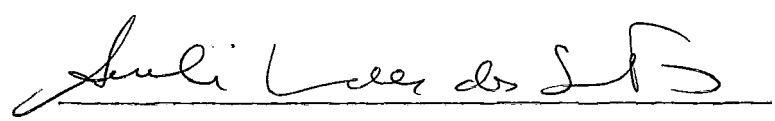
MECANISMOS DE TRANSMISSÃO DE MENSAGENS EM SISTEMAS DISTRIBUIDOS: UM MODELO SEMÂNTICO

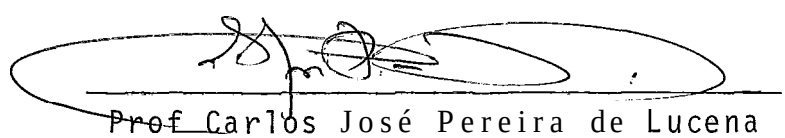
Sheila Regina Murgel Veloso

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

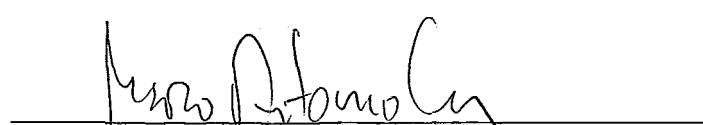
Aprovada por:



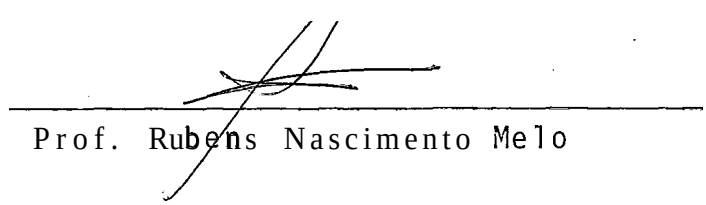
Profa. Sueli Mendes dos Santos
Presidente



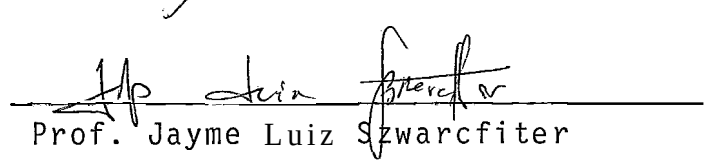
Prof Carlos José Pereira de Lucena



Prof. Marco Anton o Casanova



Prof. Rubens Nascimento Melo



Prof. Jayme Luiz Szwarcfiter

Rio de Janeiro, RJ - BRASIL

Outubro de 1985

VELOSO, SHEILA REGINA MURGEL

Mecanismos de Transmissão de Mensagens em Sistemas Distribuidos: Um Modelo Semântico (Rio de Janeiro) 1985.

VI , 167p. 29,7cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas, 1985).

Tese - Universidade Federal do Rio de Janeiro, COPPE.

1. Redes de Computadores, Semântica. I. COPPE/UFRJ II. Título (série).

Aos meus pais,
ao Paulo Augusto,
à Paula e Flávia.

AGRADECIMENTOS

A Profa. Sueli, não só por sua segura e prestimosa orientação acadêmica, como também por sua paciente convivência com minha desorganização e incertezas; aos Profs. Lucena, Maibaum e Jayme, pela colaboração ao longo do doutorado; ao Prof. Casanova, por suas inestimáveis sugestões; ao Prof. Rubens, por sua confiança; ao Ademar, pelos desenhos; à Ruth, por seu excelente trabalho de datilografia e dedicação; à Sueli e Denise, pela solidariedade e dedicação; ao Felipe e Nelson por seu desprendimento; aos amigos do departamento de Informática da PUC, em particular ao Raul, pela solidariedade e boa vontade; aos meus colegas de departamento e do Instituto de Matemática, especialmente aos Profs. Luiz Adauto, Marién e Jovana, pelo apoio recebido; aos meus pais, pelo estímulo, apoio e compreensão sempre presentes; à Paula e Flávia, pela (nem sempre paciente) espera pelo término da tese; à Heloísa pela grande ajuda no momente certo; à Lia pela colaboração no "leva e traz"; à Zilda pela valiosa ajuda no campo doméstico e particularmente ao Paulo Augusto, que participou de todas etapas desse trabalho, pela paciência, dedicação e estímulo, sem os quais, certamente, essa tese não teria sido possível.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

MECANISMOS DE TRANSMISSÃO DE MENSAGENS EM SISTEMAS DISTRIBUÍDOS: UM MODELO SEMÂNTICO

Sheila Regina Murgel Veloso

Setembro 1985

Orientador: Sueli Mendes dos Santos

Programa : Engenharia de Sistemas e Computação

Nesse trabalho apresentamos mecanismos para transmissão de mensagens em ambiente distribuído, enfatizando seu uso disciplinado, e um modelo operacional para sua semântica. Estes mecanismos permitem também criação e ligação dinâmicas de processos. Conceitos como confiabilidade, modularidade e flexibilidade, úteis em programação paralela e distribuída, nortearam a concepção desses mecanismos. Na semântica operacional incorporamos aspectos realistas tais como velocidade variável e desconhecida da transmissão de mensagens pela rede e o não determinismo do processamento paralelo. A semântica operacional modela uma modalidade de lógica temporal, que é usada para expressar propriedades dos mecanismos num nível mais abstrato.

Abstract of Thesis presented to COPPE/UFRJ as partial fultillment of the requirements for the degree of Doctor of Science (D.Sc.)

MECANISMOS DE TRANSMISSÃO DE MENSAGENS EM SISTEMAS DISTRIBUÍDOS: UM MODELO SEMÂNTICO

Sheila Regina Murgel Veloso

September 1985

Chairman: Sueli Mendes dos Santos

Department: Engenharia de Sistemas e Computação

In this work we present a mechanisms for message transmission in a distributed environment, emphasizing their disciplined usage, and an operational model for their semantics. These mechanisms also permit dynamic creation and interconnection of processes. Concepts such as reliability, modularity and flexibility, useful in parallel and distributed programming,, were instrumental in the canception of these mechanisms. The operational semantics embodies realistic aspects such as variable and unknown rate of message transmission through the network and non determinism in parallel processing. The operational semantics models a temporal logic modality which is used to express properties of the mechanisms on a more abstract level.

ÍNDICE

CAPÍTULO I

INTRODUÇÃO

Atualmente uma das áreas de pesquisa em teoria de linguagens de programação que tem despertado maior interesse é a de programas concorrentes, especialmente no que diz respeito a seu projeto, definição, análise e verificação.

Por muito tempo o enfoque das pesquisas em programação concorrente foi dirigido para aplicações em sistemas convencionais com controle centralizado. Estes sistemas forçavam uma sequencialização de ações que em muitas aplicações era pouco natural.

Devido ao recente desenvolvimento da tecnologia em microprocessadores, e consequentemente barateamento do custo de "hardware" e da tecnologia de transmissão de dados,tornou-se viável a utilização de sistemas distribuídos em aplicações para as quais essa solução se adequa mais elegante e naturalmente. Daí então a tendência natural de se utilizar linguagens que suportem atividades distribuídas e comunicação via mensagens ao invés de linguagens em que as atividades concorrem pelo uso de um mesmo recurso. Assim, torna-se muito importante para a definição, análise e confiabilidade dos programas usando essas linguagens, que os mecanismos que permitem a comunicação e sincronização de processos através de troca de mensagens sejam clara e precisamente compreendidos. (Esses conceitos são definidos no capítulo 2).

A importância da formalização da especificação de linguagens pode ser justificada de várias maneiras dependendo do

objetivo a ser alcançado:

1. Implementação - O construtor de um compilador ou interpretador para a linguagem precisa captar todos os aspectos da linguagem detalhadamente. Se a linguagem estiver definida de maneira ambígua, diferentes implementadores poderão produzir processadores incompatíveis uns com os outros.

2. Correção da implementação - Para se provar a correção de um programa é necessário que seu comportamento funcional seja precisamente formalizado. A implementação de uma linguagem nada mais é que um programa cujo comportamento funcional é (parcialmente) dado pela especificação da linguagem por ele implementado. Assim uma formalização das especificações é uma condição necessária para a prova da correção da especificação.

3. Automatização da implementação - Aespecificação formal introduz a possibilidade de automatização (mesmo que parcial) do processo de construir compiladores ou interpretadores. Esta em geral consiste de programas que tem como entrada as especificações da linguagem e dão como saída processadores (ou partes de processadores) para a linguagem.

4. Prova de correção - A inadequação da depuração de erros por meio de teste para programas concorrentes dirige a verificação de algumas propriedades do programa por meio de provas racionais. Se essas provas pretendem ter um rigor matemático é necessário que as propriedades dos mecanismos linguísticos da linguagem às quais elas (provas) se referem sejam rigorosamente formalizadas. Por outro lado, a definição for

mal de uma linguagem pode sugerir uma metodologia para construção de programas que, por construção, obedecem a suas especificações. Assim, a prova de correção do programa integra a sua própria construção.

5. Melhor projeto de linguagem - O tratamento formal de estruturas linguísticas permite-nos penetrar na natureza fundamental das linguagens de programação. Desta forma, ficam mais visíveis algumas particularidades da linguagem que poderiam passar desapercebidas num tratamento informal, tais como semelhanças entre linguagens aparentemente diferentes e diferenças entre linguagens aparentemente semelhantes. Mais ainda, a formalização permite isolar as razões que explicam por que certas combinações de mecanismos levam a interações complicadas ou inconsistentes, ou por que a presença de algumas caracteristicas em programas dificulta suas provas de correção. O uso de ferramenta formal no estágio do projeto de linguagens é certamente um suporte para se projetar "melhores" linguagens de programação, no sentido de serem mais naturais, conceitualmente claras e poderosas.

A contribuição deste trabalho é esclarecer o comportamento de programas concorrentes cujos processos se comunicam remotamente(*) através de trocas de mensagens, para que no futuro próximo se tenha uma caracterização desse estilo de programação pelo menos tão completa quanto hoje se tem de programação concorrente usando variáveis compartilhadas. ((*) Usamos aqui a termo remoto para troca de mensagens em sistemas distribuidos , distinguindo de troca de mensagem através de compartilhamento de recurso).

Na litetura encontram-se técnicas desenvolvidas para verificação de programas concorrentes, como em Misra e Chandy |62|, Good et al |35|, Levin e Gries |55|, Apt et al |5| e Cunha |25|, para citar alguns, porém os mecanismos estudados nessas abordagens são bastante simples e em geral o ambiente de computação é estático.

O que se pretende aqui é dar a semântica de mecanismos que permitem comunicação de processos através 'de mensagens, em ambientes com tanto criação de processos como topologia dinâmicas, (este conceito será introduzido mais precisamente no capitulo 2) que possam ser introduzidos em linguagens sequenciais usuais, a fim de se obter programas cujo princípio estrutural seja: processos sincronizando remotamente suas atividades através de mensagens, num ambiente dinâmico.

A interface entre os mecanismos e as máquinas que os suportam, está sendo elaborada por Cláudio Kirner (Universidade Federal de São Carlos).

Os mecanismos aqui propostos são semelhantes aos propostos por Ruggiero e Bressan em |76|, porém ao tentarmos dar um significado formal aos mesmos algumas modificações foram introduzidas na sintaxe e finalmente na semântica pretendida. Essas modificações foram impostas pelo desejo de esses mecanismos poderem ser usados de maneira confiável.

Na descrição de qualquer linguagem é necessário empregar-se uma metalinguagem, que é a linguagem ou notação empregada para descrever a primeira. Quanto mais formal é a descriação da linguagem, mais formal a metalinguagem. Assim, se estivéssemos nos propondo a dar uma descrição puramente formal dos mecanismos expostos aqui, poderíamos optar pela lin-

guagem de definição de Viena, VDL, difundida por Wegner |85| , que é talvez a mais antiga e melhor maneira de se descrever a semântica operacional. No entanto, quem já usou VDL para definir uma linguagem, a mais simples que seja, tem idéia do tamanho que estas definições podem ter, além de que, o detalhamento de cada uma das definições dos predicados e instruções em VDL pode afastar o leitor das principais características da semântica descrita. Por isso, optamos por uma forma de descrição dos mecanismos que seja precisa o suficiente para qualquer implementador dos mesmos poder compreender todas peculiaridades destes sem ambiguidade, mas natural o bastante para que a leitura desse trabalho possa ser proveitosa para outros além dos implementadores.

A organização dos capítulos desse trabalho é como se segue. No capitulo 2 damos os conceitos envolvidos em programação paralela em geral, com referências às implementações dos mesmos em linguagens desenvolvidas recentemente.

No capitulo 3 damos a descrição informal da sintaxe dos mecanismos propostos, com indicações de como deve ser obtida uma sintaxe abstrata dos mesmos, e justificamos as restrições impostas por motivos semânticos aos mecanismos.

No capítulo 4 damos semântica operacional dos programas no que se refere aos mecanismos aqui propostos.

No capitulo 5, damos os primeiros passos a seguir em direção de uma semântica axiomãtica para programas usando tais mecanismos.

O capitulo 2 é subdividido em quatro seções, onde abordamos problemas específicos da notação de programação concorrente, apresentamos algumas formas de especificação de exe-

cução paralela, damos uma breve descrição de primitivas de comunicação em ambiente com compartilhamento de memória e discutimos primitivas para troca de mensagens.

Os capítulos 3 e 4 formam a parte central desse trabalho. O capítulo 3 é subdividido em duas partes: na primeira a sintaxe dos mecanismos é apresentada e na segunda apresentamos as razões pelas quais foram feitas restrições à sintaxe, exemplificando os problemas típicos que poderiam surgir caso os mecanismos fossem usados irrestritamente.

No capítulo 4 é dada a semântica dos programas usando tais mecanismos. Supomos que a rede de transmissão das mensagens é confiável, mas levamos em conta que: 1) os processos podem estar sendo executados em localidades distintas e portanto suas ações não precisam ser necessariamente intercaladas, e 2) as mensagens são transmitidas de maneira imprevisível pela rede, modelando-se assim situações computacionais que ocorrem na realidade.

No capitulo 5 listamos algumas propriedades dos mecanismos que podem servir de subsidio para uma axiomatização da semântica dos mesmos usando lógica temporal. Essas propriedades estão coerentes com o modelo operacional do capítulo 4, mas são expressas de uma forma mais abstrata. Elas pretendem ser parte da especificação de linguagens usando tais mecanismos e prover fundamentos para verificação de propriedades de segurança e de vida de programas nessas linguagens.

CAPITULO II

PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA

Neste capítulo abordaremos os primeiros conceitos relacionados ao projeto e construção de programas concorrentes e distribuidos e algumas notações para descrever computações paralelas. Para tal, mencionaremos algumas linguagens de programação cuja relevância se deve ao fato de que, durante seus projetos, conceitos novos foram introduzidos ou mesmo porque apresentam um progresso no sentido de se obter uma compreensão mais precisa das interrelações entre determinados mecanismos de programação.

Dividiremos o capitulo em quatro seções. Na primeira discutiremos os três problemas básicos a todas notações de programação concorrente, quais sejam: como expressar execução concorrente, como processos se comunicam e como processos sincronizam suas execuções. Na segunda, abordaremos algumas formas de especificação de execução paralela: co-rotinas, comandos *fork/join* e *cobegin/coend* e declaração de processos. Na terceira, daremos uma breve descrição de primitivas de sincronização usadas quando a comunicação é feita via variáveis compartilhadas e maneiras de implementá-las através de espera ocupada, regiões criticas, monitores e expressões de caminho. Na quanta seção discutiremos primitivas para troca de mensagens foaalizando a discussão nas seguintes questões semânticas: comunicação sincrona e ou assíncrona entre processos, especificação dos canais de comunicação e tratamento de falhas.

Essa exposição não pretende ser exaustiva, mas relatar alguns conceitos básicos e problemas relacionados a programação concorrente.

Seção 1 - *Processos e suas Interações*

Um programa sequencial especifica que a execução dos comandos de uma lista de comandos será feita sequencialmente. Um processo é uma seqüência de operações feitas uma de cada vez. (A definição precisa de uma operação depende do nível de detalhe que usamos para descrever um processo). Assim a execução de um programa sequencial é um processo.

Processos são chamados concorrentes, segundo Brinch Hansen |11|, se suas execuções se intercalam no tempo. Toda vez que a primeira operação de um processo puder começar antes que a Última operação de outro processo terminar, esses dois processos são chamados de concorrentes.

Um programa concorrente especifica execução de dois ou mais processos que podem ser executados concorrentemente.

É preciso notar aqui que na literatura esses conceitos não são sempre entendidos dessa forma. O conceito de concorrencia entre processos aparece ligado, às vezes, a interação entre processos, enquanto que o fato de as operações dos processos poderem ser efetuadas simultaneamente ou intercaladas é que caracteriza o paralelismo entre processos. Usaremos aqui a noção de processo concorrente segundo Brinch Hansen, quando há ou não interrelação entre os processos.

A execução de processos concorrentes pode ser simultânea ou intercalada. Execução intercalada é obtida quando proces

sos compartilham um ou mais processadores e é chamada de multiprogramação, (Dijkstra |27|). Multiprogramação é estruturada sobre um núcleo do sistema operacional que multiplexa os processos e processadores (i.e., gerencia o uso dos processadores pelos processos). A execução simultânea é chamada de multiprocessamento se os processadores compartilham uma mesma memória como em Jones e Schwarz |44|, ou de processamento distribuído, se os processadores estão ligados por uma rede de comunicação. Em geral, um programa concorrente cuja execução é feita por processamento distribuído é chamado de programa distribuido.

Existem formas hibridas de execuções, por exemplo, processadores em sistemas distribuidos são frequentemente multiprogramados.

A velocidade de execução de cada processo depende da abordagem usada. Quando cada processo é executado no seu próprio processador, a velocidade de cada processo é fixa (talvez desconhedida). Quando processos compartilham um mesmo processador, é como se cada processo estivesse sendo executado num processador de velocidade variável;.

Como, no entanto, o que se deseja é entender um programa concorrente em função de seus componentes sequenciais e suas interações, independentemente da maneira como são executados, não são feitas suposições sobre as velocidades de execução dos processos concorrentes, exceto que estas são positivas.

Processos podem cooperar, no sentido de um processo necessitar de ações feitas por outros processos, e também podem competir, no sentido de dois ou mais processos compartilharem um mesmo recurso. Por exemplo, considere o exemplo de um "buffer" de tamanho limitado, usado por dois processos: o produtor

que coloca itens no "buffer" e o consumidor que retira itens do "buffer". Os dois processo cooperam entre si, pois o processo consumidor precisa que o consumidor retire itens do "buffer", já que o tamanho deste é limitado.

Um exemplo comumente dado para ilustrar competição entre recursos é o dos "dinning philosophers" (Hoare [41]), onde garfos são compartilhados e os filósofos competem entre si pelo uso exclusivo dos mesmos.

Tanto cooperação como competição entre processos exige alguma forma de comunicação entre os processos.

Comunicação entre processos permite que a execução de um processo interfira na execução do outro. A comunicação entre processos pode ser feita através de variáveis compartilhadas (variáveis que podem ser referenciadas por mais de um processo) ou por troca de mensagens.

Em geral, quando processos se comunicam há necessidade de uma certa sincronização de suas execuções, pois processos são executados em velocidades imprevisíveis e, para haver comunicação entre processos, um precisa efetuar uma certa ação e o outro detetá-la. A comunicação entre processos só vai funcionar se esses dois eventos "ação" e "deteção de ação" ocorrem nessa ordem. Assim, podemos considerar a sincronização entre processos como um conjunto de restrições sobre a ordem dos eventos gerados por eles.

O programador então emprega certos "mecanismos de sincronização" para atrasar a execução de um processo a fim de que essas restrições possam ser satisfeitas.

Por exemplo, no caso de acesso a um "buffer" de tamanho limitado pelo produtor e consumidor, mencionado acima, o

processo consumidor precisa esperar até que haja itens no "buffer" para poder consumí-los, enquanto que o produtor precisa esperar até que haja lugar no "buffer" para poder colocar itens no mesmo.

O enfoque da sincronização como restrições sobre eventos se origina na abordagem à semântica do programa do ponto de vista operacional: uma execução de um programa concorrente é considerada como uma sequência de "ações atômicas", cada uma resultando na execução de uma operação indivisível. Essa sequencia vai permitir que ações atômicas de um processo de intercalem com ações atômicas de outros. Porém, nem todas as formas de intercalação de ações vão resultar em um comportamento aceitável do programa. É aí, então, que entra a sincronização para garantir que não ocorram intercalações de ações que levem a comportamantos inaceitáveis.

Outra maneira de se considerar o papel da sincronização é através da abordagem axiomática. Nessa abordagem, a semântica dos comandos é definida por axiomas e regras de inferência, dando origem a um sistema formal denominado de lógica de programação. Teoremas nessa lógica, conforme métodos desenvolvidos por Floyd |32|, especificam uma relação entre os comandos (S) e dois predicados P e Q; P é uma pré-assertiva de S e Q é uma pós-assertiva de S.

Os axiomas e regras de inferência são escolhidos de modo que os teoremas da forma (P)S(Q) tenham a seguinte interpretação:

se a execução dos comandos S começar num estado que satisfaz P e se a execução de S terminar, então a pós-assertiva Q é sa-

tisfeita no estado então atingido. (Lamport |49| usa a mesma notação com outro significa pretendido). Assim, os comandos são considerados como gerando relações entre predicados. Os sistemas formais, consistindo dos axiomas e regras de inferência tais que os teoremas tem a intepretação acima menciona, são ditos sistemas de prova para oorreção parcial de programas. Nos sistemas de prova de correção total de programas, na interpretação dos teoremas, não há a suposição de que os comandos terminem suas execuções. Isto inclusive é uma propriedade que se quer verificar através dos axiomas e regras.

Um programa anotado ou um resumo de prova é uma forma de se apresentar o programa e uma prova de sua correção juntos. Consiste em intercalar o texto do programa com assertiva de forma que cada tripla (P)S(Q) (constituída por: - a assertiva P que precede textualmente S no programa anotado, o oomando S e a assertiva Q, que segue textualmente S no programa anotado), seja um teorema na lógica de programação.

Quando a execução concorrente entre processos for possível, a prova de um processo sequencial é válida só se a execução concorrente de outros processos não invalidar as asserti vas que aparecem nessa prova. Uma maneira de estabelecermos essa validade é supormos que o código de programa entre quaisquer duas assertivas de um programa anotado é executado atomicamente e então provarmos teoremas mostrando que a execução .de nenhum comando em um processo invalida as assertivas da prova dos outros. Esses teoremas constituem a chamada prova de não interferência (Ashcroft |4|, Keller |45|, Lamport |47, 48, 49|, Lamport e Schneider |50| e Owicki |67, 681).

Lógica temporal tem sido usada como fundamento para argumentação sobre programas concorrentes, como proposto em Pnueli |72, 73|.

A lógica temporal é uma extensão da lógica de primeira ordem contendo operadores que permitem as fórmulas serem interpretadas como propriedades que mencionam a passagem de tempo (por exemplo, a fórmula $\Box P$, usando o operador $\Box$, é interpretada como P é sempre verdade, a partir de agora).

As dificuldades inerentes à programação concorrente se referem a não funcionalidade dos programas como também a não contigüidade na execução de processos paralelos. (Programas funcionais são aqueles que para uma determinada entrada darão sempre a mesma saída, independente das máquinas usadas para sua execução, da velocidade relativa dessas máquinas e das diversas execuções do programa).

A não funcionalidade dos programas significa que não podemos inferir a relação entrada-saída dos processos concorrentes somente a partir das relações entrada-saída dos seus processos sequenciais, vistos isoladamente. Em programas concorrentes o comportamento dos seus processos sequenciais pode depender dos seus tempos de execução e não mais será verdade que todas as suas execuções apresentarão os mesmos resultados. No entanto é desejável que os processos sequenciais apresentem um comportamento funcional, no sentido de que em certos pontos de sua computação se encontrem em determinados estados esperados.

A não contigüidade na execução dos processos é devida à comunicação e à sincronização entre os mesmos. No entanto as restrições ao modo como os processos são executados, impostas

pela comunicação e sincronização, podem levar a situações indesejáveis tais como bloqueio perpétuo (deadlock) ou adiamento indefinido (starvation).

A lógica temporal tenta evitar esses problemas da seguinte forma:

i) analisa e formaliza propriedades em termos sequênciais de execuções;

ii) fundamenta seus princípios em pressuposições de descontinuidade.

Assim, os mecanismos de sincronização controlam interferências de duas maneiras. Primeiro, eles podem atrasar a execução de um processo até que uma certa condição (assertiva) seja verdadeira. Dessa forma a pré-assertiva do comando subsequente está garantidamente verdadeira (cdntando que esta não sofra interferência pela execução de outros comandos). Segundo, um mecanismo de sincronização pode ser usado para assegurar que um bloco de comandos seja executado como se fosse uma operação indivisível, eliminando-se assim a possibilidade de interferência de comandos de outros processos com .as assertivas que ocorrem dentro desse bloco.

As duas abordagens, operacional e axiomãtica são úteis: a operacional para se explicar como os mecanismos de sincronização funcionam e a axiomática para uma melhor compreensão do comportamento de um programa e argumentação sobre sua correção. Como o número de intercalações consideradas na abordagem operacional cresce exponencialmente com o número de processos e o

tamanho dos processos do programa, a abordagem axiomática parece ser mais promissora para a compreensão de programas concorrentes. No entanto, em programas distribuídos com número variável de processos, e interconexões entre processos, o número de teoremas a serem provados na prova de não-interferência também é, muito grande, e depende da própria execução, a não ser que se restrinja o grau de paralelismo entre os processos, de forma que nesses programas as dificuldades das duas abordagens tendem a se igualar.

Seção 2 - *Especificação de Execução Concorrente*

Vamos descrever alguns mecanismos linguísticos para expressar execução em paralelo (concorrente). Cada um deles pode ser usado em computações cujo número de processos é fixo (ambiente estático) ou usado em combinação com os mecanismos de criação de processos, para especificar computações tendo um número variável de processos (ambiente dinâmico).

2.1 - Co-rotinas

Co-rotinas são análogas a sub-rotinas, só que permitem transferência de controle de uma forma simétrica ao invés de hierárquica (Conway |22|). O controle é transferido entre co-rotinas através do comando *resume*. Execução de um *call* é como a execução de uma chamada de procedimento, *call*, porém a co-rotina evocada retorna o controle à rotina original executando outro *resume*. Ainda, qualquer outra co-rotina pode retornar o controle à rotina original. (Por exemplo, a co-rotina C1 pode *resume* C2, que *resume* C3 que resume C1). O comando *resume* é o único meio de transferência de controle entre co-rotinas e

essa transferência pode ser feita entre quaisquer co-rotinas. Cada co-rotina pode ser interpretada como implementando um processo e a sincronização entre processos é feita através da execução do *resume*.

Co-rotinas são mais apropriadas para programas compartilhando um único processador. Não são muito adequadas para processamento realmente feito em paralelo já que sua semântica só permite a execução de uma rotina de cada vez. Comandos para implementar co-rotinas foram usados em SIMULA |65| e mais recentemente em MODULA-2 |87|.

2.2 - Comandos *Fork* e *Join*

O comando *fork* especifica que uma dada rotina vai começar a ser executada. Porém, a rotina que chama e a rotina chamada prosseguem concorrentemente. A rotina solicitante pode sincronizar sua execução com o término da execução da rotina solicitada, executando o comando *join*. Execução de Join pela rotina solicitante causa bloqueio desta até que a rotina solicitada termine sua execução. As primeiras aplicações *fork/join* aparecem em Conway |23| e Dennis |26|.

Pelo fato de *fork* e *join* poderem ser usados em comandos condicionais e em laços (loops), uma cuidadosa compreensão da execução do programa é necessária para sabermos quais rotinas serão executadas concorrentemente. Porém, quando usados de uma forma disciplinada, esses comandos são bastante práticos e poderosos. Por exemplo, *fork* é um mecanismo direto para criação dinâmica de processos, incluindo multiativação do mesmo texto de programa.

O sistema operacional UNIX |77| usa intensamente variantes dos comandos *fork* e *join*, PL/I e MESA |63| usam comandos semelhantes a *join* e *fork*.

