo inativo - **stop** -.

O axioma associado ao operador **exit**:

$$\textbf{exit} \;\text{—}\; \delta \longrightarrow \textbf{stop}$$

A ação $\delta$ é chamada pseudo-evento, também conhecida como pseudo-porta, e está relacionada à terminação com sucesso de um processo quando associado ao operador **exit**, assim, o uso normal de uma porta com este nome na especificação não possui nenhum significado de terminação.

As regras de inferência que descrevem a semântica do operador de composição sequêncial (>>) são:

$$\text{i)} \quad \frac{B1 \;\text{—}\; \mu \longrightarrow B1'}{B1>>B2 \;\text{—}\; \mu \longrightarrow B1'>>B2}$$

$$\text{ii)} \quad \frac{B1 \;\text{—}\; \delta \longrightarrow B1'}{B1>>B2 \;\text{—}\; i \longrightarrow B2}$$

. **Desabilitação** $\longrightarrow$ B := B1 [> B2

O operador desabilitação é útil para expressar situações onde um processo pode ser interrompido pela ocorrência de uma situação de exceção. Estas situações são típicas em sistemas de comunicação, como o caso de disconexão.

Um processo B1 pode ser desabilitado por um processo B2 de acordo com as seguintes regras:

$$\text{i)} \quad \frac{B1 \;\text{—}\; \mu \longrightarrow B1'}{B1[>B2 \;\text{—}\; \mu \longrightarrow B1'[>B2}$$

$$\text{ii)} \quad \frac{B1 \;\text{—}\; \delta \longrightarrow B1'}{B1[>B2 \;\text{—}\; \delta \longrightarrow B1'}$$

$$\text{iii)} \quad \frac{B2 \;\text{—}\; \mu^{+} \longrightarrow B2'}{B1[>B2 \;\text{—}\; \mu^{+} \longrightarrow B2'}$$

O processo $B_1$ pode (iii) ou não (i,ii) ser interrompido pela ocorrência do primeiro evento de $B_2$. No primeiro caso, o controle é transferido do processo interrompido $B_1$ para o processo interruptor $B_2$. No segundo caso, o processo interrompível $B_1$ participa de um evento. Se o evento não for uma terminação com sucesso (i), $B_2$ continua a ser um processo interruptor, caso contrário (ii) $B_2$ deixa de existir, ou seja, como o processo que $B_2$ esperava interromper terminou, então ele mesmo é desabilitado.

**. Não-Determinismo e eventos internos**

A descrição de comportamentos não determinísticos favorece o nível de abstração representado em especificações LOTOS:

Ex.: seja a EC

```
B :=   a; b; stop
    [] c; d; stop
```

Caso o ambiente de B ofereça somente um dos eventos **a** ou **c**, a escolha será determinística. No caso de serem oferecidos os dois eventos, não será possivel determinar qual dos ramos será o escolhido.

Um caso específico deste exemplo é:

```
B :=   a; b; stop
    [] a; c; stop
```

onde o resultado da observação do evento **a** não é determinado.

Um outro tipo de não determinismo pode ser modelado pelo evento interno (i).

Ex.: seja a EC

```
B :=   i; a; stop
    [] b; c; stop
```

Como o evento interno ocorre independente da influência do ambiente, a escolha entre os ramos é apenas parcialmente determinável (no caso da escolha do evento **b**).

No exemplo anterior, o ambiente ainda possui alguma influência na escolha. O caso em que o ambiente não pode influir de modo algum, é quando todos os ramos da EC inicia com um evento interno.

Ex.:

```
B :=   i; a; stop
    [] i; b; stop
```

## 2.3 - Tipos de Dados

Em LOTOS foi adotada a linguagem de especificação de tipos de dados abstratos ACT-ONE [EHRIG 85] para representação de valores, expressões de valores e estruturas de dados.

A ACT-ONE apresenta as seguintes características:

.Uso de biblioteca;

.Extensão e combinação de especificações existentes;

.Parametrização de especificações e instanciação de especificações parametrizadas;

.Renomeação de especificações.

O modo mais elementar de especificação de um tipo de dado em LOTOS consiste de uma identificação (signature) e possivelmente de uma lista de equações.

**. Identificação (signature)**

A **identificação** implica na definição de nomes para os objetos de dados **(data carrier)** e para as operações. Os nomes dos objetos de dados são referenciados como classes **(sorts)**. A declaração de uma operação inclui um **domínio**, que consiste de uma lista de zero ou mais classes, e de uma **imagem**, que consiste de apenas uma classe.

O tipo elementar 'Nat_numbers' é especificado em LOTOS na forma:

```
type Nat_numbers is
     sorts nat
     opns zero: → nat
          succ: nat → nat
endtype
```

O nome associado ao tipo (Nat-numbers) permite que ele seja usado na definição ou combinação de outros tipos. A identificação consiste de uma classe simples - 'nat' - e das operações **zero** e **succ**. A operação **succ** é aplicada a um elemento de classe 'nat' e tem como resultado um elemento de classe 'nat'. O domínio e a imagem destas operações são compostos apenas por elementos de classe 'nat'. **zero** é uma operação sem argumentos que resulta em um elemento de classe 'nat', como indicado por ' $\longrightarrow$ nat'. Operações deste tipo são denominadas de **constantes**.

. **Equações**

O propósito de uma equação é declarar que dois termos sintáticamente diferentes expressam um mesmo valor. Isto possibilita a definição de novas operações, como exemplificado a seguir:

```
type Extended-nat-numbers is
     sorts nat
     opns zero : ⟶ nat
          succ : nat ⟶ nat
          _+_ : nat, nat ⟶ nat
     eqns
          forall x,y : nat
          ofsort nat
               x + zero      = x;
               x + succ(y)   = succ (x+y);
endtype
```

A primeira equação expressa o comportamento do operador '+' quando combinado com a constante 'zero'. A adição com números diferentes de 'zero' é definida na segunda equação.

A lista de equações é precedida por dois novos construtores: 'forall x,y:nat', que define duas variáveis - x e y - que serão usadas na definição das equações, e 'ofsort nat', que declara as classes de cada termo das equações.

. **Extensão e Combinação de Especificações de Tipos**

Para se especificar tipos com um grande número de operações podemos usar a combinação de tipos já existentes e, se necessário, estendê-los através da inclusão de novas classes, operações e equações. Os tipos combinados na extensão de um tipo são enumerados após o símbolo 'IS'.

Como exemplo consideremos a extensão do tipo 'Nat_ number' (realizada anteriormente pela reconstrução completa do tipo) a partir da definição original:

```
type Extended-nat-numbers is nat-numbers
     (* sorts *)
     opns _+_ : nat, nat → nat
     eqns
          forall x,y : nat
          ofsort nat
               x + zero    = x;
               x + succ(y) = succ (x+y);
endtype
```

**. Parametrização**

Algumas estruturas de dados apresentam características próprias que não dependem dos valores ou tipos de dados que são manipulados. Para evitar duplicações desnecessárias sempre que novas características não essenciais forem alteradas, usa-se o recurso da parametrização de tipos. A parametrização pode ser vista como uma especificação parcial onde somente algumas características gerais do tipo são descritas, sendo outras deixadas para considerações futuras.

Um caso típico de parametrização de tipos é a definição da estrutura de dados FILA. Uma FILA pode ser de inteiros, de caracteres, etc. Se não fosse possível a parametrização de tipos, seria necessária uma definição específica para cada tipo FILA.

Na definição paramétrica de tipos, os sub-tipos variáveis são considerados como formais e serão substituídos, com a atualização, pelos elementos reais de manipulação.

A seguir apresenta-se a definição formal do tipo FILA e a sua atualização para o tipo FILA DE NATURAIS.

```
type Queue is
    formalsorts element
    formalopns e0 :  element
    sorts queue
    opns create :--> queue
         add : element, queue --> queue
         first : queue --> element
    eqns forall x,y  : element, z : queue
          ofsort element
          first (create) = e0;
          first (add(x,create)) = x;
          first (add(x,add(y,z))) = first (add(y,z));
endtype


type Nat-numbers-queue is
    Queue actualizedby Nat-numbers using
          sortnames nat  for element
          opnnames zero for e0
endtype
```

**. Renomeação de tipos**

A renomeação da especificação de um tipo é útil no caso em que um tipo já definido precise ser usado em um ambiente específico sem, no entanto, alterações em sua semântica.

A renomeação é feita explicitamente reescrevendo-se a definição de um tipo já existente usando-se novas classe e operações.

Consideremos o caso em que o tipo FILA deva ser usado para representar uma conexão em um ambiente OSI. Por meio da renomeação temos:

```
type Connection  is
    Queue renamedby
          sortnames channel for queue
                    object  for element
          opnnames send    for add
                   receive for first
endtype
```

**. Biblioteca**

Em aplicações onde é frequente o uso de alguns tipos de dados podemos definir uma biblioteca onde estes tipos encontram-se disponíveis. Isto evita a re-

definição de tipos básicos.

A forma de definição de uma biblioteca é:

```
library ... endlib
```

## 2.4 - LOTOS COMPLETO

Foi visto na seção 2.3 as características dos operadores LOTOS, onde um evento expressava unicamente a sincronização de processos. Uma vez tendo visto como definir valores, expressões de valores e estruturas de dados; é possível agora considerar um evento como uma estrutura mais refinada. A principal vantagem do LOTOS completo é a capacidade de sincronização com passagem de valores possibilitando a comunicação entre processos.

Enquanto em LOTOS básico um evento observável era definido apenas pelo nome da porta onde era ofertado, em LOTOS completo ele é formado pelo nome de uma porta seguido por uma lista de zero ou mais ofertas, na forma:

$$g\langle v_1,\ldots,v_n\rangle$$

Como exemplo, seja: g<false,'Sara',5>, que é a oferta de um evento observável onde são ofertados um valor booleano, uma cadeia de caracteres e um valor inteiro.

As regras de inferência foram apresentadas quando da descrição de LOTOS básico para não nos envolvermos aqui com detalhes em demasia. No entanto, estas regras podem ser derivadas para LOTOS completo, a partir daquelas, apenas considerando uma transição envolvendo valores, ou:

$$B_1 \xrightarrow{\quad g\langle v_1,\ldots v_n\rangle \quad} B_1'$$

A estrutura completa de um texto LOTOS onde são considerados tanto os aspectos de controle como de dado, é definido na forma:

```
specification ident.-da-espec. [lista formal de portas] (lista
                    formal de parâmetros) : funcionalidade
          < definição de tipos >
     behavior
          < expressão de comportamento >
     where
          < definição de tipos >
          < definição de processo >
endspec
```

A definição de um processo será na forma:

```
Process ident._do_proc. [lista formal de portas]
       (lista formal de parâmetros) : funcionalidade :=
          < expressão de comportamento >
where
             < definição de tipos >
             < definição de processo >
endproc
```

Pode-se notar uma diferença básica entre as duas estruturas de definição. Em uma especificação, tipos de dados podem ser definidos antes da expressão de comportamento, enquanto que em um processo isto não é possível. A razão disto se deve ao fato de que a definição de tipos em uma especificação é de caráter global e será usada na tipificação das variáveis da lista de parâmetros formais bem como pelo próprio ambiente da especificação.

A inclusão da definição de tipos em uma especificação aumenta a potencialidade de LOTOS nos seguintes aspectos:

i) Valores poderão ser ofertados em uma porta e trocados entre processos - comunicação entre processos;

ii) Valores poderão ser usados para definir condicionalidade, p.ex.: uma EC só será possível caso determinada condição seja válida;

iii) Valores poderão ser usados na definição do comportamento de escolha generalizada;

iv) Valores poderão ser usados na instanciação de processos a partir de

uma definição paramétrica, bem como na atualização de expressões de comportamento paramétricos;

v) Valores poderão ser passados como parâmetros da terminação com sucesso de um processo.

A seguir será abordado resumidamente cada uma destas características.

**. Oferta de Valores e Comunicação entre Processos**

A estrutura de uma oferta de evento em LOTOS completo é definida na forma geral:

$$g\alpha_1 \ldots \alpha_n ,$$

onde **g** é o nome de uma porta e $\alpha_i$, $i = 1,\ldots,n$ representa uma lista de atributos. Existem dois tipos de atributos: declaração de valores e declaração de variável.

**- Declaração de Valor**

Um valor é declarado na forma:

**!E**

onde:

**E** é uma expressão de valor.

Se combinarmos um atributo de declaração de valor a uma porta, sendo a expressão de valor denotada por $\nu$, dizemos que tal composição descreve uma ação representada na forma **q** <$\nu$> (q!(2+5) descreve a ação q<7>).

De um modo geral podemos enunciar o seguinte axioma:

$$g!E;\ B \text{ —— } g < \text{valor } (E) > \longrightarrow B$$

**- Declaração de Variável**

Uma variável é declarada na forma:

**? x : T**

onde:

**x** é o nome de uma variável e **T** identifica a classe da variável.

Se combinarmos o nome de uma porta a uma declaração de variável, teremos como resultado uma ação, denotada por **g < ν >**, para cada valor ν no domínio **T** (a?y:nat, descreve o conjunto de ações { a < n >| n ∈ ℕ }).

Pode-se enunciar o seguinte axioma para a declaração de variável:

$$g?x:T;\ B(x) \text{ —— } g\langle\nu\rangle \longrightarrow B(\nu)$$

onde:

ν pode ser um valor qualquer no domínio de **T** e **B(ν)** indica que após a ocorrência da transição, ν será substituído por **x** em **B(x)**.

Retornando à forma definida inicialmente para a estrutura de uma oferta ($g\alpha_1 ... \alpha_n$), podemos ainda ter a combinação dos dois tipos de atributos na forma:

**g ! E ? x : T**

A comunicação entre processos poderá ocorrer quando dois processos relacionados por um operador de paralelismo, ofertarem eventos complementares em uma das portas pertencentes ao conjunto de portas de sincronização.

Ex.:

```
    g1!sap1?x:cep_sort!'test';g2!x;stop
||  g1!sap1!cep_3?y:string;g3!y;stop
```

As duas EC's sincronizam-se em torno do evento **g1<sap1,cep_3,'test'>**. Após a interação, a expressão acima será resumida a:

```
    g2!cep_3;stop
||  g3!'test';stop
```

Os tipos de interações - definidos por ofertas complementares - são apresentados na tabela 2.2.

| Processo A | Processo B | Condição de Sincronização | Tipo de Interação | Resultado |
|---|---|---|---|---|
| $g!E_1$ | $g!E_2$ | $Valor(E_1) = Valor(E_2)$ | Casamento de valores | Sincronização com a sincronização $x=Valor(E)$ com a sincronização $X=Y=v$ sendo $v$ do tipo T |
| g!E | g?x:T | Valor(E) é do tipo T | Passaagem de valor | |
| g?x:T | g?Y:U | T = U | Geração de valor | |

Tabela 2.2 - Tipos de interação LOTOS

**. Condicionalidade**

Uma condição deve ser expressa na forma de equação ou de expressão de valor, do tipo booleano, na qual são envolvidos parâmetros do evento estruturado ou variáveis locais e/ou globais do processo.

Existem duas formas de se expressar comportamentos condicionais em LOTOS: predicado de seleção e guarda.

**. Predicado de seleção**

O predicado de seleção determina as condições que devem ser satisfeitas no instante da ocorrência da interação. Isto implica na definição de restrições quanto aos valores negociados em uma sincronização.

Ex.: Seja a EC

B:=a?x:nat[x<3]; b!x; stop

serão possíveis as seguintes transições:

B — a<0> ⟶ b!0; **stop** ou

B — a<1> ⟶ b!1; **stop** ou

B — a<2> ⟶ b!2; **stop**

. **Guarda** ⟶ B := [cond] ⟶ $B_1$

Uma EC pode ser precedida por uma guarda, implicando que se cond = true, B se comporta como $B_1$. Caso contrário, B é equivalente a **stop**.

Ex.: Seja a EC

```
B :=  [x>3] ⟶ a!(x+1);stop
    [] [x<2] ⟶ b!x;stop
```

As seguintes transições são possíveis:

.se x=4 então B — a<5> ⟶ **stop**;

.se x=1 então B — b<1> ⟶ **stop**;

.se x estiver no intervalo [2,3] então B é equivalente a **stop**.

. **Escolha e paralelismo generalizado**

Com o operador escolha ([]) é possível expressar apenas um número finito de alternativas. O operador escolha generalizado (choice ...[]...) permite descrever comportamentos onde são envolvidos infinitas alternativas.

Ex.:
```
Choice x:nat[]a!x;stop
```
é equivalente a:

```
[] a!0;stop
[] a!1;stop
[] a!2;stop
    .
    .
    .
```

De modo idêntico, é possível generalizar a composição paralela em LOTOS.

Ex.:A expressão

```
Par g in [a,b], h in [d,e] |[f]| proc[f,g,h] (1024)
```

é equivalente a:

```
        Proc[f,a,d] (1024)
|[f]|   Proc[f,b,d] (1024)
[f]     Proc[f,a,e] (1024)
|[f]|   Proc[f,b,e] (1024)
```

**. Processos parametrizados**

Um processo pode ser definido de forma paramétrica seguindo a sintaxe:

```
process P [lista de portas formais](lista de
        parâmetros formais):funcionalidade:=
    ...
endproc
```

Na criação de uma instância do processo as listas formais são substituídas pelas listas reais. O procedimento de substituição implica na definição de valores para as variávies livres.

Ex.: Seja a definição de processo:

```
process P[g1,g2] (x1:int,x2:int,x3:int): exit:=
      g1!x1;g2!x2;g2!(x1+x2+x3); exit
endproc

P[a,b](zero,succ(zero),succ(succ(zero))
```

cria uma instância do processo P ocorrendo a seguinte renomeação:

```
[a/g1,b/g2] (zero/x1,succ(zero)/x2,succ(succ(zero))/x3)
```

Uma maneira mais direta de associação de valores a variáveis livres é através do operador **let**, que possui a sintaxe:

**let** $x_1:t_1 = E_1,...,x_n:t_n = E_n$ **in** $B(x_1,...,x_n)$

Ex.: a instância anterior poderia ter sido definida na forma:

```
process P[g1,g2]: exit :=
     let x1:int=zero,x2:int=succ(zero),
         x3:int=succ(succ(zero))
     in g1!x1;g2!x2;g2!(x1+x2+x3); exit
endproc
```

### . Composição sequencial com passagem de valores

Em LOTOS básico vimos como compor sequencialmente dois processos através do operador de habilitação (>>). Agora, com a possibilidade de se expressar valores, será muito útil a passagem de parâmetros do processo habilitador para o habilitado. Para isso serão necessárias: a generalização do conceito de terminação com sucesso; a extensão das características da linguagem com respeito à composição sequencial e paralela; e algumas restrições estáticas à linguagem.

#### .. Terminação com sucesso com passagem de valores

No LOTOS básico o **exit** era usado para expressar a terminação com sucesso de um processo. Agora, associado a ele, existirá uma lista de valores que representará os parâmetros passados para o processo habilitado.

```
Ex.:tsap!cei
        ?quality_of_service:quality_para-
                                meter_sort
        ?expedict_data_option:bool
    ;exit(quality_of_service,bool)
```

A lista de classes de valores ofertada na terminação com sucesso é chamada **funcionalidade**. No exemplo, a funcionalidade da expressão é <quality_of_service, bool>

#### .. Funcionalidade de expressões de comportamen-to

Na composição sequencial, o número e a classe dos valores que são passados em uma terminação com sucesso devem ser conhecidos. Isto implica que o universo das possíveis terminações do primeiro processo deve possuir a mesma funcionalidade. Algumas regras são necessárias para se determinar a funcionalidade de expressões de comportamento, bem como algumas restrições devem ser impostas no modo em que processos ou ECs com diferentes funcionalidades podem ser compostos.

- **stop** ⟶ A funcionalidade de um processo que não termina com sucesso é indicada por **noexit**.

func(stop) = noexit

- **exit** ⟶ A funcionalidade de um processo que termina com sucesso e sem passagem de parâmetros é indicada por **exit**.

func(exit) = exit

- **exit** $(x_1,...,x_n)$ ⟶ A funcionalidade de um processo que termina com sucesso e com passagem de parâmetros, é a lista das classes dos parâmetros passados.

func(**exit**$(x_1,...,x_n)$) = <classe de $x_1$,...,classe de $x_n$>.

- **Sequência de ações** ⟶ A funcionalidade de uma expressão não é afetada quando a ela é pré-fixada (;) uma ação.

func(a;B) = func(B)

- **Escolha** ⟶ Se $B_1$ e $B_2$ são processos que terminam com sucesso, então a funcionalidade de $B_1[]B_2$ só poderá ser definida se for imposta a restrição de que $B_1$ e $B_2$ possuam a mesma funcionalidade, que por sua vez será a funcionalidade da expressão, ou que um dos processos possuam funcionalidade **noexit**. Assim $B_1$ e $B_2$ só poderão ser combinados na forma $B_1[]B_2$ se:

.func($B_1$) = func($B_2$) = func($B_1[]B_2$);
.func($B_1$) = **noexit** e
    func($B_1[]B_2$) = func($B_2$);
.func($B_2$) = **noexit** e
    func($B_1[]B_2$) = func($B_1$);

- **Escolha generalizada** ⟶ A funcionalidade do operador de escolha generalizada é dada por:

.func(Choice...[] B') = func(B')

- **Desabilitação** ⟶ Este caso é idêntico ao discutido para o operador escolha. As restrições para composição de expressões através do operador de desabilitação são:

.func(B1) = func(B2) = func(B1[>B2);
.func(B1) = **noexit** e
func(B1[>B2) = func(B2);
.func(B2) = **noexit** e
func(B1[>B2) = func(B1);

- **Composição Paralela** ⟶ Seja **op** um dos operadores LOTOS de paralelismo (|||, || e |[A]|). Dois processos, B1 e B2, só poderão ser compostos na forma B1 op B2, se as seguin-tes condições forem satisfeitas:

.func(B1) = func(B2) = func(B1 op B2);
.func(B1) = **noexit** ou
func(B2) = **noexit** e func(B1 op B2) =
**noexit**

A composição paralela de dois processos só terá termina-ção com sucesso se ambos os processos termina-rem com a mesma lista de valores.

**.. Sintaxe da composição sequencial com passagem de valores**

Se B1 e B2 são processos e func(B1) = <t1,...,tn> então a composição sequencial de B1 e B2 é escrita na forma:

B1 >> **accept** x1:t1,...,xn:tn **in** B2

onde x1,...,xn são variáveis de B2 que recebem os valores passados por B1.

Ex.:
```
Connection_Phase[...](...)
   >> accept
        quality_of_service:quality_parameter_sort
        expedict_data_option:bool
      in
        Data_Phase[...](quality_of_service,expedited
                        _data_option,...)
```

No próximo capítulo será introduzido o ambiente de programação para sistemas de tempo real STER, que será utilizado como suporte para o desenvolvimento dos meca-nismos necessários para a tradução de especificações LOTOS para especificações de implementação.

# Capítulo 3

# Ambiente para Programação de Sistemas de Tempo Real (STER)

## INTRODUÇÃO

STER [DTIA 88] é um ambiente proposto para o desenvolvimento de software de tempo real para aplicações de controle de processos. Este modelo fornece um conjunto de ferramentas que facilitam a estruturação, produção, teste e configuração da aplicação. Suas características fundamentais são:

. Programação modular;

. Independência dos módulos em relação à arquitetura do sistema (Distribuída/Centralizada);

. Ênfase no aspecto da abstração de dados em função do uso de interfaces bem definidas nos módulos;

. Forte tipificação dos dados e estruturas;

. Possibilidade de reconfiguração dinâmica (não implementada).

O STER é baseado no sistema CONIC [KRAMER 83]. O ambiente consiste basicamente de uma metodologia orientada ao projeto de sistemas distribuídos [LOPES 86], [MAGALHÃES 86]; por duas linguagens: Linguagem para programação de módulos - LPM - (associada ao CONIC/P [KRAMER 84a]) e linguagem para configuração de módulos - LCM - (associada ao CONIC/L [KRAMER 84b]), e por um núcleo de tempo real - NTR - que dá suporte à execução da aplicação.

## 3.1 - METODOLOGIA DE DESENVOLVIMENTO

A metodologia utilizada explora o conceito de módulo, o qual mostra-se especialmente adequado aos sistemas de controle distribuído projetados para ter vida útil longa, durante a qual podem sofrer diversas modificações.

As alterações a que está sujeito um sistema de controle podem ser provocadas pela expansão do processo controlado, introdução de novas características ou ocorrência de falhas. Devido ao fato dos módulos apresentarem interfaces bem definidas, possibilita que uma expansão ou reestruturação corresponda simplesmente à inclusão, exclusão ou reconexão de módulos através de um novo conjunto de ligações entre eles. Neste sentido, o desenvolvimento de cada módulo deve ser independente da estrutura do hardware. Esta independência deve ir de um extremo onde todas as funções de controle

(módulos) estão localizadas em uma única estação, até o outro, onde cada função reside em uma estação distinta. Este fato facilita a migração de uma aplicação de um ambiente centralizado para um ambiente completamente distribuído.

Em resumo, as características básicas desta metodologia são:

. A decomposição funcional do sistema em módulos;

. Definição das interfaces dos módulos, que são constituídas por portas lógicas de entrada e saída. Uma porta é do domínio exclusivo do módulo que a define, o que permite o desenvolvimento independente dos módulos;

. Interligação das instâncias de módulos, representando a fase de configuração da aplicação. Deste modo, as funções do sistema (módulos) são compostas e definem uma função global que caracteriza o sistema como um todo.

## 3.2 - Linguagem de Programação de Módulos

A linguagem de programação de módulos (LPM) [LOPES 86], foi definida tendo como base a versão 2.4 da linguagem de programação do sistema CONIC [KRAMER 84a] e acrescenta ao PASCAL uma série de extensões que possibilitam:

. Declaração de tipos de módulos;
. Importação de definições;
. Declaração de portas de entrada e saída;
. Declaração de mensagens;
. Envio e recepção de mensagens de comunicação e sincronização;
. Mudança da prioridade de um módulo em tempo de execução;
. Suspensão temporizada de um módulo;
. Tratamento de interrupções.

Abordaremos resumidamente cada uma destas extensões devendo um estudo mais detalhado ser reportado a [LOPES 86] ou [DTIA 88].

**.Declaração de tipo de módulo**

O módulo é a maior entidade de programa que pode ser definida na linguagem LPM. O identificador que segue a palavra reservada **MODULE** é o nome do tipo do módulo que está sendo definido é será utilizado para a criação de instâncias na fase

de confguração.

```
MODULE exemplo;
        (* Importações de definições *)
        (* Declaração de portas *)
        (* Declaração de mensagens *)
        (* Bloco LPM *)
```

**.Importação de Definição**

Levando-se em consideração que a comunicação entre módulos através de troca de mensagens requer o uso de uma mesma definição nas várias declarações de portas e mensagens, é utilizado um arquivo de definição, denominado unidade de definição, onde são declarados os tipos e constantes comuns a vários módulos, exceto os tipos elementares do PASCAL.

A estrutura da unidade de definição é:

```
DEFINE arqdef;
        (* Definiçãodeconstantes*)
        (* Definiçãodetipos *)
END-DEFINE.
```

A importação de definição é feita pelo comando **USE**, na forma:

```
MODULE exemplo;
        USE arqdef.inc;
        .
        .
        .
```

**.Declaração de Portas**

A declaração de portas permite aos módulos da aplicação a realização de um trabalho cooperativo através da comunicação e sincronização pela troca de mensagens. Visando a independência do desenvolvimento dos módulos, os comandos de envio e recepção de mensagens não enderaçam diretamente outros módulos envolvidos na comunicação, mas sim portas de saída e entrada, locais a cada módulo.

A declaração de uma porta define o tipo de comunicação à qual estará associada, que pode ser:

- De modo síncrono

```
EXITPORT porta-de-saída: tipo1 REPLY tipo2
ENTRYPORT porta-de-entrada: tipo3 REPLY tipo4
```

- De modo assíncrono

```
EXITPORT porta-de-saída: tipo1
ENTRYPORT porta-de-entrada: tipo2
```

Expandindo o módulo exemplo com a declaração de portas, temos:

```
MODULE exemplo;
    USE ...;
    ENTRYPORT ...;
    EXITPORT ...;
  :
  :
  :
```

**.Declaração de Mensagens**

Em seguida serão declarados os tipos das mensagens envolvidas nas comunicações. O tipo de uma mensagem deve ser compatível com o tipo da porta onde é enviada ou recebida.

```
MODULE exemplo;
    USE ...;
    ENTRYPORT ...;
    EXITPORT ...;
    MESSAGE
        Mens1: tipo1;
        Mens2: tipo2;
          :
          :
          :
  :
  :
  :
```

**.Bloco LPM**

O bloco LPM é uma extensão de um bloco PASCAL, onde poderão ser utilizados além dos comandos, procedimentos e funções PASCAL, comandos exclusivos LPM.

```
MODULE exemplo;
      USE ...;
      ENTRYPORT ...;
      EXITPORT ...;
      MESSAGE ...;
      (* Bloco LPM *)
            (* Declaração de rótulos, constantes,
               tipos e variáveis *)
            (* Declaração de procedimentos e
               funções *)
            BEGIN-MODULE
                  (* Comandos PASCAL/LPM *)
            END-MODULE.
```

Nos pontos seguintes serão descritos os comandos LPM:

**..Comando de Ciclo Infinito**

A sequência de comandos entre **LOOP** e **END_LOOP** será executada indefinidamente. A saída do ciclo pode ser obtida através da execução de um comando **EXIT** ou por um desvio incondici-onal **GOTO**.

**..Comandos para Troca de Mensagens**

Os comandos de troca de mensagens da LPM permitem que sejam efetuadas comunicações síncronas ou assíncronas.

O envio de uma mensagem no modo assíncrono deve ser feito na forma:

```
SEND mensagem TO porta-de-saída ,
```

sendo que a mensagem deve ser do mesmo tipo da porta de saída. Notar ainda que a porta de saída deve ser do tipo assíncrona.

O envio no modo síncrono se dá na forma:

```
SENDmensagem TO porta-de-saída
    WAIT menagem-resposta
```

Neste caso, após o envio da mensagem, o módulo será suspenso até que seja recebida uma mensagem resposta. Novamente, deve haver consistência entre tipos de porta e mensagens.

Um terceiro e último comando de envio de mensagem é o modo parcialmente síncrono:

```
SEND mensagem TO porta-de-saída
     WAIT mensagem-resposta ⟶ comandos 1
     FAIL tempo ⟶ comandos 2
END_SEND
```

A sequência **comandos 1** será executada se a mensagem de resposta for recebida antes que **tempo** se esgote. Caso contrário a sequência **comandos 2** será executada.

O comando para recepção de uma mensagem é semelhante ao de envio, diferenciando-se apenas pelo sentido de comunicação.

```
RECEIVE mensagem FROM porta-de-entrada
     REPLY mensagem-resposta
```

Quando no início da execução do comando não houver uma mensagem disponível na porta de entrada, o módulo será suspenso até que chegue uma. Caso o comando de recepção inclua a cláusula **REPLY**, uma mensagem de resposta será enviada tão logo uma mensagem seja recebida.

Caso seja necessário algum tratamento sobre dados recebidos antes da resposta ser retornada, a **LPM** permite que a mensagem de resposta seja enviada pelo comando:

```
REPLY mensagem TO porta-de-entrada
```

A referência a uma porta de entrada num comando **REPLY** sem que haja ocorrido uma recepção anterior na mesma porta não terá efeito algum.

Uma mensagem recebida pode ser imediatamente enviada a uma porta de saída, sem qualquer processamento, através do comando

```
FORWARD porta-de-entrada TO porta-de-saída
```

A resposta eventualmente especificada pelo receptor final da comunicação será enviada diretamente ao emissor inicial.

A LPM permite ainda a recepção seletiva, onde o usuário pode especificar uma espera temporizada por uma mensagem a partir de um conjunto de portas. A habilitação de uma porta para efetivar a recepção de uma mensagem será determinada por uma guarda.

```
P/RSELECT
      WHEN ... (* guarda 1 *)
      RECEIVE mens. FROM pe1
            ⇒ comandos 1
OR_SELECT
      WHEN ... (* guarda 2 *)
      RECEIVE mens. FROM pe2
            ⇒ comandos 2
  :
  :
OR_SELECT
      TIMEOUT n;
            ⇒ comandos 3
  :
  :
ELSE_SELECT
      comandos 4
END_SELECT
```

São consideradas elegíveis as cláusulas não precedidas por guardas ou associadas a guardas cuja avaliação resulta no valor **True**. Caso mais de uma das portas referenciadas nos comandos de recepção das cláusulas elegíveis tenha recebido pelo menos uma mensagem, um desses comandos deve ser escolhido. A escolha será feita de acordo com o tipo de seleção. Em **PSELET** é selecionada a mensagem associada à cláusula de maior prioridade. A prioridade neste caso decresce a partir da primeira cláusula, ou seja, a primeira cláusula é a mais prioritária. No comando **RSELECT** a escolha é aleatória.

O tempo que um módulo deve esperar pela chegada de uma mensagem pode ser limitado por uma ou mais cláusulas de tempo. Uma cláusula de tempo elegível é selecionada se após esperar o número de unidade de tempo especificado (n), o módulo não tiver recebido nemhuma mensagem. Se várias cláusulas de tempo forem elegíveis é selecionada aquela que especifica a menor espera. A cláusula **ELSE** é equivalente a uma cláusula de tempo com espera zero. Quando não houver mensagens disponíveis nas portas de entrada, o uso desta cláusula impede que o módulo seja suspenso.

**..Procedimentos e Funções Pré-definidas.**

A LPM incorpora diversos procedimentos e funções pré-declaradas que podem ser usadas em conjunto com os comandos:

- PROCEDURE ABORT (porta-de-entrada, motivo-da-falha)

Usado no lugar de um comando **REPLY** para cancelar uma comunicação síncrona. O usuário pode indicar o motivo do cancelamento para o usuário par.

- PROCEDURE REASON: integer

Permite ao módulo emissor da comunicação saber a razão da falha quando do cancelamento da comunicação.

- FUNCTION LINKED (porta-de-saída): Boolean

Indica se uma porta de saída está conectada ou não.

- FUNCTION QLEN (porta-de-entrada): Integer

Determina o número de mensagens disponíveis em uma porta de entrada.

- PROCEDURE SETPRIORITY (nível-de-prioridade)

Usado para mudança de prioridade de um módulo em tempo de execução. Existem cinco níveis de prioridade: SYSTEMPR, HIGHPR, NORMALPR, LOWPR, LOWESTPR.

- PROCEDURE DELAY (período-de-tempo)

Atrasa a execução do módulo por um determinado período de tempo.

- FUNCTION TIME: Longinteger

Obtém o valor, em unidades de tempo, assinalado pelo relógio do sistema.

- FUNCTION INTALOC (vetor-físico): Integer

Mapeia uma interrupção física em uma equivalente lógica.

- PROCEDURE WAITIO (vetor-lógico)

Usado pelo módulo para indicar a espera da ocorrência de uma interrupção.

- FUNCTION MODULE-ID: Integer

Permite, em tempo de execução, obter o número inteiro usado pelo tradutor LCM para identificação do módulo.

## 3.3 - Linguagem de Configuração de Módulos

A linguagem de configuração de módulos (LCM) [LOPES 86] - baseada na versão 2.3 da linguagem de configuração do sistema CONIC - é a ferramenta oferecida ao usuário para configurar a aplicação. Um programa de configuração é formado por declarações de tipos de módulos que comporão a aplicação, pela criação das instâncias destes e pela conexão das portas das instâncias dos módulos.

O exemplo abaixo apresenta a estrutura de um programa de configuração:

```
CONFIGURAÇÃO Aplicação;
     INSTANCE
          M1,M2:Modulo 1;
             .
             .
             .
          Mn:Modulo n;
     CREATE
          M1/P=LOWPR,
          M2(10),
            .
            .
            .
          Mn/S=64/H=8;
     LINK
          M1.p1, M2.p1, M10.p1 TO M3.p1;  (* I *)
          M4.p1 TO M2.p2, M3.p4, M6.p3;   (* II *)
            .
            .
            .
          M5.p3 TO Mn.p10   (* III *)
END-CONFIG.
```

A declaração das instâncias de módulos, através da diretiva **INSTANCE**, define cópias lógicas de cada tipo de módulo, onde são associados nomes lógicos a cada instância.

Na criação de instâncias, por meio da diretiva **CREATE**, são determinados os valores dos parâmetros reais com que estas serão criadas (no caso da instância M2, o parâmetro real é 10), bem como podem ser usadas diretivas para definir prioridade inicial das instâncias (/P), e o tamanho de memória que cada uma pode usar para o o STACK (/S) e HEAP (/H).

A conexão das portas dos módulos (diretiva **LINK**) estabelece os canais lógicos de comunicação através dos quais poderão interagir. Em um comando de conexão especifica-se sempre a ligação de portas de saída a portas de entrada. Os tipos de conexão permitidos são:

. Uma porta de saída a uma porta de entrada - permitido para portas assíncronas e síncronas (III no exemplo);

. Uma porta de saída a várias portas de entrada - permitido apenas para portas assíncronas (II);

. Várias portas de saída a uma porta de entrada - permitido para portas dos dois tipos (I).

A identificação de uma porta deve ser precedida pelo nome da instância. Isto é necessário pois as portas das instâncias de um mesmo tipo de módulo possuem o mesmo nome. É ainda importante ressaltar que a conexão entre uma porta de saída e uma porta de entrada só será permitida caso ambas sejam do mesmo tipo.

## 3.4 - Suporte de Tempo Real

Um núcleo de tempo real (NTR) [COELLO 86] implementa os serviços que dão suporte, em tempo de execução, aos comandos, procedimentos e funções pré-declaradas da LPM que não fazem parte do PASCAL padrão.

O núcleo oferece:

. Configuração física dos módulos componentes da aplicação - que consiste nos serviços de carga e interconexão dos módulos -;

. Troca de mensagens entre módulos - traduzido nas primitivas de envio de uma mensagem a uma porta e recepção de uma mensagem de uma porta -;

. Controle dinâmico de execução - suportado por serviços que permitem a mudança de prioridade e suspensão temporária de módulos;

. Tratamento lógico de interrupções - que consiste em serviços que possibilitam um mapeamento do vetor físico de interrupções em um elemento do vetor lógico de interrupções -.

O esquema de escalonamento utilizado pelo núcleo é preemptivo, onde o módulo de maior prioridade em condições de executar deverá ter a posse da CPU.

O núcleo oferece, também, um conjunto de serviços indispensáveis para aplicações de tempo real, como: leitura do relógio de tempo real, cancelamento de uma comunicação síncrona, determinação da razão de uma falha, teste de conexão de uma porta de saída, e verificação de uma quantidade de mensagens enfileiradas numa porta de entrada.

# Capítulo 4

# Metodologia para o mapeamento de especificações LOTOS em aplicações STER

## INTRODUÇÃO

No capítulo 1 foi apresentada a importância da definição de uma metodologia de projeto e realização de sistemas. Na fig. 1.2 discutiu-se a importância do mapeamento da especificação LOTOS final em uma especificação inicial STER.

A questão mais importante a ser considerada na estratégia de mapeamento, é a manutenção, na especificação STER, das principais propriedades formalizadas na especificação LOTOS. A principal destas propriedades é a arquitetura do sistema.

A grande dificuldade do mapeamento entre uma especificação LOTOS e uma especificação STER, são as distinções semânticas entre os elementos das duas linguagens. Um mecanismo de mapeamento consistente deve garantir a presevarção da natureza semântica dos elementos do mundo LOTOS quando descritos pelos elementos do mundo STER.

Neste capítulo serão discutidas as características conceituais consideradas na definição da metodologia de mapemanto.

## 4.1 - CONDIÇÕES DO STER COMO AMBIENTE DE MAPEAMENTO

Como vimos, em LOTOS um sistema é especificado como um conjunto de processos que executam independentemente e que podem comunicar-se entre si. Os processos LOTOS são definidos a partir de sua interface com o ambiente sendo vistos por este como uma caixa preta que executa um conjunto sequencial de ações (ofertas de sincronização). Os processos podem também executar ações não observáveis pelo ambiente. Estas mesmas características são encontradas nas ambienete STER.

Apesar destas semelhanças, LOTOS é fundamentalmente diferente do STER em muitos aspectos:

*i*) A semântica de cada operador é bastante complexa e, normalmente, não encontra-se operadores com características similares no STER. Exemplo disto são operadores como CHOICE ([]) e DISRUPT ([]), aos quais está associado o conceito de desabilitação de processos de modo não trivial;

*ii*) Alguns aspectos distinguem os mecanismos de sincronização em LOTOS daqueles existentes no STER:

*ii*.a) Uma porta LOTOS é um objeto abstrato compartilhado por vários processos e não possui um tipo fixo associado. No STER uma porta é propriedade de um processo e possui tipo fixo;

*ii*.b) LOTOS envolve tipos de sincronização não existentes no STER como: casamento de valores e geração de valores;

*ii*.c) No STER uma comunicação pode ocorrer sempre que existir uma entidade emissora e uma receptora convenientemente interligadas, e ocorrerá quando ofertas forem apresentadas. Já em LOTOS, condições adicionais podem restringir a ocorrência de um evento. Isto deve-se ao fato de que o conceito de complementaridade [MILNER 80] é mais refinado em LOTOS do que nas STER;

*ii*.d) Em LOTOS a sincronização pode ocorrer entre dois ou mais processos (multi-sincronismo). Nas LPDs ela ocorre sempre entre dois processos.

*iii*) Situações de não determinismo;

*iv*) Definição de comportamentos de caráter generico (CHOICE ...[]...).

Portanto, o mapeamento de uma especificação LOTOS em uma especificação STER não poderá ser realizado diretamente. Para isso, mecanismos adicionais deverão ser criados para que sejam contornadas as diferenças existentes entre os modelos considerados na definição de cada linguagen.

## 4.2 - METODOLOGIA DE MAPEAMENTO

O modelo de mapeamento de especificações LOTOS em especificações STER proposto neste trabalho baseia-se em duas estratégias: usar o recurso de execução multitarefa existente no STER para representar a característica de independência na execução dos processos LOTOS, e definir uma filosofia de tradução dos elementos LOTOS em estruturas específicas no ambiente STER, regidas por regras que garantam a presevação da natureza semântica dos elementos.

No modelo, as características estáticas (estrutura de dados, definição de processos e estrutura de composição dos processos) e dinâmicas (comportamento dinâmico dos processos) da especificação serão mapeadas através de mecanismos distintos.

Com relação à estrutura estática, pode-se enumerar as seguintes considerações:

*i*) Cada processo LOTOS será mapeado em um módulo STER (chamado Módulo Básico) separado. Estes módulos interage entre si sob a coordenação de um módulo administrador (chamado Módulo Gerenciador). Para seguir uma metodologia definida, o processo **specification** (ou seja o comportamento inicial da especificação associado à cláusula **behavior**) será representado por um módulo específico, chamado Módulo Especificação.

*ii*) O Módulo Gerenciador coordena a transação entre os módulos básicos conforme a relação de composição existente entre eles, usando para isso um mapa de composição representado na forma de árvore (Árvore de Composição). A função básica do Módulo Gerenciador é prover mecanismos que permitam a presevação das características semânticas dos elementos LOTOS agora descritos por elemetos STER;

*iii*) A estrutura de dados abstratos será traduzida para uma estrutura de dados concretos através de um conjunto de funções e procedimentos PASCAL, agrupados em arquivos que serão importados pelos módulos básicos de acordo com os tipos associados ao processo que descreve o módulo (este modelo será introduzido como proposta, pois neste trabalho não serão discutidas questões relativas à estrutura de dados).

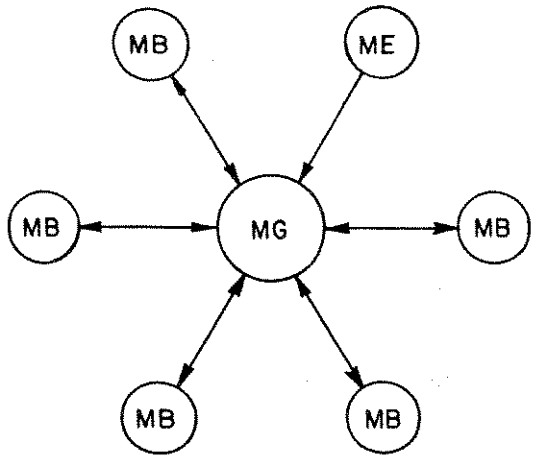



Figura 4.2.1 - Estrutura da Configuração da Especificação

ιv) Na configuração da aplicação os Módulos Básicos e Módulo Especificação se ligarão apenas ao Módulo Gerenciador (fig. 4.2.1). Isto implica que a configuração será dependente da especificação apenas no tipo e número dos módulos básicos;

A estrutura dinâmica da especificação será representada pela evolução da execução dos módulos básicos.

Embora os operadores LOTOS tenham sido definidos de modo a aceitarem como operandos qualquer expressão de comportamento (desde uma simples oferta de evento até um processo), para reduzir o número de mecanismos necessários para a tradução dos operadores, retringiu-se o uso deles em duas situações destintas: na descrição da composição de processos e na expressão de comportamentos básicos.

Desta forma, o modelo proposto classifica, de maneira bem particular, os operadores LOTOS em operadores básicos e operadores de composição. Os operadores básicos são aqueles associados à sequêcia (;) ou escolha ([]) de ofertas de eventos. Os operadores de composição são aqueles usados para definir a estrutura de composição dos processos na especificação (|[A]|, ||, |||, [>, [], >>). Um caso particular é o operador CHOICE ([]) que é classificado como básico e de composição. Isto se deve ao fato de que o uso deste operador é fundamental nas duas situações.

Portanto, durante a especificação deve-se restringir o uso de operadores básicos somente para os casos em que os operandos sejam ofertas de evento, e a utilização de operadores de composição apenas nos casos em que os operandos sejam processos. Assim, enquanto o operador ; pode ser usado para definir um comportamento na

forma *a;P* (onde *a* é uma oferta de evento e *P* é um processo), seu uso será restrito para descrever apenas sequências de eventos (*a;b*, onde *a* e *b* são ofertas de evento); ou, embora um operador como ||| possa ser usado para descrever a composição entre duas expressões quaisquer (p.ex.: *a;b* ||| *c;d* , onde *a,b,c,d* são ofertas), será usado apenas para compor processos (*P1* ||| *P2* , onde *P1* e *P2* são processos).

Os módulos básicos serão compostos por uma sequência de estruturas LPM, responsáveis pela tradução dos operadores LOTOS que definem os comportamento dos processos. Como vimos, devido à incompatibilidade semântica existente entre os operadores LOTOS e os operadores LPM, torna-se impossível a tradução direta entre eles. A solução considerada para este inconveniente será a decomposição dos operadores LOTOS em função dos seus operandos e de suas regras semânticas (fig. 4.2.2).

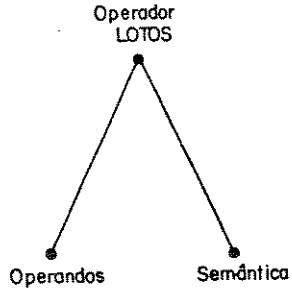



Figura 4.2.2 - Forma de representação dos operadores LOTOS

Os comandos LPM não são suficientes para descrever a funcionalidade envolvida na manipulação de operandos para todos os operadores LOTOS. Assim, o ambiente STER (fig. 4.2.3) deverá ser modificado por meio da inclusão de novas funções (chamadas de funções LPM* ) que serão definidas usando como suporte os comandos LPM existentes. O ambiente STER modificado será referenciado como ambiente LSTER (fig. 4.2.4).

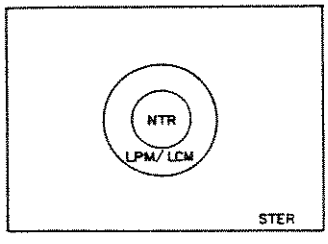



Figura 4.2.3 - Ambiente STER

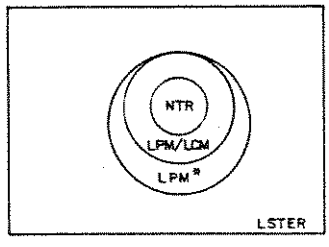



Figura 4.2.4 - Ambiente LSTER

Operadores básicos serão tratados localmente aos Módulos Básicos. No entanto, como a definição da ocorrência de um evento é de responsabilidade do Módulo Gerenciador, a tradução dos operadores será realizada em duas fases: primeiramente, o módu-

lo básico prepara as ofertas associadas ao operador (operandos) na forma de mensagens, e envia ao Módulo Gerenciador. Após a definição da ocorrência de um evento, o Módulo Gerenciador enviará mensagem resposta aos módulos participantes do evento. Em seguida, a partir desta mensagem, o Módulo Básico (participante da interação) terá condições de realizar a análise semântica relativa ao operador.

