Universidade Federal de Santa Catarina

Programa de Pós-Graduação em Engenharia de Produção

Márcio Quintaes Marchini

LindaTalk: Suporte Distribuído à Programação Concorrente Orientada a Objetos

Dissertação submetida à Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Engenharia

Florianópolis

1994

LindaTalk: Suporte Distribuído à Programação Concorrente Orientada a Objetos

Márcio Quintaes Marchini

Esta dissertação foi julgada adequada para obtenção do Título de Mestre em Engenharia, Especialidade em Engenharia de Produção, e aprovada em sua forma final pelo programa de Pós-Graduação.

Osmar Possamai, Dr.

Coordenador do Curso

Banca Examinadora:

Luiz Fernando Jacintho Maia, Dr.

Orientador

Jean-Marie Farines, Dr.

Fernando Álvaro Ostuni Gauthier, M. Eng.

à memória de Roberto Quintaes Marchini

Agradecimentos

Este trabalho não pode ser considerado de forma isolada, com um início e fim durante o Programa de Mestrado. Embora ele se apresente aqui sob esta forma, ele é resultado de esforço e apoio de diversas pessoas ao longo dos últimos anos, e a essas pessoas é que desejo aqui agradecer, mantendo uma ordem cronológica.

A primeira delas é Plínio Marchini, amigo e pai cujo apoio constante me permitiu chegar até este ponto. Sem sua amizade e apoio ilimitado, este trabalho não poderia ter sido realizado.

Outra figura importante é o professor e amigo Luiz Fernando Bier Melgarejo, cujo convite para integrar o grupo de pesquisa no Laboratório de Software Educacional (EDUGRAF) da UFSC me permitiu travar contato com tecnologias na época desconhecidas, como programação orientada a objetos e Smalltalk. Através do EDUGRAF, um ambiente que propicia o desenvolvimento individual de cada um através de muita liberdade e cooperação, pudemos desenvolver nosso trabalho. O agradecimento, na verdade, vai a todos aqueles que um dia integraram o EDUGRAF e colaboraram para seu funcionamento como ele é, bem como pelas valiosas sugestões de todos neste trabalho.

Ao meu orientador, Luiz Fernando Jacintho Maia, pela compreensão com a minha maneira própria de conduzir meus trabalhos, e seu total apoio.

Finalmente, a todos os amigos que passam pelas nossas vidas e deixam dentro de nós um pouquinho de si. Obrigado a todos!

Sumário

Relação de Figuras

Relação de Tabelas

Relação de Listagens

Resumo

Problemas complexos são geralmente decompostos em subproblemas menores, que sejam tratáveis mais facilmente. O mesmo vale para sistemas de computação, os quais contam com uma gama rica de abordagens de decomposição (funcional, procedural, etc). Dentre estas, a decomposição orientada a objetos tem ganho cada vez mais espaço, dada sua riqueza e poder na modelagem e implementação de sistemas informáticos.

A possibilidade de programar sistemas multiprocessadores e sistemas em redes de computadores, por outro lado, favoreceu as linhas de programação paralela/concorrente/distribuída.

Contudo, se de um lado a orientação a objetos clássica promove uma modelagem natural de entidades no domínio do problema, por outro lado ela falha na tentativa de expressar atividades concorrentes/paralelas. Já sistemas que suportam a noção de processos paralelos, tais como Occam, Conic, Ada, etc, permitem preencher esta lacuna. Contudo, o poder de modelagem e abstração de entidades fica bastante limitado neste tipo de abordagem, levando geralmente à produção de sistemas dificeis de adaptar, manter e reusar.

Modelos com suporte à programação paralela orientada a objetos, tais como Emerald, ConcurrentSmalltalk, Act-1, ABCL/1, etc surgem na tentativa de unificar objetos no sentido clássico de orientação a objetos com a noção de processos paralelos e comunicantes. Porém, tanto nesta abordagem quanto na programação orientada a objetos clássica e alguns modelos de programação concorrente/paralela/distribuída, a metáfora de interação entre objetos/processos é a mesma: troca de mensagens.

Troca de mensagens conforme presente em sistemas concorrentes orientados a objetos apresentam diversas fraquezas no que toca a implementação, manutenção e reusabilidade de sistemas distribuídos. Nossa proposta busca incorporar a uma linguagem orientada a objetos clássica - Smalltalk - um modelo que suporte a programação paralela/distribuída com um maior grau de flexibilidade. Este modelo é o de Espaço de Tuplas, de Linda. Através de um pequeno conjunto de primitivas, tem-se um modelo simples de criação e coordenação de processos ortogonal à linguagem em que se insere o modelo (Smalltalk, no caso). Através do uso extensivo do modelo, acreditamos ser possível a construção de sistemas realmente distribuídos e orientados a objetos com um maior grau de flexibilidade em sua implementação, reusabilidade e manutenção.

Palavras-Chave: Linda, Smalltalk, Programação Concorrente Orientada a Objetos

Abstract

Complex problems are often decomposed into smaller problems which can be treated more easily. The same is valid for computing systems, which have a large set of decomposition methodologies (functional, procedural, etc). Among these, object-oriented decomposition (and programming) has been used more widely along the past years due to its richness and power to model problems and implement computing systems.

The availability of multiprocessing and networked systems, on the other hand, promoted the development of another row of programming languages/paradigms: the parallel/concurrent/distributed approach.

Although object-oriented programming easily maps entities from the problem domain, it fails when it attempts to express parallel/concurrent activities. On the other hand, parallel/concurrent paradigms like Occam, Conic, Ada, etc easily express parallel activities, but lack the power to model entities from the problem domain, making it difficult to produce, maintain, adapt and reuse computing systems.

Systems which support the object-oriented concurrent programming paradigm, like Emerald, ConcurrentSmalltalk, Act-1, ABCL/1, etc try to combine the classic notion of objects from object-oriented languages to processes in concurrent languages. However, in all of these approaches the communication metaphor is the same: message passing.

Message-passing presents weaknesses concerning the implementation and reuse of distributed systems. Our work, at the other side, adds a flexible parallel/distributed paradigm to a classic object-oriented language (Smalltalk). Here, we combine the Linda Tuple Space model to Smalltalk, adding a new dimension in modeling and programming distributed systems in commercially-available Smalltalk implementations.

Keywords: Linda, Smalltalk, Object-Oriented Concurrent Programming

1 - Introdução

Informatizar atividades através do uso de computadores já é comum em diversos campos da vida humana. Sua utilização é uma realidade, e a demanda parece crescer em taxas mais elevadas do que nossa capacidade de suprir a área com sistemas informatizados.

Buscando sempre aumentar a velocidade e qualidade com que sistemas informatizados são produzidos, inúmeras pesquisas vêm sendo feitas. Na área de *software*, diversas linhas de pesquisa têm buscado prover modelos computacionais naturais à modelagem de situações reais. Dentre elas, destacamos as técnicas de análise de sistemas, bancos de dados, inteligência artificial, especificação formal e linguagens de programação. Procurando buscar formas de diminuir o chamado *fosso semântico* [Takahashi88] entre a realidade como ela é [1] e como ela é modelada/programada em computadores, estas diversas áreas da informática tentam encontrar modelos mais naturais e produtivos no ciclo de desenvolvimento de *software*.

Várias destas áreas, contudo, cruzaram-se em uma área conhecida como orientação a objetos. Do lado da inteligência artificial, redes semânticas [Rich88] e *frames* [Fikes85] são "primos" conhecidos da orientação a objetos. Em bancos de dados, por outro lado, notou-se que o difundido modelo relacional não se mostrava adequado para aplicações não-convencionais [Kent79]. Propostas como GemStone [Maier86] buscaram incorporar um modelo semântico mais rico aos bancos de dados [Peckam88], dando origem aos bancos de dados orientados a objetos. Modelos mais ambiciosos tentam unificar modelos de inteligência artificial com bancos de dados, propondo os chamados bancos de conhecimento [Mattos91].

As linguagens de programação, entretanto, foram as ferramentas iniciais onde as primeiras idéias puderam ser testadas, suas limitações pesquisadas e novos modelos propostos. Ainda hoje é garantido o nicho de aplicações onde não se chega a utilizar modelos com riqueza semântica tão grande quanto os modelos da inteligência artificial, nem a necessidade de persistência provida pelos bancos de dados/conhecimento. É aqui que se encaixam ainda hoje as linguagens de programação, herdando diversas características de modelos alternativos.

1.1 - Objetivo do Nosso Trabalho

Neste trabalho, procuramos propor um modelo a ser incorporado ao modelo de objetos, buscando aumentar seu poder de expressão. Mais especificamente, aumentar o poder de linguagens orientadas a objetos[2] quanto à capacidade de expressar, sincronizar e comunicar atividades paralelas. Nossa opção aqui foi por um modelo ortogonal a implementações comerciais existentes de Smalltalk, de forma que nosso trabalho aqui apresentado possa ser utilizado efetivamente em modelagem/desenvolvimento de aplicações reais[3] em Smalltalk.

Optamos aqui por uma abordagem algorítmica onde o paralelismo é descrito explicitamente pelo programador. Uma linha alternativa à adotada aqui é a de que programas devem ser descritos pelo programador e paralelizados por ferramentas como compiladores/geradores de

[1] Não entraremos aqui em discussões filosóficas sobre a subjetividade ou não da realidade. Nosso sentido aqui é como ela é percebida pelo observador.

[2] A ênfase é Smalltalk [Xerox81] [BYTE81] [LaLonde90].

[3] A utulização da programação concorrente orientada a objetos beneficiaria, acreditamos, qualquer campo onde se utilize o computador na solução de problemas. No caso específico da Engenharia de Produção, por exemplo, áreas beneficiadas seriam: programação linear, programação não-linear, algorítmos genéticos, redes neuronais, etc.

código. Esta opção é interessante por permitir que velhos programas, escritos de forma seqüencial, possam ser trazidos para novas plataformas nos quais podem ser paralelizados. Por outro lado, acreditamos ser mais natural pensar na modelagem explicitando o paralelismo do que escondendo-o, como ressalta Papert em [Papert80]. O fenômeno do paralelismo é tão comum e natural que ambientes deveriam favorecer sua representação explícita, e não condená-la. Neste sentido, seguimos a linha de [Carriero89a].

Começamos, então, com um breve levantamento dos principais conceitos envolvidos na programação orientada a objetos, relacionando-os estreitamente com Smalltalk, devido ao seu pioneirismo, elegância e crescente aceitação. Logo em seguida, um breve levantamento de ambientes/linguagens com suporte para a programação paralela, buscando identificar as semelhanças de tais sistemas/ferramentas na construção de sistemas onde a expressão de atividades paralelas se faz necessária. Segue-se daí um estudo de alguns sistemas que procuraram unificar a orientação a objetos no sentido clássico, conforme será visto no início deste trabalho, com a programação paralela, levando a propostas conhecidas como linguagens concorrentes orientadas a objetos. Em seguida apresentamos Linda, um modelo alternativo, tido como *ad hoc* por alguns, que não adota a metáfora de trocas de mensagens entre atividades paralelas, e sim a idéia de um Espaço de Tuplas. Finalmente, terminamos este trabalho com nossa proposta, que busca prover mecanismos de expressão de atividades paralelas no ambiente Smalltalk, procurando aumentar seu poder de expressão, apresentando projetos onde este nosso trabalho aqui descrito têm sido utilizado com êxito.

2 - Programação Orientada a Objetos

Linguagens com suporte à programação orientada a objetos já são, há tempos, uma realidade de mercado. Algumas destas são simplesmente novas versões de linguagens anteriores, onde se incorporam as idéias de objetos. São exemplos C++ [Stroustrup86] , Modula-3 [Harbinson92], etc. Outras delas, entretanto, foram projetadas para suportar o paradigma de objetos, não tendo compromisso com nenhuma versão/linguagem anterior. São exemplos Smalltalk, Eiffel [Meyer88], etc.

Como consequência da diversidade, não há ainda consenso sobre que características mínimas uma linguagem deste tipo deve possuir. Algumas são fortemente tipadas, outras não; algumas suportam herança simples, outras herança múltipla, e assim por diante.

Os diversos conceitos relativos à orientação a objetos surgem explicitamente em Smalltalk, e é por isso que a escolhemos para, aqui, falar sobre programação orientada a objetos.

2.1 - Smalltalk

A origem de Smalltalk confunde-se com a origem do próprio computador pessoal e das interfaces gráficas (GUIs). Hoje disponível em diversas plataformas de hardware, Smalltalk teve sua origem no Palo Alto Research Center (PARC) da Xerox, na década de 70. Smalltalk foi concebido como "*...o componente de software do sistema pessoal de computação idealizado por Alan Kay, o Dynabook*" [Goldberg81]. Tal sistema pessoal seria portátil, de alta performance, tela de altíssima definição e multimídia. Para que os usuários finais fossem capazes de tanto absorver como produzir material no Dynabook, era necessário uma linguagem de alto nível e grande poder de expressão. Esta linguagem seria Smalltalk, tornando o Dynabook um meio de comunicação [Kay90].

Smalltalk teve várias versões intermediárias (Smalltalk-72, Smalltalk-74, etc), mas foi apresentado ao grande público como Smalltalk-80, em [BYTE81]. Hoje, o sistema é comercializado pela subsidiária ParcPlace, sob nomes de ObjectWorks\Smalltalk e VisualWorks. Outras versões existem, ligeiramente diferentes, por empresas como Digitalk [Digitalk-Win] [Digitalk-286], OTI [OTI-ENVY], etc. Versões de domínio público também existem, como GNU Smalltalk [FAQb], Little Smalltalk [FAQb], etc.

Hoje Smalltalk já é sucesso de mercado, sendo utilizado desde em sistemas de controle de processos [OTI-ENVY] até aplicações financeiras [Parcplace] [Business93]. Esse sucesso se deve à grande potencialidade oferecida pelo sistema, que apresentaremos a seguir.

2.1.1 - O Ambiente

Ao contrário de linguagens de programação como Pascal, Smalltalk[4] é um sistema integrado [Tesler81]. Engloba uma interface gráfica manipulável através de um dispositivo apontador como *mouse* e um conjunto de ferramentas integradas no ambiente. O compilador da linguagem de programação em si é apenas um dos componentes. Assim, ao contrário de ambientes convencionais com o famoso ciclo "edita-compila-*linka*-executa" , Smalltalk

[4] Características apresentadas aqui são do projeto inicial. Posteriormente modificações foram incorporadas, como utilização de interfaces (GUIs) já existentes, etc

apresenta todas as ferramentas num ambiente só. O editor de textos/programas, o compilador/gerador de código e a própria máquina virtual. Esta última sendo necessária, pois Smalltalk gera um código intermediário, os chamados *bytecodes* [Krasner81]. Isso facilita a portabilidade do sistema através de múltiplas plataformas (como existentes hoje), sem necessariamente comprometer a performance: o código intermediário é expandido para o código binário nativo em tempo de execução [5].

2.1.2 - O Modelo

Smalltalk apresenta uma série de conceitos que lhe são peculiares [Ingalls81], alguns deles sendo inspirados de outros sistemas. A noção principal é de objetos. Virtualmente falando, tudo em Smalltalk são objetos, os quais são criados a partir de classes. Tais objetos interagem através de trocas de mensagens. Esses conceitos foram inspirados em SIMULA [Kirkerud90].

Modelar problemas em Smalltalk, então, significa seguir a modelagem orientada a objetos. Ao contrário de pensar em um sistema centrado em suas atividades (processos), esta modelagem fundamenta-se em torno das entidades que compõem o domínio da aplicação em si [6]. Tais entidades são representadas como objetos em Smalltalk. Conforme similaridades entre grupos de objetos, são adicionadas classes no sistema. Tais classes têm a função de organizar a modelagem, permitindo definir relações entre as classes de objetos. Assim, um ou mais objetos em Smalltalk pertencem a uma classe. Esta, por sua vez, pode possuir uma superclasse [7] e uma ou mais subclasses. Tal classificação não é feita a esmo, e sim seguindo um conjunto de princípios de modelagem a serem vistos aqui.

2.1.3 - Objetos e Mensagens

No modelo de objetos, estes são a base de construção de software. Através de mensagens trocadas entre si, tais objetos interagem conforme o programa. Objetos têm "memória" própria, expressa na forma de variáveis. Assim, atributos de cada objeto (cada instância, a ser detalhado a seguir) são privados para cada um deles, somente sendo alterados pelo próprio objeto em si. Tal alteração, usualmente, ocorre em consequência da recepção de uma mensagem vinda de outro objeto, que a solicita.

Embora a idéia de objetos que se comunicam leve intuitivamente à idéia de paralelismo, isto não ocorre no modelo de objetos "tradicional". Aqui, existe apenas a noção de um *thread*[8] de execução. Ao enviar uma mensagem M ao objeto B, o objeto A fica bloqueado, à espera do resultado, a ser retornado por B. Desta forma, o envio de mensagens se assemelha em muito à chamada de procedimentos, e o fato de os objetos não serem paralelos entre si normalmente faz com que eles sejam comparados, inadequadamente, a estruturas de dados passivas, como registros em linguagens como Pascal, C, etc.

---

[5] Tal característica varia de implementação para implementaçao. Os produtos da ParcPlace e Digitalk, por exemplo, já apresentam tal característica.

[6] Aqui existe uma certa subjetividade em relação ao paradigma. Segundo [Meyer88] , a noção de objetos surge como sendo *orientada a dados*. Segundo [Yonezawa87a] , entretanto, a metáfora de objetos que trocam mensagens induz à noção de paralelismo. Isso induziria a se pensar que tal abordagem seria, então, *orientada a processo*. Acreditamos que isso, contudo, varia conforme a abordagem seja clássica ou não (falaremos disso mais adiante).

[7] A proposta original de Smalltalk compreende herança simples.

[8] Segundo [Wegner89] *thread* é "...o menor elemento executável dentro de um processo. Um *thread* é uma estrutura de dados que se torna ativa quando carregada no processador".

Para expressar paralelismo, Smalltalk provê objetos que representam distintos *threads* de execução. Desta forma, para expressar paralelismo na interação entre os objetos que compõem um determinado programa em Smalltalk, é necessário criar um objeto processo, o qual contém a sequência de computação desejada. Como, fatalmente, estes objetos precisarão sincronizar suas atividades, ou trocar "dados" (leia-se objetos) entre si, Smalltalk provê ainda um mecanismo de sincronização entre tais processos: semáforos. Através destes, a comunicação entre os diferentes *threads* pode ser obtida através de variáveis compartilhadas entre os processos. Controlando o acesso a tais variáveis (as quais descrevem objetos) , os processos distintos trocam informação entre si. Este tipo de abordagem é típico de sistemas fortemente acoplados, ou seja, aqueles onde existe a noção de memória compartilhada.

Por um lado, esta abordagem levou a linguagens com suporte a programação orientada a objetos com uma ênfase maior ainda para este sentido clássico: objetos são como registros, e a execução de código associado à recepção de uma mensagem corresponde à evocação de um procedimento. Nesta classe de linguagens se enquadram C++, C+@ [Fleming93], Modula-3, Eiffel, etc.