2.3 - Comando *Cobegin*

O comando *cobegin* (primeiramente chamado de *parbegin* por Dijkstra |28|) é uma forma estruturada de denotar a execução concorrente de um conjunto de comandos. Execução de um *cobegin* S1//..//Sn *coend* causa execução paralela de S1,...,Sn, sendo que cada Si pode ser qualquer comando, inclusive um *cobegin*, ou um bloco com declarações locais. A execução de *cobegin* termina quando todos os Si's tiverem terminado.

Cobegin só pode ativar um número fixo de processos, contrastando com *fork/join*, que pode ser usado para criar um número arbitrário de processos, mas tem sido usado em muitas linguagens. Sua sintaxe explicita quais rotinas são executadas concorrentemente e constitui uma estrutura de controle com uma entrada e uma saída, o que permite qua a transformação de estado provocada por um *cobegin* seja compreendida por si só, e então usada para o entendimento do programa do qual faz parte.

Variantes de cobegin foram introduzidas em ALGOL 68 |82|, Communicating Sequential Processes (CSP) |43|, Edison |15| e Argus |57|.

2.4 - Declarações de Processos

Programas grandes, em geral, são estruturados como uma coleção de rotinas sequenciais que são executadas em paralelo. Embora essas rotinas possam ser declaradas como procedi-

mentos e ativadas por *cobegin* ou *fork*, a estrutura do programa fica melhor delineada se a declaração de uma rotina indica se ela vai ou não ser executada concorrentemente. Essa flexibilidade pode ser obtida através de declaração de processos.

Em algumas linguagens concorrentes, por exemplo, DISTRIBUTED PROCESSES (DP) |14| e SYNCRONIZING RESOURCES (SR) |1|, uma coleção de declarações de processos é equivalente a um cobegin onde cada proces'so declarado é um componente do cobegin. Isso significa que existe exatamente uma instãncia de caáa processo.

Outras linguagens têm um mecanismo específico, fork ou outro semelhante, para evitar instanciações dos processos declarados. Em Pascal Concorrente |13| e Modula |86|, por exemplo esse mecanismo só pode ser usado na inicialização do programa, o que fixa o número de processos mas permite criação de múltiplas instanciações dos processos declarados. Outras linguagens, como por exemplo, PLITS |30| e ADA |81|, permitem que processos sejam criados em qualquer momento durante a execução, fazendo que haja computação com um número variável de processos.

Seção 3 - *Primitivas de Sincronização Baseadas em Variáveis Compartilhadas*

Quando a comunicação entre processos é feita através de variáveis compartilhadas, dois tipos de sincronização são úteis: exclusão mútua e sincronização condicional.

Em geral, o trecho de código de programa executado por um processo que acessa recursos que são compartilhados, mas

que não podem ser acessados simultaneamente, é chamado de seção ou região crítica. O problema então se reduz a garantir que as regiões críticas de dois processos não sejam executadas ao mesmo tempo.

Exclusão mútua assegura que uma sequência de comandos seja tratada como se fosse uma operação indivisível. Sincronização condicional é usada para coordenar a execução de processos concorrentes quando um recurso compartilhado se encontra num estado em que uma certa operação não pdde ser executada sobre ele. Qualquer processo pretendendo executar essa operação será atrasado até que o estado do recurso mude, através da ação de outros processos. (Por exemplo, o processo consumidor de itens de um buffer ficará atrasado, se o buffer estiver vazio, até que um processo produtor coloque itens no buffer).

Existem vários mecanismos para implementar esses dois tipos de sincronização. Aqui vamos apenas mencionar o nome e descrição sumária de alguns deles, uma vez que não são partes essenciais desse trabalho.

3.1 - Espera Ocupada

Uma maneira de implementar sincronização é fazer cada processo atribuir valores e testac variáveis compartilhadas. Essa abordagem funciona razoavelmente bem para implementar sincronização condicional, mas não tão bem para implementar exclusão mútua. A fim de analisar uma condição, um processo atribui um valor a uma variável compartilhada. Para esperar até que aquela condição seja satisfeita, o processo fica testando essa variável até que ela atinja o valor desejado. O nome de "espe-

ra ocupada" vem do fato de o processo ficar testando repetidamente a variável enquanto espera pela condição, sem realizar nenhum processamento Útil.

Para implementar exclusão mútua usando espera ocupada, a sinalização e a espera das condições são combinadas em protocolos cuidadosamente elaborados como por exemplo, em Peterson |71|, já que a atribuição-teste de variável não e uma operação indivisível e pode acontecer que dois ou mais processos tenham acesso simultâneo às variáveis que deveriam ser usadas de forma mutuamente exclusiva. Além disso, protocolos que usam somente espera ocupada são difíceis de projetar, compreender e provar correção.

Outras desvantagens do uso de espera ocupada é que, primeiro, o processador executando um laço de espera ocupada poderia estar sendo usado mais eficientemente enquanto a condição não é satisfeita e segundo, ao se ler um programa que usa espera ocupada, fica difícil distinguir as variáveis compartilhadas que estão sendo usadas para implementar a sincronização, das que são usadas para comunicação, propriamente dita, entre processos.

3.2 - Semáforos

O conceito de semáforos foi introduzido por Dijkstra |27, 28|. A maior contribuição obtida através do conceito de semáforos é que o processo ao ser bloqueado é colocado numa fila de espera de forma que libera o processador e evita a condição esperada para ser desbloqueado fique sendo testada repetidamente. Um semáforo é uma variável inteira, cujo valor é não

negativo e sobre o qual estão definidas duas operações: P e V. Se s é um semáforo o processo executando P(s) fica atrasado até que s > 0 e então executa s <-- s-1.

V(s) incrementa o semáforo s de 1.

Ambas operações P e V são consideradas indivisíveis.

Para implementar exclusão mútua, cada seção crítica é precedida por uma operação P e seguida por uma operação V sobre o mesmo semáforo.

Para implementar sincronização condicional, são usadas variáveis compartilhadas para representar a condição e um semáforo é associado à condição. Um processo após tornar a condição verdadeira avisa esse fato executando V sobre o semãforo. Um processo espera pela condição executando P sobre o semáforo.

As operações P e V são implementadas como rotinas de baixo nivel do sistema operacional e são acessíveis aos processos através de chamadas do supervisor.

Semáforos são uma ferramenta poderosa e flexível para comunicação entre processos, podendo ser usados para implementar a maioria dos casos de sincronização. Porém, as operações P e V são primitivas não estruturadas, o que dá origem às seguintes desvantagens:

- soluções complexas para problemas simples;
- uso irrestrito para o programador: não existem regras para uso de semáforos e as seções criticas são dificeis de reconhecer pelo texto do programa;

- não existem testes em tempo de compilação para detetar erros no uso de semáforos (se falta algum P ou V, ou se estão sendo usados em lugar indevido) ou para testar se as variáveis compartilhadas estão sendo usadas só dentro de regiões críticas,

- prova de correção difícil;

- tanto exclusão mútua como sincronização condicional não programadas usando o mesmo par de primitivas, o que dificulta a identificação de para qual propósito uma dada operação P ou V está sendo usada.

3.3 - Regiões Críticas Condicionais

As dificuldades acima mencionadas são superadas pelo uso de uma ferramenta estruturada que restringe a utilização do recurso compartilhado dentro de seções do programa, chamadas regiões críticas [10, 12, 41], e que permitem deteção de erros em tempo de compilação. As variáveis compartilhadas são explicitamente agrupadas, em recursos. Cada variável compartilhada pode pertencer no máximo a um recurso e só pode ser acessada em regiões críticas condicionais, que se eferem àquele recurso.

Exclusão mútua é obtida ao se garantir que execuções de diferentes regiões críticas referindo-se ao mesmo recurso não são intercaladas.

Sincronização condicional é obtida pelo uso explícito de condições booleanas no comando de região crítica condicional.

Um recurso r, contendo as variáveis vl,...,vn é declarado, segundo notação por Andrews e Schneider |3|, por

resource r: vl,...,vn

(Essa notação combina os aspectos das notações propostas em |10, 12 e 41|).

As variáveis em r só podem ser acessadas dentro de um comando de região crítica condicional que se refira a r. Tal comando tem a forma (segundo Andrews e Scheneider |3|) *region* r *when* B *do* S, onde B é uma expressão booleana e S uma lista de comandos.

A execução do comando *region* atrasa o processo até que B seja verdadeiro, quando então S é executado. A avaliação de B e execução de S não podem ser interrompidas por outro comando *region* que se refira ao mesmo recurso. Assim B permanece verdadeiro até que S comece a ser executado.

Programas escritos usando regiões críticas condicionais são adequados à abordagem axiomática. Cada região crítica condicional implementa uma operação sobre o recurso a que se refere. Associa-se, a cada recurso, um invariante Ir, que deve ser verdadeiro após cada execução de uma operação sobre o recurso. A expressão booleana B na região crítica g escolhida de tal forma que a execução da lista de comandos se inclue num estado satisfazendo Ir & B e termine satisfazendo Ir. Assim o invariante será sempre verdadeiro, a não ser talvez dentro de alguma região critica associada ao recuros do invariante.

Portanto, já que a execução de regiões críticas de um recurso não pode ser interrompida pela execução de outras do

mesmo recurso, as provas dos processos são livres de interferência caso:

i) variáveis locais a um processo só aparecem na prova daquele processo;

ii) variáveis de um recurso aparecem apenas em assertivas dentro de regiões críticas referindo-se aquele recurso.

Desta forma, pela escolha apropriada dos invariantes dos recursos, um programa concorrente será compreendido em termos de seus componentes sequenciais.

A principal desvantagem de regiões críticas condicionais é a espera ocupada controlada, implícita na sua implementação.

Regiões criticas condicionais são usadas como mecanismo de sincronização em EDISON |15|, que é uma linguagem projetada especificamente para sistemas com multiprocessadores. Variantes desse conceito foram usadas em DISTRIBUTED PROCESSES |14| e ARGUS |57|.

3.4 - Monitores

Regiões críticas condicionais (r.c.c.), apresentadas em 3.3, são de cara implementação em sistemas com um único processador. Além disso, as operações sobre os recursos feitas através de r.c.c., ficam espalhadas pelos diferentes processos, o que dificulta a compreensão do programa. O conceito do monitor, proposto pr Dijkstra |28| e desenvolvido por Brinch Hansen |11| e Hoare |42|, resolve essas dificuldades.

Um monitor coloca num mesmo lugar tanto a defin'ição como as operações sobre um recurso. Assim, um recurso sujeito a acesso concorrente é visto como um módulo (conceito introduzido por Parnas em |70|), o que permite ao programador, ao usá-lo, não se preocupar com detalhes de sua implementação e, ao programar o monitor que implementa o recurso, não se preocupar com a maneira que o recurso vai ser usado.

Um monitor consiste de um conjunto de variáveis compartilhadas, usadas para armazenar o estado do recurso, e alguns procedimentos que implementam as operações sobre o recurso. Um monitor tem também um código de inicialização das variáveis que é executado uma Única vez antes de qualqur procedimento. Os valores das variáveis compartilhadas são mantidos entre ativações dos procedimentos do monitor e só podem ser acessados de dentro do monitor. Os procedimentos são mutuamente exclusivos no tempo, de forma que os invariantes associados ao recurso são preservados. Os procedimentos do monitor podem usar dados locais e parâmetros, que tomam novos valores a cada ativação do procedimento.

Os procedimentos do monitor usam dados locais e primitivas de sincronização para escalonar o uso de recurso de forma desejada. A estrutura.de um monitor segundo Hoare |42| é:

monitor nome do monitor
  vun declaração de variáveis permanentes
  procedure nome do procedimento (par. formais)
    begin corpo do procedimento *end;*

... declarações de outros procedimentos do monitor ...

  begin inicialização das variáveis permanentes *end*
end

A chamada de um procedimento de um monitor é feita de maneira usual: nome_do_monitor.nome_do_procedimento (par. reais).

Em |42|, Hoare propôs o conceito de variável de condição para atrasar processos executando chamadas de procedimentos de monitores. Duas operações podem ser feitas sobre uma variável de condição: *signal* e *wait*.

Variáveis de condição podem ser representadas por uma fila, inicialmente vazia. Quando um processo executa cond.*wait*, ele é colocado na fila correspondente à condição cond e libera a exclusão mútua do monitor. Quando um processo executa um cond.*signal*, um dos processos da fila de cond é reativado e garante que o processo sinalizado re-entre no monitor antes que outros possam entrar. As variáveis de condição são declaradas localmente ao monitor, portanto, essas operações só podem ser usadas dentro do monitor.

O escalonamento do uso de recursos é responsabilidade do programador, não estando implícito nas primitivas *wait* e *signal*. Estas operações estão implementadas sem usar espera ocupada.

Outras abordagens para sincronização evoluiram do conceito de variáveis de condição: filas com as operações *delay* e *continue*, usadas em PASCAL CONCORRENTE |13|, variáveis de condição com operação *notify* usados em MESA |63, 51|.

O principal valor do conceito de monitor é que ele impõe restrições ao programador que podem ser testadas em tempo de compilação e, portanto, reduzem o número de erros em potencial, dependentes do tempo, no momento de execução.

Tais restrições são:

- um processo só pode ter acesso ao recurso através de um procedimento do monitor;

- as variáveis que representam o estado do recurso são acessíveis só dentro dos monitores;

- a sincronização dos processos que competem por um recurso é manipulação inteiramente dentro do monitor.

Várias linguagens foram propostas e implementadas usando monitores para sincronizar o acesso a variáveis compartilhadas. As linguagens mais importantes, no sentido de terem inspirado outras linguagens, foram PASCAL CONCORRENTE |13| e MODULA |86|.

3.4. - Expressões de Caminho

As operações definidas por um monitor são executadas de maneira mutuamente exclusiva. As outras formas de sincronização dos procedimento do monitor são feitas de maneira explícita através das operações sobre as variáveis de condição ou mecanismos similares. Consequentemente, a sincronização das operações do monitor é feita através de código espalhado pelo monitor.

Outra forma de definir um módulo sujeito a acesso concorrente é através de um mecanismo com o qual o programador especifica num só lugar, em cada módulo, todas as restrições sobre a execução das operações definidas no módulo. A implementação das operações fica separada da especificação das restrições, e o código que reforça as restrições é gerado por um com

pilador. Essa é a filosofia de uma classe de mecanismos de sincronização, chamada expressões de caminho. Expressões de caminho foram definidas por Campbell e Habermann |17|. Variantes e extensões de expressões de caminho foram propostas posteriormente por Lauer e Campbell |52|, Habermann |37|, Flon e Habermann |31|, Campbell |18| e Lauer e'Shields |53|. Vamos apresentar a proposta de Campbel |18|, que foi incorporada em PATH PASCAL |19|.

Quando expressões de caminho são usadas, um módulo, que implementa um recurso, tem uma estrutura semelhante a de um monitor: variáveis compartilhadas, que armazenam o estado do recurso, e procedimentos que efetuam operações sobre o recurso. Expressões de caminho são introduzidas no cabeçalho de cada recurso, definindo as restrições sobre a ordem na qual as operações devem ser executadas.

A sintaxe de uma expressão de caminho é: *path* lista de caminhos *end*.

A lista de caminhos contém os nomes da operações e operadores de caminho. Esses operadores incluem:

";" especificando sequenciação,

"," especificando concorrência,

"n:(lista de caminhos)" especificando o número máximo de ativações concorrentes da lista de caminhos,

"|lista de caminhos|" especificando um número arbitrário de ativações da lista de caminhos.

Expressões de caminho foram baseadas e motivadas pela abordagem operacional à semântica do programa. Uma expressão de caminho define todas as sequências permissíveis de execuções de uma operação sobre um determinado recurso. Esse conjunto de operações pode ser visto como uma linguagem formal, na qual cada sentença é uma seqüência dos nomes das operações. Dai a semelhança entre expressões regulares e expressões de caminho.

Expressões de caminho, apesar de constituirem uma forma elegante de expressar restrições na ordem das operações, são de pouca valia para especificar sincronização condicional, já que a decisão de poder executar uma operação ou não pode depender do estado do recurso de uma forma que não esteja diretamente relacionada com o histórico das operações já executadas sobre o recurso (p. ex. o problema dos escritores/leitores com preferência para escritores, e acesso justo para ambos. Neste caso é necessário saber o número de leitores e escritores esperando para implementar a sincronização desejada, e não somente a ordem das operações já efetuadas).

Nestes casos, além do uso de expressões de caminhos, outros mecanismos devem ser introduzidos. Apesar de vários esforços terem sido feitos para estender o conceito de expressões de caminho a fim de solucionar o problema da sincronização condicional, nenhuma das extensões se manteve tão simples e elegante quanto o conceito original. Contudo, Schields [79], Schaw [78] e Best [9] mostraram a utilidade de expressões de caminho na especificação de computação concorrente.

Seção 4 - *Primitivas de Sincronização Baseadas em Troca de Mensagens*

Um dos aspectos fundamentais no compartilhamento de recursos é a manutenção de integridade dos recusos. Isto pode ser obtido por duas maneiras: 1) garantindo-se exclusão mútua entre processos que competem pelo mesmo recurso ou cooperam entre si através da utilização do recurso ou, 2) assegurando-se que apenas operações permitidas a partir de uma definção previamente estabelecida sejam efetuadas sobre um recurso.

O primeiro enfoque foi a principal preocupação das técnicas de semáforos e regiões críticas descritas anteriormente.

O uso de troca de mensagens entre processos visa a abordagem do segundo enfoque. A troca de mensagens é implementada através do uso de processo gerente destinado a disciplinar a utilização de um determinado recurso. Esse processo será o Único a manipular diretamente o recurso. Desse modo, somente as operações definidas no corpo do processo gerente é que poderão ser realizadas sobre o recurso.

Regiões críticas, monitores e expressões de caminho podem ser consideradas como uma extensão do conceito de semáforos e constituem maneiras estruturadas de controlar acesso a variáveis comparti1hadas. Outro tipo de generalização de semáforos é troca de mensagens, que pode ser visto também como uma evolução de semáforos no sentido de estende-los para transmitir sinais com conteúdo.

Quando troca de mensagens é usada como instrumento de comunicação e sincronização, os processos mandam e recebem men

sagens, ao invés de ler e modificar variáveis compartilhadas.

A troca de mensagens permite comunicação pois um processo, ao receber uma mensagem obtém valores do processo remetente, e também permite sincronização, já que uma mensagem só pode ser recebida após ter sido mandada, o que restringe a ordem na qual esses dois eventos podem ocorrer.

Dentro do contexto de sistemas com memória compartilhada um dos melhores exemplos do uso de troca de mensagens é o sistema RC4000 descrito em Brinch Hansen (11|. Nesse sistema dois processos concorrentes cooperam através do envio de mensagens. As mensagens são transmitidas de um processo para outro através de 'buffers' de mensagens que são selecionadas a partir de um "pool" comum a todos os processos. É definido um monitor que administra uma fila de mensagens para cada processo. As mensagens são colocadas nessa fila quando chegam vindas de algum outro processo. A fila de mensagens faz parte da descrição do processo. Depois de tratar a mensagem, o processo receptor retorna uma resposta através do mesmo buffer de onde veio a mensagem.

Como mencionamos acima, trocas de mensagens podem ser usadas em sistemas de memória comparti1hada, mas recentemente esse conceito está mais ligado a sistemas de memóris distribuída.

Nestes sistemas, a comunicação entre processos através de troca de mensagem se adapta de maneira muito natural e tem sido considerada na literatura como um método bastante Útil e de grande generalidade.

Em sistemas de trocas de mensagens vários tipos de primitivas são estudadas, visando-se a construção de primiti - vas poderosas que possam prover sincronização e transmissão de

dados.

Essas primitivas são do tipo *send*(envia)/*receive*(recebe). Um processo executa um nend quando quer passar uma informação para outro processo e esse outro recebe a informação executando, um *receive*.

A forma geral segundo Andrews e Schneider |3| de um comando *send* é *send*<lista de expressões> *to* <designador do destino>.

Os valores das expressões na hora que o *send* e executado estão contidos na <lista de expressões> . O designador de destino dá ao programador controle sobre o destino da mensagem e, portanto, permite ao programador controlar quais comandos podem receber aquela mensagem.

Uma mensagem é recebida pela execução da primitiva *receive*, cuja forma geral segundo |3| é:

receive <lista de variáveis> *from* <designador da origem>

O designador de origem permite ao programador controlar de onde as mensagens partiram e, portanto, quais os comandos que poderiam ter mandado as mesmas. Recebimento de uma mensagem causa primeiro atribuição dos valores na mensagem à lista de variáveis do receive e depois destruição da mensagem.

Várias questões são levantadas na literatura com relação ao sistema de troca de mensagens (Gentleman |34| e Andrews e Schneider |3|).

Essas questões, segundo Gentleman |34|, podem ser classificadas em:

a) questões semânticas: - primitivas síncronas e/ou assíncronas
- canais de comunicação
- tratamento de falhas

b) questões sintáticas: - formato e tamanho de mensagens

A primeira questão semântica relaciona-se com o fato de se a execução dos comandos de comunicação causa ou não atraso. Esta questão tem influência direta no grau de paralelismo entre os processos.

Dizemos que um comando bloqueia se sua execução causa atraso em quem o envocou e não bloqueia se não causa atraso.

As primitivas *aend* e *receive* podem ser bloqueadas ou não, e segundo Mannig et al |60|, existem quatro possibilidades:

bS,bR - *aend* bloqueado e *receive* bloqueado
(modo totalmente síncrono)

bS,nR - *aend* bloqueado e *receive* não bloqueado
(modo meio síncrono)

nS,bR - *aend* não bloqueado e *receive* bloqueado
(modo meio síncrono)

nS,nR - *aend* e *receive* não bloqueados
(modo totalmente assíncrono)

O modo totalmente síncrono garante a indivisibilidade da transmissão de mensagem enquanto que o totalmente assíncro-

no garante o máximo de paralelismo entre processos.

Primitivas bloqueadas têm a vantagem de terem implementação mais simples e podem combinar, de forma elegante, sincronização com transmissão de mensagens. Cada processo isoladamente pode ser totalmente sequencial. Além disso, o que é importante do ponto de vista de implementação, não existe o problema de gerenciamento de "pool" de "buffers" para guardar mensagens para serem processadas.

Quando são usadas primitivas não bloqueadas uma mensagem enviada para um processo que não está pronto para receber deve ser guardada até que isso ocorra. Nesse caso, mensagens são enfileiradas e "buffers" são usados para guardá-las.

Como, por definição, um processo usando *send* não bloqueado pode enviar um número indefinido de mensagens sem ser bloqueado e como para cada implementação o número de "buffers" é limitado, há necessidade de dar um tratamento a esse problema na hora da implementação. Uma forma é limitar o número de send's não bloqueados por processo, ficando o processo bloqueado após esse número ser atingido até que algumas das suas mensagens enviadas tenham sido recebidas. (Nesse caso os buffers estão alocados a cada processo remetente, como, por exemplo, em Accent |74|). Outra forma é retornar um código ao processo que quer enviar a mensagem, indicando que não há buffers disponíveis. (Nesse caso os buffers estão associados aos canais de comunicação, como proposto por Knott |46|). Se o sistema tiver uma capacidade ilimitada de fornecer "buffers", o processo não é atrasado nunca ao executar um *send* Nesse caso pode passar arbitrariamente à frente do processo receptor. Consequentemente, quando uma mensagem é recebida ela pode conter informação sobre o

estado do processo remetente que não é necessariamente o estado atual desse processo.

Envio de mensagens através de sendo bloqueado faz com que o momento da troca de mensagem represente o ponto sincronização na execução de ambos os processos o que envia e o que recebe. A mensagem então recebida representa o estado atual do processo remetente.

A forma bloqueada da operação receive é considerada a mais natural, já que, em geral, um processo que está pronto para receber uma mensagen não tem nada mais a fazer se não recebê-la.

Em alguns casos, é preciso controlar não somente se há mensagens a serem recebidas. Nesse caso o comando *receive* condicional provê essa característica.

Tal comando tem a forma geral, segundo Andrews [3], *receive* <lista-de-variáveis> *from* <designador de destino> *when* B e permite somente o recebimento das mensagens que faze B verdadeiro.

Variantes desse comando, baeado nos "guarded comands" de Dijkstra [29], permitem que um comando *receive* tenha várias opções de recebimento de mensagens e efetue uma determinada ação, dependendo da opção escolhida.

A segunda questão semântica refere-se à especificação dos canais de comunicação entre processos. Um canal de comunicação pode ser definido como o par de designações de destino e de origem na troca de informação.

Existem várias maneiras de designar canais de comunicação: endereçamento implícito, endereçamento direto (ou explícito), endereçamento indireto e endereçamento funcional (essa

classificação é segundo Vinter et al |83|).

Endereçamento implícito: um processo pode se oomunicar com um Único processo do sistema. Nesse tipo de endereçamento, os designadores de destino e de origem não precisam ser especificados nas primitivas. Esse tipo de endereçamento é o mais simples e em geral é usado quando um processo é criado para fazer um certo tipo de serviço e só pode se comunicar com o seu criador. Endereçamento implícito, porém, não permite que dois processos quaisquer se comuniquem, que é o mínimo que se pode esperar de um dispositivo de comunicação.

Endereçamento explícito: nomeação explícita do nome do processo com o qual se quer fazer contato. Muitos sistemas usam esse tipo de endereçamento, por exemplo, CSP de Hoare |43|. O endereçamento explícito pode ser simétrico, quando o nome do processo é incluido nas primitivas de *send* e *receive* ou assimétrico, como em ADA |81|, onde uma tarefa ("task") que chama deve conhecer o nome da tarefa chamada, mas a tarefa chamada não conhece o nome de quem a chama.

Comunicação entre processos usando endereçamento explícito funciona como um sistema de dutos ('pipeline') onde a informação flue de uma forma tal que a saída de um processo serve de entrada para o outro processo.

Comunicação com endereçamento explícito é fácil de ser implementada e possibilita ao processo receptor controlar o momento que recebe mensagens de cada um dos remetentes (no caso simétrico). Porém, requer conhecimento global de todos os processos do sistema e não é flexível o suficiente para manipular algumas situações comuns em sistems distribuidos, tais como migração de processos, ou prestação do mesmo serviço por vá

rios processos, restringindo assim a intercomunicação dos processos.

Entretando, THOTH |20| tem todos os mecanismos de comunicação com endereçamento explicito, o que evidencia que esse tipo de endereçamento se mostra adequado para projetar um sistema completo de comunicação.

Endereçamento indireto: usa nomes globais ou caixas postais ("mailboxes") nas primitivas de comunicação. Uma caixa postal pode aparecer como designador de destino em qualquer comando *send* de qualquer processo ou como designador de origem em qualquer comando *receive* de qualquer processo. Quando mensagens são enviadas para uma dada caixa postal, podem ser recebidas por qualquer processo que execute um *receive* tendo essa caixa postal como designador de origem.

O endereçamento indireto é adequado ao modelo cliente/servidor de comunicação entre processos. (Neste modelo processos servidores prestam serviços para clientes. Quando um cliente requer um serviço, envia uma mensagem requisitando-o para um dos servidores. Um servidos atende ciclicamente aos pedidos dos clientes, retornando ou não à mensage de serviço completo para o cliente). Endereçamento direto para esse modelo causa problema, já que em geral, o servidor deve estar pronto para receber pedidos de qualquer cliente; e se houver mais de um cliente, deveríamos ter um *receive* para cada cliente. Com uso de caixas postais, um cliente deposita o pedido na caixa postal de onde o servidos vem retirá-lo.

Implementação de caixas postais pode ser muito cara, já que, quando uma mensagem é depositada numa caixa postal, todos os processos que podem recebê-la devem ser avisados, e, quan

do essa mensagem for recebida por um desses receptores, todos os outros devem ser avisados de que aquela mensagem não está mais na caixa postal.

Para evitar esses problemas de implementação, um caso especial de caixa postal foi proposto por Balzer |7|, que é o sistema de portas. Nesse sistema todos os comandos *receive* referentes a uma mesma porta só podem ocorrer num Único processo. O sistema de portas se adequa bem ao modelo de clientes/servidores com um único servidor.