Os operadores de composição serão tratados também em duas fases distintas. A manipulação de operandos será realizada pelos Módulos Básicos, enquanto que o tratamento semântico será da responsabilidade do Módulo Gerenciador.

O Módulo Gerenciador ao receber uma mensagem associada a um operador de composição, produzirá alterações na árvore de composição que passará a considerar a nova estrutura de composição de processos. A recepção das ofertas de evento ralativas aos operadores básicos, serão tratadas segundo um modelo de execução que avaliará a possibilidade da ocorrência dos eventos de acordo com as regras semânticas dos operadores que compõem os processos.

As características gerais do modelo são apresentadas resumidamente na tabela da fig. 4.2.5.

| Especificação LOTOS | Aplicação STER | Módulo Básico | Módulo Gerenciador |
|---|---|---|---|
| .Processo | .Módulos Básicos | | |
| .Est. Dados abstratos | .Est. Dados concretos | .Inclusão de arquivos | |
| .Operadores básicos | .Troca de Mensagens | .Func. LPM* Alt. árv. | .Alteração da árvore |
| .Operadores de Composição | .Troca de Mensagens | .Func. LPM* oferta de evento | .Aval. regras semânticas |
| .Evento (Sinc. ou Comunicação) | .Comun. entre Proc.(≥2) via Mod. Gerenc. | .Mensagem de partic. no evento | .Definição da ocorrência de um evento |

Figura 4.2.5 - Características gerais do modelo

## 4.3 - Modelo de Execução

Quando uma especificação é observada do ponto de vista de execução, os processos serão vistos como entidades capazes, ou não, de participarem de uma interação.

Como um processo básico (processo cujo comportamento é descrito com o uso apenas de operadores básicos) apresenta diretamente suas ofertas de eventos, poderá interagir com outros processos através de qualquer um destes eventos.

Um processo não-básico (processo desrito pela composição de outros processos) poderá interagir com outros processos somente por meio de um dos eventos possíveis de ocorrer com a participação dos processos instanciados por ele. Esta participação poderá ser de modo dependente ou de modo independente, de acordo com as características semânticas do operador que compõe os processos instanciados. No modo independente não é necessário que os processos instanciados participem conjuntamente do evento - **e** - (caso dos operadores [], [>, ||| e |[A]| onde **e** $\notin$ [A]). Já no modo dependente, o processo não-básico só poderá participar de um evento se os dois processos que instanciou apresentarem ofertas complementares (ou ofertas casadas, cnf. tabela 2.2) deste evento (caso dos operadores ||, |[A]| e **e** $\in$ [A]). Isto implica que a derivação de ofertas de eventos de um processo não-básico dependerá das características semânticas do operador de composição usado na instanciação dos novos processos.

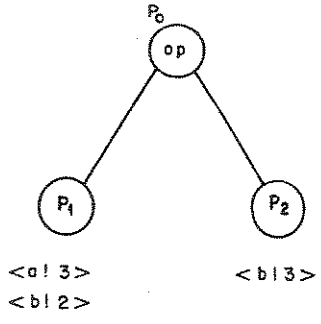
Estes conceitos podem ser resumidos com o seguinte exemplo:

Seja o processo não-básico $P_0$ definido na forma:

$P_0 := P_1$ op $P_2$ ,

onde: $P_1$ e $P_2$ são processos básicos.

Seja ainda o ciclo de execução com o seguinte cenário de ofertas:




O processo $P_1$ poderá participar de um dos eventos <a!3> ou <b!2> e $P_2$ de <b!3>

Caso **op** seja o operador |[b]|, $P_0$ só poderá participar do evento <a!3>,

pois <b!2> e <b!3> não são ofertas complementares. Assim, $P_0$ só poderá participar de um evento em **b** se, p.ex.:, $P_2$ ofertar no lugar de <b!3>, o evento <b!2>.

O modelo de execução utilizado pelo Módulo Gerenciador na definição da ocorrência de eventos, basea-se em uma estrutura em forma de árvore que registra o modo como estão compostos e relacionados os processos LOTOS na especificação, em procedimentos de avaliação e tratamento de interações e em um conjunto de regras de derivação que aplicadas aos diversos nós da árvore de composição definem o conjunto de ofertas relativas a cada um.

O modelo apresentado nesta seção foi proposto por C. WU e G. BOCHMANN [WU 89].

### 4.3.1 - Modelo da Árvore de Composição

A Árvore de Composição é definida para representar a estrutura hierárquica de composição dos processos em uma especificação.

As principais características da Árvore de Composição são:

.Tipo binária;

.Inclui o identificador dos processos instanciados em cada ciclo de execução;

.Composta de nós distribuídos em diversos níveis - representação da estrutura hierárquica da especificação;

.A cada nó estão associadas duas sub-árvores - uma esquerda e outra direita;

.O nó tem a função de descritor da relação existente entre os processos intanciados a partir dele;

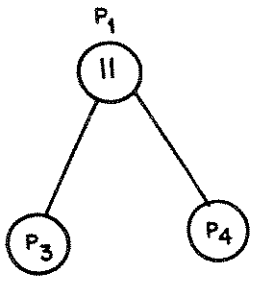
.Existem dois tipos de nó: interno e terminal. O nó interno descreve um dos operadores LOTOS de composição. Já o nó terminal representa um processo básico.

→ **Exemplo**

$$P_1 := P_3 \,||\, P_3$$

$$P_3 := \ldots$$

$$P_4 := \ldots$$




.Um nó será criado sempre que um novo processo for instanciado a partir de um processo já existente;

.Um nó terminal em um dado ciclo de execução poderá tornar-se interno nos ciclos seguintes, devido à expansão do processo descrito pelo nó;

→ **Exemplo**

Considerando a Expansão do exemplo anterior

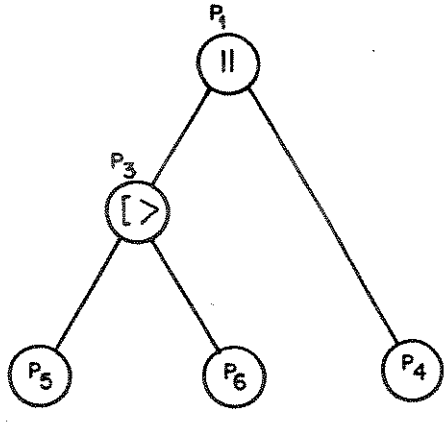
$P_1 := P_3 \,||\, P_4$

$P_3 := P_5 \,[>\, P_6$

$P_5 := \ldots$

$P_6 := \ldots$

$P_4 := \ldots$




.Cada sub-árvore está associada a um processo LOTOS. Um nó terminal é uma sub-árvore que representa um processo básico. Um nó não terminal é a raiz de uma sub-árvore que representa um processo não básico (processo descrito pela composição de outros processos) - fig. 4.4.1.

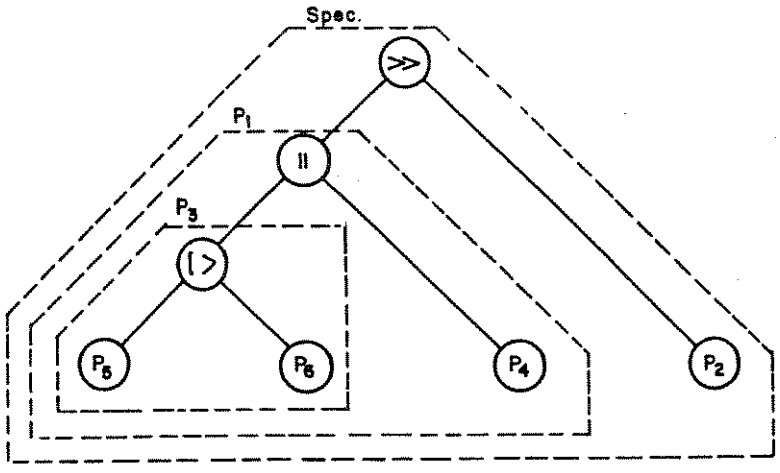



Figura 4.4.1 - Estrutura da Árvore de Composição - sub-árvores como processos

### 4.3.2 - Representação de ofertas LOTOS e tratamento de interações

Uma oferta genérica será representada pela tupla

$$OEg = < \{< m_i, v_i, t_i, c_i > \mid i = 1,...,n\}, G > \quad \text{(Eq. 4.1)}$$

Um processo LOTOS oferece um evento (**OE**) em uma porta **g**, ao qual está associado um certo número de parâmetros. Para cada parâmetro i (i = 1,...,n) o seu modo ($m_i$) indica se um valor fixo ($e_i$) é esperado ($m_i$=!) ou se diferentes valores de um certo tipo ($t_i$) são aceitáveis ($m_i$=?). Neste caso, uma variável $V_i$ assumirá o valor aceito e o predicado $c_i$ (que depende do valor de $V_i$) poderá restringir o universo dos valores aceitáveis. Uma guarda geral **G**, possivelmente dependendo de todas as variáveis da interação poderá impor restrições adicionais.

Deste modo, na Equação 4.1, temos:

OEg ⟶ oferta da porta g;
$v_i = V_i$ se $m_i$ = ?;
$v_i = e_i$ se $m_i$ = !;
$t_i$ é o tipo de $v_i$;
$c_i$ é true se $m_i$ = !;

Consideremos que dois processos $P_1$ e $P_2$ fizeram as seguintes ofertas:

$$P_1 : OE1g = \langle \{ < m1_i,\ v1_i,\ t1_i,\ c1_i > | i = 1,...,n \},\ G1 \rangle$$

$$P_2 : OE2g = \langle \{ < m2_i,\ v2_i,\ t2_i,\ c2_i > | i = 1,...,m \},\ G2 \rangle$$

O casamento de duas ofertas é descrito como:

Matched (OE1g, OE2g) = true ,

e é verificado se as seguintes condições forem satisfeitas (teste de complementaridade):

a) n = m;

b) $t1_i = t2_i$ para i = 1,...,n;

c) Para cada i = 1,...,n uma das seguintes condições for satisfeita:

ι) $(m1_i = m2_i = !) \wedge (e1_i = e2_i)$;

ιι) $(m1_i = !) \wedge (m2_i = ?) \wedge (c2_i(e1_i))$
($V2_i$ assume o valor de $e1_i$);
obs.: $c1_i(e2_i)$ ≡ condição c1 avaliada sobre o valor $e2_i$.

ιιι) $(m1_i = ?) \wedge (m2_i = !) \wedge (c1_i(e2_i))$
($V1_i$ assume o valor de $e2_i$);

ιv) $m1_i = m2_i = ?$;

d) Existem valores $e_j$ atribuíveis às variáveis livres $V1_j$ e $V2_j$ (para todo j em que a condição (iv) é satisfeita) e as demais variáveis $V1_i$ e $V2_i$ assumem os valores definidos pelas condições (iii) e (ii) respectivamente, de modo que G1 e G2, além de todos os $c1_i$ e $c2_i$, sejam verdadeiros.

Se as ofertas (OE1g e OE2g) casam-se, as restrições associadas à suas condições poderão ser combinadas e representadas por uma oferta simples (hipotética) $OE'_g$ escrita como:

$$OE'_g = Derive\ (OE1g, OE2g)$$

A oferta derivada terá a forma:

$$OE'_g = \langle \{ < m'_i, v'_i, t'_i, c'_i > \ | i = 1,...,n \}, G' \rangle ,$$

e deverá satisfazer as seguintes condições:

a) Para cada $i = 1,...,n$ dependendo da condição válida na fase anterior:

caso i): $m'_i = !$, $v'_i = e1_i$, $c'_i = true$;
caso ii): $m'_i = !$, $v'_i = e1_i$, $c'_i = true$;
caso iii): $m'_i = !$, $v'_i = e2_i$, $c'_i = true$;
caso iv): $m'_i = ?$, $v'_i = v1_i$, $c'_i = c1_i \wedge c2^*_i$;
obs.: $c2^*_i$ é o predicado obtido $c2_i$ substituindo $v2_i$ por $v1_i$.

b) $G' = G1 \wedge G2^*$
obs.: $G2^*$ é o predicado obtido de G2 substituindo a $V2_i$ por $V1_i$.

Devido à generalidade da representação na Eq. 4.1 (pag. 4.21), torna-se complexa a sua implementação.

Neste trabalho será considerada uma representação simplificada da Eq. 4.1, onde será desprezada a característica de oferta combinada (será possível apenas ofertas simples, a?x:int; a!3, não sendo permitido a?:int!3) e não será tratado predicados de seleção.

No modelo proposto, uma oferta será representada na forma da tupla:

$OEPg = < ip,g,m,v,t >$ (Eq. 4.2)

onde:

.**ip** é o identificador do processo que oferta o evento;
.**g** é a porta em que é ofertado o evento;
.**m** é o atributo associado ao evento;
.v é o valor envolvido na oferta;
.t é o tipo do valor ou da variável definida na oferta.

### 4.3.3 - Regras de Derivação

As regras de derivação serão aplicadas à Árvore de Composição com o objetivo de derivar um conjunto de eventos através dos quais a especificação poderá evoluir.

Como todos os nós da árvore estão associados à definição de um processo específico, cada um deles deverá ser submetido ao procedimento de derivação.

Devido à natureza hierárquica da árvore (onde só será possível derivar um conjunto de ofertas de eventos para um nó caso as ofertas relativas aos seus nós descendentes já tiverem sido derivados) o procedimento de derivação deverá ser no sentido ascedente, isto é, a partir das folhas até a raiz.

Como os nós terminais estão associados a módulos (processos) que fazem explicitamente suas ofertas, os conjuntos de eventos possíveis nestes nós serão derivados diretamente das ofertas e serão estruturados na forma de listas.

A definição dos eventos possíveis para cada nó interno será a combinação de duas fases de avaliação, uma relativa à complementaridade das ofertas e a outra relativa ao tipo de composição entre os processos descritos pelas duas sub-árvores originadas do nó.

Só será possível a definição da ocorrência de um evento quando toda a estrutura da especificação for pesquisada, o que coincidirá com a derivação do conjunto de eventos possíveis para o nó raiz.

As regras de derivação serão divididas em duas classes: aquelas aplicadas a nós terminais e aquelas aplicadas a nós não-terminais. As regras relativas a nós terminais serão responsáveis apenas pela atribuição de uma lista - vazia ou não - de acordo com as ofertas apresentadas pelos módulos. As regras para nós não-terminais estabelecem as leis para derivação da lista a ser atribuída a um nó interno, e são baseadas nas características semânticas dos operadores associados aos nós.

Nas regras que seguem será usada a seguinte notação:

$LOE_B$ - Lista de ofertas de eventos atribuída ao nó B,

\# - Um dos operadores |[A]|, ||, |||, [>, [] ou >>;

B → B1 # B2 - B1 e B2 são os nós raízes das sub-árvores nascidas de B e que estão relacionados pelo operador #;

[A] - Lista de portas;

OEg - Oferta de evento na porta g.

**.Regras para Nós-Terminais**

$\iota$) $LOE_P = \{OE1,...,OEn\}$
se o processo P ofertar OE1,...,OEn.

$\iota\iota$) $LOE_P = \varnothing$
se o processo P não apresentar nenhuma oferta

**.Regras para Nós não-Terminais**

$\iota$) $LOE_B = LOE_{B1}$
se $B \rightarrow B_1 >> B_2$

$\iota\iota$) $LOE_B = LOE_{B1} \cup LOE_{B2}$
se:
$B \rightarrow B_1 [] B_2$ ou
$B \rightarrow B_1 [> B_2$
$B \rightarrow B_1 ||| B_2$

$\iota\iota\iota$) $LOE_B = \{OEg \mid OEg = derive(OE1g,OE2g)$
$OE1 \in LOE_{B1}$ e $OE2g \in LOE_{B2}$ e
Matched (OE1g, OE2g) = true$\} \cup$
$\{OE1g' \mid OE1g' \in LOE_{B1}\} \cup$
$\{OE2g'' \mid OE2g'' \in LOE_{B2}\}$

se
$B \Rightarrow B_1 |[A]| B_2$, $g \in [A]$ e
$g', g'' \notin [A]$

A seguir será definida uma função bastante utilizada nos procedimentos de derivação descritos na seção 5.3.4

**.Derivação**

*Def.* - Seja **L1** e **L2** listas de ofertas de evento e **op** um dos operadores LOTOS $|[A]|$, $||$, $|||$, $[>$, $[]$ ou $>>$.

**Derivação** é uma transformação que a partir da tupla

< L1, L2, op >

obtém **L3**, que é uma uma lista de ofertas de evento derivada de **L1** e **L2** pela aplicação das regras de avaliação correspondente a **op**.

## 4.4 - Dinâmica de execução da Aplicação

No início da execução da Aplicação o Módulo Especificação enviará ao Módulo Gerenciador a expressão de comportamento do processo **specification**, necessária para a construção da árvore de composição inicial (ACI). Após a criação da ACI o Módulo Especificação será suspenso definitivamente.

O Módulo Gerenciador percorrerá a ACI para identificar os Módulos Básicos que devem ser habilitados e os ativará. Para permitir que estes executem, o Módulo Gerenciador reduzirá sua prioridade (a prioridade do Módulo Gerenciador passará a **lowpr** enquanto que a prioridade dos Módulos Básicos permanecerá em **normalpr**), fazendo com que o Núcleo de Tempo Real (NTR) reescalone a fila de módulos em estado de Pronto (FP). Com o reescalonamento, os Módulos Básicos ativados ocuparão as primeiras posições da fila e o Módulo Gerenciador a última.

Um Módulo Básico durante sua execução poderá: ofertar eventos, instanciar novos processos (i.e., solicitar a ativação de novos módulos) e executar funções de terminação.

**.Oferta de evento**

Cada Módulo Básico permanecerá em execução até envolver-se em uma oferta de evento. Neste momento ele enviará a(s) sua(s) oferta(s) ao Módulo Gerenciador e ficará à espera da definição da ocorrência de um dos eventos que ofertou, passando, portanto, para o estado de bloqueio e consequentemente sendo suspenso (fig 4.3.1).

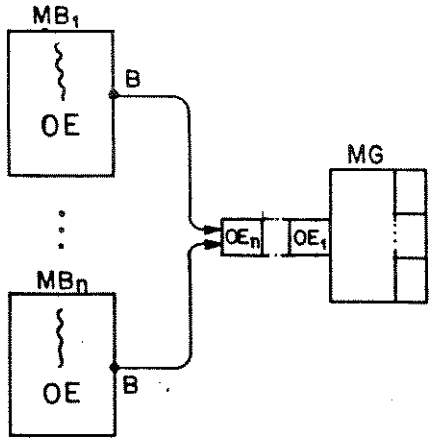



Figura 4.4.1 - Módulos Básicos ofertam eventos

Após a suspensão de todos os Módulos Básicos, o Módulo Gerenciador passará novamente a ser executado. A primeira ação do Módulo Gerenciador, após o reinício da execução, será aumentar sua prioridade. Isto irá garantir o tratamento das fases seguintes sem interrupção. Em seguida, o Módulo Gerenciador receberá todas as ofertas enviadas pelos Módulos Básicos (fig. 4.3.2) e avaliará, de acordo com a estrutura da Árvore de Composição, que eventos são possíveis. Caso mais de um evento seja possível ocorrer, um deles será escolhido aleatoriamente.

Em seguida, o Módulo Gerenciador enviará aos Módulos Básicos envolvidos na interação uma mensagem_resposta com os valores negociados na ocorrência do evento (no caso de interação por geração de valor, o Módulo Gerenciador escolherá um valor qualquer no domínio definido pelas ofertas. Já no caso de interação por casamento de valores, o valor ofertado será recebido como confirmação) - fig. 4.3.3.

Os módulos não envolvidos na interação permanecerão bloqueados, sendo que sua(s) oferta(s) continuarão registradas no Módulo Gerenciador. Logo após o envio das mensagens_resposta aos módulos que participaram da interação, o Módulo Gerenciador reduzirá outra vez sua prioridade provocando novamente o reescalonamento da FP pelo NTR (assumindo a última posicão da fila) e consequentemente sua suspensão.

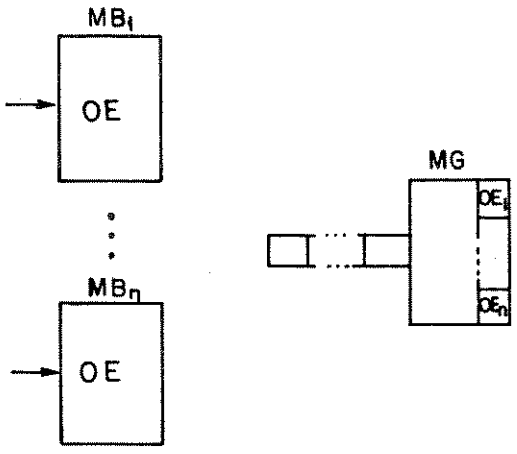



Figura 4.4.2 - Bloqueio dos Módulos Básicos e retorno à execução do Módulo Gerenciador

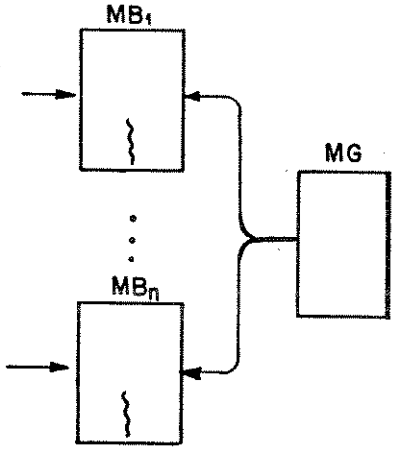



Figura 4.4.3 - Mensagem_resposta de ocorrência de interação

Com a suspensão do Módulo Gerenciador, cada Módulo Básico envolvido na interação receberá a uma mensagem_resposta e continuará a sua execução até que realize nova oferta de evento. Ao procedimento completo de recepção das ofertas, definição da ocorrência de uma interação e envio de mensagens_resposta foi dado o nome de Ciclo de Execução do Modulo Gerenciador (CE).

**.Instanciação de Processos**

Um Módulo Básico pode realizar a instanciação de novos processos (ou módulos) através da execução de uma função LPM* associada a uma dos operadores [], [>, >>, |[A]|, || ou |||. Após solicitar uma instanciação o Módulo Básico será suspenso e não participará mais da execução da Aplicação a menos que seja reativado.

Em cada ciclo de execução o Módulo Gerenciador, antes de entrar na fase de recepção das ofertas, verificará se algum Módulo Básico executou uma função de instanciação. Neste caso ele irá atualizar a Árvore de Composição e ativará os novos Módulos Básicos instanciados. Como anteriormente, o Módulo Gerenciador reduzirá sua prioridade para permitir que os novos módulos ativados entrem em execução. Caso estes façam novos pedidos de instanciação, o procedimento se repetirá.

**.Terminação**

Um módulo Básico ao encerrar sua execução enviará ao Módulo Gerenciador uma mensagem de terminação, que poderá ser, ou não, com passagem de parâmetros.

O Módulo Gerenciador irá tratar a terminação de um processso de acordo com a semântica dos operadores envolvidos com o módulo que está solicitando a terminação.

Após o teste das condições de terminação, o Módulo Básico retornará ao seu estado inicial, ficando suspenso à disposição de nova ativação.

Como consequência da terminação de um módulo, poderá ocorrer a desativação de módulos (P1[>P2, onde P1 é o processo em terminação) ou a ativação de novos módulos ( P1>>P2).

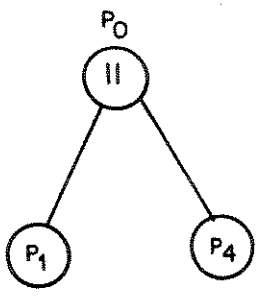
O procedimento de avaliação da terminação de processos produzirá as alterações que forem necessárias na Árvore de Composição - segundo as regras de atualização do Apêndice A.

→ **Exemplo**

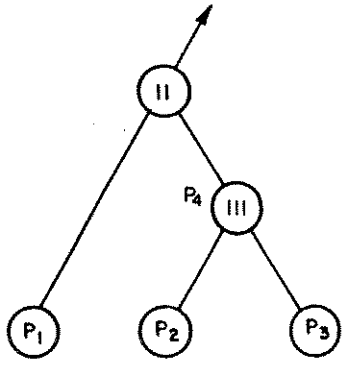
Seja a especificação

```
Po:= P1 || P4
  P1 := a;b;exit
  P4 := P2 ||| P3
    P2 := b;exit
    P3 := a;exit
```

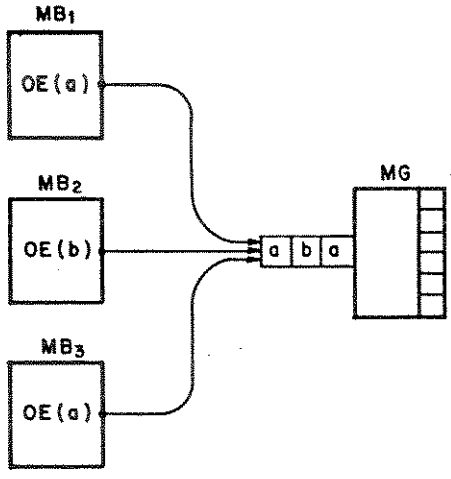
Inicialmente o Módulo Gerenciador receberá a estrutura da Árvore de Composição e a partir dela ativará os módulos MB1 e MB4, que representam os processos P1 e P4 respectivamente.



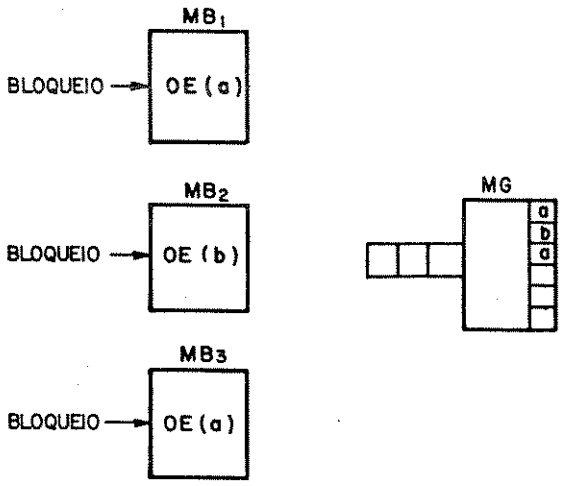

Como o processo P4 executa apenas a instanciação dos processos P2 e P3, sua interação com o Módulo Gerenciador será para atualização da árvore.



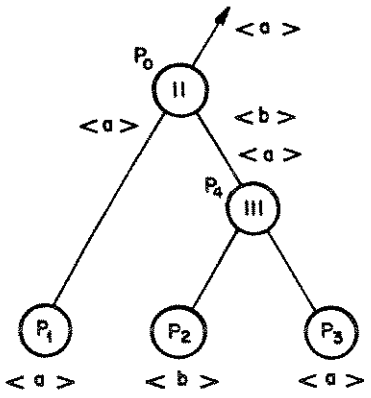

O Módulo Gerencidor retorna a executar e ativará os novos módulos criados (MB2 e MB3). Com a ativação, estes módulos entram em execução. A ação básica de cada um dos módulos MB1, MB2 e MB3 será a apresentação de ofertas.



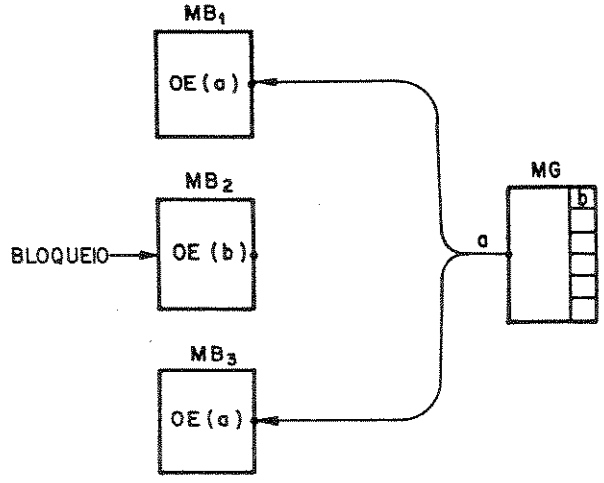

Os módulos ofertantes ficam bloqueados à espera da definição da ocorrência de um evento. O Módulo Gerenciador retornará à sua execução recebendo os eventos ofertados pelos Módulos Básicos.



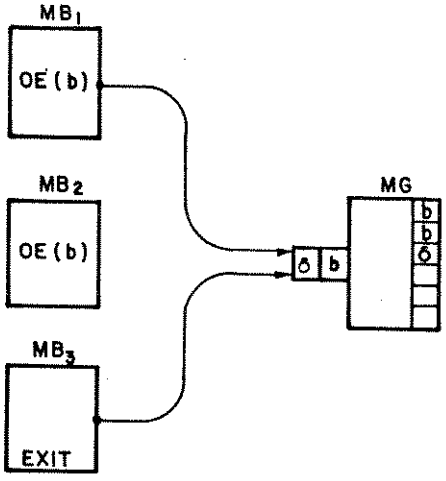

O Módulo Gerenciador avalia as ofertas de acordo com a estrutura da árvore, conforme as regras de derivação, e define que somente o evento **a** é possível.



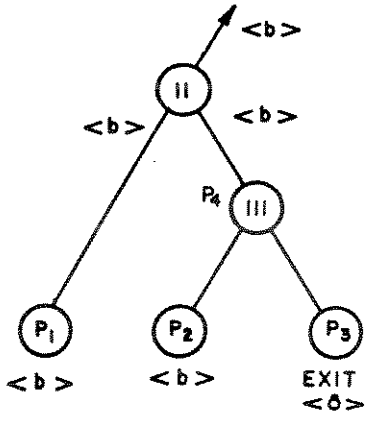

Após a definição da ocorrência da interação entre os módulos MB1 e MB3, pelo evento **a**, o Módulo Gerenciador envia mensagens_resposta a estes módulos. O módulo MB2 por não ter participado da interação continua bloqueado.

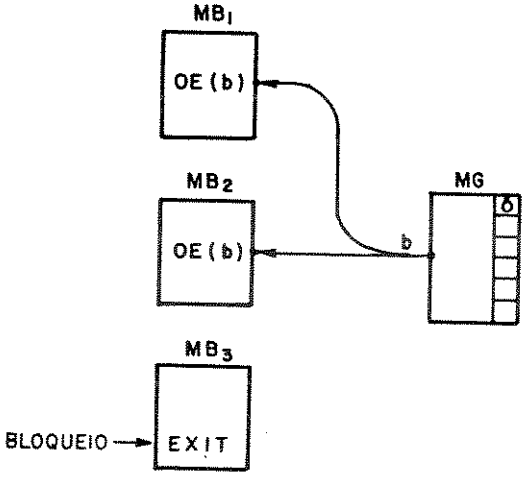


Com a recepção das mensagens de resposta, os módulos MB1 e MB3 voltam à execução e tornam a apresentar novas ofertas. O módulo MB1 oferta o evento **b** enquanto que o módulo MB3, em procedimento de terminação, oferta o pseudo-evento δ, associado ao operador EXIT.

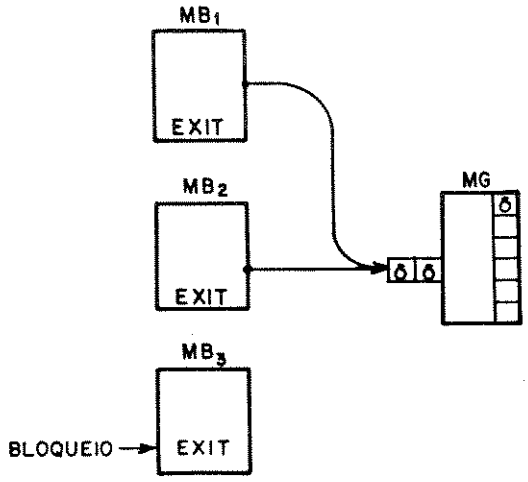



O Módulo Gerenciador avalia o novo cenário de ofertas atribuído à árvore e define a ocorrência da interação entre MB1 e MB2 por meio do evento **b**. Em seguida envia a nova mensagem_resposta desbloqueando os módulos MB1 e MB2 - MB3 continua bloqueado.

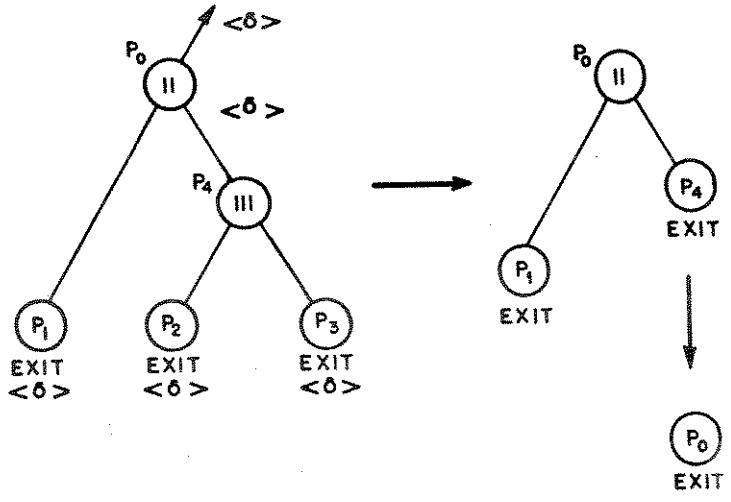


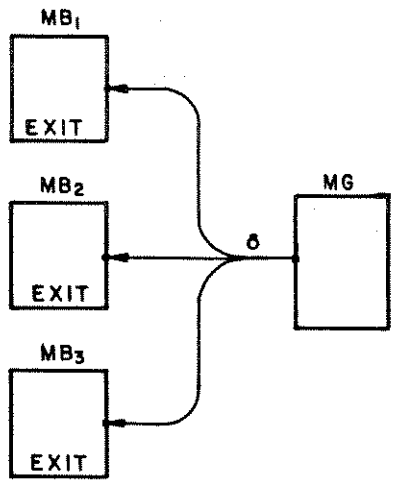

Os módulos MB1 e MB2 - agora também em fase de terminação - apresentam ofertas na pseudo-porta δ.




Só então o procedimento de terminação pode ser concluido. Conforme as regras de derivação para nós não-terminais que descrevem operadores de paralelismo, e pela regra de atualização da árvore relativa a este operador (Apêdice A), a árvore passará pelas seguintes transformações:




A mensagem_resposta enviada para os módulos é de confirmação de terminação.




Com a recepção das mensagens de confirmação de terminação, os módulos entram em estado de bloqueio indefinido.

No Próximo capítulo serão descritas as considerações utilizadas para a implementação dos mecanismos de mapeamento dos operadores LOTOS e na construção do Módulo Gerenciador.

# Capítulo 5

# Características de Implementação

## INTRODUÇÃO

Neste capítulo serão apresentadas as principais características de construção dos Módulos Básicos e do Módulo Gerenciador. Na seção 5.1 serão descritas as funções LPM* e as construções que representarão os elementos LOTOS. Na seção 5.2 serão detalhadas as características gerais da estrutura da Aplicação LSTER. Finalmente, na seção 5.3 será apresentada a composição do Módulo Gerenciador.

## 5.1 - FUNÇÕES DE MAPEAMENTO DOS OPERADORES LOTOS

Um princípio importante observado na definição das funções de mapeamento foi a relação um-para-um entre os objetos envolvidos (operadores $\longrightarrow$ funções/contrutores). Este princípio é indispensável na criação de uma imagem legível na qual seja possível identificar facilmente os diversos operadores.

Nesta seção os procedimentos (funções LPM*) e as construções definidas para mapear os operadores LOTOS serão apresentados. Durante a implementação a principal preocupação foi a simplificação e não a eficiência dos mecanismos propostos.

### 5.1.1 - OPERADORES BÁSICOS

**.Evento Atômico**

O bloco construtivo básico de uma especificação LOTOS é o evento atômico, o qual é usado para representar uma interação entre processos comunicantes. Associado à definição de evento atômico existe o conceito de PORTA ou PONTO DE INTERAÇÃO (recurso abstrato compartilhado por processos).

Ao contrário da abordagem considerada em LOTOS, uma porta STER é um objeto concreto que pertence a um único módulo. Este fato impossibilita o mapeamento de uma porta LOTOS diretamente em uma porta STER.

Decidiu-se manter a natureza abstrata da porta LOTOS no nosso modelo de execução. Neste sentido, uma porta LOTOS continua sendo um objeto simbólico manipulado pelos módulos da Aplicação LSTER.

**.Oferta de evento**

Procedimentos distintos foram utilizados para mapear os dois tipos básicos de ofertas de evento:

**..Declaração de valor**

Formato

```
L1(g,v,t)

onde:
   .g é a porta onde é  ofertado  o
    valor;
   .v é o valor ofertado;
   .t é o tipo de v.
```

Procedimento

```
PROCEDURE L1 (g:...;v:...;t:...);
   BEGIN
      WITH m1 DO
           p:=pr;
           gt:= g;
           m1:=...; (* 1 *)
           v1:=v;
           ti:=t
      END;
           SEND m1 TO PS2;
   END;

onde:
   .pr é o identificador do módulo;
    (Seção 5.2.3);
   .m1 identifica o tipo de atributo.
```

**..Declaração de variável**

Formato

```
L2(g,t)
onde:
   .g é a porta na qual é declarada a
     variável;
   .t é o tipo da variável.
```

Procedimento

```
PROCEDURE L2 (g:...;t:...);
   BEGIN
      WITH m1 DO
           p:=pr;
           gt:= g;
           m1:= ...; (* ? *)
           v1:= ...;
           t:= t
      END;
           SEND m1 TO PS2;
   END;
```

A mensagem **m1** é composta conforme a Eq. 4.2 - pag. 4.25 -.

Após preparada, a mensagem **m1** é enviada pela porta **PS2**, que é

usada especificamente para este fim (seção 5.2.3).

No caso da declaração de variável, o valor v é irrelevante, sendo definido apenas para manter o formato da mensagem. Deve-se notar, também, que o nome que identifica a variável não é enviado na mensagem. A atribuição do valor negociado à variável é feito localmente após a recepção da mensagem de confirmação da ocorrência de um evento.

Deverá ainda ser definido um mecanismo adicional para o envio do predicado de seleção (este mecanismo não foi implementado).

Os procedimentos L1 e L2 são responsáveis apenas pelo envio das ofertas, enquanto que a recepção da mensagem_resposta de participação em uma interação fica na responsabilidade de um outro procedimento - SINC.

A razão para separar o envio da mensagem de oferta da recepção da mensagem de confirmação, deve-se ao fato de que a cada oferta está associada uma mensagem de tipo L1 ou L2. Assim, se o procedimento de recepção de confirmação estiver atrelado ao procedimento de envio de oferta, será impossível descrever a situação em que um módulo apresenta múltiplas ofertas (ver discussão do operador CHOICE adiante).

**..SINC**

O procedimento SINC além da recepção de mensagem de confirmação de participação em um evento, será também responsável pela recepção da mensagem de desabilitação.

Formato

SINC

Procedimento

```
PROCEDURE SINC;
   BEGIN
      RSELECT
         RECEIVE m10 FROM PE2;
         => (* Desabilitação *)
      OR_SELECT
         RECEIVE m2 FROM PE3;
      END_SELECT
```

A mensagem de desabilitação (**m10**) será recebida pela porta **PE2**, enquanto que a mensagem de confirmação de participação em uma interação (**m2**) será recebida na porta **PE3**.

→ **Exemplo**

```
a!3      ⟶  L1(a,3,int);SINC
b?x:int  ⟶  L2(b,int);SINC
```

**.Inação ( STOP )**

Em LOTOS, um processo ao executar um operador STOP entra em um estado de completa inação, não apresentando mais nenhuma oferta ao ambiente. Este estado é equivalente a um DEADLOCK.

Para representar este operador, o módulo ao executar um operador STOP é bloqueado à espera de uma mensagem em uma porta não conectada (DEADLOCK).

Formato

```
Stop
```

Procedimento

```
PROCEDURE STOP;
    BEGIN
        RECEIVE m10 FORM PE
    END;
```

Se observarmos o arquivo de configuração da aplicação (fig. 5.2.6) veremos que não existe nenhuma ligação envolvendo a porta PE.

**.Sequencialidade ( ; )**

A sequencialidade de ações LOTOS será representada pelo operador ; do PASCAL. Devido ao fato de alguns operadores LOTOS serem mapeados por mais de um procedimento STER, nem todos os pontos-e-vírgulas presentes no corpo de um módulo LPM estão necessariamente mapeando um operador de sequência LOTOS.

→ **Exemplo**

```
a?x:int                L2(a,int);SINC
  ;          ⟶              ;
b!x                    L1(b,x,int);SINC
```

**.Escolha ( CHOICE - [] )**

o operador CHOICE define uma expressão de comportamemto a partir da composição de duas outras expressões, na forma:

B := B1 [] B2 ,

onde B se comportará como B1 ou como B2 dependendo de qual das duas expressões realizar a primeira interação.

Em relação à definição de B1 e B2, elas tanto podem ser a criação de um processo, como a descrição direta de ações básicas LOTOS, como exemplificado a seguir;

```
i) B := B1 [] B2
      B1 := ...
      B2 := ...

ii B := a;b;stop [] c;d;stop

iii) B := a;b;stop [] B1
```

O operador CHOICE será representado de dois modos distintos, dependendo de como são definidas as expressões de comportamento.

Na forma (i) mapeamos o operador em uma mensagem de requisição de expansão da árvore.

A mensagem_resposta de confirmação de participação em uma interação (m2) inclui uma informação relativa à porta em que ocorreu a interação (seção 5.2.1). Isto permite saber, no caso de múltiplas ofertas, qual dos eventos foi escolhido. Devido a este fato, o operador CHOICE na forma (ii) será traduzido:

.Pelo envio de tantas mensagens - L1 ou L2 - quantos forem os eventos envolvidos;

.Pela execução do comando de recepção de mensagem_resposta (SINC) uma única vez;

.Por um comando CASE (PASCAL) que identificará, a partir da informação que indica o nome da porta onde foi definida a ocorrência de uma interação, qual das ofertas foi a escolhida.

→ **Exemplo**

A expressão de comportamento

```
a?x:int;P1
```

será representada por:

```
L2(a,int);
L1(b,true,int);
SINC;
CASE porta OF
    'a': P1
    'b': P2
END
```

O uso do operador CHOICE na forma (iii) não será permitida na nossa estrutura de compilação, devido à natureza distinta do tratamento das duas formas anteriores. Para contornar este problema, a expressão deve ser reescrita para uma das outras duas formas.

→ **Exemplo**

```
B:= a;b;stop [] B1
 B1 := c;d;stop
```

deve ser reescrita para:

```
B:= B2 [] B1
 B1 := c;d;stop
 B2 := a;b;stop
```

ou para:

```
B := a;b;stop [] c;d;stop
```

**.Condicionalidade**

A condicionalidade em LOTOS é definida pelo uso de Predicados ou Guardas.

O Predicado é uma condição que deve ser testada no momento da ocorrência do evento (seção 2.4). Por este motivo ele foi incluido como parâmetro a ser avaliado pelo Módulo Gerenciador, no procedimento de definição da ocorrência de evento (na versão atual o Módulo Gerenciador não faz análise de condições de predicado).

Uma expressão de comportamento pode ser precedida por uma Guarda ( [cond] → B ). A guarda indica quando a expressão é possível (cond = true) ou não (cond = false).

Como a Guarda avalia uma condição local ao módulo, ela pode ser representada por uma cláusula IF...THEN do PASCAL.

→ **Exemplo**

A expressão de comportamento

```
   [cond1] → a;B1
[] [cond2] → b;B2
[] [cond3] → c;B3
```

será representada por:

```
IF  cond1 THEN L(a);
IF  cond2 THEN L(b);
IF  cond3 THEN L(c);
SINC
CASE porta OF
    'a' : B1;
    'b' : B2;
    'c' : B3
END
```

### 5.1.2 - Operadores de Composição

Como os operadores de composição são usados basicamente para a solicitação de construção ou atualização da Árvore de Composição (seção 4.2), eles serão mapeados seguindo um formato padrão. Neste formato, a mensagem de solicitação é composta pelo código relativo à função (cd) e pelos identificadores do nó a ser expandido (pr1) e dos nós a serem acrescentados (pr2 e pr3).

Na definição das estruturas dos procedimentos serão deixadas em aberto a definição de alguns tipos. Com a introdução do arquivo de definição da Aplicação (seção 5.2.1) as dependências serão resolvidas.

**.Escolha ( CHOICE - [] )**

O operador CHOICE quando usado para definir uma expressão a partir da composição de duas outras expressÕes, será definido por:

Formato

```
CHOICE (proc1,proc2,proc3)
```

Procedimento

```
PROCEDURE CHOICE (pr1,pr2,pr3:...);
  BEGIN
    WITH m9 DO
      cd := 1;
      p1 := pr1;
      p2 := pr2;
      p3 := pr3;
    END;
    SEND m9 TO PS1;
  END;
```

**.Paralelismo com Sincronização Restrita (|[A]|)**

Formato

```
PAR (proc1,proc2,proc3)
```

Procedimento

```
PROCEDURE PAR (pr1,pr2,pr3:...,[A]);
  BEGIN
    WITH m9 DO
      cd := 2;
      p1 := pr1;
      p2 := pr2;
      p3 := pr3;
    END;
    SEND m9 TO PS1;
  END;
```

Deverá ser adotado um mecanismo específico para o envio do conjunto de portas de sincronização ([A]). Atualmente, este mecanismo é implementado de modo simples como a passagem de um conjunto de caracteres (**set of char**). Nesta forma as portas usadas na especificação deverão ser identificados por simples caracteres.

**.Paralelismo sem Sincronização ( ||| )**

Formato

```
PARI (proc1,proc2,proc3)
```

Procedimento

```
PROCEDURE PARI (pr1,pr2,pr3:...);
   BEGIN
      WITH m9 DO
           cd  := 3;
           p1  := pr1;
           p2  := pr2;
           p3  := pr3;
      END;
      SEND m9 TO PS1;
   END;
```

**.Paralelismo com Sincronização Completa ( || )**

Formato

```
PARS (proc1,proc2,proc3)
```

Procedimento

```
PROCEDURE PARS (pr1,pr2,pr3:...);
   BEGIN
      WITH m9 DO
           cd  := 4;
           p1  := pr1;
           p2  := pr2;
           p3  := pr3;
      END;
      SEND m9 TO PS1;
   END;
```

**.Composição Sequencial ( Enable - >> )**

Forma

```
ENABLE (proc1,proc2,proc3)
```

Procedimento

```
PROCEDURE ENABLE (pr1,pr2,pr3:...);
   BEGIN
      WITH m9 DO
           cd  := 5;
           p1  := pr1;
           p2  := pr2;
           p3  := pr3;
      END;
      SEND m9 TO PS1;
   END;
```

Um mecanismo específico deverá tratar a definição das condi-

ções associadas à cláusula ACCEPT (este mecanismo não foi implementado).

**.Terminação com Sucesso ( EXIT )**

Formato

```
EXITY
```

Procedimento

```
PROCEDURE EXITY;
   BEGIN
      WITH m9 DO
         cd := 6;
         p1 := ...;(* irrelevante *)
         p2 := ...;(* irrelevante *)
         p3 := ...;(* irrelevante *)
      END;
      SEND m9 TO PS1;
      RECEIVE m10 FROM PE2;
      RECEIVE m10 FROM PE;
   END;
```

A LPM possui um comando EXIT e por isto usamos na identificação deste construtor, o nome EXITY.

A definição de P1,P2 e P3 foi mantida apenas por questão de padronização da mensagem **m9**.

Após o envio da mensagem **m9**, o módulo fica bloqueado na espera da ocorrência do pseudo-evento ( $\delta$ ) - ver exemplo da seção 4.4.

O Módulo Gerenciador usará a mesma porta que é utilizada para desabilitação de um processo, para confirmar a terminação do módulo.

A razão do uso da segunda cláusula de recepção - que irá impor um DEADLOCK ao módulo - se deve à interpretação semântica do operador EXIT (seção 2.2). Após a ocorrência do pseudo-evento o processo entra em um estado de inação (comparar com operador STOP, seção 5.1.1).

**.Disrupção ( DISRUPT - >> )**

Formato

```
DISRUPT (proc1,proc2,proc3)
```

Procedimento

```
PROCEDURE DISRUPT (pr1,pr2,pr3:...);
  BEGIN
    WITH m9 DO
      cd := 7;
      p1 := pr1;
      p2 := pr2;
      p3 := pr3;
    END;
    SEND m9 TO PS1;
  END;
```

**.Instanciação de Processos**

Um processo LOTOS pode instanciar outros processos através da solicitação de expansão da Árvore de Composição - sendo incluídos na árvore os novos processos instanciados - e pela ativação dos módulos que descrevem estes processos.