Por outro lado, contudo, esta limitação levou a linguagens onde os objetos seriam paralelos, autônomos. Nesta classe de linguagens enquadram-se ConcurrentSmalltalk [Yokote87], ABCL/1 [Yonezawa87b] [Yonezawa90b], Act-1 [Lieberman87] [Agha87] , etc.

2.1.4 - Classes e Instâncias

Smalltalk, como outros sistemas orientados a objetos, faz distinção entre a descrição de um objeto e o objeto em si. Objetos similares podem ser descritos por uma mesma descrição geral. Tal descrição é denominada classe, enquanto cada objeto descrito por uma classe é chamado instância daquela classe [Robson81].

A memória de cada objeto, conforme descrito anteriormente, é composta pelas suas variáveis de instância. Embora seja a classe que descreva quais variáveis de instância os objetos que a ela pertencem terão, serão as instâncias que efetivamente as terão, cada qual com valores distintos.

As classes, em Smalltalk, também têm a responsabilidade de armazenar o software de cada objeto [Robson81], ou seja, os métodos a serem executados em resposta a mensagens que são recebidas pelo objeto. Assim, ao receber uma mensagem o objeto reage conforme o método associado àquela mensagem. Como a classe que armazena tais métodos, instâncias de uma mesma classe reagirão de forma similar a uma mesma mensagem. A diferença no resultado se dará, provavelmente, nos diferentes valores das variáveis de instância de cada objeto. Estas variáveis de instância, por sua vez, podem ser identificadas por nomes, os quais são descritos também na classe.

Algumas linguagens procuraram uma abordagem alternativa, onde a noção de classe não existe. Assim, o sistema seria construído a partir de objetos protótipos. Estes armazenariam tanto as variáveis de instância quanto os métodos. Cada novo objeto X criado teria, potencialmente, um objeto protótipo Y que conteria os métodos que X não tem (não precisa ter). Assim, ao tentar tratar uma mensagem, X pode optar por delegar a Y o tratamento da mesma. Self [Ungar87] é um exemplo deste tipo de abordagem, embora em sua proposta

inicial não apareça a noção de paralelismo, ou seja, segue a linha de Smalltalk neste aspecto. Por outro lado, Act-1 é um exemplo onde a noção de paralelismo está presente.

Uma classificação mais detalhada das diversas variações do termo "orientado a objetos" pode ser encontrado em [Wegner89].

2.1.5 - Herança, Tipagem e Acoplamento Dinâmico

Em Smalltalk, assim como outros sistemas orientados a objetos com suporte para herança, um objeto pode herdar atributos de outro. Mais especificamente, uma nova classe pode ser adicionada ao sistema, como sendo subclasse de uma já existente, herdando as descrições definidas na classe já existente. A nova classe pode definir novos métodos, assim como novas variáveis de instância que seus objetos terão. Desta forma, Smalltalk apresenta um caráter incremental e diferencial em suas classes. Incremental pois as classes vão sendo criadas e adicionadas no sistema gradativamente, uma após a outra, cada vez adicionando mais variáveis de instância e mais métodos. O aspecto diferencial se deve ao fato de uma nova classe poder, por exemplo, redefinir o comportamento de um determinado método já definido em sua superclasse. Desta forma, objetos pertencentes a esta nova classe terão comportamento diferenciado, dada pela especialização definida pelo nova implementação de método na nova classe adicionada.

A possibilidade de herança é, sem dúvida, um dos maiores atrativos em sistemas com suporte à orientação a objetos. Enquanto a modularidade é beneficiada através do ocultamento de informação (*information hiding*) [Takahashi88] provido pelos objetos, é a herança de atributos e métodos que favorece enormemente a reusabilidade de código. Mais além, a herança permite que sistemas complexos sejam refinados e adaptados mais rapidamente, pois componentes similares são reaproveitadas e alterações são inseridas de maneira relativamente fácil, graças à generalização/especialização possibilitada pelo mecanismo de classificação.

Estritamente falando, tipagem não existe no sistema Smalltalk. A qualquer momento, determinada variável que denota um objeto pode passar a denotar um outro qualquer, sem que o sistema indique qualquer violação de tipagem. O fato de uma variável poder referenciar uma instância de objeto, pertencente a uma classe qualquer, fornece a Smalltalk o mecanismo normalmente citado como tipagem dinâmica [Takahashi88]. Esta "tipagem", na verdade, tem íntima ligação com o mecanismo denominado de acoplamento dinâmico (*dynamic binding*). Aqui, a implementação de operação (método) associada ao recebimento de uma mensagem é resolvida dinamicamente, em tempo de execução do sistema. Por exemplo:

```
obj print.
```

Listagem 2.1 - Envio de Mensagem em Smalltalk

denota o envio da mensagem *print* ao objeto *obj*. Olhando este trecho, não é possível identificar a qual classe *obj* pertence. Em tempo de compilação, Smalltalk não tem condição de verificar se esta linha produzirá um erro ou não. Isso porque *obj* é uma variável que pode vir a referenciar um objeto que entenda a mensagem *print* ou não. Esta verificação, então, é postergada ao tempo de execução. Ao receber efetivamente a mensagem *print*, o objeto denotado por *obj* deverá executar o método associado à operação. Caso a classe de *obj* não defina tal método, começa uma busca em direção ascendente na cadeia de herança, até achar

onde tal método foi definido. Caso nenhuma das superclasses defina tal método, ocorre um erro [9].

2.1.6 - Gerenciamento de Memória

Durante a execução do "programa" em Smalltalk, diversos objetos são criados. Tais instâncias podem ter existência durante todo o tempo em que o programa é executado, ou não. O programador, contudo, não precisa se preocupar com o gerenciamento de memória associado à criação e remoção de tais objetos. Aqui, o sistema provê a coleta automática de porções de memória onde se encontram objetos não mais referenciados pelo sistema. Grosseiramente falando, quando um objeto não mais é referenciado de nenhuma parte da aplicação, ele deixa de existir, e a memória associada à sua existência (variáveis de instância, etc) é novamente tratada como memória livre para ser alocada para novos objetos. Essa tarefa é realizada por um componente chamado de coletor de lixo. Graças a ele os programadores vêem-se livres da gerência manual[10] de memória em cada uma de sua aplicações. Gerência automática de memória é normalmente listada como característica fundamental para caracterização de sistemas verdadeiramente orientados a objetos [Meyer88] [Kaehler81].

2.1.7 - Programando em Smalltalk

A partir de classes, objetos, mensagens, como se dá a construção de um *software* em Smalltalk ?

Esta dúvida é frequente àqueles que se iniciam no sistema. Primeiramente, programar em Smalltalk está usualmente associado a uma técnica de análise orientada a objetos [Rumbaugh91]. Seja ela feita desta forma ou através da prototipação [Diederich87] [Tozer87], produzir *software* final em Smalltalk requer muito mais que apenas entender os conceitos básicos.

2.1.7.1 - Reusando e Adaptando

O primeiro passo no desenvolvimento de aplicações no modelo é tornar-se familiar com a extensa biblioteca de classes/métodos/objetos que acompanham o sistema. Tornar-se familiar com o sistema é fundamental para que se reuse componentes pré-fabricados em novas aplicações [Goldberg90]. Ao invés de recriar porções inteiras de rotinas, o sistema promove, através do mecanismo de classificação, associado à extensa biblioteca, a utilização direta de métodos, classes e objetos já disponíveis. Estruturas de dados como dicionários, filas, vetores, etc são apenas uma pequena parte desta biblioteca.

Haverá, certamente, casos em que a bilioteca não é de todo adequada. Nestes casos, a herança permite refinar componentes existentes, criando novos elementos que satisfazem as

---

[9] Tal erro consiste em a máquina virtual despachar a mensagem *#doesNotUnderstand:* ao mesmo objeto.

[10] O programador necessita alocar explicitamente memória em tempo de execução, devendo liberá-la quando esta não mais estiver sendo utilizada. Assim, além da aplicação propriamente dita, o programador necessita se preocupar também com o gerenciamento devido de memória, certificando-se sempre que somente será liberada memória realmente não mais utilizada. A gerência automática, por outro lado, permite garantir um gerenciamento livre de enganos acidentais por parte do programador.

necessidades específicas da aplicação. Isso favorece enormemente a prototipação e a facilidade de adaptação de aplicações.

Por outro lado, o refinamento de classes já existentes deve ser feita com cuidado, para que estes novos componentes possam tornar-se tão úteis quanto as classes já existentes. É importante lembrar que as classes já existentes são fruto de anos de pesquisa e consolidação do modelo. Normalmente a adição de novas classes passa por adaptações até atingir um estado em que se estabiliza e apresenta genericidade suficiente para ser reusada em novas aplicações. A produção de novas classes se relaciona intimamente com o domínio da aplicação a desenvolver. Isso faz com que, embora uma primeira aplicação em determinado domínio em Smalltalk tenha a tendência a ser mais demorada (principalmente se suas classes são desenvolvidas com cuidado para promover a reusabilidade) , novas aplicações tendem a ser fáceis de se produzir, devido às características anteriormente citadas [Goldberg87]. A capacidade de produzir novas classes que sejam reusáveis permite diferenciar programadores Smalltalk experientes ou não. [Rumbaugh91] descreve uma série de princípios que programadores experientes tendem a seguir para produzir componentes úteis, genéricos e reusáveis.

A reusabilidade de componentes de outras linguagens, por outro lado, tambem é possível. Seja ela através do interfaceamento com "módulos" produzidos em outras linguagens como C ou até mesmo com bibliotecas dinâmicas.

2.1.7.2 - A Interface

Após dominar a biblioteca de classes do sistema, é preciso entender o modelo de construção de novas aplicações interativas. Embora existam implementações Smalltalk que se destinam a aplicações sem interfaces gráficas [OTI-ENVY], o usual é utilizar-se do sistema para construção de aplicações interativas. Isso implica entender o mecanismo que Smalltalk utiliza para tentar prover alguma independência entre a aplicação em si e a sua interação com o usuário. Temos, aqui, mais um conjunto de classes que se destinam unicamente à interação com o usuário, e que podem ser refinadas para produzir novos componentes de interface. Além de dominar a biblioteca de classes em si, é necessário entender o paradigma que Smalltalk utiliza para despachar eventos junto à interface para os componenetes da aplicação propriamente dita: o Model-View-Controller [ParcPlace92a]. Algumas implementações usam modelos diferentes, usualmente tentando tornar-se mais próximo das GUIs que os suportam. A tarefa de construção das interfaces das aplicações é hoje enormemente facilitada por *kits* que permitem construir e editar interativamente a aplicação de uma interface, diminuindo enormemente seu tempo de desenvolvimento [ParcPlace92b].

Dominando a bibioteca de classes já existentes, a forma de se produzir novos elementos reusáveis, e o paradigma de construção de interfaces interativas em Smalltalk, o próximo passo é a forma de mapear a aplicação real, o problema existente, para código executável Smalltalk.

2.1.7.3 - Paralelismo em Smalltalk

Embora a idéia de objetos que se comunicam por troca de mensagens traduza, de forma intuitiva, a noção de paralelismo, esta não foi a abordagem adotada no projeto de Smalltalk. Envio de mensagens, aqui, teve a mesma semântica de evocação de procedimento, e a computação é feita de forma seqüencial [Deutsch81]. Assim, a modelagem de problemas inerentemente paralelos, em Smalltalk, dá-se com a noção de processos. Aqui, então, temos duas modelagens distintas:

- O emprego de objetos para modelar as entidades da aplicação
- O emprego de processos para modelar o paralelismo da aplicação

A forma de criar um novo *thread* de execução em Smalltalk é enviando a mensagem *fork* a uma instância de Context. Assim , no exemplo

```
metodoQualquer
| temp1 |

    [ codigoSmalltalk-A ]  fork.
    codigoSmalltalk-B
```

Listagem 2.2 - Paralelismo em Smalltalk

o código descrito por *codigoSmalltalk-A* e *codigoSmalltalk-B* serão executados em paralelo[12] . O ambiente do contexto do bloco[13] será o mesmo do contexto do método. Desta forma, mesmo as variáveis temporárias do método são comuns ao processo criado pelo *fork*. No caso acima, a variável *temp1* é visível tanto no *metodoQualquer* quanto no processo descrito pelo bloco *[codigoSmalltalk-A]* . Isso exige um recurso para exlusão mútua a estas variáveis compartilhadas conforme veremos a seguir.

2.1.7.3.1 - Exclusão Mútua em Smalltalk

O conjunto de classes básicas em Smalltalk provê um modelo conhecido para exclusão mútua: *semáforos*. Modelados como objetos normais de Smalltalk, instâncias de semáforos são a forma de se conseguir garantir exclusão mútua entre processos a recursos compartilhados (no nosso exemplo anterior, o objeto descrito pela variável *temp1*). A forma usual de utilização de semáforos, em Smalltalk, dá-se como segue:

---

[12] Usamos o termo paralelo em um sentido lógico. Se ele ocorre ou não, de fato, varia conforme a plataforma e a implementação.

[13] Em Smalltalk, o contexto de um método, responsável por armazenar informações como variáveis locais, etc, é armazenado em um objeto Smalltalk - Context

```
metodoQualquer
| contador mutex|

"Inicializações"
mutex := Semaphore new.
mutex send.
contador := 1.

"Primeiro Processo"
[ 10 timesRepeat:
    [  mutex wait.
       contador := contador + 1.
       mutex send.
    ]
]   fork.

"Segundo Processo"
10 timesRepeat:
    [  mutex wait.
       contador := contador + 1.
       mutex send.
    ].
```

Listagem 2.3 - Exclusão Mútua em Smalltalk

onde temos dois processos, cada qual incrementando o valor de uma mesma variável compartilhada - *contador*.

2.1.8 - Análise Orientada a Objetos

Modelar sistemas para posterior implementação em linguagens de programação é uma atividade complexa e subjetiva. Associadas aos paradigmas de programação normalmente surgem os métodos de análise e desenvolvimento de software. A análise estruturada [Rocha87], por exemplo, teve importância elevada em associação com a programação estruturada de linguagens como Pascal, etc. Na febre dos bancos de dados relacionais, por exemplo, [Date89] também desperta grande interesse como ferramenta de modelagem. Com a orientação a objetos o fenômeno é similar; [Rumbaugh91], [Booch88] são exemplos de propostas de análise para este novo paradigma. [Rumbaugh91], contudo, tem sido considerado um dos métodos de análise orientada a objetos de grande poder de expressão, combinando idéias de diversas outras propostas, inclusive de campos como bancos de dados.

É importante salientar que embora a nível de modelagem possam surgir objetos inerentemente paralelos que interagem entre si, Smalltalk não suporta diretamente esta idéia. Conforme dito anteriormente, um objeto, ao enviar uma mensagem, fica bloqueado até que o objeto receptor da mensagem retorne-lhe um valor. Desta forma, existe, a princípio, apenas um *thread* de execução em Smalltalk. Esta limitação é vencida, como vimos, com a utilização de (objetos-) processos. Temos, então, dois níveis distintos a: de um lado, objetos que trocam mensagens entre si, representando entidades da aplicação, mas que não apresentam poder de expressão suficiente para captar paralelismo; de outro lado, processos que representam sequências de atividades, potencialmente paralelas, mas que não modelam as abstrações que compõem o domínio da aplicação.

Esta dicotomia é forçada devido à implementação do modelo. Ela força a utilização de mecanismos de baixíssimo nível como semáforos para possibilitar a interação e comunicação entre processos. Algumas propostas, tal como ConcurrentSmalltalk (a ser visto mais adiante), buscam unificar as duas facetas em uma entidade só: objetos paralelos que se comunicam através de trocas de mensagens. Outras, como DistributedSmalltalk [FAQa], buscam adicionar mecanismos alternativos para implementações comerciais de Smalltalk, para facilitar o uso real (nao-acadêmico) desta tecnologia. Neste último enfoque encaixa-se nosso trabalho. Após um breve levantamento de paradigmas de programação paralela e de programação paralela orientada a objetos, proporemos um modelo que permite expandir o modelo de Smalltalk como ele é em implementações comerciais, adicionando um poder de expressão maior e mais uniforme.

2.1.9 - Facetas da Orientação a Objetos

Baseado nas diferentes características que uma linguagem orientada a objetos suporta, ela pode ser classificada em determinada categoria, conforme sugere [Wegner89]. São exemplos destas características suporte a classes, suporte a herança, abstração de dados, tipagem, concorrência, persistência, etc. Esquematicamente, temos:

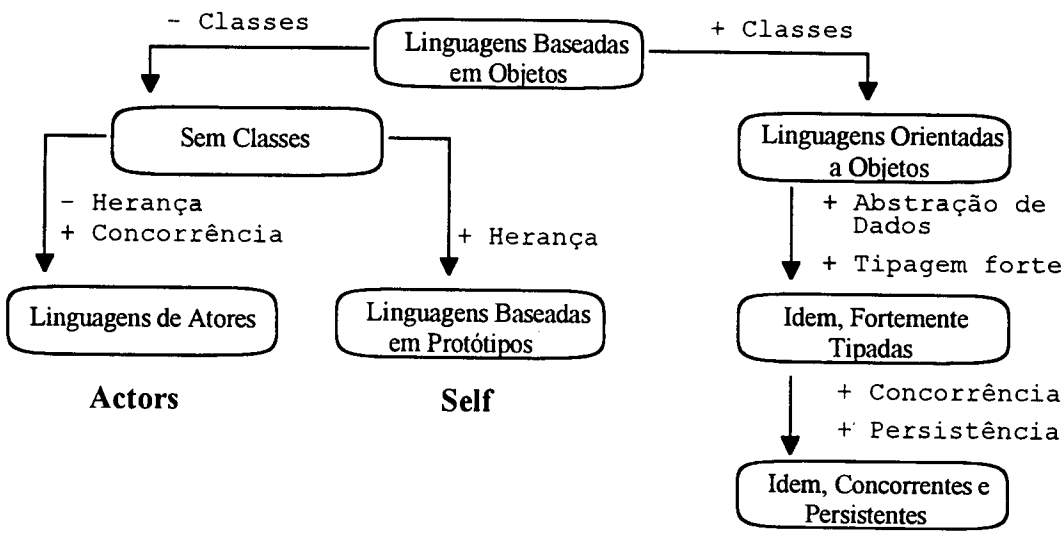

Figura 2.1 - Facetas da Orientação a Objetos

Visando tornar mais claras as possibilidades de inserção de paralelismo em linguagens orientadas a objetos é que seguem-se os capítulos deste trabalho.

3 - Programação Paralela

Um programa seqüencial especifica uma execução seqüencial de uma lista de comandos. Sua execução é denominada processo [Andrews83]. Um programa concorrente especifica dois ou mais programas seqüenciais que podem ser executados concorrentemente na forma de processos paralelos.

Multiprogramação é a execução concorrente de um programa onde os processos compartilham um ou mais processadores. Multiprocessamento, por outro lado, é a execução concorrente de um programa onde os processos possuem um processador cada um, e a memória que eles acessam é compartilhada. Esta abordagem caracteriza sistemas fortemente acoplados. Processamento distribuído é a execução concorrente de um programa onde os processos possuem um processador cada um, não possuindo memória compartilhada (são conectados por uma rede de comunicação). Estas abordagem caracteriza sistemas fracamente acoplados.