O sistema de 'mailboxes' pode servir também para implementar mensagens de difusão ("broadcasting"). Mensagem de difusão é um sinal que não especifica o receptor. Pode-se definir a mensagem para ser recebida pelo primeiro processo pronto a receber a mensagem ou por todos os processos de algum conjunto pré-estabelecido.

O sistema RIG |6| combina endereçamento explícito e indireto, exigindo a identificação do processo e da mailbox nos mecanismos de comunicação.

Endereçamento funcional: estabelece a conexão de processos em termos da necessidade de prestar ou requisitar serviços, dinamicamente. Nesse tipo de endereçamento, o canal de comunicação é uma entidade nomeada no sistema. A identidade dos processos que usam o canal de comunicação não é visível nem pelo servidor, nem pelo cliente. Esse tipo de endereçamento permite que o canal de comunicação não fique sempre associado a um determinado processo e possa ser transmitido junto com as mensagens. Isto é, ao longo da execução de um processo, o processo adquire a capacidade de usar um canal de comunicação ('capability') dinamicamente, podendo perder esse privilégio

ou não durante sua execução.

Endereçamento funcional tem sido usado como conceito básico em alguns sistemas atuais, tais como em Accent |74|, Demos |8| e em SR |2|. Por exemplo, em DEMOS, os canis de ligação, denominados *link*, são criados pelos processos receptores e cada processo, ao ser criado tem associado a ele, uma tabela que dá quais os canais que ele pode usar para mandar mensagens. O conjunto de 'links' de um processo define seu ambiente. Os 'links' são usados para enviar mensagens, mas também podem ser passados junto com as mensagens assim como duplicados, criados e destruidos. Destruição e duplicação de um link aparecem associados aos comandos de envio de mensagens e causam respectivamente, a destruição do canal após ser usado para passar a mensagem ou a duplicação do canal.

Por exemplo, na figura 1, o processo A manda um pedido de serviço para processo C através do linkl. Link 2 é criado por A. Processo C manda mensagens através do link 2 para processo A e para processo B através do link 3. Link 4 é criado por C, mas é uma cópia do link 2. Processo B então manda mensagens para A através do link 4.

Os canais de comunicação estabelecidos através de endereçamento funcional tem algumas características dependentes de tempo, tais como, o número de processos com acesso a eles pode mudar com o tempo.

Quando os canais de comunicação são estabelecidos através de endereçamento indireto e por endereçamento funcional algumas restrições são feitas ao uso do canal. Por exemplo:

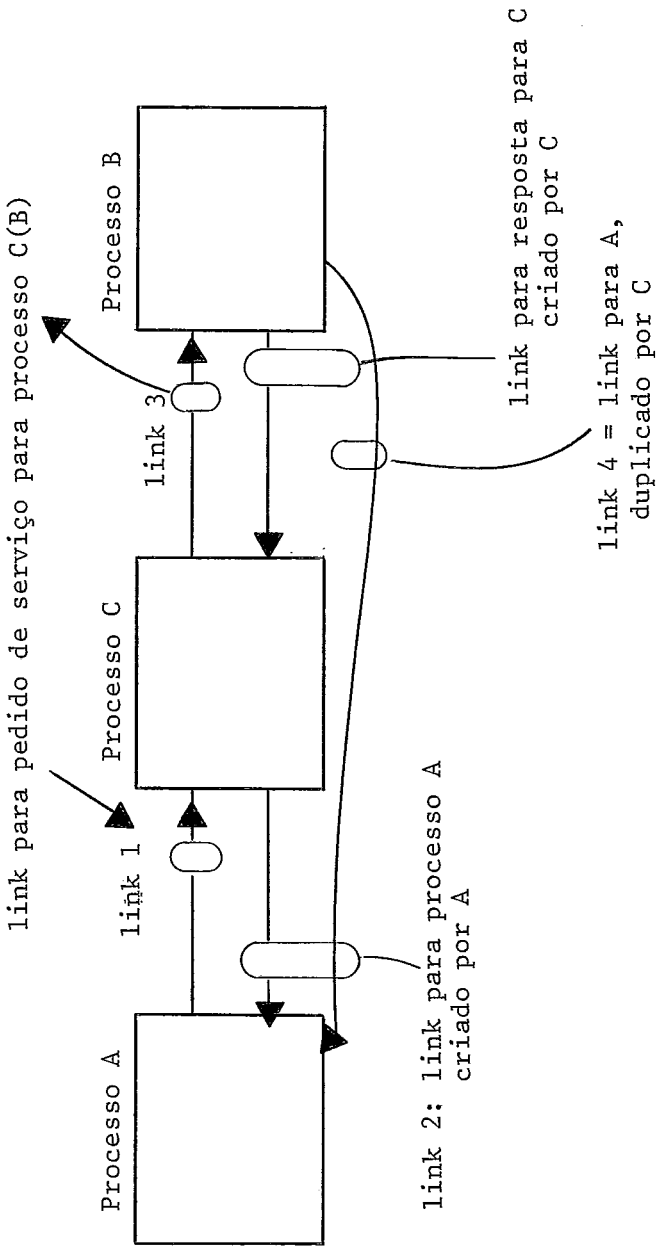

link para pedido de serviço para processo C(B)
Processo A
Processo C
Processo B
link 1
link 3
link 2: link para processo A
criado por A
link para resposta para C
criado por C
link 4 = link para A,
duplicado por C

Figura 1

i) direcionalidade - que consiste em estabelecer se um mesmo processo pode usar o mesmo canal para mandar e enviar mensagens. Por exemplo, no caso de portas, se a mesma porta pode ocorrer num *receive* e num nend no mesmo processo.

Se pelo menos um processo tem acesso a um canal de comunicação para mandar e receber mensagens, o canal é dito bidirecional, caso contrário é dito unidirecional.

A maior parte dos sistemas atuais usa canais unidirecionais, sendo que em TRIX |84| canais bidirecionais são usados.

ii) freqüência de uso - que concerne às limitações na freqüência em que um canal pode ser usado. Por exemplo, em DEMOS |8| e ROSCOE |80|, que usam endereçamento funcional, um canal é criado com o Único objetivo de retornar uma única mensagem e é destruido automaticamente, logo após ser usado.

iii) transferência de direitos - consiste na capacidade de um processo transferir o acesso de um canal de comunicação para outro processo, com ou sem perda por parte do processo que faz a transferência, do acesso ao canal. Transferência de direitos está associada aos canais de comunicação estabelecidos dinamicamente (como as "capabilities" em Andrews |2|).

Ainda com respeito a canais de comunicação, cabe aqui tratarmos de topologia dos canais de comunicação, que é a interrelação dos processos usando o mesmo canal.

Em geral, a topologia pode ser classificada em: um a um, um para muitos, muitos para um e muitos para muitos. Além dessa classificação, a topologia pode ser vista sob o aspecto de se as conexões entre processos são estabelecidos estática ou dinamicamente.

A terminologia aqui é a mesma usada na descrição de PCL por Lesses et al |54| que provê especificação de uma ampla variedade de estruturas topológicas para canais unidirecionais.

O canal simples oferece uma conexão um para um. Uma unica fila guarda todas as mensagens. Endereçamento implícito e explícito só permitem topologia um a um.

Conexões de difusão ("broadcasting") e de leitura múltipla são conexões de um para muitos. Associam muitos receptores para cada remetente de mensagens. A diferença entre essas conexões é que toda mensagem num canal de difusão pode ser recebida pelos receptores enquando que no canal de leitura múltipla, cada mensagem colocada no canal é recebida pelo primeiro processo que tem acesso ao canal para receber mensagens.

A topologia de leitura múltipla é Útil quando se tem um serviço prestado por um conjunto de servidores, cujos desempenhos são equivalentes.

A topologia de difusão pode ser simulada por um conjunto de canais simples,enquanto que leitura múltipla se simulada por canais simples onera o sistema pela necessidade de gerenciamento dos processos servidores.

Conexões muitos para um são obtidas por canais de concentração e de escrita múltipla. Esses dois conceitos são os "duais" dos conceitos de difusão e leitura múltipla respectivamente. Isto é, qualquer mensagem mandada pelos remetentes é re

cebida pelo receptor numa conexão de escrita múltipla. Enquanto que numa conexão de concentração, cada mensagem recebida é a concatenação das mensagens enviadas por cada comando de envio, de cada remetente.

Conexões muitos para muitos são combinações de conexões muitos para um e um para muitos. Assim temos:

- escrita-múltipla/difusão - onde cada mensagem enviada é transmitida para todos os receptores.
- escrita múltipla/leitura múltipla - qualquer mensagem mandada só é recebida pelo primeiro receptor requisitando o canal.
- concentração/difusão - a concatenação das mensagens de cada remetente é transmitida para todos receptores.
- concentração/leitura múltipla - a concatenação das mesagens mandadas por cada remetente é recebida pelo primeiro receptor que usa o canal.

Como dissemos anteriormente, outra maneira de considerar a topologia é se ela é estática ou dinâmica. Numa topologia dinâmica as conexões entre processos podem mudar durante a execução do programa, isto é, os designadores de destino e origem dos comandos de envio e destino podem ser computados em tempo de execução. Numa topologia estática os designadores de destino e origem são fixados no tempo de compilação.

Topologia estática, apesar de amplamente usada apresenta duas desvantagens básicas, a saber:

1 - restringe a comunicação a canais conhecidos ao tempo de compilação.

2 - não libera o canal de comunicação entre os processos, isto é, se um processo eventualmente tem acesso a um canal, esse acesso será permanentemente do processo. (No caso de acesso a arquivos, por exemplo, é preferível alocar canais de comunicação ao arquivo de maneira dinâmica).

É preciso notar que criação dinâmica de processos não implica em topologia dinâmica: novos processos podem ser criados em tempo de execução mas se suas conexões em potencial tiverem sido estabelecidas em tempo de compilação, a topologia é estática. Por outro lado, em linguagens como GYPSY |35|, em que capacidades (capacibilities) de comunicação podem ser passadas como parâmetros entre processos, um conjunto estático de processos pode exibir uma topologia dinâmica.

Topologia dinâmica também pode ser obtida em linguagens que tenham variáveis cujos valores são processos ou monitores, já que o exato valor dessas variáveis num determinado ponto da execução pode não ser necessariamente determinado estaticamente.

A terceira questão semântica colocada sobre as primitivas de comunicação refere-se ao tratamento de falhas. Vamos dar aqui uma breve discussão sobre o assunto já que este por si só já é assunto para um outro trabalho de igual envergadura.

Sistemas de comunicação podem falhar e falhas devem ser evitadas ou então tratadas de modo a não se tornarem catastróficas para o sistema. Assim, podemos considerar duas aborda-

gens no caso de falhas de sistemas: evitá-las ou torerá-las. No primeiro caso, métodos são estudados de tal forma a ,garantir que durante as diversas fases do desenvolvimento do sistema não ocorram erros. Um desses métodos é verificação formal de correção de programas. Na segunda abordagem, admite-se que falhas podem ocorrer e são procuradas maneiras de evitar que essas falhas provoquem danos grandes ou mesmo colapso do sistema.

Vamos dar aqui alguns erros de processamento que podem ocorrer no sistema:

i) Bloqueio perpétuo - causado por um ciclo de processos enviando mensagens uns para os outros ou esperando receber mansagens uns dos outros. Um dos métodos para evitar bloqueio perpétuo é o uso de grafo de bloqueio. Os nós desse grafo são processos e os arcos são dirigidos no sentido do envio de mensagens. Se houver ciclo neste grafo pode haver bloqueio perpétuo. Portanto, evitam-se bloqueios evitando-se ciclos no grafo.

ii) Criação e destruição dinâmicas de processos podem causar falha de comunicação, quando um processo tenta se comunicar com um processo que já não existe. Se as primitivas de troca de mensagens são primitivas bloqueadas, um processo pode ficar bloqueado esperando pelo término de comunicação com processos que foram destruidos, o que não é desejável. Uma providência a ser tomada é notificar os processos da destruição dos processos com os quais estão tentando se comunicar, o que pode ser feito na implementação das primtivas.

iii) Primitivas bloqueadas podem gerar outro tipo de falha de comunicação.Por exemplo, suponha que um processo A envia uma mensagem para o processo B e fica bloqueado esperando por uma mensagem de volta de B. Se o processo B não envia a resposta de volta ao processo A, este ficará bloqueado indefinidamente, esperando pela resposta. Uma forma de atacar esse problema é incluir na implementação das primiticas uma alternativa de desbloqueio ou saída por tempo (time out).

"Time out" permite ao programador decidir qual a atitude a tomar em caso de falhas ocorrendo em tempo de execução, porém pode ser muito difícil de dimensionar o tempo de espera, se o sistema não é voltado para uma aplicação específica.

Linguagens como ADA |81|, GYPSY |35|, MESA |63|, têm mecanismos para tratamento de falhas.

A questão sintática mencionada refere-se ao formato da mensagem e pode ter influência sobre a eficiência no sistema de comunicação entre processos.

Normalmente, em um sistema, processos devem responder a vários tipos de menagens. Uma forma de tratar formatos variáveis de mensagens é tratar mensagens como chamada de função. Nessa sintaxe temos um cabeçalho com a declaração da função (mensagem) e a chamada da função com nome e lista de argumentos diferentes para cada tipo de mensagem.

Quanto ao tamanho da mensagem temos mensagens de tamanho fixo ou mensagens de tamanho variável. A primeira alternativa é de implementação mais fácil embora exija uma escolha para a fixação do tamanho mais apropriado. Ao se fixar o tamanho

da mensagem, normalmente, leva-se em conta o fato de que na sua maioria as mensagens são curtas. (eg. ROSCOE |80|, TOTH |20|). Para mensagens longas, uma alternativa é separar a mensagem em vários segmentos fixos ou criar uma primitiva diferente para mensagens longas.

Mensagens de tamanho variável são mais dificeis de serem implementadas devido aos problemas de armazenamento nos "buffers". Em algumas aplicações, no entanto, podem ser preferiveis a mensagens de tamanho fixo, como, por exemplo, ACCENT |74|, MEDUSA |66| e CLU |56|, que usam mensagens de tamanho variável.

4.2 - Chamada remota de procedimento

As primitivas *send* e *receive* da seção 4.1 são suficientes para programar qualquer tipo de interação de processos usando troca de mensagens. Outra pesposta à demanda de primitivas de comunicação para sistemas distribuidos é a chamada remota de procedimento, que é uma máquina (possivelmente virtual) chamando outra. Em DISTRIBUTED PROCESS |14| essa abordagem é usada. O estudo de chamada remota de procedimento resulta na busca de uma primitiva de comunicação adequada a nível de linguagem para uso em sistemas distribuidos.

A comunicação em sistemas distribuidos pode ser caracterizada pela transferência de informação e controle entre programas autônomos executados em distintas máquinas (virtuais). Chamada remota de procedimento pode ser caracterizada como transferência de controle síncrono em nível de linguagem entre programas em espaços de endereçamento disjuntos, cujo meio primá-

rio de comunicação é um canal. O canal caracteriza o termo 'remota', e assim a comunicação não é feita através de memória compartilhada.

A chamada remota de procedimento é especialemnte adequada para o modelo de clientes/servidores e, porque a chamada remota de procedimento se dá entre processos autônomos, falha em um processo não significa falha do outro. Esse isolamento e deteção de falhas é um problema fundamental em sistemas distribuídos, distinguindo-os: de programas locais, com chamadas locais, cujas falhas podem fazer parar o processo que chama e o que chamou.

Quando chamada remota de procedimento é usada, o cliente interage com o servidor através de um comando de chamada (call). Esse comando tem a forma similar a uma chamada de procedimento usada em programas sequenciais e funciona da seguinte forma:

i) os valores dos argumentos são passados para o servidor referido e o processo solicitante fica bloqueado até que o serviço tenha sido executado e resultados retornados através dos argumentos de saída.

O procedimento remoto pode ser especificado como uma declaração, como no caso de uma declaração de procedimento numa linguagem sequencial, ou como um comando, que pode aparecer em qualquer lugar do programa, como outro comando qualquer.

Em |3|, Andrews e Schneider usam a seguinte notação para exemplificar os dois casos, respectivamente:

- *remote procedure* service (*in*:value-parm-i *out*: result-param)
  corpo do procedimento
  end

- *accept* service (*in*:value parm; *out*:result-parm) → corpo

No primeiro caso, o procedimento service é implementado como um processo. Esse processo (servidor) espera por recebimento de uma mensagem, contendo os valores para os parâmetros de entrada, vindos de um processo solicitante (cliente). Executa o corpo do procedimento e manda uma mensagem de volta para o cliente contendo os valores dos parâmetros de saída.

Uma declaração de procedimento remoto pode ser implementada por um processo cíclico como em |2|, (nesse caso, chamadas para o mesmo procedimento remoto são executadas sequencialmente), ou um novo processo pode ser criado para cada chamada, como em DP |14|. (Nesses casos diferentes instanciações do servidor podem ser executadas concorrentemente, e podem necessitar de sincronização, se tiverem variáveis compartilhadas).

Na segunda abordagem da especificação, o processo servidor (que contém o comando *accepk* service ... no exemplo acima) fica bloqueado até receber uma mensagem, resultande de uma chamada para service, e então executa o corpo usando os valores dos parâmetros de entrada e de variáveis acessíveis no escopo do comando *accept* e, como na primeira abordagem, manda uma mensagem de volta ao cliente e continua sua execução.

Quando a especificação do procedimento remoto é feita como um comando *(accepk* ou outro anãlogo), a chamada remota de procedimento é chamada de "rendez-vous" |81| pelo fato de o cli-

ente e servidores se 'encontrarem' durante a execução do corpo do comando *uccepk* e, depois, cada um seguir o seu caminho.

Rendez-vous apresenta algumas vantagens sobre declaração do procedimento remoto tais como:

- as chamadas (calls) dos clientes podem ser servidas à escolha do servidor.

- o servidor pode obter diferentes resultados para chamadas para o mesmo serviço, usando comando *accept* com corpos diferentes.

- o servidor pode prover mais que um tipo de serviço, em particular, o comando *accept* pode ser combinado com comandos seletivos que fazem com que um servidor espere e selecione um dos serviços solicitados |81, 2|.

Existem mais considerações a serem feitas a respeito de chamada remota de procedimento (por exemplo, o tratamento de exceções e verificação de tipos), mas não cabem ao escopo desse trabalho. Para maiores detalhes, indicamos a tese de doutorado de Nelson |67|.

Para finalizar, as primitivas *send/receive* e chamada remota de procedimento são usadas em linguagens cuja interação entre processos se faz através de troca de mensagens. A escolha entre elas se dá devido à aplicação a que elas estão sendo usadas. Em nível de sistemas operacionais, em geral, as primitivas *send* e *receive* são preferidas. Essa opção é talvez justificada pela grande variadade de comunicação entre processos ba-

seadas em troca de mensagens em sistemas com um ou mais processadores. Ao nível de linguagem, o mecanismo de chamada de procedimento é, em geral, preferido principalmente se essas linguagens já são procedimentais.

Outro aspecto da escolha entre elas é o modelo usado: por exemplo, no modelo de "pipeline", as primitivas *send* e *receive* são preferidas, enquando que no modelo cliente/servidor, chamadas remotas de procedimentos são mais usadas.

CAPÍTULO III

MECANISMOS PARA CRIAÇÃO, COMUNICAÇÃO E SINCRONIZAÇÃO DE PROCESSOS

3.1. *Definição Informal dos Mecanismos para Criação, Comunicação e Sincronização de Processos*

Vamos definir um conjunto de mecanismos para comunicação e sincronização usados em programação concorrente de sistemas distribuídos baseada em troca de mensagens.

A concepção dos mecanismos, usando portas de entrada e saída como receptáculos de mensagens, visa à modularidade dos processos paralelos assim como à adaptação dos mesmos a qualquer sistema distribuido cuja arquitetura consistia de: (a) um número fixo de módulos autônomos de processamento; (b) um sistema de interconexão que permita troca de informação entre esses módulos.

Daremos aqui somente idéia da sintaxe dos mecanismos visando aos aspectos semânticos dos mesmos sem nos prendermos a detalhes concretos da sintaxe.

A abstração desses módulos de processamento será denominada por nade (Liskov, em |55| usa um conceito semelhante e denomina a abstração de tais módulos por *guardian*) e sua especificação tem a forma:

nade <nome do nó>
  input ports: <enumeração das portas de entrada do nó>
  output ports:<enumeração das portas de saída do nó>

import:<lista das portas importadas pelo nó>

export:<lista das portas exportadas pelo nó>

local processes:<lista dos nomes dos processos locais ao nó>

<declaração de constantes do tipo porta com seus valores>

<declaração dos processos locais ao nó>

<ativação inicial de processos e inicialização de variáveis do nó>

Mais adiante daremos detalhes sobre as declarações da especificação do nó.

A abstração de um programa usando os mecanismos aqui propostos teria a seguinte sintaxe:

program <nome do programa>

global processes: <lista dos nomes dos processos globais do programa>

nudea: <lista dos nomes dos nós do programa>

all input ports: <enumeração de todas as portas de entrada>

all output ports: <enumeração de todas as portas de saída>

<declarações dos processos globais>

<declarações dos nós como acima>

all input ports e *all output ports* são declarações de tipos e enumeram todas as portas de entrada e de saída respectivamente. (União discriminada dos tipos *input ports* (*output ports*) que ocorrem no programa).

Note-se que estamos supondo que não usamos o mesmo nome para portas de nós diferentes, para simplificar a notação.

Caso contrário os nomes das portas seriam acompanhados pelos nomes dos respectivos nós.

O tipo *input ports (output pohkn)* é declarado por cada nó do sistema como uma enumeração. Portando cada nó tem um número fixo de portas e o conjunto de portas dos nós são mutuamente disjuntos. As portas de um nó serão usadas exclusivamente por processos ativos no nó, a não ser que sejam declaradas de exportação, como veremos adiante. Variáveis do tipo porta de entrada são declaradas através de declarações de variáveis *input ports* e só ocorrem, localmente, nos corpos dos processos.

Processos poderão ser declarados dentro ou fora de nós. Processos declarados fora de nós são globais ao sistema, podendo ser ativados por qualquer nó do sistema, enquanto que processos declarados dentro de um nó são locais a este nó, só podendo ser ativados dentro deles.

Por exemplo,

Exemplo 1

```
program P
  global processes: Q |lista de processos globais|
  nodes N1,N2         |lista dos nomes dos nós|
  all input ports: I1,I2,E
  all oukpuk ports: O,F,S
  process Q (pb: all input ports)
   "
                      |declaração do corpo de Q|
   "
```

```
nade N1
  input ports: I1,I2
  output ports: O
  import: -
  export: -
  local pnaceaaea: PR |lista dos nomes de processos locais
                      a N1|
  pnoceaaea PR  (pe: input ports)
"
"                    |declaração do corpo de PR|
"
  6egin
    create PR (11);    |ativação de processo declaração em N1|
    create Q (I2)      |ativação de processo declaração glo-
                         balmente|
  end N1

nade N2
  input ports: E
  output pots: -
  import: -
  export: -
  local processes: -
  begin
    create Q(E) |ativação em N2 de processo declaração glo-
                  balmente|
  end N2
```

No exemplo anterior e nos a seguir fazemos comentários entre barras |,|.

No exemplo anterior duas encarnações do processo Q poderão estar ativas (uma em N1 e outra em N2). O processo PR só pode ser ativado em N1, já que foi declarado nesse nó.

Chamamos de encarnação de um processo à instanciação da declaração do processo quando substituimos os parâmetros formais do cabeçalho de sua declaração por nomes de portas. Sobre ativação de encarnações falaremos quando tratarmos do comando *create*.

A declaração de um processo associa um código de um programa sequencial a um nome. Daremos aqui a sintaxe dos comandos aqui propostos, a descrição da semântica será feita de maneira informal, sendo sua formalização dada no próximo capítulo.

A sintaxe da declaração do processo é:

process <nome do processo> {<<lista de parâmetros de portas de saída>:<tipo das portas>; <lista de parâmetros de portas de entrada>:<tipo das portas>>} <programa sequencial>

Os tipos das portas que aparecem no cabeçalho são *input ports* ou *output ports* quando o processo é declarado dentro de um nó, ou *all input ports* ou *all output ponta* quando o processo é declarado fora de nós (confrontar com exemplo 1).

Além dos parâmetros de portas que ocorrem no cabeçalho, o processo pode também declarar variáveis do tipo porta de entrada, localmente. (Note que uma variável declarada localmente como porta de saída não terá utilidade, já que, como veremos adiante, as operações sobre portas de saída: ligações, criações de processos e envio de mensagens, dentro de um pro-

cesso não usam variáveis locais ao processo). O processo porém, não pode declarar novas portas, isto é, dentro do processo não será permitido haver declarações de tipo *hnpuk ports* ou *output ports*.

Os valores das variãveis declaradas localmente num processo como sendo do tipo *hnpuk ports* varrem os nomes das portas de entrada enumeradas na declaração desse tipo ou na declaração de portas de importação que ocorrem no nó onde o processo é criado.

Um processo pode ter acesso a portas de entrada de outro nó dinamicamente (através da opção *reply to*, que veremos mais adiante). Assim variáveis declaradas localmente a um processo podem também ser do tipo *all hnpuk ponkn*.

Por exernplo:

Exemplo 2:

```
nade N
  hnpuk ports: E1,E2
  output ports: 01,02
  local processes: P
  processe P (ps: vukpuk ports; pe: hnpuk ports)
    var p: hnpuk ponkn,
      rep: all hnpuk ports
  begin
      send m to ps;
      receive n from pe reply to rep;
      p:= E1
  end P
```

```
begin
  cneake P(O1,E1)
end N
```

(Aqui e nos demais exemplos omitiremos as declarações cujas listas a que se referem são vazias).

Sobre os comandos *nend* e *receive* falaremos mais adiante.

A ativação de uma encarnação de um processso é feita através do comando

cneake <nome do processo> (<lista de nomes de portas>).

Os nomes de processo que aparecem na declaração do processo e no comando *cneake* são os mesmos. Os nomes das portas que ocorrem na declaração dos processos são parâmetros formais, tais como os que aparecem em declarações de procedimentos. A associação dos nomes de portas aos parâmetros é feita através do comando *cneake*, tal como numa chamada de procedimento, onde os parâmetros são passados por valor. As portas que ocorrem no comando *cneake* são nomes de portas declaradas no nó onde o processo está sendo ativado; portanto são constantes.

O comando *cneake* pode ocorrer tanto na parte de ativação do nó como dentro da declaração de um processo, possibilitando assim criação dinâmica de processos.

No exemplo a seguir, na fase de inicialização de N, uma encarnação do processo P é ativada através de *cneake* P(O). Durante a execução dessa encarnação, os processos Q e G são ativados através de *cneake* Q(A) e *cneake* G(B) respectivamente. Note

que G não é declarado em N, mas as portas usadas tanto nas ativações de P, Q e G são declaradas em N.

Exemplo 3

```
node N
  input ports: A
  output ports: O,B
  local processes: P,Q
  processes P (p:output ports)
    beghn
   "
   "
   "
        create Q(A);
        create G(B);
   "
   "
   end

  process Q (pb: input ports)
    begin
   "
   "
    end Q

  begin
    create P(O)
  end N
```

Poderíamos ainda, ter duas encarnações no mesmo processo ativas no mesmo nó, por exemplo:

```
node M
  input ports: I,C
  output ports: F,S
  local processes:P1

  process   P1 (pls: outrput ports: ple: input ports)
    begin
    "
    "          |declaração do corpo de P1|
    end P1

  begin
    create P1(I,F);
    create P1(S,I);
    create Q(I);
    create Q(C)
  end M
```

Nesse exemplo, Q é um processo declarado globalmente e tem duas encarnações ativadas em M.

Notamos que uma mesma porta pode ser usada por encarnações de um mesmo processo (a porta I no exemplo anterior está sendo usada por duas encarnações do processo P1) como pode ser usada por encarnações de processos diferentes (a porta I no exemplo anterior está sendo usada para ativar encarnações do processo P1 e do processo Q). Restrições sobre o uso da

mesma porta por encarnações ativas serão impostas posteriormente.

