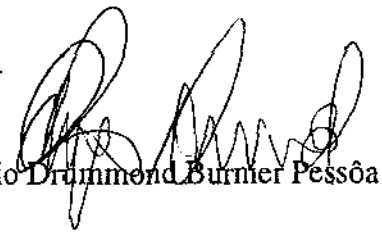
CONCORRÊNCIA EM LINGUAGENS DE COMANDOS

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Mauricio Alberto Fernández e aprovada pela Comissão Julgadora.

Campinas, 30 de setembro de 1994.



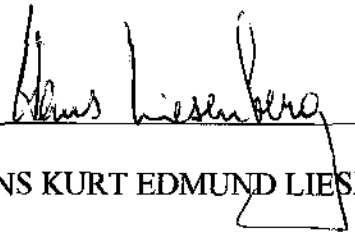
Prof. Dr. Rogério Drummond Burnier Pessôa de Melho Filho

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

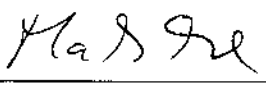
Tese de Mestrado defendida e aprovada em 30 de Setembro de 1994

pela Banca Examinadora composta pelos Profs. Drs.

Prof (a). Dr (a). ROGÉRIO DRUMMOND BURNIER PESSOA DE MELLO FILHO



Prof (a). Dr (a). HANS KURT EDMUND LIESENBERG



Prof (a). Dr (a). MARKUS ENDLER

Dedicado a María de Luján Sánchez

Agradecimentos

O Departamento de Ciência da Computação da UNICAMP colocou à disposição os recursos necessários para fazer este trabalho. O CNPq financiou parcialmente a sua elaboração.

Como estrangeiro não posso deixar de agradecer ao Brasil por ter me brindado a oportunidade de fazer este mestrado.

O amor e a compreensão da minha companheira, a quem dedico este trabalho, têm sido muito importante para mim em tantas coisas da vida.

Celso Gonçalves Jr. me deu força quando mais a precisei. Cassius di Cianni, Bill Coutinho e Teles discutiram muitas idéias deste trabalho. Todos os colegas do A_Hand me deram seu apoio.

Meu orientador, Rogério Drummond, tem me animado com sua vontade para levar as cóisas para a frente e sua iniciativa para fazer coisas novas.

Grande parte deste mestrado está nas horas de estudo de Teoria da Computação com os amigos Marcus Vinícius de Andrade, Mario Harada, Fabio Nogueira e Thierson Couto.

Não posso deixar de lembrar a amizade do Fred Cavalcante nos primeiros meses em Campinas.

Sandra Brisolla e Hernán Piñón Arias, cada um por seu lado, são para mim um exemplo de solidariedade, minha gratidão com eles é imensa.

Resumo

A execução de programas é uma das formas de reutilização de *software* mais bem sucedidas graças a um módulo que atua entre o usuário e o núcleo do sistema operacional. Estes módulos permitem que o usuário tenha acesso a uma máquina virtual de nível superior àquela fornecida pelos serviços do sistema. Originalmente estes módulos processavam ordens que chegavam ao computador pela leitora de cartões, na época do time sharing surgem as linguagens de comandos que definem a sintaxe que o usuário deve utilizar para executar programas a partir de seu terminal; hoje existem interfaces gráficas de usuário para desempenhar estas funções. Na arquitetura do Unix o interpretador da linguagem de comandos é um módulo separado do núcleo do sistema, o que permitiu a aparição de várias linguagens de comandos para esse sistema.

Na prática as linguagens de comandos do Unix têm se desempenhado como ótimas linguagens de desenvolvimento rápido de programas. Arquivos contendo comandos podem ser executados da mesma forma como são executados outros programas. Recursos sintáticos para executar vários comandos concorrentemente, de forma que a saída de um alimente a entrada de outro, permitem a construção de programas mediante a utilização de peças de funcionalidade mais elementar. Além disto, a interatividade agiliza o ciclo codificação-teste.

Este trabalho propõe uma extensão ao modelo de concorrência das linguagens de comandos do Unix que sirva de base para a definição de uma linguagem de comandos distribuída. As principais características do modelo proposto são: suporte a vários paradigmas de interação entre processos distribuídos, recursos definicionais, recursos funcionais, abstração procedural e suporte à concorrência interna.

Abstract

Every operating system architecture includes a module that simplifies program execution by users, converting program execution in one of the most succesful cases of software reusablity. This module have evolved from early job control languages to graphical interfaces. Command languages came with time sharing systems and were the means used to type commands into the system.

Unix command languages have proven to be excellent languages for rapid program development. Files containing shell commands may be executed like any other program. Sintactic devices for concurrent program execution, with the output of one program feeding the input of another, allow program construction by reusing simpler pieces. Besides, the interactive nature speeds up the coding-testing cycle.

An extension to the concurrency model of Unix command languages is herewith proposed. This extension should be the basis for a further especification of a distributed command language. The highlights of the proposed model are: support to various paradigms of distributed processes interaction, features for functional and definitional programming, procedural abstraction and support for internal concurrency.

Lista de figuras

Lista de tabelas

Capítulo 1

Linguagens de Comandos e Desenvolvimento de *Software*

Às vezes, basta-me uma partícula que se abre no meio de uma paisagem incongruente, um aflorar de luzes na neblina, o diálogo de dois passantes que se encontram no vaivém, para pensar que partindo dali construirei pedaço por pedaço a cidade perfeita, feita de fragmentos misturados com o resto, de instantes separados por intervalos, de sinais que alguém envia e não sabe quem capta.

Italo Calvino: *As cidades invisíveis.*

Dentre as contribuições que os sistemas operacionais vieram a trazer encontram-se a recuperação, carga e execução de programas. Assim, o termo *programa* adquire mais um significado: deixa de significar apenas uma especificação executável da solução de um problema, e passa também a ser um produto imaterial que pode ser reutilizado infinitas vezes, e até comercializado. Mediante o uso inteligente de algum mecanismo de parametrização um mesmo programa pode adaptar-se a diversas situações.

Assim os programadores atualmente (re) utilizam muitos programas no processo de desenvolvimento de *software*. Esta reutilização acontece de duas formas:

- ferramentas que auxiliam o programador no desenvolvimento de *software* cujo código não faz parte do produto final e
- programas que são ativados durante a execução do sistema.

Editores, compiladores, depuradores são exemplos do primeiro grupo. No segundo grupo têm-se programas ordenadores (*sorts*), conversores de formatos, editores não interativos e programas que fazem a impressão .

O programador utiliza-se de uma linguagem de comandos para ativar a execução dos programas. Durante a execução do sistema a linguagem de comandos continua a desempenhar um papel importante, já que muitas vezes é o próprio interpretador da

Na definição de um modelo de computação para uma linguagem de comandos tem que se estabelecer a concorrência desse modelo.

linguagem de comandos quem ativa a execução dos diversos programas que compõem um sistema.

Neste trabalho os programas são enxergados como pacotes reutilizáveis de código que fazem parte de programas mais complexos, ao invés de considerá-los unidades isoladas que só precisam de uma linguagem de comandos para entrar em execução. Interpretadores de linguagens de comandos podem ser o cimento que junta componentes de *software*, da mesma maneira que o sistema de execução de uma linguagem é o agente que ativa funções e procedimentos. Contudo, a utilização de linguagens de comandos para o desenvolvimento de programas merece ser analisada mais detalhadamente.

A reutilização de código encapsulado em funções, procedimentos e tipos abstratos de dados foi bem sucedida pelo fato de que estes conceitos viabilizaram um mecanismo de abstração com uma semântica clara e consistente. Em princípio a utilização dos programas poderia ser feita sob o mesmo modelo de chamadas a procedimentos: o sistema operacional apenas deveria oferecer um mecanismo de passagem de parâmetros para o programa a ser executado, e outro mecanismo que permita a devolução de algum valor quando da finalização do programa. No entanto, este modelo forçaria e execução sequencial dos programas, desaproveitando assim os recursos dos sistemas operacionais multitarefa. Portanto, as formas de aproveitar a execução concorrente de vários programas devem ser estudadas a fundo.

Este trabalho visa a definição de um modelo de concorrência para uma linguagem de comandos léxica, tomando como ponto de partida as linguagens de comandos do sistema operacional Unix. As qualidades de tais linguagens de comandos são analisadas a seguir.

1.1 Linguagens de comandos do Unix

As linguagens de comandos do sistema operacional Unix têm-se mostrado muito úteis no desenvolvimento rápido de programas partindo de peças de utilização geral. A respeito deste fato, Peter Wegner diz [1989]:

> *"The UNIX environment breaks down traditional distinctions such as that between command languages and programming languages, between execution under terminal control and programmer control, between user models of computation. It enhances reusability of components by allowing a given component to be reused by application programmers and system programmers, from terminals or within command languages programs, for program development or for execution."*

O seguinte código é um exemplo clássico de como se pode resolver o problema da correção ortográfica de um texto em poucas linhas da linguagem *shell*[1], se se dispõe

de uma lista ordenada das palavras corretas num arquivo de nome `dicionário` [Bentley, 1986].

```
tr 'A-Z' 'a-z'  texto |\
   tr -c 'a-z' '\012' |\
      sort -u |\
         common -2 dicionário -
```

A figura 1-1 ilustra o fluxo de dados no programa. Observe-se que só foram utilizados programas de uso geral. O `tr` faz traduções de caracteres num texto, na primeira aparição maiúsculas por minúsculas e na segunda qualquer caractere que não letra pelo caractere de fim de linha. O `sort` ordena linhas de texto, a ordenação é controlada pelos parâmetros; no exemplo ela é feita com eliminação das linhas duplicadas. O `common` analisa o que têm em comum dois arquivos ordenados; a parametrização do exemplo pede as linhas que aparecem no segundo arquivo mas não no primeiro[1].

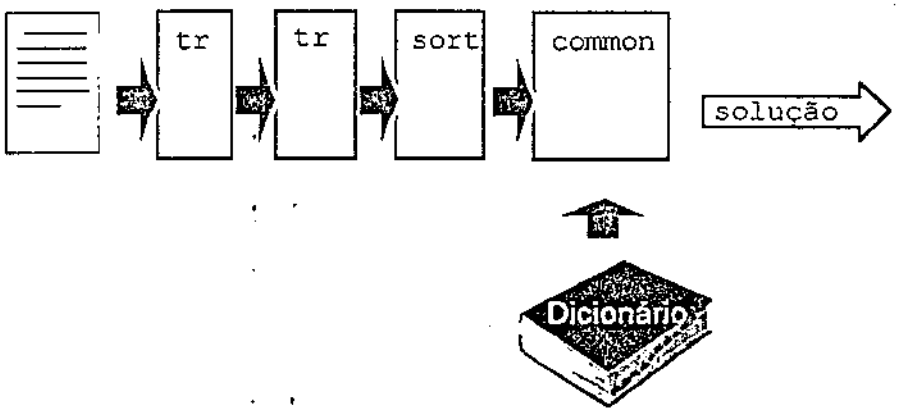
Em UNIX um *pipe* é um buffer de tamanho fixo que é colocado entre um produtor e um consumidor. Para colocar (retirar) dados no *pipe* utilizam-se as mesmas chamadas ao sistema que são utilizadas para gravar (ler) dados num arquivo. A única diferença semântica é que no *pipe* a "gravação" ("leitura") bloqueia se o buffer estiver cheio (vazio).

A filosofia do UNIX é que, sempre que for possível, os programas tenham uma entrada e uma saída que não esteja ligada a nenhum objeto. Essa entrada e saída genéricas são denominadas entrada padrão e saída padrão. A ligação só é feita no momento da sua execução. Um *pipe* pode ser colocado entre a saída padrão de um comando e a entrada padrãp de outro. O operador 'l' tem dois operandos: programas com seus respectivos parâmetros; ele especifica que os programas devem ser executados com um *pipe* conectando a saída do programa à sua esquerda com a entrada do programa a sua direita.

1. Hoje há muitas linguagens de comandos para o Unix, no entanto o modelo computacional delas continua ser o mesmo da original [Ritchie e Thompson, 1974]; é a este modelo que se faz referência sob o nome genérico de *shell*.

1. O carater '-' sinaliza que como segundo arquivo deve usar-se a entrada padrão.

FIGURA 1-1 Fluxo de dados no protótipo de corretor ortográfico



1.2 Prototipagem Rápida

A invocação de um comando custa muitas ordens de magnitude a mais do que uma chamada a um procedimento; a comunicação de dados por um *pipe* é muito mais cara do que a passagem de parâmetros. Em outras palavras, a *shell* é inadequada como linguagem de programação de aplicações nas quais a eficiência na execução seja crucial. Porém há uma diversidade de aplicações que não colocam exigências muito fortes quanto à eficiência; por exemplo, programas para serem executados poucas vezes (instaladores de um pacote de *software*) e programas de gerência do sistema.

Duas características fazem da shell uma ferramenta apropriada para a prototipagem rápida: ligação dinâmica e interatividade.

Mais importante ainda é a utilidade das linguagens de comandos para a *prototipagem rápida* [Zave, 1984; Luqi, 1989; Haeberer, Veloso e Baum, 1988]. Trata-se de escrever uma especificação operacional do sistema a desenvolver (protótipo) logo no início do processo de desenvolvimento. Em seguida, o protótipo é executado por um interpretador de alto nível. Isto fornece um retorno precoce sobre o sistema e serve como base para uma implementação mais eficiente mediante um processo iterativo de refinamento-teste.

Existem sistemas de prototipagem orientados a problemas específicos [Luqi, 1989]. Sistemas e linguagens de prototipagem de uso geral são um problema em aberto [Gabriel, 1989]. Um dos primeiros trabalhos a formalizar este ciclo de vida do *software* ressalta as virtudes da *shell* como ferramenta de prototipagem [Zave, 1984]. Duas características fazem da *shell* uma ferramenta apropriada para a prototipagem

rápida: ligação dinâmica e interatividade; procura-se neste trabalho fortalecer a capacidade de modelagem da *shell*, melhorando o modelo de concorrência.

1.2.1 Ligação dinâmica

Devido à forma interpretada da *shell* os valores da entrada padrão e saída padrão de cada um dos comandos envolvidos são resolvidos em tempo de execução.

1.2.2 Interatividade

A *shell* é uma linguagem interpretada, na qual os comandos são executados a medida que vão sendo lidos. Se os comandos forem oferecidos pelo terminal, tem-se um retorno imediato da sua execução, sem que seja necessário escrever um programa para isso. Isto ajuda na depuração e facilita o teste exaustivo dos programas, pois agiliza a iteração correção-teste. Além disto, muitas vezes o programador não sabe como vai fazer um programa, e só depois de experimentar interativamente com os comandos disponíveis ele descobre o programa.

Ainda, trechos de um programa escrito na linguagem de comandos podem ser testados rapidamente para a depuração ou para tirar dúvidas acerca da sintaxe ou semântica de uma determinada construção.

1.3 O Projeto A_Hand

O A_Hand é um Ambiente de desenvolvimento de *software* baseado em Hierarquias de Abstração em Níveis Diferenciados [Drummond e Liesenberg, 1987]. Um dos objetivos do A_Hand, particularmente significativo para este trabalho, é simplificar o desenvolvimento de grandes programas (centenas de milhares de linhas de código). O ciclo de vida destes programas caracteriza-se pela distribuição, tanto espacial quanto temporal, do trabalho entre várias pessoas.

O projeto comporta as seguintes linhas de trabalho:

- metodologias,
- hipertexto e técnicas de documentação,
- ferramentas de *groupware*,
- linguagens de programação,
- sistemas distribuídos.

Este trabalho concentra-se nos últimos dois itens. Na área de linguagens o maior esforço tem-se dedicado à definição, extensão e implementação da linguagem Cm [Silva, Liesenberg e Drummond, 1988; Furuti, 1991; Teles, 1993]. A Cm é um dialeto da linguagem "C" com recursos para programação modular e programação orientada a objetos; suas principais características incluem polimorfismo paramétri-

co, módulos, herança múltipla e tratamento de exceções. Um programa Cm é uma hierarquia de classes. Três mecanismos possibilitam a reutilização de código em Cm: a herança, a agregação e o polimorfismo. Paralelamente a este trabalho está em andamento a definição de uma extensão à Cm para torná-la uma linguagem de programação de objetos distribuídos [Gonçalves, 1994]. O *run time system* da Cm distribuída será implementado utilizando o sistema de suporte a aplicações distribuídas OMNI [Drummond e Di Cianni, 1992].

Programas Cm podem ser combinados entre si para formar programas mais complexos, a exemplo da *shell*. A LegoShell [Drummond, 1989;Piñón Arias, 1990] é uma linguagem gráfica para compor programas Cm. Um tipo embebido na linguagem Cm, denominado `port`, possibilita que os seus programas possam trocar dados com o mundo exterior. A partir da LegoShell é possível conhecer essas portas e interconectá-las. A figura 1-3 mostra uma computação LegoShell. Os programas Cm são representados com caixas. No perímetro dessas caixas podem se observar as representações gráficas das portas. Parâmetros podem ser especificados para cada programa apertando o botão que se encontra no vértice superior direito da caixa correspondente.

A contribuição deste trabalho é a definição de um modelo de concorrência baseado no modelo da shell que sirva para uma posterior definição de uma linguagem de comandos léxica.

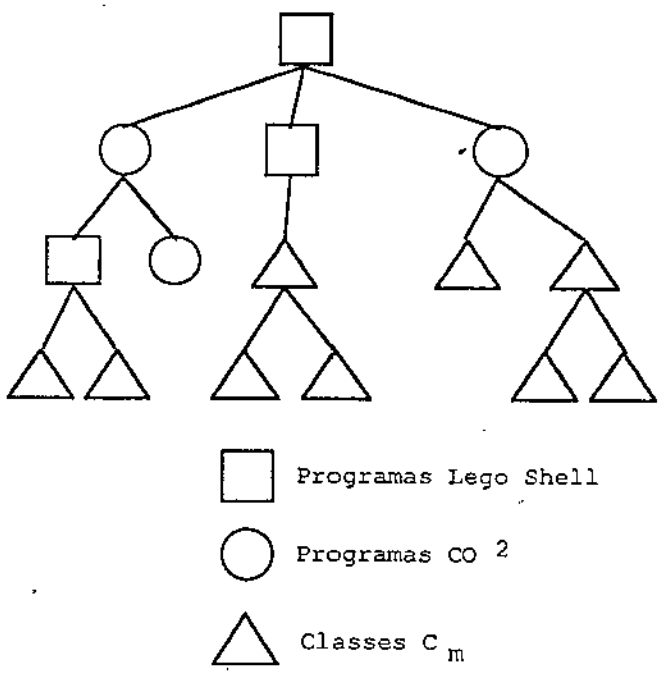
A LegoShell deve ser complementada com uma linguagem léxica que atenda os casos com os quais ela não pode lidar. A contribuição deste trabalho é a definição de um modelo de concorrência baseado no modelo da *shell* que sirva para uma posterior definição de uma linguagem de comandos léxica. Por razões históricas essa linguagem é denominada CO^2 (linguagem de Comandos Orientada a Objetos). Para facilitar a redação decidiu-se chamar "a CO^2" ao modelo apresentado neste trabalho; contudo, vale esclarecer que o objetivo não foi a definição de uma linguagem de programação. Uma linguagem de comandos deverá complementar este modelo com . outros tópicos (por exemplo, combinar expressões regulares com nomes de arquivos, programas, etc.).