Para cooperarem entre si em rumo a uma atividade comum, processos concorrentes precisam se comunicar e sincronizar suas atividades. Tais comunicações podem se dar por variáveis compartilhadas (caracterizando os sistemas fortemente acoplados) ou trocas de mensagens (caracterizando os sistemas fracamente acoplados) [Andrews83]. Para expressar computações concorrentes, três questões principais precisam ser levadas em conta:

1) Como indicar execução concorrente ?

2) Que modo de comunicação inter-processos utilizar ?

3) Que mecanismo de sincronização utilizar ?

Na sua interação, processos fatalmente trocam dados. Estes dados, justamente, tem sido tratados de forma negligente por abordagens clássicas de programação paralela. Procuraremos analisar duas abordagens, ressaltando, além dos quesitos acima, como tais modelos expressam os dados, ou seja, como permitem representar, organizar e extender abstrações do domínio do problema.

Semáforos, monitores e *path expressions* [Andrews83] são exemplos típicos de mecanismos para prover comunicação e sincronização de processos em sistemas fortemente acoplados. Nosso interesse aqui, contudo, é em mecanismos para sistemas fracamente acoplados, propiciando o processamento distribuído, conforme explicado anteriormente.

Trocas de mensagens, por outro lado, podem ser vistas como um mecanismo de extender as propostas acima, de forma a englobar os dados[14] bem como a sincronização. Mostraremos aqui duas abordagens baseadas em trocas de mensagens, Occam e CONIC.

3.1 - Occam

Derivada do trabalho de Hoare sobre CSP [Hoare85], a linguagem Occam é usualmente associada aos Transputers[15] [Pountain87]. Utilizados normalmente como placas aceleradoras

[14] É justamente aqui que está o nosso interesse. Buscaremos ressaltar como estes dados trocados entre os processos se relacionam com a produção e manutenção de *software*, conforme enfatiza [Meyer88].

[15] *Transputer*, basicamente, é um dispositivo VLSI com memória, processador e *links* de comunicação para conexão direta com outros *transputers*

para algumas aplicações (principalmente computação gráfica) os *transputers* têm em Occam sua "linguagem de máquina". A importância de Occam, contudo, deriva principalmente do embasamento formal descrito pelo trabalho de Hoare.

3.1.1 - O Modelo de Occam

Occam promove a modelagem por processos[Pountain87]. Estes podem ser combinados, através de construções[16], formando outro processo. Não existe a noção de memória compartilhada; para efetivar a comunicação e sincronização de suas atividades, os processos se comunicam através de canais unidirecionais. Apenas um processo envia mensagens por um canal, e apenas um processo recebe mensagens por um canal. Esquematicamente temos:

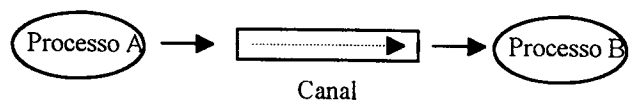

Figura 3.1 - Comunicação em Occam

3.1.1.1 - Processos Primitivos

Em Occam, os processos primitivos são:

- Atribuição
- Nulo
- Comando de Entrada
- Parada
- Comando de Saída

O primeiro deles (atribuição) é tambem encontrado em linguagens de programação mais comuns, como Modula-2 [Wirth85] etc. O comando de atribuição surge na forma:

```
v := e
```

Assim, será atribuído um valor igual ao resultado da expressão dada por *e* à variável *v*.

O comando de entrada, por sua vez, tem a seguinte forma:

```
c ? v
```

Onde o resultado da leitura pelo canal *c* será atribuído à variável *v*. O processo fica bloqueado, esperando um valor pelo canal, até que este esteja disponível para leitura.

O comando de saída, por sua vez, tem a forma:

```
c ! e
```

Onde o valor da expressão dada por *e* será enviado pelo canal *c*. Aqui, também, o processo ficará bloqueado até que o processo que lê o valor pelo canal também evoque a primitiva de

[16] Utilizamos o termo "construções" como tradução de "constructs". Uma tradução alternativa seria "construtores".

comunicação (no caso, de leitura). Assim, em Occam, a comunicação entre os processos é síncrona, ocorrendo somente quando ambos processos envolvidos evocam as respectivas primitivas de comunicação sobre um mesmo canal.

Os processos nulo (SKIP) e de parada (STOP) têm aplicação especial. SKIP é um processo que não faz nada e então termina. É normalmente usado onde, pela sintaxe de Occam, a presença de um processo faz-se necessária e o programador não deseja inserir nenhuma ação especial. SKIP, então, serve para atender à sintaxe e à necessidade do programador neste caso[17]. STOP, por sua vez, é um processo que não faz nada e não termina. Quando STOP é usado dentro do escopo de um processo seqüencial (ex. SEQ, a ser visto adiante) este é parado. Nada mais pode acontecer no processo. Todo processo que precisar se comunicar com este (que incluía o STOP) não terminará. O efeito de STOP, então, propaga-se (potencialmente) pelo programa como um todo.

3.1.1.2 - Construções

Para formar novos processos, os processos primitivos são agrupados através de construções. Em Occam, as construções principais são:

- SEQ
- IF
- PAR
- WHILE
- ALT

3.1.1.2.1 - Seq

A construção SEQ estabelece que os processos nele contidos são executados seqüencialmente, por ordem de listagem. Assim, o processo dado pela construção[18]

```
SEQ
          conversorAD ? leitura
          temperatura := leitura * fatorDeEscala
          mostrador ! temperatura
```

Listagem 3.1 - Construção SEQ de Occam

estabelece um processo formado por três processos primitivos, os quais são executados seqüencialmente. Ou seja, primeiramente é lido um valor através do canal de comunicação *conversorAD* (potencialmente um conversor analógico-digital), em seguida o valor da temperatura é calculado a partir deste valor, e finalmente este valor é enviado através do canal de comunicação *mostrador* [19].

---

[17] Veja o exemplo da construção IF, mais adiante

[18] Omitiremos, por ora, as declarações de tipos, constantes e variáveis

[19] No exemplo não estão listados nem o processo que envia o valor de leitura pelo canal *conversorAD* nem o que recebe a temperatura pelo canal *mostrador*.

3.1.1.2.2 - Par

A construção PAR , por sua vez, descreve um novo processo formado pela execução paralela dos processos listados. Assim, a construção a seguir

```
PAR
        SEQ
                conversorAD ? leitura
                temperatura := leitura * fatorDeEscala1
                mostrador ! temperatura
        SEQ
                comando ? aberturaDaValvula
                aberturaReal := aberturaDaValvula * fatorDeEscala2
                conversorDA ! aberturaReal
```

Listagem 3.2 - Construção PAR de Occam

descreve um processo formado pela execução paralela de outros dois processos. Estes dois processos, por sua vez, foram criados pela construção SEQ. Assim, temos um processo formado por dois processos paralelos, cada qual formado por três processos seqüenciais [20]. Esquematicamente, para a construção PAR[21], temos:

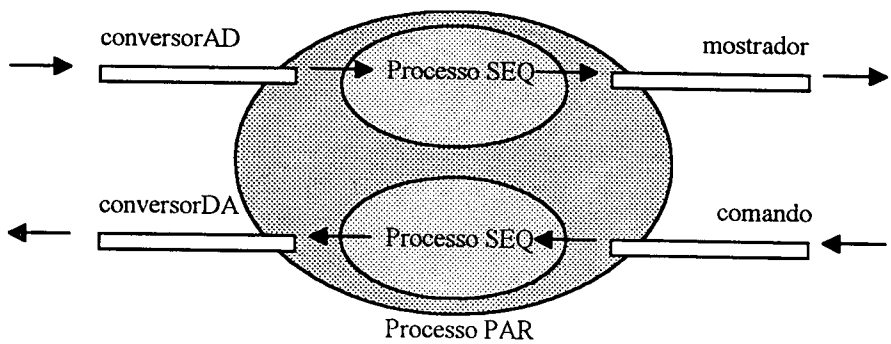

Figura 3.2 - Construção PAR de Occam

3.1.1.2.3 - If

A construção de escolha, em Occam, é o IF. Ele engloba uma série de processos, cada qual precedido por uma condição. Cada uma delas é examinada em ordem de listagem, e a primeira que resulta num valor verdadeiro tem seu processo correspondente executado. Assim, no exemplo:

---

[20] O exemplo, além de demonstrar a construção PAR, mostra que uma construção de Occam pode ser formado a partir de outras construções. Isso define, assim, uma forma de "agregação" de processos a partir de outros mais simples, que o compõem.

[21] Uma forma alternativa de PAR é PRI PAR, que permite estabelecer uma hierarquia de prioridades dos processos baseada em sua ordem de listagem. Maiores detalhes podem ser encontrados na bibliografia específica de Occam.

```
IF
      x = 1
          canal1 ! y
      x = 2
          canal2 ! y
      TRUE
          SKIP
```

Listagem 3.3 - Construção IF de Occam

caso *x* tenha o valor 1, *y* será enviado pelo *canal1*. Caso *x* tenha o valor 2, *y* será enviado pelo *canal2*. Caso *x* tenha um valor distinto desses, a condição TRUE é verdadeira, e o comando nulo é executado[22].

3.1.1.2.4 - Repetição

A repetição de uma série de comandos surge de duas formas:

- Repetição por um determinado número de vezes
- Repetição enquanto uma condição é verdadeira

A primeira forma tem estrutura geral

```
"REP" indice = base FOR contador
          ...
```

Listagem 3.4 - Repetição em Occam

onde "REP" é, na verdade, SEQ, PAR, IF ou ALT. Neste caso, é proibido alterar o valor da variável de *indice* (seja pela atribuição simples ou pelo comando de leitura através de um canal). Um exemplo, com a construção PAR, pode ser dado por:

```
PAR  i = 0 FOR 10
SEQ
     canal [i] ? dado
     ...processaODadoLido
           canal [i+1] ! dadoProcessado
```

Listagem 3.5 - Repetição com PAR em Occam

A repetição enquanto uma condição for verdadeira, por sua vez, surge como no exemplo que segue:

---

[22] A sintaxe de Occam requer algum processo seguindo a condição. No caso, como não se desejava nenhuma ação no caso de x não igualar 1 nem 2, utilizou-se a condição TRUE seguida do processo (obrigatório) SKIP. Por outro lado, caso se omitisse as duas últimas linhas, o fato de x não ser igual a 1 nem 2 faria com que o processo IF dado como exemplo fosse equivalente a um STOP.

```
SEQ
      x := 0
      WHILE x >= 0
          SEQ
                teclado ? x
                video ! x
```

Listagem 3.6 - Repetição Condicional em Occam

onde, após a inicialização de *x*, será efetuada sua leitura pelo canal *teclado* e envio para o canal *video* enquanto *x* for maior ou igual a zero. A repetição somente termina quando a condição de teste passar a ter o valor falso.

3.1.1.2.5 - Alt

A construção ALT, de forma geral, permite monitorar diversos guardas simultaneamente e executar o processo associado ao primeiro guarda a estar pronto. Caso dois ou mais guardas tornem-se prontos simultaneamente, um deles é escolhido de maneira não-determinística. Para entender o quê é um guarda e o que significa estar pronto, apresentaremos um exemplo:

```
ALT
      canal1 ? x
          ...processo1Qualquer
      canal2 ? y
          ...processo2Qualquer
      canalN ? z
          ...processoNQualquer
      .
      .
      .
```

Listagem 3.7 - Construção ALT em Occam

O que parece ser um processo de leitura, dado por *canal1 ? x* (e seus sucessores: *canal2 ? y* , etc) são, na verdade, guardas. Mais especificamente, são guardas de entrada[23]. No exemplo, eles especificam tentativas de leitura por canais distintos. A princípio, pode ser que nenhum deles seja "bem sucedido" (ou seja, os respectivos processos que enviam as mensagens pelos canais ainda não executaram a primitiva de envio - *canalN ! variavelQualquer*) e eles falham. Quando um dos processos que envia mensagem por um dos canais o fizer, o guarda correspondente torna-se pronto , e será o "escolhido" pela construção ALT. Esta escolha implica em executar o processo correspondente ao guarda (no exemplo, *processoNQualquer*).

Os guardas podem surgir combinados a expressões booleanas, como por exemplo:

---

[23] Assim como CSP, Occam não provê guardas de saída. O porquê, e como contornar sua falta em situações em que isso se faça necessário, pode ser encontrado na bibliografia específica de Occam.

```
ALT
      (y = 0) & canal1 ? x
          ...processo1Qualquer
      canal2 ? y
          ...processo2Qualquer
```

Listagem 3.8 - Guardas em Occam

Neste exemplo, o primeiro guarda somente estará pronto quando *y* for igual a zero e o processo que envia um valor pelo canal *canal1* tiver evocado a respectiva primitiva de saída (*canal1 ! variavelQualquer*).

Uma variação simples do ALT é o PRI ALT, que permite estabelecer a forma de "desempate" caso dois ou mais guardas se tornem prontos simultaneamente.

Com esta construção, caso haja "empate" entre dois guardas, o primeiro deles (na ordem de listagem do programa) será beneficiado (priorizado). Assim, no exemplo:

```
SEQ
    executando := TRUE
    WHILE executando
          PRI ALT
                fim ? qualquerCoisa
                      executando := FALSE
                canal1 ? valor1
                      ... processo1Qualquer
                canal2 ? valor2
                      ...processo2Qualquer
                .
```

Listagem 3.9 - Construção ALT Priorizado em Occam

Sabe-se que o processo definido pela construção WHILE terminará assim que um valor estiver disponível pelo canal fim graças à construção com prioridade. Sem a prioridade, o processo terminaria, mas eventualmente.

3.1.2 - Modelando os Dados

Até este ponto omitimos declarações de tipos e constantes para simplificar os exemplos. Passaremos, agora, a descrever os mecanismos de Occam para representar os dados.

3.1.2.1 - Tipos Primitivos

A exemplo de outras linguagens de programação (ex. Modula-2 [Wirth85]) , Occam provê alguns tipos primitivos de dados na linguagem. Os principais são:

·INT	Um valor inteiro, com sinal
·BYTE	Um valor inteiro, geralmente na faixa 0-255, normalmente usado para representar caracteres
·BOOL	Um valor booleando, *verdadeiro* ou *falso* (*TRUE* ou *FALSE*)
·REAL	Um número com ponto flutuante

Tabela 3.1 - Tipos em Occam

Occam provê, ainda, a possibilidade de definir vetores de dados de um mesmo tipo. Um exemplo seria:

```
[12] BOOL chaveAberta
```

onde se define um vetor com doze elementos do tipo BOOL. Cadeias de caracteres, por sua vez, são representadas como vetores de elementos do tipo BYTE.

3.1.2.2 - Agregação e Comunicação

Processos que interagem através de um canal o fazem graças a um protocolo de comunicação conhecido por ambos (tanto o processo que envia quanto o que recebe o dado). Isso equivale a dizer que caso um processo envie um valor do tipo INT por um canal, o processo que receberá tal dado deve esperar também um valor do tipo INT. Nos casos em que tais processos precisam trocar dados mais estruturados (ex. uma agregação de um INT, um BOOL e um REAL) deve-se declarar um protocolo com tal estrutura, e declarar o canal de comunicação como sendo daquele protocolo específico. Um exemplo seria:

```
PROTOCOL  Mistura IS INT; BOOL; REAL :
CHAN OF Mistura canal1:
...processoDadoPorUmConstruto
```

Listagem 3.10 - Agregação de Tipos em Occam

3.1.2 - Reusando Processos

Pode-se atribuir nomes a processos através de declarações PROC, seguida por um processo (denominado de corpo do procedimento). Um exemplo seria:

```
PROC imprime.caracter ( [] BYTE string , CHAN OF BYTE can , INT i)
    can ! string [i]
    ...
```

Listagem 3.11 - Dando Nomes a Processos em Occam

Uma referência (e evocação) deste processo seria na forma:

```
BYTE str [7] :
INT n:
CHAN OF BYTE canal:
SEQ
    str := "controle"
    n := 4
    imprime.caracter (str , canal, n )
```

Listagem 3.12 - Evocando Processos Nominados em Occam

O corpo do procedimento definido por PROC será executado sempre que seu nome surgir no corpo de um processo (como no exempo acima). A ocorrência de um nome de um procedimento em um processo é denominada uma instância do procedimento.

PROC permite utilizar um único procedimento através de evocações do mesmo por parte de diferentes processos, de forma análoga a procedimentos em linguagens tradicionais (ex. Modula-2).

3.1.3 - Construção de *Software*

A arquitetura de um programa final em Occam é resultado da combinação de vários componentes básicos. Tais componentes visam resolver situações clássicas, a serem detalhadas a seguir.

3.1.3.1 - Multiplexação

É comum a situação em que vários processos desejam se comunicar com (enviar uma mensagem para) um único processo. Como isso não é possível via um único canal de comunicação (já que não é possível compartilhar um único canal) utiliza-se normalmente um processo multiplexador, o qual permite efetuar comunicações do tipo 1 para N (1-N), representado esquematicamente abaixo.

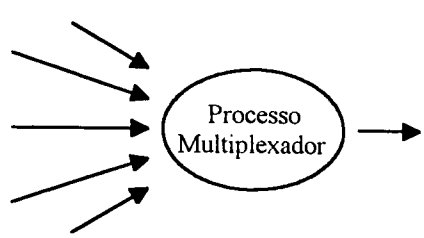

Figura 3.3 - Multiplexação em Occam

```
WHILE TRUE
    SEQ
        ALT
          entrada1 ? dado
                SKIP
          entrada2 ? dado
                SKIP
          .
          .
          entradaN ? dado
                SKIP

          ... processa o dado
          saida ! dadoProcessado
```

Listagem 3.13 - Multiplexação em Occam

3.1.3.2 - Demultiplexação

Em situações em que se deseja enviar um dado por um canal, a ser escolhido dentre N canais possíveis, utiliza-se o demultiplexador. Baseado em um critério de escolha qualquer, ele determina qual canal (e qual processo) receberá o dado. De forma geral, um demultiplexador tem a seguinte estrutura:

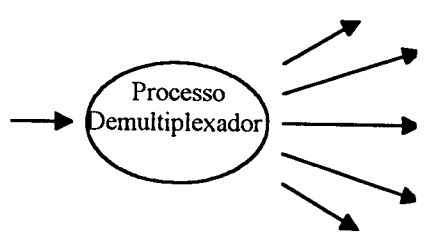

Figura 3.4 - Demultiplexação em Occam

```
SEQ
    entrada ? dado
    IF
        ...condição 1
         canal1 ! dado
        ...condição 2
         canal2 ! dado
         .
         .
         .
        ...condição n
         canaln ! dado
```

Listagem 3.14 - Demultiplexação em Occam

3.1.3.3 - *Pipe*

Quando se precisa conectar a saída de um processo qualquer à entrada de um outro, mas com a necessidade de fazer algum processamento entre os dois, utiliza-se o *pipe*[24]. O processo que serve de *pipe* simplesmente recebe o dado, processa, e passa-o para o próximo processo. Esquematicamente, teríamos:

Figura 3.5 - Pipes em Occam

```
SEQ
    canalEntrada ? dadoDeEntrada
    ...processa o dadoDeEntrada
    canalSaida ! dadoDeSaida
```

Listagem 3.15 - Pipes em Occam

A utilização de tais *pipes* em série também é bastante comum, levando a uma estrutura conhecida usualmente por *pipeline*:

Figura 3.6 - Pipeline em Occam

```
PAR

    SEQ
        canalN ? dadoDeEntrada
        ...processa o dadoDeEntrada
        canalN1 ! dadoDeSaida

    SEQ
        canalN1 ? dadoDeEntrada
        ...processa o dadoDeEntrada
        canalN2 ! dadoDeSaida

    SEQ
        canalN2 ? dadoDeEntrada

   ...processa o dadoDeEntrada
```

Listagem 3.16 - Pipeline em Occam

[24] Optamos por não traduzir o termo *pipe*, por ser bastante conhecido desta forma.