Como o STER não possui mecanismo de reconfiguração dinâmica, só será permitido a criação de instâncias de um módulo no momento da elaboração do Arquivo de Configuração. Isto implica que se deve conhecer de antemão que processos serão instanciados e quantas instâncias de cada um serão criadas.

Esta limitação, apesar de restringir a potencialidade do mecanismo de instanciação presente em LOTOS, normalmente não criará situações incontornáveis, desde que uma especificação que tenha sido desenvolvida segundo um estilo orientado à implementação não fará uso de instanciação em uma forma intratável segundo este modelo.

Além disso, o ambiente CONIC permite a reconfiguração dinâmica através da criação de novos módulos, dealocação de módulos já existentes e da reconfiguração das ligações das portas. Neste esquema, estaríamos mais próximos da forma de instanciação em LOTOS. Assim, a expansão do STER seria uma outra possibilidade de contornar este problema.

A outra forma de instanciação considerada é a **recursão**. Foi criado um procedimento específico, desde que os procedimentos de atualização e ativação são desnecessários neste caso. Este procedimento simplesmente levará a execução do módulo para o estado inicial de descrição do comportamento do processo LOTOS - **inic2**, Seção 5.2.3.

Formato

```
AutoExec
```

Procedimento

```
PROCEDURE AutoExec;
    BEGIN
        GOTO inic2
END;
```

→ **Exemplo**

i) Instanciação pela criação de novos processos.

A expressão

$$P_0 := P_1 \,||\, P_2$$

será traduzida nas seguintes ações:

| $MB_{P0}$ | MG |
|---|---|
| PARI ($P_0$,$P_1$,$P_2$) | .Expandir a árvore<br>.Ativar módulos $P_1$ e $P_2$ |

ii) Recursão

O Processo

```
process P1[...](...):noexit :=
        .
        .
        .
        P1[...](...)
endproc
```

será traduzido para:

```
MODULE P1;
        :
        :
   BEGIN_MODULE
        :
        :
    LOOP
       inic2:...
            :
            :
        AutoExec
   END_LOOP
END_MODULE.
```

A estrutura completa de um módulo que descreve um processo LOTOS será apresentada na seção 5.2.3.

**.Criação de Escopos de Portas (hide...in...)**

No lugar de definir mecanismos especí-ficos para o tratamento de escopos, decidiu-se reduzir os nomes de portas particulares a cada escopo em um conjunto de identificado-res únicos dentro do universo de nomes usados no texto da especifi-cação. Este procedimento permite que o Módulo Gerenciador trate todas as ofertas como se pertencessem a um único nível.

→ Exemplo

A especificação

```
Spec := P1[a,b] || P2[c]
        P1[a,b] := hide c in ...
        P [c] := ...
```

será reescrita na forma:

```
Spec := P1[a,b] || P2[c]
        P1[a,b] := hide d in ...
        P2[c] := ...
```

**.Comportamentos Generalizados**

Devido à natureza restrita de um ambiente de implementação, onde os recursos não podem ser usados ilimitadamente, torna-se inconveniente o uso de comportamentos generalizados. Além disso, uma especificação desenvolvida em um estilo orientado à implementação normalmente trata de situações permitidas em condições de

realização do sistema. Por isto, as construções LOTOS que induzam a comportamentos generalizados não serão tratadas neste trabalho.

**.Definição Local de Variáveis (Let ... in ...)**

O operador de definição local

```
Let ... in B(...)
```

é traduzido por um conjunto de atribuições PASCAL, associado ao(s) módulo(s) que descrevem os processos envolvidos na expressão B(...).

→ Exemplo

Seja a seguinte atribuição local

```
P := Let x:int = 10 in ...
```

Será traduzida para

```
MODULE P;
    .
    .
    .
  BEGIN_MODULE
      x:=10;
        .
        .
        .
  LOOP
        .
        .
        .
  END_LOOP
END_MODULE.
```

**. Eventos Internos**

Em uma especificação os eventos internos estão associados à descrição de comportamentos não determinísticos.

O ambiente LSTER não inclui nenhum mecanismo específico de tratamento de tais situações. Assim, no desenvolvimento de protótipos, os eventos internos deverão ser substituídos por comportamentos determinísticos manipulados por mecanismos de decisão aleatória. Este procedimento será utilizado no exemplo 6.1 (comparar a descrição da especificação do apêndice D com a apresentada em [LOGRIPPO 88]).

No caso em que eventos internos são utilizados para descrever cláusulas de tempo, as primitivas DELAY e TIMEOUT poderão ser usadas no mapeamento.

### 5.1.3 - Funções Especiais

Além dos procedimentos SINC e AUTOEXEC, foram definidos outros procedimentos importantes na arquitetura do modelo. Estes procedimentos são chamados de especiais porque não traduzem diretamente nenhum operador LOTOS.

**.Espera de Ativação**

Este procedimento tem por função definir o estado inicial dos Módulos Básicos. Primeiramente, será passado o identificador do processo que o módulo representa. Este identificador será usado pelo procedimento de ativação de módulos (seção 5.3.2). Em seguida o módulo ficará bloqueado à espera da mensagem de ativação.

Formato

```
EnbProc
```

Procedimento

```
PROCEDURE EnbProc;
   BEGIN
      SEND (* ident. *) TO PS3;
      RECEIVER m10 FROM PE1 ;
   END;
```

**.Suspensão do Módulo Especificação**

Este procedimento será usada sempre como a última instrução do Módulo Especificação. Sua função será manter o módulo bloqueado indefinidamente.

Formato

```
Endy
```

Procedimento

```
PROCEDURE ENDY;
   BEGIN
      RECEIVE m10 FORM PE
   END;
```

**.Suspensão de um Módulo Expandido**

No caso em que um módulo executa uma função de instanciação, deverá ficar em um estado no qual possa ser desabilitado. Este fato será importante no mecanismo de tratamento do operador DISRUPT.

Formato

```
Endb
```

Procedimento

```
PROCEDURE ENDB;
    BEGIN
        RECEIVE m10 FORM PE
    END;
```

## 5.2 - Estrutura da Aplicação STER

### 5.2.1 -Arquivo de Definição (TipMes.def)

Vimos no Capítulo 3 - seção 3.2 - que a função de uma unidade de definição na LPM é evitar que uma mesma estrutura de tipos e variáveis seja escrita repetidas vezes nos vários módulos que compõem a aplicação.

Na Aplicação LSTER que descreve uma especificação LOTOS, a unidade de definição será usada para definir os tipos das mensagens e das portas que são usadas nas funções e procedimentos de mapeamento dos operadores LOTOS. Estas funções serão usadas por todos os módulos da Aplicação, justificando assim o uso de importação de um arquivo de definição de tipos.

Como não nos detemos em detalhes relativos à estrutura de dados, será descrita uma arquitetura apenas simbólica para o arquivo de definição (o apêndice B apresenta o arquivo atualmente utilizado). Serão detalhadas, no entanto, algumas mensagens cuja composição é importante para a discussão completa do modelo de implementação.

A estrutura básica do arquivo de definição é:

```
DEFINE      TipMes;

    TYPE
                :
                :
        MENS1  = ... ; (* Mensagem usada no mecanis-
                          mo de oferta de evento *)

        MENS2  = ... ; (* Mensagem de confirmação de
                          participação em interação *)

        MENS3  = ... ; (* Mensagem de identificação
                          do módulo*)

        MENS4  = ... ; (* Mensagem usada no mecanis-
                          mo de envio do conjunto de
                          portas de sincronização *)


    MENS5 = ... ; (* Mensagem de identificação
                     de módulo*)

    MENS6 = ... ; (* Mensagem de identificação
                     de porta *)

    MENS7 = ... ; (* Código das funções e pro-
                     cedimentos de mapeamento dos
                     operadores LOTOS *)

    MENS8 = ... ; (* Mensagem usada no meca-
                     nismo de envio da definição
                     de predicado *)

    MENS9 = ... ; (* Mensagem de criação/atua-
                     lização da árvore *)

   MENS10= ... ; (* Mensagem de desabilitação e
                     confirmação de terminação *)

                :
                :
END_DEFINE.
```

Os procedimentos responsáveis pelo mecanismo de oferta de evento (seção 5.1.1) farão uso da mensagem **mens1**. Na seção 4.5.1, vimos que o modelo de representação de uma oferta possui um formato definido (Eq. 4.2 - pag. 4.25). A mensagem **mens1** foi estruturada a partir deste padrão, na forma:

```
MENS1 = RECORD
          p1 : ...; (* Identificador do módulo *)
          g1 : ...; (* Identificador de  porta *)
          m1 : ...; (* Identificador de atributo *)
          v1 : ...; (* Tipo da variável ou valor
                       negociado *)
          t1 : ...; (* Identificador de  tipo  da
                       variável ou valor negociado *)
        END
```

A confirmação de participação em uma interação será transportada pela mensagem **mens2**. Novamente esta mensagem deve ser definida segundo um formato fixo, ou

```
MENS2 = RECORD
          g : ...; (* Identificador de  porta *)
          v : ...; (* Tipo da variável ou valor nego-
                      ciado *)
        END
```

Na seção 5.2.3 será abordada a necessidade do uso de dois tipos de identificadores para um Módulo Básico. Um módulo será identificado pelo nome do processo LOTOS que descreve - ou por um nome qualquer que o torne único entre todos os módulos da Aplicação - (cnf. a estrutura da mensagem **mens5**), e pelo seu identificador STER (representado pela mensagem **mens3**).

O mecanismo que tratará o envio do conjunto de porta de sincronização restrita (|[A]|) é representado no arquivo de definição pela mensagem **mens4**.

A **mens8** representa o conjunto de variáveis que serão usadas pelo mecanismo de transmissão das condições envolvidas no predicado de seleção.

Do mesmo modo que foi usada uma mensagem para identificar o nome de um módulo, foi definida uma mensagem para identificação do nome de uma porta.

Um dos tipos de informação comum a quase todas as funções é o código com que serão identificadas. Para este fim, definimos a mensagem **mens7**. A mensagem completa que será manipulada por estas funções será do tipo **mens9**, que é composta na forma:

```
MENS9 =RECORD
          cd : mens7;
          p1 , p2 , p3 : mensn5
       END
```

A mensagem de desabilitação ou de confirmação de terminação é definida pela mensagem **mens10**.

## 5.2.2 - Módulo Especificação

A estrutura do Módulo Especificação é

```
MODULE  Spec;
   USE  TipMes.inc;
   EXITPORT
      PS1: mens9;
   ENTRYPORT
      PE: mens10;
   MESSAGE
      m9: mens9;
      m10: mens10;
   BEGIN_MODULE
      LOOP
         .
         .
         .
         Endy;
      END_LOOP;
   END_MODULE.
```

onde:

i) O arquivo **TipMes.inc** é usado para a definição dos tipos das portas $\mathbf{PS_1}$ e **PE** e das mensagens associadas a estas portas;

ii) A porta de saída $\mathbf{PS_1}$ é usada na comunicação com o Módulo Gerenciador;

iii) A porta de entrada **PE** é usada pela funções EXIT E ENDY;

iv) **m9** e **m10** definem os tipos das mensagens associadas às portas $\mathbf{PS_1}$ e **PE** respectivamente,

v) O corpo principal do módulo (**BEGIN_MODULE** ... **END_MODULE**) será composto por uma sequência de funções LPM* que descrevem o comportamento do processo **specification**;

vi) O uso de **LOOP**...**END_LOOP** faz parte da estrutura padrão em todos os módulos STER. Para o Módulo Especificação não terá efeito, pois a função ENDY suspende indefinidamente a execução do módulo.

Na fig.5.2.4 é exemplificado o procedimento de construção do Módulo Especificação.

```
Specification  Exemplo ... :noexit

     type     ...    endtype

     behavior
          P1 [...]([...]) || P2[...]([...])

     where
          process P1[...]([...]):noexit :=
               P3[...]([...]) >> P4[...]([...])

            where
               process P3[...]([...]):noexit :=
                    ...
               endproc
               process     P4[...]([...]):noexit     :=
                    ...
               endproc
          endproc
          process P2[...]([...]):noexit :=
               ...
          endproc
```

a) Especificação LOTOS resumida

```
MODULE  Spec;

   USE  TipMes.inc;

   EXITPORT
     PS1: mens9;

   ENTRYPORT
     PE: mens10;

   MESSAGE
      m9: mens9;
      m10: mens10;

   BEGIN_MODULE
      LOOP
           Pars('Spec','P1','P2');
           Endy;
      END_LOOP;
   END_MODULE.
```

b) Módulo de Especificação

Figura 5.2.4 - Procedimento de derivação do Módulo Especificação

A fig. 5.2.4a mostra o texto da especificação LOTOS simplificada. A descrição do módulo é apresentada na fig. 5.2.4b .

### 5.2.3 - Módulo Básico

A estrutura genérica de um módulo básico é mostrada a seguir:

```
MODULE Exemplo;

    USE TipMes.inc;

    EXITPORT
        PS1:    mens9;
        PS2:    mens1;
        PS3:     mens3;

    ENTRYPORT
        PE1,PE2,PE: mens10;
        PE3: mens2;

    MESSAGE
        m1: mens1;
        m2:  mens2;
        m3:  mens3;
        m10:  mens10;

    (*  $INCLUDE: 'Arq .Dad' *);
        .
        .
        .
    (* $INCLUDE: 'Arq_n.dad' *);

    CONST
        pr  = 'Exemplo';

    LABEL
        inic1,inic2;

    PROCEDURE          .....;EXTERN;

    BEGIN_MODULE
        inic1:EnbProc;
              .
              .
              .
        LOOP
            inic2:....;
                  .
                  .
                  .
        END_LOOP;
    END_MODULE.
```

onde:

*i*) o arquivo **TipMes.inc** possui as definições dos tipos das portas e das mensagens;

*ii*) existem três tipos de portas de saída que interligam o módulo básico ao módulo gerenciador:

.$PS_1$ - porta usada para o envio de mensagens de atualização da Árvore de Composição;

.$PS_2$ - porta para o envio das mensagens de oferta de evento;

.$PS_3$ - porta usada para envio do identificador do módulo;

iii) As portas de entrada (interligam o módulo gerenciador ao módulo básico) são:

.$PE_1$ - porta usada para ativação do módulo;

.$PE_2$ - porta para recepção de mensagem de desabilitação;

.$PE_3$ - para recepção dos parâmetros de confirmação da participação em um evento;

.PE - usada pelas funções EXITY e STOP;

iv) As mensagens **m1, m2, m3** E **m10** serão usadas nos serviços de envio de oferta de evento, recepção da confirmação de participação em interação, envio de identificador e recepção de mensagem de desabilitação, respectivamente;

v) Em seguida aparece um conjunto de instruções de inclusão das definições dos tipos de dados e das variáveis;

vi) A constante $P_r$ identificará o processo na árvore de composição. Assim, todas as mensagens de manipulação da árvore farão referência aos nomes dos processos, definidos como constantes;

vii) **LABEL** definirá dois tipos de rótulos: **inic1,** que indicará o estado inicial do módulo **(EnbProc),** e **inic2,** que identificará em que ponto inicia a descrição do comportamento do processo LOTOS. Estes rótulos serão referenciados pelos procedimentos de tratamento de desabilitação (função SINC) e AUTOEXEC, respectivamente (seção 5.1.1 e 5.1.2);

viii) Em seguida, as funções LPM* **(PROCEDURE ...;EXTERN)** são definidas como externas. Este procedimento dará consistência à fase de compilação do programa PASCAL derivado do módulo (.LPM), já que tais funções estão em uma biblioteca e, portanto, somente acessíveis durante a fase de LINK;

ix) Por fim, temos a descrição do corpo principal do módulo **(BEGIN_MODULE...END_MODULE).** Em **LOOP...END_LOOP** teremos o conjunto de instruções que traduzirão o comportamento dos processos. Todas as instruções existentes antes de LOOP estarão associadas aos procedimentos e funções de inicialização.

O Módulo Básico será identificado de duas maneiras:

. **IDENTIFICADOR MÓDULO** (module_id), que será usado para iden-

tificar o processo na Árvore de Composição;

.NOME DO PROCESSO (pr='Exemplo'), usado pelo Módulo Básico para "Personalizar" as ofertas de eventos.

### 5.2.4 - Módulo Gerenciador

A estrutura do módulo é:

```
MODULE Gerenciamento;

    USE TipMes.inc;

    ENTRYPORT
        PE1:  mens9  QUEUE  n;
        PE2:  mens1 QUEUE n;
        PE3:  mens3 QUEUE n;

    EXITPORT
        PS1[1..n]: mens10;
        PS2[1..n]: mens10
        PS3[1   n]; mens2;

    MESSAGE
        m1: mens1;
        m2:  mens2;
        m3:  mens3;
        m10:  mens10;

    TYPE
        (* Definições locais *)

    VAR
        (* Definições locais *)

    (* PROCEDURES *)

    BEGIN_MODULE
        LOOP

            (* Corpo Principal)

        END_LOOP;
    END_MODULE.
```

onde - além das características já abordadas nas seções 5.2.2 e 5.2.3 quanto ao uso do arquivo de definição, das portas de entrada e saída e das mensagens - temos:

*i*) As portas de entrada são definidas incluindo uma fila devido às características das comunicações entre os módulos básicos e o módulo gerenciador (muitos-para-um);

*ii*) As portas de saída serão definidas na forma de uma família de portas. Isto porque para as funções de ativação, resposta de participação em uma interação e desabilitação, existirá uma porta associada a cada módulo (no modelo declaramos a existência de **n** processos);

*iii*) **TYPE** definirá tipos de apontadores para as contruções de árvore, tabelas e listas; *iv*) **VAR** criará variáveis de uso geral e os **heads** para as listas e tabelas;

v) O comportamento principal do Módulo Gerenciador será descrito na forma de procedimentos de acordo com as seguintes fases:

1. Manipulação da árvore de composição;
2. Ativação de módulos básicos;
3. Criação e gerenciamento de tabelas;
4. Manipulação das ofertas de evento;
5. Avaliação de sincronizações;
6. Mensagens de participação em sincronização;

*vi*) O fluxograma da fig. 4.2.5 descreve o comportamento do corpo principal do módulo.

A redução do nível de prioridade visa sempre levar os Módulos Básicos à execução, suspendendo momentaneamente o Módulo Gerenciador. Já o aumento da prioridade tem o objetivo de dar autonomia - embora que executando funções desbloqueantes - ao Módulo Gerenciador para realização de uma lista de tarefas . Assim, depois de receber as ofertas de eventos enviadas pelos Módulos Básicos, o Módulo Gerenciador só será suspenso - através de nova redução do nível de prioridade - após a realização completa de suas funções, que será marcada pelo envio de mensagens_resposta aos Módulos Básicos participantes da interação.

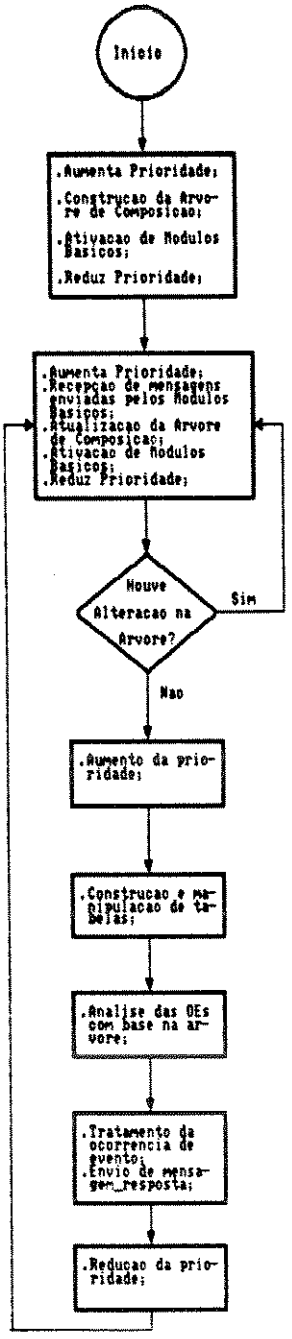



Figura 5.2.5 - Procedimento de execução do Módulo Gerenciador

## 5.2.5 - Arquivo de Configuração

O arquivo de configuração será o responsável pela definição das cópias lógicas de cada tipo de módulo e dos parâmetros reais com os quais serão criadas as instâncias. Estabelecerá também as ligações devidas entre os módulos.

O formato genérico do arquivo de configuração é mostrado na fig. 5.2.6, onde:

i) **INSTANCE** define instâncias de módulos;

ii) **CREATE** cria as instâncias definidas;

iii) **LINK** permitirá que sejam feitas as ligações entre os módulos. Existem seis classes de ligações.

1. De envio de mensagens de manipulação da árvore;
2. De envio de ofertas de evento;
3. De envio de identificadores dos módulos;
4. De ativação de módulos;
5. De resposta da ocorrência de evento;
6. Desabilitação de módulos;

```
CONFIGURATION Exemplo_Config;

    INSTANCE

        sp: Spec;
        p1: Exemplo;
        .
        .
        .
        pn: ...;
        g: Gerenciador;

    CREATE

        sp /S=.../H=...,
        p1 /S=.../H=...,
        .
        .
        .
        pn /S=.../H=...,
        g /S=.../H=...;

    LINK

        s.PS1 TO g.PE1;
        p1.PS1 TO g.PE1;
        .
        .
        .
        pn.PS1 TO g.PE1;
        p1.PS2 TO g.PE2;
        .
        .
        .
        pn.PS2 TO g.PE2;
        p1.PS3 TO g.PE3;
        .
        .
        .
        pn.PS3 TO g.PE3;
        g.PS1[1] TO p1.PE1;
        .
        .
        .
        g.PS1[n] TO pn.PE1;
        g.PS2[1] TO p1.PE2;
        .
        .
        .
        g.PS2[n] TO pn.PE2;
        g.PS3[1] TO p1.PE3;
        .
        .
        .
        g.PS3[n] TO pn.PE3;

END_CONFIG.
```

Figura 5.2.6 - Estrutura do Arquivo de Configuração

### 5.2.6 - Arquivos de Dados

A estrutura de dados em uma especificação LOTOS encontra-se pulverizada nas definições dos processos. De modo semelhante a um procedimento em uma linguagem de programação (p. ex. PASCAL), cada processo poderá definir tipos de dados locais. O processo **specification** estabelece o escopo mais externo, por este motivo os tipos introduzidos por ele serão globais para todos os processos presentes na especificação.

A construção de uma estrutura concreta que represente uma estrutura de dados abstratos deverá levar em conta:

*i*) os tipos primitivos referenciados;

*ii*) as operações (OPNS), que identificam as variáveis e funções definidas pelos novos tipos;

*iii*) as equações (EQNS), que definirão as funções introduzidas;

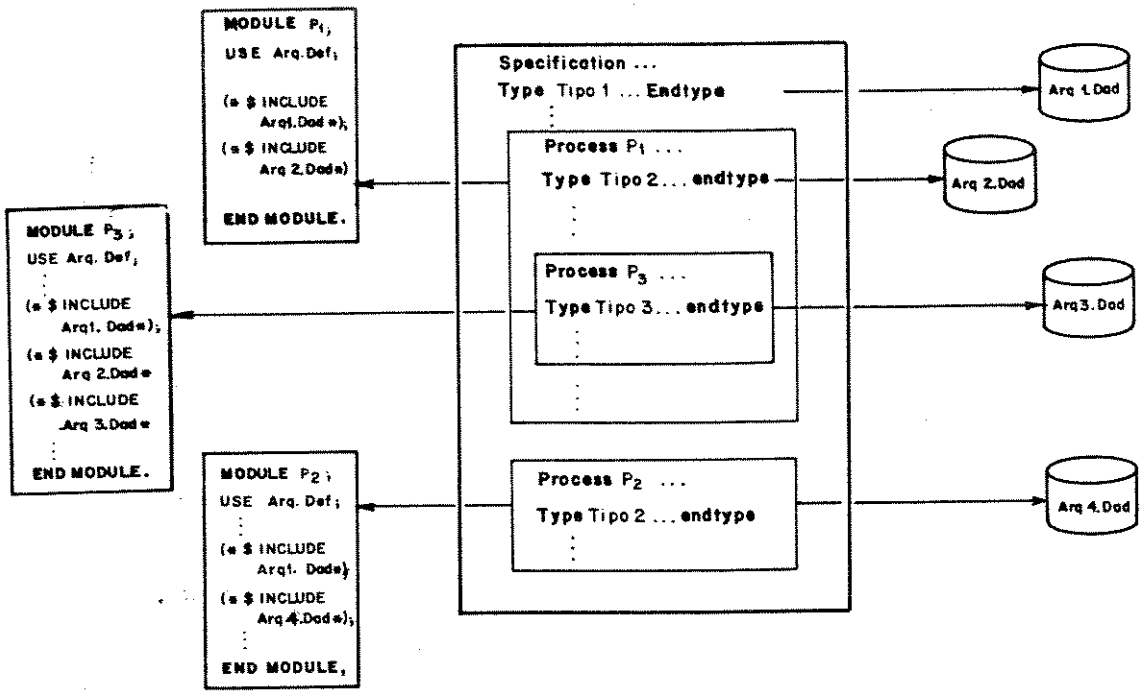
O modelo de implementação descrita nesta seção é apenas uma proposta (usada no exemplo do capítulo 6). Como já foi dito, neste trabalho não foram abordadas as questões relativas às estruturas de dados, por isto esta proposta não foi submetida a nenhuma análise detalhada de consistência.

É proposto o seguinte procedimento para a derivação de estruturas concretas:

*i*) A estrutura de dados definida (tipos, variáveis e funções) pelo processo **specification**, serão traduzidos em tipos concretos associados ao arquivo de dados **Arq1.Dad**.

*ii*) As estruturas decladas em outros processos serão traduzidos para arquivos separados (**Arq2.Dad**,...). Estes serão incluídos nos módulos de acordo com o escopo a que pertencem os processos que representam.

A seguir é apresentado um exemplo no procedimento de tradução da estrutura de dados.




A definição de tipos elementares seguirá o mesmo procedimento. Assim, o arquivo que representa uma determinada estrutura fará uso de arquivos auxiliares relativos aos tipos elementares.

## 5.3 - Estrutura de execução do Módulo Gerenciador

Na seção 5.2.4 vimos que o comportamento principal do Módulo Gerenciador é composto por um conjunto de procedimentos que realizam ações definidas dentro do modelo de execução considerado. Nesta seção descreveremos a estrutura lógica de cada um destes procedimentos.

### 5.3.1 - Construção da Árvore de Composição

O Ciclo de Execução do Módulo Gerenciador inicia-se com a recepção das mensagens enviadas pelo Módulo Especificação para a criação da Árvore de Composição inicial (fig. 5.3.1).

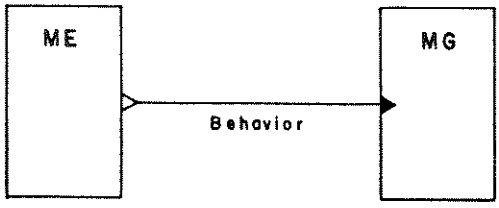



Figura 5.3.1 - Troca de mensagens para criação da árvore de composição

A árvore poderá ser expandida ou podada de acordo com a execução de funções de instanciação pelos Módulos Básicos (fig. 5.3.2) ou pelo tratamento de desabilitações ou terminações.

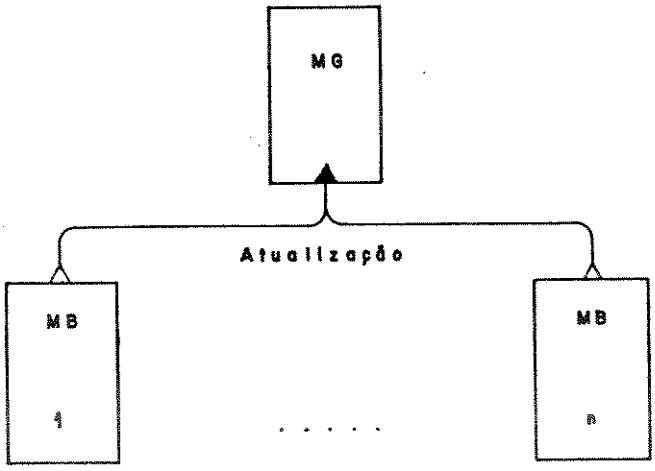


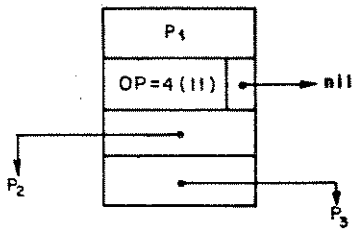

Figura 5.3.2 - Atualização da Árvore de Composição

**. Estrutura de implementação da Árvore de Composição**

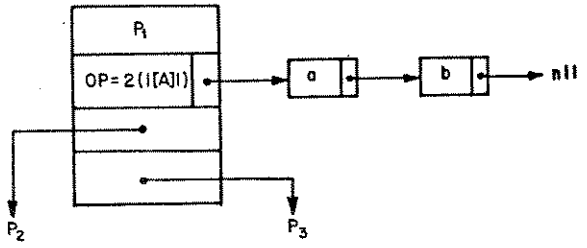
A Árvore de Composição foi implementada a partir da definição da estrutura dos nós. Cada nó é composto por um conjunto de informações. Um campo identifica o nome do processo descrito no nó. Um segundo campo é usado para identificar o operador LOTOS associado ao nó. Existem ainda dois campos que contêm apontadores para as sub-árvores criadas pelo nó. No caso de nós terminais os três últimos campos são preenchidos com valores nulos ($\nu(\iota\iota)=\emptyset$, $\nu(\iota\iota\iota) = \nu(\iota\nu) =$ **nil**).

A seguir serão apresentadas algumas expressões de comportamento LOTOS e a sua representação em forma de árvore.

$\rightarrow P_1 := P_2 \;||\; P_3$




$\rightarrow P_1 := P_2 \;|[a,b]|\; P_3$



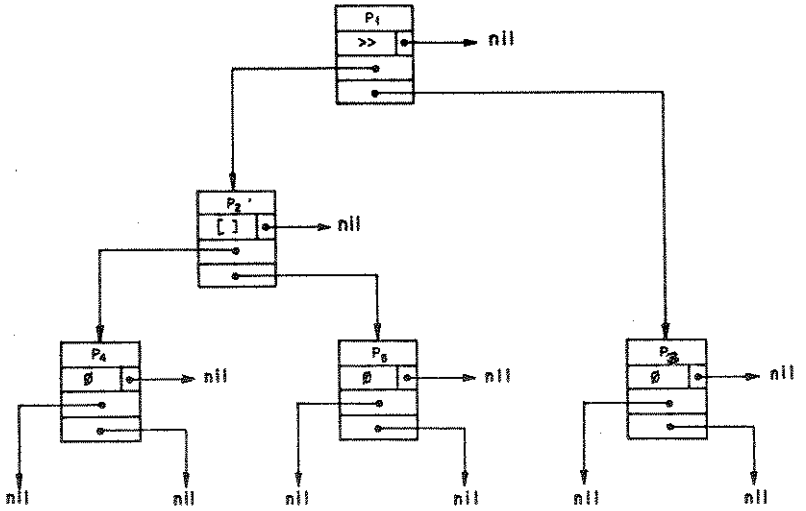

$\rightarrow P_1 = P_2 >> P_3$

$P_2 := P_4 \;[]\; P_5$

$P_4 := \ldots$

$P_5 := \ldots$

$P_3 := \ldots$



## .Procedimento de Criação da Árvore

Sendo as mensangens recebidas do Módulo Especificação no formato

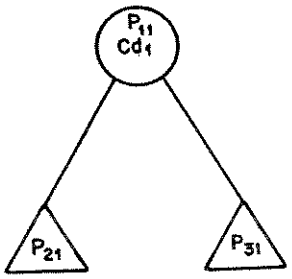
$$M_i = < cd_i , P_{1i} , P_{2i} , P_{3i} >$$

onde i é a ordem da mensagem.

i) Recepção da primeira mensagem.

$$M_1 = < cd_1 , P_{11} , P_{21} , P_{31} >$$

ii) Inicialização da árvore na forma:

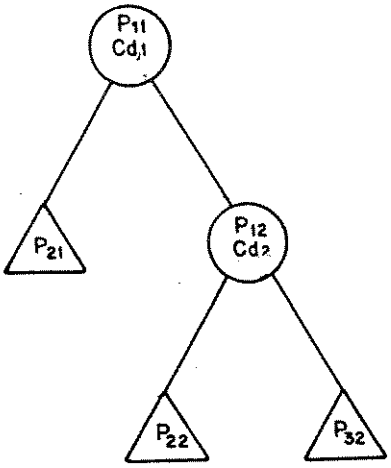



iii) A cada nova mensagem $M_i$ recebida, expandir a árvore do ponto que identifique $P_{1i}$ com $P_{2(i-1)}$ ou $P_{3(i-1)}$

Seja a mensagem

$$M_2 = < cd_2 , P_{12} , P_{22} , P_{32} > \text{ e}$$
$$P_{12} = P_{31}$$

Expandir a árvore inicial para:

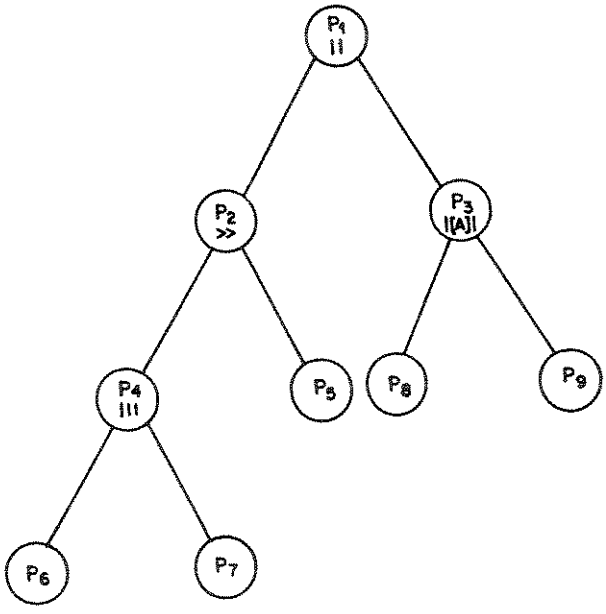



→ **Exemplo**

Seja o conjunto de mensagens recebidas pelo Módulo Gerenciador:

$$M_1 = < \,||\,, P_1, P_2, P_3 >$$
$$M_2 = < >>, P_2, P_4, P_5 >$$
$$M_3 = < \,|||\,, P_4, P_6, P_7 >$$
$$M_4 = < \,|\,[A]\,|, P_3, P_8, P_9 >$$

Após o tratamento das quatro mensagens teremos a árvore:



. **Procedimento de Expansão**

O procedimento que executa a expansão é o mesmo usado na construção da árvore, ou seja, expandir a folha identificada pelo primeiro processo da mensagem ($P_{11}$).

. **Procedimento de Poda**

A Árvore de Composição será percorrida no modo **postorder** [WIRTH 76], sendo o procedimento de avaliação de cada nó o seguinte:

I - Processo descrito pela sub-árvore esquerda participou do evento?

< Sim > → Operador associado ao nó é:

< [ ] > → .Desabilitar o processo relativo à sub-arvore direita;
.Atualizar a Árvore de Composição;
.Passo III.

< outros > → Ocorreu pseudo-evento?

< Sim > → Atualizar Árvore de Composição;

< Não > → Passo III

< Não > → Passo II

II - Processo descrito pela sub-árvore direita participou do evento?

< Sim > → O operador associado ao nó é:

< [ ] ou [ > > → .Desabilitar o processo relativo à sub-árvore esquerda;
.Atualizar a Árvore de Composição;
.Passo III

< outros > → Ocorreu pseudo-evento?

< Sim > → Atualizar Árvore de Composição;

< Não > → Passo III

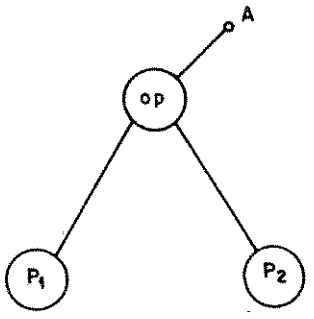
< Não > → Passo III

III-.Atualizar a informação quanto à participação, ou não, do processo relativo ao nó, no evento;
.Encerrar pesquisa recursiva do nó.

A atualização da Árvore de Composição se dará conforme as regras de atualização descritas no apêndice A.

→ **Exemplo**

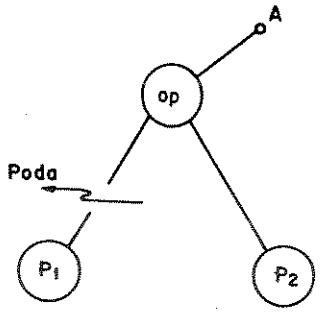
.Seja a árvore



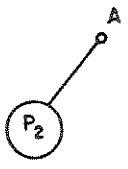

onde **op** é um dos operadores >>, [> ou []

A terminação de $\mathbf{P}_1$ com sucesso - exit - (no caso de **op** = >>) ou a participação de $\mathbf{P}_2$ em um evento (no caso de **op** = [] ou **op** = [>), produzirá as seguintes modificações na árvore:

i) Poda de $\mathbf{P}_1$ por razões de desabilitação ou terminação;




ii) Substituição do nó interno (relativo a **op**) por $\mathbf{P}_2$ (ver apêndice A);



## 5.3.2 - Ativação de Módulos Básicos

Após a criação ou atualização da Árvore de Composição, o Módulo Gerenciador entrará na fase de ativação.

Todos os Módulos Básicos ainda não ativados serão habilitados, menos aqueles ligados às sub-árvores direitas nascidas de nós que descrevem operadores ENABLE (>>) - p.ex.: $MB_2$ na fig. 5.3.3.

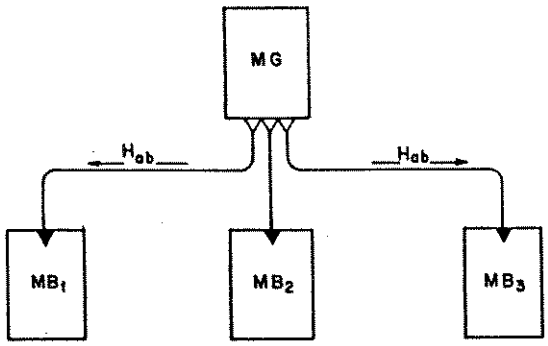



Figura 5.3.3 - Ativação de Módulos Básicos

**. Procedimento**

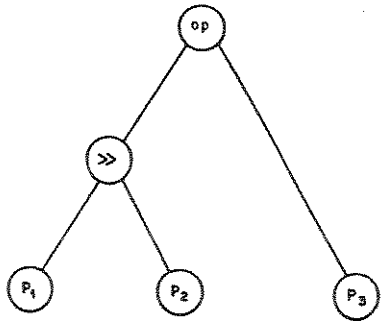
I) Analisar a árvore na forma de um procedimento Preorder [WIRTH 76];

.Percorrer sub-árvores direitas somente se os nós aos quais estão ligadas não estiverem associadas a operadores ENABLE (>>).

II) Enviar mensagem de habilitação aos Módulos Básicos ainda não ativado.

**→ Exemplo**

Seja a árvore



onde: **op** é um dos operadores |[A]|, | |, | | |, [> ou []. Após a execução do procedimento de habilitação, os módulos $P_1$ e $P_3$ serão ativados (cnf fig. 5.3.3).

Após a execução do procedimento de habilitação, o Módulo Gerenciador reduzirá sua prioridade. Com isto os módulos que foram ativados entrarão em execução. Este procedimento não implicará em nenhuma alteração nos módulos anteriormente ativados e que estejam bloqueados devido à ofertas de eventos pendentes.

## 5.3.3 - TABELAS AUXILIARES

Além da Árvore de Composição foram definidas duas tabelas para auxiliar o Módulo Gerenciador nas diversas análises realizadas.

**.Tabelas de Ofertas de Eventos (OE) por processo**

Após a recepção das OEs, o Módulo Gerenciador cria uma tabela onde constará, para cada processo, uma lista de suas OEs.

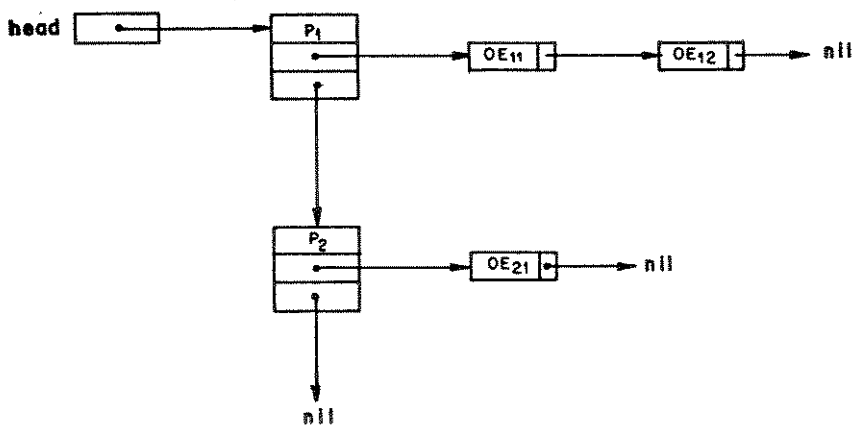



Figura 5.3.4 - Tabela de OEs por processos

O conteúdo desta tabela será visto pelo Módulo Gerenciador como o

conjunto de processos possíveis de participarem em um evento a cada ciclo de execução.

Quando o Módulo Gerenciador estiver na fase de avaliação das ofertas, para cada Módulo Básico encontrado na árvore será feita uma pesquisa nesta tabela para obter a sua lista de OEs (seção 4.5.2).

Na fase de tratamento da ocorrência de evento quando for determinada a participação de um processo em uma intera-ção, a sua entrada nesta tabela será retirada.

**.Tabelas de processos por portas**

Esta tabela será criada para relacionar todos os processos que fazem oferta em uma determinada porta. Foi considerado um formato semelhante ao da fig. 5.3.4.

Segundo o nosso modelo de execução, ao final da fase de tratamento da ocorrência de evento, será conhecida apenas a porta e as condições em que ocorreu o evento (valores negociados). Para que sejam identificados os processos que participaram da interação, a tabela de processos por portas será pesquisada.

A porta onde ocorreu uma interação deverá ter sua entrada, nesta tabela, deletada. O restante da tabela permanecerá inalterada para o próximo ciclo de execução.

### 5.3.4 - ANÁLISE DAS OFERTAS DE EVENTOS COM BASE NA ÁRVORE DE COMPOSIÇÃO

Durante esta fase, as ofertas serão avaliadas quanto à semântica dos operadores LOTOS associados aos nós da Árvore de Composição. Ao final desta fase será obtida uma lista com todos os eventos cuja ocorrência é possível. Somente na fase seguinte é que será definida a ocorrência de um dos eventos desta lista. Assim, todos eventos ofertados serão considerados sem que seja feito nenhum tipo de escolha (esta observação visa esclarecer como será considerada a característica de exclusão nos operadores |||, [> e []).

**Procedimento**

I) Realizar busca do tipo postorder a partir da raiz até ao nível das folhas;

II) Para cada folha encontrada, recuperar da tabela de ofertas de eventos por processo a lista de OEs relativa ao processo associado à folha;

II) Para cada nó interno, derivar uma lista resultante a partir das listas apresentadas pelas suas sub-árvores.

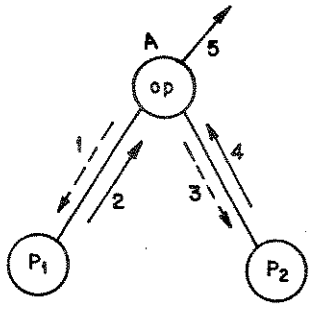
A vantagem da estrutura da árvore para a análise realizada por este procedimento é que:

*i*) Qualquer nó considerará as sub-árvores nascidas de si como se fossem Módulos Básicos;

*ii*) O resultado de cada chamada recursiva será a oferta de uma lista de eventos, podendo ser uma lista nula no caso de não haver nenhuma oferta associada ao processo ou sub-árvore pesquisada.

→ **Exemplo 1**

Seja a árvore:




onde:

.A é um nó interno ao qual está associado um dos operadores LOTOS de composição (op);

.$P_1$ e $P_2$ são Módulos Básicos;

.A seta pontilhada (⟶) representa uma busca recursiva;

.A seta cheia (⟶) representa a resposta a uma busca recursiva;

A análise sobre o nó **A** será feita na seguinte forma:

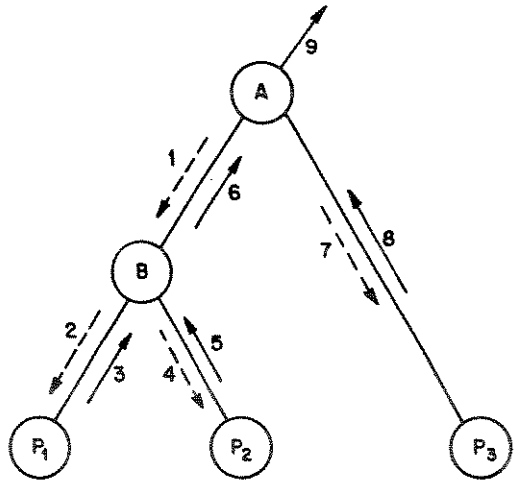
ι) Identificar a lista de eventos ofertados por $\mathbf{P}_1$ (avaliação da sub-árvore esquerda descrita para sequência de setas 1 e 2);

ιι) Identificar a lista de oferta de eventos de $\mathbf{P}_2$ (avaliação da sub-árvore direita, setas 3 e 4).

ιιι) Elaborar uma lista de $OE_s$ resultante e apresentá-la como resposta para a análise recursiva do nó ao qual está associada esta sub-árvore.

**Exemplo 2**

Seja agora a árvore:




A árvore será pesquisada na seguinte forma:

.Como o nó A não é terminal, então analisar sub-árvores:

.(sub-árvore esquerda)_A ≡ nó B. Como o nó B não é terminal, analisar sub-árvores:

.(sub-árvore esquerda)_B ≡ folha P1. Como é um Módulo Básico, então:

.Identificar a lista de OEs de P1($LOE_{P1}$) e atribuí-la à lista de OEs da sub-árvore esquerda ($LOE_{sae}$);

$$LOE_{sae} = LOE_{P1}$$

.Apresentar resultado ($LOE_{sae}$) ao nó B;

.(sub-árvore direita)_B ≡ folha P2. Como P2 é básico:

.Associar $LOE_{P2}$ à lista de OEs da sub-árvore esquerda ($LOE_{sad}$);

$$LOE_{sad} = LOE_{P2}$$

.Apresentar $LOE_{sad}$ ao nó B;

.Determinar lista de OEs resposta do nó B e apresentar ao nó A. Como o nó B é a raiz da sub-árvore esquerda de A, a lista de OEs resultante será apresentada como $LOE_{sae}$

$$LOE_{sae} = \text{Derivação } (LOE_{sae}, LOE_{sad}, op_B)$$

.(sub-árvore direita)_A ≡ folha P3 Como P3 é básico:

.$LOE_{sad} = LOE_{P3}$
.Apresentar resultado ($LOE_{sad}$) ao nó A

.Determinar

$$LOE_{sae} = \text{Derivação } (LOE_{sae}, LOE_{sad}, op_A)$$

.Apresentar lista resultante ($LOE_{sae}$ - ↗9) à fase seguinte - tratamento da ocorrência de evento.

O modelo de derivação - seção 4.5.2 - foi implementado segundo os procedimentos descritos a seguir.

**.Busca e derivação de listas de OEs**

O objetivo deste procedimento é derivar uma lista terminal onde serão apresentados todos os eventos possíveis de ocorrer baseado nas condições impostas pela semântica dos operadores LOTOS distribuídos na árvore.

**Procedimento**

I) Por meio de uma busca postorder, varrer a árvore para definição das listas de $OE_s$ associadas a cada nó.

Sempre a partir de duas listas obtidas junto às sub-árvores, será gerada uma terceira lista de acordo com a semântica do operador LOTOS descrito no nó.

II) No caso de nós terminais, derivar lista ($LOE_{sad}$ ou $LOE_{sae}$) diretamente das listas de ofertas dos processos (tabela de ofertas de eventos por processo);

III) No caso de nós internos, um procedimento, específico para cada operador, será chamado para tratar a derivação.

**.Derivação de listas em nós terminais**

Durante o procedimento de busca das listas na Árvore de Composição, ao se atingir um nó terminal será criada uma nova lista ($LOE_{sad}$ ou $LOE_{sae}$ - dependendo da posição do nó na árvore, como mostra a fig. 5.3.5 -) que será simplesmente uma cópia da lista de $OE_s$ - a partir da tabela de ofertas de eventos por processos - do processo identificado no nó, ou

$$LOE_{sad/e} = LOE_{Proc}$$

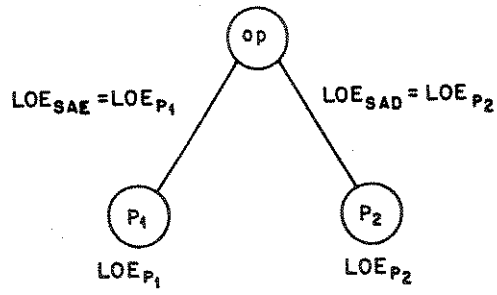



Figura 5.3.5 - Derivação de lista em nó terminal.