As seguintes regras mostram como deve ser o relacionamento entre os programas escritos em Cm, LegoShell e CO^2:

programa-A_Hand → programa Cm
| computação LegoShell
| programa CO^2
programa-CO^2 → composição de programas-A_Hand
(segundo a sintaxe da CO^2)
computação-LegoShell → composição de programas-A_Hand
(segundo a sintaxe da LegoShell)
programa-Cm → classe-Cm
classe-Cm → composição de classes-Cm
(segundo a sintaxe de Cm)

A figura 1-2 mostra uma hierarquia de abstrações que cumpre com estas regras. Observe-se que eventualmente programas escritos em CO^2 podem ser uma folha da árvore; isto quer dizer que além de ser útil para compor programas, a CO^2 deve também ser uma linguagem de programação.

FIGURA 1-2 Hierarquia de objetos A_Hand



1.4 Um Modelo de Concorrência para CO²

O modelo de concorrência da *shell* está formado por um universo de agentes que possuem *uma* entrada e *uma* saída de dados. Esses agentes podem ser dispostos de forma que a saída de um deles alimente a entrada de um outro e assim por diante. Colocam-se a seguir dois casos de estudo que mostram as limitações do modelo da *shell* que se desejam superar.

1.4.1 Fluxo em várias dimensões

O encadeamento de comandos em *shell* com o operador 'l' permite a expressão daqueles casos nos quais a informação flui em uma única direção e em um único sentido. Os casos em que os dados fluem em mais do que uma direção ou sentido

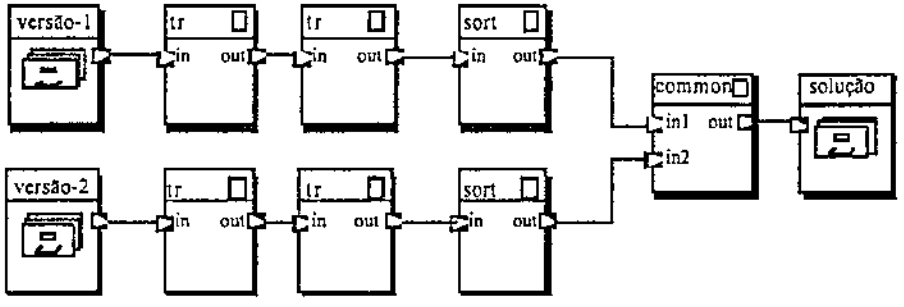
não podem ser expressados com tanta naturalidade. Imagine-se ter duas versões de um documento e deseja-se saber as palavras introduzidas na segunda versão que não estavam na primeira. Note-se que pode-se reutilizar a solução ao problema da correção ortográfica. Afinal este problema nada mais é do que uma correção ortográfica da segunda versão na qual o dicionário é uma lista ordenada de todas as palavras da primeira versão. O seguinte código de *shell* expressa este enfoque.

```
tr 'A-Z' 'a-z'  versão-1 |\
   tr -d 'a-z' '\012' |\
     sort -u > palavras-1
tr 'A-Z' 'a-z'  versão-2|\
tr -d 'a-z' '\012' |\
     sort -u |\
        common -2 palavras-1 -
```

Conceitualmente as listas das palavras de ambas as versões poderiam ser computadas em paralelo e alimentar o comando common simultaneamente. A falta de flexibilidade do modelo para expressar esta situação força a expressão seqüencial de um problema inerentemente paralelo.

A LegoShell permite expressar tal paralelismo. A figura 1-3 mostra a computação LegoShell correspondente.

FIGURA 1-3 Exemplo de computação LegoShell



1.4.2 Cliente - servidor

A segunda limitação levantada é a falta de suporte sintático para ligar dinamicamente programas clientes com programas servidores. A *shell* pode ativar um servidor de um banco de dados, por exemplo, e depois ativar clientes desse servidor. No entanto, não há recursos da *shell* envolvidos na ligação dos clientes com o servidor. A ligação entre clientes e o servidor se faz de forma independente da *shell*, por vezes ela já está fixa no código, ou então se utilizam mecanismos globais de resolução de nomes (NIS, DNS [SUN, 1990], X.500). Esta última não é ligação dinâmica, há ligação estática a um nome simbólico. É este nome simbólico quem estabelece a ligação entre

clientes e servidor; embora o endereço físico desse nome seja resolvido (dinamicamente) durante a execução.

Suponha-se o seguinte sistema de teleconferência simplificado. O sistema está formado por um agente que cuida da interface com o usuário e outro que controla a concorrência entre todos os usuários de uma mesma sessão. Para iniciar uma sessão de teleconferência instancia-se um gerenciador de concorrência e, depois, para cada participante da conferência, executa-se um agente ligado ao gerenciador de concorrência. Claramente, em um mesmo instante podem existir várias sessões ativas, independentes entre si, cada uma com seu próprio gerenciador. Para isto é desejável que a ligação entre agentes e o gerenciador correspondente seja feita no momento de executar os programas cliente e servidor.

• • •

O modelo de concorrência da CO^2 foi desenvolvido com a capacidade de lidar com situações como as esboçadas nesta seção. Além disto teve-se como objetivo preservar as características da *shell* salientadas na seção 1.2: ligação dinâmica e interatividade. O modelo permite ainda permitir o desenvolvimento aninhado em vários níveis de abstração, respeitando as regras de composição de programas do A_Hand apresentadas na seção 1.3.

1.5 Organização do trabalho

No capítulo 2 se analisam com mais rigor os casos discutidos na seção 1.4.; individualizam-se as diferenças entre programas com filtros e programas com clientes e servidores. Ainda no mesmo capítulo se inclui um estudo dos modelos de concorrência que mais influíram na definição do modelo para a CO^2 detalhando as contribuições e limitações de cada modelo com relação aos objetivos deste trabalho.

A contribuição deste trabalho encontra-se nos capítulos 3, 4, 5 e 6, neles descreve-se detalhadamente o modelo de concorrência para a CO^2. Depois de apresentar um resumo das características do OMNI o capítulo 3 introduz o modelo primitivo de CO^2; a persistência em CO^2 também é tratada neste capítulo. A programação com este modelo é operacional; no capítulo 4 se discutem as desvantagens disto, para tanto se torna a analisar a interação entre processos e se mostra como melhorar a expressividade da CO^2 mediante a incorporação de recursos de programação *definicional* e funcional. O aninhamento de programas CO^2 é apresentado no capítulo 5. Até aqui o modelo é de processos seqüenciais que se comunicam entre si; no capítulo 6 se estuda a necessidade de introduzir mecanismos de concorrência interna, alguns exemplos são mostrados junto com a sua aplicabilidade para lidar com os casos de concorrência no acesso a arquivos.

A implementação de um protótipo do modelo é apresentada no capítulo 7 comentando as ferramentas e a metodologia utilizadas; inclui-se o código de C++ das partes mais críticas do protótipo e uma discussão sobre os objetivos dessa implementação. Encerrando o trabalho, no capítulo 8 discutem-se os resultados obtidos.

Capítulo 2

Modelos de Concorrência

Actually, concurrency is not the real issue —after all concurrency has been exploited for a long time in the software revolution caused by time-sharing. The key difference between the now classic problem of operating systems, and our desire to exploit concurrency in the context of cooperative computing is that in the former there is little integrartion between "jobs" or "processes" that are executed concurrently... Our problem is quite the reverse: we wish to have a number of processes work together in a meaningful manner.

Gul Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*

Para obter algum proveito da possibilidade de executar concorrentemente muitos comandos é preciso compreender os problemas que dita concorrência introduz, bem como as formas úteis de interação entre atividades concorrentes.

Depois de discutir os pontos de interesse deste trabalho em conceitos básicos da programação concorrente, apresentam-se os diferentes tipos de processos que podem ser envolvidos em um programa distribuído. Ainda neste capítulo apresentam-se os modelos de concorrência que influíram na definição da CO^2.

2.1 Programação Concorrente

A programação concorrente tem atingido diversas áreas da ciência da computação. A área de sistemas viu-se envolvida, desde o início, com os problemas de comunicação, sincronização, exclusão mútua, tolerância a falhas, etc. As formas de expressar o paralelismo, a comunicação e a sincronização, atingiram a área de linguagens. Para quem trabalha em algoritmos, o paralelismo é objeto de estudo como uma forma de construir algoritmos mais eficientes. A área teórica também viu-se envolvida na verificação de programas concorrentes.

A disponibilidade comercial de arquiteturas que dão suporte a algum tipo de paralelismo, e o custo decrescente delas, tem levado os problemas da concorrência à engenharia de *software*.

2.1.1 Concorrência no *software*

Nas primeiras gerações de sistemas o processamento era feito na modalidade *batch*. Era necessário um grande esforço na fase de análise para modelar um sistema cujo domínio não tivesse uma relação óbvia com esta modalidade de processamento.

O advento da computação interativa permitiu ampliar o domínio de aplicação dos sistemas de *software*. Nestes sistemas já aparecem problemas de concorrência, especialmente no acesso a bancos de dados compartilhados; no entanto o controle de concorrência é ocultado por camadas de *software* básico. O programador de aplicativos não precisa estar ciente dessa concorrência: o sistema operacional de tempo compartilhado implementa a abstração de um processador virtual para cada processo ao passo que assuntos de concorrência de maior nível, invisíveis para o sistema operacional, são tratados, por exemplo, por um sistema gerenciador de banco de dados.

Plataformas formadas por múltiplos computadores interconectados por uma rede de comunicação de dados de alta velocidade são hoje acessíveis para um amplo leque de usuários, em decorrência dos avanços nas tecnologias de transmissão de dados e a grande diminuição nos custos do *hardware*. Em seguida surge a demanda de sistemas cuja execução é distribuída pelas unidades de processamento que compõem estas plataformas. Bal [1990] cita quatro razões para distribuir uma aplicação:

- melhora do desempenho do aplicativo executando simultaneamente diferentes partes de um mesmo programa,
- aumento da confiabilidade do programa replicando funções e dados em vários processadores,
- estruturação do programa como uma coleção de serviços especializados, atribuindo um ou mais processadores a cada serviço,
- atendimento à distribuição geográfica inerente ao aplicativo.

Uma solução freqüentemente adotada para assistir ao programador de aplicações é acrescentar alguma linguagem de programação seqüencial com uma biblioteca para criação e comunicação entre processos. Ao respeito Bal diz:

> *"Such a library is easy to implement. This method is frequently used by commercial systems [...], however, this approach also has severe disadvantages and still leaves most of the real problems to the applications programmer."*

Em outras palavras, esta solução não possui meios específicos adequados para resolver os problemas de concorrência que são colocados pela distribuição da aplicação. Os recursos destas bibliotecas se apresentam ao programador na forma de elementos

próprios da programação seqüencial, tais como procedimentos, funções, primitivas de entrada e saída ou tipos abstratos de dados. Estes elementos não conseguem captar as características da programação concorrente em toda sua amplitude. Tome-se como exemplo o caso da programação orientada a objetos. O primeiro passo, obviamente, é associar processos a objetos, afinal ambos se comunicam trocando mensagens. Em seguida começam as divergências. É claro que um processo não sempre precisa ficar bloqueado à espera de uma resposta a uma mensagem que ele enviou. Pode ser, também, o caso em que um processo receba uma requisição que não pode responder temporariamente (uma operação de extração em um *buffer* vazio), mas poderia adiar a resposta desde que consiga processar novas requisições enquanto isso.

Se faz preciso, então, um modelo de computação que contemple a concorrência desde o início.

2.1.2 Distribuição ou concorrência?

Nos últimos parágrafos misturaram-se inflexões da palavra *distribuição*, com o vocábulo *concorrência*. Não há consenso na literatura sobre o significado de cada termo. A expressão *programa distribuído*, ou *sistema distribuído*, tende a ser utilizada para salientar o fato de que partes do programa são executadas em distintos pontos que não compartilham memória, mas que conseguem comunicar-se mediante a troca de mensagens. A palavra *concorrência* tradicionalmente está mais associada às técnicas para expressar o paralelismo *potencial* de um programa, sem prestar especial atenção à realização desse paralelismo. Considerou-se mais adequado para o título do trabalho o termo *concorrência*, uma vez que o objetivo deste é definir a expressão da execução paralela de vários comandos, de forma que cooperem entre si para produzir um trabalho útil.

2.2 Interação entre processos concorrentes

Para poder integrar a concorrência ao processo de desenvolvimento de software deve-se compreender as formas de parcelar um sistema numa quantidade de módulos concorrentes. Andrews [1991] distingue três tipos de interação entre processos concorrentes:

- .filtros,
- clientes e servidores,
- parceiros.

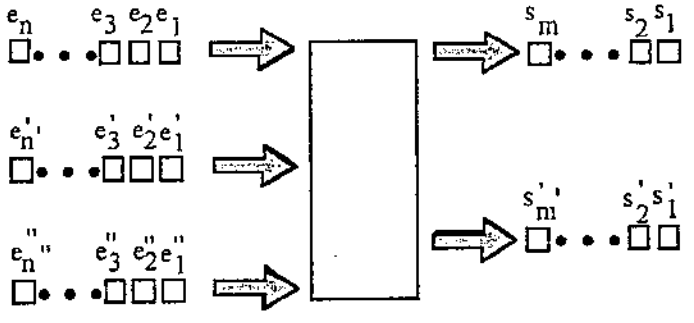
2.2.1 Filtros

Filtros são processos que agem como produtores e consumidores. Um filtro consume dados, provavelmente provenientes de outros filtros, e produz novos dados, os quais podem vir a ser consumidos por outros filtros.

A figura 2-1 representa o funcionamento de um filtro com três fluxos de dados de entrada e dois de saída, filtros não têm estado. Os fluxos de saída são exclusivamente funções dos fluxos de entrada. Note-se que não se está colocando nenhum relacionamento entre cada unidade de dados de entrada e de saída. Este relacionamento eventualmente pode existir, como no caso de um filtro que troca maiúsculas por minúsculas; no entanto, o caso mais geral de relacionamento é entre os fluxos completos dos dados, a exemplo de um filtro que ordena linhas de texto ou que conta a quantidade de caracteres. Portanto, é importante que os filtros disponham de mecanismos para sinalizar o começo e o fim de um fluxo de dados.

Para conseguir um máximo de reutilização é desejável que os filtros sejam "caixas pretas", com entradas e saídas não-resolvidas, de sorte que sua ligação possa ser adiada até o momento em que o filtro é executado.

FIGURA 2-1 Esquema de filtro



$<si>i=1,m=f(<ei>i=1,n\ ,\ <e'i>i=1,n'\ ,\ <e''i>i=1,n'')$

$<s'i>i=1,m'=g(<ei>i=1,n\ ,\ <e'i>i=1,n'\ ,\ <e''i>i=1,n'')$

2.2.2 Clientes e servidores

Da mesma forma que os filtros, os servidores produzem dados em função dos dados que recebem de entrada. No entanto, neste modelo cada unidade de dados de saída esta relacionada com alguma unidade de dados de entrada. As unidades de dados de entrada chamam-se de *requisições* e as de saída *respostas*. A rigor, nem sempre é preciso que exista uma resposta. Toda requisição provoca uma quantidade determinada de ações, e a ausência de resposta pode ser considerada como o retorno de um valor que é ignorado.

Um processo que envia uma requisição a um servidor é chamado de cliente. Diferentemente dos filtros, os servidores precisam que os dados produzidos por eles sejam encaminhados ao destino certo. Geralmente este destino é o cliente que fez a requisição; no entanto, pode-se imaginar uma situação na qual o cliente peça ao servidor para enviar a resposta para um terceiro processo. Desta forma pode-se implementar um *pipelining* de servidores; um servidor atende as requisições, despachando-as para terceiros, que fazem o verdadeiro processamento delas e retornam as respostas ao cliente original.

No modelo de clientes e servidores não faz sentido a sinalização de fim de fluxo de dados. Geralmente um servidor é um processo que após uma fase de inicialização permanece num ciclo infinito processando requisições.

As respostas não necessariamente devem sair do servidor na ordem que chegaram as respectivas requisições. Suponha-se dispor de um processo que implementa um *buffer*. A ele podem chegar requisições do tipo *('a', x)*, solicitando-lhe acrescentar o dado *x* ao *buffer*, e requisições `'r'` para retirar um dado do *buffer*. O processo responde com respostas `'ok'` às primeiras requisições e com o dado extraído do *buffer* às segundas. Dada a seguinte seqüência de requisições:

e_1 = *('a', 1)*
e_2 = *('a', 2)*
e_3 = `'r'`
e_4 = `'r'`

tem-se como resultado a seguinte seqüência de saída:

s_1 = `'ok'`, respondendo a e_1
s_2 = `'ok'`, respondendo a e_2
s_3 = 1, respondendo a e_3
s_4 = 2, respondendo a e_4

No entanto, a seguinte seqüência de requisições:

e_1 = (`'a'`, 1)
e_2 = `'r'`
e_3 = `'r'`
e_4 = (`'a'`, 2)

produz como saída

s_1 = `'ok'`, respondendo a e_1
s_2 = 1, respondendo a e_2
s_3 = `'ok'`, respondendo a e_4
s_4 = 2, respondendo a e_3

Esta perda de relacionamento temporal entre requisições e respostas é uma das questões que não se consegue resolver elegantemente com recursos de programação seqüencial. A questão é tratada detalhadamente no capítulo 6.

2.2.3 Parceiros

Os filtros realizam sua função com total independência entre si. Os servidores reagem aos pedidos dos clientes, existe entre eles uma relação mestre-escravo. Em um programa com processos parceiros todos desempenham a mesma função, cooperando democraticamente entre eles, em prol de um objetivo comum. O algoritmo distribuído de exclusão mútua de Lamport [1978] é um exemplo de programação com parceiros.

Além de ser a unidade de paralelismo, filtros, clientes e servidores são candidatos a módulos de *software* reutilizáveis em diversos contextos. Os parceiros foram excluídos porque, pela sua própria natureza, não seriam reutilizáveis em diferentes contextos. Na seguinte seção, entre outras coisas, se analisa a capacidade de alguns modelos de concorrência para expressar programas com filtros, e com clientes e servidores, bem como a reutilização destes módulos.

2.3 Modelos e Linguagens

Apresentam-se nesta seção os modelos de concorrência que mais influíram na definição da CO^2. Prestou-se especial atenção aos seguintes tópicos:

- capacidade para modelar os esquemas de filtros e cliente-servidor,
- reutilização das unidades de paralelismo,
- adequação aos requisitos de aninhamento, interatividade e ligação dinâmica colocados na seção 1.4.

2.3.1 CSP

CSP [Hoare, 1978] foi importante para este trabalho por tratar-se da primeira notação a enxergar um programa concorrente como sendo formado por uma quantidade de processos seqüenciais, comunicando-se por meio de primitivas de entrada e saída. Como o próprio Hoare diz, não deve ser enxergada como uma linguagem de programação. Contudo, constitui um bom modelo para analisar muitos problemas que coloca a programação distribuída. Por causa da ligação precoce dos processos envolvidos nas primitivas de entrada e saída, é impossível executar uma parte do programa sem dispor do resto; uma implementação de uma linguagem interpretada baseada em CSP é inviável.

Processos num programa CSP só podem interagir com processos do mesmo programa, não há forma de eles interagirem com processos genéricos, a serem instanciados

a posteriori. Isto torna-os inutilizáveis fora do contexto para o qual foram escritos. A arquitetura dos programas CSP só tem dois níveis: não há construções que possam aninhar-se permitindo trabalhar em diferentes níveis de abstração.

O modelo mostra-se adequado para programas que possam decompor-se em filtros. Já a divisão em clientes e servidores apresenta alguns problemas, originados na impossibilidade de uma ligação dinâmica entre o servidor e os clientes. Uma solução seria prever todos os potenciais clientes no código do servidor atendendo ao seguinte esquema:

```
*[ Ai?requisao -> processar resposta; Ai!resposta;]
```

Neste esquema toda requisição deve ser respondida antes de aceitar uma nova requisição. Isto pode sugerir que respostas não podem ser adiadas, uma vez que é impossível lembrar o destinatário de uma resposta (rótulos de processos não podem ser armazenados). No entanto, as construções não determinísticas podem adiar o processamento de alguma requisição desde que a decisão não dependa do conteúdo dos dados que vêm na requisição. Uma vez que as identidades dos processos não podem ser comunicadas, a delegação da atenção das requisições não pode ser implementada.