3.1.3.4 - Conector

O conector[25] é uma generalização do *pipe*. Enquanto a junta propaga o dado em um sentido só , os conectores o fazem em dois sentidos distintos: de norte para sul e de oeste para leste, conforme segue.

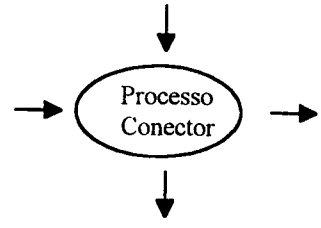

Figura 3.7 - Conectores em Occam

```
PAR

  SEQ
    canalOeste ? dadoOeste
    ...processa o dadoOeste
    canalLeste ! dadoOeste

  SEQ
    canaNorte ? dadoNorte
    ...processa o dadoNorte
    canalSul ! dadoNorte
```

Listagem 3.17 - Conectores em Occam

Através dos componentes básicos anteriormente descritos, aplicações (potencialmente complexas) em Occam são construídas. Componentes mais elaborados podem ser encontrados na bibliografia específica.

3.1.4 - Considerações

Occam apresenta um mecanismo elegante para expressar e sincronizar atividades concorrentes. Beseia-se em comunicação síncrona. Utiliza um mecanismo direto e estático para identificação de canais de comunicação [Andrews83]. Uma de suas limitações é a falta de capacidade de expressar o comportamento dinâmico de sistemas, seja pelo número de componentes paralelos (em Occam o número de processos é estático, conhecido em tempo de compilação) ou pelo mecanismo de identificação dos canais de comunicação [Stemple86]. Um caso típico são situações do tipo cliente/servidor. Um servidor deveria aceitar mensagens de quaisquer clientes. Se houver mais que um cliente no modelo, fica difícil expressar este tipo de situação. O modelo que apresentaremos a seguir supera este tipo de dificuldade.

Por outro lado, os dados em Occam são simplesmente tipos primitivos. Aplicações complexas, onde novos tipos devem surgir naturalmenete em decorrência do domínio do

[25] Denominamos conector o que é denominado array na bibliografia. Optamos por não utilizar o termo vetor para evitar conflitos com vetores no sentido de estrutura de dados.

problema (algo que a orientação a objetos clássica trata sem problemas) apresentam dificuldades para serem tratadas em Occam. Além das limitações claras no que tange exclusivamente a programação paralela, Occam também apresenta deficiências claras para expressar novos domíniios e respectivas operações de manipulação. Diversos quesitos apontados por [Meyer88] estão, claramente, ausentes em Occam.

3.2 - Conic

Desenvolvido no Imperial College, em Londres , Conic pode ser definido como um conjunto de ferramentas para construção de sistemas distribuídos [Kramer83] [Sloman86a] [Sloman86b].

Sua filosofia parte do princípio de que sistemas embutidos[26] e/ou distribuídos têm um longo ciclo de vida. Entretanto, mudanças (principalmente incrementais) ocorrem, e necessitam ser efetivadas enquanto a aplicação está ativa. Assim sendo, o modelo Conic se baseia num conjunto de princípios que buscam esta flexibilidade, as quais passaremos a descrever.

3.2.1 - Modularidade

A linguagem de programação de Conic se baseia em Pascal [Jensen75], com extensões para modularidade e troca de mensagens. Basicamente, existem dois tipos de módulos na linguagem de programação de Conic:

- Módulo de Tarefa
- Módulo de Definição

Um módulo do tipo Tarefa define uma tarefa seqüencial, auto-contida (processo). Módulos desse tipo são compilados em separado, independentemente da aplicação final. Nomes de entidades[27] nos módulos são locais, permitindo que alterações não causem impacto nos demais módulos que comporão a aplicação.

Um módulo do tipo Definição permite a introdução de extensões à linguagem. Dessa forma, é possível definir domínios e respectivas operações de manipulação, expandindo os tipos disponíveis na linguagem. Tais definições, estando em um módulo em separado, permitem seu compartilhamento por módulos Tarefa distintos. Note, contudo, que o modelo não disciplina o uso de tais módulos. Assim, é natural em Conic a existência de um único módulo com definições de todas as constantes, por exemplo, utilizadas por uma aplicação. Isso contrasta com boas técnicas de construção de software, como aponta [Meyer88].

---

[26] Sistemas embutidos são caracterizados por serem executados em geral em um processador dedicado, não tendo a noção de usuário. Assim, tais sistemas geralmente não possuem sistemas de interface com usuário, mas somente interfaceamentos específicos com periféricos ou sistemas de comunicação acoplados.

[27] Neste contexto, entidades são *portos*, *tipos*, *módulos*, etc.

Um exemplo de módulo de Definição segue abaixo.

```
define definiçõesDeString: comprimento, copia
opaque: string;

const strmax = 128;

{Definição do domínio }
type string =   record
                      comp: integer;
                      ch: array [1..strmax] of char;
                end;

{Definição das operações sobre o domínio }

function comprimento(s:string) : integer;
 ...
procedure copia(s1,s2 : string);
...
end.
```

Listagem 3.18 - Módulo de Definição em Conic

Neste exemplo, o módulo de definição *stringdefs* estabelece um novo domínio - *string* - em Conic. Além de definir como o domínio *string* é representado (no caso, um *RECORD* de Conic/Pascal) este módulo lista também funções/procedimentos de manipulação sobre o domínio *string*. Assim, o módulo como um todo define uma extensão aos tipos já existentes em Conic: o tipo *string*.

Os módulos Tarefa, por sua vez, são os que compõem a aplicação final propriamente dita. Tais módulos, conforme definidos pela linguagem, são apenas tipos, não tendo existência em tempo de execução. Instâncias de tais módulos são as entidades que existem em tempo de execução da aplicação. Tais módulos podem ter sua instanciação parametrizada, e várias instâncias de um mesmo tipo de módulo Tarefa podem existir em uma mesma aplicação. Segue um exemplo de módulo Tarefa parametrizado.

```
task module Sensor (limite: integer )

type leitura=   record
                      valor:      integer;
                      perigo:     boolean;
                end;

type tipoDoSinal = ...

exitport  Alarme :          leitura;

entryport Temporizador:     tipoDoSinal;
          Pedido:           tipoDoSinal reply leitura;

var       valorLido:  leitura;
          sinal:            tipoDoSinal;

function leSensor(): integer;
{Executa operação de I/O para ler o sensor}
    ...
 end;

begin
    valorLido.valor := 0;
    valorLido.perigo:= false;
    loop
      select
          receive sinal from Temporizador
          => with valorLido do
                begin
                      perigo := false;
                      valor:= leSensor();
                      if valor> limite then
                      begin
                            perigo:= true;
                            send valorLido to Alarme;
                      end;
                end;
      or
          receive sinal from Pedido reply valorLido;
      end;
    end;
end.
```

Listagem 3.19 - Módulo Tarefa em Conic

Por ora não detalharemos construções como *select*, *receive* nem as abstrações *exitport* ou *entryport*. Por enquanto, interessa-nos apenas perceber que o módulo Tarefa sensor é parametrizado com um valor-limite para seu sensor; leituras acima deste valor são tratadas de forma especial. Assim, diferentes instâncias da mesma tarefa podem existir, cada qual tendo como temperatura-limite um valor distinto. O trecho de código acima representa tarefas que, uma vez instanciadas, lerão um sensor sempre que um sinal for recebido de um temporizador (*clock*) ou informarão o último valor lido sempre que receberem um pedido. Se a leitura (feita com base no sinal do temporizador) for maior que um determinado valor previamente estabelecido, a tarefa sinaliza um alarme. Esquematicamente, temos:

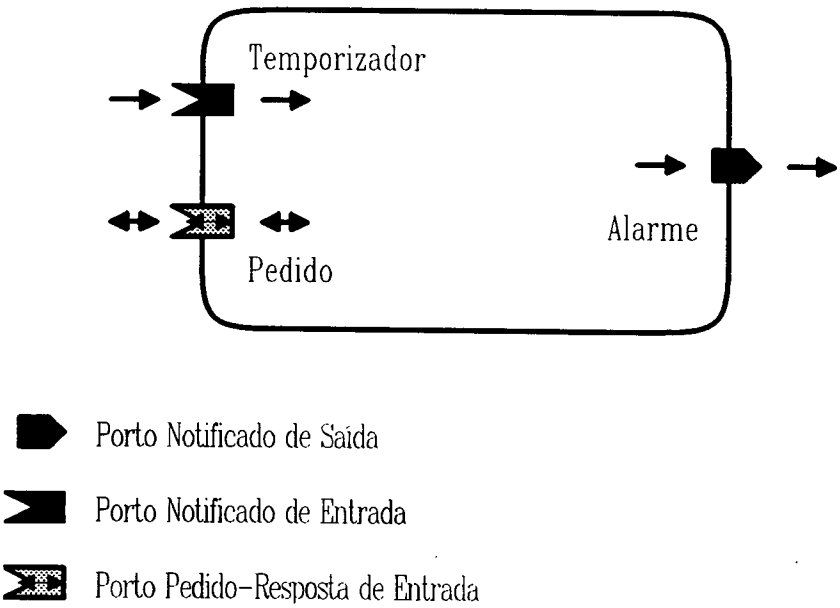

Figura 3.8 - Portos de Comunicação em Conic

3.2.2 - Comunicação entre Tarefas: Portos

A abstração existente em Conic que permite que processos se comuniquem e sincronizem suas atividades são os portos. Estes podem ser subdivididos em duas categorias: os portos pedido-resposta (*request-reply*) e os portos de notificação (*notify*). O primeiro tipo visa abranger situações de comunicação onde uma tarefa envia uma mensagem e espera outra como resultado [28] - uma comunicação tipicamente bidirecional. O segundo tipo visa abranger casos onde uma tarefa envia uma mensagem para outra tarefa qualquer, sem se interessar em nenhuma resposta - uma comunicação tipicamente unidirecional.

3.2.2.1 - Portos de Entrada e de Saída

Duas tarefas que queiram se comunicar o fazem através de portos. Tarefas enviam mensagens através de portos de saída (*exitports*, do sensor visto anteriormente), e recebem-nas através de portos de entrada (*entryports*, do mesmo exemplo). Assim sendo, duas tarefas se comunicam desde que o porto de saída de uma delas seja conectado ao porto de entrada da outra. Tal conexão é feita na fase de configuração de um sistema em Conic. Nesta etapa, o sistema como um todo é definido e, através da configuração, são feitas as conexões entre os portos de saída e de entrada das tarefas. Separando a entidade de comunicação (porto) em duas entidades distintas (porto de entrada e de saída) Conic busca viabilizar a produção de módulos Tarefa reusáveis independentemente da arquitetura do sistema que irão compor. Tal arquitetura, com respectivas ligações inter-modulares, é definida separadamente com a linguagem de configuração. Contraste-se com Occam, onde tais ligações fazem parte da

[28] Uma resposta para este envio

linguagem, e não podem ser dissociadas dos módulos (processos, em Occam) componentes. Assim, em Conic, consegue-se independência de configuração para os módulos Tarefa.

3.2.2.2 - Primitivas de Comunicação

Basicamente as primitivas de Conic são de envio de mensagem por um porto de saída ou de recepção de mensagem por um porto de entrada. Os portos são tipados, sendo que somente mensagens do mesmo tipo do porto podem trafegar por ele. Assim, no exemplo do sensor visto anteriormente, o porto *Alarme* somente "aceitará" mensagens do tipo leitura. Violação a esta tipagem é detectada em tempo de compilação.

Os módulos Tarefa podem ou não situar-se em diferentes estações (que formam o sistema distribuído - uma rede por exemplo) . Esta alocação de tarefas por estação não compromete os módulos Tarefa nem a configuração do sistema. Isso porque:

- A nível de módulos Tarefa, todas as referências a portos são locais, baseadas em seus nomes.

- A nível de configuração do sistema, suas conexões são independentes da localização física dos módulos, uma vez que se baseiam apenas nos seus nomes.

As transações entre tarefas, conforme visto anteriormente, podem dar-se através de portos pedido-resposta ou de portos de notificação.

A comunicação em portos de notificação provê o envio unidirecional de mensagens (primitiva *send to*), com possibilidade de múltiplos receptores (comunicações do tipo 1 para N, com 1 tarefa enviando e potencialmente N tarefas recebendo a mensagem). A operação de envio é assíncrona, não bloqueando, assim , o processo que envia a mensagem. Pode-se ainda associar uma fila de mensagens de tamanho fixo a cada porto de entrada conectado ao porto de saída que "produz" tais mensagens. Estas são armazenadas em ordem de chegada, sendo que a mensagem mais velha é descartada quando uma nova mensagem é produzida e não existe espaço na fila. Neste tipo de comunicação, um exemplo típico seria como o da figura a seguir:

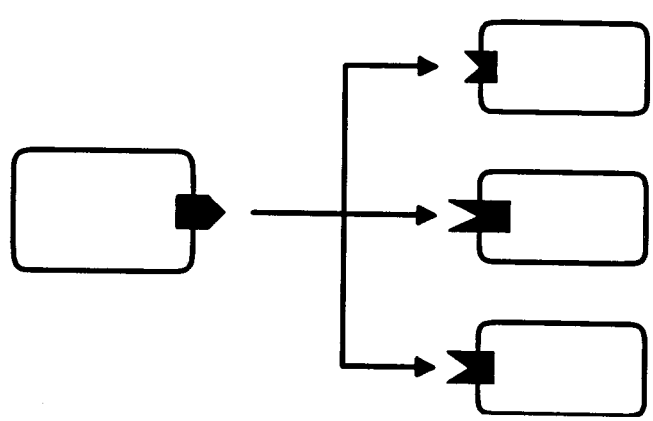

Figura 3.9 - Portos de Notificação em Conic

Neste caso, temos um porto de notificação de saída conectado a três portos de notificação de entrada com filas de mensagens de tamanhos distintos.

Já a comunicação em portos pedido-resposta provê envio bidirecional síncrono de mensagens (primitiva *send to wait*). A tarefa que envia a mensagem fica bloqueada até que a resposta seja recebida do destinatário original da mensagem. Uma cláusula de falha no comando permite que a tarefa que envia a mensagem desista de seu envio caso expire um período de tempo pré-fixado. A tarefa que recebe a mensagem pode efetuar algum processamento antes de responder ou, se preferir, passar o pedido adiante, permitindo a resposta por parte de terceiros. Neste tipo de comunicação, um exemplo típico seria como o da figura a seguir:

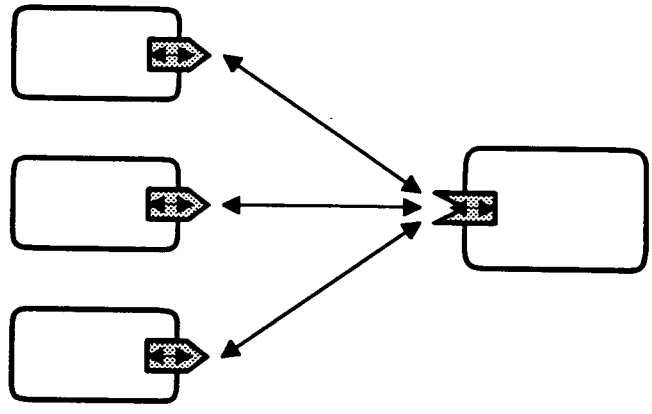

Figura 3.10 - Portos Pedido-Resposta em Conic

Neste caso, temos 3 portos de saída do tipo pedido-resposta conectados a um porto pedido-resposta de entrada.

Funções em Conic permitem saber se existe porto de entrada conectado ao respectivo porto de saída, quantas mensagens estão armazenadas na fila do porto de entrada ou a razão da falha na primitiva.

3.2.3 - Não-Determinismo

Qualquer uma das primitivas *receive*, *receive-reply*, *receive-forward* ou *receive-abort* podem ser combinadas na construção *select*. Isso possibilita que a tarefa espere mensagens de um número qualquer de portos de entrada. Um guarda opcional pode preceder cada primitiva de recepção para definir condições sob as quais mensagens deveriam ser recebidas. Caso vários guardas resultem, após sua avaliação, numa condição verdadeira, um deles é escolhido de forma não-determinística e os comandos a ele associados são avaliados. Um limite de tempo (*timeout*) pode ser usado para limitar o período de espera em um comando da construção *select*.

Um exemplo do uso da construção select segue no exemplo abaixo.

```
...
select
    when Guarda1
          receive pedido1 from portoDeEntrada1 reply sinal
    or
    when Guarda2
          receive pedido2 from portoDeEntrada2
          => ...
                forward pedido2 to portoDeSaida1
    or
          receive mensagem3 from portoDeEntrada3
          => ...
    ...
    or
    when GuardaN timeout periodoDeTempo
          => {Ação de timeout}
end;
```

Listagem 3.20 - Não-Determinismo em Conic

3.2.4 - Criando o Sistema Distribuído

Até agora foram vistos comandos da linguagem de programação de Conic, utilizados para criar os módulos de Tarefa e de Definição. A criação do sistema final, que poderá ser distribuído ou não, é feita utilizando a linguagem de configuração de Conic. Enquanto a primeira permite definir novos tipos de módulos, a segunda é que cria as instâncias de tais módulos e interconecta os portos das mesmas (ligando os portos de saída aos portos de entrada). Esta linguagem permite também definir mudanças que possam ocorrer no sistema, as quais são efetivadas dinamicamente, sem necessidade de parar a aplicação.

3.2.4.1 - Um Exemplo

Segue um exemplo onde uma família[29] de módulos *leito* é monitorada por um módulo *enfermeira*. Os módulos *leito* são instanciados com uma determinada taxa de amostragem e situados em diferentes estações da rede.

[29] Família, em Conic, é um Array de Instâncias de Módulo de Tarefa

```
system hospital;

use       monitorDeLeitos, unidadeEnfermeira;
const           numeroDeLeitos = 4;
          taxaDeAmostragem = 100;
          nodoDaEnfermeira= 5;


create family   k: [1..numeroDeLeitos]
          leito[k]: monitorDeLeitos (taxaDeAmostragem) at node (k);

create    enfermeira: unidadeEnfermeira at node (nodoDaEnfermeira);

link family     k: [1..numeroDeLeitos]
          leito[k].Alarme          to    enfermeira.Alarme[k];
          enfermeira.Monitora [k] to    leito[k].Estado;
end.
```

Listagem 3.21 - Um Exemplo de Sistema em Conic

Esquematicamente, tal sistema, após a instanciação de seus módulos e conexão dos portos, pode ser visualizado como a figura a seguir.