**.Derivação de listas em nós não terminais**

A derivação de listas em nós não terminais será regida pela semântica do operador associado ao nó e serão usadas nos procedimentos de derivação.

**..Paralelismo com sincronização restrita (|[A]|)**

*i*) Para cada elemento de $LOE_{sae}$ cuja porta se encontra em [A], verificar se existe em $LOE_{sad}$ alguma OE na mesma porta. Neste caso, para cada ocorrência, derivar oferta na forma abaixo e incluir na lista resultante - $LOE_{sa}$ - (fig. 5.3.6).

SE matched ($OE_1$,$OE_2$)
ENTÃO
    OE = Derivação ($OE_1$,$OE_2$)

Onde:

$OE_1 \in LOE_{sae}$ ;
$OE_2 \in LOE_{sad}$

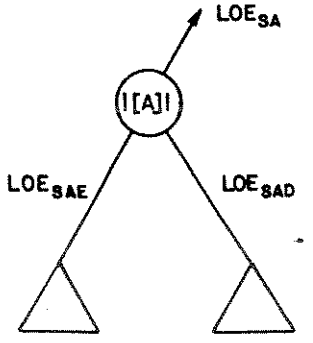



Figura 5.3.6 - Derivação de Lista em nó com operador de paralelismo restrito |[A]|

*ii*) As $OE_s$ de $LOE_{sad}$ e $LOE_{sae}$ não satisfeitas em (i), devem ser incluídas em $LOE_{sa}$.

Na fig. 5.3.7 temos um fluxograma detalhado deste procedimento.

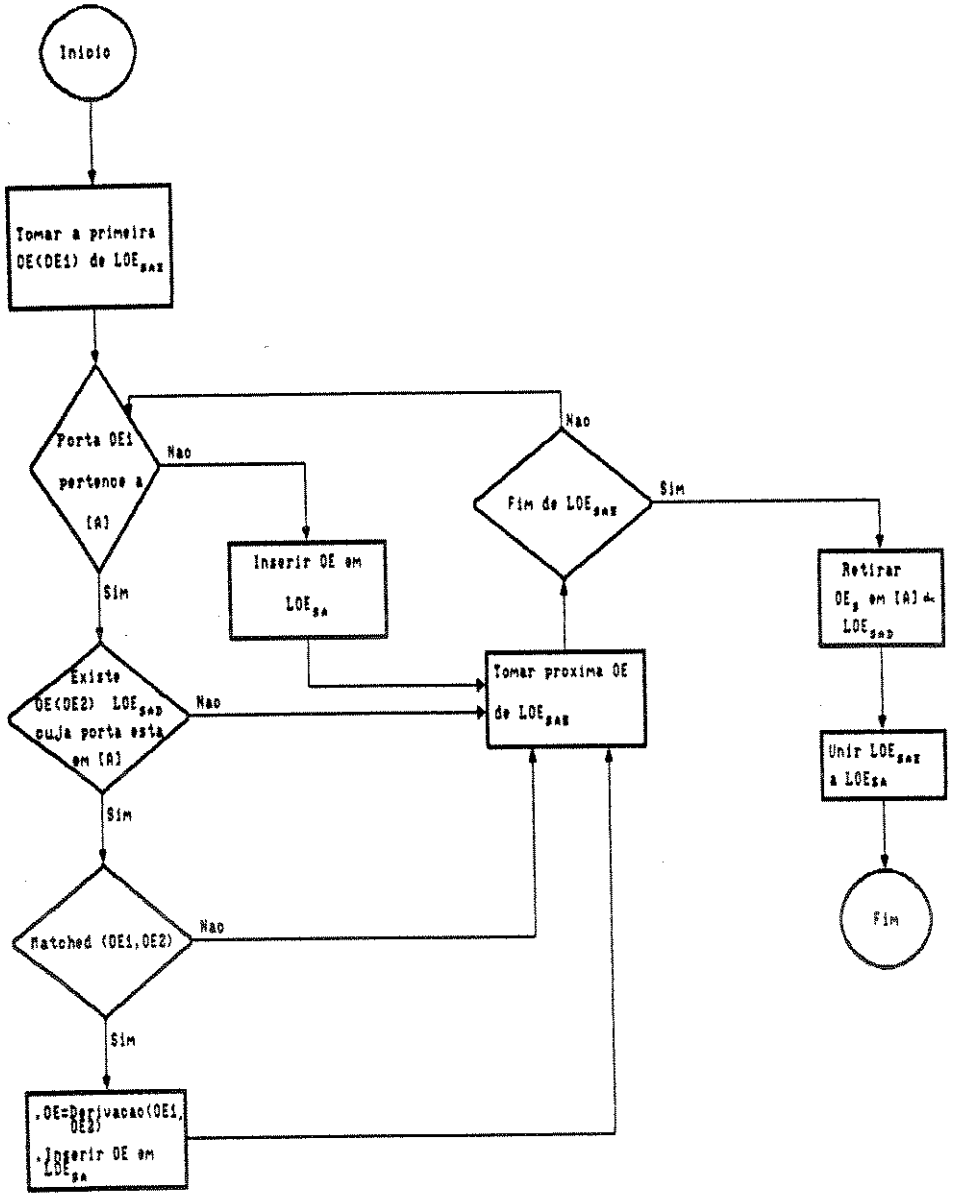



Figura 5.3.7 - Fluxograma do procedimento de derivação de lista em nó com paralelismo restrito - $|[A]|$

**..Paralelismo com sincronização completa**

No caso de nós com operador de paralelismo com sincronização em todas as portas, o procedimento para derivação de lista será um caso particular do procedimento anterior, ou seja:

I) Para cada OE de $LOE_{sae}$, verificar ofertas na mesma porta presentes em $LOE_{sad}$. Neste caso derivar oferta na forma discutida no caso anterior;

II) As OEs não tratadas por (i) devem ser desconsideradas.

O fluxograma mostrado na Fig. 5.3.7 pode ser usado neste caso, desde que feitas as seguintes considerações:

i) Como a sincronização deve ser completa, não haverá testes sobre conjunto de portas;

ii) Só haverá insersão de OE em $LOE_{sa}$ no caso de ofertas derivadas.

**..Paralelismo sem sincronização(|||), Disrupt ([>) e Choice ([])**

Estes três operadores serão tratados de modo idêntico devido à semelhança de suas regras de avaliação (cnf. regra (ii) para nós não-terminais da seção 4.5.2), sendo ainda um caso particular do procedimento de tratamento do operador de sincronização restrita.

Neste caso, as duas listas - $LOE_{sad}$ e $LOE_{sae}$ - serão unidas e atribuídas a $LOE_{sa}$, como

$$LOE_{sa} = LOE_{sae} \cup LOE_{sad}$$

**.Enable (>>)**

Este operador será tratado pela atribuição direta da lista de OEs da sub-árvore direita à lista derivada, na forma

$$LOE_{sa} = LOE_{sad}$$

isto porque só poderão existir processos ativos associados à sub-árvore esquerda.

→ **Exemplo 3**

Seja a especificação LOTOS

```
specipication Exemplo [g1,g2,g3,g4,g5,g6] : noexit
 type ... endtype
 behavior
     P1[g1,g2,g3,g4,g5]
     |[g1,g3,g4,g5]|
     P2[g1,g3,g5,g6]
  where
   process P1[a,b,c,d,f] : noexit :=
     P3[d,f]
     |||
     P4[a,b,c]
    where
     process P3[a,b] : noexit :=
         let y:int = 1 in
           ( a?x:int;P3[a,b]
             []
             b!y;P3[a,b]
           )
     endproc
     process P4[a,b,c] : noexit :=
         P5[a,b,c](1,2,3)
         ||
         P6[c](3)
      where
       process P5[a,b,c](x,y,z:int) : noexit :=
             a!x;P5[a,b,c](x,y,z)
          [] b!y;P5[a,b,c](x,y,z)
          [] c!z;P5[a,b,c](x,y,z)
       endproc
```

```
        process P6[a](x:int) : noexit :=
          a!x;P6[a](x)
        endproc
      endproc
    endproc
    process P2[a,b,c,d] : noexit :=
      P7[a,b]
      [>
      P8[c,d](2)
     where
      process P7[a,b] : noexit :=
          let x:int = 3 in
            ( a!x:int;P7[a,b]
              []
              b?y:int;P7[a,b]
            )
      endproc
      process P8[a,b](x:int) : noexit :=
            a?y:int;P8[a,b](x)
          [] b!x;P8[a,b](succ(x))
      endproc
    endproc
endspec
```

A árvore de composição que descreve a estrutura da especificação é representada na figura 5.3.8.

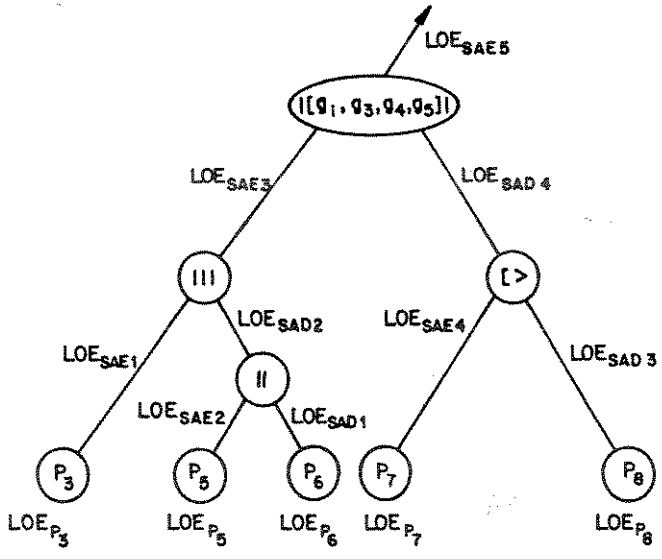



Figura 5.3.8 -Árvore de Composição referente à especificação.

De acordo com a regra para derivação de lista de OEs em nós terminais, temos;

$$LOE_{P3} = \{<g_4?int>,<g_5!1>\}$$

$$LOE_{P5} = \{<g_1!1>,<g_2!2>,<g_3!3>\}$$

$$LOE_{P6} = \{<g_3!3>\}$$

$$LOE_{P7} = \{<g_1!3>,<g_3?int>\}$$

$$LOE_{P8} = \{<g_5?int>,<g_6!2>\}$$

Conforme a descrição do exemplo 2 , a análise das OEs segundo a estrutura da Árvore de Composição será:

1. $LOE_{sae1} = LOE_{P3}$

2. $LOE_{sae2} = LOE_{P5}$

3. $LOE_{sad1} = LOE_{P6}$

4. $LOE_{sad2} = Derivação\ (LOE_{sae2}, LOE_{sad1}, ||)$

   $= Derivação\ (\{<g_1!1>,<g_2!2>, <g_3!3>\},\{<g_3!3>\},||)$

   $= \{<g_3!3>\}$

5. $LOE_{sae3} = Derivação\ (LOE_{sae1}, LOE_{sad2}, |||)$

   $= Derivação\ (\{<g_4?int>,<g_5!1>\}, \{<g_3!3>\},|||)$

   $= \{<g_4?int>, <g_5!1>,<g_3!3>\}$

6. $LOE_{sae4} = LOE_{P7}$

7. $LOE_{sad3} = LOE_{P8}$

8. $LOE_{sad4} = Derivação\ (LOE_{sae4}, LOE_{sad3}, [>)$

   $= Derivação\ (\{<g_1!3>,<g_3?int>\}, \{<g_5?int>,<g_6!2>\},[>)$

   $= <g_1!3>,<g_3?int>\},\{<g_5?int>, <g_6!2>$

9. $LOE_{sae5}$ = Derivação ($LOE_{sae3}$, $LOE_{sad4}$, $|[A]|$)

= Derivação ({<$g_4$?int>,<$g_5$!1>, <$g_3$!3>},{<$g_1$!3>,<$g_3$?int>, <$g_5$?int>,<$g_6$!2>}, $|[g_1,g_3,g_4,g_5]|$)

= {<$g_5$!1>, <$g_3$!3>, <$g_6$!2>}

## 5.3.5 - TRATAMENTO DA OCORRÊNCIA DE EVENTO

Nesta fase será tratada a definição da ocorrência de um evento, ou seja, a partir da lista de $OE_s$ derivada do primeiro nível de recursão (análise da raiz) será escolhido um dos eventos.

**. Procedimento**

i) Escolher aleatoriamente, ou em interação com o usuario, um dos eventos pertecente à lista de $OE_s$ final;

ii) Consultar a tabela de processos por porta e identificar os módulos envolvidos no evento escolhido;

iii) Chamar procedimento de atualização da Árvore de Composição - Seção 5.3.1;

iv) Chamar serviço para envio de mensagem resposta aos módulos envolvidos no evento escolhido - Seção 5.3.6;

v) Atualizar a tabela de processos por porta, retirando:

v.1) Entrada relativa à porta onde ocorreu evento;

v.2) Todas as ocorrências, nas outras entradas, dos módulos partipantes do evento escolhido;

vi) Atualizar a tabela de $OE_s$ por processo, retirando as entradas associadas aos módulos participantes da interação;

Considerando o exemplo 3 da seção 5.3.4 e supondo que:

1) O evento escolhido seja

.$<g_5$ !1>

.Processos Envolvidos - segundo a avaliação da tabela de processos por portas (fig. 5.3.10)

**P3 e P8**

.Chamar procedimento de atualização da Árvore de Composição

.**P7** deve ser desabilitado

.Chamar serviço para envio de mensagem_resposta de desabilitação;

.Atualizar a Árvore de Composição - que passará da que é mostrada na Fig. 5.3.8 para a da Fig. 5.3.9, segundo o procedimento de poda - seção 5.3.1.

.Chamar o procedimento de envio de mensagem_resposta aos processos envolvidos na interação.

. A tabela de processos por porta - que tinha inicialmente o formato da fig. 5.3.10 - passará para a que é mostrada na fig 5.3.11, tendo sido retirado $g_5$ segundo (v.1), $g_4$ e $g_6$ devido a (v.2) e P7 de acordo com procedimento de atualização da Árvore de Composição por ocorrência de desabilitação - Seção 5.3.1.

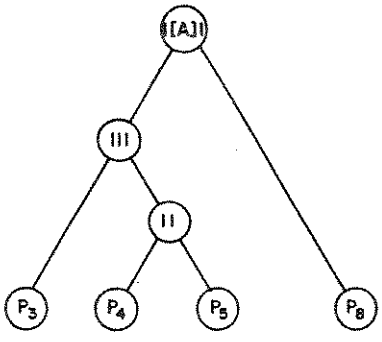



Figura 5.3.9 - Árvore de composição atualizada com a ocorrência de $< g_5$ !1 >

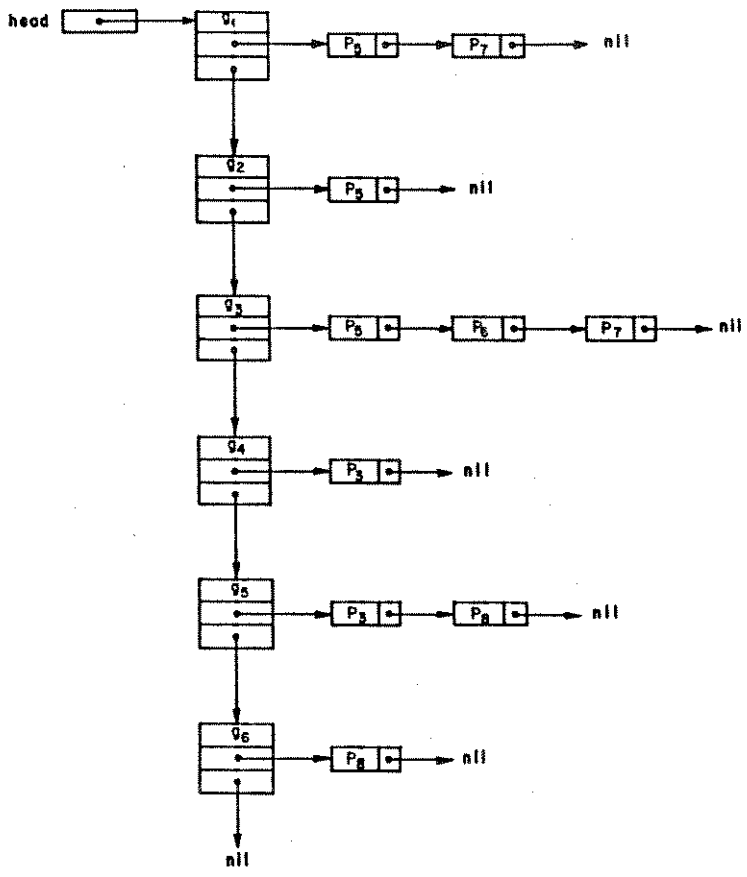



Figura 5.3.10 - Tabela de processos por portas inicial relativa ao exemplo 3 da seção 5.3.4.

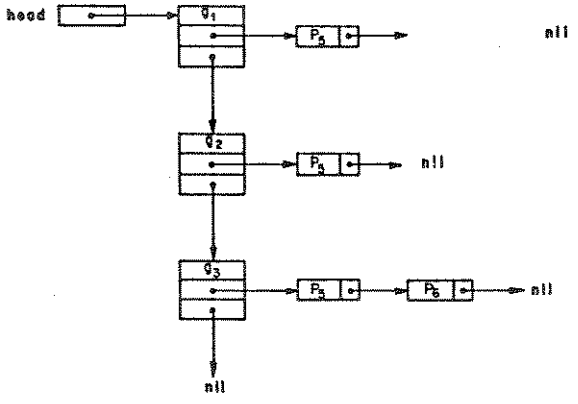



Figura 5.3.11 - Tabela de processos por porta modificada pela ocorrência do evento $< g_5 ! 1 >$

.A Tabela de $OE_s$ por processos que inicialmente tinha a forma apresentada na fig.5.3.12, resutará na forma mostrada na Fig. 5.3.13, onde $P_3$ e $P_8$ foram retirados de acordo com (vi) e P7 devido ao tratamento da desabilitação - Seção 5.3.1.

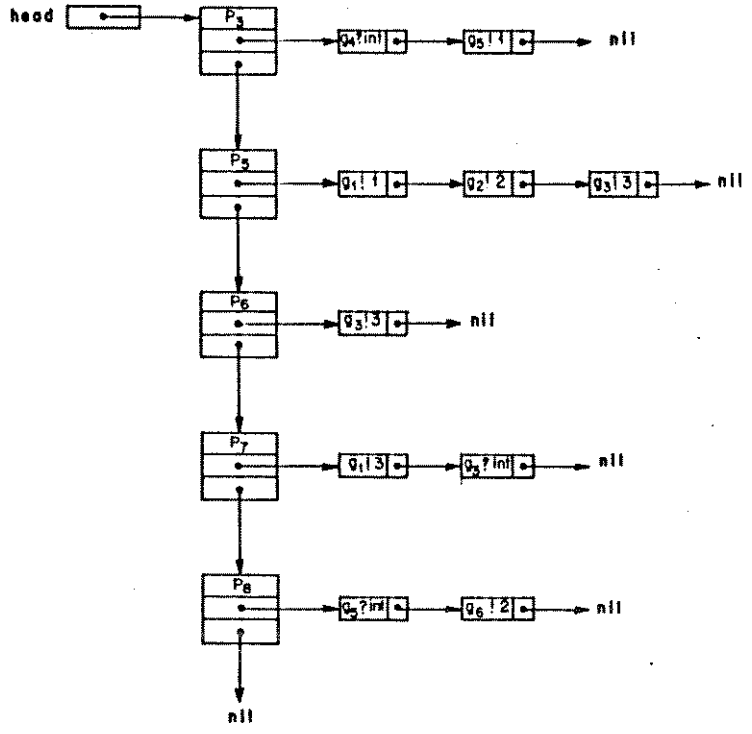



Figura 5.3.12 - Tabela de OEs por processo inicial relativa ao exemplo 3 da seção 5.3.4.

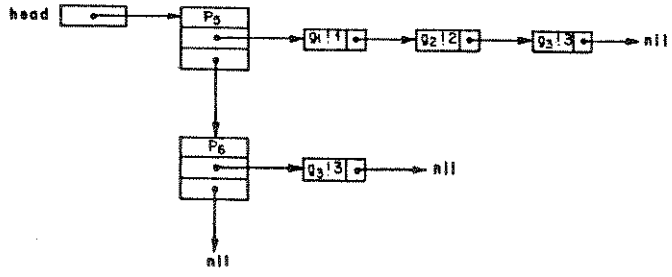



Figura 5.3.13 - Tabela de OEs por processo modificada pela ocorrência do evento < $g_5$!1 >

2) Supondo agora a ocorrência do evento

.$<g_3!3>$

.Processos envolvidos

**P5, P6 e P7**

.Não haverá atualização na Árvore de Composição;

.Chamar o procedimento de envio de mensagem_resposta aos processos envolvidos na interação.

A tabela de processos por porta da fig. 5.3.10, terá a forma mostrada na fig. 5.3.14;

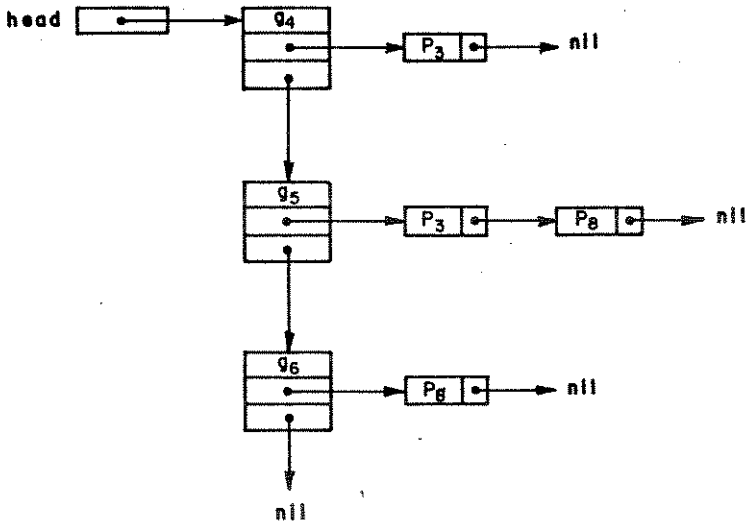



Figura 5.3.14 -Tabela de processos por porta modificada pela ocorrência do evento $< g_3 ! 3 >$

A Tabela de $OE_s$ por processo será alterada da que aparece na fig. 5.3.12 para a que é mostrada na Fig. 5.3.15;

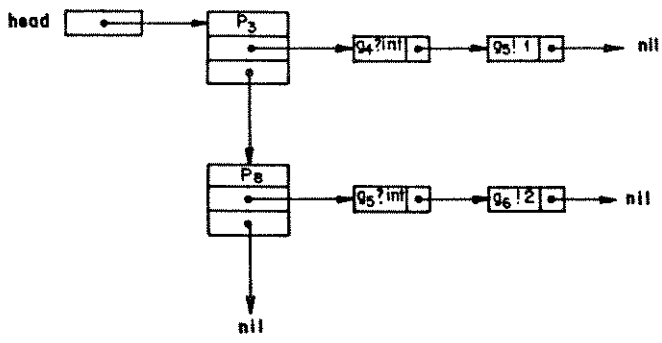



Figura 5.3.15 - Tabela de OEs por processo modificada pela ocorrência do evento < $g_3$!3 >

## 5.3.6 - Envio de mensagens_resposta aos Módulos Básicos

A última fase do ciclo de execução do Módulo Gerenciador será o envio de mensagens_resposta aos Módulos Básicos que participaram de uma interação, ou que se envolveram em alguma situação de desativação.

Através das mensagens de participação em interação, os Módulos Básicos recebem as condições negociadas na ocorrência do evento.

Por questão de simplificação, as mensagens de confirmação de terminação (aquelas que informam aos módulos que o procedimento de finalização foi concluído com sucesso) foram consideradas como sendo do mesmo tipo das mensagens de desabilitação.

Na fig. 5.3.16 encontramos o esquema de ligação de portas entre o Módulo Gerenciador e os Módulos Básicos para o envio das mensagens_resposta de participação em interação.

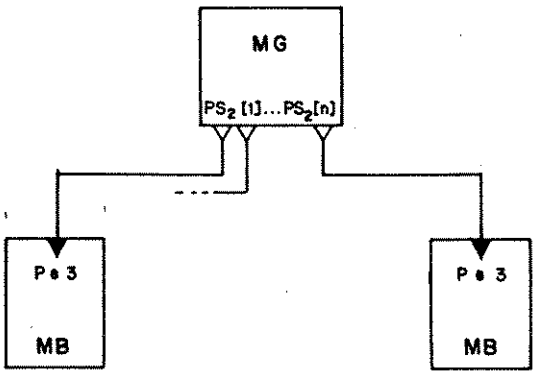



Figura 5.3.16 - Mensagem Resposta de Participação em interação.

O esquema de ligação para o caso de mensagens de desabilitação será idêntico ao da Fig. 5.3.16, sendo que um novo conjunto de portas foi definido para o Módulo Gerenciador ($PS_3[1]...PS_3[n]$ ) e uma nova porta para os Módulos Básicos ( $PE_2$).

Após o envio de todas as mensagens associadas à ocorrência de evento, desabilitação ou confirmação de terminação, o Módulo Gerenciador diminuirá sua prioridade permitindo assim que os Módulos Básicos recebam as mensagens_resposta e continuem suas execuções. O módulo que não participou do evento continuará bloqueado (cnf. função SINC, seção 5.1.1) ainda à espera de ocorrência de um evento segundo o seu conjunto de ofertas pendentes.

No próximo capítulo será apresentada a execução de uma especificação LOTOS, que será importante para demonstrar as características de simulação existentes no ambiente LSTER.

# Capítulo 6

# Exemplo de Aplicação

A especificação apresentada nesta seção foi adaptada de [LOGRIPPO 88] que descreve um serviço provedor para a interligação de dois usuários.

A estrutura geral do sistema pode ser vista na fig. 6.1.1.

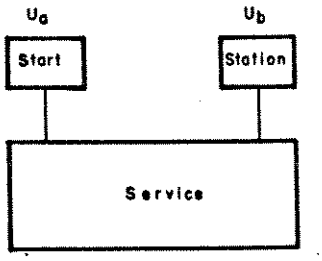



Figura 6.1.1 - Estrutura geral do Serviço Provedor.

Para se obter as características de prototipagem/simulação, a especificação foi descrita de uma forma completa (ou fechada) onde o serviço é composto paralelamente com dois usuários - $U_a$ **(start)** e $U_b$ **(station)** - para a geração espontânea de eventos.

```
hide g,d in
( start [g] (1)
|||
  station [d]
)
|[d,g]|
 service [g,d]
```

Neste exemplo é considerada somente a hipótese do usuário $U_a$ enviar dados ao usuário $U_b$. Por questões de simplificação $U_a$ apenas inicializa (start) o procedimento de envio - observar que o processo $U_a$ é instanciado com o parâmetro 1 que indica o valor inicial da sequência - sendo a sequência de dados gerada internamente pelo serviço. $U_b$ é uma estação remota sumidora de dados.

```
process start [inic] (mes_inic:int) :noexit :=
     inic!mes_inic; stop
endproc

process station [deliv] :noexit :=
     deliv?mes:int; station[deliv]
endproc
```

O processo **service** é constituído por duas entidades pares (sender e receiver) que se comunicam por um canal (processo crazyfifo) sujeito a falhas e à consequente perda de dados. O esquema de composição dos processos considera o processo crazyfifo como responsável pelo serviço n-1 que dá suporte ao serviço pres-tado pelo processo service (serviço n).

```
process service [go,deliv] :noexit :=
     go?mes_inic:int;
     hide a,b,c,e,f,h in
       ( sender[a,c,f](mes_inic,0)
       |||
         receiver[b,e,h,deliv](1)
       )
       |[a,b,c,e,f,h]|
       crayfifo [a,b,c,e,f,h] (new)
   where
     .
     .
     .
endproc
```

Três comportamentos básicos compõem a estrutura do processo **sender**. A partir do recebimento de um novo crédito, enviado pelo processo receiver, terá início o procedimento de envio de dados. A cada dado enviado, o crédito é decrementado e o próximo dado na sequência é preparado para o envio. Devido à possibilidade de perdas de dados, uma solicitação de retransmissão poderá ocorrer a qualquer momento.

```
process sender [out,req,retry] (mes:int,cred:int):noexit :=
       [gt(cred,zero)] -> out!mes;
                          sender[...](succ(mes),pred(cred))
    [] [eq(cred,zero)] -> req?new_cred:int;
                          sender[...](mes,new_cred)
    [] retry?expect:int; sender[...](expect,cred)
endproc
```

O processo **crazyfifo** é composto de dois processos. Um deles (fifo) descreve o comportamento da fila e o outro (lostdata) a perda de dados no canal de comunicação.

```
process crazyfifo [mes_in,mes_out,cred_in,cred_out,
                   retry_in, retry_out] (Q:queue):noexit :=
     hide p in
          fifo[mes_in, mes_out, cred_in,
               cred_out, retry_in, retry_out,p](new)
          |[p]| lostdata[p]
   where
     .
     .
     .
endproc
```

O processo **fifo** usa uma estrutura de dados fila (queue) para representar a capacidade do canal. As operações permitidas sobre a fila são:

.Criação de uma nova fila - **new(q)**, que cria uma fila vazia **q**;

.Inclusão de um dado na fila - **add(mes,q)**, que descreve a fila resultante pela inclusão do dado **mes** na fila **q**;

.Avaliação da quantidade de dados da fila - **empty(q)**, que verifica se a fila **q** está ou não vazia;

.Retirada do primeiro elemento da fila - **first(q)**. A fila resultante depois da retirada do primeiro elemento de uma fila é representada por **rem(q)**.

O processo fifo poderá:

.Receber um novo dado enviado pelo processo sender através da porta **mes_in** e inclui-la na fila (add(mes,q));

.Caso a fila não esteja vazia (empty(q)=false), enviar o primeiro elemento (first(q)) para o processo receiver ou simular a perda do dado (interação com o processo lostdata);

.Repassar solicitação de retransmissão e concessão de novo crédito do processo receiver para o processo sender.

```
processfifo [mes_in, mes_out, cred_in, cred_out,
        retry_in,retry_out,lst] (Q:queue) :noexit :=

  mes_in?mes:int; fifo[...](add(mes,Q))
[][not(empty(Q))] → (   mes_out!first(Q);
                        fifo[...](rem(Q))
                    [] lst!first(Q);
                        fifo[...](rem(Q))
                    )
[]retry_in?expect:int; retry_out!expect;
  fifo[...](Q)
[]cred_in?cred:int; cred_out!cred;
  fifo[...](Q)

endproc
```

O processo lostdata simula a perda de dados durante a transmissão.

```
process lostdata [lst] :noexit :=
    lst?data:int; lostdata[lst]
endproc
```

Por fim, teremos a entidade par remota representada pelo processo receiver. Este processo será responsável pela concessão de crédito ao processo sender (grant), o que manterá o procedimento de transmissão de dados. Além disso, o processo receiver poderá receber um novo dado, caso em que verificará se não foi quebrada a sequência devido à perda de dados. Quando se tratar de um dado válido (eq(ms,expect)) este será repassado ao usuário $U_b$ (station), caso contrário (gt(ms,expect)) será solicitada a retransmissão do dado perdido (expect).

```
process receiver[mes-in,grant,retry,deliv] (expect:int)
    :noexit :=

      mes-in?mes:int;
        (   [eq(mes,expect)] -> deliv!mes;
                                receiver[...](succ(expect))
         [] [gt(mes,expect)] -> retry!expect;
                                receiver[...](expect)
        )
    [] grant!succ(succ(succ(succ(succ(zero)))));
          receiver[...](expect)

endproc
```

A estrutura geral detalhada do sistema pode ser vista na fig. 6.1.2.

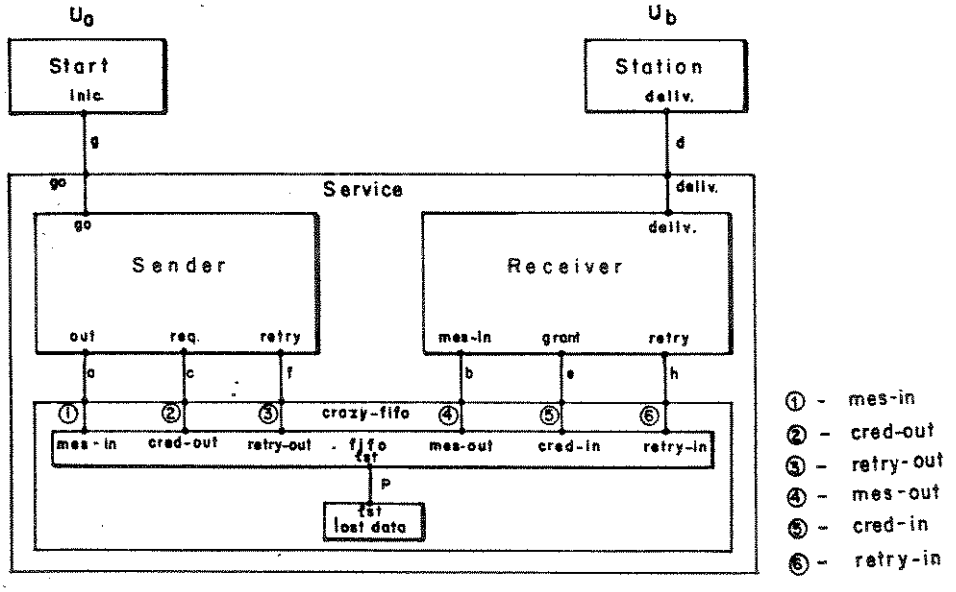



Figura 6.1.2 - Estrutura detalhada do Serviço Provedor.

A especificação LOTOS completa para este exemplo encontra-se descrita no apêndice C.

. **Estrutura da Aplicação LSTER**

Por questões de simplificação será apresentada a descrição de apenas dois módulos LSTER e do arquivo de configuração na forma reduzida. A descrição completa de todos os módulos da aplicação encontra-se no apêndice B.

**- Módulo Especificação (Spec)**

O Módulo Especificação, que descreve o comportamento associado à cláusula **behavior** terá a forma:

```
MODULE Spec;
        .
        .
        .
    BEGIN_MODULE
      LOOP
          Par('p0','p1','service',['d','g']);
          Pari('p1','user','station');
          Endy
      END_LOOP;
    END_MODULE.
```

**- Módulo Sender**

O Módulo Sender apresenta um maior nível de detalhamento da estrutura de mapeamento. Podemos observar pela comparação com o processo sender que a tradução é clara e direta.

```
MODULE Sender;
      .
      .
      .
   BEGIN_MODULE
      inic1:EnbProc;
      mes:=succ(0);
      cred:=0;
         LOOP
         inic2:...;
         IF gt(cred,0) THEN L1('a',mes,'int');
         IF eq(cred,0) THEN L2('c','int');
         L2('f','int');
         Sinc;
         CASE m2.g OF
               'a': BEGIN
                       mes:=Succ(mes);
                       cred:=Pred(cred);
                       AutoExec
                    END;
               'c': BEGIN
                       new_cred:=m2.v;
                       cred:=new_cred;
                       AutoExec
                    END;
               'f': BEGIN
                       expect:=m2.v;
                       mes:=expect;
                       AutoExec
                    END
         END
         END_LOOP;
   END_MODULE.
```

**- Arquivo de Configuração**

```
CONFIGURATION Serviço;
     INSTANCE
            p1:start;
            p2:service;
            p3:sender;
            p4:crazyfifo;
            p5:receiver;
            p6:station;
            p7:fifo;
            p8:lostdata;
            b1:b;
            s1:spec;
     CREATE
            p1/S=64/H=64,
            p2/S=64/H=64,
            p3/S=64/H=512,
            p4/S=128/H=512,
            p5/S=64/H=512,
```

```
            p6/S=64/H=512,
            p7/S=64/H=512,
            p8/S=64/H=512,
            S1/S=64/H=512,
            b1/S=2500/H=2000;
      LINK
            s1.PS1 TO b1.PE1;
            p1.PS1 TO b1.PE1;
            .
            .
            p1.PS2 TO b1.PE2;
            .
            .
            p1.PS3 TO b1.PE3;
            .
            .
            b1.PS1[1] TO p1.PE1;
            .
            .
            b1.PS2[1] TO p1.PE2;
            .
            .
            b1.PS3[1] TO p1.PE3;
            .
            .
END_CONFIG.
```

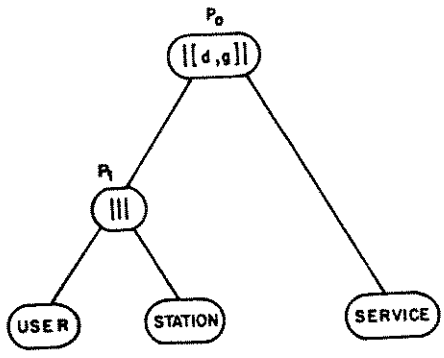
. **Execução**

Inicialmente, cada módulo participante da aplicação, exceto o Módulo Especificação, executará o procedimento de inicialização, onde passará ao módulo gerenciador o identificador do nome do processo que descreve, e em seguida entra em estado de bloqueio, no qual ficará à espera de ativação. Esta fase funciona como a inicialização da aplicação.

Deve-se notar a importância da ordem de cria-ção dos módulos - CREATE - no arquivo de configuração. O Módulo Spec aparece como penúltimo na lista de criação sendo seguido pelo módulo gerenciador. Isto implica que o Módulo Spec só entrará em execução após a inicialização de todos os módulos básicos.

A próxima fase da execução será a criação da árvore de composição inicial (ACI). Para isso, o Módulo Especificação se comunicará com o módulo gerenciador através dos construto-res PAR (...) e PARI (...).

A ACI criada pelo módulo gerenciador é representado no monitoramento da execução por:




```
Arvore de Composicao
p: p0
op:              2
ptr1: p1
ptr2: service

p: p1
op:              3
ptr1: start
ptr2: station

p: start
op:              0
ptr1: nil
ptr2: nil

p: station
op:              0
ptr1: nil
ptr2: nil

p: service
op:              0
ptr1: nil
ptr2: nil

Fim da Arvore de Composicao
```

Após a criação da ACI, o Módulo Gerencidor entrará na fase de ativação de módulos básicos, representado na execução por:

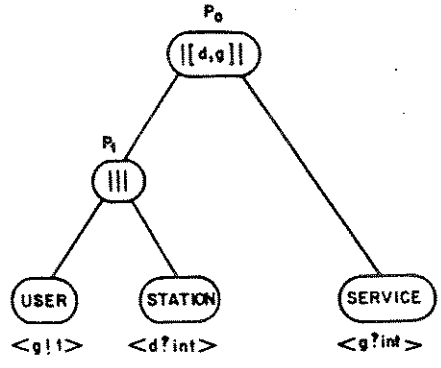
```
Fase de Ativacao de Processos
Habilita Proc.: start
Habilita Proc.: station
Habilita Proc.: service
Fim da Fase de Ativacao de Processos


Start -) ok.
Station -) ok.
Service -) ok.


Lista de Processos Habilitados
Proc.: start
Proc.: station
Proc.: service
Fim da Lista de Processos Habilitados
```

O Módulo Gerenciador reduz sua prioridade promovendo o reescalonamento da fila de pronto e a consequente execução dos módulos básicos ativados. Estes módulos apresentam suas ofertas e em seguida entram em estado de bloqueio à espera de uma mensagem de participação em evento.

Com o bloqueio dos módulos básicos, o Módulo Gerenciador retorna à execução através da atualização da árvore de composição. Como não houve nenhum pedido de instanciação de novos processos, a árvore não sofre alteração.

Em seguida o Módulo Gerenciador recebe as ofertas de eventos apresentadas pelos módulos básicos e cria a Tabela de OEs por Processo.

```
Tabela de OEs por Processos
Proc.: start
g: g
■: !
v: 1
t: int
c: TRUE
Proc.: station
g: d
■: ?
v: 0
t: int
c: TRUE
Proc.: service
g: g
■: ?
v: 0
t: int
c: TRUE
Fim da Tabela de OEs por Processos
```



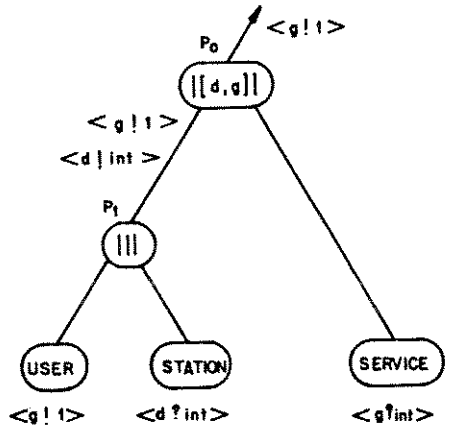

A lista de eventos possíveis é definida pela avaliação da árvore de composição conforme as OEs apresentadas. A seguir pode-se acompanhar o procedimento completo de avaliação da árvore pelas listas parciais de eventos possíveis relativas a cada nó.

Lista parcial de eventos relativa ao no: start

g: g
m: !
v: 1
t: int
c: TRUE
Fim de lista

Lista parcial de eventos relativa ao no: station

g: d
m: ?
v: 0
t: int
c: TRUE
Fim de Lista




Lista parcial de eventos relativa ao no: p1

g: g
m: !
v: 1
t: int
c: TRUE
g: d
m: ?
v: 0
t: int
c: TRUE
Fim de Lista

Lista parcial de eventos relativa ao no: service

g: g
m: ?
v: 0
t: int
c: TRUE
Fim de Lista

Lista parcial de eventos relativa ao no: p0

g: g
m: !
v: 1
t: int
c: TRUE
Fim de Lista

Em seguida um dos eventos da lista de eventos possíveis é escolhido (em interação com o usuário - execução passo-a-passo - ou aleatoriamente - geração espontânea de eventos -).

```
Lista de Eventos Possiveis
(1) g: g
    m: !
    v: 1
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis

Entre com o numero do evento escolhido
1

Porta escolhida: g
```

Os participantes da interação são obtidos, a partir da Tabela de Processos por Porta, na entrada relativa à porta onde ocorreu o evento.

```
Tabela de Processos por Portas
Porta: g
Proc.: start
Proc.: service
Porta: d
Proc.: station
Fim da Tabela de Processos por Portas
```

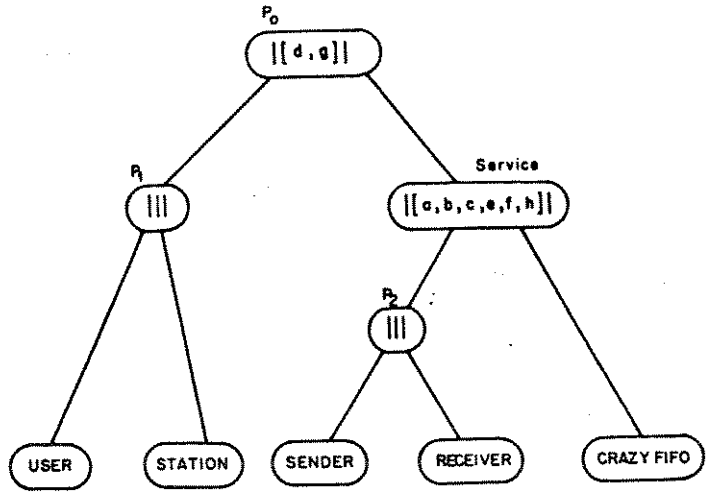
O Módulo Gerenciador novamente reduz sua prioridade permitindo a nova execução dos módulos básicos participantes da interação.

```
Start: Sinc. com valor 1
Service: Recebi start 1
```

O Módulo Start encerra sua participação na execução através do construtor STOP. O Módulo Service expande-se através da instanciação dos módulos Sender, Receiver e Crazyfifo.

Ao tornar à execução, o módulo gerenciador atualiza a árvore de composição segudo a expansão do nó terminal SERVICE.

```
Arvore de Composicao Atualizada
p: p0
op:              2
ptr1: p1
ptr2: service

p: p1
op:              3
ptr1: start
ptr2: station

p: start
op:              0
ptr1: nil
ptr2: nil

p: station
op:              0
ptr1: nil
ptr2: nil

p: service
op:              2
ptr1: p2
ptr2: crazyfifo

p: p2
op:              3
ptr1: sender
ptr2: receiver

p: sender
op:              0
ptr1: nil
ptr2: nil

p: receiver
op:              0
ptr1: nil
ptr2: nil

p: crazyfifo
op:              0
ptr1: nil
ptr2: nil

Fim da Arvore de Composicao Atualizada
```



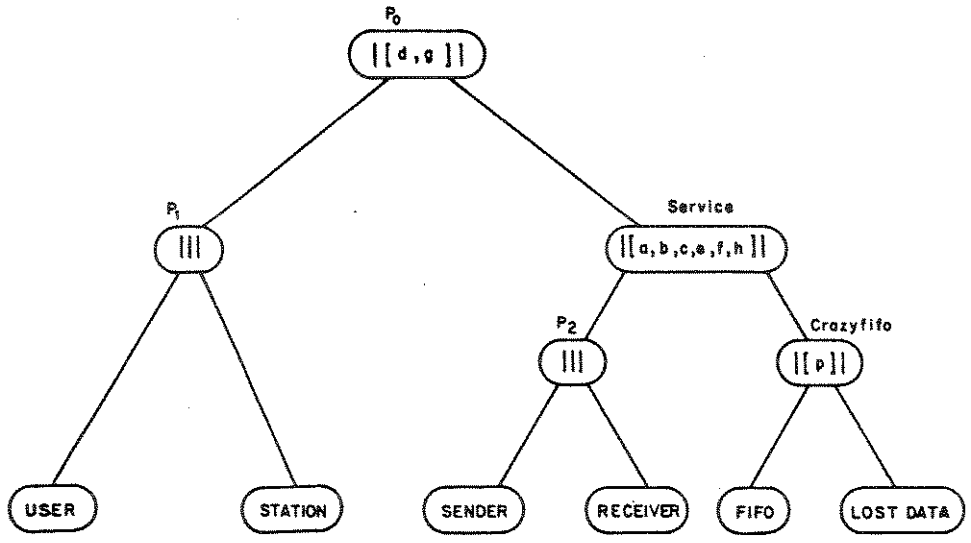

O Módulo Gerenciador ativa os novos módulos instanciados.

```
Fase de Ativacao de Processos
Habilita Proc.: sender
Habilita Proc.: receiver
Habilita Proc.: crazyfifo
Fim da Fase de Ativacao de Processos
```

```
Sender -) ok.
Receiver -) ok.
Crazyfifo -) ok

Lista de Processos Habilitados
Proc.: start
Proc.: station
Proc.: service
Proc.: sender
Proc.: receiver
Proc.: crazyfifo
Fim da Lista de Processos Habilitados
```

Os novos módulos apresentam suas ofertas. Enquanto os módulos Sender e Receiver fazem ofertas de eventos, o Módulo Crazyfifo solicita novas instanciações - Fifo e Lostdata.

Como o Módulo Gerenciador trata todas as alterações da árvore de composição antes de receber as OEs, novamente atualiza a árvore, incluindo os nós terminais Fifo e Lostdata, e ativa os novos módulos.