Um processo é uma unidade que encapsula código e dados. É natural desejar a reutilização de um processo em diferentes contextos, para os quais seu comportamento se mostre adequado. A ligação precoce de um nome com o processo com o qual se deseja interagir limita o grau de reutilização dos mesmos. A tendência tem sido os processos nomearem, nas primitivas de entrada e saída, objetos especiais denominados portas. Assim, quando um programador escreve o código de um processo consegue abstrair-se do verdadeiro destino ou origem dos dados. A ligação se produz numa etapa posterior do ciclo de vida do programa. A seguir, discute-se um ambiente de desenvolvimento de programas distribuídos cujo modelo é de processos seqüenciais que se comunicam por portas.

2.3.2 O ambiente de programação REX.

O REX [Kramer et al, 1989] é um ambiente de desenvolvimento de programas distribuídos. Processos seqüenciais no REX são escritos em extensões a linguagens bem conhecidas (C, C++, Pascal e Modula-2), chamadas *linguagens de programação*. Uma das tais extensões é a inclusão de portas. Portas podem ser de entrada ou de saída.

O código de um processo é como uma classe, chamada de *componente*, a partir da qual se podem criar uma ou mais instâncias dele. Cabe à *linguagem de configuração* a instanciação de processos e a interligação das portas dessas instâncias.

Para validar a correção na montagem de uma configuração as componentes são acompanhadas de uma especificação de suas interfaces. A especificação da interface

de uma componente abrange o nome e o tipo das portas de entrada, o nome e o tipo das portas de saída e o tipo dos parâmetros formais. O tipo de uma porta é o tipo das mensagens que entram ou saem por ela.

A linguagem de configuração é uma linguagem *definicional* que descreve a estrutura do programa, isto é, quais as instâncias envolvidas e como elas estão conectadas. Darwin, o nome desta linguagem, introduz a concorrência no modelo. Programas Darwin podem ser aninhados, uma vez que eles também podem ser componentes, ou seja, podem ter portas de entrada e saída e parâmetros formais. Há uma disjunção entre os programas que se pode escrever em Darwin e aqueles que se pode codificar utilizando as linguagens do REX. O aninhamento só tem lugar a nível de Darwin. Configurações Darwin não podem ler, processar e escrever dados; em algum nível do aninhamento a componente deve ser escrita em alguma das linguagens de programação do REX.

Uma configuração Darwin só pode ser executada depois de ser compilada por completo; portanto o modelo não é adequado para uma abordagem interpretada.

Filtros reutilizáveis podem ser programados nas linguagens de programação do REX, uma vez que eles não devem importar-se com a origem nem o destino dos dados, apenas ler os dados das portas de entrada e escrever os resultados nas portas de saída.

Portas de entrada são valores de primeira ordem, ou seja, podem ser comunicados de um processo a outro. Para implementar clientes e servidores, o cliente manda junto com a requisição o valor da porta de entrada para onde deve ser encaminhada a resposta. As implementações das linguagens do Darwin fornecem "açúcar sintático" para esconder isto do programador.

Quanto a Darwin não se pode dizer se ele é apto para implementar filtros, clientes e servidores. Ele só junta componentes que podem ser filtros, clientes e servidores, e põe eles em cena.

2.3.3 Actors

A contribuição de *actors* [Agha, 1986] a este trabalho foi na compreensão da criação dinâmica de unidades de paralelismo. Neste modelo o universo está formado por agentes computacionais chamados atores. A única forma pela qual um ator pode influir em outro é enviando para este uma mensagem. Ao chegar a seu destino, a mensagem é colocada numa fila de mensagens que todo ator tem; um comportamento lhe especifica o que deve fazer com o conteúdo da primeira mensagem da fila. O funcionamento de um ator é retirar a primeira mensagem da fila, fazer o que o comportamento lhe diz que é para ser feito com essa mensagem, retirar a próxima mensagem da fila e assim por diante.

As ações que um comportamento pode especificar são: criar novos atores, enviar mensagens para outros atores e fixar um novo comportamento que guie o processamento da próxima mensagem na fila. A seguir analisam-se estas ações por separado.

Para enviar uma mensagem a um ator é preciso conhecer o endereço dele. Cada ator tem uma lista de conhecidos[1], aos quais pode enviar mensagens. A lista de conhecidos é inicializada no momento de criação de um objeto e re-inicializada toda vez que se especifica o novo comportamento do ator. Um ator também conhece o endereço dos atores que ele cria; além disso, os endereços de atores podem ser comunicados a outros atores por meio de mensagens.

A criação de um novo ator envolve a especificação do comportamento que deve processar a primeira mensagem e a inicialização da lista de conhecidos.

A substituição do comportamento é um mecanismo projetado para eliminar os efeitos colaterais, apontados com uma das principais desvantagens da programação imperativa. O modelo puro não possui um mecanismo de atribuição com o qual se possa modificar o estado. A mudança de estado é feita de uma vez ao especificar-se o comportamento para processar a próxima mensagem e a nova lista de conhecidos. O estado não pode ser modificado aos poucos pelo efeito colateral dos comandos.

Um programa de *actors* é uma seqüência de definições de comportamentos, seguidas por um ou mais comandos de criação de atores e envio de mensagens. O esquema lembra um programa funcional formado por definições de funções seguidas pela invocação a uma delas. O modelo é adequado para ser implementado por um interpretador. O código dos comportamentos pode reutilizar-se em vários programas. O aninhamento de atores não é possível. A expressão de programas como composição de filtros é possível, bem como a interação cliente-servidor, embora esta seja trabalhosa por causa do baixo nível do modelo.

2.4 Discussão

Na primeira parte do capítulo destacou-se a importância de um modelo de computação concorrente no processo de produção de software. A seguir analisaram-se as formas de decompor um programa em vários processos interatuantes. Reconheceram-se três tipos de interação entre processos: filtros, cliente-servidor e parceiros. Filtros e servidores são adequados para ser reutilizados em diferentes contextos; os primeiros por serem transformadores de dados, os segundos por ser apropriados para implementar tipos de dados abstratos distribuídos. Os processos parceiros têm um fim muito específico, que faz com que não sejam reutilizáveis por outros programas.

1. *Do inglês *acquaintance list*.

Discussão

A segunda metade do capítulo apresentou três enfoques que influíram na definição do modelo de concorrência para a CO^2. O CSP foi um bom marco teórico para estudar sistemas formados por processos que se comunicam por troca de mensagens. O ambiente de programação REX é uma boa aplicação para desenvolvimento de *software* dos conceitos do modelo de CSP enriquecido com portas. O modelo de *actors* forneceu suporte para a compreensão da criação dinâmica de processos. A seguinte tabela sumariza as diferenças entre as três abordagens.

TABELA 14-1 Resumo das características de CSP, REX e Actors

	CSP	**REX**	**Actors**
Programação com filtros	Possível	Possível	Possível
Programação com clientes e servidores	Limitada	Possível	Possível
Reutilização das unidades de paralelismo	Impossível	Possível	Possível
Desenvolvimento aninhado	Impossível	Só ao nível de Darwin	Impossível
Implementação interativa	Impossível	Impossível	Possível
Ligação dinâmica das unidades de paralelismo	Impossível	Possível	Possível

Capítulo 3 Concorrência em CO^2

O modelo do REX tem muitos pontos em comum com o A_Hand. Há uma correspondência entre as linguagens de programação do REX e a Cm, bem como a Darwin tem alguma semelhança com a LegoShell e a CO^2. No A_Hand a comunicação entre processos também é feita num esquema de portas. No entanto, programas Cm têm acesso às facilidades de criação dinâmica de processos, criação dinâmica e conexão de portas, ao passo que programas CO^2 devem conseguir tanto ler e escrever portas quanto realizar transformações nos dados. Acrescenta-se que a CO^2 deve ser uma linguagem interpretada enquanto Darwin é compilada.

Apresenta-se neste capítulo uma breve descrição do sistema de suporte ao desenvolvimento de aplicações distribuídas do A_Hand e depois o modelo primitivo de concorrência CO^2.

3.1 O Sistema OMNI

OMNI [Drummond e Di Cianni, 1992; Di Cianni, 1994] é um dos subprojetos do A_Hand, e trata-se de um sistema de suporte ao desenvolvimento de aplicações distribuídas. Três módulos compõem o OMNI:

- o gerenciador de processos,
- o módulo de portas e
- o servidor de nomes.

3.1.1 O gerenciador de processos

Mediante o gerenciador de processos o programador pode iniciar a execução de programas em qualquer máquina de um domínio sob sua supervisão. A premissa básica é oferecer as facilidades de criação de processos e troca de sinais do Unix num ambiente distribuído. Processos recebem uma identificação única no tempo e no espaço. Esta identificação pode ser utilizada para enviar sinais. A exemplo do Unix, existe entre processos um relacionamento pai-filho. Um processo pai é sinalizado quando da morte de um filho.

3.1.2 O servidor de nomes

Processos podem registrar uma ligação entre um nome e um valor junto ao servidor de nomes, de maneira que outros processos consigam conhecer essa ligação. Exemplos de valores que podem ligar-se ao servidor são identificadores de processos e identificadores de portas (estes são definidos na próxima subseção); no entanto a flexibilidade do modelo permite a ligação de qualquer tipo de dados. Entre os destaques do servidor de nomes do OMNI estão sua estrutura distribuída, a qual torna-o tolerante a falhas, e seu esquema de permissões.

3.1.3 O módulo de portas

O módulo gerenciador de portas permite a criação, destruição, conexão e desconexão de portas de entrada e de saída, bem como a troca de mensagens entre si. Ao criar-se uma porta o sistema retorna um objeto que identifica univocamente a porta em todo o domínio. Este objeto é chamado de *identificador de porta*. Há dois tipos de portas de entrada no OMNI: *conectáveis* e *não conectáveis*. Dependendo do tipo de portas a comunicação entre processos pode ser *com conexão* ou *sem conexão*.

3.1.3.1 Comunicação com conexão:

Para um processo enviar informação a outro, é preciso que exista uma conexão entre uma porta de um a uma porta do outro. Assim, todo dado escrito na porta de saída do primeiro pode ser lido pelo segundo na sua porta de entrada. Contudo, não é necessário que ambas as portas estejam conectadas antes de qualquer operação de entrada ou saída nelas. A conexão de portas pode ser feita por qualquer processo que conheça os dois identificadores das portas, portanto a exigência de alguma ordenação temporal entre as operações de conexão, entrada e saída seria uma restrição impensável. Uma escrita numa porta de saída não conectada bloqueia o processo dono da porta até ela ser conectada a uma outra porta de entrada; analogamente, um processo é bloqueado se ler de uma porta ainda não conectada. Mesmo que a porta de entrada esteja conectada, a leitura bloqueia se não houver nada a ser lido[1]

1. O controle de fluxo pode bloquear a saída também.

Há três formas pelas quais um processo, que não o dono da porta, consegue conhecer o identificador da porta. A primeira é mediante uma consulta ao servidor de nomes. Outra forma é que lhe seja comunicada por outro processo. Por último, o processo pai sempre conhece os identificadores de portas atribuídas às portas declaradas na interface dos seus filhos. A exemplo do REX, a interface descreve as portas de um programa. Por exemplo, pode-se imaginar uma implementação em OMNI do comando `common` do exemplo de corretor ortográfico colocado no capítulo 1. O comando teria declaradas na sua interface as portas de entrada `in1` e `in2` e a porta de saída `out`. O esquema encoraja um estilo de programação no qual há dois tipos de programas, aqueles que processam dados lidos nas suas portas de entrada e comunicam os resultados escrevendo nas suas portas de saída, e aqueles que despacham a execução dos anteriores e pedem a conexão das suas portas.

3.1.3.2 Comunicação sem conexão:

A comunicação com conexão caracteriza-se pela existência de três fases: abertura, troca da informação e encerramento. Tanto no envio quanto na recepção de dados não é preciso nomear a origem ou o destino desses dados, pois eles só fluem pela conexão. Já na comunicação sem conexão não há uma abertura que anteceda a comunicação, nem há encerramento da conexão. Toda vez que um dado é enviado, precisa-se identificar a porta de entrada que deve recebê-los. Essa porta de entrada deve ser não conectável.

Pode-se imaginar, contudo, uma pseudo-conexão. Construída sobre OMNI, uma pseudo-conexão envolve uma porta de saída e outra de entrada sem conexão. Este recurso resolve a ligação de um cliente com o seu servidor no momento de ser executado. O processo dono da porta de saída é sinalizado para ele lembrar que toda escrita nessa porta deve ser traduzida como o envio de uma mensagem à porta de entrada. A CO^2 fornece uma sintaxe equivalente para tratar conexões e pseudo-conexões.

3.1.3.3 Conectores especiais

Uma comunicação com conexão envolve uma (e só uma) porta de saída e uma (e só uma) porta de entrada. Outros tipos de conexão podem ser imaginados:

1. várias portas de saída conectadas a uma porta de entrada,
2. uma porta de saída conectada a várias de entrada,
3. várias portas de entrada conectadas a várias de saída.

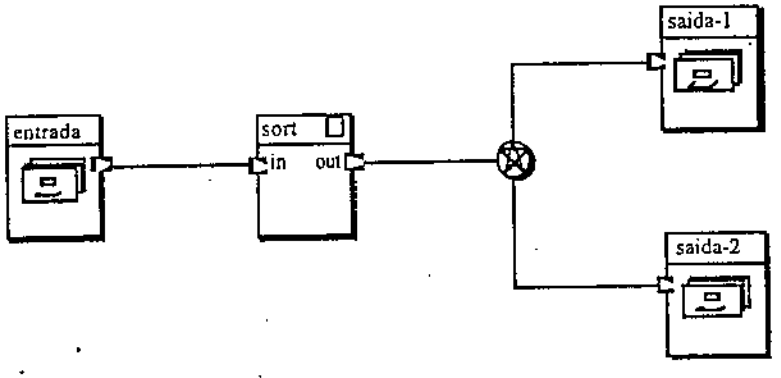
Poder-se-ia argumentar que o primeiro caso está, em certa forma, englobado pelas portas não conectáveis. No entanto, a falta do encerramento da conexão faz com que o receptor não possa saber se já recebeu todos os dados. O esquema não é adequado se o processo consumidor deve ordenar as mensagens que recebe por essa porta, por exemplo. É necessário algum mecanismo de propagação dos sinais de fim do fluxo dos dados dos emissores.

O segundo caso merece ser examinado com mais atenção. Drummond e Liesenberg [1987] distinguem duas semânticas de conexão para este caso, o *broadcast* e o *mail-box*. O OMNI implementa estes dois tipos de conexão. Ambas as semânticas são estendidas para o terceiro caso.

A semântica de *broadcast* recebe seu nome pelo fato de comportar-se como uma emissora de rádio, todos os ouvintes recebem a mesma informação. Um objeto, chamado de *broadcast*, recebe todos os dados. Estes dados são misturados não deterministicamente, só se garantindo a preservação da ordem dos dados provenientes da mesma fonte. Cópias idênticas do fluxo de dados resultante de tal mistura são enviadas a todas as portas de entrada que estiverem ligadas ao conector

Um *mailbox* é como uma porta de entrada compartilhada por muitos processos[1]. Por vezes o objetivo é que cada unidade de dados seja consumida só por um dentre vários processos. O conector M envia cada unidade de dados que recebe a só um processo. Duas unidades de dados vindas da mesma origem que porventura chegarem ao mesmo destino preservam a ordenação temporal.

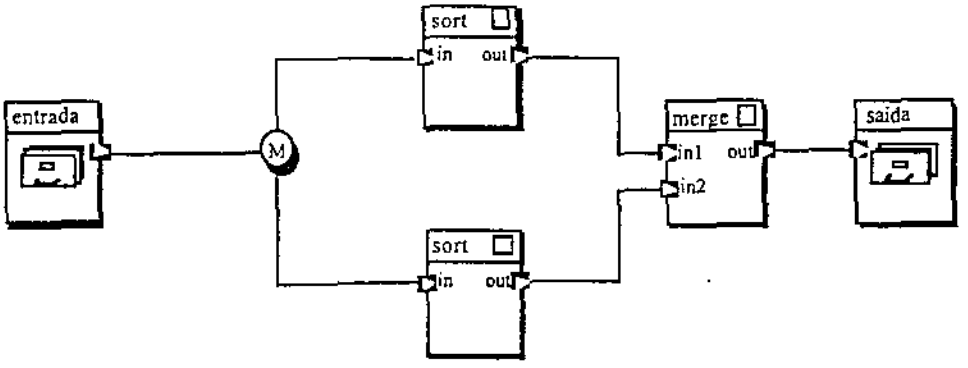
FIGURA 3-1 Computação LegoShell envolvendo conector estrela



A LegoShell fornece suporte sintático para a expressão de ambos os conectores. A figura 3-1 mostra uma computação envolvendo um conector *broadcast*, onde os arquivos `saida-1` e `saida-2` recebem uma réplica ordenada do arquivo `entrada`. Na figura 3-2 tem-se um exemplo de computação com o conector M: o trabalho de ordenação do arquivo `entrada` é distribuído entre duas instâncias do comando `sort`, a saída destas é unida, mantendo a ordem, pelo comando `merge`.

1. Vide [Andrews e Schneider, 1983]

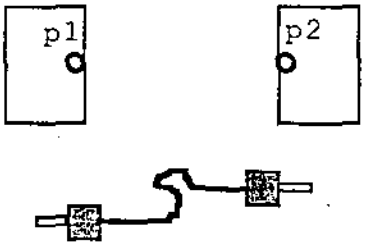
FIGURA 3-2 Computação LegoShell envolvendo conector M



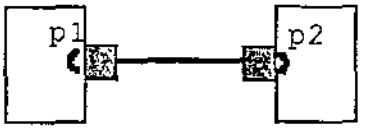
3.2 Execução de Comandos em CO^2

Para o OMNI portas de entrada e portas de saída são entidades equivalentes, quando da conexão. A figura 3-3 mostra a ilusão que OMNI oferece. Portas de entrada e saída são soquetes, e conectar duas portas é estender um cabo entre os soquetes.

FIGURA 3-3 Esquema de conexão de portas em OMNI



antes da conexão



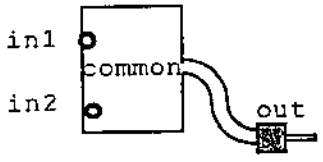
depois da conexão

Sobre o OMNI, a CO^2 constrói uma outra abstração. A figura 3-4 mostra a visão que CO^2 oferece dos comandos. Como já se disse, os comandos têm uma declaração da sua interface, que indica como eles interagem com o mundo exterior. Cada porta de entrada declarada na interface do comando é enxergada como sendo um soquete.

Toda vez que o comando é instanciado (executado) esses soquetes ganham uma identificação que é chamada de endereço de correio, ou simplesmente endereço. As portas de saída declaradas na interface são vistas como cabos saindo do comando, com um *conector macho* na sua ponta. Quando o comando é executado todos esses *conectores machos* devem ser conectados a um soquete já existente. Para fazer esta conexão é preciso conhecer o endereço do soquete. Um nome é ligado aos endereços atribuídos às portas de entrada de um comando, de forma que possam ser posteriormente referenciados em uma conexão[1].

Note-se a diferença entre o nome de uma porta de um comando (especificada na interface do comando) e o nome ligado ao endereço atribuído a uma porta de entrada de uma instância particular desse comando.

FIGURA 3-4 Visão de um comando em CO[2].