Vamos chamar um processo P de *ancestral* do processo Q se:

- P ativa Q através de *cheake* Q(...) dentro da declaração de P

OU

- P ativa Q1 e Q1 é ancestral de Q.

Um processo globalmente declarado não pode ser ancestral de nenhum processo (vide restrições na seção seguinte, para justificativa de tal imposição).

Suporemos que o sistema onde o conjunto .de mecanismos está sendo usado dispõe de mecanismos próprios de inicialização dos diversos nós e que estes serão sempre executados.

As encarnações ativas de processos se comunicam entre si somente através de menç'agens. As mensagens são colocadas em portas de saída e recebidas através de portas de entrada.

O comando usado para enviar mensagens é um comando de envio assíncrono, cuja sintaxe é:

send <nome de mensagem> *to* <parâmetro de porta de saída> {*reply to* <porta de entrada>)

Aqui e na descrição dos demais comandos usaremos chaves {,} para indicar que o comando entre elas é opcional.

A mensagem valor <nome de mensagem> é colocada na porta valor <parâmetro de porta de saída>. Se a essa mensagem quisermos associar um nome de porta de entrada, usamos a opção *reply ko*, sobre a qual daremos mais detalhes adiante.

O comando usado para recebimento de mensagens é um comando de comunicação síncrona, isto é, uma recepção bloqueada. Sua sintaxe é:

receive <nome de mensagem> {*from* <parâmetro de porta de entrada> {*reply to* <de porta de entrada>) {*when time ouk* <expressão> *to* <comando>}

Não vamos nos deter na sintaxe das mensagens usadas nos comandos *receive* e *nend*, mas supomos que os objetos usados nos dois comandos no lugar de <nome de mensagem> tem o mesmo tipo.

A primitiva *receive* bloqueia o processo até que alguma mensagem seja colocada na porta de entrada, a não ser que a opção *when time auk* seja usada, quando o bloqueio dura no máximo o tempo especificado em <expressão>. Se, ao final desse tempo, nenhuma mensagem tiver sido colocada na porta, o comando <comando> será executado.

Se P é um processo em cuja declaração ocorre um comando *nend* ou *receive* usando opção *reply to* <porta de entrada>, então <porta de entrada> é:

- variável local de P declarada como (*all*) *inpuk ports*, e no caso da opção estar sendo usada no comando *nend* também pode ser:
- parâmetro formal do tipo (*all*)*input ports* que ocorre no cabeçalho de P,
- constante, do tipo *input ports*, declarada no nó onde P é declarado ou do tipo *all inpuk ports*, quando P é declarado globalmente, ou
- constante ocorrendo na lista de portas importadas pelo nó onde P é declarado.

No exemplo do fim dessa seção ilustramos o uso de *reply to*.

Uma mensagem pode chegar a um processo só se existir uma ligação entre a porta de saída onde a mensagem foi colocada e a porta de entrada através da qual a mensagem será recebida. Uma ligação entre portas é estabelecida através da execução dos comandos link's cujas sintaxes são:

Rinh in <porta de saída> *to* <porta de entrada>

e

Rinh <parâmetro de porta de saída> *to* <porta de entrada>

O comando *Rinh in* só pode aparecer na parte da ativação de processos na inicialização do nó. Ele estabelece enlaces iniciais entre encarnações de processos enquanto que o comando *Rinh* só pode ocorrer dentro da declaração de processos, para estabelecer ou refazer ligações das portas usadas pelas encarnações desses processos, dinamicamente.

No comando *link in* <porta de saída> e <porta de entrada>, são respectivamente, constantes do tipo *output ports* e do tipo *input ports* declaradas no nó em cuja inicialização o comando *Rinh in* ocorre, sendo que, <porta de entrada> ainda pode ser uma porta declarada como de importação (*import*) nesse mesmo nó.

As portas de saída que ocorrem em comandos *link in* devem também ocorrer em listas de portas de comandos *create* que aparecem na parte de ativação do nó, i.e., não é o caso de termos *Rinh in* P *to* Q e *P* não ocorre em nenhuma lista de portas de comandos *create* na ativação inicial do nó.

A execução de *Rhnh in* é bloqueada, caso

1. a porta de saída a que se refere estiver sendo usada por alguma encarnação de processos ativa no nó ou
2. se essa porta estiver ligada a outra porta de entrada e nem todas mensagens colocadas na porta de saída tiverem sido transmitidas.

No comando *link*, <parâmetro de porta de saída> é um parâmetro formal do tipo (*all*) *output ports* que ocorre no cabeçalho da declaração do processo onde *link* ocorre e <porta de entrada> pode ser:

- variável do tipo (*all*) *input ports* declarada no processo onde o *Rhnh* ocorre, ou
- nome de porta de entrada do nó onde o processo no qual *Rhnh* ocorre é ativado, ou importada por esse nó.

A execução do *link* bloqueia a encarnação ativa do processo onde ele ocorre, se a porta a que ele se refere estiver ligada a uma porta de entrada, diferente daquela a que o comando se refere se houver mensagem na porta de saída ainda a ser transmitida.

Uma ligação entre portas feita pela execução dos comandos *Rinh* ou *Rinh in* destrói automaticamente qualquer outra ligação usando a mesma porta de saída. Logo, durante a execução de um programa uma porta de saída estará ligada a no máximo uma porta de entrada.

A necessidade de se ter acesso a portas externar ao nó pode ser atendida como em MODULA |79|. As portas de entrada

de um nó acessiveis diretamente a processos de outros nós deverão aparecer numa declaração *export* no nó possuidor da porta e numa declaração *import* nos nós com acesso à porta. Essas declarações têm a seguinte sintaxe:

export <lista de portas de entrada>

import <nó origem 1>.<lista de portas de entrada>
"
"
"
<nó origem n>.<lista de portas de entrada>

Uma porta de entrada recebida através da opção *reply to* de um *receive* pode ser utilizada somente após a execução de um comando *link* que liga uma porta de saída a essa porta de entrada. Logo, as portas enviadas através da opção *reply to* de um *nend* podem ser ligadas dinamicamente. Assim, o uso da opção *reply to* permite que uma porta de saída seja ligada a uma porta de entrada de outro nó, sem que haja necessidade das declarações *import/export* nos respectivos nós.

Por exemplo:

Exemplo 4

```
node N1
  inpuk ports: PE1
  output ports: PS1
  import: N2.PE2
  local processes: P
  process P(pfs: output ports; pfe: inpuk ports)
    var m: mensagem,
    repl: mensagem 1
    begin
     "          ]atribuição de valor a m|
      aend m to pfs reply to pfe;
      receive repl from pfe
    end P

  begin     ]ativação de N1|
    link in PS1 to PE2;
    create P (PS1,PE1)
  end N1

node N2
  input ports: PE2
  output ports: PS2
  export: PE2
  local phoceanea : PR
  process PR (pfs2: output ponkn ; pfe2: inpuk ports)
    var n: mensagem,
      resp: mensagem 1,
    replyport: all inpuk ports
```

```
    begin
      receive n from pfe2  reply to replyport;
      ----------  [atribuição de valor a resp|
      Rhnh pfs2 to replyport;
      send resp to pfs2
    end PR

  beghn     |ativação do nó N2|
    create PR (PS2,PE2)
  end N2    |fim da ativação de N2|
```

No exemplo acima replyport toma valor de PE1, que é o valor atual do parâmetro pfe usado no *reply to* do comando *aend* do processo P. PE1 é uma porta declarada em N1 e é usada para ser ligada a PS2, que é uma porta de N2 e nesse caso não há a declaração *import* N1.PE1/*export* PE1 em N2 e N1 respectivamente.

A declaração *import/export* fixa a visibilidade de portas de um nó por outro nó no tempo de compilação, e com a opção *reply to*, a visibilidade pode ser ampliada dinamicamente em tempo de execução.

Daremos agora um exemplo mais detalhado.

Exemplo 5

```
program exemplo       |nome do programa|
  global processes: PG  |PG é o nome de um processo global|
  nodes: N1,N2,N3     |lista dos nomes dos nós|
  all output ports: N1:S,PS2,PS5,PS6;
                    N2:OP8;        |todas portas de saída|
                    N3:PS10,PS11
```

```
all input ports: N1:PE1,PE3,PE4;
                 N2;IP7;        |todas portas de entrada|
                 N3:PEN9;
process PG(ps: all output ports) |declaração do processo glo
                                  bal|
  var porta: all input ports
  begin
   "
   "
   "
  end PG  |fim da declaração de PG|

nade N1   |declaração do nó N1|
  input ports: PE1,PE3,PE4     [portas de entrada de N1|
  output ports: S,PS2,PS5,PS6  |portas de saída de N1|
  import: N2.IP7   Ideclaração de importação|
  const: E1 = PE1  |declaração e inicialização de constantes
                    do tipo porta|
  local processes: PR1, PR2   (processos declarador em N1|
  process PR1 (pf2: output ports; pf1: input ports)` [declara
                                                  ção de PR1|
    var m,n; mensagem
    begin
      link pf2 to PE3;  |liga a ponta associada a pf2 com
                        . PE3, que é uma porta de N1|
      aend m to pf2; [coloca mensagem m na porta associada a
                      pf2|
      receive n from pf1  |recebe mensagem n na porta associa
                           da da pf1|
    end PR1    |fim do processo PR1|
```

```
|declaração de PR2|
    process PR2 (pf5,pf6: output ports; pf3,pf4: input ports)
      var req,rep: mensagem
      begin
        receive req from pf3;
        nend req to pf5 reply to pf4;  |coloca req e a porta
              associada a pf4 na porta de saída associada a pf5|
        receive rep from pf4;
        nend rep to pf6
      end PR2     |fim do processo PR2|

[ativação dos processos e inicialização de variáveis globais de
 N1|

    6egin
      link in PS5 to IP7; |liga PS5 à porta IP7 importada de
                           N2|
      link in PS6 to E1;  |liga duas portas do mesmo nó|
      create PG(S);       |ativa processo global PG associando
                           a porta S ao parâmetro ps|
      create PR1 (PS2,E1); |ativa PR1 associando E1 a pf1 e PS2
                            a pf2|
      create PR2 (PS5,PS6,PE3,PE4)
    end  N1   |fim da ativação dos processos pelo nó N1|
```

```
node N2
  hnpuk ports: IP7
  output ports: OP8
  export: IP7 |declaração de portas exportadas para outros
               nós|
  local processes: PR3
  process PR3 (pf8: output ports; pf7: input ports)
    var msg: record of ...
      flportl, flport2, varport: all input ports |variáveis lo
                  cais ao processo que podem tomar qualquer
                   porta de entrada do programa como valor|

    begin
      receive msg from pf7 reply to varport |recebe um valor
                  para msg e um para varport através da porta
                   associada a pf7|
      cabe msg-sell of
                  "
                  "
              flportl:= varport
                  "
                  "
        end  |fim de opção 1 de case|

     opção 2: begin
                  "
                  "
                  link pf8 to flport2; |liga a porta associada
                        a pf8 ao valor associado a flport2,que
                        é uma porta qualquer do programa|
```

```
                    aend msg to pf8 reply to varport  |retransmi
                                te a mensagem e a porta recebidas pa-
                                 ra a porta associada a pf8|
    end     |fim da opção 2 do case|
  end PR3   |fim da declaração do processo PR3|

|ativação de processos|

    begin
      create PR3 (OP8,IP7)   |associa IP7 a pf7 e OP8 a pf8,
                                ativando PR3|
    end N2     |fim da ativação|

  node N8
    input ports: PEN9
    output ports: PS10, PS11
    import: N2.IP7
    const: P9 = PEN9;
           S10 = PS10;
           S11 = PS11
    local processes: PR4,PR5
    process PR4 (pf10: output ports)
      var msg: mensagem

      begin
        create PR5 (PS11,PEN9); |ativa PR5 dinamicamente com
                                  portas declaradas no nó|
        aend msg to pf10 replyto P9  /envia msg e P9 para a
                                       porta associada à pf10|
      end PR4  |fim do processo PR4|
```

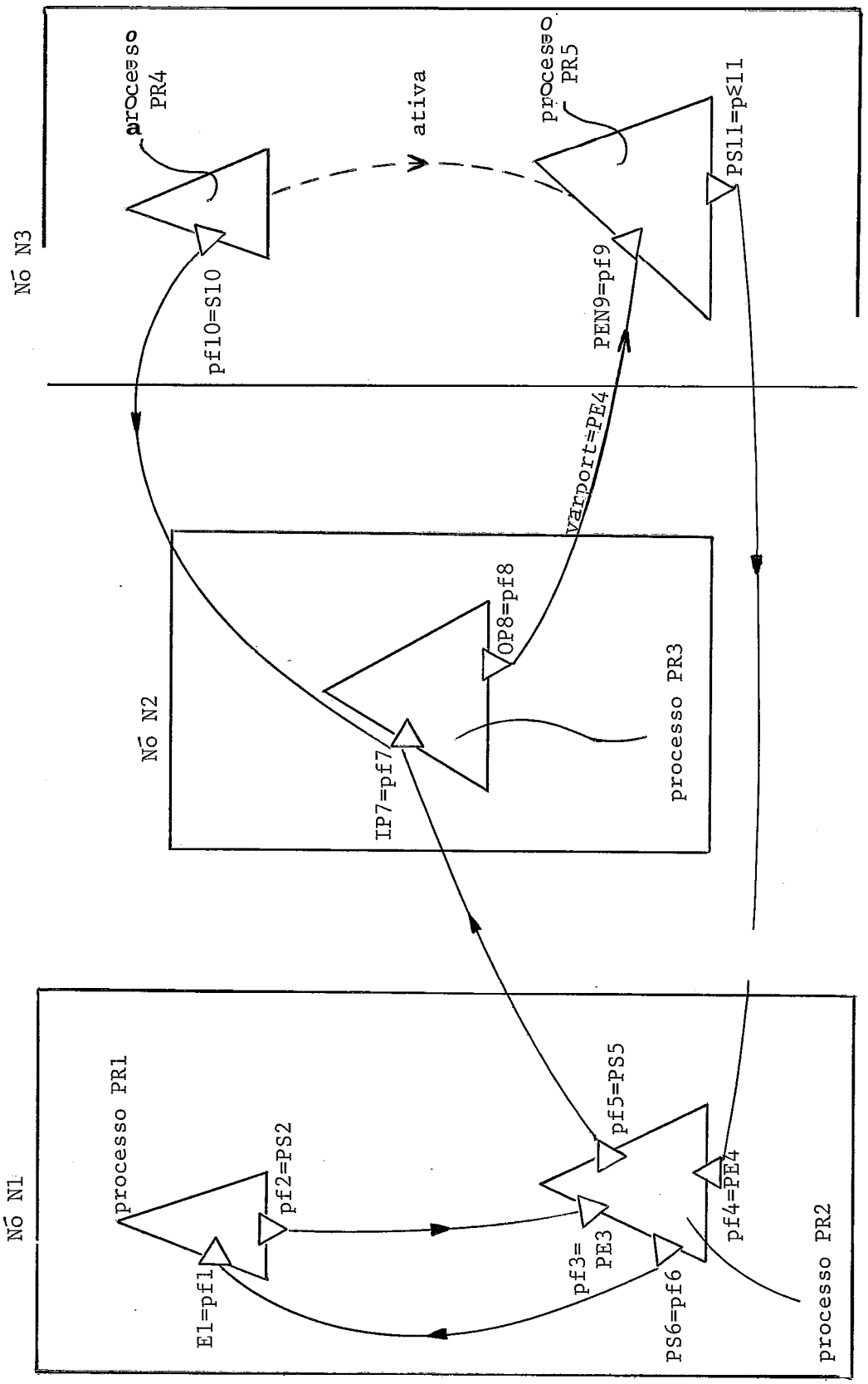
```
    process PR5 (pfll: output ports; pf9: input ports)
      van msg, reply: mensagem,
          repl: all input ports

      begin
        receive msg from pf9 reply repl;
        link pfll to repl; |ligação dinâmica de portas: a as-
              saciada à pfll e a recebida através de reply to
              no receive|
        aend reply to pfll
      end PR5    |fim da declaração de PR5|

|ativação de processos|

    begin
      Rhnh P10 to IP7;
      create PR4(S10)
    end N3   |fim da ativação de processos feita na fase de ati-
              vação|
  end exemplo
```

Damos a seguir uma possível configuração das ligações entre os processos.


Nó N1
processo PR1
E1=pf1
pf2=PS2
Nó N3
processo PR4
pf10=S10
ativa
processo PR5
PS11=p≤11
PEN9=pf9
varport=PE4
Nó N2
OP8=pf8
IP7=pf7
processo PR3
pf5=PS5
pf3= PE3
PS6=pf6
pf4=PE4
processo PR2

A descrição informal dos mecanismos aqui feita dá a informação necessária para a definição da sintaxe abstrata dos mesmos usada na especificação da semântica operacional através da linguagem de definição de Viena (VDL), detalhada em Wegner |85|. Esta definição formal nós não daremos aqui já que o detalhamento da definição da sintaxe de cada comando descrito anteriormente iria sobrecarregar em demasia a notação, e não pretendemos nos prender a detalhes formais sacrificando a clareza da exposição. Daremos, no entanto, os seletores e predicados associados a cada comando, usados na construção da sintaxe abstrata. O significado pretendido dos predicados associados a cada comando é que o predicado será satisfeito somente pelo comando ao qual ele está associado e identifica a forma geral do comando. Os seletores, por sua vez, dão os componentes básicos de cada comando.

Daremos a seguir uma lista dos comandos com o predicado e seletores associados a cada um deles.

comando: *create* <nome de processo>(<lista de nomes de portas>)

predicado: is-create

seletores: nome-proc, lista-arg

comando: aend <nome de mensagem> *to* <parâmetro de porta de saída>

predicado: is-send

seletores: s-mensagem, s-porta-saída

comando: *aend* <nome de mensagem> *to* <parâmetro de porta de saída>
reply to <nome de porta de entrada>
predicado: is-send-reply
seletores: sr-mensagem, sr-porta-saída, sr-porta-reply

comando: *receive* <nome de mensagem> *from* <parâmetro de porta de entrada>
predicado: is-receive
seletores: r-mensagem, r-entrada

comando: *receive* <nome de mensagem> *from* <parâmetro de porta de entrada> *reply to* <nome de porta de entrada>
predicado: is-receive-reply
seletores: rr-mensagem, rr-porta-entrada, rr-porta-reply

comando: *receive* <nome de mensagem> *from* <parâmetro de porta de entrada> *when time out* <expressão> *do* <comando>
predicado: is-receive-timeout
seletores: rt-mensagem, rt-porta-entrada, rt-tempo-espera, rt-saída-por-tempo

comando: *receive* <nome de mensagem> *from* <parâmetro de porta de entrada> *reply to* <nome de porta de entrada>
when time out <expressão> *do* <comando>
predicado: is-receive-reply-timeout
seletores: rrt-mensagem, rrt-porta-entrada, rrt-porta-reply, rrt-tempo-espera, rrt-saída-por-tempo

comando: *link in* <nome de porta de saída> *to* <nome de porta de entrada>
predicado: is-link in
seletores: porta-origem, porta-destino

comando: *link* <parâmetro de porta de saída> *to* <porta de entrada>
predicado: is-link
seletores: 1-porta-origem, 1-porta-destino

A escolha dos nomes dos predicados e seletores foi feita tal que não precisemos nos deter em detalhes sobre suas aplicações. Por exemplo, para qualquer comando c, is-create (c) se e só se o identificador de c é *create*.

nome-proc (*create* P(p1,...,pn)) = P
lista-arg (*create* P(p1,...,pn) = (p1,...,pn)
rt-saída-por-tempo (*receive* n *from* p when *time out* t do S) = S

Não vamos mencionar aqui os componentes dos construtores da sintaxe das declarações de processos, nós e programa. Faremos, no entanto, uso de alguns deles para simplificar a descrição da semântica operacional. Assim vamos supor que as listas de parâmetros formais de portas de entrada e de saída que ocorrem no cabeçalho da declaração de um processo podem ser acessadas através dos seletores lista-par-porta-entrada e lista-par-porta-saída, respectivamente.

3.2. *Restrições Impostas à Sintaxe e Recomendações ao uso doa Mecanismos Devido a Razões de Ordem Semântica.*

Listaremos aqui restrições que impomos à sintaxe e algumas recomendações para o uso dos mecanismos aqui propostos. A justificativa para tais restrições têm origem nas características semânticas que queremos que os mecanismos apresentem, assim como disciplinar o uso dos mecanismos a fim de evitar que ocorram certas situações indesejáveis durante a execução dos programas usando os mesmos.

I - Restrições Referentes a Ativação de Encarnações de Processos.

I.1. Uma porta pode ser usada na ativação de dois ou mais processos num mesmo nó, porém o acesso a essa porta é mutuamente exclusivo, ficando o processo que executa, digamos *cheake* Q(A,B), bloqueado até que as portas A e B sejam liberadas pelas encarnações que as usam.

Permitir que duas encarnações de processe tenham acesso à mesma porta, sem exclusividade mútua pode levar a situações desagradáveis, tais como por exemplo:

Exemplo 6

Suponha que tenham sido executadas sem bloqueio *create* P(A) e *create* Q(A)

```
onde  process P (pf: output ports)
          "
          "
        begin
          repeat forever
            begin
              aend n to pf;
              Rinh pf to PE
        end
    end P

  process Q (qf: output ports)
     "
     "
    begin
      repeat forever
        begin
         Rinh qf to PE';
         send m to qf
        end
  end Q
```

PE, PE' declaradas no nó onde as encarnações P(A) e Q(A) estão ativas.

Então, dependendo das velocidades de P(A) e Q(A) as mensagens mandadas por P(A), por exemplo, podem ser envidas para PE' ou PE, sem controle do programador. Similarmente, para as mensagens mandadas por Q(A).

1.2. Não há criação recursiva de encarnações de processos.

Essa restrição impede uma situação de bloqueio perpétuo, como por exemplo:

Exemplo 7

```
process P (pf: input ports)
  begin
    create P(A)
  end P
```

onde A é porta declarada no mesmo nó onde P é declarado.

A primeira encarnação ativa de P usando A, P(A) fica bloqueado a espera da liberação por ela mesma da porta A.

Notaremos aqui que o uso da mesma porta na ativação de dois processos diferentes restringe o paralelismo entre eles,, embora opção pode ser, em alguns casos, desejável, já que o número de portas do nó é fixo. (Se não houvesse essa flexibilidade do uso de porta, o número máximo de encarnações ativas durante a execução do programa seria limitado pelo número de portas dos nós.

1.3. Um processo declarado globalmente não é ancestral de nenhum processo.

Caso contrário haveria necessidade de um gerenciamento global de todas as portas de todos os nós, o que sacrificaria a modularidade dos nós.

Por exemplo:

Exemplo 8

Suponha que P é declarado globalmente como

```
process P (x: all input ports)
  begin
    "
    "
    "
    create Q(A)
    "
    "
    "
  end P
```

A porta A deveria ser do tipo *all input ports* ou *all output ports* para poder ocorrer em P. Isso porém iria permitir que quando uma encarnação de P fosse ativada dentro de um nó, digamos N, que não possui a porta A, N teria acesso à porta A, para poder colocar ou receber mensagens, ou estabelecer ligações, através da encarnação de Q ativa em N. Além disso ao tentar executar *create* Q(A) essa encarnação ativa de P teria que ter informação sobre o estado da disponibilidade de A, isto é, se A estava sendo usada por outra encarnação ativa de processo. Isso acarretaria num gerenciamento global das portas, terminando com a autonomia dos nós.

Não há, no entanto, restrição sobre um processo declarado num nó ser ancestral de um processo declarado globalmente.

II - Restrições Sobre Comandos de Recebimento e Envio de Mensagens

II.1. O uso de parâmetros formais nos comandos de recebimento e envio de mensagens, mais as restrições sobre ativação de encarnações de processos, mantêm a modularidade de cada nó e processo. Isto é, somente portas de nó podem ser usadas na ativação de processos no nó e somente a encarnação usando uma porta pode "agir" sobre a porta", isto é, fazer ligações com a porta, receber mensagens da porta ou enviar mensagens pana a porta.

11.2. A opção de saída por tempo (*when time out*) associada ao comando de recebimento de mensagens permite detetar falhas no sistema de transmissão de mensagens, assim como evita que o processo receptor fique bloqueado por tempo indeterminado a espera de uma mensagem que não chega, quer por falha do sistema ou por erro do programador (por exemplo, o processo remetente não liga a ponta de saída à porta referenciada no *receive*).

Contudo, pode ocorrer perda de mensagem quando, por exemplo, a mensagem for transmitida exatamente na hora em que o comando opcional, <comando>, começar a ser executado. Em geral, nada pode ser feito para evitar esse tipo 'de problema. Aconselha-se, no entanto, nos casos em que o processo remetente queira se certificar do recebimento ou não de uma determinada mensagem, a bloquear o processo remetente através de um *receive* que aguarda uma resposta do processo, para o qual foi en

viada aquela mensagem, no caso de ter havido saída por tempo.

Por exemplo:

Exemplo 9

```
process P1 (p: output ports; q: input ports)
   "
   "
  beghn
    Rinh p to P;
    nend m to p;
    receive resp from q;
    "
    "
  end

process P2 (r: output ports; s: input ports)
  "
  begin
  "
    Rinh r to  x;
    receive n from s when time out t do
                  beghn
                    repl:= failure;
                    send repl to r;
                    "
                    "
                    "
                  end
```

```
if (repl = failure) then S1
                    else begin
                           repl = ok;
                           send repl to r
                           "
                           "
                           "
                           end
  end P2
```

onde: P é uma porta de entrada do nó N, onde P2 está declarado e as encarnações de P2 em N são ativadas usando P no lugar correspondente a s.

X é uma porta de entrada do nó onde P1 está declarado e as encarnações de P1 são ativadas usando X no lugar de q.

Note que, neste caso, o paralelismo entre P1 e P2 fica prejudicado.

III. Restrições Impostas aos Comandos de Ligação entre Portas

III.1. Ligações feitas na fase inicial do nó.

O comando *link in* é bloqueado, caso a porta de saída não esteja livre, isto é, esteja sendo usada por alguma encarnação ou existam mensagens ainda a serem transmitidas nessa porta de saída para outro destino.

Esta restrição evita que uma ligação entre portas seja desfeita indevidamente.

Por exemplo:

Exemplo 10

```
  "
  "
  "
Rinh in A to X;
create P(A);
Rinh in A to Y
  "
  "
  "
```

onde o processo P "deseja" que as mensagens colocadas na porta A sejam enviadas para a porta X.

Se não houvesse bloqueio no *Rinh in*, talvez, durante a execução da encarnação do processo P, a ligação entre A e X poderia ser desfeita por *Rinh in* A to Y. Com o bloqueio, *Rinh in* A *to* Y só será executado após P liberar a porta A e todas as mensagens colocadas pela encarnação P(A) na porta A terem sido transmitidas para a porta X.

Recomendação 1:

Uma boa política de programação será usar os comandos *Rinh in* antes dos *create* que têm porta em comum, já que a ativação de processos pode ficar bloqueada e um processo ativo não ter suas portas ligadas por causa do bloqueio decorrente da criação de outros processos.

Isto é, considere o exemplo 11 abaixo:

Exemplo 11

```
  "
  "
  "
create Q(A);
cheake P(A);
link in A to X
  "
  "
  "
```

onde o processo Q é declarado por:

```
process Q(f:...)
  "
  "
  "
  send n to f
```

Neste exemplo o comando *create* P(A) só será executado após a porta A ser liberada pela encarnação do processo Q, ativada por *create* Q(A). No processo Q uma mensagem será colocada na porta A, porém essa mensagem só será enviada após a encarnação do processo P liberar a porta A, o que pode não ocorrer nunca (P por exemplo, pode ser cíclico) ou, na melhor das hipóteses, causará ineficiência na transmissão de mensagens.

Recomendação 2:

Se a mesma porta de saída for usada por duas encarnações diferentes de processo, recomenda-se que as ligações entre essa porta e as de entrada não sejam feitas na fase de inicialização do nó, mas sim, dentro dos processos que as usam. Essa recomendação evita que ligações desejadas por uma encarnação de processo sejam desfeitas por outra encarnação de processo.