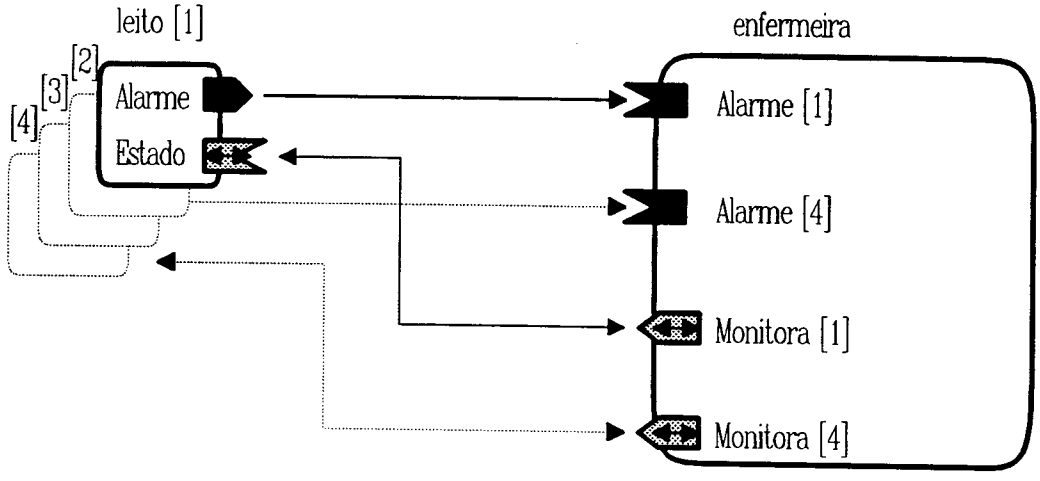

Figura 3.11 - Um Exemplo de Sistema em Conic

3.2.5 - Agregação

Assim como no modelo de objetos, o modelo de tarefas de Conic também permite a agregação. Através da linguagem de configuração, duas ou mais tarefas podem ser agregadas formando uma nova definição de módulo de Tarefa, chamado módulo de Grupo. Aqui, também, a agregação visa aumentar o nível de abstração, permitindo que novos componentes de software (tarefas, no caso) possam ser definidos em termos de outros componentes. Como exemplo, poderíamos criar um servidor de arquivos a partir de um servidor de diretórios e de um controlador de disco.

Segue um segundo exemplo, já na linguagem de configuração de Conic.

```
group monitorDeLeitos (taxaDeAmostragem: integer );

use pacientes:  tipoAlarme,
          tipoSinal,
          tipoEstadoDoPaciente;

exitport   Alarme: tipoAlarme;

entryport  Estado: tipoSinal
                reply       tipoEstadoDoPaciente;

{ Agora define estrutura do grupo }

use monitoramento, visualizadorDeLeitos;

create    monitor :monitoramento (taxaDeAmostragem);
          visor:      visualizadorDeLeitos;

link
          {Interno}
          monitor.Pedidos          to    visor.Area1;
          monitor.Dados            to    visor.Area2;
          visor.Teclado            to
monitor.Entrada;

          {Interface}
          monitor.Alarme           to    Alarme;
          Estado                   to    monitor.Estado;
```

Listagem 3.22 - Agregação de Módulos em Conic

Esquematicamente, o exemplo acima pode ser representado pela figura a seguir.

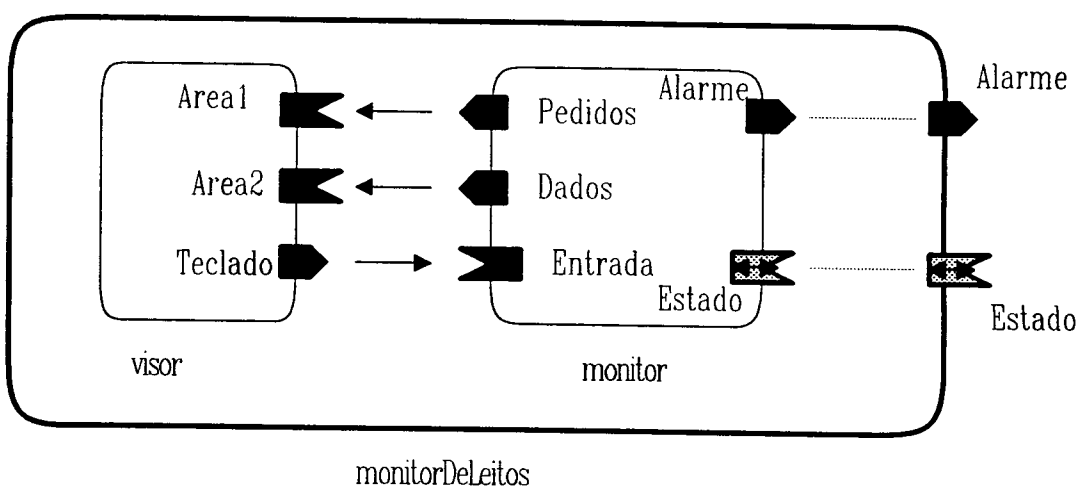

Figura 3.12 - Agregação de Módulos em Conic

3.2.6 - Adaptação

Conforme [Meyer88], uma qualidade que um software deve ter é a facilidade de adaptação a mudanças na sua especificação. O modelo de Conic prevê a alteração da arquitetura dos módulos do sistema enquanto o mesmo ainda está funcionando. Através do módulo de Mudança da linguagem de configuração, altera-se a estrutura do sistema dinamicamente. Dentre as mudanças previstas incluem-se adição/remoção de instâncias de módulos, instalação/remoção de novos tipos de módulos e reconfiguração da malha de conexão dos portos.

Um exemplo, em linguagem de configuração de Conic, segue abaixo.

```
change hospital;

const     outraTaxa= 500;
    nLeitos = 4;

unlink    family       k:[1..nLeitos]
          leito[k].Alarme          from  enfermeira.Alarme[k];
          enfermeira.Monitora[k]   from  leito[k].Estado;

delete          enfermeira;
remove    moduloEnfermeira;
use             diario,enfermeiraAtualizada;

create          log: diario (outraTaxa);
          novaEnfermeira: enfermeiraAtualizada;

link      family       k:[1..nLeitos ]
          log.PegaDado[k]          to    leito[k].Estado;
          leito[k].Alarme          to    novaEnfermeira.Alarme[k];
          novaEnfermeira.Monitora[k]     to leito[k].Estado;

link      novaEnfermeira.Historico       to    log.Historico;

end.
```

Listagem 3.23 - Adaptação Dinâmica de um Sistema em Conic

Esquematicamente, a nova configuração após tal mudança, pode ser representada pela figura a seguir.

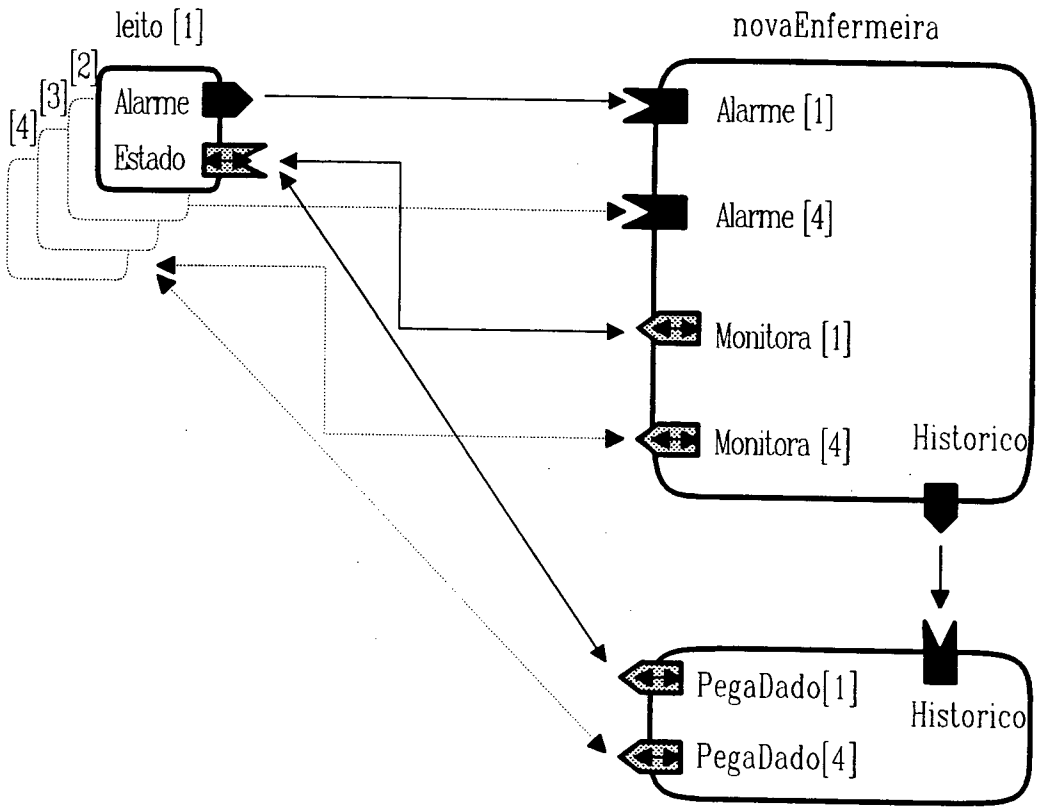

Figura 3.13 - Adaptação Dinâmica de um Sistema em Conic

Esta característica de Conic possibilita sua utilização em sistemas embutidos distribuídos. As circunstâncias sob as quais tais modificações podem, efetivamente, ser feitas em um sistema em andamento, bem como suas implicacões, podem ser encontradas com maior detalhe em [Sloman86a].

3.2.7 - Considerações Finais sobre Occam e Conic

Ao contrário de Occam, Conic possibilita expressar situações cliente/servidor naturalmenete. A utilização de portos caracteriza um caso especial de *mailbox*[30], onde o nome do mailbox só pode aparecer associado a operações de *receive* de um processo apenas [Andrews83]. As duas linguagens de Conic, por outro lado, permitem o uso disciplinado deste tipo de característica. Ao contrário de Occam, em Conic as topologias de conexão entre os processos podem ser alteradas dinamicamente e sem causar impacto algum sobre os módulos componentes, o que é altamente desejável [Meyer88]. A modularidade é promovida através deste tipo de perspectiva, aumentando a reusabilidade dos componentes.

[30] Mailbox é um mecanismo de nomes globais para identificação de canais de comunicação entre processos [Andrews83].

Por outro lado, Conic provê alguns mecanismos básicos para a representação de dados, como o ocultamento de informação. Fazendo uma analogia pelo que é sugerido em [Wegner89], podemos dizer que Módulos de Definição onde novos tipos sejam definidos caracterizam uma linguagem baseada em objetos. Entretanto, Conic não apresenta um mecanismo de refinamento de tipos, como o mecanismo de herança. Assim, não se pode garantir que um módulo possa estar fechado e aberto ao mesmo tempo, como aponta [Meyer88]. Mudanças de tipos se fazem "abrindo" um módulo existente, para posteriormente "fechá-lo" após a alteração, comprometendo demais módulos que dele faziam uso.

4 - Programação Concorrente Orientada a Objetos

Na tentativa de combinar os benefícios das linguagens orientadas a objetos e das linguagens para programação paralela, surgiram os modelos para programação concorrente orientada a objetos. Combinando os mecanismos de abstração providos pela orientação a objetos clássica com a possibilidade de expressar atividades paralelas, tais linguagens/modelos buscam abrir os horizontes dos dois paradigmas em que se inspira.

Selecionamos duas delas para aprofundamento aqui. Uma delas por buscar tal proposta em um modelo clássico de importância histórica, Smalltalk. A segunda delas por apresentar mecanismos inovadores, que unificam a noção de interrupção com a de recebimento de mensagens, apresentando ainda facilidades para delegação de mensagens.

4.1 - ConcurrentSmalltalk

Buscando oferecer um modelo não disponível na versão original de Smalltalk[31], a proposta de ConcurrentSmalltalk [Yokote87] visa a inserção do modelo de *programação concorrente orientada a objetos* na plataforma Smalltalk.

4.1.2 - Unificando Objetos e Processos

ConcurrentSmalltalk adota o paradigma de programação concorrente orientada a objetos - Objetos são similares a processos (da programação paralela tradicional) e se comunicam por troca de mensagens (similar à comunicação inter-processos). Neste modelo, um objeto não é somente a unidade de programação modular sob uma perspectiva orientada a objetos - é também a unidade de execução paralela.

4.1.3 - Princípios de Projeto

Diversos fatores nortearam o projeto de ConcurrentSmalltalk. Dentre eles, destacamos os seguintes.

4.1.3.1 - Compatibilidade

Ao invés de projetar uma linguagem completamente nova, ConcurrentSmalltalk extende o modelo de Smalltalk de forma a manter, com este, o máximo de compatibilidade possível. Assim, um conjunto de (fontes de) classes e métodos em Smalltalk podem ser trazidos diretamente para ConcurrentSmalltalk sem alterações. Mais ainda, uma imagem [32] Smalltalk pode ser carregada diretamente por ConcurrentSmalltalk. Isso permite uma maior reusabilidade de componentes previamente desenvolvidos para Smalltalk dentro do ambiente ConcurrentSmalltalk.

---

[31] Neste texto, usamos o termo *Smalltalk* para se referir ao produto da Xerox inicialmente batizado de Smalltalk-80.

[32] Em Smalltalk não existe a noção de *programa*. O sistema é sempre salvo em seu último estado - a imagem - contendo os objetos, os métodos compilados, etc. Uma nova sessão de trabalho é iniciada recarregando-se a imagem do sistema.

4.1.3.2 - Extensões

Para prover o modelo de objetos paralelos comunicantes são introduzidos em ConcurrentSmalltalk construções de concorrência, mecanismos de sincronização e objetos atômicos, descritos a seguir.

4.1.3.2.1 - Construções De Concorrência

Além de adotar a semântica de chamada de procedimento no caso de envio de mensagens, ConcurrentSmalltalk permite ainda:

- Que o objeto que recebe uma mensagem continue a execução após retornar um resultado
- Que um objeto envie diversas mensagens ao mesmo tempo

Assim, o modelo suporta a evocação de métodos com semântica de chamada de procedimento (para compatibilidade com Smalltalk) e também troca de mensagens como formas de interação entre os objetos. Tais mecanismos são chamados, na nomenclatura de ConcurrentSmalltalk, de evocação síncrona de método e evocação assíncrona de método.

O envio assíncrono de mensagem (o qual causará uma evocação assíncrona de método) surge da seguinte forma:

```
objetoQualquer mensagemQualquer &.
```

Listagem 4.1 - Envio Assíncrono de Mensagem em ConcurrentSmalltalk

Aqui, uma mensagem denotada pelo seletor *mensagemQualquer* será enviada ao objeto denotado pela variável *objetoQualquer* e o objeto que envia tal mensagem não ficará bloqueado, continuando a execução imediatamente.

Por outro lado, a sintaxe que permite que o objeto continue a execução do método mesmo após retornar um resultado é com o uso de "^^", conforme o exemplo a seguir:

```
metodoQualquer
    | local1 local2... valorARetornar|

    "...Código Smalltalk antes do retorno do resultado..."
    ^^ valorARetornar.
    "...Código Smalltalk após o retorno do resultado ..."
```

Listagem 4.2 - Continuando a Execução após o Retorno em ConcurrentSmalltalk

Aqui, o objeto executa um código qualquer (denotado por "*...Código Smalltalk antes do retorno do resultado...*") , retorna um resultado (denotado por ^^ *valorARetornar*) e continua a execução. O uso de "^^" ao invés de "^" (usual em Smalltalk) permite diferir, respectivamente, se o objeto que recebe a mensagem continua ou não a execução após o retorno de uma resposta à evocação do método.

Conforme os exemplos acima, na evocação assíncrona de método temos as seguintes características garantidas por parte do objeto que envia a mensagem:

- Comunicação unidirecional
- Envio de mensagem não implica em transferência de controle de execução
- Uma mensagem é recebida, necessariamente, após ter sido enviada

Por outro lado, temos as seguintes características garantidas por parte do objeto que recebe a mensagem:

- Um processo é evocado através da recepção da mensagem
- Existe uma relação mestre-escravo entre o objeto que envia e o que recebe a mensagem

4.1.3.2.2 - Mecanismos De Sincronização

Em evocações assíncronas de métodos, quando diversas mensagens são enviadas diversas respostas serão recebidas. Nestes casos, surge a necessidade de identificar que resposta corresponde a qual mensagem enviada. Em ConcurrentSmalltalk, *CBox* é um objeto utilizado para esta finalidade, identificando mensagens enviadas para posterior sincronização. Cada vez que é realizada uma evocação assíncrona de método, um novo objeto *CBox* é imediatamente retornado pelo sistema. Quando uma resposta for retornada pelo método evocado, esta será temporariamente armazenada no objeto *CBox* anteriormente instanciado.

Enviando a mensagem *receive* ao objeto *CBox* que armazena a resposta pode-se ter acesso à mesma. Caso a resposta ainda não tenha sido armazenada no objeto *CBox*, a execução deste objeto (que enviou *receive* ao objeto *CBox*) será suspensa. Permanecerá nessa forma até que uma resposta seja disponível no referido *CBox*.

A utilização de objetos *CBox* se dá, de forma geral, como segue:

```
metodoQualquer
    "baseDeDado é uma variável de instância"
          | cb resposta |
          cb := baseDeDado valorAssociadoA: #umaChaveQualquer &.
          "... Código Smalltalk qualquer ..."
          resposta := cb receive.
```

Listagem 4.3 - Utilizando *CBoxes* em ConcurrentSmalltalk

Aqui, envia-se uma mensagem assíncrona a um objeto (denotado por *baseDeDado valorAssociadoA: #umaChaveQualquer &*) e continua-se a execução. A variável *cb* será a referência ao objeto *CBox* que armazenará o resultado da evocação assíncrona do método escolhido. O objeto que enviou a mensagem continua sua execução normalmente (denotado por "... *Código Smalltalk qualquer* ...") e, posteriormente, espera até que o resultado esteja disponível em *cb* (denotado por *resposta := cb receive*).

4.1.3.2.3 - Objetos Atômicos

Em Smalltalk, como vimos no início deste trabalho, os semáforos são a forma de garantia de exclusão mútua a recursos compartilhados. Como se sabe, porém, semáforos distribuem ao longo do código de uma aplicação o controle sobre a exclusão mútua destes recursos, dificultando a boa modularidade e a manutenção.