Para executar um comando o programador deve efetuar as seguintes ações:

- especificar os parâmetros do comando,
- colocar um nome a todo endereço atribuído às portas de entrada declaradas na interface e
- especificar os endereços aos quais devem conectar-se as portas de saída.

A forma mais trivial de conseguir o último item é especificar o nome ligado a um endereço num comando anterior.

Suponha-se ter um comando de nome `gerenciador`, que implementa o gerenciador de concorrência do sistema de teleconferência discutido na página 1-8. A interface deste comando tem uma porta de entrada de nome `controle`, na qual lê as requisições que os clientes lhe enviam. A seguinte linha provoca a execução deste comando ligando o nome `in-g` ao endereço atribuído à porta `controle`:

```
gerenciador controle:in-g;
```

1. Salienta-se a diferença entre os termos *conexão* e *ligação*. O primeiro denota o ato de conectar um *conector macho* a um soquete. O segundo é um relacionamento que se estabelece entre um nome e um valor, neste caso um endereço de correio.

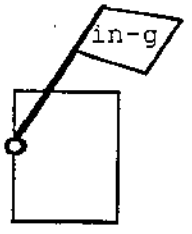
Depois de ativar a execução de uma instância do gerenciador e obter o endereço atribuído à porta de entrada, o interpretador regista a ligação do nome `in-g` a esse endereço e fica pronto para processar a próxima linha sem esperar a finalização do gerenciador. A parte superior da figura 3-5 mostra a única perturbação visível no espaço de nomes da CO^2 depois da execução deste comando. Só tem o nome `in-g` ligado a um endereço de correio. Assumindo que o programa que implementa o cliente se chama `interface`, e que esses programas enviam suas requisições para o gerenciador pela porta `requisições`, a seguinte linha executa uma instância deste comando conectando a sua porta de saída ao gerenciador:

```
interface requisições>in-g;
```

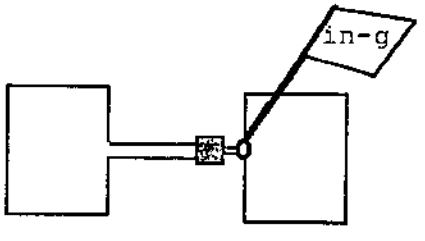
A especificação de um nome para uma porta de entrada é uma operação semelhante à atribuição: depois de executar-se o comando o nome fica ligado ao endereço com o qual foi instanciada a porta correspondente. A conexão de uma porta de saída a um endereço de correio assemelha-se à instanciação da lista de conhecidos no modelo de *actors*. Pode-se pensar uma porta de saída como sendo um conhecido ao qual o comando manda informação; a ligação desse conhecido com um destino real é só feita ao executar-se o comando.

FIGURA 3-5 Seqüência de comandos CO^2

```
gerenciador controle:in-g;
```




```
interface requisições>in-g;
```



3.2.1 Visualização de dados na tela

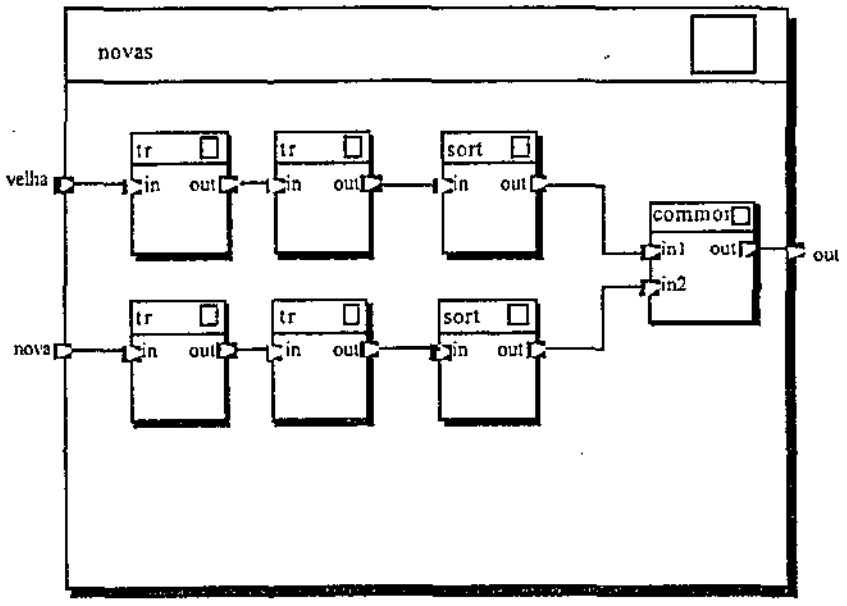
No início de uma sessão interativa com a CO^2 existe um endereço que envia para a tela toda informação que chegar a ele. Esse endereço é acessível com o nome `tty`. A figura 3-6 mostra a abstração em LegoShell da parte que faz o processamento na computação da figura 1-3, página 1-8. A seguinte linha de código executa essa computação de maneira que a sua saída seja visualizada na tela:

```
novas velha:velha-versão nova:nova-versão out>tty;
```

Para alimentar os endereços `velha-versão` e `nova-versão` com o conteúdo dos arquivos `versão-1` e `versão-2`, respectivamente pode-se utilizar o comando `cat`:

```
cat "versão-1" out>velha-versão;
cat "versão-2" out>nova-versão;
```

FIGURA 3-6 Abstração de uma computação LegoShell



3.3 Execução distribuída

O OMNI permite que um processo inicie outro processo em um computador remoto com as mesmas facilidades com que o iniciaria localmente. O mesmo pode ser afirmado sobre a conexão de portas. Desta forma todo o que for afirmado sobre a CO^2 é aplicável para execução remota. O modelo almejado para a CO^2 é de distribuição explícita; ou seja o programador especifica na linha de execução do comando o nome do computador onde ele deve ser executado da seguinte forma:

```
gerenciador@servidor controle:in-g;
```

O comando anterior pede a execução do programa `gerenciador` no computador `servidor`. Salienta-se que o modelo não prevê nenhum mecanismo de migração de código: o programa, cuja execução remota foi solicitada, deve existir no computador remoto com o nome especificado no comando que requereu sua execução.

3.4 Persistência em CO²

É de se esperar que em algum momento se deseje que os dados produzidos possam sobreviver à execução do programa, de forma que possam ser utilizados por futuras instâncias do mesmo programa, ou até por outros programas. Nesta seção explica-se a sintaxe para salvar os dados de uma porta de saída em objetos persistentes.

A implementação destes objetos persistentes foi feita utilizando-se o sistema de arquivos de Unix. Na definição da persistência cuidou-se de não interferir na semântica do modelo. Construções sintáticas são oferecidas para criar um endereço de correio associado a um arquivo de forma que todo dado que chegar a esse endereço é gravado no arquivo. Como a qualquer outro endereço de correio, uma porta de saída pode conectar-se a ele. A computação da figura 1-3, página 1-8, assumindo que a parte que faz o processamento está abstraída na computação `novas` (como mostra a figura 3-6), é codificada da seguinte forma:

```
novas velha:velha-versão nova:nova-versão out>(> "saída");
```

A expressão entre parênteses é avaliada antes de começar a execução do comando. O resultado desta avaliação é um endereço de correio ao qual todo dado que chega é gravado no arquivo `saída`. Se no momento da avaliação da expressão não existe um arquivo com esse nome, ele é criado; no caso de o arquivo existir, ele é re-inicializado, destruindo-se os dados que eventualmente possa ter. Caso a intenção seja não destruir esses dados, e sim acrescentar mais informação ao objeto já existente, a notação é `'(>> "saída")'`.

A falta de sincronismo do interpretador com os comandos por ele ativados coloca problemas de concorrência no uso de arquivos; estes problemas são abordados no capítulo 6.

3.5 Conectores especiais

Até aqui tem-se visto apenas exemplos de conexão ponto a ponto. A CO² também fornece suporte sintático para os conectores especiais. O seguinte código é a expressão da computação LegoShell da figura 3-1:

```
broadcast in:s2b out>(> "saida-1"),(> "saida-2");
sort in:e2s out>s2b;cat "entrada" out>e2s;
```

A expressão em CO^2 da computação da figura 3-2 é a seguinte:

```
merge in1:s2m1 in2:s2m2 out>(> "saída");
sort in:mb2s1 out>s2m1;sort in:mb2s2 out>s2m2;
mailbox in:e2mb out>mb2s1,mb2s2;
cat "entrada" out>e2mb;
```

3.6 Discussão

Depois de uma ligeira apresentação dos conceitos centrais do sistema OMNI apresentou-se o modelo primitivo de concorrência da CO^2, incluindo a utilização de arquivos e conectores especiais. Nos próximos três capítulos apresentam-se os aprimoramentos feitos sobre este modelo primitivo de forma que ele se adeqüe aos objetivos discutidos na seção 1.4 ("Um Modelo de Concorrência para CO2"). O capítulo 4 descreve como encapsular o modelo primitivo de forma a aumentar o nível da linguagem. Os estudos feitos sobre desenvolvimento aninhado são a matéria do capítulo 5 O capítulo 6. inclui uma extensão ao modelo para lidar com concorrência interna.

Capítulo 4

Programação em CO^2

Can thought about things be so much different from things? Can thinking processes be so unlike the actual process of things? In short, can thinking be so far removed from reality?

David Hilbert: *On the Infinite*

O modelo computacional primitivo da CO^2 é claramente operacional. Os modelos operacionais caracterizam-se pela necessidade de especificar os passos computacionais para atingir a solução. Uma vez que o fluxo de controle é explicitamente tratado pelo programador o paralelismo existente num programa também deve ser especificado por ele. Já nos modelos *definicionais*[1] um programa é composto de regras, equações, restrições e outras propriedades da solução desejada, sem preocupar-se com a forma em que essa solução é calculada. Nestes modelos o paralelismo de um programa é, de certa forma, invisível ao programador [Ambler, Burnett e Zimmerman, 1992].

A construção de um programa em CO^2 pode ser vista como a especificação de um grafo orientado. Adotando a terminologia da LegoShell, estes grafos são chamados de computações. Os vértices de uma computação são os comandos e os arquivos, e as arestas vêm a representar fluxos de dados entre esses vértices. Tal como foi apresentada até aqui, a CO^2 oferece primitivas para especificar um vértice. A declaração das arestas é feita junto com a declaração do vértice do qual elas saem; quando esta declaração é feita todos os destinos das arestas já devem existir. O programador deve ordenar a criação de vértices de forma a respeitar esta restrição. Assim, computações com ciclos não poderiam ser expressadas. Além desta limitação, este mecanismo de construção dos programas nem sempre reflete a forma de raciocinar do programador.

Em este capítulo analisa-se quão adequada é a CO^2 para expressar os diferentes tipos de interação entre processos, introduzindo novos recursos sintáticos para

1. O termo é uma adaptação de *definitional*.

favorecer a expressão de determinados casos onde o modelo primitivo se mostra inadequado.

4.1 Interação entre processos revistada

Neste trabalho faz-se questão de prover uma notação útil para expressar programas mediante a reutilização tanto de filtros quanto de clientes e servidores. A seguir discutem-se as motivações que levaram a acrescentar recursos de outros paradigmas ao modelo primitivo da CO^2.

Uma computação envolvendo clientes e servidores está estreitamente relacionada à concepção do mundo da *orientação a objetos*. Esta escola vê o mundo como sendo composto por uma comunidade de objetos. A única forma que um objeto tem de influenciar o comportamento de outro é enviando-lhe uma mensagem. Para um objeto enviar uma mensagem a outro aquele deve conhecer a identificação deste. Há implicitamente uma *restrição de seqüência*: a existência de um objeto (e portanto sua criação) é necessariamente anterior ao conhecimento da identidade desse objeto.

A programação distribuída com clientes e servidores se ajusta a este paradigma de programação. Na CO^2 a operação de conectar portas de saída a endereços, quando da execução de um programa, pode ser vista como a comunicação da identidade de outros processos (objetos) ao novo processo. O modelo permite expressar com naturalidade situações nas quais exista um relacionamento cliente-servidor entre os processos envolvidos. No entanto, o modelo força uma expressão pouco natural de programas com filtros. A restrição de seqüência força um raciocínio "de trás para a frente", o consumidor deve ser instanciado antes do que o produtor. Prova disto é a seguinte implementação em CO^2 da computação da figura 3-6, página 3-8:

```
common "-2" in1:c1 in2:c2 out>(> "saída");
sort "-u" in:s1 out>c1;sort "-u" in:s2 out>c2;
tr "-d" "a-z"  "\012" in:t12 out>s1;
tr "-d" "a-z" "\012" in:t22 out>s2;
tr "A-Z" "a-z" in:t11 out>t12;
tr "A-Z" "a-z" in:t21 out>t22;
cat "versão-1" out>t11;
cat "versão-2" out>t21;
```

4.2 Recursos Definicionais

Recursos definicionais foram incluídos na especificação da CO^2 de forma que as computações possam escrever-se sem ter que obedecer a restrições de seqüência

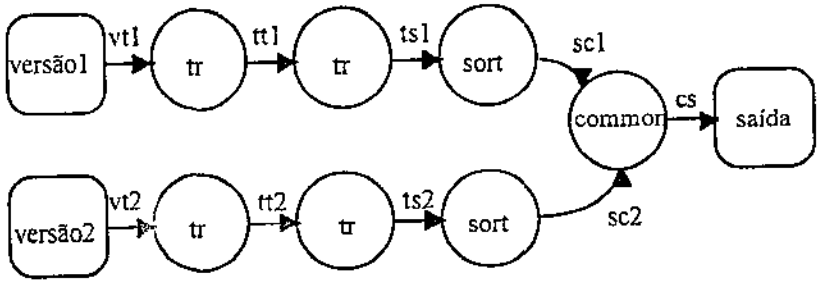
O esquema almejado é o seguinte: parte-se de um desenho da computação (grafo) com todos as suas arestas rotuladas. Escolhe-se arbitrariamente um primeiro vértice, e os restantes são escolhidos atendendo a algum critério de percurso do grafo. Ins-

tancia-se cada vértice nomeando as portas de entrada com o rótulo da aresta que chega em essa porta e conectando cada porta de saída ao rótulo da aresta que sai dela. Para suportar este estilo de programação, foi definida uma construção sintática, chamada de *bloco definicional*.

A figura 4-1 mostra o grafo para a computação da figura 1-3, página 1-8. Tanto os nomes das portas quanto os parâmetros foram propositadamente omitidos. Partindo do vértice `versão1` e percorrendo o grafo em profundidade (ignorando a orientação das arestas) o bloco definicional para expressar o grafo é o seguinte:

```
cat "versão1" out>vt1 &&
tr "A-Z" "a-z" in:vt1 out>tt1 &&
tr "-d" "a-z" "\012" in:tt1 out>ts1 &&
sort "-u" in:ts1 out>sc1 &&
common "-2" in1:sc1 in2:sc2 out>(> "saída") &&
sort "-u" in:ts2 out>sc2 &&
tr "-d" "a-z" "\012" in:tt2 out>ts2 &&
tr "A-Z" "a-z" in:vt2 out>tt2 &&
cat "versão2" out>vt2;
```

FIGURA 4-1 Fluxo de dados no problema das novas palabras



A semântica desta construção é simples de entender para quem já entendeu o modelo básico. O operador '`&&`' dá a sensação de avaliação simultânea dos comandos. Só resta explicar a interação entre um bloco definicional e o contexto no qual ele está inserido.

Em uma sessão de CO^2 existe sempre um ambiente. Este ambiente é uma função que vai de um espaço de nomes a um valor:

Ambiente:Nome→Valor

Até aqui esses valores só podem ser endereços de correio, no entanto também podem ser inteiros, *strings*, funções, etc. A execução de um comando provoca uma alteração neste ambiente —independentemente da semântica particular do comando— em decorrência da ligação de nomes a endereços. Se o nome já estava ligado a um valor no ambiente antes da execução do comando esse vínculo é quebrado. Todas as ligações de nomes a portas de entrada feitas num bloco definicional ficam registadas no

ambiente durante e após a execução dele. Os nomes que não foram ligados no bloco permanecem ligados aos mesmos valores aos quais estavam ligados antes da execução dele. Dentro do bloco o ambiente é o mesmo para todos os comandos que o compõem.

O seguinte trecho de código é uma forma alternativa de expressar o bloco definicional anterior combinando um bloco definicional com outros comandos. Observe-se o "percurso" dos nomes `cs`, `vt1` e `vt2`.

```
set cs (> "novas");
tr "A-Z" "a-z" in:vt1 out>tt1 &&
tr "-d" "a-z" "\012" in:tt1 out>ts1 &&
sort "-u" in:ts1 out>sc1 &&
common "-2" in1:sc1 in2:sc2 out>cs &&
sort "-u" in:ts2 out>sc2 &&
tr "-d" "a-z" "\012" in:tt2 out>ts2 &&
tr "A-Z" "a-z" in:vt2 out>tt2;
cat "versão2" out>vt2;cat "versão1" out>vt1;
```

No código acima foi introduzido o comando `set`. Este comando envolve um nome e uma expressão que é avaliada, registando-se no ambiente a ligação entre o nome e o resultado da avaliação.

A implementação dos blocos definicionais utilizando-se das primitivas do OMNI é trivial. Ela é feita em duas fases: na primeira todos os comandos envolvidos são executados registando-se os endereços de todas as portas, e na segunda são feitas as conexões.

A principal contribuição da introdução dos recursos definicionais é que eles possibilitam a expressão da topologia de uma computação sem ter que se preocupar com a forma em que ela é construída. Entretanto, perdem-se as vantagens do modelo interpretado, pois os todos os comandos envolvidos nos blocos definicionais são executados de uma vez. Na próxima seção apresenta-se uma extensão ao modelo primitivo de forma que seja possível flexibilizar a restrição de seqüência sem perder as vantagens do modelo interpretado.

4.3 Programação com fluxos de dados

Uma das vantagens do modelo interpretado é a possibilidade de executar o programa aos poucos, parando eventualmente para conferir se os resultados parciais batem com os valores esperados. Como foi dito na seção 4.1 o modelo primitivo força um raciocínio "de trás para a frente", limitando assim os benefícios do modelo interpretado. O ideal seria atingir um modelo interpretado mais flexível na restrição de seqüência.

No exemplo de correção ortográfica da página 1-3 a idéia é que o primeiro comando troque todas as minúsculas por maiúsculas, o segundo produza uma saída de uma palavra por linha, e assim por diante. No entanto, pode ser o caso de que o programador não conheça os comandos na profundidade suficiente e que duvide se os comandos realmente estão fazendo o que ele deseja que façam. Seria desejável que ele conseguisse executar o primeiro comando, ter a possibilidade de parar para inspecionar os resultados desse passo, executar o segundo, e assim por diante. Nem o modelo primitivo nem os recursos definicionais oferecem esta facilidade.

Para contornar esta limitação foram introduzidos os fluxos de dados[1] na CO[2]. Para criar um fluxo de dados com o conteúdo do arquivo texto com todas as suas maiúsculas substituídas por minúsculas executam-se os seguintes dois comandos:

```
set faz-minusculas (streamize in out "tr 'A-Z' 'a-z'");
set minusculas (faz-minusculas (streamize-file "texto"));
```

De posse do fluxo de dados armazenado na variável `minusculas`, seu conteúdo poder ser visualizado na tela com o seguinte comando:

```
feed minusculas tty;
```

Um fluxo de dados pode ser utilizado quantas vezes se deseje, o seguinte comando armazena o conteúdo do fluxo no arquivo `as-minusculas`:

```
feed minusculas (> "as-minusculas");
```

A execução do corretor ortográfico continuaria com os seguintes dois comandos:

```
set faz-palavras (streamize in out "tr '-c' 'a-z' '\012'");
set palavras (faz-palavras minusculas);
```

O custo de introduzir fluxos de dados foi quase zero, uma vez que foi realizado utilizando o modelo primitivo. A função `streamize` retorna uma *receita genérica* para computar o fluxo de dados, mas não o computa. A aplicação dessa receita genérica a um fluxo de dados retorna uma *receita pronta* para computar o fluxo de dados, mas também não o computa. O comando `feed` provoca a execução dessa receita. Tudo isto foi conseguido aplicando conceitos elementares de programação funcional, como é mostrado a seguir.