Por exemplo:

Exemplo 12

```
  "
  "
  "
Rinh in PS to PE;
create Q(PS');
create P(PS);
  "
  "
```

onde

```
process Q (pf: ...)          process P1 (pf': ...)
    "                   e        "
    "                            "
    "                            "

  create P1(PS)                  link pf' to PE
    "                              "
    "                              "
```

A ligação de PS a PE, em princípio esperada pelo processo P pode ser desfeita pelo processo P1 através de *Rhnh* pf' *to* PE (já que o valor de pf' é PS), se *create* P1(PS) for executado antes de *cneake* P(PS).

Com a recomendação 2 teriamos:

Exemplo 13

```
cneake Q(PS);
create P(PS)
```

onde

```
process Q(pf:...)          process P1 (pf':...)
begin                      begin
  "                          "
  "                          "
  "                          "
   create P1 (PS)             Rhnh pf' to PE
  "                          "
  "                          "
  "                          "
   end Q                     end P1
```

```
process P(p:...)
  "
  "
  "
begin
  link p to PE
   "
   "
   "
```

Assim se Pl for ativado antes de P, a ligação PS-PE'será feita e as mensagens colocadas por Pl em PS, enviadas para PE. E, após a liberação de PS por Pl, P usa PS e liga-a com PE. similarmente, se P for ativado antes de Pl, não causando maiores problemas e preservando a integridade dos processos.

III.2. Uso de parâmetros formais nos oomandos de ligação entre portas.

A restrição de usar somente parâmetros formais no lugar das portas de saída que são usadas pelo comando *link*, mais a restrição de cada porta ser usada exclusivamente por um processo de cada vez, e a recomendação 2 evitam a destruição indevida de uma ligação feita num processo ativo por outro processo ativo, como vimos nos diversos exemplos acima.

Note que só uma das restrições acima citadas não é suficiente para evitar destruição indevida de ligações, como pode ser verificado facilmente.

1. uso de parâmetros formais no coniando *link* sem uso mutuamente exclusivo de portas na ativação de processos: confrontar como exemplo 6.

2. uso mutuamente exclusivo de portas na ativação de processos sem uso de parâmetros formais no comando *link*.

Exemplo 14

```
  "
  "
  "
create P(ps);
create Q(ps');
  "
  "
  "
```

onde

```
process P(p:...)              process Q(q:...)
   "                             "
   "                             "
   "                  e          "
link ps' to PE                link q to PE;
   "                             "
   "                             "
```

Dependendo das velocidades de P(ps) e Q(ps'),ps' pode estar ligado a PE ou a PĒ.

Recomendação 3:

Se dentro da declaração de um processo P ocorre nend < > *to* pf e não ocorre nenhum comando *link* usando pf, recomenda-se que antes de cada ativação do processo P, seja feita uma ligação usando a porta que substitui pf nessa ativação.

Essa recomendação evita casos como:

Exemplo 15

Suponha que

```
create P(p);
create Q(p)
```

sejam executados na inicialização, onde:

```
process Q (fq: input ports)
   "
   "
  begin
    receive n from fq
  end Q

process P (fp: output ports)
   "
   "
```

```
  beghn
    "
    "..
     aend m to fp
end P
```

A mensagem mandada por P(p) não pode chegar até Q(q) em virtude da falta de ligação entre as portas p e q.

Comentário sobre a restrição de não se poder declarar tipos de portas dentro dos processos:

Essa restrição não permite que processos possuam portas e assim evitam, por exemplo, que as mensagens colocadas numa porta por uma encarnação possam deixar de ser transmitidas por que essa encarnação termina sua execução.

Daremos a seguir exemplo do uso dos mecanismos aqui descritos para transmissão de um arquivo de tamanho desconhecido de mensagens entre dois processos, supondo-se que a rede é confiável.

```
programa transmissão de arquivo
  global process: -
  naden N1,N2
  all output ports: N1:sm1,st1;
  all hnpuk ports: N2:sac
                   N1;eac;
                   N2:em2,et2
```

```
node N1 .
  input ports: eac
  output ports: stl, sml
  import: N2.et2
  local processes; origem
  process origem (psm, pst: output ports; peac: input ports)
    |envia arquivo de tamahho desconhecido ã priori e recebe
      resposta de recebimento do arquivo|
    var: qe: integer  |declaração das variãveis locais ao pro
                        cesso origem|
         m: mensagem,
         conf: Boolean
      "
      "
    begin
      qe:= 0;
      "inicializa arquivo"

      while -eof do begin
                      head m;
                      qe:= qe+1;
                      nend m to psm
                    end
                od  |manda arquivo para psm e qe = tamanho do
                      arquivo|
      link pst to et2;
      nend qe to pst reply to peac;  |manda tamanho de arquivo
                                e porta para posterior ligação|
```

```
        receive conf from peac  |recebe confirmação de recebimen_
                                to|
    end .origem

  begin  |inicialização do nõ|
    link in sml to em2;
    create origem (sml,stl,eac)
  end N1  |fim da declaração do nõ N1|

node N2
  input ports: em2,et2
  output ports: sac
  export: et2
  local processes: destino
  process destino (psac: output ports; pem,pet: input ports)
     |recebe um arquivo de tamanho desconhecido ã priori e acu-
       sa rebecimento|
    var: qr, qar: integer,
         msg: mensagem,
         ack: Boolean ,
         presp: all input ports
    begin
      qr:= o ; qar:= o;
      while qar = o do
        begin
          receive msg from pem;
          qr:= qr+1;
          receive qar from pet reply to presp when time out 1
                    do skip
        end
```

```
                od  |recebe mensagem do arquivo até receber
                        o tamanho do arquivo|
        while qr<qar do  |o número de mensagens recebidas não é
                        igual ao número de mensagens envia-
                        das|
          begin
            receive msg from pem;
            qr:= qr+1
          end
                od  |todas mensagens enviadas são recebi-
                        das|
        ack:= T;
        link psac to presp;  |liga psac à porta recebida em pet|
        aend ack ko pasc
      end destino

    begin      |inicialização do nó N2|
      create destino (sac, em2, et2)
    end N2  |fim da declaração de N2|
end transmissão de arquivo  |fim do programa transmissão de ar-
                                quivo|
```

Comentário sobre o exemplo 16.

Como estamos supondo a rede confiável, todo o arquivo será transmitido. Porém, como não temos informação nem sobre a ordem de transmissão de mensagens pela rede, nem de quanto tempo cada mensagem leva para ser transmitida, usamos a opção *when Rime out* no comando de recebimento que recebe o tamanho do arquivo.

A opção *reply to* foi usada, no caso de querermos modificar o programa para recebimento de arquivos de diversos nós de origem, não precisarmos fazer modificações na declaração do processo destino, só nas declarações globais ao programa, a saber: *nudea*, *all input ports*, *all output ports*.

CAPÍTULO IV

SEMÂNTICA OPERACIONAL

Daremos aqui a definição de estado de um programa com respeito aos mecanismos propostos neste trabalho.

Notamos que, para caracterizarmos um estado, não basta apenas apontadores para os próximos comandos a serem executados e os valores das variáveis de cada processo, como é feito usualmente. Vamos precisar, também, de informação sobre quais são os canais de comunicação existentes em um estado, já que a ligação entre portas é feita e desfeita dinamicamente. Como há criação dinâmica de processos ativos, vamos precisar, além disso, de informação sobre as encarnações ativas nesse estado.

Assim, o estado global de um programa fica determinado pelo estado de interconexão das portas e o estado de cada nó (lembramos aqui que o número de nós é fixo). Vamos supor que o Único meio dos processos se comunicarem é através de troca de mensagens sem memória compartilhada, já que nosso principal objetivo é dar a semântica dos mecanismos aqui propostos. Nas aplicações desses mecanismos em sistemas com memória compartilhada, o estado de cada nó teria de fornecer também informação sobre o estado das variáveis compartilhadas.

O estado de cada nó dá o estado das portas do nó e o estado de cada encarnação ativa de processo no nó. O estado das portas diz quais mensagens estão em cada porta e quais processos usam que portas naquele nó. O estado de cada encarnação

ativa no nó dá o estado das variáveis locais ao processo ativo e identifica com portas no nó os parâmetros formais do cabeçalho da declaração do processo.

Vamos descrever o estado global da maneira seguinte. Primeiro daremos uma descrição informal da representação do estado, com o significado pretendido para cada componente dessa representação do estado. (Note que o significado ("semântica") do estado é dado através das transições e essas serão tratadas mais tarde). Depois, apresentaremos a estrutura sintática do estado usando mecanismos de construção de tipos de Hoare, como em Pascal |40|. Por fim, descreveremos a transição de estados.

Faremos aqui uma observação sobre a notação usada. Se a definição da semântica operasional fosse dada através de VDL, a leitura desse trabalho seria por demais maçante. Por isso optamos por descrever a máquina abstrata usando mecanismos lingüísticos de linguagem de programação do tipo de Pascal. A transição de estados é feita através da descrição dos procedimentos que substituem as definições das instruções em VDL. Essa opção se deve ao fato de que especificações formais se tornam muito mais aceitáveis e compreensíveis se são expressas numa linguagem familiar à comunidade dos usuários, implementadores e projetistas de linguagens, em lugar de uma esotérica metalinguagem formal.

4.1. Descrição Informal do Estado Global

Dado um programa P, declarado segundo a sintaxe descrita no capítulo III, com n nós, o estado global de P é uma árvore cuja raiz é G. Vamos identificar o nome do i-ésimo nó

que ocorre na lista da declaração nodes do programa com o seu lugar nesta lista,

Os filhos de G são G.L, G.Ni,...,G.Nn, tais que G.L é uma folha, enquanto que, para cada i=1,...,n, G.Ni é uma árvore.

G.L é uma função que dá o estado de interconexão das portas, i.e.

G.L : all output ports → all input ports U(Δ)

onde:

G.L(p) é a porta de entrada que está ligada a p no estado G, se houver. Caso contrário, G.L(p) = A

Para cada i=1,...,n, a árvore G.Ni é definida como se segue. Sua raíz é chamada G.Ni . De G.Ni saem arestas rotuladas por conj-enc-ativ, tab-disp, tab-bags-portas-entrada e tab-bags-portas-saída para vértices, tais que:

G.Ni.conj-enc-ativ e G.Ni.tab-disp são folhas e G.Ni.tab-bags-portas-saída e G.Ni.tab-bags-portas-entrada são árvores.

Para cada i=1,...,n vamos chamar de portas-i o conjunto de todas as portas declaradas localmente no nó i, isto é:

portas=i = input porta (declaradas no nî i) U output ports (declaradas no nó i).

Neste contexto input ports e output ports se referem aos conjuntos das portas declaradas desse gênero ocorrendo no nó i.

Agora vamos descrever os filhos de G.Ni: G.Ni.conj-enc.ativ, G.Ni.tab-disp, G.Ni.tab-bags-portas-entrada e G.Ni.tab-bags-portas-sarda.

G.Ni.conj-enc-ativ é um conjunto de ponteiros.

G.Ni.tab-disp é uma função:

G.Ni.tab-disp:portas-i → processos U(Ω)

onde

G.Ni.tab-dis(p) é o nome do processo que tem pelo menos uma encarnação ativa em i usando p, se houver.. Caso contrário, G.Ni.tab-disp(p) = Ω;

processos é o conjunto de nomes dos processos declarados no nó i ou declarados globalmente.

(Assim, G.Ni.tab-dis dá a disponibilidade das portas do nó i.)

G.Ni.tab-bags-portas-safda é uma árvore cuja raiz chamamos de G.Ni.tab-bagç-portas-saída. Da raíz saem arestas, rotuladas por simples e comb, para folhas.

G.Ni.tab-bags-portas-entrada é uma árvore cuja raíz chamamos de G.Ni.tab-bags-portas-entrada. Da raíz saem arestas, rotuladas por simples, comb, simples-rec e comb-rec, para folhas.

Vamos agora descrever as folhas de G.Ni.tab-bags-portas-entrada e G.Ni.tab-bags-portas-saída.

G.Ni.tab-tabs-portas-saída.simples: output ports → Bag - of-mensagens

e

G.Ni.tab-bags-portas-saida.comb: output ports → Fair-bag-of-(mensagens x all input ports)

G.Ni.tab-bags-portas-entrada.simples (-rec) : input ports → Fair-bag-of-mensagens

G.Ni.tab-bags-portas-entrada.comb (-rec) : input ports → Fair-bag-of-(mensagens x all input ports).

("Bags-of" e "Fair-bags-of" serão definidas mais adiante).

As mensagens presentes numa porta de saída p do nó i são aquelas que já foram enviadas para a porta p mas ainda não foram transmitidas pela rede. Essas mensagens, no estado G, ficam representadas por G.Ni-tab-bag-portas-saída.simples(p) e por G.Ni.tab-bags-portas-saída.comb(p). As mensagens em G.Ni.tab-bags-portas-saída.comp(p) são aquelas associadas às portas de entrada enviadas através de comando usando a opção *reply to*.

Todas as mensagens já transmitidas para uma porta de entrada p, até o instante representado por G, estão representadas por G.Ni.tab-bags-portas-entrada.simples(p) e por G.Ni.tab-bags-portas-entrada.comb(p). (onde G.Ni.tab-bags-portas.comb(p) contém as mensagens associadas a portas), As mensagens já recebidas pela porta p estão representadas por G.Ni.tab-bags-portas-entrada.simples-rec(p) e G.Ni.tab-bags-portas-entrada.comb-rec(p), conforme o recebimento tenha sido feito através de comando sem ou com opção *reply to*, respectivamente.

Faremos aqui uma pausa na descrição do estado para

darmos alguns detalhes sobre os tipos Bag-of e Fair-bag-of.

O tipo Bag-of-itens está definido em |36| com o nome de Multiset-of-itens. Introduzimos aqui um anglicismo, usando o nome "bag" para objetos do tipo Multiset-of.

Intuitivamente uma "bag" b é um conjunto cujos elementos podem ser repetidos, i.e., a função característica de um item pode ter valor maior que 1, caso exista mais de uma cópia desse item na "bag".

As operações sobre "bags" são análogas às sobre conjuntos. Em |36|, na definição de "bags", ocorrem, entre outras, as operações:

choose: Bag-of-itens → itens

member-of?: Bag-of-itens x itens → Boolean

empty-bay:() → Bag-of-itens

Intuitivamente, a operação choose escolhe um item da "bag", member-of testa se um determinado item está na "bag" e empty-bag cria a "bag" vazia.

A operação choose tem a seguinte propriedade:

i) member-of?(p, choose(p)) = true se p ≠ empty-bag.

Vamos usar as seguintes notações no que se refere a "bags", quando m é um objeto do tipo itens e B,S, são objetos do tipo Bag-of-itens:

$\sqcup$ - símbolos usado para união entre "bags"

char(m,B): o valor da função característica de m na "bag" B

$m \varepsilon B$ se e só se member-of? (B,m) = true

$B \subset S$ se B é uma "sub-bag" de S, i.e., para todo objeto do tipo itens, se member-of? (B,m) então member-of? (S,m).

Assim, $m \varepsilon B$ se $char(m,B) \geq i$ e $B \subset S$ se e só se para todo m , se

mεB então mεS e char(m,B) ≤ char(m,S).

A operação |_| pode ser definida por,:

|_| : Bag-of-itens x Bag-of-itens → Bag-of-itens
com a propriedade de mεB |_| S se e só se mεB v mεS.

Vamos usar |_| com a seguinte propriedade adicional:
char (m,B |_| S) = charm(m,B) + char(m,S).
Isto é, a união de duas "bags" é a "união disjunta" dessas "bags", no sentido de termos na união de duas "bags" todas as cópias de itens que ocorrem nessas "bags".

Por convenção: p ⊖ m = p |_| (m), onde (m) é a "bag" com um Único elemento m, com uma só cópia de m.

A diferença entre "bags" é uma função,
0 : Bag-of-itens x Bag-cf-itens → Bag-of-itens e tem a seguinte propriedade:
m ε B θ S se e só se char(m,B) ≥ i e char(m,B) > char(m,S))

Fair-bag-of-itens é um gênero especial de Bag-of-itens em que a operação choose tem a seguinte propriedade adicional:

ii)c ε p → <> (choose(p)=c) onde
c é uma constante e p é variável
<> é o operador eventualmente de lógica temporal. (Pneulli |73|).

A propriedade i) diz que choose é uma função de escolha, enquanto que ii) diz que os itens de um Fair-bag-of-itens são retirados numa certa ordem, não importa aqui qual, mas que a escolha não é completamente arbitraria. (Uma possível implementação para Fair-bag-of-itens é filas de itens, por exemplo).

Quando estivermos nos referindo a "bags" que são "fair-bags", chamaremos a operação choose de take, para reforçar o fato de que essa operação satisfaz (ii).

Agora vamos descrever os elementos apontados pelos ponteiros em G.Ni.conj-enc-ativ. Cada ponteiro em G.Ni.conj-enc-ativ aponta para o estado de uma encarnação ativa de um processo no nó i.

Se p.é um ponteiro, usaremos a notação p↑, como em Wirth|88|, para o objeto apontado por p.

Para cada a∈G.Ni.conj-enc-ativ, a↑ é uma árvore cuja raíz chamamos de a↑ e os filhos de a↑ são: a↑.pontexto , a↑.tab-par, a↑.tab-val e a↑.tab-esp. a↑.pontexto e a↑.tab-esp são árvores.

af.pontexto é o texto do processo, de cuja encarnação estamos falando, com um apontador. Este aponta para o próximo comando a ser executado, no texto do processo, pela encarnação ativa a↑. (Aqui estamos identificando o estado da encarnação com o nome da encarnação).

Sobre objetos do tipo texto com apontador, cuja definição está na parte 4.2, agem as seguintes operações: first, in, down, is-last, current.

- first dá o texto com o apontador apontando para o seu primeiro comando;
- is-last é verdade se o apontador do texto aponta para um comando após o qual não há, no texto, nenhum comando a ser executado;
- current dá o comando apontado pelo ponteiro do texto;

- in move o apontador para dentro do comando apontado pelo apontador do texto, se esse comando é tal que, sua execução dependen de teste que ocorre no comando. Caso contrário in não altera o apontador.

(Assim, se o apontador de um texto aponta para receive n from e when time out dt do S, in faz esse apontador apontar para S.)

- down move o apontador para o próximo comando do texto, se houver.

(Assim, por exemplo, se o apontador estiver apontando para A no texto A;B down faz o ponteiro apontar para B.)

A↑.tab-par associa as portas usadas pela encarnação aos parâmetros formais da declaração do processo.

(Para simplificar a notação vamos supor, sem perda de generalidade, que serão usados parâmetros diferentes para cada processo. Caso contrário, teríamos que associar aos parâmetros os nomes dos processos.)

a↑.tab-val dá os valores das variáveis locais do processo nessa sua encarnação no estado G.Ni.

a↑.tab-esp é uma árvore com raíz a↑.tab-esp, cujos filhos são: a+.tab-esp.valexp e a↑.tab-esp.tempo (que são folhas).

a↑.tab-esp.valexp dá o valor da expressão usada na opção de saída por tempo, se o apontador do texto, a↑.pontexto, aponta para um comando usando tal opção, ou indef, em caso contrário.

a↑.tab-esp.tempo dá o tempo real de espera no caso de o ponteiro de a↑.pontexto apontar para comando de recebimento usando opção de saída por tempo, ou zero em caso contrário.

O estado inicial E, é dado por

E.L(p) = A para todas as portas do tipo *all output ports* declaradas no programa.

Para cada i=1,...,n, x ∈ (comb, simples) temos:

E.Ni.tab-bags-portas-saída.x(p) = ∅ para cada p declarada em i como

output ports

E.Ni.tab-bags-portas-entrada.x(-rec)(p) = ∅ para cada p declarada em i como

input ports

onde comb e simples referem-se às "bags" das mensagens associadas ou não a portas, respectivamente.

E.Ni.conj-enc-ativ = (Ii) onde:

Ii↑.pontexto é o texto do processo de inicialização do nó i com o ponteiro apontado para o seu primeiro comando,
Ii↑.tab-val=∅, já que estamos supondo que não há compartilhamento de variáveis,
Ii↑.tab-par = ∅
Ii↑.tab-esp.valexp = indef
I↑.tab-esp.tempo = 0

4.2 - *Estrutura Sintática*

Formalmente, a sintaxe (representação) do estado global de um programa vai ser dada usando declarações de tipo e mecanismos para construção de tipos estruturados de dados, de Hoare, em que um tipo é descrito em termos de outros tipos. A

definição dos tipos usados na descrição de um tipo será levada ate encontrarmos um tipo usualmente conhecido (por exemplo; integer, niapping, etc) ou cuja descrição cuja descrição seja obtida diretaniente do texto do prograina.

O estado global G é uni objeto 'do tipo estado global, i.e.:

```
obj G: estado global
type estado-global::record (L:estado-rede;
                            Ni:estado-nó(i);
                            ..............
                            Mn:estado-nó(n))
type estado-rede::mapping from all output ports to origin

type origin::union (all input ports, enum(Δ))
type estado-nó(i:nó)::record (
          conj-enc-ativ:set of pointer to estado-encarn(i);
          tab-disp:dis-porta(i);
          tab-bags-portas-entrada:conteúdo-porta-entrada(i);
          tab-bags-portas-saída:conteúdo-portas-saída(i))
type no::enum("Nós declaradas como nodes no programa")

type estado-encar(i:nó) ::
     record (pontexto:texto-com-apont(P)(process(i));
             tab-par:assoc-par(P)(process(i));
             tab-esp:tempo-esp(P)(Process(i));
             tab-val:val-var(P)(process(i)))

type process (i:nó)::union (global process, process decl(j))
```

type process decl(i:nó)::*union* (*enum* ("Processos declarados no nó-i"), enum (IN-i))

onde IN-i é o processo de inicialização do nó i.

kype global process::*enum* ("processos declarados globalmente no programa")

type texto-com-apont(P:process(i:nó))::TAD

(*sorts*:refcmd, *Bool*,cmd)

ops:first:( ) → refcmd

in, down:refcmd → refcmd

is-last:refcmd → *Bool*

current:refcmd → cmd

onde TAD abrevia tipo abstrato de dados

type-assoc-par (P:process(i:nó))::*mapping from* param(P) *to* portas(i)

type parm(P:process(i:nó))::*enum*("parâmetros que ocorrem no cabeçalho da declaração do processo P)

kype portas(i:nó)::*union*(input ports(i), output ports(i)

kype input ports(i:nó)::*enum*("portas declaradas em i como *input ports*")

kype output ports(i:nó)::*enum*("portas declaradas em i como *output ports*")

kype all input ports:*enum*("portas declaradas como *all input ports*")

kype all output ports::*enum* ("portas declaradas como *all output ports*")

kype portas decl(i:nó)::*union*(portas(i), import(i))

type import(i:nó)::*enum*(nomes de portas que ocorrem na declaração *import* do nó i)

type tempo-esp(i:nó)::*record*(valexp:tempo-max tempo:Integer))

type temp-max::*union*(integer, *enum*(indef))

kype val-var(P:process(i:nó))::*mapping from* var-decl(P)
to values (P)

type var-decl(P:process(i:nó)::*enum* "variáveis declaradas localmente no processo P")

onde o *tipo* values (P) consiste dos domínios dos valores das variáveis declaradas no processo P acrescidos de indef

kype dis-porta(i:nó)::*mapping from* portas(i)
ko *union* (process(i), *enum* (0))

kype conteúdo-portas-entrada(i:nó)::*record*(simples,simples-rec:
mapping from input ports(i) *to* Fair-bag-of-mensagens(i);
comb-rec, comb:*mapping from* input ports(i) *to* Fair-bag-of-*record*
(men:mensagens(i); pen:all input ports))

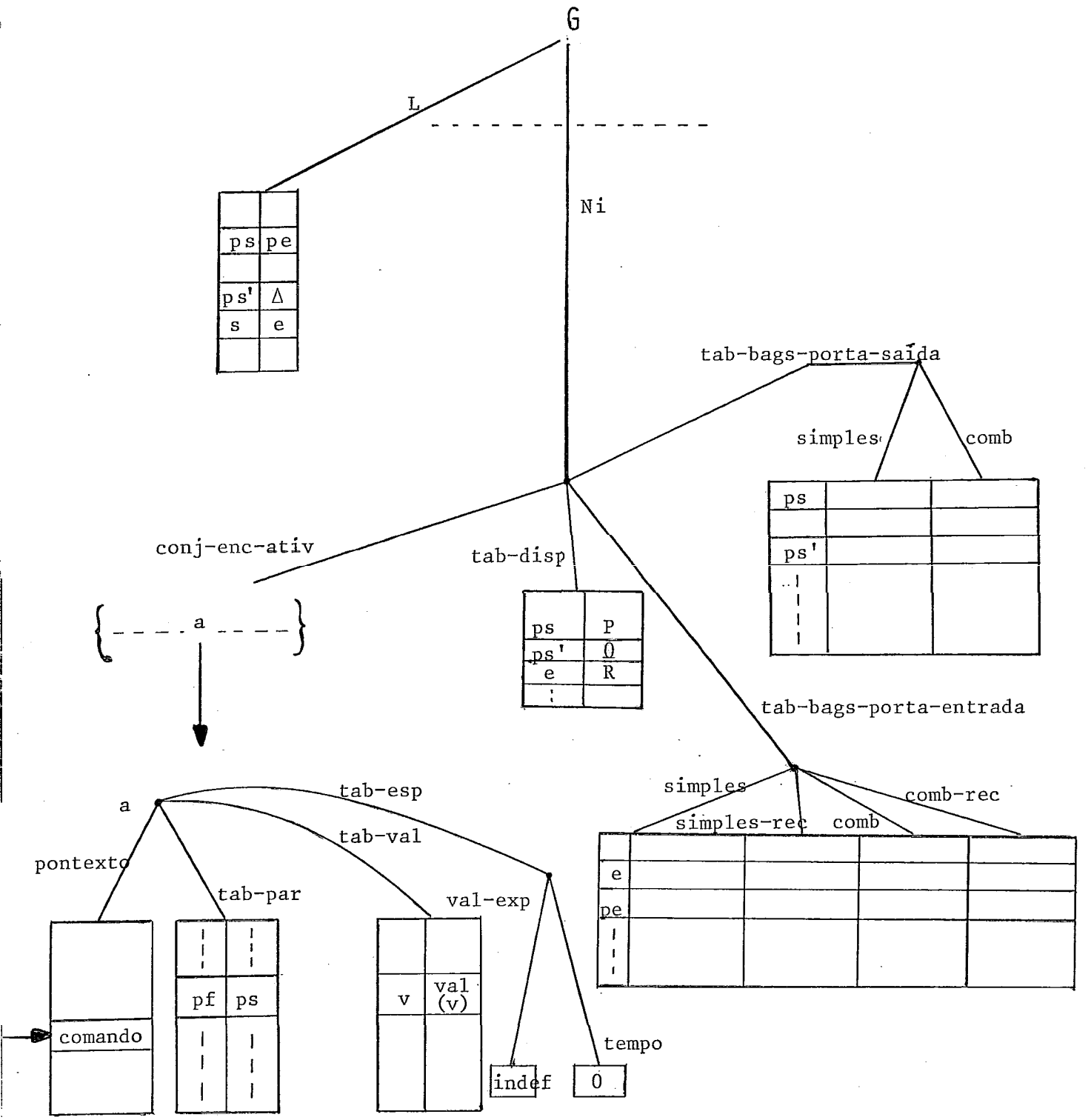
onde mensagens(i) é tipo declarado no texto do programa.Não especificamos o tipo mensagens(i), já que não estamos nos retendo no conteúdo das mensagens nem na questão de associar ou não tipos de mensagens às diversas portas dos nós.

O estado inicial E já foi definido informalmente, sua definição formal segue imediatamente da informal, sendo que para cada i:nó,

E.Ni.conj-enc-ativ = (Ii) onde

Ii↑.pontexto:texto-com-apont(IN-i) e

current(Ii↑.pontexto) = first(Ii↑.pontexto)



Estado Global

4.3 - *Transição de Estados*

A semântica operacional de um programa será dada por um conjunto C de seqüências de estados. Cada seqüência em C é uma seqüência, possivelmente infinita, G0,G1,...,Gn onde GQ é o estado inicial e Gi+1 é obtido de Gi através de uma transição de estado.