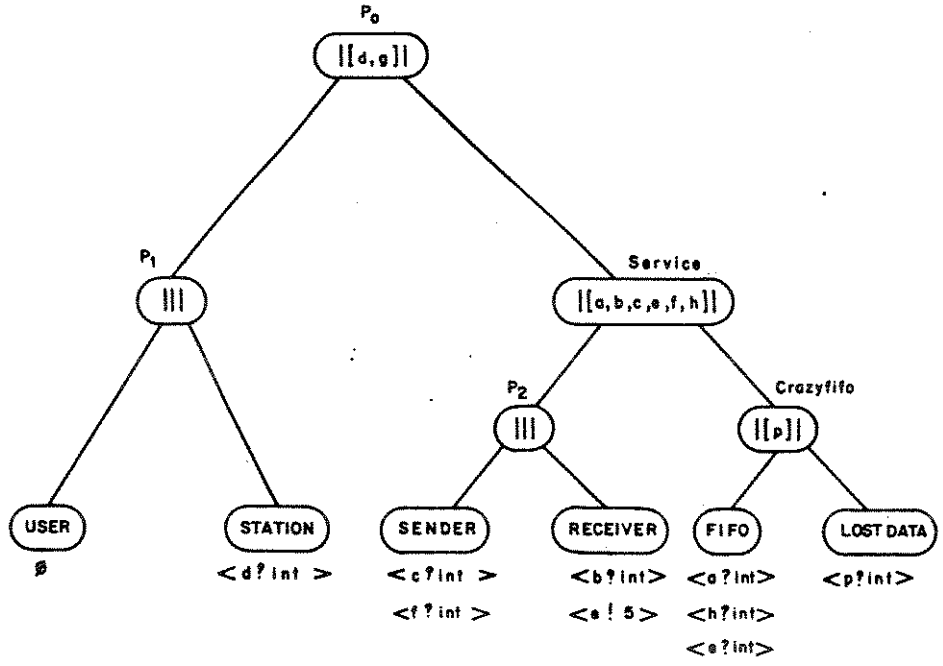


```
Arvore de Composicao Atualizada
p: p0
op:             2
ptr1: p1
ptr2: service

p: p1
op:             3
ptr1: start
ptr2: station

p: start
op:             0
ptr1: nil
ptr2: nil

p: station
op:             0
ptr1: nil
ptr2: nil

p: service
op:             2
ptr1: p2
ptr2: crazyfifo

p: p2
op:             3
ptr1: sender
ptr2: receiver

p: sender
op:             0
ptr1: nil
ptr2: nil

p: receiver
op:             0
ptr1: nil
ptr2: nil

p: crazyfifo
op:             2
ptr1: fifo
ptr2: lostdata

p: fifo
op:             0
ptr1: nil
ptr2: nil

p: lostdata
op:             0
ptr1: nil
ptr2: nil

Fim da Arvore de Composicao Atualizada
```

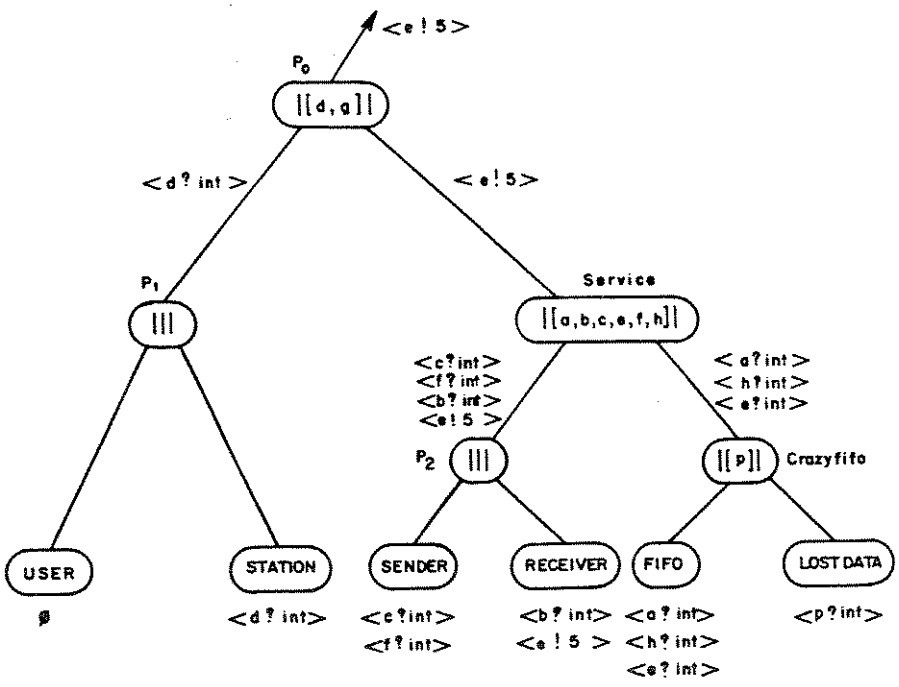
```
Fase de Ativacao de Processos
Habilita Proc.: fifo
Habilita Proc.: lostdata
Fim da Fase de Ativacao de Processos

Fifo -) ok.
Losldata -) ok.

Lista de Processos Habilitados
Proc.: start
Proc.: station
Proc.: service
Proc.: sender
Proc.: receiver
Proc.: crazyfifo
Proc.: fifo
Proc.: lostdata
Fim da Lista de Processos Habilitados
```

Após as alterações na árvore, o Módulo Gerenciador passa a receber todas as OEs apresentadas.

```
Tabela de OEs por Processos
Proc.: station
g: d
■: ?
v: 0
t: int
c: TRUE
Proc.: sender
g: c
■: ?
v: 0
t: int
c: TRUE
g: f
■: ?
v: 0
t: int
c: TRUE
Proc.: receiver
g: b
■: ?
v: 0
t: int
c: TRUE
g: e
■: !
v: 5
t: inl
c: TRUE
Proc.: fifo
g: a
■: ?
v: 0
t: int
c: TRUE
g: h
■: ?
v: 0
t: int
c: TRUE
g: e
■: ?
v: 0
t: inl
c: TRUE
Proc.: lostdata
g: p
■: ?
v: 0
t: inl
c: TRUE
Fim da Tabela de OEs por Processos
```



A nova lista de eventos possíveis é derivada a partir das OEs e da árvore de composição.




```
Lista parcial de eventos relativa ao no: start
Fim de Lista


Lista parcial de eventos relativa ao no: station
g: d
m: ?
v: 0
t: int
c: TRUE
Fim de Lista


Lista parcial de eventos relativa ao no: p1
g: d
m: ?
v: 0
t: int
c: TRUE
Fim de Lista
```

```
Lista parcial de eventos relativa ao no: sender
g: c
m: ?
v: 0
t: int
c: TRUE
g: f
m: ?
v: 0
t: int
c: TRUE
Fim de Lista


Lista parcial de eventos relativa ao no: receiver
g: b
m: ?
v: 0
t: int
c: TRUE
g: c
m: !
v: 5
t: int
c: TRUE
Fim de Lista


Lista parcial de eventos relativa ao no: p2
g: c
m: ?
v: 0
t: int
c: TRUE
g: f
m: ?
v: 0
t: int
c: TRUE
g: b
m: ?
v: 0
t: int
c: TRUE
g: e
m: !
v: 5
t: int
c: TRUE
Fim de Lista
```

```
Lista parcial de eventos relativa ao no: fifo
g: a
m: ?
v: 0
t: int
c: TRUE
g: h
m: ?
v: 0
t: int
c: TRUE
g: e
m: ?
v: 0
t: int
c: TRUE
Fim de Lista


Lista parcial de eventos relativa ao no: lostdata
g: p
m: ?
v: 0
t: int
c: TRUE
Fim de Lista



Lista parcial de eventos relativa ao no: crazyfifo
g: a
m: ?
v: 0
t: int
c: TRUE
g: h
m: ?
v: 0
t: int
c: TRUE
g: e
m: ?
v: 0
t: int
c: TRUE
Fim de Lista
```

```
Lista parcial de eventos relativa ao no: servico
g: e
m: !
v: 5
t: int
c: TRUE
Fim de Lista


Lista parcial de eventos relativa ao no: p0
g: e
m: !
v: 5
t: int
c: TRUE
Fim de Lista
```

A tabela de processos por porta indica os módulos participantes na interação.

```
Tabela de Processos por Portas
Porta: d
Proc.: station
Porta: c
Proc.: sender
Porta: f
Proc.: sender
Porta: b
Proc.: receiver
Porta: e
Proc.: receiver
Proc.: fifo
Porta: a
Proc.: fifo
Porta: h
Proc.: fifo
Porta: p
Proc.: lostdata
Fim da Tabela de Processos por Portas


Lista de Eventos Possiveis
(1) g: e
    m: !
    v: 5
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis

Entre com o numero do evento escolhido
1

Porta escolhida: c
```

Os módulos participantes da interação tornam à execução.

```
Receiver: Enviei novo credito : 5
Receiver -) ok.
Fifo: Recebi novo credito: 5
```

O Processo Sender receberá a ordem de crédito na próxima interação ocorrida na especificação. Os passos desta interação, conforme o monitoramento da execução, é apresentada, sem comentários adicionais, a seguir:

```
Tabela de OEs por Processos
Proc.: station
g: d
m: ?
v: 0
t: int
c: TRUE
Proc.: sender
g: c
m: ?
v: 0
t: int
c: TRUE
g: f
m: ?
v: 0
t: int
c: TRUE
Proc.: lostdata
g: p
m: ?
v: 0
t: int
c: TRUE
Proc.: receiver
g: b
m: ?
v: 0
t: int
c: TRUE
g: c
m: !
v: 5
t: int
c: TRUE
Proc.: fifo
g: c
m: !
v: 5
t: int
c: TRUE
Fim da Tabela de OEs por Processos
```

```
Tabela de Processos por Portas
Porta: d
Proc.: station
Porta: c
Proc.: sender
Proc.: fifo
Porta: f
Proc.: sender
Porta: p
Proc.: lostdata
Porta: b
Proc.: receiver
Porta: e
Proc.: receiver
Fim da Tabela de Processos por Portas
```

```
Lista de Eventos Possiveis
(1) g: c
    m: !
    v: 5
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis

Entre com o numero do evento escolhido
1

Porta escolhida: c


Sender: Recebi novo credito : 5
Sender -) ok.
Fifo: Enviei novo credito: 5
Fifo -) ok.
```

**. Resultados da Prototipagem**

A prototipagem da especificação proporcionou a deteção de algumas características não desejáveis do sistema. Apesar da simplicidade do serviço, pôde-se verificar que o sistema poderia evoluir para situações não permitidas.

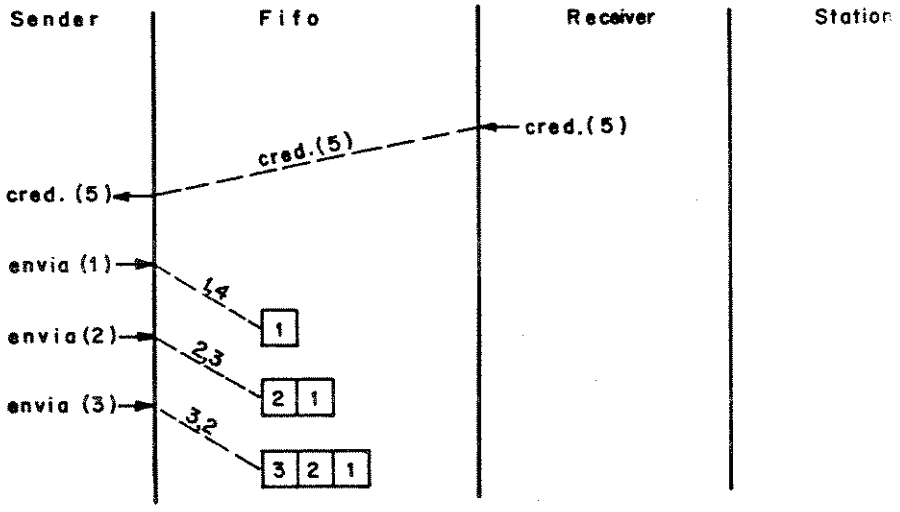
Serão descritas duas destas situações. Para simplificar a apresentação, serão mostradas apenas alguns trechos do monitoramento da execução.

O propósito aqui será unicamente de verificar a especificação e não de propor soluções para os erros porventura encontrados.

**- Situação 1**

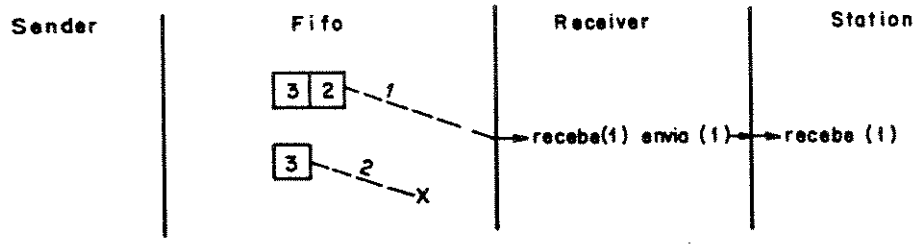
Consideremos inicialmente a sequência de eventos, descritos anteriormente, relativa à concessão de crédito. Com o novo crédito recebido a entidade sender envia os dados 1, 2 e 3 (observar a decrementação do crédito, no diagrama temporal, a cada dado enviado. p.ex.: 1,4; 2,3 ...)

```
Sender: Enviei: 1
Sender -) ok.
Fifo: Recebi dado: 1
Fifo -) ok.

Sender: Enviei: 2
Sender -) ok.
Fifo: Recebi dado: 2
Fifo -) ok.

Sender: Enviei: 3
Sender -) ok.
Fifo: Recebi dado: 3
Fifo -) ok.
```



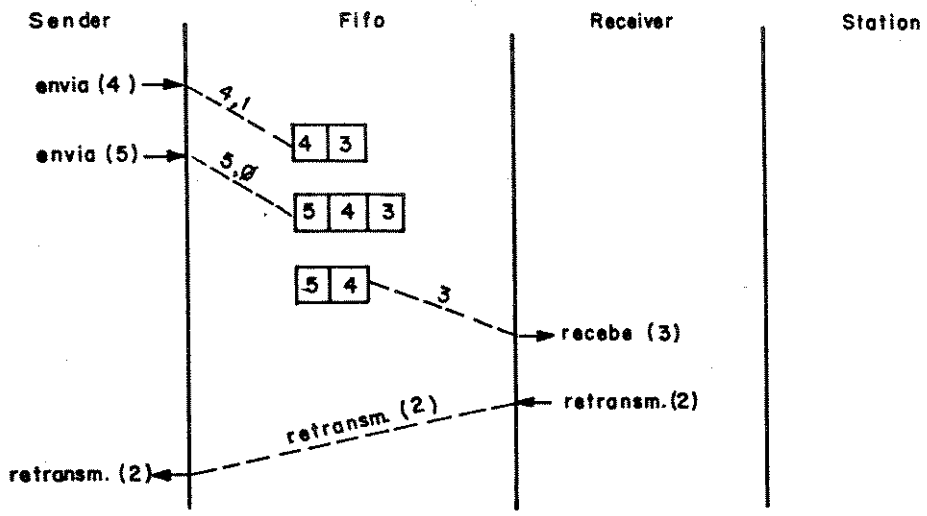

Em seguida a entidade receiver recebe o dado 1 e o entrega à estação. Devido a problemas no canal de comunicação o dado 2 é perdido.

```
Receiver: Recebi dado: 1
Fifo: Enviei dado : 1
Fifo -) ok.

Station: Recebi: 1
Station -) ok.
Receiver: Enviei dado: 1
Receiver -) ok.

Lostdata: Recebi dado perdido 2
Lostdata -) ok.
Fifo: Dado perdido : 2
Fifo -) ok.
```




Na sequência, a entidade sender envia os dados 4 e 5 e entra em um estado no qual não poderá transmitir nenhum outro dado por não possuir crédito. A execução segue com a entidade receiver recebendo o dado 3. Como o dado esperado é o 2, esta entidade solicita que ele seja retransmitido.

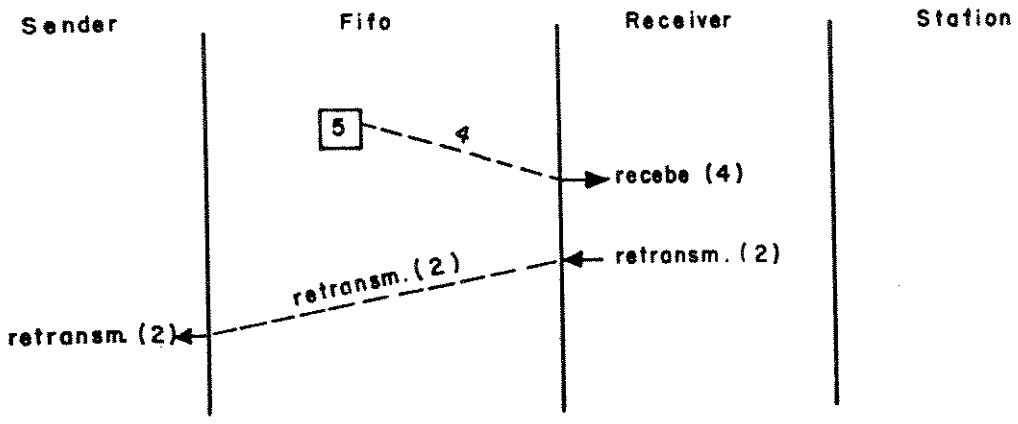
```
Sender: Enviei: 4
Sender -) ok.
Fifo: Recebi dado: 4
Fifo -) ok.

Sender: Enviei: 5
Sender -) ok.
Fifo: Recebi dado: 5
Fifo -) ok.

Receiver: Recebi dado: 3
Fifo: Enviei dado : 3
Fifo -) ok.

Receiver: Enviei solicitacao de retransmisao do dado: 2
Receiver -) ok.
Fifo: Recebi pedido para retransmitir dado: 2

Sender: Recebi pedido para retransmitir dado: 2
Sender -) ok.
Fifo: Enviei pedido para retransmitir dado: 2
Fifo -) ok.
```




Apesar da solicitação de retransmissão a entidade sender não poderá enviar o dado 2, pois não possui crédito. Este fato pode ser observado pela lista de eventos possíveis onde estão presentes: b!4 (receiver interage com fifo para receber dado 4); e!5 (receiver envia novo crédito) e i!4 (perda do dado 4). Na lista não existe o evento a!2, que representaria o envio do dado 2 pela entidade sender. Para dar sequência à execução, foi escolhido o evento b!4. Novamente por não ser o dado esperado é solicitada a retransmissão do dado 2.

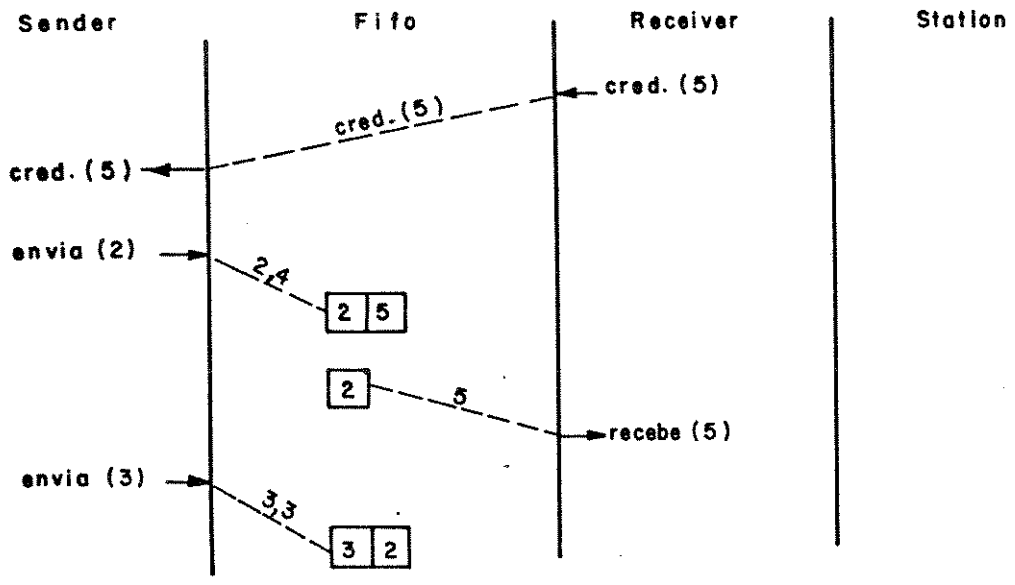
```
Lista de Eventos Possiveis
(1) g: b
    m: !
    v: 4
    t: int
    c: TRUE

(2) g: c
    m: !
    v: 5
    t: int
    c: TRUE

(3) g: p
    m: !
    v: 4
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis
```

```
Entre com o numero do evento escolhido
1

Porta escolhida: b

Receiver: Recebi dado: 4
Fifo: Enviei dado : 4
Fifo -) ok.

Receiver: Enviei solicitacao de retransmisao do dado: 2
Receiver -) ok.
Fifo: Recebi pedido para retransmitir dado: 2

Sender: Recebi pedido para retransmitir dado: 2
Sender -) ok.
Fifo: Enviei pedido para retransmitir dado: 2
Fifo -) ok.
```




Somente após a concessão de um novo crédito a entidade sender consegue enviar o dado solicitado. Em seguida consideramos a sequência de eventos descritos no diagrama temporal.

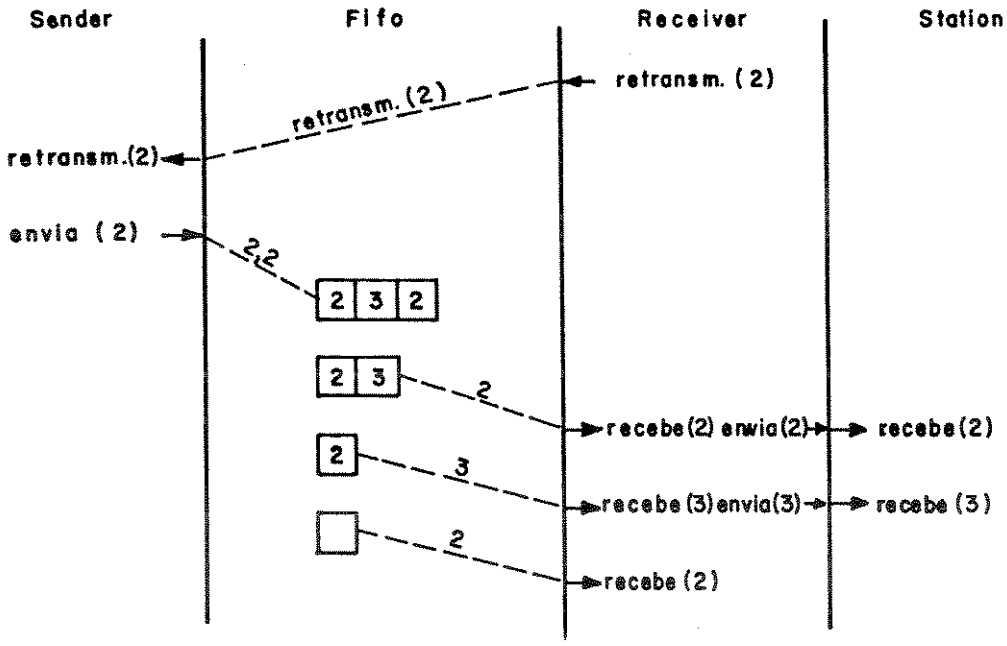
```
Receiver: Enviei novo credito : 5
Receiver -) ok.
Fifo: Recebi novo credito: 5

Sender: Recebi novo credito : 5
Sender -) ok.
Fifo: Enviei novo credito: 5
Fifo -) ok.

Sender: Enviei: 2
Sender -) ok.
Fifo: Recebi dado: 2
Fifo -) ok.

Receiver: Recebi dado: 5
Fifo: Enviei dado : 5
Fifo -) ok.

Sender: Enviei: 3
Sender -) ok.
Fifo: Recebi dado: 3
Fifo -) ok.
```




A recepção do dado 5 provoca nova solicitação de retransmissão do dado 2. Com isso a entidade sender envia novamente este dado. Esta é uma situação indesejável porque existem na fila dois dados 2. A ocorrência do envio do último dado 2 se deve à incorreções na especificação. Na sequência dos eventos é recebido os dados 2 e 3. Em seguida o outro dado 2 é recebido.

```
Receiver: Enviei solicitacao de retransmissao do dado: 2
Receiver -) ok.
Fifo: Recebi pedido para retransmitir dado: 2

Sender: Recebi pedido para retransmitir dado: 2
Sender -) ok.
Fifo: Enviei pedido para retransmitir dado: 2
Fifo -) ok.

Sender: Enviei: 2
Sender -) ok.
Fifo: Recebi dado: 2
Fifo -) ok.

Receiver: Recebi dado: 2
Fifo: Enviei dado : 2
Fifo -) ok.

Station: Recebi: 2
Station -) ok.
Receiver: Enviei dado: 2
Receiver -) ok.
```

```
Fifo: Enviei dado : 3
Fifo -) ok.
Receiver: Recebi dado: 3

Station: Recebi: 3
Station -) ok.
Receiver: Enviei dado: 3
Receiver -) ok.

Fifo: Enviei dado : 2
Fifo -) ok.
Receiver: Recebi dado: 2
```




- **Situação 2** - Deteção de DEADLOCK.

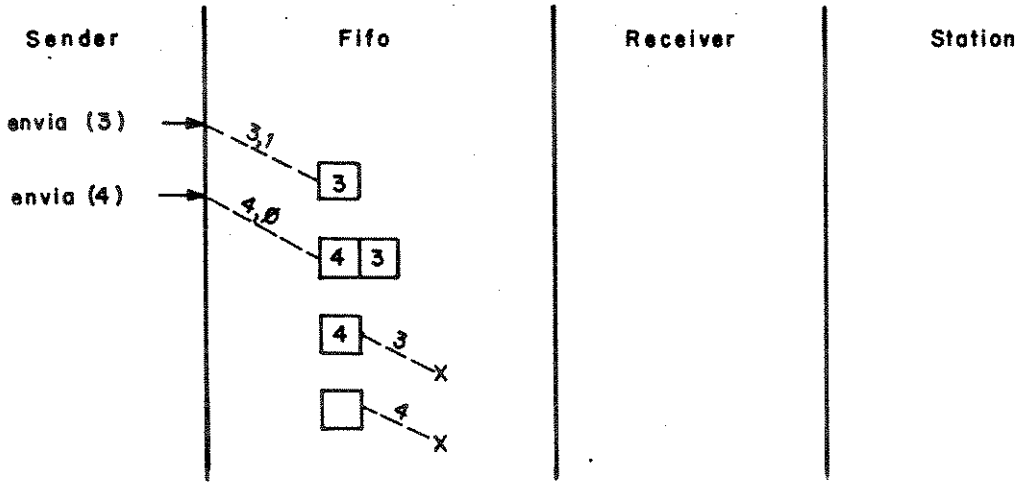
O ambiente de execução é capaz de detetar a presença de DEADLOCK na especificação.

Considerando ainda a execução descrita na situação 1, podemos observar que a recepção do dado 2 pela segunda vez põe em deadlock o processo receiver (neste

processo não é prevista a recepção de um dado com ordem de sequência inferior à esperada). O DEADLOCK do processo receiver pode ser detetado pela ausência de novas ofertas apresentadas por ele (examinar lista de eventos possíveis nas etapas de execução seguintes).

Em continuação à execução, sejam os eventos relacionados no diagrama temporal abaixo.

```
Lista de Eventos Possiveis
(1) g: a
    m: !
    v: 3
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis

Entre com o numero do evento escolhido
1

Porta escolhida: a

Sender: Enviei: 3
Sender - ) ok.
Fifo: Recebi dado: 3
Fifo - ) ok.
```



```
Lista de Eventos Possiveis
(1) g: a
    m: !
    v: 4
    t: int
    c: TRUE

(2) g: p
    m: !
    v: 3
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis

Entre com o numero do evento escolhido
1

Porta escolhida: a

Sender: Enviei: 4
Sender -) ok.
Fifo: Recebi dado: 4
Fifo -) ok.

Lostdata: Recebi dado perdido 4
Lostdata -) ok.
Fifo: Dado perdido : 4
Fifo -) ok.

Lostdata: Recebi dado perdido 3
Lostdata -) ok.
Fifo: Dado perdido : 3
Fifo -) ok.
```

Neste ponto o sistema entra em DEADLOCK. O processo sender não evolui por não possuir crédito. O processo receiver já estava em DEADLOCK. O processo fifo bloqueia-se por não existir nenhuma ação a executar (fila vazia e nenhuma solicitação de envio ou entrega de dado).

O próximo ciclo de execução deriva uma lista nula de eventos possíveis. Isto implica que todo sistema está bloqueado e portanto na existência de um DEADLOCK.

```
Lista de Eventos Possiveis
Fim da Lista de Eventos Possiveis


ATENCAO!!! Existe Deadlock na Especificacao
```

# Capítulo 7

# Conclusão

No início deste trabalho foi discutido, de maneira introdutória, a importância da definição de uma metodologia de projeto e realização de sistemas, utilizando como linguagem para representação do modelo do sistema uma técnica de descrição formal.

Existem dois aspectos importantes em uma metodologia desta natureza. O primeiro é que com o aumento da complexidade dos sistemas torna-se indispensável o uso de técnicas formais para se descrever o sistema de forma clara, não ambígua, precisa e concisa. O segundo aspecto é o ganho de estruturação e sistematização obtida seguindo-se uma metodologia coerente e bem definida.

A principal característica da metodologia apresentada é a divisão em duas fases distintas do processo de projeto. Na primeira fase serão definidos os aspectos arquitetuais do sistema. Neste sentido vimos a importância do uso de LOTOS devido ao seu alto poder de abstração, das suas propriedades de estruturação (cnf. o uso de estilos de especificação), e do seu formalismo (favorecendo a contrução de ferramentas de apoio ao projeto, p.ex.: editores, analisadores, simuladores, etc.), que permite a definição de uma arquitetura bastante expressiva do sistema. Na segunda fase serão definidos os aspectos de realização do sistema, onde questões mais voltadas à implementação serão abordados.

Segundo esta metodologia, no fim da fase arquitetural deverá ocorrer a tradução da especificação abstrata final em uma especificação de implementação inicial. O grande problema desta tradução são as diferenças semânticas dos elementos das duas linguagens de descrição (no nosso caso: LOTOS na fase arquitetural e STER na fase de realização).

Apesar da importância desta metodologia e da necessidade de um estudo mais detalhado para a sua definição, este trabalho não foi encaminhado por esta linha. Primeiro porque este é um assunto que foge ao escopo de uma tese de mestrado. Em segundo lugar, e certamente o fator determinante, é que muitas destas questões só se tornaram mais claras nas fases finais deste trabalho.

Portanto, a ênfase principal do trabalho foi a definição dos mecanismos de tradução de uma especificação LOTOS em uma aplicação STER, sendo que a metodologia foi apresentada para definir o contexto em que se inseria o trabalho.

O trabalho foi dividido em duas etapas principais. Na primeira etapa foi definida a estratégia de mapeamento, que levava em conta a inclusão de um módulo extra (Módulo Gerenciador) à aplicação STER (exercendo o papel de sevidor semân tico), necessário para que se pudesse migrar entre os dois mundos. Na segunda etapa todos os mecanismos, tanto para a construção do Módulo Gerenciador como na construção dos demais módulos da aplicação, foram implementados.

Duas considerações importantes marcaram a definição da estratégia de mapeamento. Por um lado a utilização do modelo de execução de especificações LOTOS (cnf. [WU 89]) se adequou convenientemente à nossa proposta, visto que a funcionalidade necessária para Módulo Gerenciador seria completamente obtida implementando-se tal modelo. A outra consideração importante foi a estratégia adotada para a dinâmica de execução da aplicação STER, onde foi bastante explorado o recurso de mudança de prioridade do Módulo Gerenciador, tendo sido obtido um interessante mecanismo de bloqueio e desbloqueio dos módulos STER em função da oferta(s) de evento(s) e da participação em uma interação, respectivamente. Para a validação destas duas considerações foram fundamentais as conclusões tiradas com a prototipagem do exemplo descrita no capítulo 6.

Os resultados do trabalho permitiram a conclusão da viabilidade da proposta inicial. Os principais destes resultados foram:

.A definição de regras de mapeamento favorece a proposição de uma metodologia de projeto e realização como aquela discutida no capítulo 1;

.A comparação dos resultados de simulação da especificação final LOTOS com os resultados da prototipagem da aplicação inicial LSTER, permite a análise da consistência da tradução. Neste caso poderíamos comparar os resultados obtidos com a execução do protótipo (aplicação LSTER) construído no capítulo 6, com a simulação da especificação LOTOS;

.A substituição da simulação, como utilizada na fase arquitetural, pela prototipagem na fase de realização, permite que seja seguida uma mesma política de refinamento de especificações, mesmo considerando distintos os objetivos de cada fase.

Um aspecto importante com relação à prototipagem, é que a execução do protótipo poderá se dar de três formas: Através da geração automática de cenários de teste, o que permite a verificação exaustiva do protótipo; Através da validação de cenários propostos, o que permite a verificação de aspectos específicos, e da execução passo-a-passo. No caso do exemplo descrito no capítulo 6, os cená-

rios de teste apresentados poderiam ter sido gerados automaticamente ou propostos a partir de uma análise *a priori* da especificação.

Apesar das conclusões, devem ser consideradas, criticamente, algumas questões que, se trabalhadas, abrirão espaço para novos trabalhos.

Uma destas questões, agora muito oportuna, seria a consideração de forma mais consistente da metodologia apresentada no capítulo 1. Neste sentido seria indispensável o desenvolvimento de um projeto completo que permitiria uma análise tanto global da viabilidade da metodologia, quanto de aspectos específicos, relativos à implementação, difíceis de serem equacionados sem a necessária experiência. Recentemente temos trabalhado no projeto e implementação (animação) de uma célula de manufatura flexível.

Ainda na linha de validação da metodologia proposta, certamente caberá a consideração de mais duas questões. Uma delas é avaliação do desempenho de LOTOS como linguagem de descrição utilizada na fase arquitetural. Quais são suas vantagens e desvantagens? Até que ponto, e em que condições, ela poderia ser substituída por uma outra técnica formal (p.ex: ESTELLE, SDL, VDM, Z, etc.)? Seria ela, de fato, uma técnica indispensável para descrever uma expressiva arquitetura do sistema? Além destas indagações, seria importante um detalhado estudo para se definir de que modo deverão ser utilizados os estilos de especificação e demais potencialidades LOTOS (p.ex.: qual deve ser o limite entre a especificação de aspectos do sistema na forma de estruturas abstratas de dados e na forma de expressões de comportamento; até que ponto seria útil a recomposição de processos na forma de um único processo (utilizando o Teorema da Expansão) com o fim de reduzir overheads) objetivando um maior desempenho das próximas etapas da metodologia de projeto e realização do sistema.

A outra questão seria relativa ao ambiente utilizado para o desenvolvimento da fase de realização. Até que ponto o domínio da aplicação deverá influenciar na escolha deste ambiente? Quais semelhanças são importantes existir entre o modelo utilizado na definição da linguagem usada na fase de realização, com o modelo em que se basea a técnica de descrição formal (p.ex: modularidade, para o caso LOTOS * STER)?

Uma outra linha de trabalho bastante interessante, seria considerar a possibilidade de derivação de realizações distribuídas. Neste sentido seria

muito útil repensar a proposta de mapeamento discutida neste trabalho à luz do ambiente STER distribuído [GUIMARÃES 90] e da proposta de implementação distribuída de especificações LOTOS [BOCHMANN 89].

# APÊNDICES

# APÊNDICE A - REGRAS DE ATUALIZAÇÃO

. *Notação:*

.B, B', B1, B1',... → Define expressões de comportamento;

.G → Denota o universo de portas que podem ser usadas na definição das expressões de comportamento (portas úteis);

.A → Um sub-conjunto qualquer de G;

.g,g1,...,gn → São portas definidas em G;

.i → Descreve uma ação não observável (interna);

.ACT → Conjunto de eventos definidos em G ∪ {i};

.$\mu$ → Um evento qualquer em ACT;

.$\delta$ → Pseudo evento que marca a terminação de um processo;

.$G^+$ → G ∪ {$\delta$};

.$g^+$ → Um evento qualquer em $G^+$;

.$ACT^+$ → ACT ∪ {$\delta$};

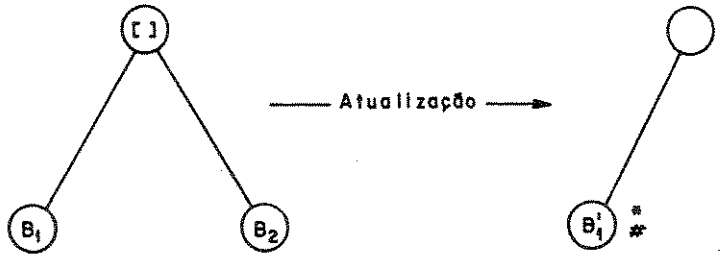
.$\mu^+$ → Um evento qualquer em $ACT^+$.

.* →Indica que após a terminação com sucesso do comportamento associado a uma sub-árvore, esta poderá ser podada da árvore.

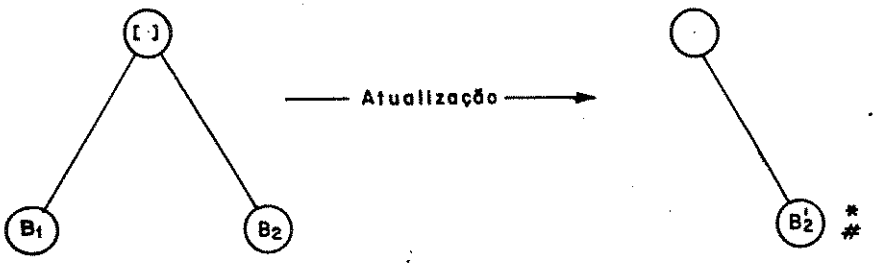
.# → Indica que um nó pode substituir seu nó ascendente.

ι) **Choice ( [ ] )**

ι) $$\frac{B1 \xrightarrow{\mu^+} B1'}{B1\ [\ ]\ B2 \xrightarrow{\mu^+} B1'}$$

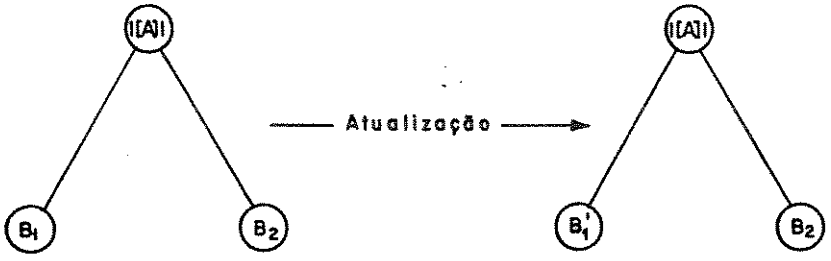


ιι) $$\frac{B_2 \xrightarrow{\mu^+} B_2'}{B_1 \; [\,] \; B_2 \xrightarrow{\mu^+} B_2'}$$

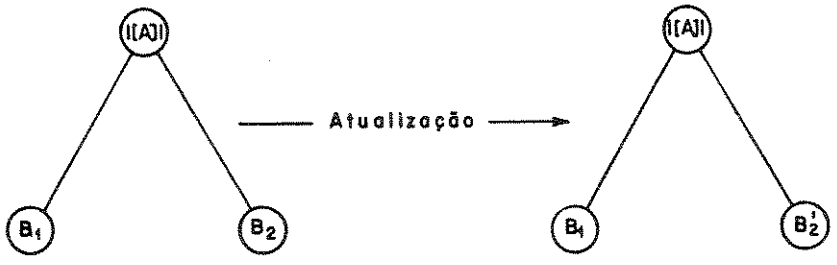



ιι) **Paralelismo** ( $|[A]|$ )

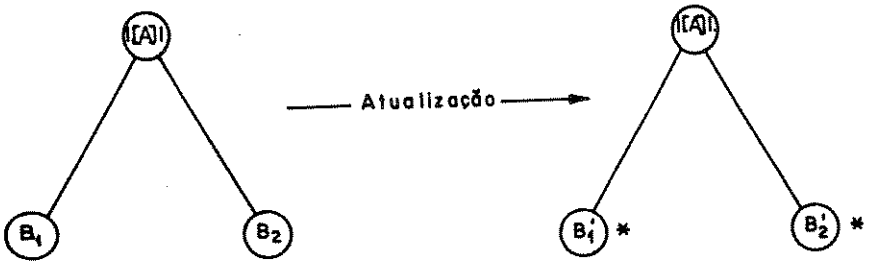
ι) $$\frac{B_1 \xrightarrow{\mu} B_1'}{B_1 |[A]| B_2 \xrightarrow{\mu} B_1' |[A]| B_2} \quad , \mu \notin A$$




ιι) $$\frac{B_2 \xrightarrow{\mu} B_2'}{B_1 |[A]| B_2 \xrightarrow{\mu} B_1 |[A]| B_2'} \quad , \mu \notin A$$

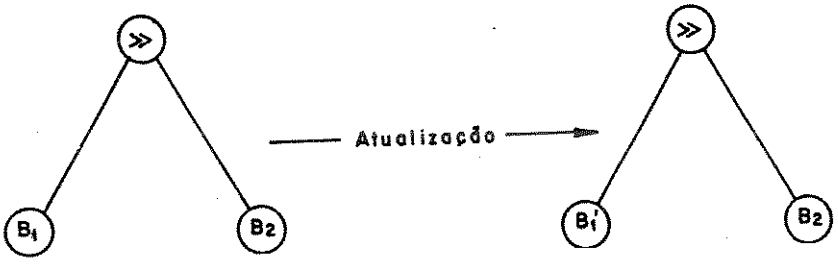


iii) $$\frac{B1 \xrightarrow{g^+} B1',\ B2 \xrightarrow{g^+} B2'}{B1 \,|[A]|\, B2 \xrightarrow{g^+} B1' \,|[A]|\, B2'},\ g^+ \in A \cup \{\delta\}$$

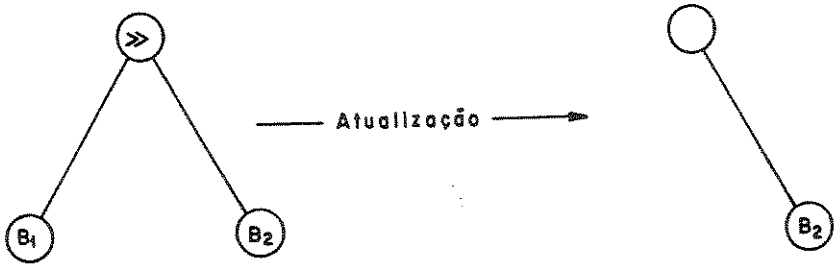



iii) **Composição Sequencial**

i) $$\frac{B1 \xrightarrow{\mu} B1'}{B1 >> B2 \xrightarrow{\mu} B1' >> B2}$$

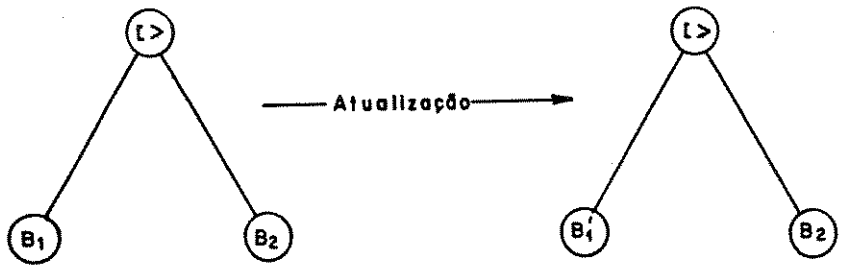



ii) $$\frac{B1 \xrightarrow{\delta} B1'}{B1 >> B2 \xrightarrow{i} B2}$$

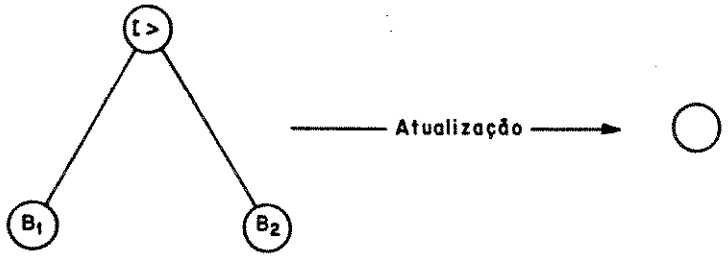


(v) **Desabilitação**

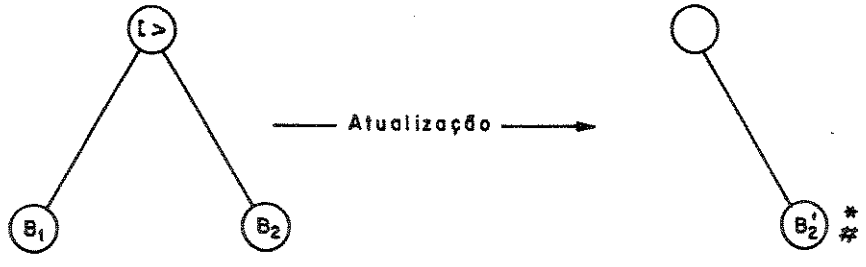
i) $$\frac{B_1 \xrightarrow{\ \mu\ } B_1'}{B_1 [> B_2 \xrightarrow{\ \mu\ } B_1' [> B_2}$$




ii) $$\frac{B_1 \xrightarrow{\ \delta\ } B_1'}{B_1 [> B_2 \xrightarrow{\ \delta\ } B_1'}$$




iii) $$\frac{B_2 \xrightarrow{\ \mu^+\ } B_2'}{B_1 [> B_2 \xrightarrow{\ \mu^+\ } B_2'}$$



# APÊNDICE B - APLICAÇÃO LSTER - SERVIÇO PROVEDOR SIMPLES

Os arquivos apresentados neste apêndice compoêm a aplicação STER relativa à especificação do Serviço Provedor descrito no exemplo do capítulo 6.