1. Em inglês, *streams*.

O comando

```
set faz-minusculas (streamize in out "tr 'A-Z' 'a-z'");
```

é traduzido para

```
fn faz-minusculas (f)
   (lambda (dest)
      (let tr 'A-Z' 'a-z' in:x out>dest
         in (f x)));
```

A expressão

```
(streamize-file "texto")
```

é traduzida para

```
(lambda (dest) (cat "texto" out>dest))
```

logo, a aplicação

```
(faz-minusculas (streamize-file "texto"))
```

evolui da seguinte forma:

```
(lambda (dest)
     (let tr 'A-Z' 'a-z' in:x out>dest
        in ((lambda (dest) (cat "texto" out>dest)) x);
...
(lambda (dest)
     (let tr 'A-Z' 'a-z' in:x out>dest
        in (cat "texto" out>x)));
```

Ou seja que o comando

```
set minusculas (faz-minusculas (streamize-file "texto"));
```

seria equivalente a

```
fn minusculas (dest)
   (let tr 'A-Z' 'a-z' in:x out>dest
        in (cat "texto" out>x)));
```

Por outro lado, o comando

```
feed minusculas tty;
```

é internamente traduzido para

```
(minusculas tty);
```

Sendo que `minusculas` nada mais é do que uma função, a aplicação anterior resulta na seguinte expressão:

```
(let tr 'A-Z' 'a-z' in:x out>tty
     in (cat "texto" out>x));
```

Retirando o formalismo funcional a expressão é equivalente a

```
tr 'A-Z' 'a-z' in:x out>tty;
cat "texto" out>x;
```

Que seriam os comandos que deveriam ser executados no modelo primitivo para conseguir o efeito desejado.

4.4 Discussão

Recursos para acrescentar a potência semântica da CO^2 foram apresentados neste capítulo. Depois de uma breve discussão sobre os casos nos quais o modelo primitivo da linguagem se mostra fraco, foram apresentadas as construções sintáticas que permitem ao programador expressar em um estilo definicional partes de um programa. Se bem que limitados no seu escopo de aplicação, os blocos definicionais introduzem as vantagens da programação definicional para resolver alguns casos. A implementação destes blocos é feita sem prejudicar a eficiência, quando comparada com a mesma solução expressa num estilo operacional. O custo notacional de acrescentar estas construções é quase zero: apenas um operador é introduzido, que dá a idéia de uma avaliação em paralelo de todos os comandos envolvidos. A principal restrição dos blocos definicionais é que só podem envolver comandos externos, o *broadcast* e o *mailbox*. Outros comandos, como o `set`, não podem ser incluídos num bloco definicional.

Nas linguagens definicionais não existem restrições de seqüência; isto resulta em facilidades para a verificação de programas e para o aproveitamento do paralelismo. Na década de 70 estas linguagens mereceram especial atenção na pesquisa em linguagens de fluxo de dados, chegando-se a estudar construções iterativas definicionais [Arvind, Gostelow e Wolfe, 1978, capítulo 2; Ackerman, 1982]; soluções mistas também foram abordadas [Kessels, 1977; Treleaven, Hopkins e Rautenbach, 1982].

A computação com fluxos infinitos de dados foi tratada também em linguagens de fluxo de dados [Arvind, Gostelow e Wolfe, 1978, capítulo 5]. Contudo, cabe destacar que estes tipos de dados não foram estudados só no contexto do paralelismo; listas infinitas podem ser implementadas em linguagens funcionais com avaliação tardia [Abelson e Sussman, 1985].

Capítulo 5

Níveis de Abstração Diferenciados

A cada uno de los muros de cada hexágono corresponden cinco anaqueles; cada anaquel encierra treinta y dos libros de formato uniforme; cada libro es de cuatrocientas diez páginas; cada página, de cuarenta renglones, cada renglón, de unas ochenta letras de color negro.
Jorge Luis Borges: *La Biblioteca de Babel*

A CO^2 deve permitir o desenvolvimento em níveis diferenciados de abstração. Isto significa a possibilidade de trechos de código de CO^2 serem abstraídos em comandos, que possam ser utilizados por outros programas CO^2 e assim por diante. Mais ainda, essas abstrações devem ser manipuladas com a mesma sintaxe e semântica com que são tratados os programas em Cm e LegoShell, tal como foi esquematizado na figura 1-2, página 1-7.

Este capítulo trata sobre a construção e utilização de arquivos de comandos da CO^2. O capítulo foi organizado em duas partes. Na primeira apresenta-se a abstração de blocos definicionais, e ainda nesta primeira parte começa-se a delinear um novo exemplo. A segunda metade mostra como codificar processos que se comunicam com outros processos utilizando-se de primitivas de entrada e saída; o desenvolvimento do exemplo é finalizado nesta parte.

5.1 Configurações

Blocos definicionais podem ser abstraídos em *configurações*. Uma configuração comporta um nome, parâmetros formais, portas de entrada, portas de saída e um bloco definicional. Cada porta de entrada da configuração é um nome ligado à porta de entrada de um e só um comando do bloco definicional. Uma porta de saída da configuração é um nome de um endereço genérico ao qual se conecta a porta de saída de algum comando envolvido no bloco definicional; esse nome não deve estar ligado a nenhuma porta de entrada no escopo do bloco definicional. Os parâmetros formais

de uma configuração são nomes que podem aparecer como argumentos nos comandos do bloco definicional. A seguir, apresenta-se uma configuração que implementa um corretor ortográfico:

```
#!/usr/ahand/bin/co2 -c -I texto -o erros dicionário
tr "A-Z" "a-z" in:texto out>tt &&
tr "-d" "a-z" "\012" in:tt out>ts &&
sort "-u" in:ts out>sc &&
common "-2" in1:sc in2:i2 out>erros &&
cat dicionário out>i2;
```

Para poder executar esta configuração o código deve estar armazenado em um arquivo que tenha permissão de execução; o nome do arquivo é o nome com que a configuração é invocada. A primeira linha descreve a interface desta configuração[1]. O parâmetro '-c' informa à CO[2] que se trata do código de uma configuração. O identificador após o parâmetro '-I' é o nome de uma porta de entrada com conexão. Analogamente, o identificador que segue ao parâmetro '-o' é o nome de uma porta de saída. Assim, esta configuração tem uma porta de entrada com conexão de nome `texto` e uma porta de saída de nome `erros`. Portas de entrada sem conexão são indicadas com o parâmetro '-i'. Supondo que o código apresentado acima esteja armazenado num arquivo de nome `corretor`, as seguintes linhas permitem ver na tela os erros ortográficos do arquivo `tese`, segundo o dicionário `computação`.

```
corretor "computação" texto:entrada erros>tty;
cat "tese" out>entrada;
```

A seguir apresenta-se um exemplo de cálculo de fecho.

Exemplo 5.1.

Dispõe-se de um grafo conexo orientado, cada vértice é identificado com uma seqüência de letras e dígitos. Deseja-se saber todos os vértices atingíveis a partir de um vértice v_1. O grafo é representado com um arquivo de texto. Toda linha do arquivo tem o nome de dois vértices separados por um espaço. Cada linha representa uma aresta que vai do primeiro vértice ao segundo[2].

O algoritmo está baseado na aplicação a exaustão do seguinte raciocínio: há um caminho do vértice v_1 para um outro vértice v_2 se há uma aresta de v_1 para v_2, ou então se há um caminho desde v_1 para um terceiro vértice v_3 e há uma aresta de v_3 para v_2.

1. Ao se invocar a execução de um arquivo que, ao invés de conter código do processador, contém texto, o núcleo do Unix chama a programa cujo nome está imediatamente depois da sequencia '#!' para interpretar o conteúdo desse arquivo. O restante desta primeira linha é passado como parâmetro a este interpretador.

2. A título de exemplo diga-se que o arquivo poderia ser a saída de um comando que analisa a dependência direta entre as funções de um programa.

FIGURA 5-1 Algoritmo para cálculo de fecho

Entrada: $G = (V, E)$ *(um grafo orientado)* $e\ v_1 \in V$
Saída: $S \subseteq V$ *(vértices atingíveis desde* v_1*)*
$Q_{antes} \leftarrow 0$
$S = \{v \in V / (v_1, v) \in E\}$
$Q_{depois} \leftarrow |S|$
enquanto $Q_{antes} != Q_{depois}$
$Q_{antes} \leftarrow Q_{depois}$
$S = \{v \in V / \exists x \in S_{ciclo\text{-}anterior} : (x,v) \in E\}$
$Q_{depois} \leftarrow |S|$
fim enquanto

A figura 5-1 apresenta o algoritmo. O parágrafo seguinte é a implementação em CO[2] deste algoritmo. Assume-se que o grafo está armazenado no arquivo G e que o conjunto solução no arquivo S, sendo que cada linha dele representa um elemento (um vértice).

```
set q-antes "0";
cat "G" out>g2in &&
inicializa in:g2in out>(> "S") v1;
set q-depois (quant-linhas "S");
while (q-antes != q-depois)
   set q-antes q-depois;
   cat "G" out>g2ac &&
   cat "S" out>s2ac &&
   acrescenta solução:s2ac grafo:g2ac out>(> "T");
   cat "T" out>(> "S");
   set q-depois (quant-linhas "S");
end;
```

No código têm-se dois comandos, `inicializa` e `acrescenta`, os quais efetuam, respectivamente, a inicialização do arquivo que implementa a solução e a adição de novos nomes de vértices a ele. A função `quant-linhas` conta as linhas de um arquivo.

A inicialização é só um filtro que imprime a segunda palavra de todas as linhas que comecem com v_1[1]. Mais interessante é a implementação da configuração `acrescenta`. A figura 5-2 mostra a representação em LegoShell do esquema almejado. A representação do grafo entra pela porta `grafo` e a solução (parcial) pela porta `solução`. O comando `join` implementa os operadores de junção e projeção da álgebra relacional[2] sobre relações materializadas em arquivos de texto, desde que

1. Esta ação pode ser especificada simplesmente em *awk* com a linha "`$1 == v1 {print $2}`"
2. Vide [Date, 1986]

ordenadas pela chave de junção. A saída deste comando deve ser acrescentada, eliminando os dados duplicados, à solução parcial. Um conector estrela é introduzido com o intuito de que a solução parcial alimente tanto o `join` quanto o comando que implementa a operação de união de conjuntos. O código em CO^2 de `acrescenta` é o seguinte:

```
#!/usr/ahand/bin/co2 -c -I grafo -I solução -o out
broadcast in1:solução out1>b2j out2>b2u &&
join "-o" "1.2" in1:grafo in2:b2j out>j2u &&
união in:b2u out>out;
```

A figura 5-3 ilustra a configuração `união`. Os dados que chegam por duas portas `in1` e `in2` são misturados não deterministicamente pelo conector M e enviados a um `sort` com o parâmetro '`-u`'. O código desta configuração é apresentado a seguir.

```
#!/usr/ahand/bin/co2 -c -I in1 -I in2 -o out
mailbox in1:in1 in2:in2 out1>mb2s &&
sort "-u" in:mb2s out>out;
```

FIGURA 5-2 Esquema da configuração `acrescenta`

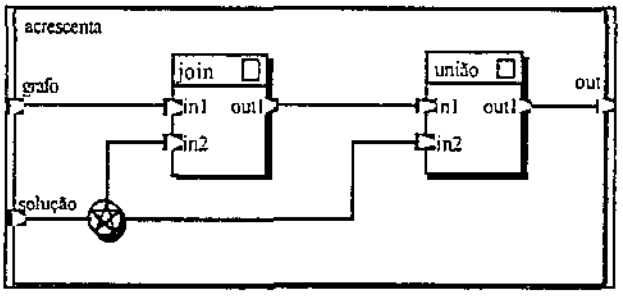
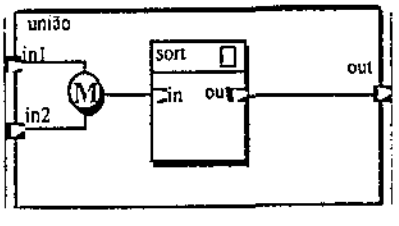


FIGURA 5-3 Esquema da configuração `união`



5.2 Entrada e saída

As configurações não servem quando é necessário que o código CO^2 faça algum tipo de processamento com os dados. Uma vez que o modelo é de processos que se co-

municam mediante primitivas de entrada e saída, a CO^2 precisa de suas próprias primitivas para que seus programas possam comunicar-se com outros processos.

A primitiva de saída envolve o endereço de correio ao qual se destinam os dados e os próprios dados. Para imprimir um texto na tela é só enviá-lo para o endereço de correio ligado a `tty`:

```
tty ! "Onde queres revólver sou coqueiro";
```

Utilizando a configuração `corretor`, da seção anterior, uma correção interativa poderia ser feita da seguinte forma:

```
corretor "aurélio" texto:entrada erros>tty;
entrada ! "A tonga da mironga do kabuletê";
unset entrada;
```

O comando `unset` desliga um nome no ambiente. Se o nome estiver ligado a um endereço de correio conectável o processo dono desse endereço recebe a sinalização de que a conexão foi fechada.

5.2.1 Tipos

O esquema de tipos da *shell* é muito simples. Os argumentos passados aos comandos e os valores das variáveis são todos cadeias de caracteres, não existe outro tipo de dados. As unidades de dados que atravessam um *pipeline* de comandos são, geralmente, linhas de texto. O esquema pode parecer limitado; no entanto, é esta característica que faz que partes de programas possam ser testadas e depuradas, separada e interativamente. O texto é facilmente imprimível, sem depender fortemente do dispositivo de saída. Utilizar texto como tipo dos dados de entrada faz com que seja simples a geração de dados para teste.

A CO^2 tem mais do que um tipo de dados. Até aqui apareceram, além das cadeias de caracteres, endereços de correio e funções. Não é de interesse comunicar funções de um comando a outro, pois a linguagem na qual um comando foi implementado tem que ser invisível fora dele, e as funções não têm uma representação homogênea para todas as linguagens do A_Hand. Já os endereços de correio (identificadores de portas) são uma abstração fornecida pelo OMNI a todas as linguagens do A_Hand. No que resta deste capítulo, e no seguinte, mostra-se a necessidade de os comandos trocarem endereços de correio entre eles.

A criação de um endereço de correio é feito por uma função embutida na CO^2.

```
set minha-entrada (create-port "x");
```

O parâmetro 'x' pede uma porta conectável. A saída de um comando pode-se conectar a esta porta.

```
wc "-l" in:entrada-wc out>minha-entrada;
```

Uma função para ler um endereço de correio e outra para ler uma linha de texto são fornecidas pela CO^2; ambas as funções de leitura bloqueiam caso a porta não tenha dados suficientes para completar a operação. A seguinte função retorna a quantidade de linhas de um arquivo.

```
fn quant-linhas (arquivo)
   let
      set minha-entrada (create-port "x");
      cat arquivo out>entrada-wc &&
      wc "-l" in:entrada-wc out>minha-entrada;
   in
      (get-line minha-entrada);
```

5.3 Programas que processam informação

O código de uma configuração, a exemplo de Darwin, não faz nenhum processamento, só especifica os comandos que se quer ativar, e o fluxo da informação entre eles. Com referência à figura 1-2, página 1-7, configurações são necessariamente nós não-folha da árvore de objetos do A_Hand. Nestá seção explica-se o uso de código CO^2 para escrever programas que recebem dados por portas de entrada, fazem computações com esses dados e enviam dados para outros processos.

Uma vez que há primitivas para ler de portas de entrada, um programa CO^2 pode receber informação de qualquer processo que conheça o endereço de correio dessa porta. A exemplo do Cm, um programa CO^2 pode ter portas declaradas na sua interface; os endereços de correio destas portas, criadas implicitamente pelo sistema de execução da CO^2, são conhecidos pelo agente que provocou a execução do programa.

O nome de uma porta declarada na interface de uma configuração é apenas o nome com que se exporta o endereço de correio da porta de algum comando envolvido na configuração. Por exemplo, na figura 5-2 `grafo` é o nome com que é exportado o endereço de correio associado à porta `in1` do `join`, embora haja dois ícones que sugerem a existência de duas portas. A principal diferença entre os arquivos de comandos introduzidos nesta seção, chamados de *roteiros*, e as configurações, é que nos roteiros as portas declaradas na interface devem ser criadas a cada vez que o roteiro é executado.

O seguinte é o código de um roteiro que conta todas as linhas que chegam pela sua porta `in`:

```
#!/usr/ahand/bin/co2 -r -I in -o out
set conta 0;
while (conectado in)
   set conta (conta + 1);
   set linha (get-line in);
```

```
end;
out ! conta;
```

O parâmetro '-r' indica que o arquivo de comandos é um roteiro. A função conectado bloqueia o processo até que tenha dados disponíveis para a leitura ou então até que a conexão seja fechada.

A seguir apresenta-se uma implementação muito simples de um gerenciador de recursos. Este gerenciador recebe dois tipos de requisições: pedindo a permissão para utilizar o recurso e devolvendo a dita permissão. Este programa poderia ser ligado dinamicamente a um protótipo de gerenciador de banco de dados ou então pode vir a ocupar o lugar do gerenciador de concorrência descrito na seção 1.4.2, página 1-7, onde a permissão significaria o direito de falar do participante. No código supõe-se a existência de uma estrutura de dados que implementa uma fila.

```
#!/usr/ahand/bin/co2 -r -i requisições -o console
set estado "livre";
inicializa a fila esperando
while (1)
   set tipo-req (get-line requisições);
   if (tipo-req = "pede") then
      set pediu (get-mail-address requisições);
      if (estado = "ocupado") then
         coloca pediu no fim da fila esperando
      else
         pediu ! "vai";
         set estado  "ocupado";
      end;
   else
      if (tipo-req = "devolve") then
         if (fila esperando está vazia) then
            set estado "livre";
         else
            Retira primeiro elemento de esperando
                e chama-o de prox
            prox ! "vai";
         end;
      else
         console ! "requisição desconhecida" tipo-req
      end;
   end;
end;
```

O '-i' indica que a porta requisições é não-conectável. Supondo que este código se encontra armazenado num arquivo de nome gerenciador, o programador pode executar ele com a seguinte linha:

```
gerenciador requisições:req console>tty;
```

Se o programador quiser participar da sessão interativamente, ele poderia pedir a permissão digitando os seguintes comandos:

```
set uma-porta (create-port "n");
req ! "pede" uma-porta;
set resposta (get-line uma-porta);
```

5.4 Discussão

Apresentaram-se os recursos da CO^2 que permitem o desenvolvimento de programas em distintos níveis de abstração. Tanto as configurações quanto os roteiros CO^2 podem ser utilizados por código de CO^2 ou por computações da LegoShell, utilizando-se a mesma sintaxe usada para executar comandos escritos em Cm. À diferença da LegoShell, a CO^2 é capaz de processar dados.

A diferenciação entre configurações e roteiros foi necessária devido ao fato de que estas precisam da criação de novos endereços de correio nos quais possam ler dados e aquelas só necessitam exportar endereços de correio que pertencem a outros comandos.

A comunicação entre processos é mediante primitivas de entrada e saída. Estas primitivas estão fortemente orientadas a linhas de texto. No entanto, considerou-se necessário acrescentar a possibilidade de incluir endereços de correio nas primitivas de entrada e saída para os casos em que seja preciso um encaminhamento de respostas.