Aqui vamos tratar somente das transições de estado que se referem aos comandos propostos nesse trabalho. As outras transições, referentes aos outros comandos, não serão descritas por serem supostamente conhecidas.

A transição de estado global é causada por dois tipos de transições, que podem ocorrer em paralelo: transição autônoma e transição comandada.

A transição autônoma é causada pela transmissão de mensagens através da rede de transmissão.

Não vamos modelar a rede mas vamos supô-a confiável, porém com atraso (possivelmente variável) finito. Isto é, uma mensagem, uam vez colocada na rede de transmissão, chega a seu destino após um tempo finito.

Vamos simular o comportamento da rede através de uma escolha não determinística. Esta consiste em selecionar, dentre as mensagens que estão sendo transmitidas pela rede, algumas consideradas como já tendo atingido seus destinos.

Vamos denotar por trans(G) o conjunto de todas as famílias B= {Bx(ps)}x, ps onde x ∈ (coinb, simples) e ps varre as portas declaradas como *all output ports* tais que, para cada nó i e cada porta de saída ps declarada no nó i, com G.L(ps) definida, se tenha Bx(ps) ⊆ G.Ni.tab-bags-portas.x(ps).

Cada B ε transm(G) representa uma possível escolha de mensagens nas portas de saída em G, cuja transmissão no próximo estado foi completamente efetuada.

A transição comandada é causada pelas execuções dos próximos comandos de encarnações ativas de cada nó.

Vamos simular a independência entre as velocidades de cada encarnação ativa de processo escolhendo não deterministicamente, em cada nó, as encarnações ativas que vão executar o seu próximo comando. Isto é, vamos denotar por Prog(G) o conjunto de todas as famílias $P = \{Prog\text{-}i\}, i \varepsilon \{1,...,n\}$, tais que para cada i se tenha:

$$Prog\text{-}i \subset Hab\ il(Ni) \subseteq G.Ni\ conj\text{-}enc\text{-}ativ\ e \bigcup_{i \le i \le n} Prog\text{-}i \neq \emptyset$$

onde Hab il(Ni), que será definido posteriormente, representa um conjunto de encarnações ativas em G prestes a executarem seu próximo comando.

Cada P ε Prog(G) representa uma possível escolha de encarnações ativas em cada nó que no próximo estado terão executado seu próximo comando.

Para cada P ε Prog(G) vamos definir o estado global G[P], daí podemos definir uma relação entre estados globais, comand, tal que G comand G' se e só se existe P ε Prog(G) tal que G' = G[P]. G' representa o estado atingido a partir de G, ao se estabilizar todas as mudanças de estado das encarnações em P.

Antes de definirmos G[P], vamos dar algumas definições para simplificar a notação, ressalvando que, como no caso da descrição informal de G, a notação usada será a mais preci-

sa possível, porem informal o suficiente para preservar a legibilidade. A formalização dos conceitos aqui expostos usando VDL segue imediatamente da descrição informal que estamos apresentando, mas foge ao interesse central do trabalho3

Vamos usar os símbolos usuais de lógica , V , V, v, &, Λ , → , etc, mais os predicados e seletores da sintaxe abstrata dos mecanismos linguísticos considerados nestes trabalho, para abreviar as descrições de propriedades na meta-linguagem. Assim estas se tornam mais consisas e compreensíveis.

Definições e abreviações.

Para i:nó e para cada P:process(i), vamos definir:

i) cabeça1ho(P) = lista-porta-saída(P)ᵔlista-porta-entrada (P) onde lista-porta-saída(P) = 1ista-par-porta-saída(declaração de P) e lista-porta-entrada(P) = lista-par-porta-entrada(dec1aração de P).

2) par(P) = {p: param(P)/is(cabeçalho(P),p)}
onde is-in:list-of-itens x itens → *Boolean* é uma função que testa se um determinado objeto do tipo itens ocorre ou não numa lista de itens.

3) se 1 é uma lista de itens e s, um item, chamaremos de lugar (s,l) o lugar que s ocupa na lista 1 se s ocorre em 1 e usaremos a notação 1.j para identificar o j-ésimo item da lista 1, se i≤j≤length(1).

Para cada i=l,...,n vamos abreviar G.Ni.conj-enc-ativ por Ative-i.
As definições e abreviações a seguir se referem a um i fixo (i=1,...,n).

Para cada p:output ports(i), abreviamos G.Ni.tab-bags-portas-saída.x(p) por i-Bag.x(p) e para cada p:input ports(i), abreviamos

G.Ni.tab-bags-portas-entrada.x(p) θ
G.Ni.tab-bags-portas-entrada.x-rec(p) por i-Bagen.x(p).
onde x ε (simples,comb)

4) Para cada a ε Ative-i vamos abreviar current(a↑.pontexto) por a$ e definimos:

4.1) arg(a$) = {p:portas(i)/is-in(lista-arg(a$),p}

4.2) apto-a-ligar(a$) ↔
(is-link in(a$) v is-link(a$) & (is-link in(a$) & (G.L(porta-origem(a$) # porta-destino(a$) v G.L(porta-origem(a$) ≠ A)
→ i-Bag.x(porta-origem(a$) = (I) & (is-link(a$) & (G.L(porta-real(a$) ≠ A v G.L(porta-real(a$) ≠ val(l-porta-destino(a$))
→ i-Bag.x(porta-real(a$) = (I))

onde

porta-real(a$) = a+.tab-par(C)
C = s-porta-saída(a$)) se is-send(a$)
= sr-porta-saída(a$) se is-send-reply(a$)
= r-porta-entrada(a$) se is-receive(a$)
= rr-porta-entrada(a$) se is-receive-reply(a$)
= rrt-porta-entrada(a$) se is-receive-timeout(a$)
= rrt-porta-entrada(a$) se is-receive-reply-timeout(a$)
= l-porta-origem se is-link(a$)
val(sel-(a$) = sel(a$) se se1(a$):portas decl(i)

= a↑.tab-val(sel(a$) se sel(a$):var-decl(P) e
P:process(i) e
a↑:texto-com-apont(P)

= a↑.tab-par(v) se v ∈ par(P) e
a↑:texto-comapont(P) e
P:process(i)

(sel é o seletor da definição da sintaxe abstrata do comando a$)

Isto é, val(v) dá o valor de v na encarnação representada por a↑ ou v se v é nome de porta. Podemos estender a notação val(v) para expressões v, cujas variáveis ocorrendo em v são locais ao processo cuja encarnação é representada por a↑. Assim, val(v) é o valor da expressão v calculada quando as variáveis u que ocorrem em v têm valor a↑.tab-val(u) .

4.3) available(a$) ↔ (is create(a$) v is-link in(a$) & (is-create(a$) → ∀p(p ∈ arg(a$) → G.Ni.tab-disp(p) = Ω) & (is-link in (a$) → G.Ni.tab-disp(porta-origem(a$)) = Ω).

Vamos descrever as definições 4.2. e 4.3. acima, já que as outras explicam-se por si mesmas.

Definição 4.2: apto-a-ligar(a$) diz que: a$ é um comando de ligação entre portas e caso a parta de saída a ser ligada já esteja ligada a alguma outra porta de entrada não há mensagens na porta de saída ainda a serem transmitidas.

Definição 4.3: available(a$) diz que as portas que ocorrem no comando apontado por a$ estão disponíveis.

Agora vamos descrever um subconjunto Indep-i de Ative-i, que intuitivamente é o conjunto das encarnações ativas no nó

i que podem executar os comandos apontados pelos seus ponteiros de texto no estado G e essa execução não impede o progresso das outras encarnações ativas em i,

5) a ∈ Indep-i ↔ a ∈ Ative-i & [(0)∨(i)∨(2)∨(3)∨(4)∨(5)]

onde:

(∅) = is-send(a$) ∨ is-send-reply(a$)

(1) = is-link in(a$) & available(a$) & apto-aligar(a$) & ∀b(b ∈ Ative-i & is-create(b$) → porta-origem(a$) ∉ arg(b$)).

(2) = is-create(a$) & available(a$) & ∀b(b ∈ Active-i → ∃is-create(b$) & bfa ↛ arg(b$) ∩ arg(a$)) = @) & (is-link in (b$) → porta-origem(b$) ∉ arg(a$)))]

(3) = (is-receive(a$) ∨ is-receive-replay(a$)) & i-Bagen.x (porta-real(a$)) ≠ ∅)

(4) = is-receive-timeout(a$) ∨ is-receive-reply-timeout(a$)

(5) = is-link(a$) & apto-a-ligar(a$).

Nota: na definição (2), ∅ se refere ao conjunto vazio enquanto que em (3), ∅ segue a convenção feita anteriormente sobre bags.

Agora vamos definir um subconjunto Depend-i de Ative-i que, intuitivamente, são as encarnações ativas no nó i habilitadas a executar link in's e/ou create's, porém a execução de um desses comandos por uma dessas encarnações pode excluir a execução dos comandos por outras encarnações em Depend-i. Isto é, todas as encarnações em Depend-i podem executar seu próximo comando, mas só algumas vão fazê-lo.

6) a ∈ Depend-i ↔ a ∈ Ative-i & [(6) ∨ (7)]

onde:

(6) = is-link in(a$) & apto-a-ligar(a$) & available(a$) & b(b ∈ Ative-i & available(b$) & is-create(b$) & porta-origem(a$) ∈ arg(b$))

(7) = is-create(a$) & available(a$) & [∃b(b ∈ Ative-i & b#a & is-create(b$) & available(b$) & arg(b$)∩ arg(a$) ≠ ∅) ∨ ∨ ∃b(b ∈ Ative-i & is-link in (b$) & apto-a-ligar(b$) & porta-origem(b$) ∈ arg(a$))]

Agora vamos tomar subconjuntos de Depend-i que tenham a seguinte propriedade: os comandos apontados pelos apontadores não têm portas em comum.

Isto é seja Ai = {Si}Si ⊆ Depend-i tal que, para a, b ∈ Depend-i e a # b então

(8) = a ∈ Si & b ∈ Si ↔ ((is-create(a$) & is-create(b$) → arg(a$)∩ arg(b$) = ∅) & (is-create(b$) & is-link in(a$) → porta-origem(a$) ∉ arg(b$)))

Feitas essas definições, podemos agora definir Habil (G.Ni):

Habil(G.Ni) = Indep-i ∪ Si, para algum Si ∈ Ai tomado arbitrariamente.

Agora tome H(G.Ni) = {Habil(G.Ni)}Si ∈ Ai a família de todos os Habil(G.Ni), para todas as possíveis escolhas de Si.

Cada família P = {Prog-i}i=1,...,n é tal que Prog-i ⊆ Habil(G.Ni) para algum Habil(G.Ni) ∈ H(G.Ni) e $\bigcup_{i \leq i \leq n}$ Prog-i ≠ ∅.

Assim, suponha dado P = {Prog-i}i=1,...,n. Vamos definir ainda, para cada i=1,...,n, e para cada a ∈ Ative-i,

7) prog-i(a) $\leftrightarrow$ a ε Prog-i

8) term-i(a) $\leftrightarrow$ a ε Prog-i & is-last(a$\uparrow$.pontexto)
(term-i(a) diz que a encarnação ativa a+ vai executar seu último comando).

Agora podemos finalmente definir G[P]. Para simplificar a notação chamamos G[P] por G'.

Construção de G':

a) G'.L.

Para cada i:nó e p:output ports(i)

G'.L(p) = val(porta-destino(a$) se $\exists$a(prog-i(a) & is-link in (a$)
= val(l-porta-destino(a$) se a(prog-i(a) & is-link (a$) & porta-real(a$) = p)
= G.L(p) caso contrário

Isto é, G'.L é o estado da rede depois de todas as encarnações em Prog-i, para cada nó i, estabelecerem as ligações escolhidas. Note que, pela definição de Prog-i e pelo uso exclusivo de portas G'.L é uma função,

b) Para cada i, G'.Ni é definido por seus componentes, a saber:

b.1) G'.Ni.tab-bags-portas-sarda
para cada p:output ports(i)
G'.Ni.tab-bags-portas-saída.x(p) = i-Bag.x(p) $\uplus$ mens se (9)
= i-Bag.x(p) caso contrário

onde (9) = $\exists$a(prog-i(a) & (is-send(a$) $\vee$ is-send-replay(a$) & porta-real(a$) = p).

e mens = a↑.tab-val(s-mensagem(a$)) se is-send(a$) & x = simples
= (a↑.tab-val(sr-mensagem(a$),val(sr-porta-reply(a$)))
se x = comb & is-send-replay(a$)

Isto é, para toda encarnação em Prog-i, que está prestes a executar um comando de envio em G, o valor da mensagem é adicionada à "bag" associada à porta de saída, a qual o comando *send* se refere.

b.2) G'.Ni.tab-bags-portas-entrada
para cada p:input ports(i),
G'.Ni.tab-bags-portas-entrada.x-rec(p) =
= G.Ni.tab-bags-portas-entrada.x-rec(p) θ n se (iØ)
= G.Ni.tab-bags-portas-entrada.x-rec(p) caso contrário

onde (lØ) = i-Bagen.x(p) ≠ Ø & ∃a(prog-i(a) & (is-receive(a$) v is-receive-replay(a$) v ((is-receive-timeout(a$) v is-receive-reply-timeout (a$) & (dentro-prazo(a↑) v início-espera(a↑))))).

n = take (i-Bagen.x(p))

x = simples se is-receive(a$) v is receive-timeout(a$)
= comb se is-receive-replay(a$) v is-receive-reply(a$) v is-receive-replay-timeout(a$)

Isto é, para cada encarnação em Prog-i cujo ponteiro aponta para um comando de recebimento e há mensagens a serem recebidas na porta de entrada referida por esse comando (e ainda não expirou o tempo de espera, no caso de comando com saída por tempo)., uma dessas mensagens será adicionada à "Fair-bag" das mensagens recebidas por essa porta, no estado G'.

Nota: início-espera e dentro-prazo serão definidos mais adiante, mas o significado intuitivo é indicado mneumonicamente.

c) *G'.Ni.tab-disp*

para cada porta p:portas(i)

G'.Ni.tab-disp(p) = nome-proc(a$) se (11)

= Ω se (12)

= G.Ni.tab-disp(p) nos demais casos.

onde (11) = ∃a (prog-i(a) & is-create(a$) & p ∈ arg(a$))

(12) = ∃a (term-i(a) & usa(p,a))

e usa (p,a) = $\bigvee_{P:process(i)}$ $\bigvee_{q \in par(P)}$ (a↑.tab-par(p)=q)

Isto é, G'.Ni-tab-disp é diferente de G.Ni-tab-disp caso uma das encarnações ativas em Prog-i termina sua execução ou executa um comando create.

Antes de definirmos G'.Ni-conj-enc-ativ vamos fazer algumas convenções e definições para simplificar mais a notação.

Para a ∈ Ative-i definimos

9) espera(a$) ↔ (is-receive-timeout(a$) v is-receive-reply-timeout (a$)) & (is-receive-timeout(a$) → i-Bagen.simples(porta-real(a$))=∅) & (is-receive-reply-timeout(a$) → i-Bagen.comb (porta-real(a$)) = ∅)

10) inicio-espera(a↑) ↔ espera(a$) & a↑.tab-esp.valexp # indef & a↑.tab-esp.tempo = ∅

11) dentro-prazo(a↑) ↔ espera(a$)) & a↑.tab-esp.valexp = indef & ∅ ≤ a↑.tab-esp.tempo < a↑.tab-esp.valexp.

12) decurso-prazo(a↑) ↔ espera(a$) & a↑.tab-esp.tempo = a↑.tab-esp.valexp.

Para cada v:mensagem(i), definimos:

13) recebe-simples(v,a↑) ↔ (is-receive(a↑) v is-receive-timeout(a↑) & i-Bagen.simples (porta-real(a$)) ≠ Ø & (is-receive (a$) → r-mensagem(a$) = v) & (is-receive-time-out(a$) → rt-mensagem(a$) = v & -decurso-prazo(a↑)))

14) recebe-comb(v,a↑) ↔ (is receive-reply(a$) v is-receive-reply-timeout(a$)) & (is-receive-reply(a$) → rr.mensagem (a$) = v) & (is-receive-reply-timeout(a$) → rrt-mensagem (a$) = v &-decurso-prazo (a+))& i-Bagen.comb(porta-real (a$)) ≠ Ø.

Para cada v:all input ports, definimos:

15) recebe-porta(v,a↑) ↔ is-receive-reply(a$) v is-receive-reply-timeout(a$) & (is-receive-reply(a$) → rr.porta-reply(a$) = v & -decurso-prazo(a↑)).

d) *G', Ni.conj-enc-ativ*

Chamamos G'.Ni.conj-enc-ativ de Ative'-i.

Para a ∈ Ative-i tal que -term-i(a) denotaremos por atual-a, os valores apontados por a em G e prcx-a, os valores apontados por a em G'.

Ative'-i é definido por:

16) a ∈ Ative'-i ↔ (a ∈ Ative-i & -term-i(a) v a ∉ Ative-i & ∃b (b ∈ Ative-i & prog-i(b) & is-create(atual b$↑) & K(a,b)))

onde K(a,b) ≡ a↑.pontexto::texto-com-apont(nome-proc(atual b$) & a$ = first(a↑.pontexto) & ∀p(p:param(nome-proc(atual b$))→ a↑.tab-par(p) = lista-arg(atual b$).lugar(p,cabeçalho(nome-proc(atual b$) & ∀v(v:var-decl(nome-proc(atual b$))→ a↑.tab-val(v) = indef) & a↑.tab-esp.valexp = indef & a↑.tab-val.tempo = 0).

Isto é, os elementos em Ative¹-i 'são todos os de Ative-i que não terminaram sua execução mais novos ponteiros de texto com apontadores, cujo texto é o texto de um processo ativado por alguma encarnação em Ative-i e para esse novo elemento, suas tabelas são inicializadas e o ponteiro do texto aponta para o primeiro comando do mesmo. A tabela de parâmetros apontada por esse novo elemento é completada associado as portas que ocorrem no comando *create* correspondente à ativação, aos parâmetros formais do processo cujo nome ocorre nesse comando *create*, preservando a ordem entre os parâmetros do cabeçalho da declaração do processo e da lista de portas no comando *create*.

Agora vamos descrever os componentes de a+, quando a ∈ Ative¹-i ∩ Ative-i & -term-i(a).

Para toda encarnação ativa em G que não aponta para seu Último comando, i.e., para todo a ∈ Ative-i & -term-i(a), a↑ em G' é dada por:

d.1) prox a$ = atual a$ se - prog-i(a) v início-.espera (atual a) v dentro-prazo (atual a)
= in(atual a$) se decurso-prazo(atual a)
= down(atual a$) nos demais casos

d.2) prox a.tab-esp.valexp = val(tempo-espera(atual a$) se início espera(atual a) v dentro-prazo (atual a)
= indef nos demais casos

d.3) prox a.tab-esp.tempo = atual a.tab-esp.tempo + 1 se início-espera(atual a) v dentro-prazo (atual a)
= 0 nos demais casos

d.4) prox a.tab-val(v) = take i-Bagen.simples(porta-real(atual a$) se recebe-simples(v,atual a)

(*) = take(i-Bagen.comb(porta-real(atual a$)) i
se recebe-comb(v,atual a)

(*)= take(i-Bagen-comb(porta-real(atual a$)) 2 se recebe-porta(v,atual a)
= atual a.tab-val(v) nos demais casos

((*) onde o índice se refere à projeção correspondente.)

d.5) prox a.tab-par = atual a.tab-par

Vamos agora definir o efeito sobre G de uma transmissão de mensagens feita pela rede.

Assim, para B E transm(G), vamos definir o estado global G[B]

(a') G[B].L = G.L

Para cada i=1,...,n,

(b'.1) para cada p:output ports(i), G[B].Ni,tab-bags-portas-saída.x(p) = G.Ni.tab-bags-portas-saída.x(p) θ Bx(p) se Bx(p) E B.
= G'.Ni .tab-bags-portas-saída.x(p) caso contrário

(b'.2) para cada p:input ports(i)

G[B].Ni.tab-bags-portas-entrada.x(p)=

G.Ni.tab-bags-portas-entrada.x(p) $\sqcup$ $\bigsqcup$ {Bx(ps) $\in$ B/G.L(ps)= p}

onde x $\in$ {comb,simples}

G[B].Ni.Z = G.Ni.Z para os demais seletores Z de G.Ni.

Note que G[B].Ni.tab-bags-portas-entrada.x(p) é ainda uma "Fair-bag", já que "bags" de portas de entrada são do tipo Fair-bag-of.

Intuitivamente, G[B] é o estado obtido a partir de G tal que as mensagens, cuja transmissão foi terminada, foram adicionadas às "bags" de entrada de destino. Essas mensagens são também retiradas da "bag" de saída de origem.

Agora então podemos definir uma relação entre estados globais, trans tal que G trans $\hat{G}$ se e só se: 1) existem P $\in$ Prog(G) e B $\in$ trans(G) tal que $\hat{G}$=G[P][B] ou 2) Prog(G)$\neq\emptyset$, mas existe B $\in$ transm(G) tal que $\sqcup$ B$\neq\emptyset$ e $\hat{G}$ = G[B].

Intuitivamente, se G trans $\hat{G}$, então $\hat{G}$ é o estado obtido a partir de G tal que:

1) todas as encarnações escolhidas para progredir em G efetuaram seus próximos comandos e as mensagens, cuja transmissão foi terminada, foram adicionadas às "bags" de entrada de destino e retiradas das "bags" de saída de origem.

2) nenhuma encarnação ativa pode progredir mas há mensagens a serem transmitidas em G e as mensagens cuja transmissão foi terminada, foram adicionadas e retiradas das "bags" de destino e origem, respectivamente.

Definição: Uma seqüência de estados globais o = (GØ,Gi,...) é uma <u>seqüência executável</u> se e só se GØ=E (estado inicial) e

∀i ≥ 1 (Gi-1 transGi & fair(Gi,σ) & conf(Gi,σ) & fair-prog (io)) onde

17) fair(Gi,σ) ↔ ∀s:nó,∀p:input ports(s),∀ x ∈ {comb,simples} ∀m: mensagem(s)(m ∈ mensagens-a-serem-recebidas.x(i,s,p) & ∀j (j ≥ i → ∃k ≥ j & ∃a ∈ Gk.Ns.conj-enc-ativ & (x=simples → is-receive(a&))&(x=comb→is-receive-reply(a&)) → ∃j (j ≥ i & m= take(mensagens-a-serem-recebidas.x(j,s,p))

quando

mensagens-a-serem-recebidas.x(i,s,p) =

Gi.Ns.tab-bags-portas-entrada.x(p) Θ

Θ Gi.Ns.tab-bags-portas-entrada.x-rec(p)

18) conf(Gi,σ) ↔ ∀s:nó, ∀p:output ports(s), ∀m:mensagem(s), ∀x ε {simples,comb}

(m ∈ mensagens-a-serem-enviadas.x(i,s,p) & ∃pe(Gi.L(p)=pe) → ∃j(j ≥ i) & B ∈ transm(Gj) & ∃Bx(p) ∈ B & m ∈ Bx(p)))

quando

mensagens-a-serem-enviadas.x(i,s,p) = Gi.Ns.tab-bags-portas-saída.x(p)

19) fair-prog(Gi,σ) ↔ ∀s:nó, ∀a ε Gi.Ns.conj-enc-ativ (∀j (j ≥ i) → ∃k:k ≥ j & ∃Habil(Gk.Ns) ∈ H(Gk.Ns) &a ∈ Habil (Gk.Ns)) → ∃k(k ≥ i & ∃Prog-s ε Prog(Gk) & a ε Prog-s).

Vamos aqui explicar as definições 17)-19).

Definição 77: diz que o comportamento de cada "bag" de entrada numa seqüência executável é o comportamento de uma "Fair-bag".

Definição 18: diz que a rede é confiável, isto é, toda mensagem colocada numa porta de saída será eventualmente transmitida,

Definição 19: diz que a escolha das encarnações que progridem em cada estado numa seqüência executável é uma escolha justa, no sentido de uma encarnação não poder ser habilitada um número infinito de vezes sem que seja escolhida para progredir em sua execução.

CAPÍTULO V

EM DIREÇÃO A UMA SEMÂNTICA AXIOMÁTICA.

Neste capítulo daremos algumas propriedades dos comandos apresentados neste trabalho, Estas visam a uma futura axiomatização da semântica desses comandos, que possa vir a ser usada para verificação de propriedades de segurança ("safety") e vida ("iiveness") de programas que os usem.

Propriedades de segurança e vida são, segundo Lamport |47|, definidas da maneira seguinte. Segurança é uma classificação das propriedades de programas que afirmam que um programa nunca entra num estado inaceitável; correção parcial,ausência de bloqueio perpétuo e exclusão mútua são exemplos de propriedades de segurança (correção parciai é a propriedade que afirma que se um programa começa satisfazendo uma dada pré-condição, então nunca ocorre que o programa termine com uma dada pós-condição falsa). Propriedades de vida afirmam que o programa eventualmente chega a um certo estado desejável. Em programas seqüenciais a propriedade de vida que mais interessa ser estabelecida é a terminação; em programas concorrentes outras propriedades de vida são relevantes,tais como, cada pedido de serviço é eventualmente atendido, uma mensagem chega eventualmente a seu destino,cada processo eventualmente tem acesso a um recurso compartilhado.

Vários métodos tem sido propostos na literatura para a verificação de propriedades de segurança e vida de programas concorrentes,por exemplo, em Owicki e Gries |67| , Apt et al.

|5|, Levin e Gries |55|, Misra e Chandy |62|, Owicki e Lampont |69|. Em geral, os métodos tem uma característica comum, que é tratar os processos isoladamente e, depois, mostrar a não interferência entre as provas de cada processo.

Em Misra e Chandy |62|, as assertivas envolvem a história das comunicações de cada processo, e as propriedades de segurança são expressas de tal forma que a condição a ser sempre satisfeita vale quando ainda não houve nenhuma troca de mensagem e é mostrada válida indutivamente após cada troca de mensagem. Como a topologia é estática no modelo de |62| e as assertivas são sobre histórias locais, não há necessidade de prova de não interferência. Em Apt et al. |5|, Levin e Gries |55|, o endereçamento é explícito e o método de prova apresentado envolve o estabelecimento de invariante global para o programa. Nos métodos expostos em Lamport |49| e Owicki e Lamport |69|, as pré-assertivas são já tomadas como invariantes sob a execução do programa.

No caso de programas usando os nossos mecanismos, a dificuldade reside no estabelecimento de um invariante global, devido ao dinamismo tanto da topologia como do número de encarnações ativas de processos. Há também a dificuldade em estabelecer a prova de não-interferência já que: 1) esta envolveria provas para cada encarnação ativa e cada encarnação ativa pode ter propriedades que podem depender de propriedades do processo que a ativa; 2) há interferência da rede, que é como se fosse uma "caixa preta" sobre a qual não temos muita informação e que não deterministicamente interfere no estado das portas.

As propriedades de segurança em |69| são derivadas de propriedades invariantes sob a execução do programa, enquan

to que as propriedades de vida são estabelecidas a partir de propriedades de segurança.

Vamos apresentar algumas propriedades de correção condicional dos comandos, propriedades de vida e algumas que vão diminuir o número de provas de não-interferência.

Denotaremos por "correção condicional" a propriedade que expressa que se um comando começa sua execução satisfazendo a uma certa condição então, se e quando termina esta execução, uma certa condição será verdadeira. Correção parcial é uma propriedade de correção condicional.

Usaremos uma linguagem temporal para expressar tais propriedades, já que esta parece ser uma forma bastante elegante e adequada para tratar propriedades de vida e, por uniformidade, expressaremos as outras propriedades também nessa linguagem.

A lógica temporal (linear) modela a programação concorrente como consistindo de numerosos estados (universos) e uma relação de acessibilidade entre esses universos. Esta relação especifica a possibilidade de passar de um estado para outro.

Os modelos para lógica temporal consistem de seqüências infinitas $o = S\emptyset, S1, \ldots$, tais que Si é acessível de Sj se e só se $j \leq i$.