**.Arquivo de Definição**

```
DEFINE TipMes;
   TYPE
      MENS1  = RECORD
                  p: lstring(10);
                  gt: char;
                  ml: char;
                  vl: integer;
                  ti: lstring(5);
                  cn: boolean
               END;
      MENS2 = RECORD
                  g: char;
                  v: integer
               END;
      MENS3 = RECORD
                  pr: lstring(10);
                  pid: integer
               END;
      MENS4 = set of char;
      MENS5 = lstring(10);
      MENS6 = lstring(5);
      MENS7 = 0..9;
      MENS8 = boolean;
      MENS9 = RECORD
                  cd: mens7;
                  p1,p2,p3: mens5;
                  pt: mens4
               END;
      MENS10 = boolean;
END_DEFINE.
```

```
MODULE Spec;

    USE TipMes.inc;

    EXITPORT
        PS1: mens9;

    ENTRYPORT
        PE: mensi0;

    MESSAGE
        m9:mens9;
        mi0:mensi0;

    PROCEDURE par(pr1,pr2,pr3:mens5;prt:mens4);EXTERN;

    PROCEDURE pari(pr1,pr2,pr3:mens5);EXTERN;

    PROCEDURE Endy;EXTERN;

    BEGIN_MODULE
        LOOP
            Par('p0','pi','service',['d','g']);
            Pari('pi','start','station');
            Endy
        END_LOOP;
    END_MODULE.
```

```
MODULE Start;

USE TipMes.inc;

EXITPORT
    PS1: mens9;
    PS2: mens1;
    PS3: mens3;

ENTRYPORT
    PE1,PE2,PE: mens10;
    PE3: mens2;

MESSAGE
    m1: mens1;
    m2: mens2;
    m3: mens3;
    m9: mens9;
    m10: mens10;

CONST
    pr = 'start';

LABEL
    inic1,inic2;

PROCEDURE L1(g:char;v:integer;t:mens6);EXTERN;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE stop;EXTERN;

PROCEDURE EnbProc;EXTERN;

BEGIN_MODULE
    inic1:EnbProc;
    LOOP
        inic2:writeln('Start -> ok.');
        L1('g',1,'int');
        Sinc;
        writeln('Start: Sinc. com valor ',m2.v:1);
        Stop
    END_LOOP;
END_MODULE.
```

```
MODULE Service;

USE TipMes.inc;

EXITPORT
   PS1: mens9;
   PS2: mens1;
   PS3: mens3;

ENTRYPORT
   PE1,PE2,PE: mens10;
   PE3: mens2;

MESSAGE
   m1: mens1;
   m2: mens2;
   m3: mens3;
   m9: mens9;
   m10: mens10;

CONST
   pr = 'service';

LABEL
   inic1,inic2;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE par(pr1,pr2,pr3:mens5;prt:mens4);EXTERN;

PROCEDURE pari(pr1,pr2,pr3:mens5);EXTERN;

PROCEDURE EnbProc;EXTERN;

PROCEDURE Endb;

BEGIN_MODULE
   inic1:EnbProc;
   LOOP
      inic2:writeln('Service -> ok.');
      L2('g','int');
      Sinc;
      writeln('Service: Recebi start ',m2.v:1);
      Par('service','p2','crazyfifo',['a','b','c','e','f','h']);
      Pari('p2','sender','receiver');
      Endb
   END_LOOP;
END_MODULE.
```

```
MODULE Sender;

USE TipMes.inc;

EXITPORT
   PS1: mens9;
   PS2: mens1;
   PS3: mens3;

ENTRYPORT
   PE1,PE2,PE: mens10;
   PE3: mens2;

MESSAGE
   m1: mens1;
   m2: mens2;
   m3: mens3;
   m9: mens9;
   m10: mens10;

(* $INCLUDE: 'Dt.def' *)

CONST
   pr = 'sender';
   mens_inic = 1;

VAR
  mes: integer;
  cred: integer;
  new_cred: integer;
  expect: integer;

LABEL
   inic1,inic2;

PROCEDURE L1(g:char;v:integer;t:mens6);EXTERN;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE AutoExec;EXTERN;

PROCEDURE EnbProc;EXTERN;

BEGIN_MODULE
   inic1:EnbProc;
   mes:=succ(0);
   cred:=0;
   LOOP
      inic2: writeln('Sender -> ok.');
      IF gt(cred,0) THEN L1('a',mes,'int');
      IF eq(cred,0) THEN L2('c','int');
```

```
      L2('f','int');
      Sinc;
      CASE m2.g OF
          'a': BEGIN
                  writeln('Sender: Enviei: ',m2.v:1);
                  mes:=Succ(mes);
                  cred:=Pred(cred);
                  AutoExec
               END;
          'c': BEGIN
                  writeln('Sender: Recebi novo credito : ',m2.v:1);
                  new_cred:=m2.v;
                  cred:=new_cred;
                  AutoExec
               END;
          'f': BEGIN
                  writeln('Sender: Recebi pedido para
                           retransmitir dado: ',m2.v:1);
                  expect:=m2.v;
                  mes:=expect;
                  AutoExec
               END
      END
   END_LOOP;
END_MODULE.
```

```
MODULE Crazyfifo;

USE TipMes.inc;

EXITPORT
   PS1: mens9;
   PS2: mens1;
   PS3: mens3;

ENTRYPORT
   PE1,PE2,PE: mens10;
   PE3: mens2;

MESSAGE
   m1: mens1;
   m2: mens2;
   m3: mens3;
   m9: mens9;
   m10: mens10;

CONST
   pr = 'crazyfifo';

VAR
  mes: integer;
  cred: integer;
  expect: integer;

LABEL
   inic1,inic2;

PROCEDURE par(pr1,pr2,pr3:mens5;prt:mens4);EXTERN;

PROCEDURE EnbProc;EXTERN;

PROCEDURE Endb;EXTERN;

BEGIN_MODULE
   inic1:EnbProc;
   LOOP
   inic2:writeln('Crazyfifo -> ok.');
   Par('crazyfifo','fifo','lostdata',['p']);
   Endb
   END_LOOP;
END_MODULE.
```

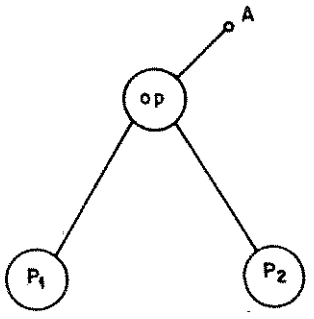
```
MODULE Fifo;

USE TipMes.inc;

EXITPORT
   PS1: mens9;
   PS2: mens1;
   PS3: mens3;

ENTRYPORT
   PE1,PE2,PE: mens10;
   PE3: mens2;

MESSAGE
   m1: mens1;
   m2: mens2;
   m3: mens3;
   m9: mens9;
   m10: mens10;

(* $INCLUDE: 'Dt.def' *)

CONST
   pr = 'fifo';

VAR
  mes: integer;
  cred: integer;
  expect: integer;

LABEL
   inic1,inic2;

PROCEDURE L1(g:char;v:integer;t:mens6);EXTERN;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE AutoExec;EXTERN;

PROCEDURE EnbProc;EXTERN;

BEGIN_MODULE
   inic1:EnbProc;
   Niew(Q);
   LOOP
      inic2:writeln('Fifo -> ok.');
      L2('a','int');
```

```
      IF not(empty(Q))
      THEN
         BEGIN
            L1('b',first(Q),'int');
            L1('p',first(Q),'int')
         END;
      L2('h','int');
      L2('e','int');
      Sinc;
      CASE m2.g OF
          'a': BEGIN
                  writeln('Fifo: Recebi dado: ',m2.v:1);
                  mes:=m2.v;
                  add(mes,Q);
                  AutoExec
               END;
           'b': BEGIN
                  writeln('Fifo: Enviei dado : ',m2.v:1);
                  rem(Q);
                  AutoExec
               END;
           'p': BEGIN
                  writeln('Fifo: Dado perdido : ',m2.v:1);
                  rem(Q);
                  AutoExec
               END;
           'h': BEGIN
                  writeln('Fifo: Recebi pedido para
                           retransmitir dado: ',m2.v:1);
                  expect:=m2.v;
                  L1('f',expect,'int');
                  Sinc;
                  writeln('Fifo: Enviei pedido para
                           retransmitir dado: ',m2.v:1);
                  AutoExec
               END;
           'e': BEGIN
                  writeln('Fifo: Recebi novo credito: ',m2.v:1);
                  cred:=m2.v;
                  L1('c',cred,'int');
                  Sinc;
                  writeln('Fifo: Enviei novo credito: ',m2.v:1);
                  AutoExec
               END
      END
   END_LOOP;
END_MODULE.
```

```
MODULE Lostdata;

USE TipMes.inc;

EXITPORT
    PS1: mens9;
    PS2: mens1;
    PS3: mens3;

ENTRYPORT
    PE1,PE2,PE: mens10;
    PE3: mens2;

MESSAGE
    m1: mens1;
    m2: mens2;
    m3: mens3;
    m9: mens9;
    m10: mens10;

CONST
    pr = 'lostdata';

LABEL
    inic1,inic2;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE AutoExec;EXTERN;

PROCEDURE EnbProc;EXTERN;

BEGIN_MODULE
    inic1:EnbProc;
    LOOP
        inic2:writeln('Lostdata -> ok.');
        L2('p','int');
        Sinc;
        writeln('Lostdata: Recebi dado perdido ',m2.v:1);
        AutoExec;
    END_LOOP;
END_MODULE.
```

→ **Exemplo**

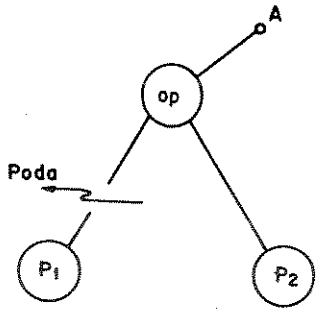
.Seja a árvore



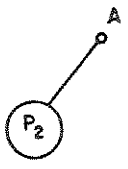

onde **op** é um dos operadores >>, [> ou []

A terminação de $\mathbf{P}_1$ com sucesso - exit - (no caso de **op** = >>) ou a participação de $\mathbf{P}_2$ em um evento (no caso de **op** = [] ou **op** = [>), produzirá as seguintes modificações na árvore:

i) Poda de $\mathbf{P}_1$ por razões de desabilitação ou terminação;




ii) Substituição do nó interno (relativo a **op**) por $\mathbf{P}_2$ (ver apêndice A);



## 5.3.2 - Ativação de Módulos Básicos

Após a criação ou atualização da Árvore de Composição, o Módulo Gerenciador entrará na fase de ativação.

Todos os Módulos Básicos ainda não ativados serão habilitados, menos aqueles ligados às sub-árvores direitas nascidas de nós que descrevem operadores ENABLE (>>) - p.ex.: $MB_2$ na fig. 5.3.3.

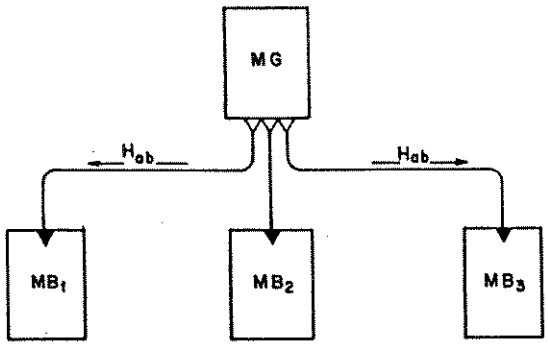



Figura 5.3.3 - Ativação de Módulos Básicos

**. Procedimento**

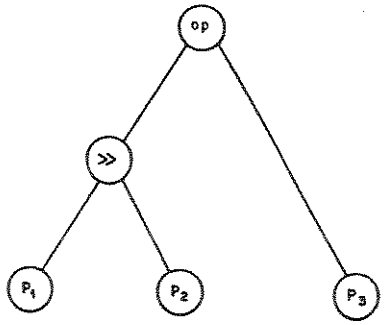
I) Analisar a árvore na forma de um procedimento Preorder [WIRTH 76];

.Percorrer sub-árvores direitas somente se os nós aos quais estão ligadas não estiverem associadas a operadores ENABLE (>>).

II) Enviar mensagem de habilitação aos Módulos Básicos ainda não ativado.

**→ Exemplo**

Seja a árvore



onde: **op** é um dos operadores |[A]|, | |, | | |, [> ou []. Após a execução do procedimento de habilitação, os módulos $P_1$ e $P_3$ serão ativados (cnf fig. 5.3.3).

Após a execução do procedimento de habilitação, o Módulo Gerenciador reduzirá sua prioridade. Com isto os módulos que foram ativados entrarão em execução. Este procedimento não implicará em nenhuma alteração nos módulos anteriormente ativados e que estejam bloqueados devido à ofertas de eventos pendentes.

## 5.3.3 - TABELAS AUXILIARES

Além da Árvore de Composição foram definidas duas tabelas para auxiliar o Módulo Gerenciador nas diversas análises realizadas.

**.Tabelas de Ofertas de Eventos (OE) por processo**

Após a recepção das OEs, o Módulo Gerenciador cria uma tabela onde constará, para cada processo, uma lista de suas OEs.

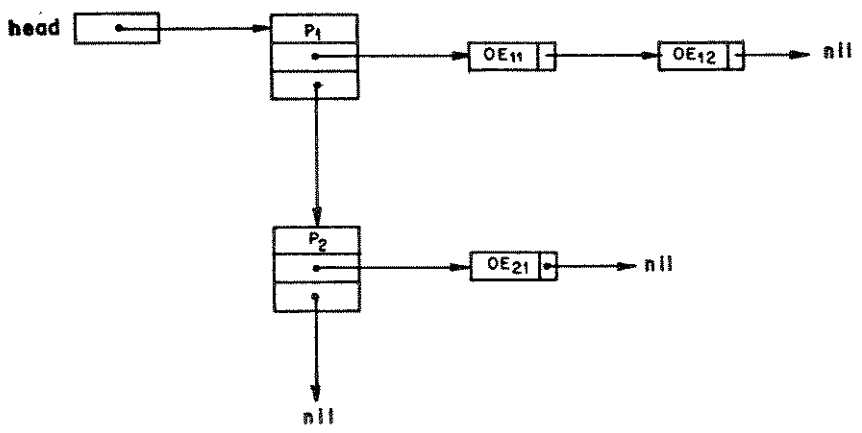



Figura 5.3.4 - Tabela de OEs por processos

O conteúdo desta tabela será visto pelo Módulo Gerenciador como o

conjunto de processos possíveis de participarem em um evento a cada ciclo de execução.

Quando o Módulo Gerenciador estiver na fase de avaliação das ofertas, para cada Módulo Básico encontrado na árvore será feita uma pesquisa nesta tabela para obter a sua lista de OEs (seção 4.5.2).

Na fase de tratamento da ocorrência de evento quando for determinada a participação de um processo em uma intera-ção, a sua entrada nesta tabela será retirada.

**.Tabelas de processos por portas**

Esta tabela será criada para relacionar todos os processos que fazem oferta em uma determinada porta. Foi considerado um formato semelhante ao da fig. 5.3.4.

Segundo o nosso modelo de execução, ao final da fase de tratamento da ocorrência de evento, será conhecida apenas a porta e as condições em que ocorreu o evento (valores negociados). Para que sejam identificados os processos que participaram da interação, a tabela de processos por portas será pesquisada.

A porta onde ocorreu uma interação deverá ter sua entrada, nesta tabela, deletada. O restante da tabela permanecerá inalterada para o próximo ciclo de execução.

### 5.3.4 - ANÁLISE DAS OFERTAS DE EVENTOS COM BASE NA ÁRVORE DE COMPOSIÇÃO

Durante esta fase, as ofertas serão avaliadas quanto à semântica dos operadores LOTOS associados aos nós da Árvore de Composição. Ao final desta fase será obtida uma lista com todos os eventos cuja ocorrência é possível. Somente na fase seguinte é que será definida a ocorrência de um dos eventos desta lista. Assim, todos eventos ofertados serão considerados sem que seja feito nenhum tipo de escolha (esta observação visa esclarecer como será considerada a característica de exclusão nos operadores |||, [> e []).

**Procedimento**

I) Realizar busca do tipo postorder a partir da raiz até ao nível das folhas;

II) Para cada folha encontrada, recuperar da tabela de ofertas de eventos por processo a lista de OEs relativa ao processo associado à folha;

II) Para cada nó interno, derivar uma lista resultante a partir das listas apresentadas pelas suas sub-árvores.

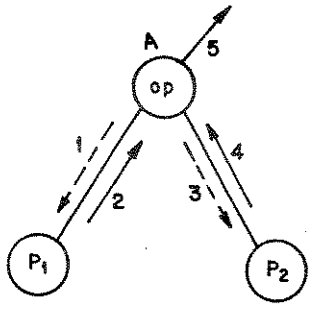
A vantagem da estrutura da árvore para a análise realizada por este procedimento é que:

*i*) Qualquer nó considerará as sub-árvores nascidas de si como se fossem Módulos Básicos;

*ii*) O resultado de cada chamada recursiva será a oferta de uma lista de eventos, podendo ser uma lista nula no caso de não haver nenhuma oferta associada ao processo ou sub-árvore pesquisada.

→ **Exemplo 1**

Seja a árvore:




onde:

.A é um nó interno ao qual está associado um dos operadores LOTOS de composição (op);

.$P_1$ e $P_2$ são Módulos Básicos;

.A seta pontilhada (⟶) representa uma busca recursiva;

.A seta cheia (⟶) representa a resposta a uma busca recursiva;

A análise sobre o nó **A** será feita na seguinte forma:

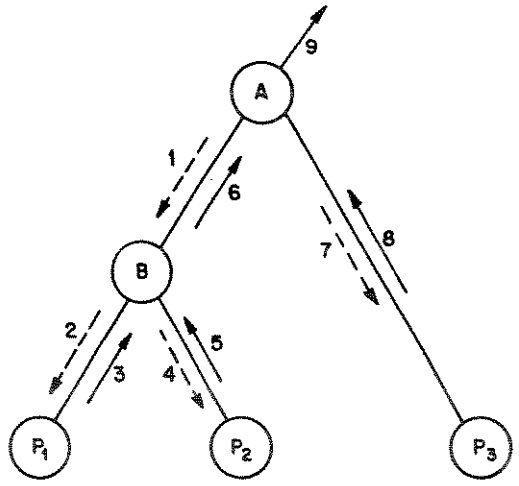
ι) Identificar a lista de eventos ofertados por $\mathbf{P}_1$ (avaliação da sub-árvore esquerda descrita para sequência de setas 1 e 2);

ιι) Identificar a lista de oferta de eventos de $\mathbf{P}_2$ (avaliação da sub-árvore direita, setas 3 e 4).

ιιι) Elaborar uma lista de $OE_s$ resultante e apresentá-la como resposta para a análise recursiva do nó ao qual está associada esta sub-árvore.

**Exemplo 2**

Seja agora a árvore:




A árvore será pesquisada na seguinte forma:

.Como o nó A não é terminal, então analisar sub-árvores:

.(sub-árvore esquerda)_A ≡ nó B. Como o nó B não é terminal, analisar sub-árvores:

.(sub-árvore esquerda)_B ≡ folha P1. Como é um Módulo Básico, então:

.Identificar a lista de OEs de P1($LOE_{P1}$) e atribuí-la à lista de OEs da sub-árvore esquerda ($LOE_{sae}$);

$$LOE_{sae} = LOE_{P1}$$

.Apresentar resultado ($LOE_{sae}$) ao nó B;

.(sub-árvore direita)_B ≡ folha P2. Como P2 é básico:

.Associar $LOE_{P2}$ à lista de OEs da sub-árvore esquerda ($LOE_{sad}$);

$$LOE_{sad} = LOE_{P2}$$

.Apresentar $LOE_{sad}$ ao nó B;

.Determinar lista de OEs resposta do nó B e apresentar ao nó A. Como o nó B é a raiz da sub-árvore esquerda de A, a lista de OEs resultante será apresentada como $LOE_{sae}$

$$LOE_{sae} = \text{Derivação } (LOE_{sae}, LOE_{sad}, op_B)$$

.(sub-árvore direita)_A ≡ folha P3 Como P3 é básico:

.$LOE_{sad} = LOE_{P3}$
.Apresentar resultado ($LOE_{sad}$) ao nó A

.Determinar

$$LOE_{sae} = \text{Derivação } (LOE_{sae}, LOE_{sad}, op_A)$$

.Apresentar lista resultante ($LOE_{sae}$ - ↗9) à fase seguinte - tratamento da ocorrência de evento.

O modelo de derivação - seção 4.5.2 - foi implementado segundo os procedimentos descritos a seguir.

**.Busca e derivação de listas de OEs**

O objetivo deste procedimento é derivar uma lista terminal onde serão apresentados todos os eventos possíveis de ocorrer baseado nas condições impostas pela semântica dos operadores LOTOS distribuídos na árvore.

**Procedimento**

I) Por meio de uma busca postorder, varrer a árvore para definição das listas de $OE_s$ associadas a cada nó.

Sempre a partir de duas listas obtidas junto às sub-árvores, será gerada uma terceira lista de acordo com a semântica do operador LOTOS descrito no nó.

II) No caso de nós terminais, derivar lista ($LOE_{sad}$ ou $LOE_{sae}$) diretamente das listas de ofertas dos processos (tabela de ofertas de eventos por processo);

III) No caso de nós internos, um procedimento, específico para cada operador, será chamado para tratar a derivação.

**.Derivação de listas em nós terminais**

Durante o procedimento de busca das listas na Árvore de Composição, ao se atingir um nó terminal será criada uma nova lista ($LOE_{sad}$ ou $LOE_{sae}$ - dependendo da posição do nó na árvore, como mostra a fig. 5.3.5 -) que será simplesmente uma cópia da lista de $OE_s$ - a partir da tabela de ofertas de eventos por processos - do processo identificado no nó, ou

$$LOE_{sad/e} = LOE_{Proc}$$

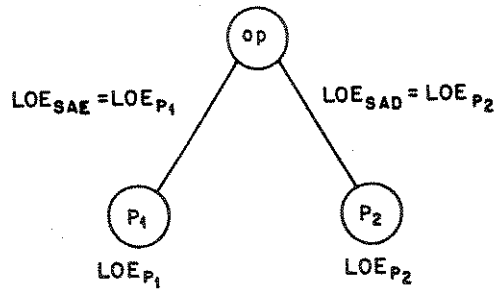



Figura 5.3.5 - Derivação de lista em nó terminal.

**.Derivação de listas em nós não terminais**

A derivação de listas em nós não terminais será regida pela semântica do operador associado ao nó e serão usadas nos procedimentos de derivação.

**..Paralelismo com sincronização restrita (|[A]|)**

*i*) Para cada elemento de $LOE_{sae}$ cuja porta se encontra em [A], verificar se existe em $LOE_{sad}$ alguma OE na mesma porta. Neste caso, para cada ocorrência, derivar oferta na forma abaixo e incluir na lista resultante - $LOE_{sa}$ - (fig. 5.3.6).

SE matched ($OE_1$,$OE_2$)
ENTÃO
    OE = Derivação ($OE_1$,$OE_2$)

Onde:

$OE_1 \in LOE_{sae}$ ;
$OE_2 \in LOE_{sad}$

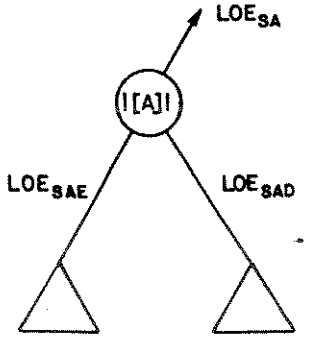



Figura 5.3.6 - Derivação de Lista em nó com operador de paralelismo restrito |[A]|

*ii*) As $OE_s$ de $LOE_{sad}$ e $LOE_{sae}$ não satisfeitas em (i), devem ser incluídas em $LOE_{sa}$.

Na fig. 5.3.7 temos um fluxograma detalhado deste procedimento.

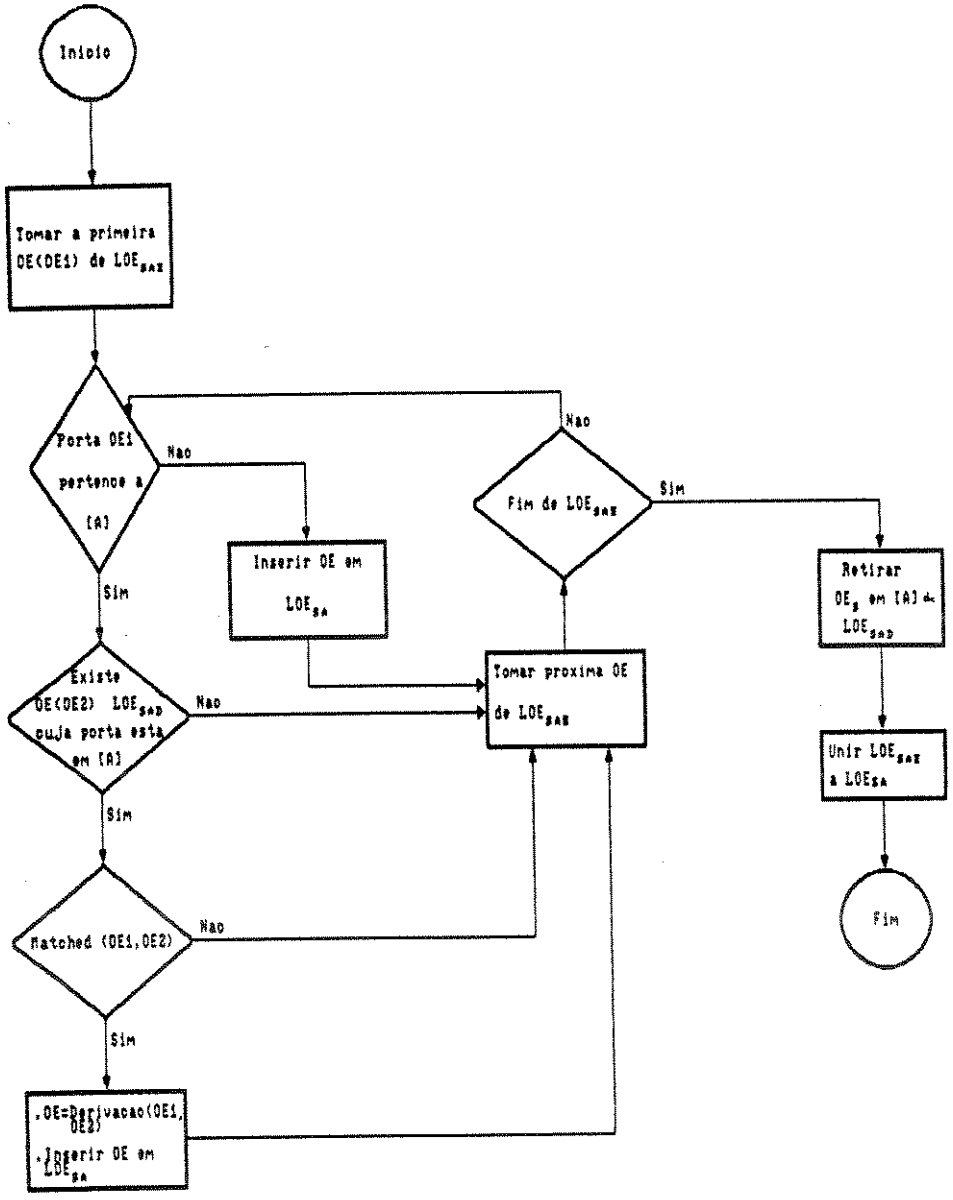



Figura 5.3.7 - Fluxograma do procedimento de derivação de lista em nó com paralelismo restrito - $|[A]|$

**..Paralelismo com sincronização completa**

No caso de nós com operador de paralelismo com sincronização em todas as portas, o procedimento para derivação de lista será um caso particular do procedimento anterior, ou seja:

I) Para cada OE de $LOE_{sae}$, verificar ofertas na mesma porta presentes em $LOE_{sad}$. Neste caso derivar oferta na forma discutida no caso anterior;

II) As OEs não tratadas por (i) devem ser desconsideradas.

O fluxograma mostrado na Fig. 5.3.7 pode ser usado neste caso, desde que feitas as seguintes considerações:

i) Como a sincronização deve ser completa, não haverá testes sobre conjunto de portas;

ii) Só haverá insersão de OE em $LOE_{sa}$ no caso de ofertas derivadas.

**..Paralelismo sem sincronização(|||), Disrupt ([>) e Choice ([])**

Estes três operadores serão tratados de modo idêntico devido à semelhança de suas regras de avaliação (cnf. regra (ii) para nós não-terminais da seção 4.5.2), sendo ainda um caso particular do procedimento de tratamento do operador de sincronização restrita.

Neste caso, as duas listas - $LOE_{sad}$ e $LOE_{sae}$ - serão unidas e atribuídas a $LOE_{sa}$, como

$$LOE_{sa} = LOE_{sae} \cup LOE_{sad}$$

**.Enable (>>)**

Este operador será tratado pela atribuição direta da lista de OEs da sub-árvore direita à lista derivada, na forma

$$LOE_{sa} = LOE_{sad}$$

isto porque só poderão existir processos ativos associados à sub-árvore esquerda.

→ **Exemplo 3**

Seja a especificação LOTOS

```
specipication Exemplo [g1,g2,g3,g4,g5,g6] : noexit
 type ... endtype
 behavior
     P1[g1,g2,g3,g4,g5]
     |[g1,g3,g4,g5]|
     P2[g1,g3,g5,g6]
  where
   process P1[a,b,c,d,f] : noexit :=
     P3[d,f]
     |||
     P4[a,b,c]
    where
     process P3[a,b] : noexit :=
         let y:int = 1 in
           ( a?x:int;P3[a,b]
             []
             b!y;P3[a,b]
           )
     endproc
     process P4[a,b,c] : noexit :=
         P5[a,b,c](1,2,3)
         ||
         P6[c](3)
      where
       process P5[a,b,c](x,y,z:int) : noexit :=
             a!x;P5[a,b,c](x,y,z)
          [] b!y;P5[a,b,c](x,y,z)
          [] c!z;P5[a,b,c](x,y,z)
       endproc
```

```
        process P6[a](x:int) : noexit :=
          a!x;P6[a](x)
        endproc
      endproc
    endproc
    process P2[a,b,c,d] : noexit :=
      P7[a,b]
      [>
      P8[c,d](2)
     where
      process P7[a,b] : noexit :=
          let x:int = 3 in
            ( a!x:int;P7[a,b]
              []
              b?y:int;P7[a,b]
            )
      endproc
      process P8[a,b](x:int) : noexit :=
            a?y:int;P8[a,b](x)
          [] b!x;P8[a,b](succ(x))
      endproc
    endproc
endspec
```

A árvore de composição que descreve a estrutura da especificação é representada na figura 5.3.8.

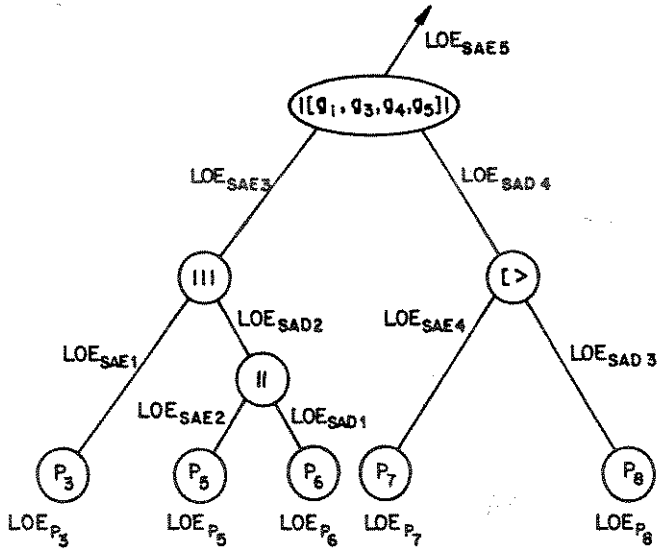



Figura 5.3.8 -Árvore de Composição referente à especificação.

De acordo com a regra para derivação de lista de OEs em nós terminais, temos;

$$LOE_{P3} = \{<g_4?int>,<g_5!1>\}$$

$$LOE_{P5} = \{<g_1!1>,<g_2!2>,<g_3!3>\}$$

$$LOE_{P6} = \{<g_3!3>\}$$

$$LOE_{P7} = \{<g_1!3>,<g_3?int>\}$$

$$LOE_{P8} = \{<g_5?int>,<g_6!2>\}$$

Conforme a descrição do exemplo 2 , a análise das OEs segundo a estrutura da Árvore de Composição será:

1. $LOE_{sae1} = LOE_{P3}$

2. $LOE_{sae2} = LOE_{P5}$

3. $LOE_{sad1} = LOE_{P6}$

4. $LOE_{sad2} = Derivação\ (LOE_{sae2}, LOE_{sad1}, ||)$

   $= Derivação\ (\{<g_1!1>,<g_2!2>, <g_3!3>\},\{<g_3!3>\},||)$

   $= \{<g_3!3>\}$

5. $LOE_{sae3} = Derivação\ (LOE_{sae1}, LOE_{sad2}, |||)$

   $= Derivação\ (\{<g_4?int>,<g_5!1>\}, \{<g_3!3>\},|||)$

   $= \{<g_4?int>, <g_5!1>,<g_3!3>\}$

6. $LOE_{sae4} = LOE_{P7}$

7. $LOE_{sad3} = LOE_{P8}$

8. $LOE_{sad4} = Derivação\ (LOE_{sae4}, LOE_{sad3}, [>)$

   $= Derivação\ (\{<g_1!3>,<g_3?int>\}, \{<g_5?int>,<g_6!2>\},[>)$

   $= <g_1!3>,<g_3?int>\},\{<g_5?int>, <g_6!2>$

9. $LOE_{sae5}$ = Derivação ($LOE_{sae3}$, $LOE_{sad4}$, $|[A]|$)

= Derivação ({<$g_4$?int>,<$g_5$!1>, <$g_3$!3>},{<$g_1$!3>,<$g_3$?int>, <$g_5$?int>,<$g_6$!2>}, $|[g_1,g_3,g_4,g_5]|$)

= {<$g_5$!1>, <$g_3$!3>, <$g_6$!2>}

## 5.3.5 - TRATAMENTO DA OCORRÊNCIA DE EVENTO

Nesta fase será tratada a definição da ocorrência de um evento, ou seja, a partir da lista de $OE_s$ derivada do primeiro nível de recursão (análise da raiz) será escolhido um dos eventos.

**. Procedimento**

i) Escolher aleatoriamente, ou em interação com o usuario, um dos eventos pertecente à lista de $OE_s$ final;

ii) Consultar a tabela de processos por porta e identificar os módulos envolvidos no evento escolhido;

iii) Chamar procedimento de atualização da Árvore de Composição - Seção 5.3.1;

iv) Chamar serviço para envio de mensagem resposta aos módulos envolvidos no evento escolhido - Seção 5.3.6;

v) Atualizar a tabela de processos por porta, retirando:

v.1) Entrada relativa à porta onde ocorreu evento;

v.2) Todas as ocorrências, nas outras entradas, dos módulos partipantes do evento escolhido;

vi) Atualizar a tabela de $OE_s$ por processo, retirando as entradas associadas aos módulos participantes da interação;

Considerando o exemplo 3 da seção 5.3.4 e supondo que:

1) O evento escolhido seja

.$<g_5$ !1>

.Processos Envolvidos - segundo a avaliação da tabela de processos por portas (fig. 5.3.10)

**P3 e P8**

.Chamar procedimento de atualização da Árvore de Composição

.**P7** deve ser desabilitado

.Chamar serviço para envio de mensagem_resposta de desabilitação;

.Atualizar a Árvore de Composição - que passará da que é mostrada na Fig. 5.3.8 para a da Fig. 5.3.9, segundo o procedimento de poda - seção 5.3.1.

.Chamar o procedimento de envio de mensagem_resposta aos processos envolvidos na interação.

. A tabela de processos por porta - que tinha inicialmente o formato da fig. 5.3.10 - passará para a que é mostrada na fig 5.3.11, tendo sido retirado $g_5$ segundo (v.1), $g_4$ e $g_6$ devido a (v.2) e P7 de acordo com procedimento de atualização da Árvore de Composição por ocorrência de desabilitação - Seção 5.3.1.

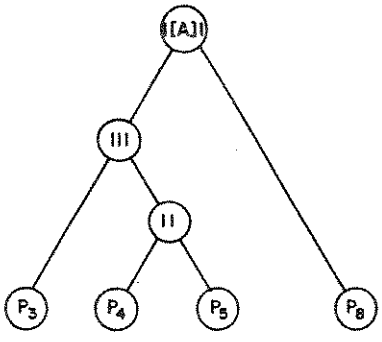



Figura 5.3.9 - Árvore de composição atualizada com a ocorrência de $< g_5$ !1 >

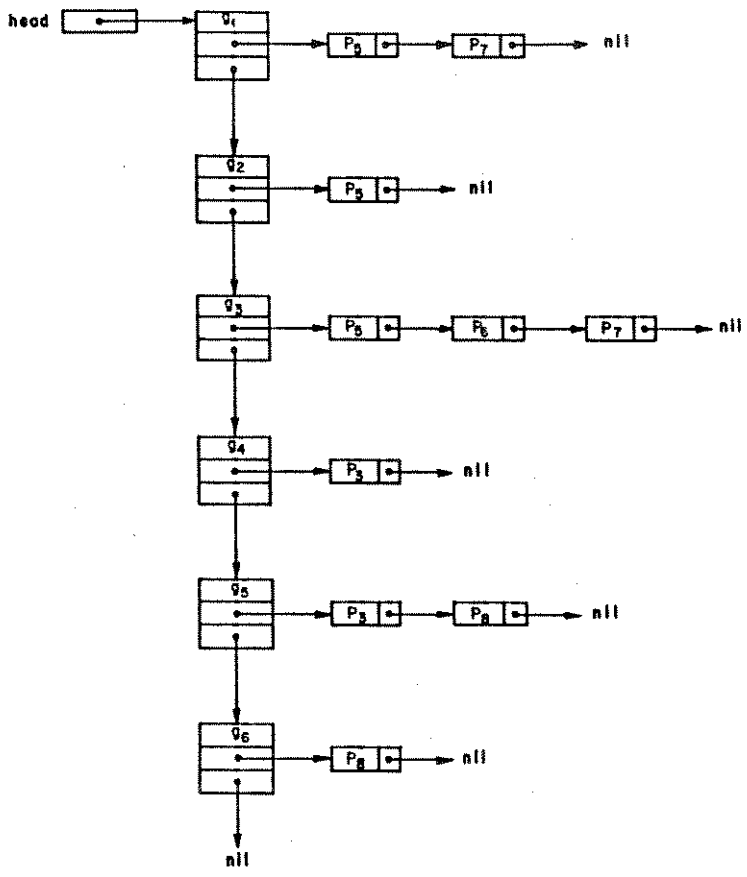



Figura 5.3.10 - Tabela de processos por portas inicial relativa ao exemplo 3 da seção 5.3.4.

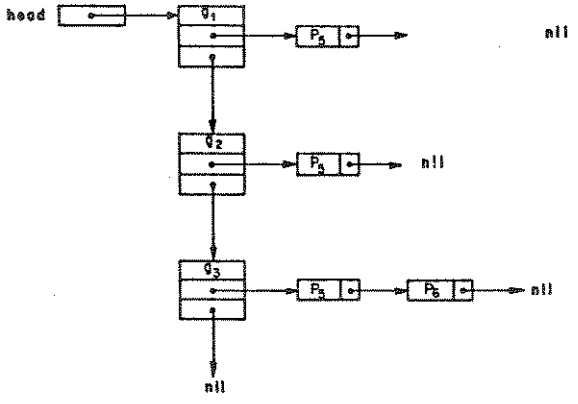



Figura 5.3.11 - Tabela de processos por porta modificada pela ocorrência do evento $< g_5 ! 1 >$

.A Tabela de $OE_s$ por processos que inicialmente tinha a forma apresentada na fig.5.3.12, resutará na forma mostrada na Fig. 5.3.13, onde $P_3$ e $P_8$ foram retirados de acordo com (vi) e P7 devido ao tratamento da desabilitação - Seção 5.3.1.

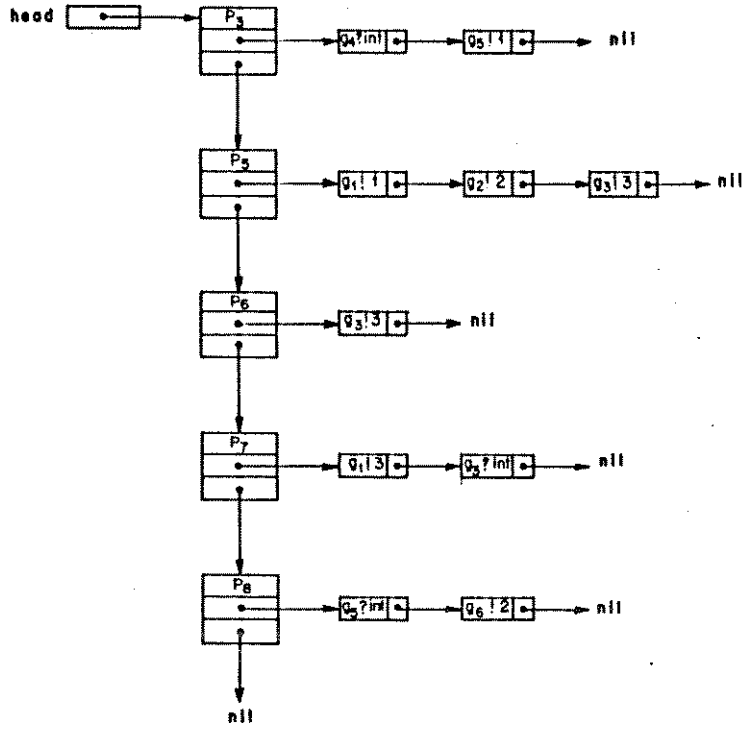



Figura 5.3.12 - Tabela de OEs por processo inicial relativa ao exemplo 3 da seção 5.3.4.

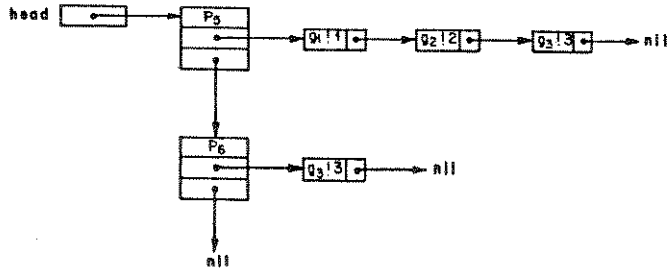



Figura 5.3.13 - Tabela de OEs por processo modificada pela ocorrência do evento < $g_5$!1 >

2) Supondo agora a ocorrência do evento

.$<g_3!3>$

.Processos envolvidos

**P5, P6 e P7**

.Não haverá atualização na Árvore de Composição;

.Chamar o procedimento de envio de mensagem_resposta aos processos envolvidos na interação.

A tabela de processos por porta da fig. 5.3.10, terá a forma mostrada na fig. 5.3.14;

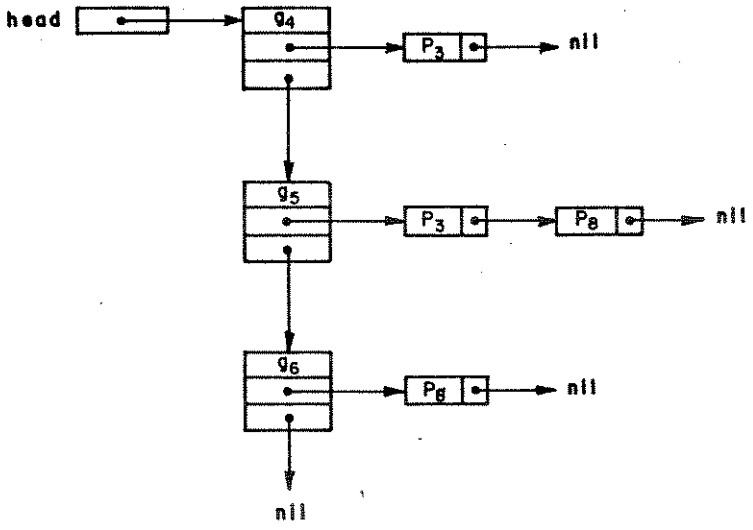



Figura 5.3.14 -Tabela de processos por porta modificada pela ocorrência do evento $< g_3 ! 3 >$

A Tabela de $OE_s$ por processo será alterada da que aparece na fig. 5.3.12 para a que é mostrada na Fig. 5.3.15;

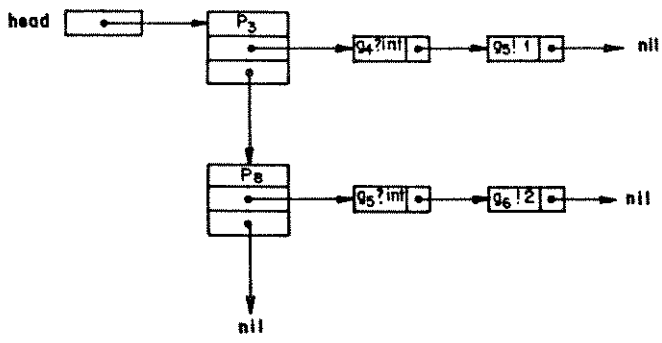



Figura 5.3.15 - Tabela de OEs por processo modificada pela ocorrência do evento < $g_3$!3 >

## 5.3.6 - Envio de mensagens_resposta aos Módulos Básicos

A última fase do ciclo de execução do Módulo Gerenciador será o envio de mensagens_resposta aos Módulos Básicos que participaram de uma interação, ou que se envolveram em alguma situação de desativação.

Através das mensagens de participação em interação, os Módulos Básicos recebem as condições negociadas na ocorrência do evento.

Por questão de simplificação, as mensagens de confirmação de terminação (aquelas que informam aos módulos que o procedimento de finalização foi concluído com sucesso) foram consideradas como sendo do mesmo tipo das mensagens de desabilitação.

Na fig. 5.3.16 encontramos o esquema de ligação de portas entre o Módulo Gerenciador e os Módulos Básicos para o envio das mensagens_resposta de participação em interação.

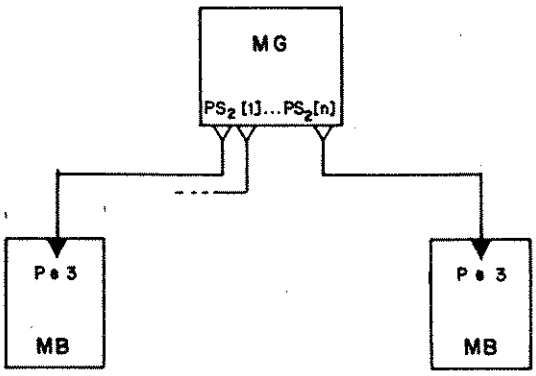



Figura 5.3.16 - Mensagem Resposta de Participação em interação.

O esquema de ligação para o caso de mensagens de desabilitação será idêntico ao da Fig. 5.3.16, sendo que um novo conjunto de portas foi definido para o Módulo Gerenciador ($PS_3[1]...PS_3[n]$ ) e uma nova porta para os Módulos Básicos ( $PE_2$).

Após o envio de todas as mensagens associadas à ocorrência de evento, desabilitação ou confirmação de terminação, o Módulo Gerenciador diminuirá sua prioridade permitindo assim que os Módulos Básicos recebam as mensagens_resposta e continuem suas execuções. O módulo que não participou do evento continuará bloqueado (cnf. função SINC, seção 5.1.1) ainda à espera de ocorrência de um evento segundo o seu conjunto de ofertas pendentes.

No próximo capítulo será apresentada a execução de uma especificação LOTOS, que será importante para demonstrar as características de simulação existentes no ambiente LSTER.

# Capítulo 6

# Exemplo de Aplicação

A especificação apresentada nesta seção foi adaptada de [LOGRIPPO 88] que descreve um serviço provedor para a interligação de dois usuários.

A estrutura geral do sistema pode ser vista na fig. 6.1.1.

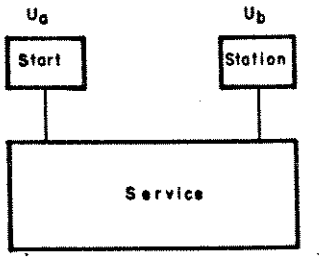



Figura 6.1.1 - Estrutura geral do Serviço Provedor.

Para se obter as características de prototipagem/simulação, a especificação foi descrita de uma forma completa (ou fechada) onde o serviço é composto paralelamente com dois usuários - $U_a$ **(start)** e $U_b$ **(station)** - para a geração espontânea de eventos.

```
hide g,d in
( start [g] (1)
|||
  station [d]
)
|[d,g]|
 service [g,d]
```

Neste exemplo é considerada somente a hipótese do usuário $U_a$ enviar dados ao usuário $U_b$. Por questões de simplificação $U_a$ apenas inicializa (start) o procedimento de envio - observar que o processo $U_a$ é instanciado com o parâmetro 1 que indica o valor inicial da sequência - sendo a sequência de dados gerada internamente pelo serviço. $U_b$ é uma estação remota sumidora de dados.

```
process start [inic] (mes_inic:int) :noexit :=
     inic!mes_inic; stop
endproc

process station [deliv] :noexit :=
     deliv?mes:int; station[deliv]
endproc
```

O processo **service** é constituído por duas entidades pares (sender e receiver) que se comunicam por um canal (processo crazyfifo) sujeito a falhas e à consequente perda de dados. O esquema de composição dos processos considera o processo crazyfifo como responsável pelo serviço n-1 que dá suporte ao serviço pres-tado pelo processo service (serviço n).

```
process service [go,deliv] :noexit :=
     go?mes_inic:int;
     hide a,b,c,e,f,h in
       ( sender[a,c,f](mes_inic,0)
       |||
         receiver[b,e,h,deliv](1)
       )
       |[a,b,c,e,f,h]|
       crayfifo [a,b,c,e,f,h] (new)
   where
     .
     .
     .
endproc
```

Três comportamentos básicos compõem a estrutura do processo **sender**. A partir do recebimento de um novo crédito, enviado pelo processo receiver, terá início o procedimento de envio de dados. A cada dado enviado, o crédito é decrementado e o próximo dado na sequência é preparado para o envio. Devido à possibilidade de perdas de dados, uma solicitação de retransmissão poderá ocorrer a qualquer momento.

```
process sender [out,req,retry] (mes:int,cred:int):noexit :=
       [gt(cred,zero)] -> out!mes;
                          sender[...](succ(mes),pred(cred))
    [] [eq(cred,zero)] -> req?new_cred:int;
                          sender[...](mes,new_cred)
    [] retry?expect:int; sender[...](expect,cred)
endproc
```

O processo **crazyfifo** é composto de dois processos. Um deles (fifo) descreve o comportamento da fila e o outro (lostdata) a perda de dados no canal de comunicação.

```
process crazyfifo [mes_in,mes_out,cred_in,cred_out,
                   retry_in, retry_out] (Q:queue):noexit :=
     hide p in
          fifo[mes_in, mes_out, cred_in,
               cred_out, retry_in, retry_out,p](new)
          |[p]| lostdata[p]
   where
     .
     .
     .
endproc
```

O processo **fifo** usa uma estrutura de dados fila (queue) para representar a capacidade do canal. As operações permitidas sobre a fila são:

.Criação de uma nova fila - **new(q)**, que cria uma fila vazia **q**;

.Inclusão de um dado na fila - **add(mes,q)**, que descreve a fila resultante pela inclusão do dado **mes** na fila **q**;

.Avaliação da quantidade de dados da fila - **empty(q)**, que verifica se a fila **q** está ou não vazia;

.Retirada do primeiro elemento da fila - **first(q)**. A fila resultante depois da retirada do primeiro elemento de uma fila é representada por **rem(q)**.