Duas pendências restam para o próximo capítulo. A primeira vem persistindo desde capítulos anteriores, mas fica mais evidente na solução do exemplo de cálculo de fecho do exemplo 5.1: o acesso concorrente a arquivos. Se o ciclo do programa for desdobrado teríamos o arquivo s servindo simultaneamente de entrada e saída de vários comandos. Um certo sincronismo entre esses comandos é necessário, para conseguir que a computação toda seja determinística. A segunda pendência é uma representação para a fila da implementação do gerenciador da seção 5.3. Embora os problemas sejam de distinta natureza ambos são resolvidos com a introdução de um novo tipo de dados.

Capítulo 6

Concorrência Interna

Concurrency involves a nondeterministic interleaving of events.

Gul Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*

São muitas as vantagens de construir um sistema distribuído mediante a composição de processos seqüenciais comunicantes. A impossibilidade de que o estado de um processo seja modificado pelos efeitos colaterais das ações executadas em outro processo, somada ao fato de que a interação entre processos só se faz em pontos facilmente individualizáveis, resulta em um código simples de verificar e depurar. No entanto, há problemas cuja solução seqüencial é muito trabalhosa ou até impossível.

Na primeira seção deste capítulo analisam-se as situações que exigem a introdução de concorrência dentro dos processos. A segunda seção apresenta o modelo adotado para a CO2. A aplicação destes mecanismos para a utilização de arquivos é discutida na terceira seção. Um balanço da solução encontrada para oferecer concorrência interna encerra o capítulo.

6.1 Por que concorrência interna?

A seguir distinguem-se dois casos que foram colocados durante a definição do modelo de concorrência da CO^2 e que trouxeram à tona o problema da concorrência interna. Estes casos não são ortogonais, por vezes um caso é apenas uma forma diferente de pensar o outro. No entanto, considerou-se adequado apresentar ambos em separado.

6.1.1 Processamento concorrente de requisições em um servidor

Considere-se um observador externo ao gerenciador de permissões cujo código foi apresentado na seção 5.3, página 5-6 . Este observador não consegue ver nem enten-

der o que se passa dentro do servidor, mas consegue ver as permissões que chegam e as respectivas respostas. Suponha-se que ele vê a seguinte seqüência de eventos: o servidor aceita uma requisição que lhe pede a permissão, em seguida ele vê o servidor responder ao cliente; a seguir ele vê o servidor aceitar outra requisição que também lhe pede a permissão; uma vez que a permissão está com outro cliente, a resposta é adiada e o servidor aceita a próxima requisição. Neste ponto o observador tem a ilusão de que o servidor está processando concorrentemente as duas últimas requisições. Embora o problema tenha sido resolvido num estilo seqüencial, o gerenciador de permissões é potencialmente concorrente.

Analise-se em detalhe esta situação. Quando o gerenciador de permissões percebe que não está com a permissão, ele enfileira o endereço do cliente que a requereu e lê a próxima requisição. Ao receber a permissão de volta o gerenciador vê se há algum cliente à espera dela. Se tiver, retira o endereço da fila e envia-lhe a permissão. Seria muito mais abstrato (e cômodo) que, ao invés de ter que especificar todos estes passos, pudesse-se especificar apenas o fato central: "não dê a permissão enquanto outro cliente estiver com ela". O enfileiramento manual das requisições adiadas poderia ser comparado ao empilhamento manual dos parâmetros reais de uma função.

Há muitos modelos de concorrência nos quais, ao invés de ter primitivas de leitura, a recepção é abstraída em funções ou procedimentos (Distributed Processes [Brinch Hansen, 1978], RPC [Birrel e Nelson, 1984]) ou métodos de objetos (Emerald [Black et al., 1986], ABCL/1 [Yonezawa, 1990], Argus [Liskov et al., 1988]). O código do procedimento, ou do método, é ativado quando da chegada de uma mensagem; o conteúdo da mensagem instancia os parâmetros formais, o código da função faz o processamento e o valor retornado é enviado ao cliente que fez a requisição.[1] No entanto, a abstração deve contemplar formas de expressar a concorrência dentro dos servidores, de sorte que algumas situações (como as que o viu observador externo ao gerenciador de permissões) possam ser expressadas. Alguns modelos não têm nenhum recurso para resolver esta situação (RPC), outros resolvem-na de maneira concorrente (Argus) e outros de maneira pseudo-concorrente (ABCL/1) [Wegner, 1990].

Na abordagem concorrente um novo *thread* é ativado para cada requisição que chegue ao servidor. Estes *threads* compartilham o mesmo espaço de dados. É claro que algum mecanismo de sincronização entre eles é necessário para proteger as regiões críticas.

No mecanismo pseudo-concorrente muitos *threads* podem estar em andamento, mas só um está em execução; o escalonamento entre estes processos não é *preemptivo*, pois cada *thread* é executada até o fim ou até que decida esperar uma condição.

1. Esta explicação é genérica, não se refere a nenhum modelo em particular; cada modelo é uma variação deste esquema.

Na tentativa de atender este tipo de situações, sem utilizar mecanismos concorrentes nem pseudo-concorrentes, pode-se ter uma porta de entrada distinta para cada tipo de requisição. Assim, a cada iteração o servidor não tentaria ler dados daquelas portas que recebem requisições que ele não está em condições de processar. A seguinte seção analisa em detalhe esta idéia.

6.1.2 Leitura de várias portas simultaneamente com bloqueio

O gerenciador de permissões apresentado recebe os dois tipos de requisições pela mesma porta. Uma alternativa seria ter uma porta para cada tipo de requisição. Assim, o gerenciador de permissões pode ser escrito da seguinte forma:

```
#!/usr/ahand/bin/co2 -r -i pede -i devolve
while (1)
   set pediu (get-mail-address pede);
   pediu ! "vai";
   eval (get-line devolve);
end;
```

Tem-se expressado o gerenciador de permissões de uma forma elegante, evitando-se a programação explícita das ações envolvidas no adiamento de requisições. Os pedidos de permissão são extraídos só quando podem ser atendidos. Esta expressão reduzida do servidor foi conseguida graças às características do problema, pois não faz sentido o servidor esperar o próximo pedido de permissão enquanto ela não for devolvida. No entanto, pode-se ter situações nas quais seja preciso esperar dados em mais do que uma porta de entrada, a exemplo de um servidor que gerencie permissões de leitura e escrita de um determinado recurso. O problema é bem conhecido: a permissão de leitura é chamada também de compartilhável, pois vários processos podem estar de posse dela simultaneamente; já a permissão de escrita é exclusiva, ela pode ser dada a um processo apenas se nenhum outro estiver com uma permissão (compartilhável ou exclusiva), e nenhuma permissão pode ser dada enquanto um processo estiver com ela. Depois de outorgar uma permissão compartilhável, o servidor deveria ser capaz de aceitar (e processar completamente) tanto novos pedidos de permissão compartilhável quanto a devolução da permissão. No entanto, uma vez que a interação entre processos concorrentes não é determinística, não se pode predizer a ordem destes eventos (novo pedido de permissão compartilhável e liberação de permissão). Logo, o servidor deve esperar ambos os eventos.

Uma alternativa para lidar com esta situação é o processo ficar em uma iteração, consultando ambas as portas até que alguma tenha dados para ler. Como já é sabido, esta solução é desaconselhável se o processo for compartilhar o processador com outros processos. É desejável que o processo fique bloqueado até uma mensagem chegar em alguma das portas e que, ao continuar-se a execução, seja possível saber qual o evento que desbloqueou o processo.

Também aqui cabem as soluções concorrentes e pseudo-concorrentes. Contudo, existe uma outra forma para resolver a leitura bloqueante de várias portas que se in-

sere com naturalidade em modelos seqüenciais: os comandos com guarda ou cláusula *select* (CSP, Distributed Processes, Ada). Com esta construção sintática o programador especifica, para cada porta, o código a ser executado quando da aceitação de uma mensagem. Condições para a aceitação da mensagem podem ser especificadas de sorte que uma mensagem não seja aceita quando o estado do servidor não está em condições de processá-la. Assim, os comandos com guarda também permitem lidar com alguns casos de atendimento concorrente de requisições.[1]

A seguinte tabela resume as diferenças entre processos seqüenciais sem comandos com guarda (estritamente seqüenciais), processos seqüenciais com comandos com guarda, processos internamente pseudo-concorrentes e processos internamente concorrentes.

TABELA 14-2 Comparação de modelos de concorrência interna

Tipo de processos	Quantidade de *threads*	Escalonamento
Estritamente seqüenciais	Uma, determinística	
Seqüenciais + comandos com guarda	Uma, não determinística	
Pseudo-concorrentes	Zero ou mais, determinísticas	Não *preemtivo*
Concorrentes	Zero ou mais, determinísticas	Preemtivo

6.2 Concorrência interna em CO[2]

A necessidade de oferecer um mecanismo de mais alto nível para expressar o processamento concorrente de requisições e de lidar com várias portas de entrada simultaneamente foram evidência suficiente de que o modelo estritamente seqüencial limitava o escopo de aplicação da CO[2].

A abordagem pseudo-concorrente foi rejeitada por não ser adequada para um modelo interativo: a execução de um comando de suspensão do processamento (*wait*) vindo do terminal significaria a perda de controle da sessão desse terminal.

1. No entanto, uma vez que o processamento de uma requisição não pode ser interrompida, se dificulta a expressão de problemas nos quais o adiamento do processamento não só depende do estado do servidor antes de receber a mensagem, senão também da informação que vier na mensagem. Vide [Wegner e Smolka, 1983; Liskov et al, 1986].

Para poder executar a cláusula *select*, é necessário ter o código inteiro dela, o que a torna pouco adequada para um modelo interativo. Adotou-se uma solução concorrente baseada em *actors*.

6.2.1 Objetos

Um novo tipo de dados, chamado de objetos, são acrescentados à CO^2. Estes são executados no mesmo espaço de endereçamento que o processo que os criou, portanto a sua criação é várias ordens de magnitude mais barata do que a execução de um comando. Os objetos introduzem concorrência de duas formas: a execução do código de inicialização do objeto em paralelo com o criador, e a possibilidade de enviar uma mensagem ao objeto sem ter que esperar o fim de seu processamento.

6.2.2 Classes

Uma *classe* é o código que rege o comportamento do objeto. A seguir exemplifica-se a definição de uma classe para implementar um semáforo.

```
class semáforo (s)
$1 = "wait" && s = 0  { set s (s - 1);}
$1 = "signal"  { set s (s + 1);}
end;
```

O identificador logo após a palavra 'class' é o nome da classe. De posse deste identificador instâncias da classe podem ser criadas com a função new. Os identificadores entre parênteses depois do nome da classe são utilizados para importar valores dentro do código da classe quando da criação de uma instância. A seguinte linha cria um semáforo binário e atribui ele ao nome sb.

```
set sb (new semáforo 1);
```

A especificação do que deve ser feito com cada mensagem é feita com uma seqüência de pares *condição-ação*. No exemplo da classe semáforo há dois destes pares. Esta construção esta inspirada no *awk* Aho, Kernighan e Weinberger, 1988, a semântica é explicada a seguir. A palavra 'end' fecha o código da classe. A exemplo do *awk*, também é suportada a condição BEGIN para ser executadas antes de processar a primeira mensagem.

6.2.3 Envio de mensagens

Há duas semânticas que caracterizam a comunicação nos modelos baseados em troca de mensagens: comunicação síncrona e comunicação assíncrona. Na comunicação assíncrona a mensagem é colocada em um *buffer* sob controle do sistema de execução, liberando o emissor da mensagem para continuar sua execução. Nos modelos baseados em comunicação síncrona o emissor é bloqueado até que a mensagem seja (pelo menos) recebida pelo destinatário; em alguns modelos o emis-

sor só é desbloqueado após um *commitment*, implícito ou explícito, do receptor. As vantagens de um tipo de comunicação são desvantagens do outro. A comunicação assíncrona permite obter mais paralelismo. No entanto, muitas vezes uma mensagem é enviada para um outro processo com o intuito de sincronizar-se com ele, ou para lhe requerer uma resposta; para estes casos a comunicação assíncrona resulta de baixo nível. Modelos que combinam ambas as semânticas foram desenvolvidos (SR Andrews, 1981; ABCL/1 Yonezawa et al, 1986). Uma característica destes modelos é que a semântica de comunicação é especificada quando do envio da mensagem, e o receptor está sempre pronto para atender ambos os tipos de comunicação. Esta característica faz mais flexível a reutilização de servidores em diferentes contextos.

A CO^2 tem os dois mecanismos de envio de mensagens. O envio síncrono apresenta-se sob a forma da função `call`; quem chama esta função recebe o valor retornado pelo objeto. Já o envio assíncrono é mais um comando da CO^2. O seguinte código expressa o acesso a uma região crítica fazendo uso do semáforo binário `sb`.

```
eval (call sb "wait");
Região crítica
sb : "signal";
```

Na primeira linha envia-se a cadeia de caracteres '`wait`' para o objeto `sb`, e espera-se a resposta do objeto. Recebendo a resposta ingressa-se na região crítica. Saindo da região crítica envia-se a cadeia de caracteres '`signal`' para o objeto `sb` sem esperar-se nenhuma resposta.

O emissor pode enviar a quantidade de dados que quiser numa única mensagem. A seguinte classe implementa a fila de endereços de correio que ficou pendente do capítulo anterior.

```
class fila()
$1 = "enfileira" && TemDados = 0{
   set TemDados 1;
   Dados $2;
   }
$1 = "retira" && TemDados = 1{
   set TemDados 0;
   return Dados;
}
$1 = "esta-vazia?" && TemDados = 1{
   return 0;
}
$1 = "esta-vazia?" && TemDados = 0 {
   return 1;
}
end;
```

Para enfileirar um valor em uma fila `F` o comando é:

```
F: "enfileira" valor;
```

E para retirar o valor que está na cabeça da fila:

```
set cabeça (call F "retira");
```

6.2.4 Semântica dos objetos

Todos os valores que compõem uma mensagem são *encapsulados* junto com o *endereço de retorno*. *Desencapsular* uma mensagem é a ação de atribuir o primeiro valor que a compõe à variável $1, o segundo a $2 e assim por diante. O endereço de retorno é atribuído automaticamente à variável `caller`. Esta variável pode ser utilizada a vontade no corpo de uma classe. *Processar uma mensagem* significa desencapsula-la e avaliar os pares *condição-ação* na ordem em que foram especificados. Ao achar-se a primeira condição verdadeira, a correspondente ação é inteiramente executada e o processamento dessa mensagem retorna *finalizado*. Se não for achada nenhuma condição verdadeira o processamento dessa mensagem retorna *adiado*.

Todo objeto tem uma fila de mensagens onde são colocadas as mensagens enviadas para esse objeto na ordem que chegarem. Ao criar-se uma instância de uma classe inicializa-se a variável `self` apontando para a própria instância (ou seja, as mensagens que forem enviadas para `self` são colocadas no fim da própria fila de mensagens) e o *código de inicialização* é executado (se houver). Depois ingressa-se num ciclo infinito no qual a fila de mensagens é percorrida desde a cabeça até o fim, *processando* as mensagens. Se o processamento da mensagem resulta adiado, a mensagem é deixada no seu lugar, passando-se a processar a seguinte mensagem na fila. No caso que o processamento de uma mensagem retorne *finalizado*, ela é removida da fila; voltando-se ao início da fila para ver se a eventual mudança de estado, decorrente do processamento, habilitou o processamento de alguma mensagem adiada. Se o fim da fila é atingido, o objeto bloqueia até uma nova mensagem chegar.

O funcionamento da máquina virtual que executa um objeto é formalizado no algoritmo da figura 6-1. Esta máquina possui um apontador para uma mensagem na fila, ou o fim dela. A mensagem marcada por este apontador é chamada de *mensagem corrente*. A ação de pegar a mensagem corrente bloqueia a máquina se o apontador estiver no fim da fila. Note-se que a cada iteração processa-se a mensagem corrente; conforme foi dito este processamento retorna sempre *finalizado* ou *adiado*.

FIGURA 6-1 Semântica operacional de um objeto

```
self ← endereço da fila de entrada
Se tem código de inicialização
Executar código de inicialização
Colocar o apontador no começo da fila.
Para sempre
   Pegar a mensagem corrente (bloqueando se não tiver)
   Processar a mensagem
   Se o processamento for finalizado
      Remover a mensagem corrente da fila
      Colocar o apontador no começo da fila
   senão
      Avançar o apontador na fila
fim para sempre
```

6.2.5 Sincronização

Quando uma mensagem é enviada com a função `call` o emissor é bloqueado até receber uma resposta. Para receber esta resposta uma fila temporária, similar à fila de chegada, é criada. O endereço desta fila é o que foi chamado anteriormente de *endereço de retorno*, o qual é encapsulado junto com os dados que comportam a mensagem. Se a mensagem for enviada na modalidade assíncrona (com o operador ':'), o endereço de retorno tem um valor reservado chamado de *nulo*.

Há três formas pelas quais um objeto pode retornar um valor a quem lhe enviou uma mensagem. Duas delas são explícitas e uma implícita.

Uma das formas explícitas já apareceu no código da classe `fila`. Trata-se do comando `return`. O comando

```
return resposta;
```

é equivalente a

```
if  (caller != nulo) then
   caller : resposta;end;
```

A outra forma explícita é denominada de *delegação da resposta*. A delegação da resposta é um mecanismo que o programador possui para especificar um endereço de retorno quando do envio de uma mensagem na modalidade assíncrona. A delegação da resposta permite que um objeto chame a um terceiro objeto para processar uma requisição e ele ficar livre para atender novas mensagens.

Suponha-se ter o objeto `servidor`. Outros objetos conhecem ele e pedem-lhe requisições com a função `call`:

```
... (call servidor requisição ...) ...
```

O objeto `servidor`, depois de ter feito a parte do crítica do processamento (aquela que modifica seu estado) cria um objeto que faz o processamento restante da requisição (por exemplo, formatação da resposta), e delega a ele a resposta para o cliente apropriado. Desta forma o objeto consegue aumentar a vazão com que ele atende as requisições. Assim, o código da classe do objeto `servidor` incluiria as seguintes linhas:

```
set terceiro (new processa-o-resto);
terceiro : dados-da-mensagem -> caller;
```

A última linha de código envia *dados-da-mensagem* ao objeto `terceiro` na modalidade assíncrona, com o endereço de `caller` no endereço de retorno. Quando o objeto `terceiro` recebe esta mensagem, ele percebe que o endereço de retorno não é *nulo* e portanto responde da mesma forma que teria feito se a mensagem tivesse sido enviada com a função `call`.

O seguinte código ajuda a compreender este mecanismo. Tem-se uma classe `math` que implementa funções matemáticas; quando recebe uma requisição para calcular o fatorial ela delega a requisição a um terceiro objeto.

```
class factorial ()
$1 = "eval" {
# eval retorna $2 * o factorial de $3
   if ($3 = 1)
      return ($2 * $3);
   else
      self: "eval" ($2 * $3) ($3 - 1) -> caller;
}
end;

class math ()
$1 = "factorial" {
   set f (new factorial);
   f:"eval" 1 $2 -> caller;
...
}
end;
```

Quando se chega ao fim do processamento de uma mensagem sem ter-se executado nenhuma das duas formas explícitas e o endereço de retorno não é *nulo*, o valor *indefinido*[1] é retornado ao emissor. O conjunto, implementado no código da figura 6-2, mostra um exemplo completo contendo delegação da resposta.