No modelo de objetos, por outro lado, estes são compartilhados (potencialmente) por diversos outros objetos (na agregação, por exemplo). Assim sendo, é provável que duas (ou mais) mensagens cheguem ao mesmo tempo para um mesmo objeto. Temos aqui duas possibilidades:

a) Serializar estas mensagens e respondê-las uma-a-uma

b) Criar um novo contexto para cada mensagem que chega, executando-os independentemente

Para manter compatibilidade com Smalltalk, que adota a opção *b)* em seu projeto original, ConcurrentSmalltalk introduz a noção de objetos atômicos para abranger também a solução *a)* . Assim, mensagens recebidas por um objeto atômico são serializadas e respondidas uma-a-uma. Desta forma, apenas um contexto de método é criado para a execução de tais mensagens.

A diferenciação entre uma classe cujos objetos são atômicos ou não-atômicos (como Smalltalk) se dá na sua própria declaração [33], como segue:

```
NomeDaSuperclasse atomicSubclass: #NomeDaClasseAtomica
          instanceVariableNames:  '...'
          ...
          ...
```

Listagem 4.4 - Definindo Classes Atômicas em ConcurrentSmalltalk

Uma particularidade na semântica de objetos atômicos é o envio de mensagens às pseudovariáveis *self* e *super*. Em Smalltalk, isto é interpretado como sendo o envio de mensagem ao próprio objeto. No caso de ConcurrentSmalltalk, esta interpretação levaria a um *deadlock*, pois um novo contexto não pode ser criado[34]. Assim, neste caso particular, mensagens para *self* e *super* têm a semântica de evocação de procedimento.

---

[33] Estritamente falando, não existe a noção de declaração de classe. Seja através do ambiente (*browsers*, etc) ou através da importação de métodos, a inserção de uma nova classe no ambiente Smalltalk se dá, também, utilizando o próprio modelo so sistema: envio de mensagens a objetos.

[34] Quando objetos atômicos recebem mensagens, estas são serializadas e respondidas uma-a-uma, utilizando um mesmo contexto de bloco. Uma mensagem a *self* e/ou *super* faz com que a mensagem atual não seja terminada ainda (pois precisa que um resultado seja retornado) e, por outro lado, não permite que um novo contexto de bloco seja criado (devido ao fato de o objeto ser atômico). Essa situação causaria, então, um *deadlock*.

4.1.4 - Exemplo

Para visualizar melhor os conceitos apresentados, listaremos um programa[35] ConcurrentSmalltalk.

```
Object atomicSubclass: #BufferLimitado
        instanceVariableNames: 'buffer tamanho max
                                le escreve espera'
        classVariableNames: ' '
        poolDictionaries: ' '
        category: 'Produtor-Consumidor'  !
```

```
! BufferLimitado methodsFor: 'inicialização' !

configuraCom: n
        buffer := Array new: n.
        max := n.
        tamanho := 0.


le := escreve := 1 !!
```

```
! BufferLimitado methodsFor: 'acesso' !

deposite: dado
        espera notNil
              ifTrue: [ espera roda &.
                    espera := nil].

        "Se cheio, associa o objeto Produtor à variável espera"
        tamanho = max
              ifTrue: [ espera := thisContext sender receiver.
                    ^ #cheio].
        ^^ buffer at: escreve put: dado. "Retorna o próprio dado
inserido"
        "Execução continua após retorno"
        tamanho := tamanho + 1.
        escreve := escreve \\ max + 1 !

remova
        espera notNil
              ifTrue: [ espera roda &.
                    espera := nil].
        "Se vazio, associa o objeto Consumidor à variável
espera"
        tamanho = 0
              ifTrue: [ espera := thisContext sender receiver.
                    ^ #vazio].
        ^^ buffer at: le .
        "Execução continua após retorno do elemento desejado"
        tamanho := tamanho - 1.
        le := le \\ max + 1 !
```

[35] Aplica-se aqui o mesmo comentário feito anteriormente para Smalltalk a respeito de *programa* e *imagem*.

```
BufferLimitado class
          instanceVariableNames: ' '  !
```

```
! BufferLimitado class methodsFor: 'instanciação' !

criaCom: max
    | novoBuffer |
          novoBuffer := super new.
          novoBuffer configuraCom: max.
          ^novoBuffer !!
```

```
! BufferLimitado class methodsFor: 'exemplo' !

exemplo
    "Exemplo de BufferLimitado"

    | buffer produtor consumidor |

          buffer := BufferLimitado criaCom: 10.
          produtor := Produtor comBuffer: buffer eNome:
'PRODUTOR'.
          consumidor := Consumidor comBuffer: buffer eNome:
'CONSUMIDOR'.
          produtor execute&.
```

```
Object subclass: #Produtor
          instanceVariableNames: 'buffer  copia meuNome'
          classVariableNames:  ' '
          poolDictionaries: ' '
          category: 'Produtor-Consumidor'  !
```

```
! Produtor methodsFor: 'inicialização' !

comBuffer: umBuffer eNome: umNome
          buffer := umBuffer.
          meuNome := umNome. !!
```

```
! Produtor methodsFor: 'privados' !

produzDado
    "Produz o dado que será armazenado no buffer"

    "...Código qualquer ... " !!

! Produtor methodsFor: 'acesso' !

deposite: dado
    | deposito |
          deposito := buffer deposite: dado.
          deposito = #cheio
                ifTrue: [ copia := dado.
                      ^#cheio]
                ifFalse: [ ^dado]. !

execute
    | deposito |
          [true]
                whileTrue: [
                      deposito := self deposite: self produzDado.
                      deposito = #cheio
                            ifTrue: [^#cheio]  "Termina execução se
buffer encher"
                      ]. !

roda
    "Eu sou reiniciado pelo objeto BufferLimitado

    | deposito |
          deposito := self deposite: copia.
          deposito = #cheio
                ifTrue: [^#cheio]
                ifFalse: [self execute]. !!
```

```
Produtor class
          instanceVariableNames: ' ' !
```

```
 ! Produtor class methodsFor: 'instanciação' !

 comBuffer: buffer eNome: nome

  ^ self new comBuffer: buffer eNome: nome. !!
```

```
Object subclass: #Consumidor
          instanceVariableNames: 'buffer meuNome'
          classVariableNames:  ' '
          poolDictionaries: ' '
          category: 'Produtor-Consumidor'  !
```

```
! Consumidor methodsFor: 'acesso' !

execute
    | dado |
          [ true ]
                whileTrue: [
                      dado := self remova.
                      dado = #vazio
                            ifTrue: [ ^#vazio ]
                            ifFalse: [ self utilize: dado]
                      ]. !!

remova
    | dado |
          dado := buffer remova.
          dado = #vazio
                ifTrue: [^#vazio]
                ifFalse: [^dado]. !!

roda
    "Eu sou reiniciado pelo objeto BufferLimitado"
          self execute. !!
```

```
! Consumidor methodsFor: 'inicialização !

comBuffer: umBuffer eNome: nome
          buffer := umBuffer.
          meuNome := nome. !!
```

```
! Consumidor methodsFor: 'privado' !

utilize: dado
    "Utilização efetiva do dado consumido"

    "...Código qualquer ... " !!
```

```
Consumidor class instanceVariableNames: ' ' !
```

```
! Consumidor class methosFor: 'instanciação' !

comBuffer: umBuffer eNome: umNome
          ^self new comBuffer: umBuffer eNome: umNome !!
```

Listagem 4.5 - Buffer Limitado em ConcurrentSmalltalk

O exemplo anterior ilustra o uso de objetos atômicos para garantir a serialização de mensagens recebidas. *BufferLimitado*, acima, é definido como sendo atômico para que mensagens recebidas tanto do produtor quando do consumidor sejam serializadas e respondidas uma-a-uma.

No método de classe exemplo, em *BufferLimitado*, pode-se observar o uso da construção de concorrência "&" como forma de iniciar novos *threads* de execução. O exemplo, contudo, não utiliza objetos *CBox* para sincronismo entre os objetos paralelos. Isso acontece porque, neste exemplo, não ocorre a situação em que um objeto envia mais do que uma mensagem assíncrona e depois queira identificar cada um dos resultados retornados.

Ainda no mesmo exemplo, no método de instância *remova* de *BufferLimitado*, podemos ver o emprego de "^^" como forma de explorar o paralelismo. Desta forma, o buffer pode continuar a execução mesmo após retornar um resultado ao consumidor.

4.1.5 - Considerações

ConcurrentSmalltalk busca eliminar a dicotomia de objetos passivos e processos ativos com um modelo uniforme. Através da forma de envio de mensagem (envio síncrono ou assíncrono) o objeto que recebe a mensagem tem comportamento que se assemelha com objetos da noção clássica de orientação a objetos ou com a de processos da programação paralela. A possibilidade de reusar "programas" Smalltalk existentes faz de ConcurrentSmalltalk um modelo altamente atrativo no que tange a reusabilidade de componentes já prontos [Meyer88].

Por outro lado, as implementações comerciais de Smalltalk continuam a evoluir, invalidando as implementações de ConcurrentSmalltalk. Isso porque ConcurrentSmalltalk modifica a máquina virtual de Smalltalk, que também continua sendo alterada em suas novas versões comerciais. Isso inviabiliza a compatibilidade entre estas diferentes linhas.

4.2 - ABCL/1

Ao contrário de ConcurrentSmalltalk, que extende um modelo já existente como o de Smalltalk, ABCL/1 [Yonezawa87b] [Shibayama87] [Yonezawa90a] parte de uma proposta completamente nova. O paradigma como um todo engloba, na verdade, um Modelo de Computação Concorrente Orientado a Objetos - ABCM/1 - e sua Linguagem de Descrição propriamente dita - ABCL/1 [Yonezawa90b].

4.2.1 - O Modelo de Computação ABCM/1

As duas noções fundamentais de ABCM/1 são os objetos - agentes de processamento de informação, com caráter antropomórfico - e as suas formas de interação mútua. Tais formas de interação foram abstraídas a partir de modelos de comunicação reais encontrados em organizações humanas/sociais. Em ABCM [36] , o sistema a ser modelado, projetado e implementado é representado como uma coleção de objetos, e a interação de seus componentes é representada através de trocas de mensagens, de forma concorrente.

4.2.1.1 - Objetos

Neste modelo, cada objeto tem sua capacidade própria de processamento, podendo ter uma memória própria e privada. Seu estado, então, pode ser dado pelo conteúdo de sua memória privada. Ao receber uma mensagem, um objeto em ABCL pode executar uma série de ações básicas:

i) Comunicar-se com outro(s) objeto(s) através de primitivas de comunicação

ii) Criar outro(s) objeto(s)

---

[36] Usaremos indistintamente as duas siglas em determinados contextos

iii) Acessar/Alterar o conteúdo de sua memória privada

iv) Executar operações variadas (operações aritméticas, etc) sobre estes valores em sua memória ou valores passados entre objetos através de primitivas de comunicação.

As mensagens que um objeto pode aceitar são determinadas por padrões nas mensagens, valores contidos nas mensagens, e o estado atual do objeto. Para definir um objeto, então, é preciso especificar:

- Como sua memória privada é representada
- Sob quê condições mensagens são aceitas
- A sequência de ações a ser executada quando uma mensagem é aceita.

De forma geral, a definição de um objeto em ABCL pode ser dada por:

```
[object nomeDoObjeto
    (state      representação da memória privada )
    (script
          (=> padrão da mensagem where restrição ... ação ...)
          ...
          (=> padrão da mensagem where restrição ... ação ...)
    )
]
```

Listagem 4.6 - Definição de Objeto em ABCL/1

4.2.1.2 - Estados de um Objeto

Os possíveis estados de um objeto em ABCL são dormente, ativo e esperando. Inicialmente, um objeto está no estado dormente. Os padrões e restrições sobre as mensagens que podem ser recebidas são aqueles descritos na definição do objeto (conforme exemplo acima). Ao receber uma mensagem que satisfaça tais padrões/restrições, o objeto torna-se ativo [37]. No estado ativo, objetos podem executar as ações básicas listadas em 4.2.1.1.

Objetos que realmente possuam uma memória local própria, privada, não podem ser ativados por mais que uma mensagem ao mesmo tempo. Mensagens a eles destinadas são, então, armazenadas em uma fila[38] . Após executar o processamento associado a uma determinada mensagem recebida, o objeto recebe a próxima mensagem da fila que satisfaz algum dos seus padrões/restrições presentes na descrição do objeto. Mensagens intermediárias que não satisfaçam estes padrões/restrições são automaticamente descartadas. Todo processo se repete até que a fila de mensagens se esvazie. Neste momento, o objeto retorna ao estado dormente. Esquematicamente, antes do tratamento de uma mensagem, teríamos:

---

[37] Se 2 ou mais satisfazem, a primeira em ordem de listagem será escolhida.

[38] Esta fila é, conceitualmente, infinita.

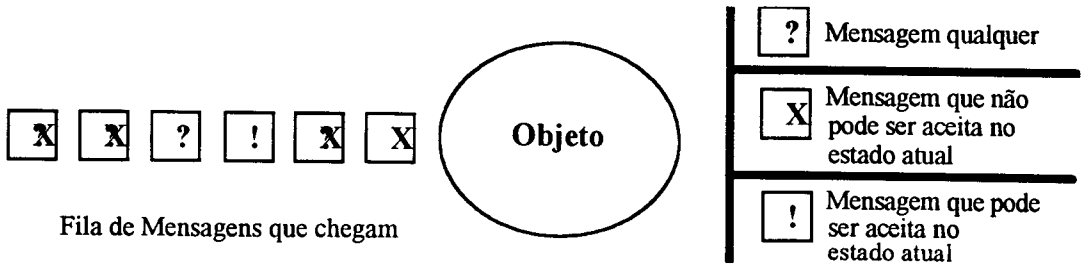

Figura 4.1 - Tratamento de Mensagens em ABCL/1

As mensagens que não podem ser aceitas no estado atual serão descartadas, e a nova situação seria:

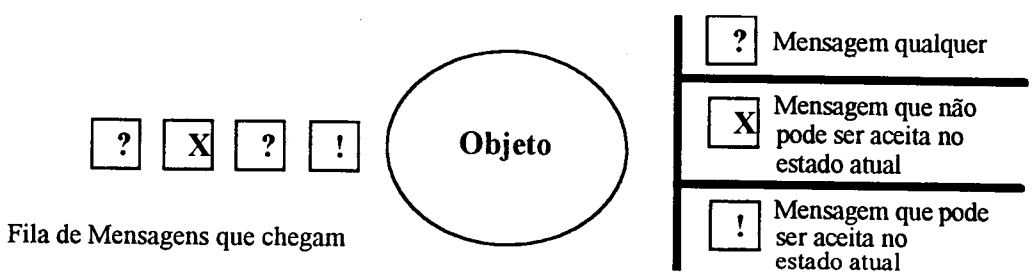

Figura 4.2 - Descarte de Mensagens Impróprias em ABCL/1

O objeto agora retiraria (da fila) a mensagem que pode ser aceita e iria tratá-la.

Estando no estado ativo, um objeto pode, por vezes, necessitar parar sua execução e esperar que determinada mensagem chegue a ele. Ele passa, então, ao estado de esperando. Análogo ao caso acima, ao receber a mensagem que satisfaz os padrões/restrições necessários é que o objeto retornará ao estado ativo. Esquematicamente, os possíveis estados de um objeto e as transições possíveis podem ser ilustradas a seguir.

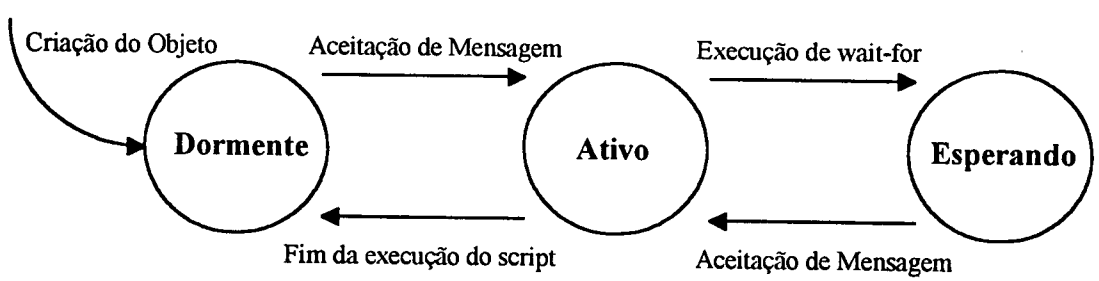

Figura 4.3 - Estados de um Objeto em ABCL/1

A forma de um objeto no estado ativo esperar uma mensagem é através da construção *wait-for*. De maneira geral, esta construção tem a seguinte forma em ABCL/1 :

```
(wait-for
    (=> padrão da mensagem where restrição ... ação ... )
    ...
    (=> padrão da mensagem where restrição ... ação ... )
)
```

Listagem 4.7 - Construção *wait-for* em ABCL/1

Esta construção, em ABCL, é também conhecido por recepção seletiva de mensagem. Um exemplo simples da necessidade de tal construção seria na modelagem de um objeto *buffer*. Sempre que estivesse em estado dormente, este *buffer* poderia receber mensagens para adicionar um novo elemento no buffer. Porém, ao tentar adicionar um elemento quando o *buffer* estiver cheio, o objeto terá , justamente, que fazer uma recepção seletiva de mensagem (neste caso, esperar pela mensagem de consumo de elemento do buffer, liberando assim um espaço para o novo elemento). A situação oposta também apresenta a mesma característica (tentativa de retirar um elemento do *buffer* quando este está vazio. Neste caso, o objeto utilizará a construção *wait-for* para que algum elemento seja colocado no buffer, para que possa ser consumido).

A forma de tratamento das mensagens que aguardam na fila difere, conforme o estado do objeto seja dormente ou esperando. Embora em ambos modos a mensagem que será escolhida será a primeira (a que chegou primeiro e) que satisfaça os padrões/restrições de recepção de mensagem do objeto, no modo esperando, contudo, apenas esta mensagem é retirada da fila e as demais permanecem onde estavam (contraste com o exemplo dado anteriormente).