Assim se $\sigma = G\emptyset, G1, \ldots$ é uma seqüência de execução de um programa concorrente P (como descrita no capítulo 4) ela origina naturalmente um modelo para a lógica temporal:

$\overline{\sigma} = \sigma$ se é infinita

$\overline{o} = G\emptyset, G1, \ldots Gk, Gk, Gk \ldots$ se $o = G\emptyset, Gi, \ldots, Gk,$

O tempo é considerado como uma seqhência de instantes discretos e o estado Gi de uma seqhência de execução σ= GØ,Gl,... representa o estado da computação no instante i.

Em geral, como por exemplo em MAnna e Pnueli |58| , Owicki e Lamport |69|, considera-se o presente como o instante Ø e o futuro como tempo i, para qualquer i>0. As referêntias ao passado são referências ao passado do futuro. (Por exemplo, pPq, onde P é o operador de precedencia, em Manna e Pnueli acima citado tem o seguinte significado intuitivo: a primeira ocorrência de p (observada do presente) precede a primeira ocorrência de q.)

As assertivas temporais são construídas a partir de assertivas imediatas por aplicação dos conectivos sentenciais, quantificadores e dos operadores temporais $\Diamond$, $\Box$, P, U, o. Uma assertiva imediata é uma função proposcional do estado do programa, enquanto que uma assertiva temporal é uma função sobre seqüências de execução.

As assertivas imediatas podem mencionar variáveis do programa, variáveis auxiliares (conceito introduzido por Clint |21|, parâmetros formais e variáveis do estado (estas se referindo às "bags" associadas às portas ou aos ponteiros dos textos das encarnações ativas). Mais adiante daremos mais detalhes sobre as assertivas imediatas.

Se a é uma seqhência de execução σ=GØ,Gi,..,Gi,... e P uma assertiva temporal, vamos usar a notação $\bar{\sigma} \models P$ para denotar que $\bar{\sigma}$ satisfaz P e vamos usar a notação $\bar{\sigma}(i)$ para a seqüência Gi,Gi+l,... para i≥l.

Assertivas imediatas (afirmações sobre estados) po-

dem ser consideradas como assertivas temporais (afirmações sobre seqüências de execução) que se referem ao presente,

Assim, se P é uma assertiva imediata $\bar{\sigma}(i) \models P$ se e só se Gi $\models$ P (mais adiante daremos o significado de Gi $\models$ P).

O significado dos operadores $\Box$, $\Diamond$, U, P e a é dado a seguir, onde R e Q são assertivas temporais:

$\bar{\sigma}(i) \models \Box R$ se e só $\forall j \geq i\ \bar{\sigma}(j) \models R$.

$\bar{\sigma}(i) \models \Diamond R$ se e só se $\exists j, j \geq i$ tal que $\bar{\sigma}(j) \models R$.

$\bar{\sigma}(i) \models RUQ$ se e só se $\exists k \geq i\ (\bar{\sigma}(k) \models Q$ e $\forall t\ i \leq t < k: \bar{\sigma}(t) \models R)$.

$\bar{\sigma}(i) \models RPQ$ se e só se $\forall k > i$ se $\bar{\sigma}(k) \models Q$ então $\exists t\ (i \leq t < k)$ e $\bar{\sigma}(t) \models R)$.

$\bar{\sigma}(i) \models oR$ se e só se $\bar{\sigma}(i+1) \models R$.

Para os concectivos sentenciais e quantificadores, a definição de satisfação é a usual, por exemplo $\bar{\sigma}(i) \models P \,\&\, Q$ se e só se $\bar{\sigma}(i) \models P$ e $\bar{\sigma}(i) \models Q$.

Assim, RUQ é interpretado como "existe um instante agora ou no futuro em que Q é verdade e até esse instante (exclusive) R é verdade". Note que P poderia ter sido definido a partir de U da mesma forma que $\Diamond$ pode ser definido a partir de $\Box$ (ie. $\bar{\sigma}(i) \models \Diamond R$ se e só se $\bar{\sigma}(i) \models -\Box -P$ e

$\bar{\sigma}(i) \models RPQ$ se e só se $\bar{\sigma}(i) \models -(-RUQ)$.

A mesma relação pode ser obtida se tomarmos t<k nas definições de P e U acima.

O significado formal dos operadores e algumas propriedades dos mesmos podem ser encontrados em [58].

Algumas fórmulas temporais tem especial relevância

quando estamos interessados em provar propriedades de programas, por exemplo, a fórmula $\Box(I \rightarrow \Box I)$ expressa que I é uma propriedade invariante do programa; a fórmula $\Box(P \rightarrow \Diamond Q)$, expressa propriedades de vida e $\Box(P \rightarrow \Box Q)$, propriedades de segurança.

Se G é um estado do programa e P é uma assertiva imediata, vamos usar a notação $G \models P$ para G satisfaz P.

A fim de que uma assertiva possa fazer referência aos ponteiros das encarnações ativas, com uma notação mais simples, vamos rotular os comandos dos corpos dos processos, supondo que se tenha um esquema de rotulação tal que: os corpos de cada processo são rotulados de forma que cada ocorrência de um comando possa ser identificada pelo rótulo, assim como, ocorrências de comandos em processos de nomes diferentes têm rótulos diferentes.

A linguagem das assertivas imediatas contém símbolos de:

1) predicados unários, at, ins, after, elect, para nos referirmos a pontos do programa.

2) os símbolos de função unária C, T, CC, CS, TS, TC, R, RS, RC, E, ES e EC.

O significado pretendido para cada um desses símbolos é:

C-p: "bag" de todas as mensagens colocadas na porta p,

T-p: "bag" das mensagens na porta p a serem transmitidas,

R-p: "fair-bag" das mensagens já recebidas pela porta p,

E-p: "fair-bag" das mensagens já transmitidas para a porta p,
CS-p (CC-p): "bag" de todas as mensagens (associadas a portas) colocadas na porta p,
TS-p (TC-p): "bag" das mensagens (associadas a portas) na porta p a serem transmitidas;
RS-p (RC-p): "fair-bag" das mensagens (associadas a portas) já recebidas pela porta p,
ES-p (EC-p): "fair-bag" das mensagens (associadas a portas) já transmitidas para a porta p.

É conveniente notar que na definição de estado não há componente correspondente a C,(CS ou CC), mas isso não é uma falta grave, pois o papel de C (CS, CC) aqui, como será visto mais tarde, é o mesmo de variáveis auxiliares, isto é, não tem influência sobre o controle do programa, só auxiliam as provas de propriedades do programa.

3) um predicado binário content, cuja interpretação é content (b,x) é verdade se e só se x está na "bag" b.

4) um predicado binário is-linked, cuja interpretação é: is linked(s,i) é verdade se e só se a porta de saída s está ligada a porta de entrada i.

5) um predicado unário disp, cuja interpretação pretendida é dar a disponibilidade da porta.

6) um predicado unário exp, cuja interpretação pretendida é dizer que o tempo de espera por mensagens expirou. Isto é, G $\models$ exp(t) se e só existe uma encarnação ativa a em G tal que is-receive-(reply)-timeout (a$) e a↑.tab-esp.tempo = val(t) e val(t) é o valor de t quando as variáveis que ocorrem em t são

atribuídos os valores correspondentes em a↑.tab-val.

7) um símbolo de função unária Take, cujo significado pretendido é o da função take da definição de "fair-bag".

8) símbolos de função binária $\cup$, θ , cujos significados pretendidos são os mesmos dos símbolos quando usados no contexto de "bags".

O significado de G$\models$P é o usual, se em P só ocorrem variáveis do programa e/ou variáveis auxiliares ou de "bags"; por exemplo, G$\models\exists$ y content(E-p θ R-p,y) significa que no estado G existe y tal que $\bigvee_{x\in(simples,comb)}$ y $\in$ i-Bagen.x (p), onde i é o nó que declara p, seguindo a notação usada no capítulo 4.

Daremos o significado pretendido de G$\models$at A, G$\models$ ins A, G$\models$ after A, G$\models$ elect A, quando A é um dos comandos apresentados nesse trabalho e G é um estado global.

Seja A um comando rotulado, a: C, ocorrendo no corpo de um processo ou na inicialização de algum nó N do programa.

1) G$\models$ at A se e só se existe uma encarnação ativa de P em G cujo ponteiro do texto aponta para o comando C com rótulo a, no caso de A ocorrer no corpo de P, ou o ponteiro do texto da inicialização de N aponta para o comando C.

Podemos estender esse conceito para quando quisermos explicitar a encarnação do processo P. Assim, se A é um comando rotulado, a:C, e l é um rótulo, G$\models$ at A [l], se e só se existe em G uma encarnação ativa de P, cujos parametros formais estão relacionados, na mesma ordem, com as portas da lista do comando l:*create*,P(...) e o pontexto dessa encarnação aponta para C com rótulo a.

2) G⊨ ins A se e só se G⊨ at B para algum comando rotulado B que faça parte do comando C,

No caso dos nossos comandos, o único comando composto é o de recebimento com opção de sarda por tempo. Assim,

G⊨ ins a: *receive* n *from* pf *when time out* dt *da* b: S se e só se G⊨ at a: *receive* ... ou G⊨ ins b: S.

Como no caso anterior, também podemos estender o conceito de G⊨ ins A para G⊨ ins A [l] onde A é um comando rotulado a: C e l rótulo:
G⊨ ins A [l] se e só se em G existe uma encarnação ativa do processo P onde A ocorre tal que os parâmetros formais de P estão associados, em ordem, as portas que ocorrem no comando l:create P(...) e o ponteiro do texto dessa encarnação aponta para um comando que é parte de C com rótulo a.

As interpretações de at A e ins A quando A é um comando usual de linguagens seqüenciais segue imediatamente da exposição acima.

3) G⊨ elect A se e só se em G existe uma encarnação ativa a em algum nó i, cujo ponteiro do texto aponta para C com rótulo a e a ∈ Prog-i.

Como anteriormente podemos estender esse conceito a G⊨ elect A [l] , se A é um comando rotulado e l um rótulo, analogamente às outras extensões.

A interpretação de after A será dada em termos do comando do texto que contém A mais imediatamente, portanto não vamos nos restringir somente aos comando A apresentados no trabalho.

Se B é A;C então G⊨ after A se e só se G⊨ at C.

Se B é C;A então G⊨ after A se e só se G⊨ after B.

Se B é w: *while*(teste)*do* A então G⊨ after A se e só se G⊨ at w.

Se A é o corpo de um processo P então G⊨ after A se e só se não existe em G encarnação ativa do processo P.

Se B é *receive* n *from* pf *when time out* dt do A então G⊨ after A se e só se G⊨ after B.

Como nos casos anteriores, podemos estender a definição de G⊨ after A para G⊨ after A[l]. Assim, se A é um comando rotulado e 1 é um rótulo, G⊨ after A[l] se e só se em G existe uma encarnação ativa do processo P onde A ocorre, usando as portas (em ordem) que ocorrem na lista de portas do comando 1: *create* P(...) e o ponteiro dessa encarnação aponta para o próximo comando a ser executado imediatamente após o comando A, se houver. Se A é o corpo do processo P, G⊨ after A[l] se e só se não houver em G nenhuma encarnação ativa de P usando as portas que ocorrem na lista do comando rotulado por 1.

Como existe uma correspondência biunívoca entre os rótulos e os comandos que ocorrem no texto do programa, podemos, sem ambigüidade, usar a notação at (l), ins (l), after(l) e elect (1) quando 1 é rótulo de um comando e at (1) [r], ins(l) [r], after (1) [r] para abreviar as extensões feitas anteriormente, onde 1 e r são rótulos e r é rótulo de comando *create*.

Vamos fazer algumas observações sobre encarnações de processos a fim de justificarmos a notação usada e estendermos o conceito de satisfação para assertivas sobre encarnações de processos.

Primeiro, pelas restrições impostas ao uso de por-

tas por encarnações de processos, duas encarnação de processos terão suas execuções em paralelo só se os conjunto de portas usadas por essas encarnações são disjuntos. Também, como o número de portas é fixo e não há ativação recursiva de encarnações de processos, o número máximo de encarnações ativas de processos em qualquer estado é limitado pelo número de vezes que o comando *cheake* ocorre no programa.

No caso de existirem dois comandos *create* usando o mesmo nome de processo P e o mesmo conteúdo das listas de portas, apesar de as encarnações ativas de P não poderem ser executadas ao mesmo tempo, podemos distinguí-las pelos rótulos dos comandos *cheake* respectivos.

Assim, podemos fazer um abuso de linguagem ao afirmar que uma encarnação ativa de um processo fica determinada pelo nome do processo seguido da lista de portas usadas na ativação daquela encarnação e o rótulo do comando *create* que a ativou. Desta forma, usaremos a notação Q(q1,q2,...qn) [1] para nos referirmos a encarnação ativa do processo Q , ativada pelo comando 1: *create* Q(q1,q2,...qn). No entanto, quando não houver possibilidade de confusão omitiremos o rótulo 1. Também podemos usar, sem ambigüidade, a notação Q [1] quando não quisermos fazer referência específica às portas usadas por essa encarnação.

E preciso notar aqui que, da maneira que definimos o estado, não há como distinguir entre duas encarnações do mesmo processo usando exatamente as mesmas portas. Isto é, se no texto ocorre *cheake* P(p) em lugares diferentes, o estado não distingue as diferentes encarnações de P ativadas pelas dife-

rentes ocorrências de *cneake* P(p). Note que no contexto da semântica operacional essa distinção não é necessária já que toda a informação sobre a encarnação de um processo, nós e programa está contida no estado global. No entando, quando nos transpomos para um nível mais abstrato de formalização da semântica, onde a notação de estado não é explicitada e se pretende fazer afirmações sobre o efeito da execução de um programa, a partir do efeito da execução dos componentes do programa, torna-se necessário distinguir uma encarnação da outra.

Assim podemos remediar essa falha, se supusermos que:

1) o texto do programa está rotulado, e

2) em cada estado G e nó i, para cada a, a ∈ G.Ni.conj-encativ,a↑ tem um componente adicional, label, que guarda o rótulo do comando *create* que ativou a encarnação a+. No caso de a+ ser o processo de inicialização do nó i, a↑.label tem um valor qualquer, distinto dos rótulos usados no texto do programa, digamos !

Assim, para a encarnação Q(q1,...,n) [1] e o estado G, definimos:

G⊨ at Q(q1,...,qn) [1] sse em G existe uma encarnação ativa de Q, ativada por 1: *create* Q(q1,...,qn), usando as portas q1,...,qn e o ponteiro desta encarnação aponta para o primeiro comando do corpo de Q;

G⊨ ins Q(q1,...,n) [1] sse em G existe uma encarnação ativa de Q, ativada por 1: *cneake* Q(q1,...,qn) e G⊨ ins A [1] para algum comando rotulado A no corpo de Q;

G⊨ after Q(q1,...,qn) [1] sse G⊨ after A [1] e A é o corpo do processo Q.

(Note que essa definição de G⊨after Q(q1,...,q)[1] não distingue se esta encarnação acabou de existir em G ou já não estava ativa em estados anteriores, ou mesmo nunca esteve ativa.)

G⊨elect Q(q1,...,n) [1] sse G⊨ elect A [1] e A é o primeiro comando do corpo de Q.

No caso da inicialização dos nós, que é como se fosse um processo cuja ativação é feita pelo sistema operacional, usaremos a seguinte notação, onde i é nome de nó: at IN-i, ins IN-i, after IN-i e elect IN-i.

Os significados pretendidos e a interpretação num estado G são similares aos casos em que esse predicado se aplicam a encarnações de processos, e usaremos a notação at A [IN-i], ins A[IN-i], elect A[IN-i] e after A[IN-i], para comando rotulada A ocorrendo na inicialização do nó i. As definições acima poderiam ser expressas usando a notação do capítulo 4, como por exemplo,

1) G⊨at a:A sse existem i e b tais que b ∈ G.Ni.conj-enc-ativ e b$ = a.

2) Se A é o corpo de um processo então G⊨ after A sse para todo i e b , se b ∈ G.Ni.conj-enc-ativ então em b↑.tab-par não ocorrem os parâmetros do aabeçalho do processo do qual A é o corpo.

Podemos estender a definição de ins para ins(nó) e ins(programa) onde nó é um nó declarado como *node* e programa um programa declarado como *program*, da seguinte forma:

G⊨ins(nó) sse G⊨ ins IN-nó ou G⊨ ins Q [r] para algum nome de processo Q, que ocorre no programa e r um rótulo

de comando *create*, cujo processo é Q, ocorrendo no texto da inicialização ou no corpo de processo declarado em nó (note-se que, na definição informal da sintaxe, não foi permitido que um processo declarado globalmente fosse ancestral de nenhum outro).

$G \models ins(programa)$ sse $G \models ins(nó)$ para algum nó declarado no programa.

Vamos listar propriedades de ins, at, elect e after, quando aplicados a alguns comandos usuais de linguagens seqüenciais, para termos mais informação sobre estes predicados. Como em Lamport |49|, introduzimos a releção entre comandos is-part-of, cujo significado é: a relação S is-part-of T significa que a expressão ou comando S é uma subexpressão ou subcomando de T.

Vamos escrever $T = S_1 \boxplus S_2 \ldots \boxplus S_n$ para denotar a seguinte fórmula

$ins\ T \leftrightarrow [ins S_1 \vee ins S_2 \vee \ldots \vee ins S_n] \wedge [S_1$ is-part-of T & S_2 is-part-of T & ... & S_n is-part-of T$]$

Para quaisquer comandos rotulados A, B, S e T, temos as seguintes propriedades:

1) at A $\rightarrow$ ins A

2) elect $\rightarrow$ at A

3) $\Box$ (at (1) $\rightarrow$ $\Box$ at (1) $\vee$ $\mathcal{G}$ elect (1))

4) $\Box$ (elect (1) $\mathcal{P}$ after (1))

5) S is-part-of A & A is-part-of B $\rightarrow$ S is-part-of B

6) Se A é um comando atômico então ins A $\leftrightarrow$ at A

(os comandos de envio, recebimento sem saída por tempo, ligação de portas e ativação de encarnação são considerados atômicos)

7) Se A é um comando composto: *begin* S;T *end* então

(7.1) A = S [+] T

(7.2) at A ↔ at S

(7.3) after A ↔ after T

(7.4) after S ↔ at T

Vamos usar as seguintes definições, se N é um nó:

at N ↔ at IN-N

ins N ↔ ins IN-N $\vee_{r:c \in Create(N)}$ ins Q [r]

onde Create (N) é o conjunto dos comandos r: *create* Q(p1,...,pn) tais que r: *create* Q(p1,...,pn) ocorre no texto de algum processo declarado em N e p1,...,pn ∈ portas-i, para n≥1 (portas-i como definido no capítulo 4).

O predicado content tem as seguintes propriedades:

a) Para cada porta de entrada p

∀z (content(X-p,z) ↔ content(XS-p,z) v content(XC-p,z))

onde X está representando R ou E.

∀z (content(RY-p,z) → content(EY-p,z))

onde Y está representando S ou C.

b) Para cada porta de saída p:

∀z (content(X-p,z) ↔ content(XS-p,z) v content(XC-p,z))

onde X está representando C ou T.

∀z (content(TY-p,z) → content(CY-p,z))

onde Y está representado S ou C.

A interpretação de XC-p no estado G, para X ε {T,R,E}, é notada por G(XC-p) e é:

G(XC-p) = G.Ni.tab-bag-portas-1.comb((p))

onde i é o nó onde G(p) está declarada e 1 é 'entrada' se G(p) é uma porta declarada como *input* ***ponta*** e é 'saída' se G(p) é declarada como ***output*** *ports*.

Similarmente para G(XS-p).

A função Take tem a seguinte propriedade:

$\forall x \forall y$ (x = Take(y) $\rightarrow$ content(y,x))

O predicado is-linked tem as seguintes propriedades:

i) is-linked(s,i) $\rightarrow$ (is-linked(s,x) $\rightarrow$ i = x)

ii) is-linked(s,i) $\rightarrow$ [] is linked(s,i) v <> (is-linked(s, x) & x # i)]

A propriedade i) diz que is-linked é uma função e a propriedade ii) diz que, uma vez feita uma ligação entre portas, esta ligação ficará sempre mantida ou eventualmente a porta de saída será ligada a outra de entrada, i.e., a Única maneira de se desfazer uma ligação entre s e i é ligando s a outra porta.

A interpretação de G $\models$ disp(p) é dada por:

G $\models$ disp(p) sse em G a porta p tem o valor Q na tabela de disponibilidade de portas do nó que possui p. Usaremos a notação disp(p1,...,pn) para abreviar disp(1)&...&disp(pn) e |disp(p1, ...,pn) para abreviar -disp(1)&...&-disp(pn). [Alternativamente poderíamos definir um predicado binário owns, cujo significado pretendido é owns(P,p) é verdade sse o processo P tem uma encarnação ativa usando a porta p e definiríamos disp(p) em

termos de owns por

$$disp(p) \leftrightarrow \bigwedge_{P \in \text{nome de processos}} \neg owns(P,p).$$

O predicado disp tem a seguinte propriedade:

$$\Box[disp(p) \;\&\; \neg\, at\, A(p) \rightarrow \circ\, disp(p)]$$

onde $\neg\, at\, A(p)$ é uma abreviatura para $\bigwedge_{r:c \in A(p)} \neg\, at\, r\, e$

$A(p) = (\, r:$ *cheake* $P(p1,...,pn)\ /\ P \in$ conjunto de nomes de todos os processos declarados no programa, $n \geq 1$ e $p \in (p1,...,pn)\,)$, i.e. $A(p)$ é o conjunto de todos os comandos *cheake* em que P ocorre,

Os predicados is-linked e disp estão relacionados por:

$$[is\text{-}linked(p,x) \;\&\; disp(p) \;\&\; T\text{-}p \neq \emptyset \rightarrow \circ\, is\text{-}linkde(p,x)]$$

onde $T\text{-}p \neq \emptyset$ é uma abreviatura para $\exists x(content(T\text{-}p,x)$.

Agora estamos prontos para dar algumas propriedades dos comandos apresentados neste trabalho. Nestas propriedades, as assertivas representadas pelas letras maiúsculas P,Q,R são assertivas temporais, cujas variáveis livres que varrem as variáveis da linguagem de programação são variáveis locais ao processo onde o comando ocorre. Os símbolos de constantes, que varrem nomes de portas, devem ser nomes de portas locais ao (ou importadas pelo) nó em que o processo onde o comando ocorre é declarado.

Para simplificar a notação, aumentando a legibilidade das fórmulas, usaremos uma notação similar a de Hoare. Porém, ao contrário desta, onde as fórmulas (P)S(Q) são primitivas, no nosso caso a fórmula $P \langle S \rangle Q$ será uma abreviatura para uma fórmula da lógica temporal, a saber:

se P e Q são fórmulas temporais e S um comando P < S > Q é definido por

(elect S & P & ○ after S) → - after S U (after S & Q)

A interpretação das propriedades no nosso modelo operacional será dada informalmente, servindo de esboço para mostrar que essas propriedades são satisfeitas nesse modelo.

a) Comandos para envio de mensagens

Os comandos de envio de mensagem (com ou sem opção *reply to*) correspondem a uma inserção do valor da mensagem na "bag" de mensagem associada à porta de saída referida pelo comando. Assim esse comando tem a seguinte propriedade condicional

Para 1: nend m *to* pf

(a.1) [P k/k ⊕ m] & k = TS-pf < 1 : *send* m *to* pf > P

(a.2) P [k/k ⊕ (m,x)] & k = TC-pf < 1 : nend m *to* pf *reply to* x > P

onde a notação Q[s/t] denota a fórmula obtida pela substituição da variável ou constante s pelo termo t, satisfazendo as restrições normalmente impostas para tais substituições (restrições estas encontradas em qualquer texto introdutório de lógica matemática). No nosso caso, estamos substituindo todas as ocorrências de uma constante por um termo.

Algumas propriedades de vida para esses comandos são:

(a.3) (at (1) → ◇ elect (1))

(a.4) (etct (1) → o after (1))

As propriedades (a.3) e (a.4) acarretam que os comandos de envio sempre terminam, i.e. $\Box$(at (1) $\rightarrow \Diamond$ after (1)).

b) Propriedades para comando de ligação de portas 1: *link* pf *to* x

A execução deste comando estabelece a ligação entre as portas nele mencionadas. Assim temos a seguinte propriedade condicional

(b.1) P < 1 : *link* pf *to* x > P & is-linked(pf,x)

Este comando tem as seguintes propriedades de vida

(b.2) $\Box$ (at (1) $\rightarrow \Diamond$ elect (1))

(b.3) $\Box$ (elect (1) $\rightarrow \emptyset$ after (1))

Assim como no caso anterior, (b.2) e (b.3) acarretam que este comando de ligação sempre termina, i.e.

$\Box$ (at (1) $\rightarrow \Diamond$ after (1)).

Este comando tem as seguintes propriedades de não interferência

(b.4) at (1) & T-pf $\neq \emptyset$ &

$$\bigvee_{y \in \text{all input ports}} (\text{is-linked}(pf,y) \;\&\; y \neq x) \rightarrow o\,(\text{at } (1))$$

onde T-pf $\neq \emptyset$ é uma abreviatura para $\exists z$ content(T-pf,z).

A fórmula (b.4) interpretada no modelo operacional diz que a execução do comando rotulado por 1 não se inicia se ainda há mensagens a serem transmitidas e a parta de saída referida neste comando já está ligada a uma outra porta de entra

da diferente da referida no comando.

(c) Propriedades para comando 1: *link in* p *to* x

A propriedade condicional é a mesma que (b.1).

Como propriedade de vida temos:

(c.1) (e l e c t (1 → o after (1))

(c.2) $\Box$(at (1) & O (disp (p) & $\Box$ $\bigwedge_{r:c \in A(p)}$ -at(r)) →

O elect (1))

Como propriedades de não interferência temos:

(c.3) $\Box$ (elect (1) → $\bigwedge_{r:c \in A(p)}$ - elect(r))

(c.4) $\Box$(at(1) & -disp (p) → o at (1))

O predicado elect (1) tem a seguinte propriedade:

(c.5) elect(1) → disp (p) & $\bigwedge_{y \in \text{all input ports}}$ (is-linked-

(p,y) & y≠x → T-p = 9)

Daremos o esboço da interpretação das propriedades (c.1) a (c.5) no modelo operacional o que justifica que elas são satisfeitas no mesmo.

(c.1) Uma vez o comando rotulado por 1 é escolhido para progredir, ele termina.

(c.2) Se 1: *link in* p *to* x é um próximo comando a ser executado e eventualmente a porta p está disponível e a partir de então nenhum comando *create* usando a porta p será executado, então o comando de ligação começará sua execução.

(c.3) Nenhum comando de ativação de encarnação de processo usando a porta p pode ser executado enquanto 1: *Rinh in* p *to* x é executado.

(c.4) O comando 1: *Rinh in* p *to* x não pode começar a execução se a porta p não está disponível.

(c.5) Se 1: *Rinh in* p *to* x vai ser executado então a porta p está disponível e se p já estiver ligada a alguma outra porta, então não há mais mensagens a serem transmitidas para esta porta de entrada.

Note que pela propriedade 3 da página 139 podemos derivar a seguinte propriedade:

$$\Box(at(1) \;\&\; \Diamond(disp(p) \;\&\; \bigvee_{r:c \in A(p)} at(r)) \rightarrow \Box\, at(1) \vee \bigcirc\, elect(1))$$

(d) Propriedades para 1: *create* P(p1,p2,...,pn)

(d.1) O efeito da execução desse comando é ativar a encarnação de P usando as portas p1,p2,...,pn, nessa ordem. Assim uma propriedade condicional do comando é:

$$R \langle 1: create\ P(p1,p2,...,pn) \rangle R \;\&\; |\; disp(p1,p2,...,pn) \;\&\; at\ P\ [1]$$

Propriedades de vida:

(d.2) $\Box(elect(1) \rightarrow a\ after(1))$

(d.3) $\Box(at(1) \;\&\; \Diamond(disp(p1,p2,...,pn) \;\&\; \Box \bigwedge_{r:c \notin Comum(1)} - at(r)) \rightarrow \Diamond\, elect(1))$

(d.4) $\Box(elect(1) \rightarrow (\bigwedge_{r:c \in Comum(1)} \bigcup_{Lin\ Comun(1)} - elect(r)))$

onde Comum (1) = { r:c / is-create (c) & arg(c) (pi,...,pn) $\neq \emptyset$ & r $\neq$ 1 }

Lin Comun (1) = { r:c / is-link in (c) & porta-origem (c) $\in$ (p1,p2,...,pn) } , usando a notação do capítulo 4.