FIGURA 6-2 Implementação de um conjunto

```
class conjunto ()
BEGIN  ( set vazio 1; )
$1 = "adicione" && vazio = 1 {
   set elemento  $2;
   set vazio 0;
   set resto (new conjunto);
}
$1 = "adicione" && vazio = 0 {
   resto :  $1 $2;
}
$1 = "pertence?" && vazio = 1 {
   return 0;
}
$1 = "pertence?" && vazio = 0 && elemento = $2{
   return 1;
}
$1 = "pertence?" && vazio = 0 {
   resto : $1 $2 -> caller;
}
end;
```

Em cada instância de uma classe a variável `self` é ligada ao endereço dessa instância. Delegando a resposta a si mesmo um objeto consegue suspender o tratamento de uma mensagem para processar outras, a exemplo da primitiva `wait` dos monitores.

Na figura 6-3 apresenta-se o código de uma classe que implementa um monitor que gerencia permissões compartilháveis e exclusivas. A política de atribuição da permissão garante num só tempo a consistência sem deixar ninguém morrer de fome [Hoare, 1974]. Toda requisição de permissão compartilhável é atendida desde que não exista ninguém esperando uma permissão exclusiva. Se chega uma requisição de permissão exclusiva enquanto existam clientes com uma permissão de qualquer tipo (compartilhável ou exclusiva), a requisição é colocada numa fila para permissões exclusivas e o atendimento é adiado. As requisições de permissão compartilhável que chegarem quando a fila de permissões exclusivas não estiver vazia são colocadas numa fila para permissões compartilháveis adiadas. Toda vez que uma permissão exclusiva for devolvida todas as requisições de permissão compartilhável que estiverem adiadas são atendidas. Quando é devolvida a última permissão compartilhável, a primeira requisição da fila de permissões exclusivas é atendida.

1. O valor indefinido é uma constante da CO^2 que aparece como resultado da avaliação de um nome sem ligação no ambiente, incompatibilidade de tipos (soma de um inteiro a um objeto, por exemplo) etc.

FIGURA 6-3 Implementação de um monitor em CO^2

```
class núcleo
BEGIN {set Habil = "Externo";}
$1 = "S" && Habil = "Externo"{
   if (XAgindo = 1 || XEsperando > 0)
      set SEsperando (SEsperando + 1);
      self : "EsperaS" -> caller;
   else
      set SAgindo (SAgindo + 1);
   end;
}
$1 = "X" && Habil = "Externo"{
   if (SAgindo > 0 || XAgindo > 0)
      set XEsperando (XEsperando + 1);
      self : "EsperaX" -> caller;
   else
      set XAgindo 1;
   end;
}
$1 = "L" && Habil = "Externo" && SAgindo > 0{
   set SAgindo (SAgindo - 1);
   if (SAgindo = 0 && XEsperando > 0)
      set Habil = "FilaX";
   end;
}
$1 = "L" && Habil = "Externo" && XAgindo = 1{
   set XAgindo 0;
   if (SEsperando > 0)
      set Habil = "FilaS";
   else if (XEsperando > 0)
      set Habil = "FilaX";
   end;
}
$1 = "EsperaS" && Habil = "FilaS"{
   set SAgindo (SAgindo + 1);
   set SEsperando (SEsperando - 1);
   if (SEsperando = 0)
      set Habil "Externo";
   end;
}
$1 = "EsperaX" && Habil = "FilaX"{
   set XAgindo 1;
   set XEsperando (XEsperando - 1);
   set Habil "Externo";
}
end;
```

Para pedir uma permissão exclusiva, a mensagem `"L"` é enviada ao monitor. A mensagem `"S"` pede uma permissão compartilhável, e a mensagem `"L"` devolve a

permissão (observe que não é necessário especificar que tipo de permissão está sendo devolvida). O código simula três filas (mensagens externas, exclusivas adiadas e compartilháveis adiadas) utilizando apenas a fila de entrada de mensagens do objeto que o implementa. Para isso as mensagens que, em tese, estão na fila de exclusivas adiadas são marcadas com "`EsperaX`", ao passo que as mensagens que estão esperando uma permissão compartilhável são marcadas com "`EsperaS`". Uma variável interna do monitor, denominada `Habil`, indica que mensagens devem ser consideradas (num nível maior de abstração: que fila ler). O enfileiramento de uma requisição compartilhável (ou exclusiva) é feito enviando a mensagem "`EsperaS`" (ou "`EsperaX`") ao próprio monitor (`self`).

A função `new` retorna assim que o endereço da fila de entrada do objeto for criado. O código de inicialização é executado em paralelo com o código do criador do objeto. Assim, pode-se colocar no código de inicialização um ciclo infinito que leia dados de uma porta de entrada e faça o processamento desses dados. Desta forma consegue-se atender várias portas com uma abordagem concorrente. A seguir apresenta-se o código de um servidor que tem três portas, uma para pedir permissões exclusivas, outra para pedir permissões compartilháveis e outra para liberar a permissão.

FIGURA 6-4 Servidor com concorrência interna

```
#!/usr/ahand/bin/co2 -r -i X -i S -i L

class HandReq (mon port req)
BEGIN {
   while (1)
      set client (get-mail-address port);
      eval (call mon req);
      client ! "vai";
   end;
}
end;

class HandLib (mon port)
BEGIN {
   while (1)
      eval(get-line port);
      mon : "L";
   end;
}
end;

set mon (new núcleo);
eval(new HandReq mon X "X");
eval (new HandReq mon S "S");
eval (new HandLib mon L);
```

6.3 Concorrência no acesso a arquivos

O problema já tinha sido antecipado no capítulo anterior: o interpretador da CO^2 não espera a finalização da execução de um comando para executar a linha de código seguinte, trazendo como consequência indesejável a concorrência no acesso aos arquivos. A questão é se a CO^2 deve fornecer mecanismos para lidar com essa concorrência ou se isto deve ser deixado nas mãos do programador. A decisão foi resolver isto dentro do espírito de uma arquitetura aberta, a exemplo de muitos sistemas bem sucedidos (Emacs, X-Windows, Lisp e o próprio Unix). Assim, manteve-se um núcleo da linguagem com acesso a arquivos básico (sem nenhum controle de concorrência) que possibilite a implementação de funções que controlem a concorrência.

Desta maneira distintos tipos de estratégias podem ser implementadas dependendo das características do sistema. Apresenta-se aqui uma abordagem baseada em permissões de leitura e escrita. No entanto, pode cogitar-se outras soluções, como a utilização de versões de arquivos.

Antes de ativar a execução de um comando, o interpretador avalia seqüencialmente todas as expressões envolvidas na linha que pede sua execução. Só depois de ter feito todas estas avaliações o comando pode ser executado. A ideia é escrever uma função que receba como único parâmetro um nome de um arquivo e que retorne um endereço de correio para escrever nesse arquivo. A única diferença entre esta função e a função primitiva '>' é que esta nova função só retorna quando não houver ninguém lendo nem escrevendo nesse arquivo.

Para cada nome de arquivo mantém-se um monitor. O relacionamento entre cada nome de arquivo e seu respectivo monitor é mantido numa estrutura de dados de nome `arquivo-monitor`.[1] Inicialmente esta estrutura está vazia, e vai sendo preenchida conforme os arquivos vão sendo utilizados. O monitor associado a um nome de arquivo obtém-se enviando a mensagem `procura-monitor` para o objeto `arquivo-monitor`.

A seguir apresenta-se a função `abre`. Esta função recebe um nome de arquivo, pede permissão para escrita ao seu monitor e retorna um endereço de correio para escrever no arquivo.

1. A implementação dessa estrutura de dados poderia ser feita utilizando-se de uma classe similar à classe conjunto vista neste capítulo. No entanto é desejável que a linguagem venha a suportar estruturas de dados mais eficientes para a implementação de tais estruturas.

```
class escritor (entrada destino monitor)
BEGIN {
   while (conectado entrada)
      destino ! (get-line entrada);
   end;
   monitor : "L";
}
end

fn pre-abre (arquivo-monitor)
   (lambda (arquivo)
      (let
         set m (call arquivo-monitor "procura-monitor"
                     arquivo);
         eval (call m "X");
         set entrada (create-port "x");
         eval (new escritor entrada (> arquivo) m);
       in
         entrada;
      )
   );

set abre (pre-abre arquivo-monitor);
```

Analogamente, precisa-se de uma função que peça permissão de leitura ao monitor correspondente, alimente com os dados do arquivo um endereço de correio e libere a permissão. Chamou-se esta função de `alimenta`.

```
fn prealimenta (arquivo-monitor)
   (lambda (arquivo destino)
      (let
         set m (call arquivo-monitor "procura-monitor"
                     arquivo);
         eval (call m "S");
         set entrada (create-port "x");
         eval (new escritor entrada destino m);
         cat arquivo out>entrada;
       in
         "ok";
      )
   );


set alimenta (prealimenta arquivo-monitor);
```

6.4 Discussão

Foi apresentado um modelo para suporte de concorrência interna fortemente inspirado no modelo de *actors*. No entanto algumas características de baixo nível de

às mensagens que ele mesmo enviou. O processamento das mensagens não pode ser adiado; se isto for necessário todas as ações envolvidas no adiamento devem ser especificadas explicitamente. Tem-se como resultado que um ator tem que lidar com uma única fila na qual misturam-se todo tipo de mensagens, incluindo respostas.

O ABCM/1 é um bom modelo de referência para a comparação do mecanismo de objetos da CO^2. A exemplo de ABCM/1 a aceitação de mensagens pode ser submetida a condições dependentes tanto do estado do objeto quanto ao conteúdo da mensagem; no entanto, em ABCM/1 as mensagens que não podem ser processadas são jogadas fora. Em compensação o processamento em um objeto de ABCM/1 pode ser suspendido à espera de uma condição; tal mecanismo não é primitivo em CO^2, tendo que ser simulado enviando uma requisição ao mesmo objeto. Os modos assíncrono e síncrono de envio de mensagens relembram os modos *past* e *now* do ABCM/1. O ABCL/1 ainda tem o modo *future* no qual o programador especifica a variável na qual quer que fique a resposta à mensagem enviada sem ter que esperar por tal resposta. Outros recursos da CO^2 existentes em ABCM/1 são o envio de mensagens alterando o endereço de resposta (delegação) e o suporte sintático para ter acesso ao endereço de resposta de uma requisição.

Pode-se dizer que o mecanismo de classes introduzido oferece recursos rudimentares de programação orientada a objetos. O primeiro valor de uma mensagem poderia ser visto como o seletor. A ligação com o método a ser executado no objeto receptor é claramente dinâmica (a própria sintaxe desnuda esta característica).

Percebe-se nos exemplos vistos que muitas vezes nomes aparecem sendo utilizados numa classe sem que tenham sido inicializados. A convenção é a mesma de *awk*: se o nome não for inicializado, ele vale 0 se ocorrer num contexto numérico, ou uma cadeia de caracteres vazia, se aparecer num contexto alfanumérico. Acredita-se que esta característica seja de grande valor para o desenvolvimento rápido de aplicativos.

Capítulo 7 Uma Implementação

A implementação de um protótipo das idéias expostas é apresentada neste capítulo. Cabe esclarecer que o objetivo desta implementação não foi fazer nenhum tipo de medição; mas se defrontar com os desafios da implementação de uma linguagem concorrente. Primeiro apresentam-se as ferramentas utilizadas, depois se discute o projeto do interpretador. Finalmente, em seção à parte, se apresenta a implementação dos mecanismos de concorrência interna.

7.1 Materiais utilizados

O protótipo foi implementado em uma arquitetura sun4 sob SunOS 4.1.1 Rev. B. A linguagem utilizada foi C++ da AT&T. Para comunicação entre processos usararam-se *sockets*. A concorrência interna foi implementada com as funções da biblioteca de *Lightweight Processes* da Sun Microsystems.

7.2 Estrutura do protótipo

O projeto é o resultado da combinação de conceitos da semântica denotacional com conceitos da programação orientada a objetos. A semântica denotacional é um formalismo para explicar o fato de que a execução de um programa opera sobre um estado;[1] a cada passo esse estado é modificado.

$$estado_1 \rightarrow estado_2 \rightarrow estado_3 \ldots$$

A passagem de um estado a outro se deve à execução dos comandos. A semântica dos comandos é expressada com a função **C**. Esta função expressa em termos funcionais como cada comando age sobre o estado. Ou seja, a equação

$\mathbf{C}\ (\gamma\ s_1) = s_2$ (onde γ é um comando e s_1 e s_2 são estados)

diz que, se o estado é s_1 e o comando γ é executado, então o estado passa a ser s_2

Embora a teoria seja mais rigorosa, para os fins deste trabalho basta considerar o estado como sendo um mapeamento de identificadores para valores, chamado ambiente. Os comandos então agem sobre o ambiente, criando novos mapeamentos entre identificadores e valores. Por vezes comandos envolvem expressões, e ao se executar um comando essas expressões devem ser avaliadas. A função semântica **E** expressa como se obtem um valor de uma expressão quando avaliada em um determinado estado. Assim, a equação

$\mathbf{E}\ (\varepsilon\ s) = v$ (onde ε é uma expressao, s é um estado e v um valor)

expressa o fato de que a avaliação da expressão ε no estado s produz o valor v.

Em suma, tem-se os seguintes domínios e funções:

Cmd, o domínio dos comandos.

Exp, o domínio das expressões.

Val, o domínio dos valores.

Ide, o domínio dos identificadores.

Env = Ide $\rightarrow$ **Val**, o domínio dos ambientes.

C: Cmd $\rightarrow$ **Env** $\rightarrow$ **Env**, a função semântica dos comandos.

E: Expr $\rightarrow$ **Env** $\rightarrow$ **Val**, a função semântica das expressões.

Definiram-se em C++ as classes `Cmd`, `Exp`, `Ide`, `Val` e `Env` para implementar os domínios homônimos. Naturalmente, na implementação em C++ muitos dos conceitos da semântica denotacional perdem seu sabor funcional. Em primeiro lugar ambientes não são funções, são objetos. A aplicação de um ambiente a um identificador é implementada mediante o método `ObterValor` da classe `Env`.

```
Val* Env::ObterValor(Ide);
```
[1]

A função semântica **C** é implementada pelo método `Exec` da classe `Cmd`.

```
void Cmd::Exec (Env&);
```

1. Para um tratamento mais rigoroso da semântica denotacional vide [Gordon, 1979; Allison, 1986].

Decidiu-se que o ambiente seja passado por referência ao invés de se retornar o novo ambiente como resultado do método.

A função semântica E foi implementada com o método `Eval` da classe `Exp`.

```
Val* Exp::Eval (Env);
```

`Exec` e `Eval` são o que em C++ se chama de funções virtuais, ou seja, funções redefinidas nas subclasses de `Cmd` e `Exp`, respectivamente. A rigor nunca existe um objeto que pertença exclusivamente à classe `Cmd`, ou a classe `Exp`, sempre são instâncias de alguma das subclasses. Exemplos de subclasses de `Cmd` são a `SetCmd` (que contem a árvore sintática do comando `set`), a `SeqCom` (bloco de comandos para serem executados sequencialmente) a `IfCmd`, a `WhileCmd` e a `ExtCmd` (contendo a árvore sintática do comando para executar um programa). Algumas das subclasses de `Exp` são `IdeExp` (uma expressão formada por apenas um identificador), `NumExp` (um número), `PlusExp` (uma soma), etc. O mecanismo de herança simplificou muito a implementação da semântica da CO[2] no código de *yacc++*, como mostra o esboço da figura 7-1. Nesta figura pode-se ver que, pela recursão da gramática, toda vez que um comando é reduzido no nível mais alto, um objeto de alguma subclasse de `Cmd` é recebido pelo código ativado quando da redução. Esse objeto contém uma representação do comando reduzido. Em seguida o método `Exec` do objeto é acionado passando o ambiente como parâmetro. Como o ambiente é passado por referência, ele eventualmente pode vir a ser alterado durante a execução do comando. O exemplo ainda inclui o código da classe `IfCmd` para mostrar o grau de simplicidade da implementação.

1. Esta sentença em C++ declara o nome `ObterValor` como sendo um método da classe `Env`, o qual espera um identificador como parâmetro e retorna um valor. `ObterValor` retorna um apontador a um valor em vez de um valor; isto se deve ao fato de que `Val` é uma superclasse. Este tipo de detalhes, que dizem mais respeito à C++ do que à programação orientada a objetos, não são considerados neste capítulo. Para mais detalhes sobre C++ vide Stroustroup, 1986.

FIGURA 7-1 Esboço do analisador sintático do protótipo

```
%{
Env TopLevel;
%}
%start prog
%union Cmd*   comm_p;
Exp*          exp_p;
Ide*          ide_p;
int           integer;
...
};
%type <comm_p> um_com set_com seq_com
%type <comm_p> if_com while_com ext_com
%type <l_comm_p> comms
%type <exp_p> uma_exp
%token <ide_p> IDENTIFICADOR
%token <integer> NUMERO
...
%%prog: prog ';' um_com
{  $3->Exec (TopLevel);
   delete $3;}
|          /* Vazio */
{}
;
um_com: set_com
{  $$ = $1; }
|       if_com
{  $$ = $1; }
|       while_com
{  $$ = $1; }
|       ext_com
{  $$ = $1; }
;
if_com: IF uma_exp THEN coms ELSE coms END
{  $$ = new IfCmd ($2, $4, $6);
   delete $2;
   delete $4;
   delete $6;}
|       IF uma_exp THEN coms END
;
coms: um_com
{  $$ = $1;}
|     coms ';' um_com
{  $$ = new SeqCom ($1, $2);
   delete $1;
   delete $2;}
;
uma_exp:IDENTIFICADOR
{  $$ = new IdeExp ($1);
```

```
   delete $1;}
|       NUMERO
{  $$ = new NumExp ($1);}
|       uma_exp '+' uma_exp
{  $$ = new PlusExp ($1, $3);
   delete $1;
   delete $3;}
...
;
...
%%
class IfCmd: public Cmd {
   Exp* Cond;
   Cmd* ThenCmd, ElseCmd;
public:
   IfCmd (Exp* ICond, Cmd* IThen, Cmd* IElse = 0) {
      Cond = ICond -> Clone ();
      ThenCmd = IThen -> Clone ();
      ElseCmd = IElse == 0? 0: IElse -> Clone ();
   }
   ~IfCmd () {
      delete Cond;
      delete ThenCmd;
      if (ElseCmd)
         delete ElseCmd;
   }
   void Exec (Env& E){
      if (Cond -> Eval (E) -> IsTrue ())
         ThenCmd -> Exec (E);
      else if (ElseCmd)
         ElseCmd -> Exec (E);
   }
}
```

A implementação da classe `ExtCmd` merece atenção à parte. A classe `ExtCmd` contém uma representação de um bloco definicional.[1] No limite ele representa um bloco definicional que executa apenas um programa. No código desta classe utiliza-se o tipo polimórfico[2] `lista`. Embora a C++ não suporte tipos polimórficos, um sucedâneo destes pôde ser implementado utilizando-se dos recursos do pré-processador da C++. A figura 7-2 mostra um esboço da implementação desta classe.

1. Veja seção 4.2, página 4-2.
2. Para aprofundar em tipos polimórficos veja [Cardelli e Wegner, 1975; Liskov et al, 1977].

FIGURA 7-2 Código em C++ da classe `ExtCmd`

```
struct InBind{
   String Porta, Nome;
};

struct OutBind{
   String Porta;
   Exp* Destino;
};

struct SingleCom{
   String NomeProg;
   lista (Exp*) Parametros;
   lista (InBind) InBinds;
   lista (OutBind) OutBinds;
};

class ExtCom{
   lista (SingleCom) Coms;
public:
ExtCom (lista (SingleCom) IComs): Coms (IComs) (;);
   Exec (Env& E){
      Para cada elemento x de Coms
         Avaliar as expressões em x.Parametros
               utilizando o ambiente E
         Iniciar execução de x.NomeProg
               passando como parametros o resultado
               da avaliação no passo anterior
         Para cada elemento i de x.InBinds
               p = endereço alocado à porta i.Porta
                  do comando recentemente executado
               E -> Ligar (i.Nome, p);
      Para cada elemento x de Coms
         Para cada elemento o de x.OutBinds
               d = x.Destino -> Eval (E);
               Conectar porta o.Nome a d
);
```

A implementação dos recursos funcionais esteve baseada em conceitos elementares de programação funcional.[1] A seguinte seção descreve a implementação dos recursos de concorrência interna.