Esquematicamente, antes de tratar a próxima mensagem no estado esperando, teríamos:

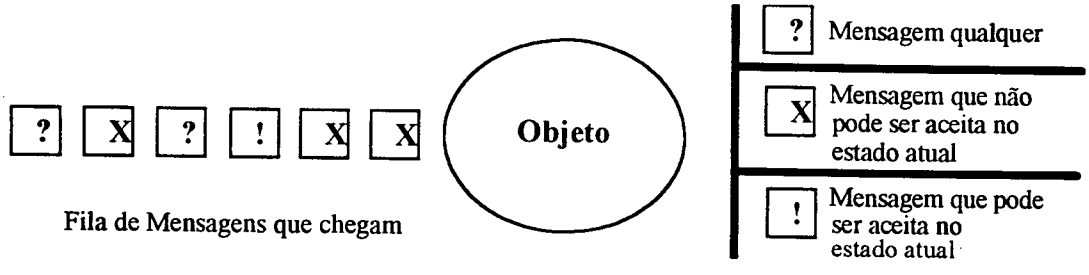

Figura 4.4 - Tratamento de Mensagens no Estado Esperando em ABCL/1

O objeto retiraria a primeira mensagem passível de tratamento, não descartando as demais. Após retirar tal mensagem, teríamos:

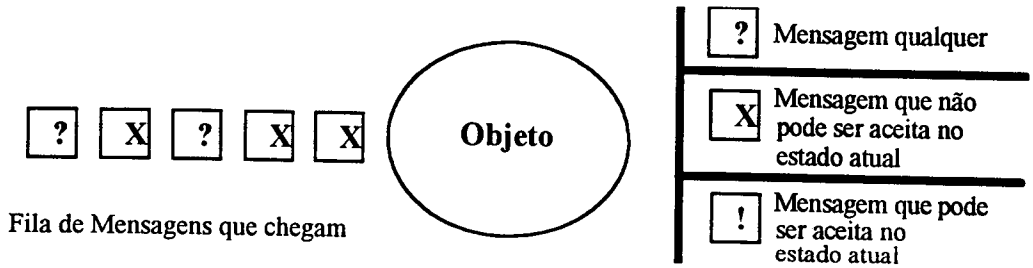

Figura 4.5 - Não-Descarte de Mensagens Impróprias em ABCL/1

Um exemplo, em ABCL/1, da implementação de um *buffer* de tamanho limitado, mostrando a construção *wait-for*, segue abaixo:

```
[object Buffer
    (state forma de representação do buffer)
    (script
          (=> [:put umElemento]
                (when buffer cheio
                (wait-for
                      (=> [:get]
                      remove o produto e retorna-o
                       )
                )
                 )
                 armazena umElemento
          )
          (=> [:get]
                (when buffer vazio
                (wait-for
                      (=> [:put  umElemento]
                      envia umElemento ao objeto que enviou o [:get]
                       )
                )
                  remova um elemento e retorne-o
                 )
          )
    )
]
```

Listagem 4.8 - Implementando um Buffer em ABCL/1

4.2.1.3 - Troca de Mensagens

As principais características envolvidas na troca de mensagens entre objetos no modelo de ABCM/1 são:

- Ponto-A-Ponto: Um objeto X só pode mandar mensagens para um objeto Y quando X passar a "conhecer" Y. Assim, não existe a noção de *broadcasting*.

- Topologia Dinâmica: A relação de "conhecimento" entre objetos é dinâmica, podendo se alterar diversas vezes durante a execução do modelo.

- Assincronismo: Um objeto X pode mandar uma mensagem para um objeto Y (que X "conheça") a qualquer momento, independente do estado em que Y se encontre.

- Chegada Garantida: Uma mensagem enviada sempre chega ao destinatário dentro de um tempo finito, e é armazenada em uma fila associada ao objeto.

- Linearidade de Chegada: Mensagens destinadas a um objeto são armazenadas em sua fila única de mensagens, por ordem de chegada. Não se assume simultaneidade na chegada de mensagens.

- Conservação de Ordem: Quando um objeto X manda duas mensagens M1 e M2 para um objeto Y, estas deverão chegar a Y na mesma ordem em que foram transmitidas por X.

- Relógio Global Inexistente: Não se assume a existência de um relógio global. De forma geral, eventos não-relacionados são considerados concorrentes.

4.2.1.3.1 - Tipos De Mensagens

Inspirados em modelos reais da comunicação humana, os tipos existentes de mensagens em ABCM/1 são: passado, presente e futuro.

- Passado (envia e segue em frente)

Caso um objeto X deseje mandar uma mensagem M para um objeto Y e continuar a execução imediatamente após o envio da mensagem, o tipo de troca de mensagem será passado. O nome deriva do fato de que quando a mensagem M causar os efeitos desejados no objeto Y, o objeto X já estará (potencialmente) executando outras atividades.

Em ABCL/1, a notação que segue é utilizada em mensagens do tipo passado.

```
[ Y <= M ]
```

Listagem 4.9 - Mensagem do Tipo Passado em ABCL/1

Este tipo de interação ocorre tipicamente quando o objeto que envia não precisa ficar bloqueado à espera de qualquer resposta por parte do objeto que recebe a mensagem. Por isso, esta forma de comunicação aumenta potencialmente o grau de concorrência de um modelo.

- Presente (envia e espera)

Mensagens deste tipo representam situações em que o objeto X que envia a mensagem precisa esperar que o objeto Y receba efetivamente a mesma e envie uma mensagem de volta a X como resposta. De certa forma, este tipo de comunicação se assemelha com a chamada de função/procedimento. Contudo, algumas diferenças são fundamentais:

i) A ativação de Y não termina após a resposta ser enviada de volta a X. Neste momento, X e Y podem continuar suas execuções simultaneamente.

ii) A resposta ao pedido não precisa ser, necessariamente, enviada pelo objeto Y. Este pode delegar a outro objeto Z a responsabilidade de enviar a mensagem-resposta.

Em ABCL/1, a notação que segue é utilizada em mensagens do tipo presente.

```
[ Y <== M ]
```

Listagem 4.10 - Mensagem do Tipo Presente em ABCL/1

Este tipo de comunicação pode ser usado em situações em que o objeto X que envia a mensagem precisa simplesmente esperar que o objeto Y a receba. Neste caso, a mensagem-resposta será, tipicamente, uma confirmação de recebimento ("*ack*"). Em outras situações, por outro lado, a mensagem-resposta propriamente dita pode interessar ao objeto X. Nestes casos, o resultado da comunicação poderá, por exemplo, ser atribuído a uma variável, como no exemplo abaixo:

```
[ x := [ Y <== M ] ]
```

Listagem 4.11 - Obtendo o Resultado de uma Mensagem Tipo Presente em ABCL/1

É preciso salientar que recursividade com trocas de mensagens do tipo presente causam *deadlock* (a exemplo do que foi mostrado em ConcurrentSmalltalk).

- Futuro (responda-me depois)

Situações em que o objeto X envia uma mensagem M para o objeto Y e necessita da resposta não imediatamente, mas em algum ponto no futuro, podem ser modeladas com trocas de mensagens do tipo futuro. Assim, o objeto X envia a mensagem M para Y e não precisa esperar pela resposta - X continua imediatamente com sua execução. Mais tarde, quando X necessitar do resultado, ele testa seu objeto especial futuro, privado, e que foi especificado no momento do envio de M. Caso o resultado tenha sido armazenado no objeto futuro, ele pode ser usado. A qualquer momento, entretanto, esta condição pode ser testada.

Em ABCL/1, a notação que segue é utilizada em mensagens do tipo futuro.

```
[ Y <= M  $ x ]
```

Listagem 4.12 - Mensagem do Tipo Futuro em ABCL/1

Neste caso, *x* denota uma variável especial chamada variável futura, que denota um objeto futuro.

4.2.1.3.2 - Modos De Mensagens

Uma vez que um objeto tenha começado a executar uma sequência de ações associadas à recepção de uma mensagem, a sequência não pode ser interrompida ou controlada por outra parte, a menos que o próprio objeto execute um comando *wait-for* e mude para o estado de esperando. Em situações reais da comunicação humana, contudo, uma pessoa pode ser interrompida, a princípio, a qualquer momento de suas atividades. O exemplo mais corriqueiro é a chamada de um telefone. As atividades podem ou não ser retomadas após a interrupção, dependendo do tipo de interrupção.

Em ABCL/1 [39], mensagens expressas são a forma de modelar tais situações. As mensagens vistas até então são, na verdade, mensagens ordinárias. Este tipo de interação tem as seguintes características:

- A condição de recepção de interrupção associada a uma mensagem expressa é especificada pelo objeto receptor da mensagem.
- As interrupções sofridas apresentam apenas um nível de prioridade. Assim, quando um objeto está executando uma sequência de ações e recebe uma mensagem expressa, este interromperá suas atividades e passará a descrever as atividades associadas à mensagem expressa. Contudo, caso chegue outra mensagem expressa enquanto o objeto está tratando a primeira, esta última não terá efeito (ou melhor, esperará na fila de mensagens, até chegar o momento de ser tratada).

O tratamento de mensagens ordinárias tem a notação vista anteriormente, conforme o exemplo a seguir:

```
(=> padrão da mensagem where restrição ... ação ... )
```

Listagem 4.13 - Tratamento de Mensagens Ordinárias em ABCL/1

Já o tratamento de mensagens expressas é denotado conforme segue:

```
(==> padrão da mensagem where restrição ... ação ... )
```

Listagem 4.14 - Tratamento de Mensagens Expressas em ABCL/1

De forma geral, em ABCL/1 pode haver um retardo desde o momento em que houve a interrupção até seu tratamento efetivo. Durante a execução de determinadas operações (por exemplo, alteração de uma variável do próprio objeto) o objeto receptor da mensagem simplesmente não deve ser interrompido. Os trechos de código onde o objeto não pode sofrer interrupção (ou melhor, tratar mensagens expressas) devem ser apontados explicitamente. Segue a notação usada:

```
(atomic ... ação ... )
```

Listagem 4.15 - Execução Atômica em ABCL/1

Em alguns casos, após o tratamento de uma mensagem expressa recebida, simplesmente não faz sentido retomar às atividades que vinham sido executadas pelo objeto (certamente pela recepção de uma mensagem ordinária). Nestes casos, no tratamento da mensagem expressa, o objeto pode notificar isso através da seguinte construção:

```
( non-resume )
```

Listagem 4.16 - Abortando Execução em Mensagens Expressas em ABCL/1

---

[39] Aqui não é em ABCM/1 , e sim em ABCL/1, devido aos problemas de provas matemáticas do modelo.

4.2.1.4 - Combinando Tipos e Modos

Para cada um dos três tipos de troca de mensagem (passado, presente e futuro), é possível ter-se os dois modos (ordinária e expressa). As possíveis combinações são denotadas como segue:

	Ordinária	Expressa
Passado	[T <= M]	[T <<= M]
Presente	[T <== M]	[T <<== M]
Futuro	[T <= M $ x]	[T << M $ x]

Tabela 4.1 - Tipos e Modos de Mensagens em ABCL/1

4.2.1.5 - Um Primeiro Exemplo

Para exemplificar o exemplo de uso de mensagens expressas, listamos a seguir uma possível modelagem de um relógio-alarme. Seu estado é definido através de duas variáveis privadas que representam, respectivamente, a pessoa a acordar e a contagem de tempo propriamente dita.

```
[object umRelogioAlarme
    (state      pessoa-a-acordar contagem)
    (script
          (=> [:comeceEAcorde Pessoa :depoisDeExpirar tempo]
                [pessoa-a-acordar := Pessoa]
                [contagem := tempo]
                (while (> contagem 0)
                      (progn
                            (consome-uma-unidade-de-tempo)
                            [contagem := (sub1 contagem) ]
                      )
                )
                [pessoa-a-acoradr <<= [:acorde ] ]
          )

          (=>> [:acorde Pessoa :depoisDeExpirar tempo]
                (non-resume)
                [Me <= [:comece-e-acorde Pessoa
                        :depoisDeExpirar tempo] ]
          )
          (=>> [:pare]
                (non-resume)
          )
    )
]
```

Listagem 4.17 - Um Relógio-Alarme em ABCL/1

Uma possível interação com este relógio-alarme poderia dar-se da seguinte forma:

```
[umRelogioAlarme <= [:comeceEAcorde P1 :depoisDeExpirar 80 ]
```

Listagem 4.18 - Ativando um Relógio-Alarme em ABCL/1

Após receber a mensagem (do tipo passado, neste caso) o relógio-alarme muda para o estado ativo, e começa a contar o tempo. Após expirar o tempo para que o alarme toque, o objeto relógio-alarme retorna ao estado dormente logo após enviar a mensagem *[:acorde]* à pessoa *P1*. Esta é mandada no modo passado para que o alarme-relógio continue sua execução, podendo receber novas mensagens. A mensagem é do tipo expressa para que a pessoa possa ser interrompida imediatamente, independente de sua atividade no momento em que ela deve ser "acordada".

As duas mensagens expressas que o relógio-alarme pode receber permitem que este pare sua atividade ou retome a contagem desde o início. A variável especial "Me" denota o próprio objeto (similar ao *self* de Smalltalk).

4.2.1.6 - Mensagens Propriamente Ditas

Até este ponto, os conteúdos das mensagens propriamente ditas não foram detalhados. Contudo, uma mensagem, em ABCL, contém informação.

De maneira geral, mensagens podem ser compostas de um único elemento ou uma sequência de *tags*, parâmetros e/ou nomes de objetos. Os *tags* são usados para distinguir os padrões das mensagens (*:put* e *:get*, no exemplo do Buffer mais acima) enquanto os parâmetros permitem definir os objetos que são trocados nas interações do sistema (*umElemento* , no caso do mesmo objeto *Buffer* citado). Nomes de objetos, por outro lado, são usados para diferentes propósitos. Um caso típico é quando um objeto X envia uma mensagem M qualquer a um objeto Y e deseja que Y envie o resultado a um terceiro objeto, Z. Neste caso, X inclui na mensagem M o nome de Z, para que Y possa referenciar Z ao responder à mensagem M.

Além destas informações, que se tornam claramente visíveis, assume-se ainda que uma mensagem possa carregar dois outros tipos de informação. Uma delas é o nome do objeto que envia a mensagem. Assim, quando uma mensagem enviada por X é recebida por Y, aquele (X) passará a ser "conhecido" por este (Y). Um objeto pode, então, decidir se aceita ou não uma mensagem baseado no objeto que a enviou [40]. Ou então, alternativamente, responder com diferentes resultados, dependendo do objeto que enviou a mensagem.

O outro tipo de informação que uma mensagem pode carregar é o destinatário da resposta. Quando um objeto Y recebe uma mensagem M do tipo presente ou futuro, vinda de X, ele deve responder com o resultado da computação do *script* associado a M. Uma vez que o destinatário desta resposta (neste caso, X) é conhecido por Y, este pode optar por solicitar a outro objeto Z que este calcule o resultado para a mensagem M e que o próprio Z retorne o resultado a X [41]. A sintaxe, para referenciar este tipo de informação, é apresentada em 4.2.3.

A presença destas duas informações durante a troca de mensagens adiciona, como se viu, um maior poder de expressão a ABCL. A forma de referenciar tais informações em um trecho de código ABCL/1 será visto mais adiante.

---

[40] "&sender" denota, em ABCL/1, o objeto que enviou a mensagem.

[41] Este é o mecanismo de delegação.

4.2.2 - A Linguagem ABCL/1

Embora neste estudo não nos interessem diretamente aspectos sintáticos das linguagens de programação associadas aos modelos de computação aqui apresentados, é justamente através de exemplos que podemos analisar características importantes de tais modelos. Descreveremos agora, de forma resumida, princípios por trás do projeto de ABCL/1 , ilustrando com exemplos as possibilidades de ABCM/1 descritas anteriormente.

4.2.2.1 - Princípios de Projeto

ABCL/1 apresenta dois princípios básicos em seu projeto:

- Semântica Definida para Troca de Mensagens: Deve ser transparente e factível no modelo computacional que a sustenta (no caso, ABCM/1)

- Praticidade: Nem todas as abstrações existentes em um modelo necessitam ser modeladas através de objetos e trocas de mensagens [42]. Estruturas de controle (*if-then-else* , etc) e algumas estruturas de dados (e operações de manipulação associadas) podem estar presentes na linguagem sem violar princípios de modelagem, segundo os projetistas de ABCL/1.

Por considerar o paralelismo um fenômeno complexo de se modelar/resolver em linguagens de computação, optou-se por uma abordagem mais conservadora no que tange a representação interna de cada objeto paralelo[43].

4.2.2.2 - Definindo e Criando Objetos

A definição de um novo objeto em ABCL/1 é feita conforme listado em 4.2.1.1. É na definição que se pode estabelecer a forma de inicialização das variáveis privadas do objeto.

A criação de objetos, na verdade, é realizada através do trecho de código de sua definição. Assim, o trecho de definição não define uma classe de objetos que podem ser instanciados, como em Smalltalk, mas sim uma instância de objeto propriamente dita. Por exemplo, a definição abaixo

```
[object  umSemaforo
    (state [contador := 1]
           [filaDeProcessos := [CriaFila <== [:nova]]] )
    (script
          ( => [:P]  ...ações associadas à primitiva P ...)
          ( => [:V]  ...ações associadas à primitiva V ...)
    )
]
```

Listagem 4.19 - Uma Implementação de Semáforo em ABCL/1

define (uma instância de) um semáforo, o qual tem suas variáveis privadas (contador e fileDeProcessos) devidamente inicializadas.

---

[42] Note o choque violento com o que é adotado em Smalltalk, por exemplo.

[43] A maioria das funçoes de manipulação são operações Lisp, onde ABCL/1 foi inicialmente implementado.

Note-se a criação da fila de processos (variável privada do semáforo). Ela é feita através do envio de uma mensagem do tipo presente ou futuro para o objeto *CriaFila*, o qual não é listado aqui. Assim, enquanto no modelo de objetos tradicional as classes são objetos responsáveis por criar instâncias, aqui em ABCL/1 este "objeto" criador é análogo a um outro qualquer. Assim, o objeto que cria e retorna diferentes semáforos poderia, por exemplo, ser definido por:

```
[object  CriaSemaforo
 (script
    (=> [:cria] !  [object umSemaforo
      (state [contador := 1]
             [filaDeProcessos := [CriaFila <== [:nova]]] )
         (script
               ( => [:P]
                    ...ações associadas à primitiva P ...)
               ( => [:V]
                    ...ações associadas à primitiva V ...)
         )
      ]
```

Listagem 4.20 - Criando Diferentes Instâncias em ABCL/1

Note que "!" denota o fato de ser retornado um determinado resultado como resposta ao envio de uma mensagem. Neste caso, o resultado que será retornado ao objeto que solicitou a criação do semáforo será justamente uma nova instância (de semáforo).

4.2.2.3 - Mensagens de Resposta

No exemplo anterior, de criação de objetos, utilizou-se a notação "!" para denotar a ação de retornar uma resposta ao objeto que enviou determinada mensagem. Na verdade, esta forma é uma abreviação da descrição mais explícita, mostrada abaixo.

```
( => padrão-do-pedido @ destino ...
                ...[destino <= expressão-resposta] ...)
```

Listagem 4.21 - Retornando Objetos em ABCL/1

Esta é a forma de referenciar uma das informações intrínsecas de uma mensagem (veja 4.2.1.6) , mais especificamente o objeto para o qual a mensagem-resposta será enviada.

Para um objeto X mandar uma mensagem M a Y e redirecionar a resposta para Z, a forma, em ABCL/1, seria:

```
[ Y <= mensagem-M @ Z ]
```

Listagem 4.22 - Redirecionando Mensagens em ABCL/1

4.2.2.4 - Paralelismo e Sincronização

Por se basear em ABCM/1 como modelo de computação, ABCL/1 apresenta mecanismos para expressar paralelismo e sincronização. São eles:

- Paralelismo

☒ Ativação Concorrente de Múltiplos Objetos

i) Mesmo quando as mensagens são transmitidas sequencialmente, caso os destinatários sejam distintos, as ações associadas ocorrem em paralelo.

ii) Mensagens podem ser enviadas simultaneamente por um objeto. Em ABCL/1 , isto é denotado pela seguinte construção paralela:

```
{ envio-de-mensagem ... envio-de-mensagem ... }
```

Listagem 4.23 - Envio Simultâneo de Mensagens em ABCL/1

i) Paralelismo entre as ações de X mandar uma mensagem M a Y e de receber a resposta associada a M (por parte do próprio X ou de outro objeto Z). Estas atividades ocorrem em paralelo.