O processo fifo poderá:

.Receber um novo dado enviado pelo processo sender através da porta **mes_in** e inclui-la na fila (add(mes,q));

.Caso a fila não esteja vazia (empty(q)=false), enviar o primeiro elemento (first(q)) para o processo receiver ou simular a perda do dado (interação com o processo lostdata);

.Repassar solicitação de retransmissão e concessão de novo crédito do processo receiver para o processo sender.

```
processfifo [mes_in, mes_out, cred_in, cred_out,
        retry_in,retry_out,lst] (Q:queue) :noexit :=

  mes_in?mes:int; fifo[...](add(mes,Q))
[][not(empty(Q))] → (   mes_out!first(Q);
                        fifo[...](rem(Q))
                    [] lst!first(Q);
                        fifo[...](rem(Q))
                    )
[]retry_in?expect:int; retry_out!expect;
  fifo[...](Q)
[]cred_in?cred:int; cred_out!cred;
  fifo[...](Q)

endproc
```

O processo lostdata simula a perda de dados durante a transmissão.

```
process lostdata [lst] :noexit :=
    lst?data:int; lostdata[lst]
endproc
```

Por fim, teremos a entidade par remota representada pelo processo receiver. Este processo será responsável pela concessão de crédito ao processo sender (grant), o que manterá o procedimento de transmissão de dados. Além disso, o processo receiver poderá receber um novo dado, caso em que verificará se não foi quebrada a sequência devido à perda de dados. Quando se tratar de um dado válido (eq(ms,expect)) este será repassado ao usuário $U_b$ (station), caso contrário (gt(ms,expect)) será solicitada a retransmissão do dado perdido (expect).

```
process receiver[mes-in,grant,retry,deliv] (expect:int)
    :noexit :=

      mes-in?mes:int;
        (   [eq(mes,expect)] -> deliv!mes;
                                receiver[...](succ(expect))
         [] [gt(mes,expect)] -> retry!expect;
                                receiver[...](expect)
        )
    [] grant!succ(succ(succ(succ(succ(zero)))));
          receiver[...](expect)

endproc
```

A estrutura geral detalhada do sistema pode ser vista na fig. 6.1.2.

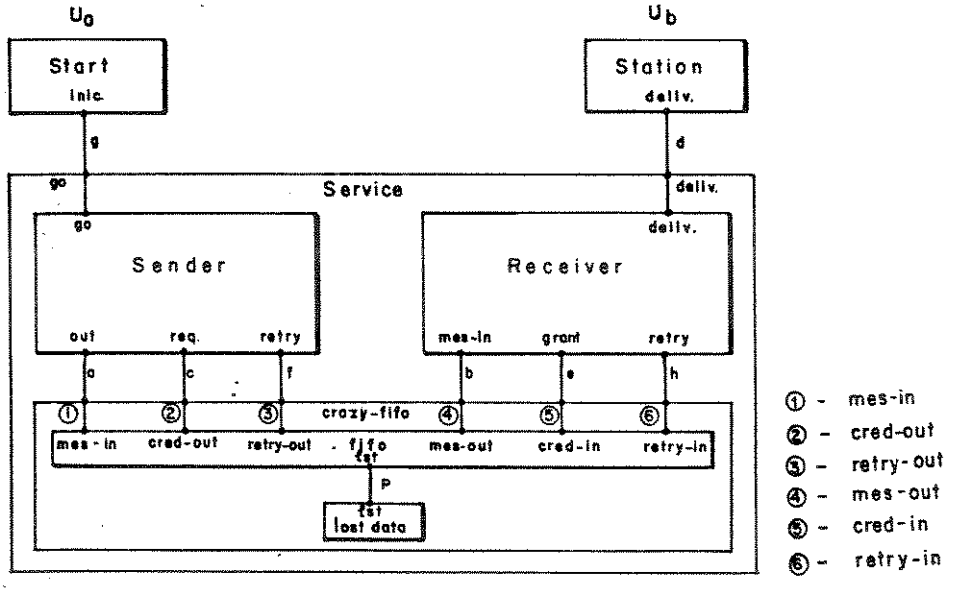



Figura 6.1.2 - Estrutura detalhada do Serviço Provedor.

A especificação LOTOS completa para este exemplo encontra-se descrita no apêndice C.

. **Estrutura da Aplicação LSTER**

Por questões de simplificação será apresentada a descrição de apenas dois módulos LSTER e do arquivo de configuração na forma reduzida. A descrição completa de todos os módulos da aplicação encontra-se no apêndice B.

**- Módulo Especificação (Spec)**

O Módulo Especificação, que descreve o comportamento associado à cláusula **behavior** terá a forma:

```
MODULE Spec;
        .
        .
        .
    BEGIN_MODULE
      LOOP
          Par('p0','p1','service',['d','g']);
          Pari('p1','user','station');
          Endy
      END_LOOP;
    END_MODULE.
```

**- Módulo Sender**

O Módulo Sender apresenta um maior nível de detalhamento da estrutura de mapeamento. Podemos observar pela comparação com o processo sender que a tradução é clara e direta.

```
MODULE Sender;
      .
      .
      .
   BEGIN_MODULE
      inic1:EnbProc;
      mes:=succ(0);
      cred:=0;
         LOOP
         inic2:...;
         IF gt(cred,0) THEN L1('a',mes,'int');
         IF eq(cred,0) THEN L2('c','int');
         L2('f','int');
         Sinc;
         CASE m2.g OF
               'a': BEGIN
                       mes:=Succ(mes);
                       cred:=Pred(cred);
                       AutoExec
                    END;
               'c': BEGIN
                       new_cred:=m2.v;
                       cred:=new_cred;
                       AutoExec
                    END;
               'f': BEGIN
                       expect:=m2.v;
                       mes:=expect;
                       AutoExec
                    END
         END
         END_LOOP;
   END_MODULE.
```

**- Arquivo de Configuração**

```
CONFIGURATION Serviço;
     INSTANCE
            p1:start;
            p2:service;
            p3:sender;
            p4:crazyfifo;
            p5:receiver;
            p6:station;
            p7:fifo;
            p8:lostdata;
            b1:b;
            s1:spec;
     CREATE
            p1/S=64/H=64,
            p2/S=64/H=64,
            p3/S=64/H=512,
            p4/S=128/H=512,
            p5/S=64/H=512,
```

```
            p6/S=64/H=512,
            p7/S=64/H=512,
            p8/S=64/H=512,
            S1/S=64/H=512,
            b1/S=2500/H=2000;
      LINK
            s1.PS1 TO b1.PE1;
            p1.PS1 TO b1.PE1;
            .
            .
            p1.PS2 TO b1.PE2;
            .
            .
            p1.PS3 TO b1.PE3;
            .
            .
            b1.PS1[1] TO p1.PE1;
            .
            .
            b1.PS2[1] TO p1.PE2;
            .
            .
            b1.PS3[1] TO p1.PE3;
            .
            .
END_CONFIG.
```

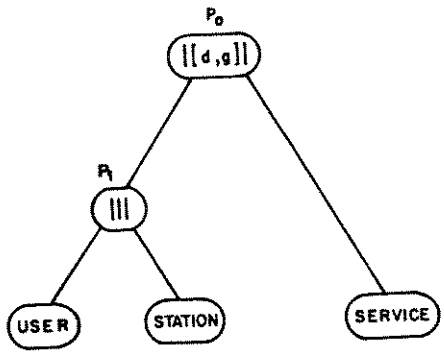
. **Execução**

Inicialmente, cada módulo participante da aplicação, exceto o Módulo Especificação, executará o procedimento de inicialização, onde passará ao módulo gerenciador o identificador do nome do processo que descreve, e em seguida entra em estado de bloqueio, no qual ficará à espera de ativação. Esta fase funciona como a inicialização da aplicação.

Deve-se notar a importância da ordem de cria-ção dos módulos - CREATE - no arquivo de configuração. O Módulo Spec aparece como penúltimo na lista de criação sendo seguido pelo módulo gerenciador. Isto implica que o Módulo Spec só entrará em execução após a inicialização de todos os módulos básicos.

A próxima fase da execução será a criação da árvore de composição inicial (ACI). Para isso, o Módulo Especificação se comunicará com o módulo gerenciador através dos construto-res PAR (...) e PARI (...).

A ACI criada pelo módulo gerenciador é representado no monitoramento da execução por:




```
Arvore de Composicao
p: p0
op:              2
ptr1: p1
ptr2: service

p: p1
op:              3
ptr1: start
ptr2: station

p: start
op:              0
ptr1: nil
ptr2: nil

p: station
op:              0
ptr1: nil
ptr2: nil

p: service
op:              0
ptr1: nil
ptr2: nil

Fim da Arvore de Composicao
```

Após a criação da ACI, o Módulo Gerencidor entrará na fase de ativação de módulos básicos, representado na execução por:

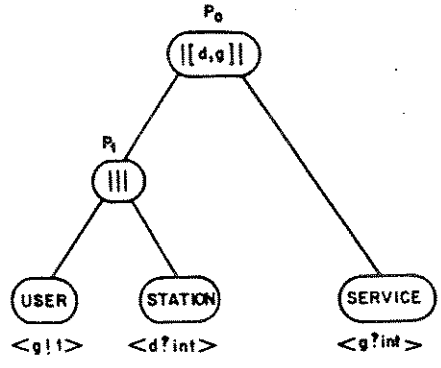
```
Fase de Ativacao de Processos
Habilita Proc.: start
Habilita Proc.: station
Habilita Proc.: service
Fim da Fase de Ativacao de Processos


Start -) ok.
Station -) ok.
Service -) ok.


Lista de Processos Habilitados
Proc.: start
Proc.: station
Proc.: service
Fim da Lista de Processos Habilitados
```

O Módulo Gerenciador reduz sua prioridade promovendo o reescalonamento da fila de pronto e a consequente execução dos módulos básicos ativados. Estes módulos apresentam suas ofertas e em seguida entram em estado de bloqueio à espera de uma mensagem de participação em evento.

Com o bloqueio dos módulos básicos, o Módulo Gerenciador retorna à execução através da atualização da árvore de composição. Como não houve nenhum pedido de instanciação de novos processos, a árvore não sofre alteração.

Em seguida o Módulo Gerenciador recebe as ofertas de eventos apresentadas pelos módulos básicos e cria a Tabela de OEs por Processo.

```
Tabela de OEs por Processos
Proc.: start
g: g
■: !
v: 1
t: int
c: TRUE
Proc.: station
g: d
■: ?
v: 0
t: int
c: TRUE
Proc.: service
g: g
■: ?
v: 0
t: int
c: TRUE
Fim da Tabela de OEs por Processos
```



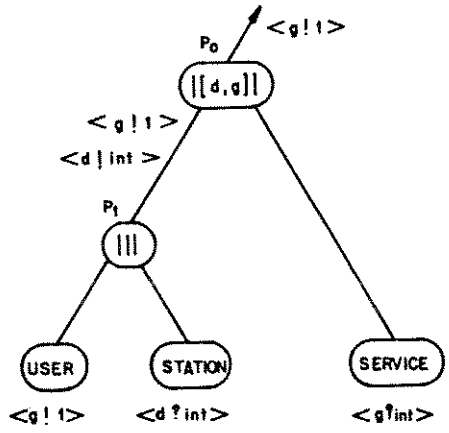

A lista de eventos possíveis é definida pela avaliação da árvore de composição conforme as OEs apresentadas. A seguir pode-se acompanhar o procedimento completo de avaliação da árvore pelas listas parciais de eventos possíveis relativas a cada nó.

Lista parcial de eventos relativa ao no: start

g: g
m: !
v: 1
t: int
c: TRUE
Fim de lista

Lista parcial de eventos relativa ao no: station

g: d
m: ?
v: 0
t: int
c: TRUE
Fim de Lista




Lista parcial de eventos relativa ao no: p1

g: g
m: !
v: 1
t: int
c: TRUE
g: d
m: ?
v: 0
t: int
c: TRUE
Fim de Lista

Lista parcial de eventos relativa ao no: service

g: g
m: ?
v: 0
t: int
c: TRUE
Fim de Lista

Lista parcial de eventos relativa ao no: p0

g: g
m: !
v: 1
t: int
c: TRUE
Fim de Lista

Em seguida um dos eventos da lista de eventos possíveis é escolhido (em interação com o usuário - execução passo-a-passo - ou aleatoriamente - geração espontânea de eventos -).

```
Lista de Eventos Possiveis
(1) g: g
    m: !
    v: 1
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis

Entre com o numero do evento escolhido
1

Porta escolhida: g
```

Os participantes da interação são obtidos, a partir da Tabela de Processos por Porta, na entrada relativa à porta onde ocorreu o evento.

```
Tabela de Processos por Portas
Porta: g
Proc.: start
Proc.: service
Porta: d
Proc.: station
Fim da Tabela de Processos por Portas
```

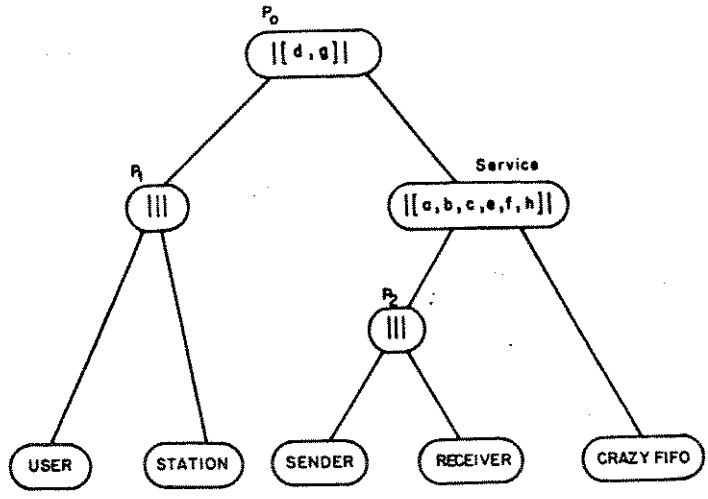
O Módulo Gerenciador novamente reduz sua prioridade permitindo a nova execução dos módulos básicos participantes da interação.

```
Start: Sinc. com valor 1
Service: Recebi start 1
```

O Módulo Start encerra sua participação na execução através do construtor STOP. O Módulo Service expande-se através da instanciação dos módulos Sender, Receiver e Crazyfifo.

Ao tornar à execução, o módulo gerenciador atualiza a árvore de composição segudo a expansão do nó terminal SERVICE.

```
Arvore de Composicao Atualizada
p: p0
op:              2
ptr1: p1
ptr2: service

p: p1
op:              3
ptr1: start
ptr2: station

p: start
op:              0
ptr1: nil
ptr2: nil

p: station
op:              0
ptr1: nil
ptr2: nil

p: service
op:              2
ptr1: p2
ptr2: crazyfifo

p: p2
op:              3
ptr1: sender
ptr2: receiver

p: sender
op:              0
ptr1: nil
ptr2: nil

p: receiver
op:              0
ptr1: nil
ptr2: nil

p: crazyfifo
op:              0
ptr1: nil
ptr2: nil

Fim da Arvore de Composicao Atualizada
```



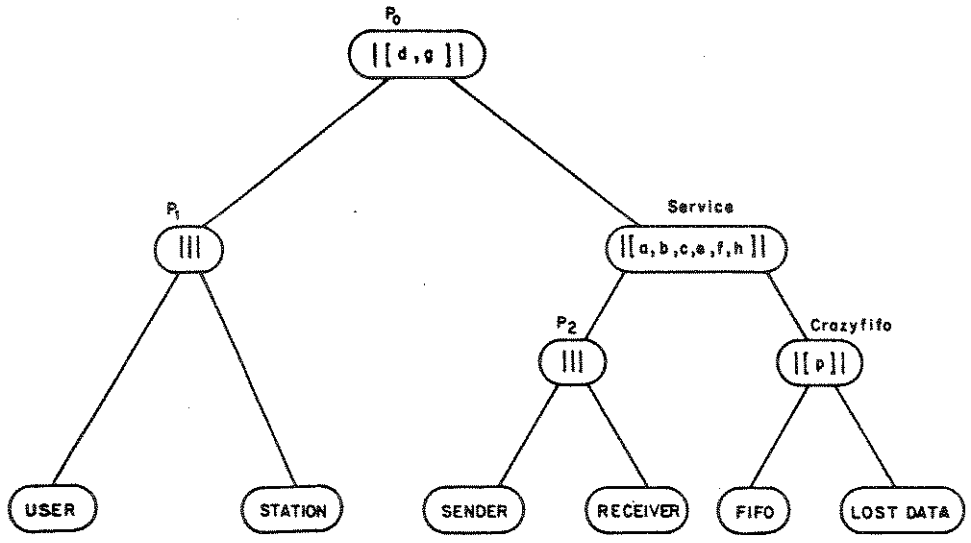

O Módulo Gerenciador ativa os novos módulos instanciados.

```
Fase de Ativacao de Processos
Habilita Proc.: sender
Habilita Proc.: receiver
Habilita Proc.: crazyfifo
Fim da Fase de Ativacao de Processos
```

```
Sender -) ok.
Receiver -) ok.
Crazyfifo -) ok

Lista de Processos Habilitados
Proc.: start
Proc.: station
Proc.: service
Proc.: sender
Proc.: receiver
Proc.: crazyfifo
Fim da Lista de Processos Habilitados
```

Os novos módulos apresentam suas ofertas. Enquanto os módulos Sender e Receiver fazem ofertas de eventos, o Módulo Crazyfifo solicita novas instanciações - Fifo e Lostdata.

Como o Módulo Gerenciador trata todas as alterações da árvore de composição antes de receber as OEs, novamente atualiza a árvore, incluindo os nós terminais Fifo e Lostdata, e ativa os novos módulos.

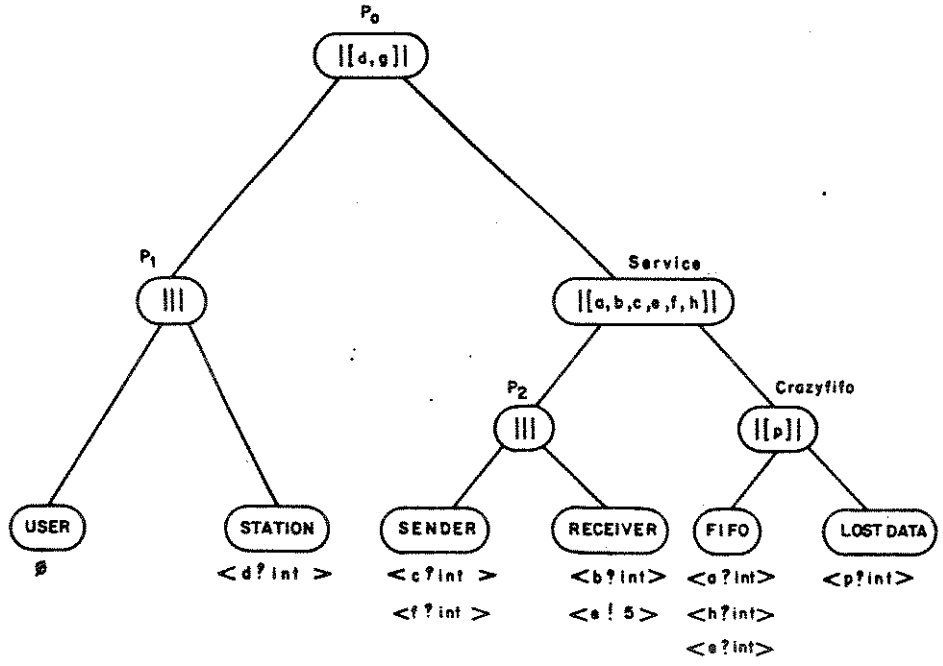


```
Arvore de Composicao Atualizada
p: p0
op:             2
ptr1: p1
ptr2: service

p: p1
op:             3
ptr1: start
ptr2: station

p: start
op:             0
ptr1: nil
ptr2: nil

p: station
op:             0
ptr1: nil
ptr2: nil

p: service
op:             2
ptr1: p2
ptr2: crazyfifo

p: p2
op:             3
ptr1: sender
ptr2: receiver

p: sender
op:             0
ptr1: nil
ptr2: nil

p: receiver
op:             0
ptr1: nil
ptr2: nil

p: crazyfifo
op:             2
ptr1: fifo
ptr2: lostdata

p: fifo
op:             0
ptr1: nil
ptr2: nil

p: lostdata
op:             0
ptr1: nil
ptr2: nil

Fim da Arvore de Composicao Atualizada
```

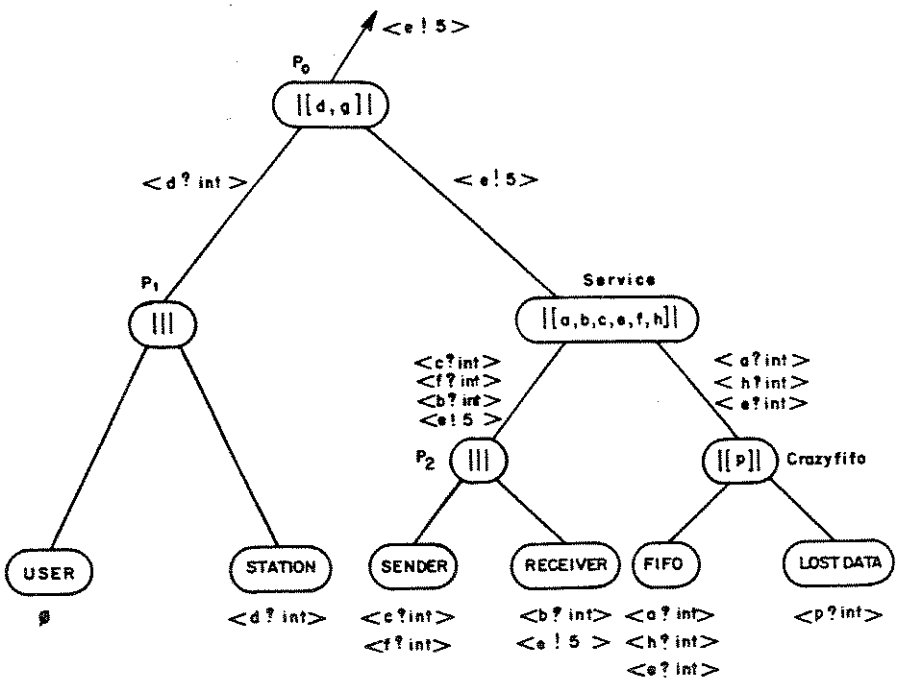
```
Fase de Ativacao de Processos
Habilita Proc.: fifo
Habilita Proc.: lostdata
Fim da Fase de Ativacao de Processos

Fifo -) ok.
Losldata -) ok.

Lista de Processos Habilitados
Proc.: start
Proc.: station
Proc.: service
Proc.: sender
Proc.: receiver
Proc.: crazyfifo
Proc.: fifo
Proc.: lostdata
Fim da Lista de Processos Habilitados
```

Após as alterações na árvore, o Módulo Gerenciador passa a receber todas as OEs apresentadas.

```
Tabela de OEs por Processos
Proc.: station
g: d
■: ?
v: 0
t: int
c: TRUE
Proc.: sender
g: c
■: ?
v: 0
t: int
c: TRUE
g: f
■: ?
v: 0
t: int
c: TRUE
Proc.: receiver
g: b
■: ?
v: 0
t: int
c: TRUE
g: e
■: !
v: 5
t: inl
c: TRUE
Proc.: fifo
g: a
■: ?
v: 0
t: int
c: TRUE
g: h
■: ?
v: 0
t: int
c: TRUE
g: e
■: ?
v: 0
t: inl
c: TRUE
Proc.: lostdata
g: p
■: ?
v: 0
t: inl
c: TRUE
Fim da Tabela de OEs por Processos
```



A nova lista de eventos possíveis é derivada a partir das OEs e da árvore de composição.




```
Lista parcial de eventos relativa ao no: start
Fim de Lista


Lista parcial de eventos relativa ao no: station
g: d
m: ?
v: 0
t: int
c: TRUE
Fim de Lista


Lista parcial de eventos relativa ao no: p1
g: d
m: ?
v: 0
t: int
c: TRUE
Fim de Lista
```

```
Lista parcial de eventos relativa ao no: sender
g: c
m: ?
v: 0
t: int
c: TRUE
g: f
m: ?
v: 0
t: int
c: TRUE
Fim de Lista


Lista parcial de eventos relativa ao no: receiver
g: b
m: ?
v: 0
t: int
c: TRUE
g: c
m: !
v: 5
t: int
c: TRUE
Fim de Lista


Lista parcial de eventos relativa ao no: p2
g: c
m: ?
v: 0
t: int
c: TRUE
g: f
m: ?
v: 0
t: int
c: TRUE
g: b
m: ?
v: 0
t: int
c: TRUE
g: e
m: !
v: 5
t: int
c: TRUE
Fim de Lista
```

```
Lista parcial de eventos relativa ao no: fifo
g: a
m: ?
v: 0
t: int
c: TRUE
g: h
m: ?
v: 0
t: int
c: TRUE
g: e
m: ?
v: 0
t: int
c: TRUE
Fim de Lista


Lista parcial de eventos relativa ao no: lostdata
g: p
m: ?
v: 0
t: int
c: TRUE
Fim de Lista



Lista parcial de eventos relativa ao no: crazyfifo
g: a
m: ?
v: 0
t: int
c: TRUE
g: h
m: ?
v: 0
t: int
c: TRUE
g: e
m: ?
v: 0
t: int
c: TRUE
Fim de Lista
```

```
Lista parcial de eventos relativa ao no: servico
g: e
m: !
v: 5
t: int
c: TRUE
Fim de Lista


Lista parcial de eventos relativa ao no: p0
g: e
m: !
v: 5
t: int
c: TRUE
Fim de Lista
```

A tabela de processos por porta indica os módulos participantes na interação.

```
Tabela de Processos por Portas
Porta: d
Proc.: station
Porta: c
Proc.: sender
Porta: f
Proc.: sender
Porta: b
Proc.: receiver
Porta: e
Proc.: receiver
Proc.: fifo
Porta: a
Proc.: fifo
Porta: h
Proc.: fifo
Porta: p
Proc.: lostdata
Fim da Tabela de Processos por Portas


Lista de Eventos Possiveis
(1) g: e
    m: !
    v: 5
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis

Entre com o numero do evento escolhido
1

Porta escolhida: c
```

Os módulos participantes da interação tornam à execução.

```
Receiver: Enviei novo credito : 5
Receiver -) ok.
Fifo: Recebi novo credito: 5
```

O Processo Sender receberá a ordem de crédito na próxima interação ocorrida na especificação. Os passos desta interação, conforme o monitoramento da execução, é apresentada, sem comentários adicionais, a seguir:

```
Tabela de OEs por Processos
Proc.: station
g: d
m: ?
v: 0
t: int
c: TRUE
Proc.: sender
g: c
m: ?
v: 0
t: int
c: TRUE
g: f
m: ?
v: 0
t: int
c: TRUE
Proc.: lostdata
g: p
m: ?
v: 0
t: int
c: TRUE
Proc.: receiver
g: b
m: ?
v: 0
t: int
c: TRUE
g: c
m: !
v: 5
t: int
c: TRUE
Proc.: fifo
g: c
m: !
v: 5
t: int
c: TRUE
Fim da Tabela de OEs por Processos
```

```
Tabela de Processos por Portas
Porta: d
Proc.: station
Porta: c
Proc.: sender
Proc.: fifo
Porta: f
Proc.: sender
Porta: p
Proc.: lostdata
Porta: b
Proc.: receiver
Porta: e
Proc.: receiver
Fim da Tabela de Processos por Portas
```

```
Lista de Eventos Possiveis
(1) g: c
    m: !
    v: 5
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis

Entre com o numero do evento escolhido
1

Porta escolhida: c


Sender: Recebi novo credito : 5
Sender -) ok.
Fifo: Enviei novo credito: 5
Fifo -) ok.
```

**. Resultados da Prototipagem**

A prototipagem da especificação proporcionou a deteção de algumas características não desejáveis do sistema. Apesar da simplicidade do serviço, pôde-se verificar que o sistema poderia evoluir para situações não permitidas.

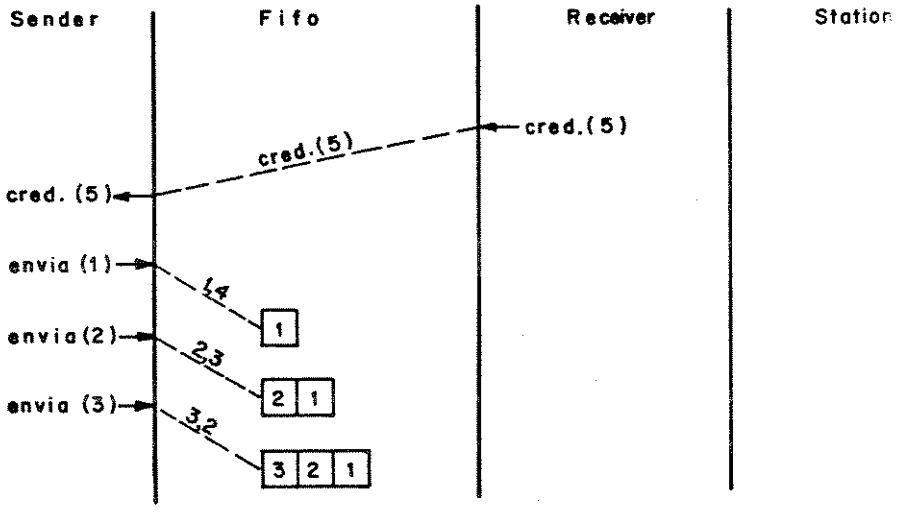
Serão descritas duas destas situações. Para simplificar a apresentação, serão mostradas apenas alguns trechos do monitoramento da execução.

O propósito aqui será unicamente de verificar a especificação e não de propor soluções para os erros porventura encontrados.

**- Situação 1**

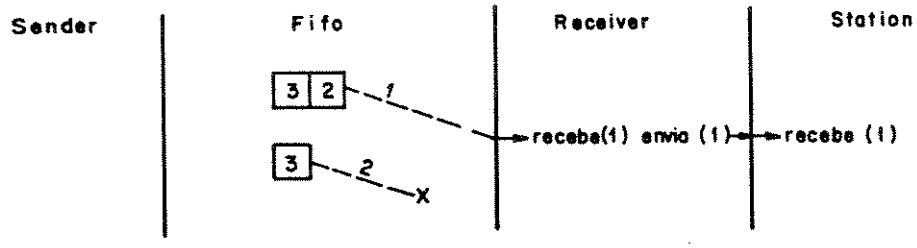
Consideremos inicialmente a sequência de eventos, descritos anteriormente, relativa à concessão de crédito. Com o novo crédito recebido a entidade sender envia os dados 1, 2 e 3 (observar a decrementação do crédito, no diagrama temporal, a cada dado enviado. p.ex.: 1,4; 2,3 ...)

```
Sender: Enviei: 1
Sender -) ok.
Fifo: Recebi dado: 1
Fifo -) ok.

Sender: Enviei: 2
Sender -) ok.
Fifo: Recebi dado: 2
Fifo -) ok.

Sender: Enviei: 3
Sender -) ok.
Fifo: Recebi dado: 3
Fifo -) ok.
```



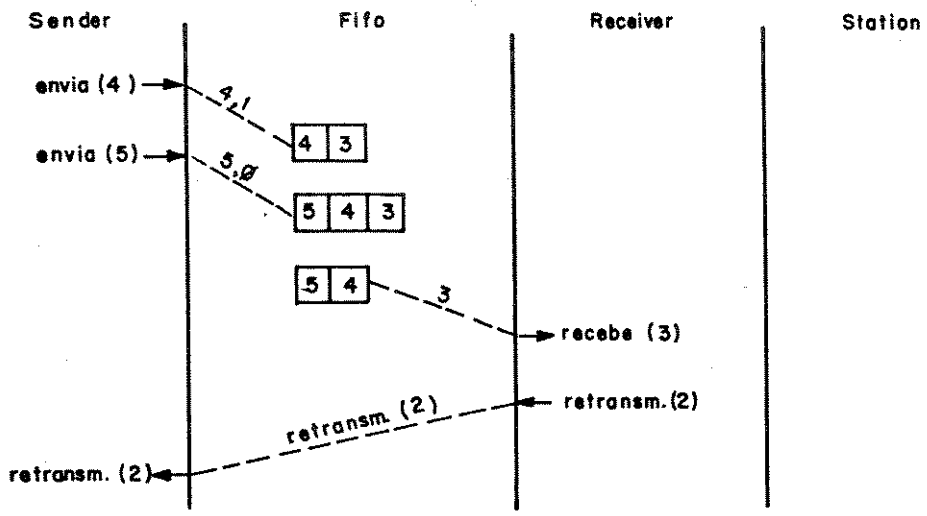

Em seguida a entidade receiver recebe o dado 1 e o entrega à estação. Devido a problemas no canal de comunicação o dado 2 é perdido.

```
Receiver: Recebi dado: 1
Fifo: Enviei dado : 1
Fifo -) ok.

Station: Recebi: 1
Station -) ok.
Receiver: Enviei dado: 1
Receiver -) ok.

Lostdata: Recebi dado perdido 2
Lostdata -) ok.
Fifo: Dado perdido : 2
Fifo -) ok.
```




Na sequência, a entidade sender envia os dados 4 e 5 e entra em um estado no qual não poderá transmitir nenhum outro dado por não possuir crédito. A execução segue com a entidade receiver recebendo o dado 3. Como o dado esperado é o 2, esta entidade solicita que ele seja retransmitido.

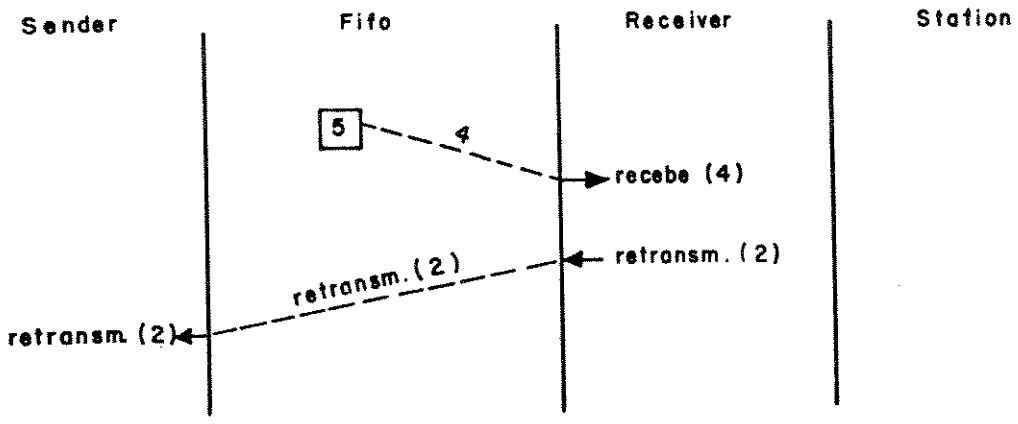
```
Sender: Enviei: 4
Sender -) ok.
Fifo: Recebi dado: 4
Fifo -) ok.

Sender: Enviei: 5
Sender -) ok.
Fifo: Recebi dado: 5
Fifo -) ok.

Receiver: Recebi dado: 3
Fifo: Enviei dado : 3
Fifo -) ok.

Receiver: Enviei solicitacao de retransmisao do dado: 2
Receiver -) ok.
Fifo: Recebi pedido para retransmitir dado: 2

Sender: Recebi pedido para retransmitir dado: 2
Sender -) ok.
Fifo: Enviei pedido para retransmitir dado: 2
Fifo -) ok.
```




Apesar da solicitação de retransmissão a entidade sender não poderá enviar o dado 2, pois não possui crédito. Este fato pode ser observado pela lista de eventos possíveis onde estão presentes: b!4 (receiver interage com fifo para receber dado 4); e!5 (receiver envia novo crédito) e i!4 (perda do dado 4). Na lista não existe o evento a!2, que representaria o envio do dado 2 pela entidade sender. Para dar sequência à execução, foi escolhido o evento b!4. Novamente por não ser o dado esperado é solicitada a retransmissão do dado 2.

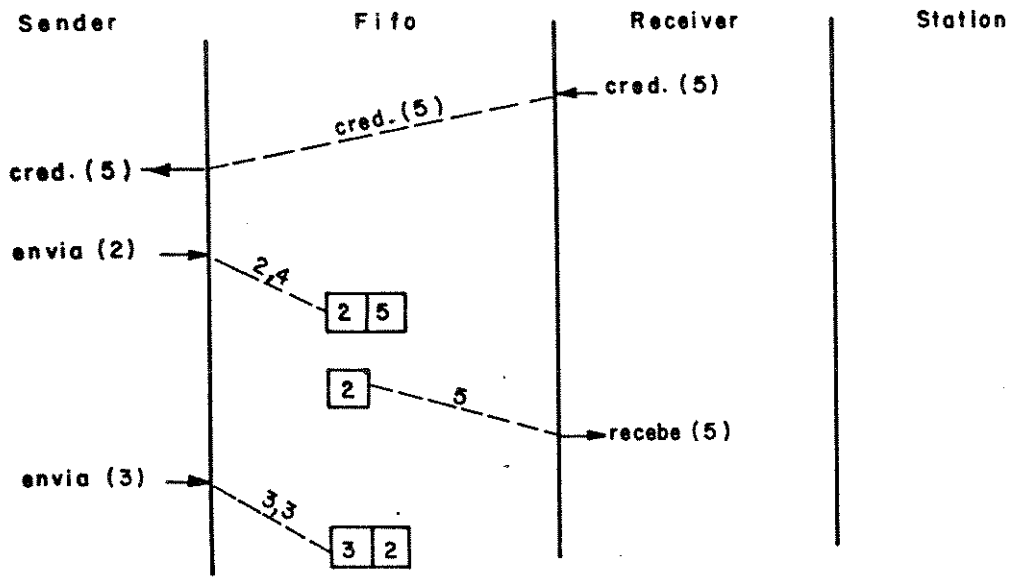
```
Lista de Eventos Possiveis
(1) g: b
    m: !
    v: 4
    t: int
    c: TRUE

(2) g: c
    m: !
    v: 5
    t: int
    c: TRUE

(3) g: p
    m: !
    v: 4
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis
```

```
Entre com o numero do evento escolhido
1

Porta escolhida: b

Receiver: Recebi dado: 4
Fifo: Enviei dado : 4
Fifo -) ok.

Receiver: Enviei solicitacao de retransmisao do dado: 2
Receiver -) ok.
Fifo: Recebi pedido para retransmitir dado: 2

Sender: Recebi pedido para retransmitir dado: 2
Sender -) ok.
Fifo: Enviei pedido para retransmitir dado: 2
Fifo -) ok.
```




Somente após a concessão de um novo crédito a entidade sender consegue enviar o dado solicitado. Em seguida consideramos a sequência de eventos descritos no diagrama temporal.

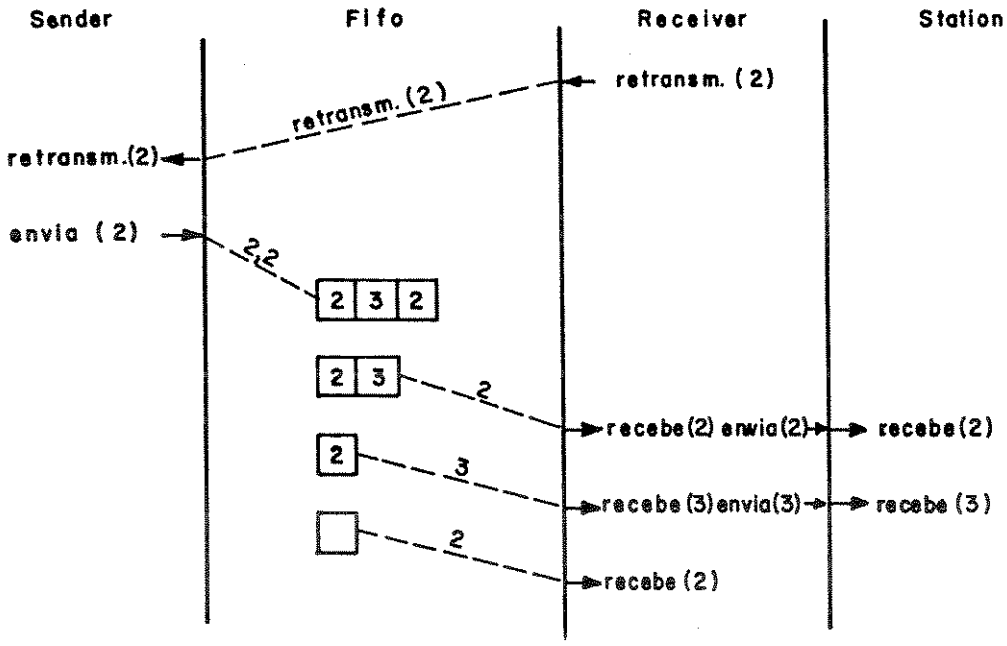
```
Receiver: Enviei novo credito : 5
Receiver -) ok.
Fifo: Recebi novo credito: 5

Sender: Recebi novo credito : 5
Sender -) ok.
Fifo: Enviei novo credito: 5
Fifo -) ok.

Sender: Enviei: 2
Sender -) ok.
Fifo: Recebi dado: 2
Fifo -) ok.

Receiver: Recebi dado: 5
Fifo: Enviei dado : 5
Fifo -) ok.

Sender: Enviei: 3
Sender -) ok.
Fifo: Recebi dado: 3
Fifo -) ok.
```




A recepção do dado 5 provoca nova solicitação de retransmissão do dado 2. Com isso a entidade sender envia novamente este dado. Esta é uma situação indesejável porque existem na fila dois dados 2. A ocorrência do envio do último dado 2 se deve à incorreções na especificação. Na sequência dos eventos é recebido os dados 2 e 3. Em seguida o outro dado 2 é recebido.

```
Receiver: Enviei solicitacao de retransmissao do dado: 2
Receiver -) ok.
Fifo: Recebi pedido para retransmitir dado: 2

Sender: Recebi pedido para retransmitir dado: 2
Sender -) ok.
Fifo: Enviei pedido para retransmitir dado: 2
Fifo -) ok.

Sender: Enviei: 2
Sender -) ok.
Fifo: Recebi dado: 2
Fifo -) ok.

Receiver: Recebi dado: 2
Fifo: Enviei dado : 2
Fifo -) ok.

Station: Recebi: 2
Station -) ok.
Receiver: Enviei dado: 2
Receiver -) ok.
```

```
Fifo: Enviei dado : 3
Fifo -) ok.
Receiver: Recebi dado: 3

Station: Recebi: 3
Station -) ok.
Receiver: Enviei dado: 3
Receiver -) ok.

Fifo: Enviei dado : 2
Fifo -) ok.
Receiver: Recebi dado: 2
```




- **Situação 2** - Deteção de DEADLOCK.

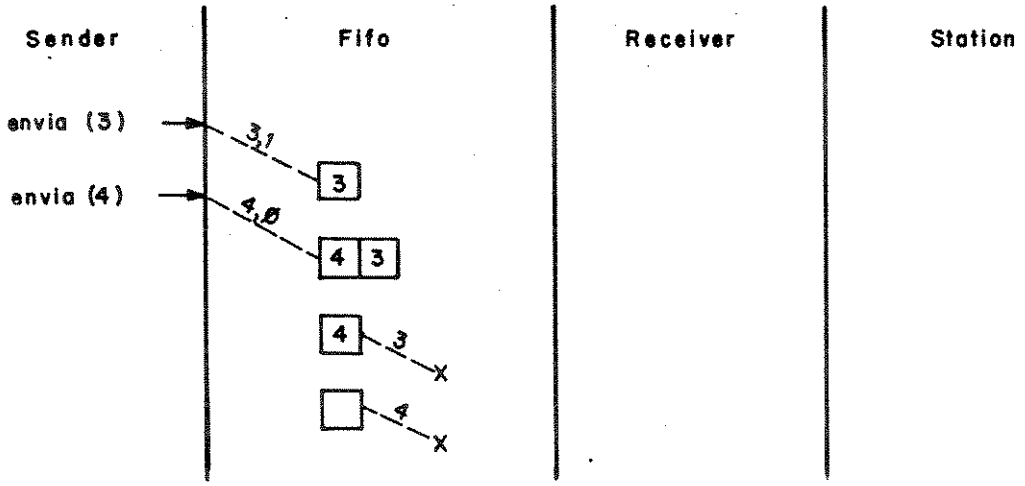
O ambiente de execução é capaz de detetar a presença de DEADLOCK na especificação.

Considerando ainda a execução descrita na situação 1, podemos observar que a recepção do dado 2 pela segunda vez põe em deadlock o processo receiver (neste

processo não é prevista a recepção de um dado com ordem de sequência inferior à esperada). O DEADLOCK do processo receiver pode ser detetado pela ausência de novas ofertas apresentadas por ele (examinar lista de eventos possíveis nas etapas de execução seguintes).

Em continuação à execução, sejam os eventos relacionados no diagrama temporal abaixo.

```
Lista de Eventos Possiveis
(1) g: a
    m: !
    v: 3
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis

Entre com o numero do evento escolhido
1

Porta escolhida: a

Sender: Enviei: 3
Sender - ) ok.
Fifo: Recebi dado: 3
Fifo - ) ok.
```



```
Lista de Eventos Possiveis
(1) g: a
    m: !
    v: 4
    t: int
    c: TRUE

(2) g: p
    m: !
    v: 3
    t: int
    c: TRUE

Fim da Lista de Eventos Possiveis

Entre com o numero do evento escolhido
1

Porta escolhida: a

Sender: Enviei: 4
Sender -) ok.
Fifo: Recebi dado: 4
Fifo -) ok.

Lostdata: Recebi dado perdido 4
Lostdata -) ok.
Fifo: Dado perdido : 4
Fifo -) ok.

Lostdata: Recebi dado perdido 3
Lostdata -) ok.
Fifo: Dado perdido : 3
Fifo -) ok.
```

Neste ponto o sistema entra em DEADLOCK. O processo sender não evolui por não possuir crédito. O processo receiver já estava em DEADLOCK. O processo fifo bloqueia-se por não existir nenhuma ação a executar (fila vazia e nenhuma solicitação de envio ou entrega de dado).

O próximo ciclo de execução deriva uma lista nula de eventos possíveis. Isto implica que todo sistema está bloqueado e portanto na existência de um DEADLOCK.

```
Lista de Eventos Possiveis
Fim da Lista de Eventos Possiveis


ATENCAO!!! Existe Deadlock na Especificacao
```

# Capítulo 7

# Conclusão

No início deste trabalho foi discutido, de maneira introdutória, a importância da definição de uma metodologia de projeto e realização de sistemas, utilizando como linguagem para representação do modelo do sistema uma técnica de descrição formal.

Existem dois aspectos importantes em uma metodologia desta natureza. O primeiro é que com o aumento da complexidade dos sistemas torna-se indispensável o uso de técnicas formais para se descrever o sistema de forma clara, não ambígua, precisa e concisa. O segundo aspecto é o ganho de estruturação e sistematização obtida seguindo-se uma metodologia coerente e bem definida.

A principal característica da metodologia apresentada é a divisão em duas fases distintas do processo de projeto. Na primeira fase serão definidos os aspectos arquitetuais do sistema. Neste sentido vimos a importância do uso de LOTOS devido ao seu alto poder de abstração, das suas propriedades de estruturação (cnf. o uso de estilos de especificação), e do seu formalismo (favorecendo a contrução de ferramentas de apoio ao projeto, p.ex.: editores, analisadores, simuladores, etc.), que permite a definição de uma arquitetura bastante expressiva do sistema. Na segunda fase serão definidos os aspectos de realização do sistema, onde questões mais voltadas à implementação serão abordados.

Segundo esta metodologia, no fim da fase arquitetural deverá ocorrer a tradução da especificação abstrata final em uma especificação de implementação inicial. O grande problema desta tradução são as diferenças semânticas dos elementos das duas linguagens de descrição (no nosso caso: LOTOS na fase arquitetural e STER na fase de realização).

Apesar da importância desta metodologia e da necessidade de um estudo mais detalhado para a sua definição, este trabalho não foi encaminhado por esta linha. Primeiro porque este é um assunto que foge ao escopo de uma tese de mestrado. Em segundo lugar, e certamente o fator determinante, é que muitas destas questões só se tornaram mais claras nas fases finais deste trabalho.

Portanto, a ênfase principal do trabalho foi a definição dos mecanismos de tradução de uma especificação LOTOS em uma aplicação STER, sendo que a metodologia foi apresentada para definir o contexto em que se inseria o trabalho.

O trabalho foi dividido em duas etapas principais. Na primeira etapa foi definida a estratégia de mapeamento, que levava em conta a inclusão de um módulo extra (Módulo Gerenciador) à aplicação STER (exercendo o papel de sevidor semân tico), necessário para que se pudesse migrar entre os dois mundos. Na segunda etapa todos os mecanismos, tanto para a construção do Módulo Gerenciador como na construção dos demais módulos da aplicação, foram implementados.

Duas considerações importantes marcaram a definição da estratégia de mapeamento. Por um lado a utilização do modelo de execução de especificações LOTOS (cnf. [WU 89]) se adequou convenientemente à nossa proposta, visto que a funcionalidade necessária para Módulo Gerenciador seria completamente obtida implementando-se tal modelo. A outra consideração importante foi a estratégia adotada para a dinâmica de execução da aplicação STER, onde foi bastante explorado o recurso de mudança de prioridade do Módulo Gerenciador, tendo sido obtido um interessante mecanismo de bloqueio e desbloqueio dos módulos STER em função da oferta(s) de evento(s) e da participação em uma interação, respectivamente. Para a validação destas duas considerações foram fundamentais as conclusões tiradas com a prototipagem do exemplo descrita no capítulo 6.