(d.5) $\Box$ (at (1) & $\bigvee_{1 \leq i \leq n}$ - disp (pi) $\rightarrow$

a (at (1) & - elect (1)))

O predicado elect (1) tem a seguinte propriedade adicional

(d.6) elect(1) $\rightarrow$ disp (p1,p2,...,pn)

O esboço da interpretação das propriedades (d.2) a (d.6) no modelo operacional será dado a seguir:

(d.2): idem a (c.1))

(d.3): Se 1: *create* P(p1,p2,...,pn) é um próximo comando a ser executado e eventualmente as portas p1,p2,...,pn estão disponíveis e sempre, a partir de então, nenhum outro comando *create* usando alguma das portas p1,...,pn, é um próximo comando a ser executado, então eventualmente o comando rotulado por 1 começará sua execução.

(d.4): 1:*create* P(p1,...,pn) não é executado simultaneamente com comandos de ativação de encarnações que usam pelo menos alguma das portas p1,...,pn nem com comandos de ligação ocorrendo na inicialização cuja porta de saída é uma das portas p1,...,pn.

A propriedade (d.5) afirma que 1:*create* P(p1,...,pn) fica bloqueada se alguma das portas p1,...,pn não está disponível.

A propriedade (d.6) diz que se 1: *create* P(p1,...,pn), não é

executado, então todas as portas p1,...,pn estão disponíveis.

(e) Propriedades para 1: *receive* n *from* pf e

r: *receive* n *from* pf *reply to* x

O efeito da execução desses comandos consiste em atribuir à variável n(e à variável x) um valor tomada da porta de entrada caso existam mensagens a serem recebidas.

Assim, eles têm as seguintes propriedades condicionais:

(e.1) P [y,n/y ⊕ k,k] & y = RS-pf & k = Take (ES-pf θ RS-pf)
< 1: *receive* n *from* pf > P

(e.1) P [y,n,x/y ⊕ k,ki,k2] & y = RC-pf & k = (k1,k2) & k = Take(EC-pf θ RC-pf)
<r: *receive* n *from* pf *reply to* x > P

Esses comandos têm as seguintes propriedades de vida:

(e.2) (at (1) & □ ES-pf ∃ RS-pf # ∅ → ◇ elect (1))

(e'.2) (at (r) & □ (EC-pf ∃ RC-pf ≠ ∅) → ◇ elect (r))

(e.3) (elect (1) → ◇ after (1))

(e.4) (elect (r) → ◇ after (r))

O precidado elect (1) (elect (r)) tem a propriedade:

(e.4) elect (1) ↔ ES-pf θ RS-pf ≠ ∅ & at (1)

(e'.4) elect (r) ↔ EC-pf θ RC-pf ≠ ∅ & at(r)

Daremos o esboço da interpretação das fórmulas (e.2) a (e.4) no modelo operacional.

A propriedade (e.2) ((e'.2)) diz que um comando de recebimento começará sua execução se eventualmente houver mensagens na porta de entrada a serem recebidas.

A propriedade (e.3) ((e'.3)) diz que a execução do recebimento, uma vez iniciada sempre termina.

A propriedade (e.4) ((4) diz que a execução do recebimento se inicia se e só se existem mensagens a serem recebidas na porta de entrada.

A propriedade (e.4) ((e'.4)) complementa a propriedade (e.3) ((e'.3)).

(f) Propriedades para 1: *receive* n *from* pf

when Rime oúk dt *do* S

O efeito da execução desse comando é o mesmo que o do comando de recebimento sem opção de saída por tempo caso existam mensagens a serem recebidas enquanto o tempo de espera não se tenha esgotado. Caso contrário, tem o mesmo efeito que a execução de S. Assim, temos as seçuinte propriedades condicionais:

(f.1) (P[y,n / y ∈ k,k] & y = RS-pf & k = Take (ES-pf Θ RS-pf) & ES-pf Θ RS-pf ≠ Ø & - exp(dt)

< 1: *receive* n *from* pf *when Rime auk* dt *do* S > Q &(y = RS-pf & P [y,n / y ∈ k,k] → Q[y,n / y ∈ k,k])))

→ P < 1: *receive* n *from* pf *when time out* dt *do* S > Q

(f.2) P & ES-pf ∈ RS-pf = Ø & exp (dt) < S > Q →

→ P < 1: *receive* n *from* pf *when time ouk* dt *do* S > Q

Esse comando tem as seguintes propriedades de vida:

(f.3) $\Box$ (elect (1) & $\Diamond$ (ES - pf θ RS-pf # $\emptyset$) &
& - exp (dt)) $\rightarrow$ $\Diamond$ after (1))

(f.4) $\Box$ (elect (1) & exp (dt) & ES-pf 8 RS-pf = $\emptyset$
$\rightarrow$ ($\bigcirc$ after S $\rightarrow$ $\Diamond$ after (1)))

(f.5) $\Box$ (at (1) $\rightarrow$ $\Diamond$ elect (1))

0 predicado elect (1) tem as seguintes propriedades adicionais.

(f.6) elect (1) & exp(dt) & ES-pf 8 RS-pf = $\emptyset$ $\rightarrow$
$\rightarrow$ σ (at S)

(f.7) elect (1) & ES-pf 8 RS-pf = $\emptyset$ & -exp (dt) $\rightarrow$
$\rightarrow$ σ at (1)

Daremos agora o esboço da interpretação de (f.3) a (f.7) no modelo operacional.

A propriedade (f.3) diz que a execução do comando rotulado por 1 termina se existem mensagens a serem recebidas enquanto o tempo de espera não se esgota.

A propriedade (f.4) diz que se o tempo do espera se esgota sem que haja mensagens a serem recebidas e a execução de S termina então a execução do comando rotulado por 1 termina.

A propriedade (f.5) diz que se o controle chega até 1, então o comando rotulado por 1 eventualmente começará sua execução.

A propriedade (f.6) diz que se o tempo de espera esgota e não há mensagens a serem recebidas então o controle passa para o primeiro comando de S.

A propriedade (f.7) diz que o controle permanece no início do comando com opção de saída por tempo rotulado por 1 se o tempo de espera não se esgotou e ainda não há mensagens a serem recebidas.

Para encarnações ativas de processos temos as seguintes propriedades:

(g.1) $\Box$ (ins P (p1,p2,...,pn) [1] &

& (ins P(p1,p2,...,pn) [1] $\rightarrow$ $\Diamond$ after P(p1,p2,...,pn) [1] $\rightarrow$

$\Diamond$ after P(p1,p2,...,pn) [1] & disp (p1,p2,...,pn)))

A fórmula (g.1) interpretada no nosso modelo operacional diz que se uma encarnação eventualmente deixa de ser ativa então existe um momento em que ela não está ativa e todas as portas que foram usadas por ela estão disponíveis.

Essa propriedade nos parece relevante, já que o predicado after estendido a encarnações de processos não distingue o primeiro estado em que a encarnação deixa de estar ativa dos outros em que ela não está ativa.

(g.2) Para rótulos 1 e r tais que r:c ε Comum (1) então:

$\Box$ - (ins P [1] & ins Q [r]) onde nome-proc (c) = Q

Isto é, duas encarnações de processos usando alguma porta em comum não podem ser executadas simultaneamente.

O processo de inicialização de cada nó i tem as seguintes propriedades:

(h.1) (at IN-i $\rightarrow$

$$\left(\bigwedge_{p \,\varepsilon\, \text{1-portas-i}} \text{disp}(p) \;\&\; \bigwedge_{p \,\varepsilon\, \text{input ports-i}} E\text{-}p = \emptyset \right.$$

$$\& \bigwedge_{y \in \text{output ports-i}} \& \bigwedge_{x \in \text{all input ports}} (\neg \text{is-linked}(y,x) \; \& \; C\text{-}y = \emptyset))$$

onde l-portas-i é o conjunto dos nomes das portas declarados como *input ports* e *output ports* no nó i, output ports-i é o conjunto dos nomes das portas declaradas como *output ports* no nó i e input ports-i é o conjunto dos nomes das portas declaradas como *input ports* no nó i e all input ports é o conjunto dos nomes de portas declaradas como *all input ports* no programa.

(h.2) $\Box$ (at IN-i $\rightarrow$ **a** elect IN-i)

O esboço da interpretação das propriedades (h.1) e (h.2) no nosso modelo operacional é como se segue:

A propriedade (h,1) diz que todas as portas declaradas no nó i como de entrada ou de saída estão disponiveis e as "bags" das portas de saída e de entrada desse nó estão vazias no começo da inicialização do nó i.

A propriedade (h.2) diz que sempre a inicialização do nó será tomada para progredir.

Pela extensão da definição de elect da página 137 e (h.2) podemos inferir que IN-i sempre começará sua execução, para todo nó i.

A confiabilidade da rede pode ser expressa da seguinte forma:

(i.1) is-linked (p,e) & content (TS-p,x) $\rightarrow$

$\bigcirc$ content (ES-e,x)

(i2) is-linked (p,e) & content(TC-p,x) $\rightarrow \bigcirc$ content (EC-e,x)

Isto é (i.1) e (i.2) interpretadas no modelo operacional dizem que se a porta p está ligada a porta e, e há mensagem x, na porta p a ser transmitida, então eventualmente essa mensagem estará na porta e (i.e. na "bag" das mensagens que já foram transmitidas para a porta e).

CAPITULO VI

CONCLUSÃO

Neste trabalho apresentamos um conjunto de mecanismos usados em programação concorrente distribuída. Estes mecanismos expressam alguns dos conceitos que nesses últimos dez anos foram introduzidos nesta área, tais como sincronização e comunicação de processos através de mensagens, endereçamento indireto, criação e topologia dinâmicas.

No Capítulo III definimos a sintaxe dos mecanismos, de forma cuidadosa porém informal.

As restrições impostas à sintaxe, as decisões de cunho semântico (tais como exclusão mútua no uso de portas pelas encarnações ativas de processos), e as sugestões para disciplinar o uso dos mecanismos nos parecem (através de um número exaustivo de exemplos) suficientes para que esses mecanismos possam ser usados de maneira confiável.

No Capítulo IV a formalização da semântica foi feita operacionalmente, supondo dada a sintaxe abstrata.

A descrição da semântica operacional foi feita em três seções, na primeira nos baseamos em técnicas de VDL para representar o estado como uma árvore não ordenada; na segunda, descrevemos o estado como um objeto de um tipo abstrato de dados e demos a definição desse tipo usando técnicas de construção de tipos a partir de outros pré definidos, como em Pascal; na terceira apresentamos a relação de transição de estados u-

sando como metalinguagem, linguagem natural suficientemente estruturada para descrever os procedimentos que, numa linguagem de programação do tipo Pascal, teriam como entrada o estado atual e como saída o próximo estado.

Apresentamos propriedades dos comandos expressas numa modalidade de lógica temporal, e satisfeitas pelo modelo operacional.

No que diz respeito aos objetivos citados na introdução, acreditamos que fornecemos subsídios necessários para uma implementação uniforme dos mecanismos aqui apresentados e esclarecemos alguns conceitos usuais em programação distribuída que facilitarão o projeto de linguagens usando esses mecanismos.

As propriedades listadas no Capítulo V, expressas numa linguagem mais abstrata do que a usada no Capitulo IV, fornecem uma aproximação à semântica axiomática. Muito há ainda a se fazer nessa direção no sentido de se obter uma axiomatização completa para os mecanismos e uma metodologia de prova de correção de programas usando os mesmos.

Outra perspectiva para futura pesquisa na área seria a viabilização dos mecanismos se introduzissemos: 1) a possibílidade de recursão na ativação de encarnação de processos, já que recursão é uma técnica que, em geral, facilita a maneira de programa; 2) a não limitação de um número fixo de portas por nó e ou a alocação dinâmica de portas, aumentando assim o espectro de aplicações dos mecanismos, sem sacrificar a autonomia dos módulos de processamento e considerassemos mensagens com tipos, que é uma situação encontrada freqüentemente na

realidade.

Das três extensões sugeridas, a Última nos parece de solução não muito difícil e a que menos mudaria a estrutura essencial da semântica dos mecanismos. As outras duas, se viáveis, deverão introduzir modificações que vão certamente aumentar a dificuldade na fase de implementação dos mecanismos.

BIBLIOGRAFIA

|1| Andrews,G.R. "Syncronizing Resources". ACM Transactions on Programming Languages and Systems 3,4, pp.405-430, (1981).

|2| Andrews,G.R. "The distributed programming languages SR - Mecanisms, design and implementation". Software Practice & Experience 12,8, pp. 719-754 (1982).

|3| Andrews,G.R. e Schneider,F.B. "Concepts and notations for concurrent programming",Computing Surveys 15, 1, pp. 3-43 (1983).

|4| Ashcroft,E.A. "Proving assertions about parallel programs", J.Computing Systems 10, 1, pp. 110-135 (1975).

|5| Apt,K.R., Francez,N. e de Roever,W.P. "A proof system for communicating sequential processes". ACM Transactions on Programming Languages and Systems 2,3, pp.359-385, (1980).

|6| Ball,J.E., Feldman,J., Law,J., Rashid,R., Rovner,P. "RIG, Rochester's Intelligent Gateway: System Overview", IEEE Transactions on Software Engineering SE-2, 4 (1976).

|7| Balzer,R.M. "Ports - A method for dynamic interprogramming communication and job control. Em Proc. AFIPS Spring Jt. Computer Conference. (Atlantic City, N.J. Maio 1971), 38 AFIPS Press, Arlington, Va.pp. 485-489 (1971).

|8| Baskett,F., Howard,J.H. e Montagne,J.T. "Task communication in DEMOS". Em Proc, 6th Symp. on Operating Systems Principles (West Lafayette, Indiana, Nov. 16-18, 1988)) ACM, N.Y. pp.23-31 (1977).

|9| Best,E. "Relational semantics of concurrent programs (with some applications). Em Proc. IFIP WG 22.2 Conferente. North Holland Publ., Amsterdam (1982).

|10| Brinch Hansen,P. "Structured Multiprogramming", Comm. ACM 15, 7, pp. 574-578 (1972).

|11| Brinch Hansen,P. "Operating Systems Principles". Prentice Hall, Englewood Cliffs, N.J. (1973).

|12| Brinch Hansen, P. "Concurrent Programming Concepts", ACM Computing Surveys 5, 4, pp.223-245 (1973).

|13| Brinch Hansen,P. "The programming language concurrent Paçcal". IEEE Transactions on Software Engineering SE-'?2, pp. 199-206 (1975).

|14| Brinch Hansen,P. "Distributed Processes: A concurrent programming concept", Comm. ACM 21, 11, pp.934-941 (1978).

|15| Brinch Hansen,P. "Edison: A multiprocessor language", Software Practice and Experience 11, 4 pp. 325-361 (1981).

|16| Britton,E.E., Stickel,M.E. "A Interprocess Communication Facility for Distributed Applications". Em Proc. of the 1980 COMPCON Conference on Distributed Computing (Fev 1980).

|17| Campbell,R.H. e Habermann,A.N. "The specification of process syncronization by path expressions" Lecture notes in Computer Science, vol 16, pp. 89-102. Springer Ver lag, N.Y. (1974).

|18| Campbell,R.H. "Path expressions: A technique for specifying process syncronization". Tese de doutorado, Computing Laboratory, University of Newcastle upon Tyne, Agosto 1976.

|19| Campbell,R.H. e Koltad,R.B. "Path expressions in Pascal". Em Proc. 4th Int. Conf. on Soft. Engineering (Munique, 17-19 Set, 1979) IEEE, NY, pp. 212-219 (1979).

|20| Cheriton,D.R., Malcolm,M.A., Melen,L.S., Sager,G.R. "THOTH, a Portable Real-Time Operating System". Comm. ACM 22, 2 (1979).

|21| Clint,M. "Program proving: co-rotine" Acta Informatica 2, 1, pp. 50-63 (1973).

|22| Conway,M.E. "Design of a separable transition diagram compiler". Comm. ACM 6, 7, pp. 396-408 (1963).

|23| Conway,M.E. "A multiprocessor system design". Em Proc. AFIPS, Fall Jt.Computer Conference (Las Vegas, Nevada, Nov.1963), vol 24, pp. 139-146 Spartan Books, Baltimore, Maryland (1963).

|24| Cook,R.P. "MOD - A language for Distributed programming" IEEE Trans. Soft, Eng. SE-6,6, pp. 563-571 (1980).

|25| Cunha, Paulo, "Design and Analysis of Message Oriented Programs". Dissertação de doutorado, University of Waterloo, Ontario, 1981.

|26| Dennis,J.B. e van Horn,E.C. "Programming semantics for multiprogrammed computations". Comm. ACM 9, 3, pp. 143-155 (1966).

|27| Dijkstra,E.W.. "The structure of "THE" multiprogramming system" Comm.ACM 11, 5, pp. 341-346 (1968).

|28| Dijkstra,E.W. "Cooperating sequential processes". Em F. Genuys (Ed.), Programming Languages. Academic Press, N.Y. (1968).

|29| Dijkstra,E.W. "Guarded Comands, nondeterminacy and formal derivation of programs. Comm. ACM 18, 8, pp. 453-457, (1975).

|30| Feldman,J.A. "High-level programming for distributed computing. Comm.ACM 22, 6, pp.353-368 (1969).

|31| Flon,L. e Habermann,A.N. "Towards the construction of verifiable software systems". Em Proc. ACM Conf. Data, SIGPLAN Not. 8, 2, pp. 141-148 (1976).

|32| Floyd,R.W. "Assigning Meanings to Programs". Em Proc. Am. Math. Soc. Symp. on Applied Mathematics, vol. 19, pp.19-31 (1967).

|33| Gabbay.D., Pnuelli,A., Shelah,S. e Stavi,J. "On the temporal analysis of fairness". Em Conf. Records of the 7th Annual ACM Symposium on Principles of Programming Languages (Las Vegas, Nev,, Jan. 28-30, 1980) pp. 163-173, ACM (1980).

|34| Gentleman,W.M. "Message passing between sequential processes: the reply primitive and administrator concept". Software Practice and Experience, 11, 5, pp. 435-466 (1981).

|35| Good,D.I., Cohen,R.M. e Keeton-Williams,J. "Principles of proving concurrent programs in Gypsy". Em Proc. of the 6th Annual ACM Symposium on Principles of Programming Languages (San Antonio, Texas, Ja 29-31, 1979), pp.42-52, ACM (1979).

|36| Guttag,J. "Notes on Type Abstraction (version 2) IEEE Transactions on Software Engineering SE-6,1, pp. 13-23 (Jan 1980).

|37| Habermann,A.N. "Path expressions". Dept of Computer Science, Carnigie-Mellon University. Pittsburg, Pensylvania (June 1975).

|38| Hanson,D.R. e Grisworld,R.W. "The SL5 procedure mechanism". Comm. ACM 21, 5, pp. 392-400 (Maio 1978).

|39| Hoare,C.A.R. "An axiomatic basis for computer programming". Comm. ACM 12, 10, pp.576-580, 583 (Out.1969).

|40| Hoare,C.A.R. "Notes on Data Structuring". Em Structured Programming, Academic Press, London e N.Y., pp 83-174 (1972).

|41| Hoare,C.A.R. "Towards a theory of parallel programming". Em C.A.R. Hoare e R.H.Perrot (Eds), Operating Systems Techniques, Academic Press, N.Y. (1972).

|42| Hoare,C.A.R. "Monitors: an operation systems structuring concept". Comm. ACM 17, 10, pp. 549-557 (Out.1974).

|43| Hoare,C.A.R. "Communicating Sequential Processes". Comm. ACM 21, 8, pp. 666-677 (Agosto 1978).

|44| Jones,A.K. e Schwarz,P. "Experience using multiprocessor systems. A status report". ACM Computer Surveys, 12, 2 pp, 121-165 (Junho 1980).

|45| Keller,R.M. "Formal verification of parallel programs". Comm.ACM 19, 7, pp.371-384 (Julho 1976).

|46| Knott, "A proposal for certain process management and intercommunication primitives". Operating Systems Review (Outubro 74, Jan.75).

|47| Lamport,L. "Proving the correctness of multiprocess programs" IEEE Trans. Software Engineering SE-3, 2, pp. 125-143 (Março 1977).

|48| Lamport,L."'Sometime' is sometimes 'not never': on temporal logic of programs". Em Conf. Rec. of 7th Ann. ACM. Symposium on Principles of Programming Languages (Las Vegas, Nev. Jan. 28-30 1980), pp. 174-185, ACM (1980).

|49| Lamport,L. "The 'Hoare Logic' of concurrent programs", Acta Inf. 14, 1, pp. 21-37 (June 1980).

|50| Lamport,L. e Schneider,F.B. "The 'Hoare Logic' of CSP and all that". Tech.Rep. TR 82-490, Dept. Computer Science, Cornell University. (Maio 1982).

|51| Lampson,B.W. e Redell,D.D. "Experience with processes and monitors in Mesa", Comm, ACM 23, 2, pp. 105-117 (Fev. 1980).

|52| Lauer,P.E. e Campbell,R.H. "Formal semantics of a class of high level primitives for coordinating concurrent processes", Acta Inf. 5, pp. 297-332 (1975).

|53| Lauer,P.E. e Shields,M.W. "Abstract specification of resource accessing disciplines: Adequacy, starvation, priority and interrupts", SIGPLAN Notices 13, 12, pp. 41-59 (Dez. 1978).

|54| Lesser,V., Serrain,D. e Bonar,J. "FCL: A process-oriented Job Control Language". Em Proc, 1st. International Conference on Distributed Computing Systems (Out.1979).

|55| Levin,G.M. e Gries,D. "A proof system for communicating sequential processes", Acta Inf 15, pp. 281-302 (1981).

|56| Liskov,B. "Primitives for Distributed Computing". Em Proc. of the 7th Symposium of Operating Systems Principles. (Pacific Grove, 1979). N.Y. ACM, pp. 33-42 (1979).

|57| Liskov,B.L. e Scheifler,R. "Guardians and actions linguistic support for robust, distributed programs". Em Proc. 9th ACM Symposium on Principles of Programming Languages (Albuquerque, N.Mexico, Jan. 25-27 1982).

|58| Manna,Z. e Pnuelli,A. "The correctness problems in Computer Science", (R.S. Boyer e J.S. Moore Eds) International Lecture Series in Computer Science, Academic Press, Londres (1981).

|59| Manna,Z. e Pnuelli,A. "How to cook a temporal proof system for your pet language". Em Proc. of the 10th Symposium on Principles of Frogramming Languages (Austin, Texas) N.Y. ACM (1983).

|60| Manning.E., Livesey,J. e Tokuche,H. "Interprocesses Communication in Distributed Systems: one view". Em Proc. IFIP (1980).

|61| Misra,J. e Chandy,K.M. "Proofs of newtorks of processes", IEEE Transactions on Soft.Eng. SE-7, 4, pp.417-426 (Julho 1981).

|62| Misra,J., Chandy,K.M. e Smith,T. "Proving safety and liveness of communicating processes with examples". Em Proc. Symp. on Principles of Distributed Computing (Ottawa, Canadá, Agosto 1982), pp. 201-208, ACM,N.Y. (1982).

|63| Mtchell,J.G., Maybury,W. e Sweet,R. "Mesa language manual version 5.0". Rep CSC-79-3, Xerox Palo Alto Research Center, (Abril 1979).

|64| Nelson,B.J. "Remote procedure call". Dissertação de doutorato, Rep CMC-CS-81-119, Dept. of Computer Science, Carnegie-Mellon Univ. (Maio 1981).

|65| Nygaard,K. e Dahl,O.J. "The development of the SIMULA languages", Preprints ACM SIGPLAN History of Programming Languages Conference, SIGPLAN Not. 13, 8, pp. 245-272 (Agosto 1978).

|66| Ousterhout,J., Scelza,D., Shinder,P. "Medusa: An experiment in distributed operating system structure". Comm. ACM 23, 2, (Fev. 1980).

|67| Owicki,S. e Gries,D. "An axiomatic proof technique for parallel programs". Acta Inf. 6, 4, pp. 319-340 (1976).

|68| Owicki,S. e Gries,D. "Verifying properties of parallel programs: an axiomatic approach", Comm. ACM 19, 5, pp. 279-285 (Maio 1976).

|69| Owicki,S. e Lamport,L., "Proving liveness properties of concurrent programs". ACM Trans. on Programming Languages and Systems, 4, 3, pp. 455-495 (Julho 1982).

|70| Parnas,D.L. "On the criteria to be used in decomposing systems into modules". Comm. ACM, 15, 12, pp. 1053-1058 (Dez.1972).

|71| Peterson,G.L. "Myths about the mutual exclusion problem", Inform. Process. Lect., 12, 3, pp. 15-116 (Junho 1981).

|72| Pnueli,A. "The temporal logic of programs". Em Proc. of the 18th Symposium on the Foundations of Computer Science (Providence, Nov. 1977), pp. 46-57, IEEE (1977).

|73| Pnueli,A. "The temporal semantics of concurrent programs". Em Lecture Notes in Computer Science, vol.70. Semantics of concurrent computation, pp. 1-20, Springer-Verlag, N.Y. (1979).

|74| Rashid,R. e Robertson,G., "Accent: A communication oriented network operating system Kernel". Pech. Rep. Dep. of Computer Science Carnigie-Mellon University (Abril 1981).

|75| Ritchie,D.M. e Thompson,K. "Phe UNIX time sharing system". Comm. ACM, 17, 7, pp. 365-375 (Julho 1974).

|76| Ruggiero,W.V. e Bressan,G. "Um modelo de programação para sistemas distribuídos", Revista Brasileira de Computação , 2, 3, pp. 131-194 (1982).

|77| Schlichting,R.D. e Schneider,F.B. "Using message passing for distributed programming: Proof rules and disciplines". Tech.Rep. TR-82-491, Dept, of Computer Science, Cornell University (Maio 1982).

|78| Shaw,A.C. "Software specifications languages based on regular expressions". Em W.E. Riddle e R.E. Fairley (Eds). Software Development Tools, pp. 148-175, Springer-Verlag, N.Y. (1980).

|79| Shields,M.W. "Adequate path expressions". Em Proc. International Symp. on Semantics of Concurrent Cornputation, Lecture Notes in Computer Science, vol.70, pp. 249-265, Springer-Verlag.

|80| Solomon,M.H. e Finkel,R.A. "The Roscoe Distributed Design of a Port-oriented Operating System". COINS Tech. Rep. Dept. of Computer and Information Science, University of Massachusets (Out. 1982).

|81| U.S. Department of Defense. "Programming language ADA: reference manual". Lecture Notes in Computer Sciences, vol.106, Springer-Verlag, N.Y., 1981.

|82| Van Wijngaarden,A., Maillonx,B.J. Peck,J.L., Koster,C. H.A., Sintzoff,M., Lindsey,C.H., Meertens,L.G.L.T. e Fisker,R.G. "Revised report on the algorithm language 'ALGOL 68". Acta Inf. 5, 1-3, pp. 1-236 (1975).

|83| Vinter,S., Ramaritham,K. e Stemple,D. "Protecting objects through the use of ports". Tech.Rep. 82-83 Computer and Information Science Dept. University of Massachusetts at Amherst (1982).

|84| Ward,S. "TRIX: A network operating systems", MIT Technical Report (Dec. 1979).

|85| Wegner,P. "The Vienna Definition Language", ACM Computing Surveys, 4, 1 (1972).

|86| Wirth, N. "Modula: A language for modular multiprogramming". Software Pract, Exp. 7, pp. 3-35 (1977)

|87| Wirth,N. "Programming in Modula-2". Springer-Verlag, N.Y. (1982).

|88| Wirht,N. "Programação sistemática". Ed. Campos, (1982).