1. Habitualmente os livros da linguagem LISP incluem uma implementação em LISP do próprio interpretador (veja [Abelson e Sussman, 1985]). Obviamente existem técnicas de otimização, no entanto a metodologia escolhida para o protótipo segue o modelo elementar.

7.3 Implementação da Concorrência Interna

A concorrência interna ficou encapsulada nas classes `Fila` e `Actor`. Essencialmente a classe `Fila` apresenta dois métodos: um para colocar uma mensagem no fim dela, e outro para retirar a mensagem que estiver na cabeça da mesma. Uma mensagem nada mais é do que uma lista de valores. A declaração de ambos os métodos é a seguinte:

```
lista (Val*) Fila::Retirar ();
void Fila::Colocar (lista (Val*));
```

Instâncias da classe `Fila` são os únicos objetos compartilhados por várias *threads* em todo o protótipo. Isto exigiu um entendimento do funcionamento da C++ para não cair em uma armadilha semântica. Em C++ objetos podem ser criados a partir de classes; esses objetos encapsulam sua representação de sorte que só é possível alterar ou consultar seu estado chamando alguma das suas funções-membro. Em um ambiente seqüencial o chamado a uma função-membro é semanticamente equivalente ao envio de uma mensagem a um objeto; no entanto, que implicações decorrem da utilização de C++ em um ambiente *multithread*? Colocado em outros termos, o que acontece se uma mensagem é enviada a um objeto por uma *thread* enquanto ele está processando uma mensagem enviada por uma outra *thread*? Em C++ envios de mensagens são traduzidas para chamadas a procedimentos, nos quais o próprio objeto é mais um parâmetro. Assim, na situação colocada ter-se-ia o código de dois procedimentos agindo concorrentemente sobre uma mesma estrutura de dados. No caso da classe `Fila` distinguiu-se uma região crítica que foi protegida com um monitor. Cabe esclarecer que se o método `Puxar` for acionado quando o objeto estiver vazio, a resposta é adiada até que uma nova mensagem for adicionada mediante o método `Colocar`.

Em conjunto, o funcionamento segue o seguinte esquema. O tipo abstrato de dados `Classe` contém uma representação de uma classe de CO[2]. São membros de `Classe` uma lista de identificadores que representa as variavéis livres (aquelas que são inicializadas quando da instanciação de um objeto) e uma lista de pares condição-ação.

```
struct CondAcao{
Exp* Cond;
lista(Cmd*) Acao;
}
class Classe: public Val{
   friend class Actor;
   lista (Ide) VarLivres;
   lista (CondAcao) Comport;
}
```

`Classe` é herdeira de `Val` para que ela possa ser ligada a seu nome no ambiente, quando uma declaração de classe é reduzida.

```
dec_classe:
CLASS IDENTIFICADOR codigo_classe
{  TopLevel.Adicionar ($2, $3);
   // $3 retorna um Classe*
}
;
```

A classe `Actor` tem três componentes: uma fila de entrada de mensagens, um ambiente privativo da instância (objeto de tipo `Env`) e uma lista de pares condição-ação.[1] Cada par condição-ação está formado por uma expressão (a condição) e uma lista de comandos (a ação). Basicamente, a classe `Actor` define os métodos `ObterFila` e `Começar`, além do construtor. O primeiro só retorna um apontador para a fila de entrada de mensagens do actor. O método `Começar` inicializa uma *thread* com o código do procedimento que implementa o algoritmo da figura 6-1, página 6-8, de forma que todas as alterações ao estado do *actor* fiquem registradas no ambiente privativo. No entanto, nesse algoritmo há uma fila que suporta uma quantidade maior de operações do que as da classe `Fila` apresentada nesta seção. Desta forma, julgou-se conveniente manter a classe `Fila` como um canal de comunicação entre *threads*, complementando-a com uma lista local a cada *thread* para emular o comportamento da fila de mensagens da figura 6-1. A figura 7-3 apresenta o código da classe `Actor`; ali pode-se ver algumas das operações suportadas pelo tipo polimórfico `lista`[2].

1. Vide seção 6.2, página 6-4.

2. A lista encapsula, entre outras coisas, um apontador para o elemento atual. O apontador pode estar no fim da lista, ou não; no último caso o método `Atual` retorna o elemento atual, no primeiro caso o método `NoFim` retorna verdadeiro e uma chamada a `Atual` é um erro. O operador sobrecarregado ++ faz avançar o apontador ao próximo elemento da lista, o operador + concatena duas listas. Para colocar o apontador no começo da lista utiliza-se o método `Rebobinar` e para remover o elemento atual o método `Retirar` deve ser usado, neste caso o apontador avança para o seguinte elemento da lista

FIGURA 7-3 Implementação da semântica de objetos

```
enum Lugar ADIADAS, FILA;
enum Estado FINALIZADO, ADIADO;
int MaquinaActor
   (Fila* AFila, Env* Amb, lista (CondAcao)* Comport){
   lista (lista (Val*)) Adiadas;
   lista (Val*) Atual;
   if (tem código de inicialização em Comport)
      processar código de inicialização com Amb;
   Lugar Processando = FILA;
   for (;;){
      // Pegar proxima mensagem
      if (Processando == ADIADAS)
         if (Adiadas -> NoFim ())
               Processando = FILA;
         else
            Atual = Adiadas - Atual ();
      if (Processando == FILA)
         Atual = AFila -> Retirar ();
      Desencapsular (Atual, Amb);
      // Processar mensagem atual
      if (Processar (Atual, Amb) == FINALIZADO){
         // Remover a mensagem corrente e
         // voltar ao inicio da fila
         if (Processando == ADIADAS)
               Adiadas -> Retirar ();
         Processando = ADIADAS;
         Adiadas -> Rebobinar ();
      }
      else // Processamento ADIADO
         // avancar o apontador na fila
         if (Processando == ADIADAS)
               Adiadas++;
         else
               Adiadas += Atual;
   }
}

   class Actor{
   Fila* AFila;
   Env* Ambiente;
   lista (CondAcao)* Comport;
public:
   Actor (Classe *C, lista (Val*) Iniciais){
      Ambiente = new Env(C -> VarLivres, Iniciais);
      AFila = new AFila;
      Comport = new lista (CondAcao) (C -> Comport);
      Ambiente -> Ligar ("self", new ActorVal (AFila));
   };
```

```
    Fila *ObterFila ()  { return AFila;};
    void Comecar (){
         Iniciar um thread com o código da função
              MaquinaActor com parâmetros AFila,
              Ambiente e Comport
    };
};
```

Uma subclasse de `Exp`, chamada de `NewExp`, é utilizada para representar a expressão new da CO[2]; quando o `Eval` de uma instância desta classe for acionado, um apontador a um objeto de tipo `ActorVal` é retornado. A classe `ActorVal` é uma subclasse de `Val` e representa uma instância de um *actor*.

```
class NewExp: public Exp{
     Ide Classe;
     lista (Exp)Inicializadores;
public:
     Val* Eval (Environment E){
        Val* X = E.ObterValor (Classe)
        if (X de tipo Classe){
           lista (Val*) L = resultado de aplicar Eval (E) a
                  todasas expressões de
                  Inicializadores;
           Actor* A = new Actor (X, L);
           A -> Começar ();
           return new ActorVal (A -> ObterFila ());
        };
     };
};
```

O método mais importante da classe `ActorVal` é `Enviar`, utilizado para enviar uma mensagem a um *actor*.

```
class ActorVal: public Val{
     Fila* AFila;
public:
     ActorVal (Fila* F){
        AFila = F;
     };
     Val* Enviar (lista (Val) L){
        AFila -> Colocar (L);
     };
};
```

Com isto ficam apresentadas as partes mais importantes da implementação do protótipo. Na seguinte seção são apresentadas as conclusões obtidas desta experiência.

7.4 Discussão

Como se disse no início do capítulo, o objetivo da implementação foi ganhar experiência na implementação de uma linguagem distribuída. Nas seções anteriores foram apresentadas as ferramentas utilizadas e os pontos mais importantes da implementação, e nesta seção se incluem os comentários sobre a experiência.

A depuração de um programa concorrente é trabalhosa devido às *race conditions*, as quais às vezes dificultam a repetição de um erro. Para complicar mais a situação, os depuradores carecem de utilidade para estes casos. É por isto que se fez questão de não escrever uma linha de código até não ter bem claros os conceitos envolvidos e definir uma boa decomposição do problema. Assim, o tempo investido na fase de estudo e projeto do sistema acabou sendo o mesmo que levou a fase de implementação. A primeira fase envolveu estudo de semânticas de linguagens de programação e das técnicas de projeto de sistemas de software, bem como o estudo da biblioteca de *lightweight processes*. A segunda fase compreendeu a codificação, teste e depuração do protótipo. Ao todo, a implementação tem sete mil e quinhentas linhas de código. Se se considera o porte do problema, esta quantidade está abaixo das expectativas; acredita-se que a utilização exaustiva de conceitos como herança e tipos polimórficos tenha contribuído muito na minimização das linhas escritas.

Dentre as pendências que restaram da implementação não se pode deixar de nomear a falta de destrutores para os objetos da classe `Actor`.

Em suma, a implementação não demanda um esforço de programação considerável; analisado em termos de custos, uma eventual implementação da CO^2 levaria a maior parte do investimento em capacitação, definição e *design*.

Capítulo 8 Conclusões

No decorrer deste trabalho foi definindo-se um modelo de concorrência visando sua aplicabilidade no estabelecimento de uma linguagem de comandos distribuída para o projeto A_Hand. Cada capítulo foi encerrado com uma discussão dos temas tratados; neste capítulo se faz uma conclusão global do trabalho. Na primeira parte discutem-se as contribuições das idéias expostas e, finalmente, na segunda se apresentam as possíveis formas de continuar o trabalho.

8.1 Contribuições

A contribuição do trabalho, abstraída em poucas palavras, é ter apresentado um estudo das linguagens de comandos como sendo linguagens de programação concorrente.

Especializando, a contribuição pode ser dividida nos seguintes pontos:

- Individualização das formas de interação entre processos que devem ser suportadas por uma linguagem de comandos com o intuito de alargar o seu domínio de aplicabilidade.
- Definição de uma notação homogênea para expressar as formas de interação entre processos individualizadas no ítem anterior; esta notação é adequada para uma implementação interpretada e interativa com ligação dinâmica dos processos.
- Distinção da necessidade de recursos definicionais para expressar com maior naturalidade alguns casos de interação entre processos e definição de recursos sintáticos para permitir a expressão destes casos.

- Caracterização da programação com filtros como sendo essencialmente funcional e estabelecimento de recursos funcionais para expressar esta situação.
- Definição de mecanismos para permitir o aninhamento de programas em vários níveis de abstração.
- Caracterização da necessidade de mecanismos de concorrência interna para facilitar a escrita de servidores, e definição de facilidades sintáticas para a expressão de concorrência dentro de um processo, visando uma implementação interpretada.

Cabe questionar a utilidade destes ítems. No que resta desta seção, se coloca à discussão a necessidade das extensões propostas ao modelo de concorrência da *shell*. Para isto, as razões do sucesso da *shell* como linguagem para desenvolvimento rápido de programas (interatividade, ligação dinâmica e execução de arquivos de comandos com equivalência sintática e semântica à execução de outros comandos) são consideradas uma premissa; o que se coloca em discussão é a validez das extensões sem relaxar esta premissa.

Construção de programas com filtros com fluxo multidimensional de informação. Dois exemplos com uma razoável dose de realismo foram colocados no trabalho; além disto, a quantidade de arquivos de comandos que geram arquivos temporários indicam que uma facilidade deste tipo não seria menosprezada por um programador.

Construção de programas com clientres e servidores. O mercado é hoje testemunha do crescimento de sistemas que de alguma forma ou outra utilizam o modelo cliente-servidor. No entanto, isto é feito no contexto de sistemas abertos; isto é, os clientes e servidores envolvidos não respondem a um único programa; pelo contrário os clientes são executados supondo que o servidor está disponível como se fosse mais um recurso computacional (memória, dispositivos, etc.) utilizado por ele. Nestes casos, como já se disse no capítulo 1, a ligação é estática. A questão neste parágrafo é se faz sentido estender o modelo de ligação dinâmica a clientes e servidores. O exemplo de teleconferência apresentado na página 1-8 não é evidência suficiente da necessidade deste recurso, pois pode-se argumentar que um servidor centralizado, bem conhecido pelos clientes, conseguiria gerenciar muitas sessões de teleconferência simultâneamente.

Em contrapartida, em defesa da ligação dinâmica pode-se sustentar que a ligação dinâmica se desempenharia melhor no caso de teste de uma nova versão do servidor, sem alterar nem recompilar, o cliente. Finalmente, não se pode deixar de reconhecer a tendência existente, no bojo da orientação a objetos, de que os arquivos tenham um tipo; desta forma, aconteceria com os arquivos uma evolução similar à experimentada com os tipos de dados: arquivos não seriam mais simples repositórios de informação prestes a ser utilizados por qualquer programa e passariam a ser um maço inseparável de dados e código. Ou seja, verdadeiros objetos que se ligariam dinamicamente a outros conforme às necessidades do programador.

Mecanismos de concorrência interna. Enquanto não houver concorrência interna o paralelismo decorre apenas da execução simultânea de um número de processos seqüenciais; sendo que a comunicação entre esses processos se faz utilizando mecanismos bem conhecidos (como chamadas a procedimentos, ou operações de entrada e saída), a concorrência é em certo ponto transparente para o programador. Os mecanismos de concorrência interna colocam o programador inevitavelmente frente à concorrência. Talvez os casos de concorrência interna fiquem confinados ao software básico, e a imensa maioria dos aplicativos distribuídos dispense recursos de concorrência interna. Entretanto, as últimas gerações de sistemas operacionais estão sendo *multithreaded*; acredita-se que a disponibilidade de uma linguagem interativa com concorrência interna é uma grande ajuda para o programador se familiarizar com estes novos conceitos, uma vez que a interatividade agiliza a iteração codificação-teste.

Por último, repete-se o que fora dito na seção 6.3, página 6-11: as classes concorrentes da CO^2 possibilitam uma arquitetura do interpretador no qual exista um núcleo, com acesso primitivo a arquivos, de maneira que o ambiente possa ser *customizado* para ter formas mais complexas de acesso a arquivos.

8.2 Trabalho futuro

A definição do modelo de concorrência deve evoluir para incluir mecanismos de tolerância a falhas. Ao mesmo tempo a CO^2 tem que se aproximar aos padrões de processamento distribuído.

Este trabalho se completa com a definição da CO^2; esta definição deve incluir recursos mais orientados à comodidade do usuário, como expansão de metacaracteres e reutilização de comandos digitados previamente. Este trabalho deve integrar a CO^2 com os outros projetos em andamento do A_Hand. No curto prazo a linguagem Cm distribuída deve estar pronta e a CO2 pode desempenhar um papel importante na fase de teste. Quanto à LegoShell, a CO^2 pode ser o código objeto num protótipo desta linguagem; porém outras formas de interação entre ambas as linguagens podem ser imaginadas: : em uma sessão de LegoShell o programador deveria poder abrir caixas e inserir código de CO^2. Isto permitiria programar utilizando as facilidades gráficas quando for possível, e só escrever código quando for necessário.

Apêndice A

Bibliografia

Agha, G. (1986), *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press series in Artificial Intelligence.

Andrews, G. R. e Schneider, F. (1983) "Concepts and Notations for Concurrent Programming", *ACM Computing Surveys*, Vol. 15, No. 1, 1983.

Andrews, G. R. (mar. 1991), "Paradigms for Process Interaction in Distributed Programs", *ACM Computing Surveys*, Vol. 23, Num. 1, pp.49-90.

Bal, H. (1990), *Programming Distributed Systems*, Silicon Press - Prentice Hall.

Bentley, J. (1986), *Programming Pearls*, Addison-Wesley Publishing Company.

Birrell, A. e Nelson, R. "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2 No. 1, 1984.

Brinch Hansen, P. (1978) "Distributed Processes: A Concurrent Programming Concept", *Comm. ACM*, Vol. 21, No. 11, 1978.

Di Cianni, C. (1994) *OMNI - Sistema de Suporte a Aplicações Distribuídas*. Tese de Mestrado, DCC - IMECC-UNICAMP, 1994.

Drummond, R. e Liesenberg, H. (1987), "A_HAND Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados", em *IV Encontro de Trabalho do Projeto ETHOS.*

Drummond, R. (1989) "LegoShell Linguagem de Computações", *III Simpósio Brasileiro de Engenharia de Software*, Recife, 1989.

Drummond, R. e Di Cianni, C. (1992)" OMNI - Sistema de suporte a aplicações distribuídas". *Anais do VI Simpósio Brasileiro de Engenharia de Software*, 1992.

Gabriel,R.P. (Mar. 1989), "Draft Report on Requirementes for a Common Prototyping System", *ACM SIGPLAN*, Vol. 24, Num. 3, pp. 93-165.

Gonçalves, C. (1994) *Objetos Distribuídos*, Tese de Mestrado, DCC-IMECC-UNICAMP, 1994.

Haeberer, A. M., Veloso, P. A. S. e Baum, G. (1988), *Formalización del Proceso de Desarrollo de Software*, Ed. Kapelusz-EBAI.

Hoare, C. A. R. (Out. 1974), "Monitors: an Operating System Structuring Concept", *Comm. ACM*, Vol. 17, Num. 10, pp. 549-557.

Hoare, C. A. R. (Ago. 1978), "Communicating Sequential Processes", *Comm. ACM*, Vol. 21, Num. 8, pp. 666-677.

Kernighan, B. e Pike, R. *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, 1984.

Kramer, J., Magee, J. Sloman, M. e Dulay, N. (1992) "Configuring Object Based Distributed Programs in REX", *IEEE Transactions on Software Engineer.*

Lamport, L. (Jul. 1978), "Time, Clocks, and the Ordering of Events in a Distributed System", *Comm. ACM*, Vol. 21, Num. 7, pp. 558-565.

Luqi (Maio, 1989), "Software Evolution through Rapid Prototyping", *IEEE Computer*, Vol. 22, Num. 5, pp. 13-25.

Piñón Arias, H. (1990), *Editor Topológico para a Linguagem de Especificação de Computações LegoShell*, Tese de Mestrado, DCC-IMECC-UNICAMP, 1990.

Ritchie, D. e Thompson, K. "The UNIX timesharing system", *Comm. ACM*, Vol. 17, No. 7, julho, 1974.

SUN Microsystems (1990) *Network Programming*, Sun Microsystems, Mountain View, 1990.

Teles, A. (1993), *A Linguagem de Programação Cm (versão 2.0x)*, Tese de Mestrado, DCC-IMECC-UNICAMP, 1993.

Wegner, P. (1990) "Concepts and Paradigms of Object Oriented Programming", *OOPS Messenger*, Vol. 1 No. 1, 1990.

Wegner, P. (1989), "Capital Intensive Software Technology", em *Software Reusability*, ed. Ted J. Biggerstaff e Alan J. Perlis, ACM Press, pp. 43-97.

Zave, P. (Fev. 1984), "The Operational versus the Conventional approach to software development", *Comm. ACM*, Vol. 27, Num. 2, pp. 104-118.