Os resultados do trabalho permitiram a conclusão da viabilidade da proposta inicial. Os principais destes resultados foram:

.A definição de regras de mapeamento favorece a proposição de uma metodologia de projeto e realização como aquela discutida no capítulo 1;

.A comparação dos resultados de simulação da especificação final LOTOS com os resultados da prototipagem da aplicação inicial LSTER, permite a análise da consistência da tradução. Neste caso poderíamos comparar os resultados obtidos com a execução do protótipo (aplicação LSTER) construído no capítulo 6, com a simulação da especificação LOTOS;

.A substituição da simulação, como utilizada na fase arquitetural, pela prototipagem na fase de realização, permite que seja seguida uma mesma política de refinamento de especificações, mesmo considerando distintos os objetivos de cada fase.

Um aspecto importante com relação à prototipagem, é que a execução do protótipo poderá se dar de três formas: Através da geração automática de cenários de teste, o que permite a verificação exaustiva do protótipo; Através da validação de cenários propostos, o que permite a verificação de aspectos específicos, e da execução passo-a-passo. No caso do exemplo descrito no capítulo 6, os cená-

rios de teste apresentados poderiam ter sido gerados automaticamente ou propostos a partir de uma análise *a priori* da especificação.

Apesar das conclusões, devem ser consideradas, criticamente, algumas questões que, se trabalhadas, abrirão espaço para novos trabalhos.

Uma destas questões, agora muito oportuna, seria a consideração de forma mais consistente da metodologia apresentada no capítulo 1. Neste sentido seria indispensável o desenvolvimento de um projeto completo que permitiria uma análise tanto global da viabilidade da metodologia, quanto de aspectos específicos, relativos à implementação, difíceis de serem equacionados sem a necessária experiência. Recentemente temos trabalhado no projeto e implementação (animação) de uma célula de manufatura flexível.

Ainda na linha de validação da metodologia proposta, certamente caberá a consideração de mais duas questões. Uma delas é avaliação do desempenho de LOTOS como linguagem de descrição utilizada na fase arquitetural. Quais são suas vantagens e desvantagens? Até que ponto, e em que condições, ela poderia ser substituída por uma outra técnica formal (p.ex: ESTELLE, SDL, VDM, Z, etc.)? Seria ela, de fato, uma técnica indispensável para descrever uma expressiva arquitetura do sistema? Além destas indagações, seria importante um detalhado estudo para se definir de que modo deverão ser utilizados os estilos de especificação e demais potencialidades LOTOS (p.ex.: qual deve ser o limite entre a especificação de aspectos do sistema na forma de estruturas abstratas de dados e na forma de expressões de comportamento; até que ponto seria útil a recomposição de processos na forma de um único processo (utilizando o Teorema da Expansão) com o fim de reduzir overheads) objetivando um maior desempenho das próximas etapas da metodologia de projeto e realização do sistema.

A outra questão seria relativa ao ambiente utilizado para o desenvolvimento da fase de realização. Até que ponto o domínio da aplicação deverá influenciar na escolha deste ambiente? Quais semelhanças são importantes existir entre o modelo utilizado na definição da linguagem usada na fase de realização, com o modelo em que se basea a técnica de descrição formal (p.ex: modularidade, para o caso LOTOS * STER)?

Uma outra linha de trabalho bastante interessante, seria considerar a possibilidade de derivação de realizações distribuídas. Neste sentido seria

muito útil repensar a proposta de mapeamento discutida neste trabalho à luz do ambiente STER distribuído [GUIMARÃES 90] e da proposta de implementação distribuída de especificações LOTOS [BOCHMANN 89].

# APÊNDICES

# APÊNDICE A - REGRAS DE ATUALIZAÇÃO

. *Notação:*

.B, B', B1, B1',... → Define expressões de comportamento;

.G → Denota o universo de portas que podem ser usadas na definição das expressões de comportamento (portas úteis);

.A → Um sub-conjunto qualquer de G;

.g,g1,...,gn → São portas definidas em G;

.i → Descreve uma ação não observável (interna);

.ACT → Conjunto de eventos definidos em G ∪ {i};

.$\mu$ → Um evento qualquer em ACT;

.$\delta$ → Pseudo evento que marca a terminação de um processo;

.$G^+$ → G ∪ {$\delta$};

.$g^+$ → Um evento qualquer em $G^+$;

.$ACT^+$ → ACT ∪ {$\delta$};

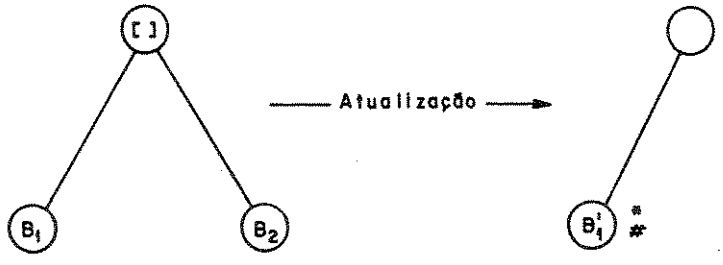
.$\mu^+$ → Um evento qualquer em $ACT^+$.

.* →Indica que após a terminação com sucesso do comportamento associado a uma sub-árvore, esta poderá ser podada da árvore.

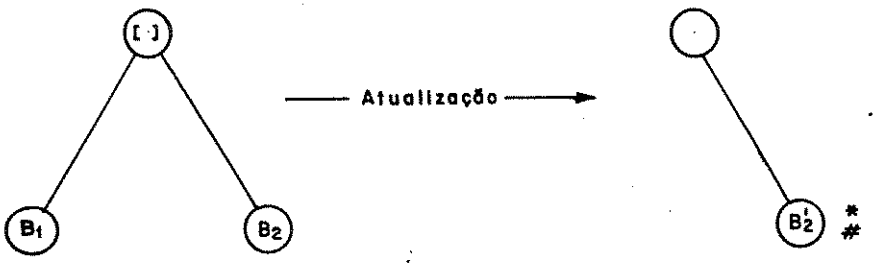
.# → Indica que um nó pode substituir seu nó ascendente.

ι) **Choice ( [ ] )**

ι) $$\frac{B1 \xrightarrow{\mu^+} B1'}{B1\ [\ ]\ B2 \xrightarrow{\mu^+} B1'}$$

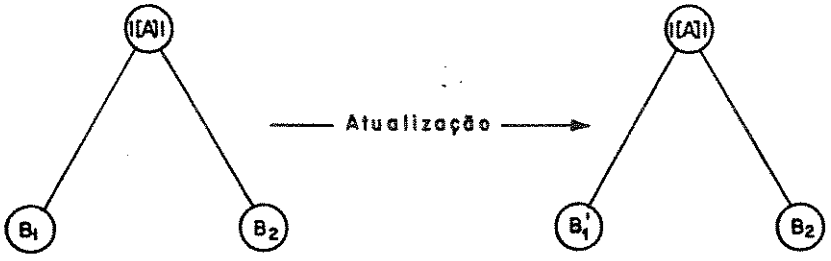


ιι) $$\frac{B_2 \xrightarrow{\mu^+} B_2'}{B_1 \; [\,] \; B_2 \xrightarrow{\mu^+} B_2'}$$

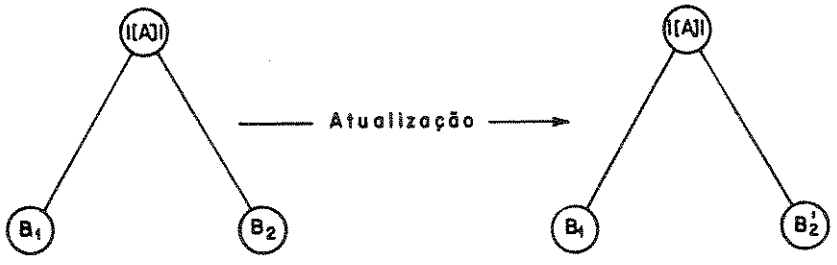



ιι) **Paralelismo** ( $|[A]|$ )

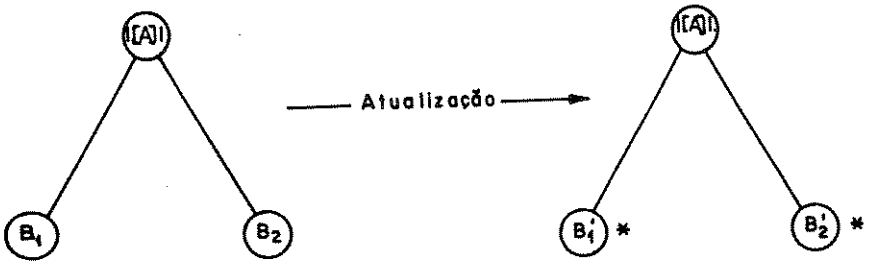
ι) $$\frac{B_1 \xrightarrow{\mu} B_1'}{B_1 |[A]| B_2 \xrightarrow{\mu} B_1' |[A]| B_2} \quad , \mu \notin A$$




ιι) $$\frac{B_2 \xrightarrow{\mu} B_2'}{B_1 |[A]| B_2 \xrightarrow{\mu} B_1 |[A]| B_2'} \quad , \mu \notin A$$

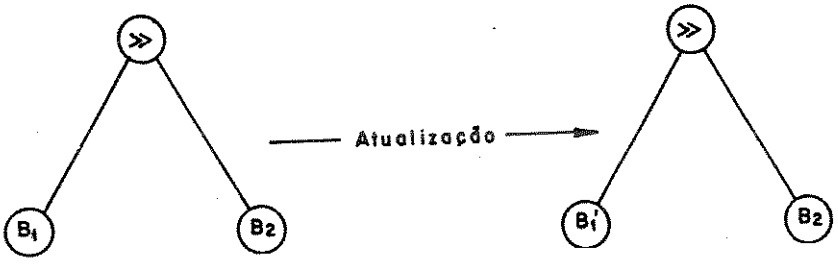


iii) $$\frac{B1 \xrightarrow{g^+} B1',\ B2 \xrightarrow{g^+} B2'}{B1 \,|[A]|\, B2 \xrightarrow{g^+} B1' \,|[A]|\, B2'},\ g^+ \in A \cup \{\delta\}$$

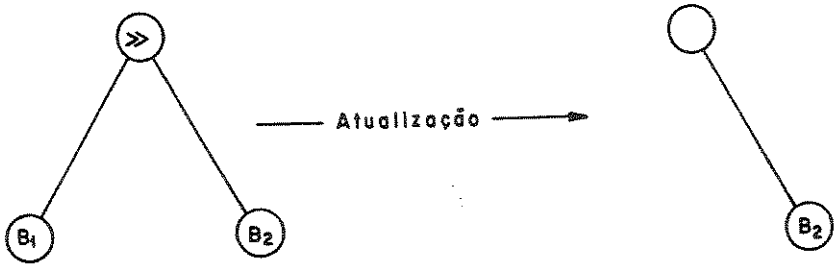



iii) **Composição Sequencial**

i) $$\frac{B1 \xrightarrow{\mu} B1'}{B1 >> B2 \xrightarrow{\mu} B1' >> B2}$$

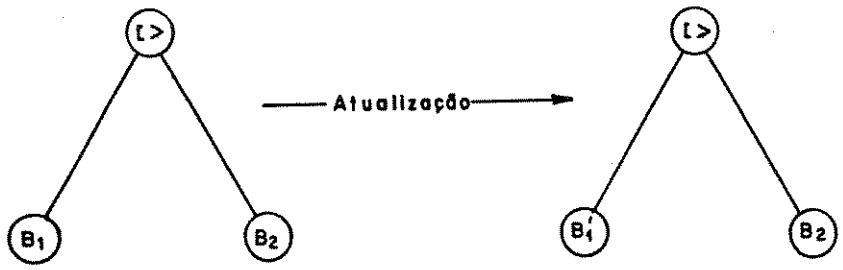



ii) $$\frac{B1 \xrightarrow{\delta} B1'}{B1 >> B2 \xrightarrow{i} B2}$$

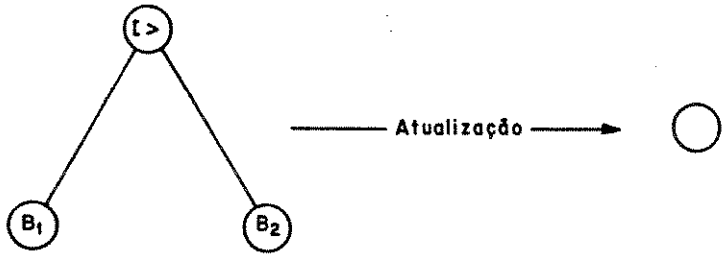


(v) **Desabilitação**

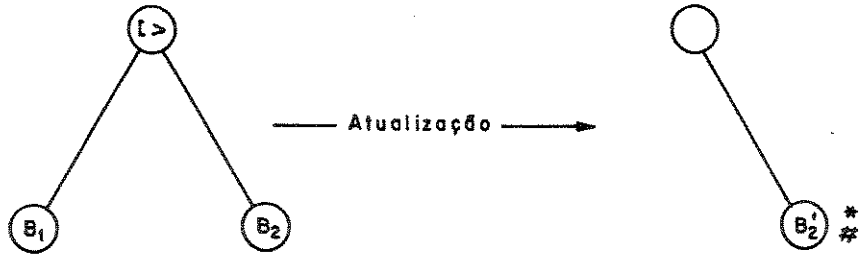
i) $$\frac{B_1 \xrightarrow{\ \mu\ } B_1'}{B_1 [> B_2 \xrightarrow{\ \mu\ } B_1' [> B_2}$$




ii) $$\frac{B_1 \xrightarrow{\ \delta\ } B_1'}{B_1 [> B_2 \xrightarrow{\ \delta\ } B_1'}$$




iii) $$\frac{B_2 \xrightarrow{\ \mu^+\ } B_2'}{B_1 [> B_2 \xrightarrow{\ \mu^+\ } B_2'}$$



# APÊNDICE B - APLICAÇÃO LSTER - SERVIÇO PROVEDOR SIMPLES

Os arquivos apresentados neste apêndice compoêm a aplicação STER relativa à especificação do Serviço Provedor descrito no exemplo do capítulo 6.

**.Arquivo de Definição**

```
DEFINE TipMes;
   TYPE
      MENS1  = RECORD
                  p: lstring(10);
                  gt: char;
                  ml: char;
                  vl: integer;
                  ti: lstring(5);
                  cn: boolean
               END;
      MENS2 = RECORD
                  g: char;
                  v: integer
               END;
      MENS3 = RECORD
                  pr: lstring(10);
                  pid: integer
               END;
      MENS4 = set of char;
      MENS5 = lstring(10);
      MENS6 = lstring(5);
      MENS7 = 0..9;
      MENS8 = boolean;
      MENS9 = RECORD
                  cd: mens7;
                  p1,p2,p3: mens5;
                  pt: mens4
               END;
      MENS10 = boolean;
END_DEFINE.
```

```
MODULE Spec;

    USE TipMes.inc;

    EXITPORT
        PS1: mens9;

    ENTRYPORT
        PE: mensi0;

    MESSAGE
        m9:mens9;
        mi0:mensi0;

    PROCEDURE par(pr1,pr2,pr3:mens5;prt:mens4);EXTERN;

    PROCEDURE pari(pr1,pr2,pr3:mens5);EXTERN;

    PROCEDURE Endy;EXTERN;

    BEGIN_MODULE
        LOOP
            Par('p0','pi','service',['d','g']);
            Pari('pi','start','station');
            Endy
        END_LOOP;
    END_MODULE.
```

```
MODULE Start;

USE TipMes.inc;

EXITPORT
    PS1: mens9;
    PS2: mens1;
    PS3: mens3;

ENTRYPORT
    PE1,PE2,PE: mens10;
    PE3: mens2;

MESSAGE
    m1: mens1;
    m2: mens2;
    m3: mens3;
    m9: mens9;
    m10: mens10;

CONST
    pr = 'start';

LABEL
    inic1,inic2;

PROCEDURE L1(g:char;v:integer;t:mens6);EXTERN;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE stop;EXTERN;

PROCEDURE EnbProc;EXTERN;

BEGIN_MODULE
    inic1:EnbProc;
    LOOP
        inic2:writeln('Start -> ok.');
        L1('g',1,'int');
        Sinc;
        writeln('Start: Sinc. com valor ',m2.v:1);
        Stop
    END_LOOP;
END_MODULE.
```

```
MODULE Service;

USE TipMes.inc;

EXITPORT
   PS1: mens9;
   PS2: mens1;
   PS3: mens3;

ENTRYPORT
   PE1,PE2,PE: mens10;
   PE3: mens2;

MESSAGE
   m1: mens1;
   m2: mens2;
   m3: mens3;
   m9: mens9;
   m10: mens10;

CONST
   pr = 'service';

LABEL
   inic1,inic2;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE par(pr1,pr2,pr3:mens5;prt:mens4);EXTERN;

PROCEDURE pari(pr1,pr2,pr3:mens5);EXTERN;

PROCEDURE EnbProc;EXTERN;

PROCEDURE Endb;

BEGIN_MODULE
   inic1:EnbProc;
   LOOP
      inic2:writeln('Service -> ok.');
      L2('g','int');
      Sinc;
      writeln('Service: Recebi start ',m2.v:1);
      Par('service','p2','crazyfifo',['a','b','c','e','f','h']);
      Pari('p2','sender','receiver');
      Endb
   END_LOOP;
END_MODULE.
```

```
MODULE Sender;

USE TipMes.inc;

EXITPORT
   PS1: mens9;
   PS2: mens1;
   PS3: mens3;

ENTRYPORT
   PE1,PE2,PE: mens10;
   PE3: mens2;

MESSAGE
   m1: mens1;
   m2: mens2;
   m3: mens3;
   m9: mens9;
   m10: mens10;

(* $INCLUDE: 'Dt.def' *)

CONST
   pr = 'sender';
   mens_inic = 1;

VAR
  mes: integer;
  cred: integer;
  new_cred: integer;
  expect: integer;

LABEL
   inic1,inic2;

PROCEDURE L1(g:char;v:integer;t:mens6);EXTERN;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE AutoExec;EXTERN;

PROCEDURE EnbProc;EXTERN;

BEGIN_MODULE
   inic1:EnbProc;
   mes:=succ(0);
   cred:=0;
   LOOP
      inic2: writeln('Sender -> ok.');
      IF gt(cred,0) THEN L1('a',mes,'int');
      IF eq(cred,0) THEN L2('c','int');
```

```
      L2('f','int');
      Sinc;
      CASE m2.g OF
          'a': BEGIN
                  writeln('Sender: Enviei: ',m2.v:1);
                  mes:=Succ(mes);
                  cred:=Pred(cred);
                  AutoExec
               END;
          'c': BEGIN
                  writeln('Sender: Recebi novo credito : ',m2.v:1);
                  new_cred:=m2.v;
                  cred:=new_cred;
                  AutoExec
               END;
          'f': BEGIN
                  writeln('Sender: Recebi pedido para
                           retransmitir dado: ',m2.v:1);
                  expect:=m2.v;
                  mes:=expect;
                  AutoExec
               END
      END
   END_LOOP;
END_MODULE.
```

```
MODULE Crazyfifo;

USE TipMes.inc;

EXITPORT
   PS1: mens9;
   PS2: mens1;
   PS3: mens3;

ENTRYPORT
   PE1,PE2,PE: mens10;
   PE3: mens2;

MESSAGE
   m1: mens1;
   m2: mens2;
   m3: mens3;
   m9: mens9;
   m10: mens10;

CONST
   pr = 'crazyfifo';

VAR
  mes: integer;
  cred: integer;
  expect: integer;

LABEL
   inic1,inic2;

PROCEDURE par(pr1,pr2,pr3:mens5;prt:mens4);EXTERN;

PROCEDURE EnbProc;EXTERN;

PROCEDURE Endb;EXTERN;

BEGIN_MODULE
   inic1:EnbProc;
   LOOP
   inic2:writeln('Crazyfifo -> ok.');
   Par('crazyfifo','fifo','lostdata',['p']);
   Endb
   END_LOOP;
END_MODULE.
```

```
MODULE Fifo;

USE TipMes.inc;

EXITPORT
   PS1: mens9;
   PS2: mens1;
   PS3: mens3;

ENTRYPORT
   PE1,PE2,PE: mens10;
   PE3: mens2;

MESSAGE
   m1: mens1;
   m2: mens2;
   m3: mens3;
   m9: mens9;
   m10: mens10;

(* $INCLUDE: 'Dt.def' *)

CONST
   pr = 'fifo';

VAR
  mes: integer;
  cred: integer;
  expect: integer;

LABEL
   inic1,inic2;

PROCEDURE L1(g:char;v:integer;t:mens6);EXTERN;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE AutoExec;EXTERN;

PROCEDURE EnbProc;EXTERN;

BEGIN_MODULE
   inic1:EnbProc;
   Niew(Q);
   LOOP
      inic2:writeln('Fifo -> ok.');
      L2('a','int');
```

```
      IF not(empty(Q))
      THEN
         BEGIN
            L1('b',first(Q),'int');
            L1('p',first(Q),'int')
         END;
      L2('h','int');
      L2('e','int');
      Sinc;
      CASE m2.g OF
          'a': BEGIN
                  writeln('Fifo: Recebi dado: ',m2.v:1);
                  mes:=m2.v;
                  add(mes,Q);
                  AutoExec
               END;
           'b': BEGIN
                  writeln('Fifo: Enviei dado : ',m2.v:1);
                  rem(Q);
                  AutoExec
               END;
           'p': BEGIN
                  writeln('Fifo: Dado perdido : ',m2.v:1);
                  rem(Q);
                  AutoExec
               END;
           'h': BEGIN
                  writeln('Fifo: Recebi pedido para
                           retransmitir dado: ',m2.v:1);
                  expect:=m2.v;
                  L1('f',expect,'int');
                  Sinc;
                  writeln('Fifo: Enviei pedido para
                           retransmitir dado: ',m2.v:1);
                  AutoExec
               END;
           'e': BEGIN
                  writeln('Fifo: Recebi novo credito: ',m2.v:1);
                  cred:=m2.v;
                  L1('c',cred,'int');
                  Sinc;
                  writeln('Fifo: Enviei novo credito: ',m2.v:1);
                  AutoExec
               END
      END
   END_LOOP;
END_MODULE.
```

```
MODULE Lostdata;

USE TipMes.inc;

EXITPORT
    PS1: mens9;
    PS2: mens1;
    PS3: mens3;

ENTRYPORT
    PE1,PE2,PE: mens10;
    PE3: mens2;

MESSAGE
    m1: mens1;
    m2: mens2;
    m3: mens3;
    m9: mens9;
    m10: mens10;

CONST
    pr = 'lostdata';

LABEL
    inic1,inic2;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE AutoExec;EXTERN;

PROCEDURE EnbProc;EXTERN;

BEGIN_MODULE
    inic1:EnbProc;
    LOOP
        inic2:writeln('Lostdata -> ok.');
        L2('p','int');
        Sinc;
        writeln('Lostdata: Recebi dado perdido ',m2.v:1);
        AutoExec;
    END_LOOP;
END_MODULE.
```

```
MODULE Receiver;

USE TipMes.inc;

EXITPORT
   PS1: mens9;
   PS2: mens1;
   PS3: mens3;

ENTRYPORT
   PE1,PE2,PE: mens10;
   PE3: mens2;

MESSAGE
   m1: mens1;
   m2: mens2;
   m3: mens3;
   m9: mens9;
   m10: mens10;

(* $INCLUDE: 'Dt.def' *)

CONST
   pr = 'receiver';
   mens_inic = 1;

VAR
  mes: integer;
  cred: integer;
  expect: integer;
  ref:integer;

LABEL
   inic1,inic2;

PROCEDURE L1(g:char;v:integer;t:mens6);EXTERN;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE AutoExec;EXTERN;

PROCEDURE EnbProc;EXTERN;

BEGIN_MODULE
   inic1:EnbProc;
   cred:=Succ(Succ(Succ(Succ(Succ(0)))));
   expect:=Succ(0);
   LOOP
      inic2: writeln('Receiver -> ok.');
      L2('b','int');
      L1('e',cred,'int');
      Sinc;
```

```
    CASE m2.g OF
        'b': BEGIN
                writeln('Receiver: Recebi dado: ',m2.v:1);
                mes:=m2.v;
                IF eq(mes,expect) THEN L1('d',mes,'int');
                IF gt(mes,expect) THEN L1('h',expect,'int');
                Sinc;
                CASE m2.g OF
                    'd': BEGIN
                            writeln('Receiver: Enviei dado: ',m2.v:1);
                            expect:=Succ(expect);
                            AutoExec
                         END;
                    'h': BEGIN
                            writeln('Receiver: Enviei solicitacao de
                                     retransmisao do dado: ',m2.v:1);
                            AutoExec
                         END
                END
             END;
        'e': BEGIN
                writeln('Receiver: Enviei novo credito : ',m2.v:1);
                AutoExec
             END
    END
  END_LOOP;
END_MODULE.
```

```
MODULE Station;

USE TipMes.inc;

EXITPORT
    PS1: mens9;
    PS2: mens1;
    PS3: mens3;

ENTRYPORT
    PE1,PE2,PE: mens10;
    PE3: mens2;

MESSAGE
    m1: mens1;
    m2: mens2;
    m3: mens3;
    m9: mens9;
    m10: mens10;

CONST
    pr = 'station';

LABEL
    inic1,inic2;

PROCEDURE L2(g:char;t:mens6);EXTERN;

PROCEDURE Sinc;EXTERN;

PROCEDURE AutoExec;EXTERN;

PROCEDURE EnbProc;EXTERN;

BEGIN_MODULE
    inic1:EnbProc;
    LOOP
        inic2: writeln('Station -> ok.');
        L2('d','int');
        Sinc;
        writeln('Station: Recebi: ',m2.v:1);
        AutoExec;
    END_LOOP;
END_MODULE.
```

**.Arquivo de Configuração**

```
CONFIGURATION Servico;

    INSTANCE
       s1:spec;
       p1:start;
       p2:service;
       p3:sender;
       p4:crazyfif;
       p5:receiver;
       p6:station;
       p7:fifo;
       p8:lostdata;
       b1:b;

    CREATE
       s1/S=64/H=512,
       p1/S=64/H=64,
       p2/S=64/H=64,
       p3/S=64/H=512,
       p4/S=128/H=512,
       p5/S=64/H=512,
       p6/S=64/H=512,
       p7/S=64/H=512,
       p8/S=64/H=512,
       b1/S=2500/H=2000;

    LINK
       s1.PS1 TO b1.PE1;
       p1.PS1 TO b1.PE1;
       p2.PS1 TO b1.PE1;
       p3.PS1 TO b1.PE1;
       p4.PS1 TO b1.PE1;
       p5.PS1 TO b1.PE1;
       p6.PS1 TO b1.PE1;
       p7.PS1 TO b1.PE1;
       p8.PS1 TO b1.PE1;
       p1.PS2 TO b1.PE2;
       p2.PS2 TO b1.PE2;
       p3.PS2 TO b1.PE2;
       p4.PS2 TO b1.PE2;
       p5.PS2 TO b1.PE2;
       p6.PS2 TO b1.PE2;
       p7.PS2 TO b1.PE2;
       p8.PS2 TO b1.PE2;
       p1.PS3 TO b1.PE3;
       p2.PS3 TO b1.PE3;
       p3.PS3 TO b1.PE3;
       p4.PS3 TO b1.PE3;
       p5.PS3 TO b1.PE3;
       p6.PS3 TO b1.PE3;
       p7.PS3 TO b1.PE3;
       p8.PS3 TO b1.PE3;
       b1.PS1[1] TO p1.PE1;
       b1.PS1[2] TO p2.PE1;
       b1.PS1[3] TO p3.PE1;
       b1.PS1[4] TO p4.PE1;
       b1.PS1[5] TO p5.PE1;
       b1.PS1[6] TO p6.PE1;
       b1.PS1[7] TO p7.PE1;
       b1.PS1[8] TO p8.PE1;

       b1.PS2[1] TO p1.PE2;
       b1.PS2[2] TO p2.PE2;
       b1.PS2[3] TO p3.PE2;
       b1.PS2[4] TO p4.PE2;
       b1.PS2[5] TO p5.PE2;
       b1.PS2[6] TO p6.PE2;
       b1.PS2[7] TO p7.PE2;
       b1.PS2[8] TO p8.PE2;
       b1.PS3[1] TO p1.PE3;
       b1.PS3[2] TO p2.PE3;
       b1.PS3[3] TO p3.PE3;
       b1.PS3[4] TO p4.PE3;
       b1.PS3[5] TO p5.PE3;
       b1.PS3[6] TO p6.PE3;
       b1.PS3[7] TO p7.PE3;
       b1.PS3[8] TO p8.PE3;

                              END_CONFIG.
```

**.Estrutura de Dados** (arquivo Dt.def)

```
CONST
      Error = 0;

TYPE  Queue = ^item;
      item = RECORD
                d: integer;
                nd : Queue;
             END;

VAR   Q: Queue;
      i: integer;

PROCEDURE rem(var q:Queue);
   BEGIN
      IF q <> nil
      THEN
         BEGIN
            q:= q^.nd;
            dispose(q)
         END
   END;

FUNCTION first(q:Queue): integer;
   BEGIN
      IF q = nil
      THEN writeln(Error)
      ELSE first:= q^.d
   END;

FUNCTION empty(q:Queue): boolean;
   BEGIN
      IF q = nil
      THEN empty:= true
      ELSE empty:= false
   END;
```

```
PROCEDURE add(m:integer;var q:Queue);
   VAR
      p:Queue;

   BEGIN
      IF q <> nil
      THEN
         BEGIN
            p:=q;
            WHILE p^.nd <> nil DO p:=p^.nd;
            new(p^.nd);
            WITH p^.nd^ DO
               BEGIN
                  d:=m;
                  nd:=nil
               END
         END
      ELSE
         BEGIN
            new(q);
            q^.d:=m;
            q^.nd:= nil
          END
   END;

   FUNCTION gt(v1,v2:integer):boolean;
      BEGIN
         IF v1 > v2
         THEN gt:=true
         ELSE gt:=false
      END;

   FUNCTION eq(v1,v2:integer):boolean;
      BEGIN
         IF v1 = v2
         THEN eq:=true
         ELSE eq:=false
      END;

   PROCEDURE Niew(var q:Queue);
      BEGIN
         q:=nil
      END;
```

# APÊNDICE C - ESPECIFICAÇÃO LOTOS - SERVIÇO PROVEDORSIMPLES

```
specification protocol: noexit

 type       boolean is
 sorts      bool
 opns       true,false              : -> bool
            not                     : bool -> bool

 eqns       forall x,y:bool

            ofsort bool

            not(true)                    = false                    ;
            not(false)                   = true                     ;

 endtype

 type       int1 is boolean
 sorts      int
 opns       zero                    : -> int
            pred,succ               : int -> int
            gt,eq                   : int,int -> bool
            if_then_else_int        : bool,int,int -> int

 eqns       forall m,n:int

            ofsort int
            succ(pred(m))                = m                        ;
            pred(succ(m))                = m                        ;

            eq(n,n)                      = true                     ;
            eq(zero,succ(n))             = false                    ;
            eq(succ(m),succ(n))          = eq(m,n)                  ;

            gt(zero,zero)                = false                    ;
            gt(succ(m),zero)             = true                     ;
            gt(succ(m),succ(n))          = gt(m,n)                  ;

            if_then_else_int(true,m,n)   = m                        ;
            if_then_else_int(false,m,n)  = n                        ;

endtype
```

```
type        buffer is int1,boolean,int
sorts       queue
opns        new                     : -> queue
            error                   : -> int
            add                     : int,queue -> queue
            rem                     : queue -> queue
            first                   : queue -> int
            empty                   : queue -> bool
            if_then_else            : bool,queue,queue -> queue

eqns        forall ms:int,Q,Q1,Q2:queue

            ofsort queue

            rem(new)                        = new                       ;
            rem(add(ms,Q))                  = if_then_else
                                              (empty(Q),new,
                                               add(ms,rem(Q)))          ;

            first(new)                      = error                     ;
            first(add(ms,Q))                = if_then_else_int
                                              (empty(Q),m,first(Q))     ;

            empty(new)                      = true                      ;
            empty(add(ms,Q))                 = false                    ;

            if_then_else(true,Q1,Q2)        = Q1                        ;
            if_then_else(false,Q1,Q2)       = Q2                        ;
 endtype

behavior
 hide g,d in
 (
  start[g](succ(zero))
  |||
  station[d]
 )
  |[g,d]|
  service[g,d]

where
  process start[inic](mes_inic:int):noexit :=
     inic!mes_inic;stop
  endproc
```

```
process service[go,deliv]:noexit :=
   go?mes_inic:int;
   ( hide a,b,c,e,f,h in
     (
      sender[a,c,f](mes_inic,zero)
      |||
      receiver[b,e,h,deliv](succ(zero))
     )
      |[a,b,c,e,f,h]|
      crazyfifo[a,b,c,e,f,h](new)
   )
where
  process sender[out,req,retry](mes:int,cred:int):noexit :=
     [gt(cred,zero)] -> out!mes;
                        sender[out,req,retry](succ(mes),pred(cred))
     []
     [eq(cred,zero)] -> req?new_cred:int;
                        sender[out,req,retry](mes,new_cred)
     []
     retry?expect:int;sender[out,req,retry](expect,cred)
  endproc
  process crazyfifo[mes_in,mes_out,cred_in,cred_out,retry_in,
                    retry_out](Q:queue):noexit :=
     hide p in
      fifo[mes_in,mes_out,cred_in,cred_out,retry_in,retry_out,p](new)
      |[p]|
      lostdata[p]
  where
   process fifo[mes_in,mes_out,cred_in,cred_out,retry_in,
                retry_out,lst](Q:queue):noexit:=
      mes_in?mes:int;
        fifo[mes_in,mes_out,cred_in,cred_out,retry_in,
             retry_out,lst](add(mes,Q))
      []
      [not(empty(Q))] -> (   mes_out!first(Q);
                             fifo[mes_in,mes_out,cred_in,cred_out,
                                  retry_in,retry_out,lst](rem(Q))
                          []
                             lst!first(Q);
                               fifo[mes_in,mes_out,cred_in,cred_out,
                                    retry_in,retry_out,lst](rem(Q))
                          )
      []
      retry_in?expect:int;retry_out!expect;
        fifo[mes_in,mes_out,cred_in,cred_out,retry_in,
             retry_out,lst](Q)
      []
      cred_in?cred:int;cred_out!cred;
        fifo[mes_in,mes_out,cred_in,cred_out,retry_in,
             retry_out,lst](Q)
   endproc
```

```
    process lostdata[lst]:noexit :=
       lst?data:int;lostdata[lst]
    endproc
  endproc
  process receiver[mes_in,grant,retry,deliv](expect:int):noexit :=
     mes_in?mes:int;
     (  [eq(ms,expect)] -> deliv!mes;
                              receiver[mes_in,grant,retry,deliv]
                                      (succ(expect))
        []
        [gt(ms,expect)] -> retry!expect;
                               receiver[mes_in,grant,retry,deliv]
                                       (expect)
     )
     []
     grant!succ(succ(succ(succ(succ(zero)))));
        receiver[mes_in,grant,retry,deliv](expect)
  endproc
endproc
process station[deliv]:noexit :=
   deliv?mes:int;station[deliv]
endproc
endspec
```

# Referências Bibliográficas

# REFERÊNCIAS BIBLIOGRÁFICAS


[BARRÉ 89]

BARRÉ, J. et al *Spécification d'Application Réparties en LOTOS: Etude du Cas du Service TP*, Anais do Seminário Franco-Brasileiro em Sistemas Informáticos Distribuídos, 11-14 Setembro, 1989, Florianópolis-SC, Brasil, P. 56 - 63.

[BIEMANS 86]

BIEMANS, F., BLONK, P. *On the Formal Specification and Verification of CIM Architeture using LOTOS*, Computers in Industry, V.7, P. 491 - 504, 1986.

[BOCHMANN 89]

BOCHMANN, G.et al *On the Distributed Implementation of LOTOS*, Proceedings of the 2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'89), 5 - 8 December,1989, Vancouver, Canada, P 175 - 194.

[BOGAARDS 88]

BOGGARDS, K. *LOTOS suported system development*, In: Turner, K. J. (ed.), Formal Description Techniques 88, P.279-294, North-Holland, 1989.

[BOLOGNESI 87]

BOLOGNESI, T., BRINKSMAN, E. *Introduction to the ISO Specitfication Language LOTOS*, Computer Networks and ISDN Systems, N. 14, P. 25-59, 1987.

[BRIAND 87]

BRIAND, J. P. et al *Execution LOTOS Specifications*, In: Sarikaya, B., Bochmann, G. (eds.), Proceedings of IFIP wokshop on Protocol Specification, Testing,

and Verification VI P 73 - 85, North-Holland, 1987.

[BRINKSMAN 85]

BRINKSMAN, E., KARJOTH, G. *Specification of the OSI Transport Service in LOTOS*, In: Yemini, Y., Strom, R., Yemini, S., Proceedings of IFIP workshop Protocol Specification, Testing, and Verification IV, P. 227 - 251, North-Holland, 1985.

[BRINKSMAN 86]

BRINKSMAN, E. *A Tutorial on LOTOS*, In: Diaz M. (ed.), Porceedings of IFIP workshop on Protocol Specification,Testing, and Verification V, P. 171-194, North-Holland, 1986.

[BRINKSMAN 87]

BRINHSMAN, E. et al *Experience with and Future of LOTOS as a Specification Language*, In: SARACCO, R., TILANUS, P. (eds.), SDL '87: State of the Art and Future Trends, Proceedings of 3rd SDL Forum, P. 439 - 450, North- Holland, 1987.

[BUDKOWSKI 87]

BUDKOWSKI, S., DEMBINSK, P. *An Introduction to ESTELLEl: A Specification Language for Distributed Systems*, Computer Networks and ISDN System, V.14, P.3-23, 1987.

[CARCHIOLO 86]

CARCHIOLO, V. et al *A LOTOS Specification of the PROWAY Highway Service*, IEEE Trans. Computers, V. C35, N. 11, P.949 - 968, November, 1986.

[CCITT Z100]

CCITT Z100 *Specification and Description Language*, 1988.

[COELLO 86]

COELLO, J. M. A. *Suporte de Tempo Real para um Ambiente de Programação Concorrente*, Dissertação de Mestrado em Engenharia Elétrica, Universidade Estadual de Campinas, Campinas - SP, 1986.

[COHEN 86]

COHEN, B. et al *The Specification of Complex Systems*, Addison-Wesley, 1986.

[DTIA 88]

DTIA-001/88 *STER - Um ambiente para desenvolvimento de software tempo-real*, IA/DEI/CTI, Campinas - SP, 1988.

[EHRIG 85]

EHRIG, H., MAHR, B. *Fundamental of Algebric Specification 1*, Springer-Verlag, Berlim, 1985.

[EIJK 88]

van EIJK, P. H. J. *Software tools for the Specification Language LOTOS*, Ph. D. Thesis, University of Twente, January, 1988.

[EIJK 90]

van EIJK, P. H. J *On the use of specification styles for autometed protocol implementation from LOTOS to C*, Proceedings of IFIP 10th. International Symposion on Protocol Specification, Testing, and Verification, 13-15 Junho, 1990, Château Laurier-Ottawa, Canada, P.153-164.

[FERNANDEZ 88]

FERNANDEZ, A. et al *PRODAT - The Derivation of an Implementation from its LOTOS Formal Specification*, In: Aggarwal, S., Sabani, K. (eds.) Proceedings of IFIP wokshop on Protocol Specification, Testing,

and Verification VII, P. 411 - 419, North-Holland, 1988.

[GILBERT 87]

GILBERT, D. *Executable LOTOS: Using PARLOG to implement an FDT*, In: Rudin, H., West, C.H. (eds.), Proceedings of IFIP workshop on Protocol Specification, Testing, and Verification VII, P. 281 - 294, North-Holland, 1987.

[GUIMARÃES 90]

GUIMARÃES, E. G. *A Distribuição do Ambiente para Desenvolvimento de Software Tempo Real - STER*, Dissertação de Mestrado em Engenharia Elétrica, Universidade Estadual de Campinas, Campinas-SP, 1990.

[GUTTAG 86]

GUTTAG, J. *Notes on the type abstration*, In: Gehami, N., McGettrick (eds.), Software Specification Techniques, P.55-74, Addison-Wesley, 1986.

[HAEBERER 89]

HAEBERER, A. M. et al *Formalización del Proceso de Desarrollo de Software*, IV EBAI, 16-19 Janeiro, 1989, Santiago del Estero, Argentina.

[HALL 90]

HALL, A. *Seven Miths of Formal Methods*, IEEE Software, V.7, N.5, P.37-50, September, 1990.

[HOARE 85]

HOARE, C. A. R. *Comunnicating Sequencial Process*, Prentice-Hall, 1985.

[HULZEN 89]

van HULZEN, W. H. P. et al *LOTOS extend with clocks*

Proceedings of the 2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE '89), 5 - 8 December,1989, Vancouver, Canada, P 157 - 174.

[HWANG 85]

HWANG, K., BRIGGS, F. A. *Computer Architecture and Parallel Processing*, McGraw-Hill, 1985.

[ISO 84]

ISO/TC97/SC16/WG1/N299 *Definition of the Temporal Ordering Specification Language LOTOS*, 1985.

[ISO 88a]

ISO/IEC JTC1/SC21/N2426 *Formal Description of the basic connection-oriented Session service in LOTOS*, Draft Technical Report DTR 9571, February, 1988.

[ISO 88b]

ISO/IEC JTC1/SC6/N4870 *Formal Description of ISO 8072 in LOTOS*, March, 1988.

[ISO 88c]

ISO/IEC JTC1/SC6/N4870 *Formal Description of ISO 8073 in LOTOS*, March, 1988.

[ISO 89]

ISO TR 9572 *Information Processing System - Open Systems Interconnection - Formal Description in LOTOS of the connection-oriented Session Protocol*, 1989.

[ISO 90]

ISO TR 10024 *Information Processing System - Open Systems Interconnection - Formal Description in LOTOS of the connection-oriented Transport Protocol*, 1990.

[ISO 8807]

International Standard ISO 8807 *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Technique Based on the Temporal Ordering of Observational Behavior*, 1988.

[ISO 9074]

ISO DIS 9074 *Information Processing Systems - Open Systems Interconnection - ESTELLE - A Formal Description Technique Based on the Extended State Transition Model*, 1988.

[KEMMERER 90]

KEMMERER, R. A. *Integrating Formal Methods into the Development Process*, IEEE Software, V.7, N.5, P.37-50, September, 1990.

[KRAMER 83]

KRAMER, J. et al *CONIC: An Integrated Approach to Distributed Computer Control Systems*, IEEE Proceedings V.130, N.1, P.1-10, January, 1983.

[KRAMER 84a]

KRAMER, J. et al *The CONIC Programing Language Version 2.4*, Research Report Doc 84/19, Department of Computing, Imperial College, London, October, 1984.

[KRAMER 84b]

KRAMER, J. et al *The CONIC Configurating Language Version 2.3*, Research Report Doc 84/20, Department of Computing, Imperial College, London, November, 1984.

[LEDGARD 76]

LEDGARD, H. F., TAYLOR, R. W. *Two views of Data Abstration*, Communication of the ACM, V.20, N.6, P.

382 - 384, June, 1989.

[LOGRIPPO 88]

LOGRIPPO, L. *An Interpreter for LOTOS, A Specification Language for Distributed Systems*, Software - Practice and Experience, V.18, N.4, P. 365 - 385, April, 1988.

[LOPES 86]

LOPES, A. B. *LPM e LCM: Linguagens para Programação e Configuração de Aplicações de Tempo-Real*, Dissertação de Mestrado em Sistemas e Computação, Universidade Federal da Paraíba, Agosto, 1986.

[MAGALHÃES 86]

MAGALHÃES, M. F. *Software para Tempo Real*, I EBAI, 17 Fevereiro - 01 Março, 1986, Campinas-Sp, Brasil.

[MENDES 88]

MENDES, S. B. T., AGUIAR,T. S. *Métodos para Especificação de Sistemas*, II EBAI, 25 Janeiro - 07 Fevereiro, 1988, Curitiba-PR, Brasil.

[MEYER 85]

MEYER, B. *On Formalism in Specifications*, IEEE Software, V.2, N.1, P.6-26, January,1985.

[MIGUEL 87]

de MIGUEL, T., MAÑAS, J. A. *An Implementation Architecture for LOTOS*, Proceedings of IBERICON '87: 1st. Iberian Conference on data Communications, 19-21 Maio, 1987, Lisbon, Portugal, V.1, P.43-62.

[MILNER 80]

MILNER, R. *A Calculus of Comunnicating Systems*, Lectures

Notes in Computer Science, V.92, Springer-Verlag, 1980.

[NOMURA 90]

NOMURA, S. et al *A LOTOS Compiler and Process Synchronization Mananger*, Proceedings of IFIP 10th. International Symposium on Protocol Specification, Testing, and Verification, 13-15 Junho, 1990, Château Laurier-Ottawa, Canada, P.165-184.

[OBAID 86]

OBAID, A. *SINAPS: A Simulator of communicating systems*, University oof Ottawa, Department of Computer Science, Technical Report 86-14, August, 1986.

[PRESSMAN 87]

PRESSMAN, R. S. *Software Engineering: A Practioner's Approach*, Second Edition, McGRAW-HILL, 1987.

[PUENTE 86]

de la PUENTE, J. A. et al *Formal Specification of Real-Time Software Systems, An Industrial Example,* Proceedings of IFAC Real Time Programming, P. 113 - 121, Hungary, 1986.

[QUEMADA 86]

QUEMADA, J. *Data Link Service LOTOS Specification*, SEDOS/C1/6 & 7/M, December, 1986.

[QUEMADA 89]

QUEMADA, J. et al *A Timed Calculus for LOTOS* Proceedings of the 2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'89), 5 - 8 December, 1989, Vancouver, Canada, P ?.

[SCOLLO 85a]

SCOLLO, G. e MINISSALE, F. *On the Specification in LOTOS of OSI protocols*, In Bucc G.e Valle G. (eds.), Computing '85, Proceedings of 8th ACM European Conference ICS'85, P.197-206,North-Holland, 1985.

[SCOLLO 85b]

SCOLLO, G. et al *Specification and Implementation of MODAM System*, In: Yemini, Y., Strom, R., Yemini S. (eds.), Proceedings workshop on Protocol Specification, Testing, and Verification IV, P. 385-422, North-Holland, 1985.

[TOCHER 86]

TOCHER, A. *OSI Transport Service: A contraint-oriented specification in LOTOS*, ESPRIT/SEDOS/C1/WP/25/1K, ICL, Kidsrove, England, August, 1986.

[TURNER 86]

TURNER, K. J. *OSI connection-Oriented network service: a connection-oriented specification in extended LOTOS (Draft 4)*, SEDOS/C1/WP/15/1K, ICL Kidsgrove, England, May, 1986.

[TURNER 89]

TURNER, K. J. *A LOTOS-Based Development Strategy*, Proceedings of the 2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'89), 5 - 8 December,1989, Vancouver, Canada, P 157 - 174.

[VISSERS 83]

VISSERS, C. A. et al *Formal Description Technique*, Proceedings fo the IEEE, V.71, N.12, P.1356-1364, December, 1983.

[VISSERS 87a]

VISSER C. A *Tre* *and Proliferations of Formal* *D* *ription Tech* *s for OSI Standards*, Anais do 5 Simpósi Br o de Redes de Computadores, 1 5 Abril, 198 Paulo-SP, Brasil, P.?.

[VISSERS 87b]

VISSERS, C. A. *Formal Specificationin in OSI*, In: Muller, G., , R. (eds.), Networks in Open Systems, LNCS 248, P.338-359, Springer-Verlag, Berlin, 1987.

[VISSERS 88]

VISSERS, C. A. et al *Architecture and Specification Style in Formal Description of Distributed Systems*, In: Aggarwal, S., Sabinani, K. (eds.), Proceedings of IFIP workshop on Protocol Specification, Testing, and Verification VIII, North-Holland, 1988.

[WALKER 88]

WALKER, D. *Introduction to a Calculus of Comunnicating Systems*, ECS-LFCS-87-22, University of Edinburg, 1988.

[WING 90]

WING, J. *A Specifier's Introduction to Formal Methods*, IEEE Computers, V.23, N.9, P.8-24, September, 1990.

[WIRTH 76]

WIRTH, N. *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

[WU 89]

WU, C., BOCHMANN, G. *An Execution Model for LOTOS Specifications*, Université de Montréal, Department d'informatique et de recherche opérationnelle, Publication # 701, October, 1989.