☒ Multicast no modo passado e futuro. ABCL/1 permite que uma mesma mensagem seja mandada para uma lista de objetos, conforme segue:

```
[ lista-de-objetos  <= mensagem ]
```

Listagem 4.24 - Multicast de Mensagens em ABCL/1

- Sincronização

1) Um de Cada Vez: Um objeto sempre executa uma única sequência de ações como resposta a uma determinada mensagem aceita. Não executa mais que uma ação simultaneamente.

2) Modo Espera: Através da construção *(wait-for)* um objeto pode sincronizar suas atividades, esperando por outra mensagem, etc.

3) Tipo Presente e Futuro: No tipo presente, a ativação do objeto que envia a mensagem é suspensa até que a resposta associada seja retornada. No tipo futuro, quando o objeto que enviou a mensagem acessa a variável futuro e o valor ainda não chegou, este objeto deverá aguardar até que o resultado chegue.

4) Construções Paralelas: Através das construções acima, modelam-se sincronismos derivantes destas construções.

4.2.3 - Modelando um Problema em ABCL/1

Para demonstrar a naturalidade e poder de expressão de ABCL/1, mostraremos um exemplo de um conjunto de objetos que modelam uma situação (potencialmente) real. Modelaremos um gerente que, encarregado de resolver determinado problema, aloca uma equipe para conseguir alcançar sua meta. Esta equipe é composta de diversas pessoas engajadas na solução do problema, onde uma delas é considerada coordenadora, devendo reportar os resultados das atividades ao gerente. Esquematicamente, temos [Yonezawa90b]:

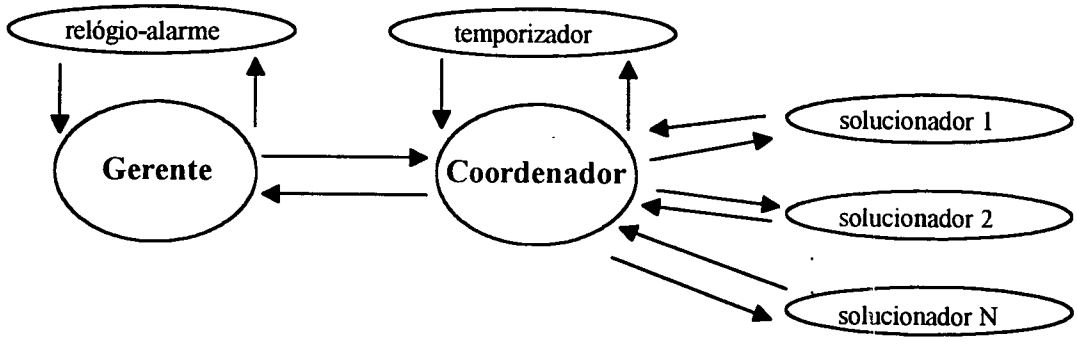

Figura 4.6 - Um Sistema em ABCL/1

Temos, no esquema acima, dois componentes adicionais: um alarme que notifica o gerente quando o tempo da equipe para solucionar o problema expirou, e outro que notifica o próprio coordenador da equipe.

Modelaremos a situação de tal forma que todos os trabalhadores tentem a solução do problema de forma independente e paralela. Ao encontrar a solução, o trabalhador manda-a imediatamente ao coordenador (o qual também tenta achar a solução). Caso várias soluções sejam encontradas dentro do prazo, o coordenador escolherá a melhor, passando-a ao gerente. Caso nenhuma solução tenha sido encontrada, o coordenador pede ao gerente para que o prazo seja prorrogado. Caso o problema não seja encontrado nem no tempo prorrogado, o coordenador avisa que não encontrou a solução e termina.

O relógio-alarme é, basicamente, o mesmo descrito em 4.2.1.5. A exemplo do semáforo, apresentado em 4.2.2.2, temos, além do relógio-alarme, o objeto responsável por criar e retornar (instanciar) relógios-alarmes [44]. Sua definição é dada por:

[44] Isso é necessário sempre que forem haver mais que uma instância de um mesmo "tipo" de objeto. No caso, dois relógios-alarme: um para o gerente, outro para o coordenador.

```
[object  CriaRelogioAlarme
 (script
    (=> [:cria]  !  [object umRelogioAlarme
                (state      pessoa-a-acordar contagem)
                (script
                      (=> [:comeceEAcorde Pessoa
                           :depoisDeExpirar tempo]
                          [pessoa-a-acordar := Pessoa]
                          [contagem := tempo]
                          (while (> contagem 0)
                               (progn
                                 (consome-uma-unidade-de-tempo)
                                 [contagem := (sub1 contagem) ]
                               )
                          )
                          [pessoa-a-acoradr <<= [:acorde ] ]
                       )
                      (=>> [:acorde Pessoa :depoisDeExpirar tempo]
                         (non-resume)
                        [Me <= [:comece-e-acorde Pessoa
                                 :depoisDeExpirar tempo] ]
                      )
                      (=>> [:pare]
                                 (non-resume)
                      )
                      (=>> [:quantoFalta]
                                 ! contagem
                      )
                )
                ]
    )
 )
```

Listagem 4.25 - Criador de Relógios-Alarmes em ABCL/1

Cada um dos trabalhadores que participará na solução do problema precisa ser criado também. Assim, precisamos da definição do objeto trabalhador e do objeto que retorna um novo trabalhador[45], conforme segue:

[45] De forma análoga a semáforos e relógios-alarme.

```
[object  CriaTrabalhador
 (script
    (=> [:criaComEstrategia EstrategiaDesejada]

          !  [object umTrabalhador
               (state [progressoNaSolucao := nil] )
               (script
                ( => [:resolva Problema]
                      ...
                      (while (naoResolveu progressoNaSolucao )
                      do ...
                           Tenta resolver o Problema segundo uma
                            EstrategiaDesejada , armazenado o estado
                            da resolução em progressoNaSolucao
                      )
                      ! progressoNaSolucao
                )
                ( =>> [:pare]      (suicide) )
                ( =>> [:estadoDaSolucao ]   !  progressoNaSolucao  )
              )
            ]
    )
 )
```

Listagem 4.26 - Criador de Trabalhadores em ABCL/1

Vemos que, uma vez criado para solucionar um determinado problema segundo determinada estratégia, este trabalhador inicia sua tarefa. A qualquer momento, contudo, ele poderá ser questionado acerca do andamento da busca pela solução ou então solicitado a parar, quando o objeto termina (*suicide*).

Falta-nos definir o gerente e o coordenador. Por existir apenas um gerente, podemos defini-lo diretamente, sem a necessidade de definir um objeto responsável por sua criação (conforme exemplos acima) :

```
[object  Gerente
  (state [alarme := CriaRelogioAlarme <== [:cria]]
         coordenador )
  (script
   ( => [:comeceProjeto  EspecificacaoIncompleta]
        (temporary spec listaDeEstrategias
         umSolucionadior intervalo)
        [spec := (constroiEspecificacaoAPartirDe
                          EspecificacaoIncompleta]
        [listaDeStrategias := (constroiListaDeEstrategiasPara spec)]
        [intervalo := (estimaIntervaloNecessarioPara spec)]
        [coordenador := [CriaCoordenador <== [:problema spec
                                              relatePra: Me] ] ]
        (while (naoVazia listaDeEstrategias) ; forma uma equipe
        do
         [umSolucionadior := [ CriaTrabalhador <==
                   [:criaComEstrategia (primeiroDe
                                        listaDeEstrategias) ] ] ]
         [coordenador <= [:adicioneTrabalhador umSolucionador] ]
         [listaDeEstrategias := (restoDe listaDeStrategias)]
        )
        [coordenador <= [:comeceAResolverNoPrazoDe intervalo]]
        [alarme <= [:comeceEAcorde Me
                    :depoisDeExpirar (+ intervalo 100)]]
        ... continua em outra atividade de gerenciamento ...
   )
   ( =>> [:podesExterderPrazo?]          ; enviada pelo coordenador
         (temporary [tempoQueFalta := (- [alarme <<== [:quantoFalta]]
                                         95]  )
         (if (> tempoQueFalta 5) then
               ! [:simComPrazo tempoQueFalta ]
               [alarme <<= [:comeceEAcorde Me
                           :depoisDeExpirar (+ (tempoQueFalta  20))]]
         else
               ! [:semProgorracao]
                 [alarme <<= [:pare] ]
         )
   )
   ( =>> [:encontrei umaSolucao]         ; enviada pelo coordenador
         ...processa a solução do problema ...
   )
   ( =>> [:naoEncontreiSolucao]          ; enviada pelo coordenador
         ...executa ações associadas ao fracasso da operação ...
   )
         ( =>> [:acorde]           ; enviada pelo relogio-alarme
               [coordenador <<= [:voceEstaMuitoAtrasado] ]
         )
   )
]
```

Listagem 4.27 - Gerente de Equipe em ABCL/1

Embora só haja um coordenador, optou-se, no código do *Gerente*, por criar um coordenador, com parâmetros. Assim, precisamos definir o que é um coordenador bem como o objeto responsável por criá-lo, conforme segue:

```
[object  CriaCoordenador
 (script
    (=> [:problema Especificacao relatePra: MeuGerente]
          !  [object umCoordenador
               (state [membrosDaEquipe := nil]
                      [alarme := [CriaRelogioAlarme <== [:cria]]]
                                 [melhorSolucao := nil] Solucoes)
               (script
                ( => [:adicioneTrabalhador umTrabalhador]
                      [membrosDaEquipe := (cons membrosDaEquipe
                                                umTrabalhador)  ]
                )
                ( => [:comeceAResolverNoPrazoDe prazo]
                      (temporary minhaSolucao)
                      [relogio <= [:comeceEAcorde Me
                                   :depoisDeExpirar (- prazo 20)]]
                      [membrosDaEquipe <= [:resolva spec] $ Solucoes]
                                    ; multicast tipo futuro
                      (while (and (not (ready? Solucoes))
                                  (null minhaSolucao))
                       do
                           ... tenta resolver o problema
                           com minha propria estrategia
                       )
                     (atomic
                     [melhorSolucao := (escolha-melhor minhaSolucao
                                        (all-values Solucoes))]
                     [MeuGerente <<= [:encontrei melhorSolucao]]
                     [alarme <<= [:pare]]
                     [membrosDaEquipe <<= [:pare]] ; multicast
                                                     expressa
                     ); atomic
                )
                ( =>> [:acorde]                        ; do alarme
                      (if (null melhorSolucao) then
                      (case [MeuGerente <<== [:podesExterderPrazo?]]
                        (is [:simComPrazo prazo ]
                             [relogio <<= [:acorde Me
                                           :depoisDeExpirar prazo] ] )
                        (is [:semProrrogação ]
                             [membrosDaEquipe <<= [:pare] ]
                            (suicide) )
                      ); case
                      ); if
                )
                ( =>> [:voceEstaMuitoAtrasado]
                      (if (null melhorSolucao) then
                     [alarme <<= [:pare]]
                     [membrosDaEquipe <<= [:pare] ]
                     [MeuGerente <= [:naoEncontreiSolucao] ]
                     (suicide)
                )
                 )
             ]
    )
 )]
```

Listagem 4.28 - Criador de Coordenador em ABCL/1

Conforme o exemplo anterior, podemos observar que ABCL provê mecanismos intuitivos de modelagem de situações reais.

4.2.4 - Considerações

ABCL, além de buscar a unificação de objetos com processos, adiciona alguns conceitos interessantes para modelagem de problemas. A possibilidade de interromper uma computação para tratamento de uma situação prioritária faz de ABCL uma poderosa ferramenta para expressão de situações reais.

Por outro lado, ABCL não apresenta facilidades para o refinamento de abstrações através de herança. O mecanismo de delegação de mensagens permite, entretanto, uma abordagem baseada em protótipos, e não em classes.

Mesmo assim, ABCL se baseia em tipos de dados presentes na linguagem em que foi implementada inicialmente (Lisp). Notamos, então, que mesmo em linguagens concorrentes orientadas a objetos surge a necessidade de expressar entidades passivas, fornecendo apenas um mecanismo de abstração. ABCL baseia-se nos tipos providos por Lisp, não podendo extendê-los de forma apropriada [Meyer88] como Smalltalk, por exemplo.

A maioria dos modelos de programação concorrente orientada a objetos baseia-se, contudo, no envio de mensagens, com identificação explícita do objeto receptor da mesma. Acreditamos que esta perspectiva é limitante, conforme detalharemos mais adiante. Na busca de um modelo mais flexível é que decorrem os próximos capítulos de nosso trabalho.

5 - Linda

Um conjunto de primitivas que englobam o modelo de programação paralela seguindo o modelo de Espaço de Tuplas. Esta é, basicamente, a proposta de Linda [Gelernter85] . Adicionando este conjunto de primitivas a uma linguagem X de programação, tem-se um dialeto de X com suporte a programação paralela.

5.1 - O Modelo

Linda é um modelo de criação e coordenação de processos ortogonal à linguagem X de programação à qual o modelo é incorporado. Para o modelo não interessa como os vários *threads* de execução computam suas operações, mas sim como eles são criados e como eles podem ser organizados em um programa coerente.

Caso dois processos precisem se comunicar, eles não trocam mensagens ou compartilham uma variável. Processos que produzem um determinado dado relevante (chamado de tupla na nomenclatura Linda) simplesmente colocam-no à disposição em uma região chamada Espaço de Tuplas. Caso um processo deseje ter acesso ao dado, precisa apenas consultar o Espaço de Tuplas à procura da tupla desejada. Criação de processos é tratada da mesma forma: um processo que queira disparar outro *thread* paralelo de execução simplesmente gera uma tupla viva. Esta tupla consiste de uma série de operações a serem executadas (definidas na linguagem X na qual o modelo foi incorporado) e que no final produz uma tupla ordinária, conforme descrito anteriormente. Esquematicamente, o Espaço de Tuplas pode ser representado pela figura a seguir.

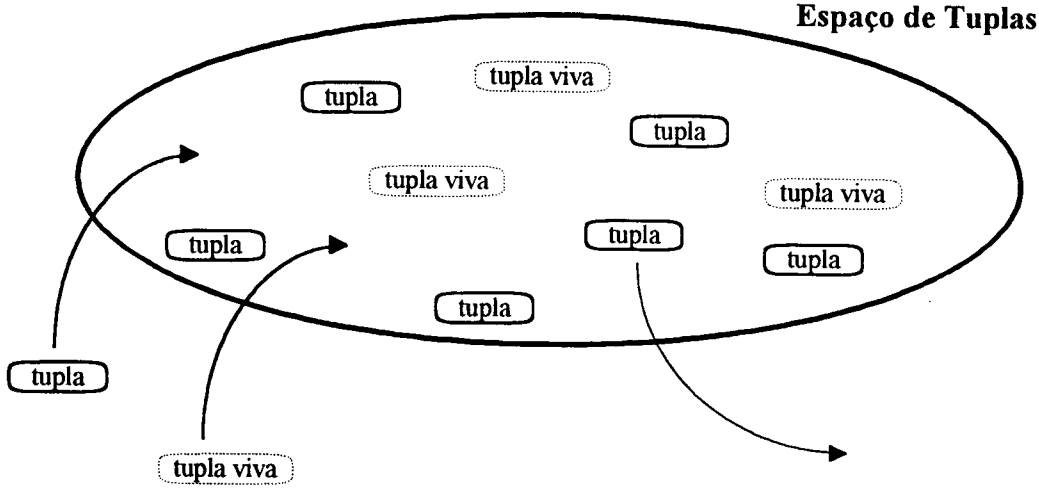

Figura 5.1 - Espaço de Tuplas de Linda

Uma implicação importante desta abordagem é que criação e comunicação entre processos são, simplesmente, duas faces da mesma moeda. Esta unificação permite organizar um conjunto de processos da mesma forma (tuplas vivas). O fato de cada tupla viva tornar-se uma tupla ordinária permite abranger outros modelos como:

- Troca de mensagens
- Estruturas de dados distribuídas
- Estruturas de dados ativas

Desta forma, Linda permite modelar situações cuja granularidade de paralelismo pode ser pequena, média ou grande.

5.2 - Tuplas

Antes de definir as primitivas do modelo e o Espaço de Tuplas em si, é preciso descrever o que são tuplas no modelo Linda.

De forma geral, uma tupla é uma coleção ordenada de objetos da linguagem X que incorpora o modelo. Caso X seja Modula-2, por exemplo, tais objetos são INTEGERs , ARRAYs, RECORDs , etc. Em outras palavras, instâncias dos tipos definidos na linguagem ou combinações destes. Um exemplo de tupla seria:

```
("João" , 23 , 92.5 )
```

Listagem 5.1 - Tupla em Linda

Neste caso, a tupla é uma coleção ordenada onde o primeiro elemento é um *string*, o segundo elemento é um *número natural*, e o terceiro elemento é um *número de ponto flutuante*.

Como o modelo se integra à linguagem, os elementos das tuplas podem ser descritos por variáveis também. Por exemplo, vamos supor o caso de um Modula-2-Linda:

```
VAR
   idade  : CARDINAL;
   peso   : REAL;
   nome   : STRING;

....
nome := "João";
idade := 23;
peso := 92.5;
...

(nome, idade,peso)
```

Listagem 5.2 - Elementos de Tuplas em Linda

Esta última tupla é análoga à apresentada anteriormente. Nesta, contudo, as variáveis denotam os objetos propriamente ditos.

5.2.1 - Parâmetros Reais

Os exemplos anteriores mostram tuplas contendo os chamados parâmetros reais. São objetos da própria linguagem, conforme apresentado. Outra forma de parâmetro são os formais.

5.2.2 - Parâmetros Formais

Além de parâmetros reais, outro tipo de parâmetro que pode fazer parte de uma tupla é o chamado parâmetro formal. Ele denota meta-objetos, ou seja, informação adicional sobre os objetos. Segue um abaixo:

```
VAR
   idade   : CARDINAL;
   ....

("João", ? idade, 92.5)
```

Listagem 5.3 - Parâmetros Formais em Linda

Aqui, esta última tupla apresentada contém como segundo elemento um parâmetro formal. Ele denota na verdade a informação de tipagem da variável *idade*. Sua aplicação torna-se clara ao apresentarmos as primitivas do modelo Linda, a seguir.

5.3 - Primitivas do Modelo Linda

Resumidamente, as primitivas deste modelo são para:

- Adicionar uma tupla ordinária no espaço de tuplas
- Retirar uma tupla ordinária do espaço de tuplas
- Ler (sem retirar) uma tupla ordinária do espaço de tuplas
- Adicionar uma tupla viva ao espaço de tuplas

5.3.1 - Adicionando Tuplas Ordinárias

A primitiva *out*, em Linda, permite adicionar uma tupla ao Espaço de Tuplas. Assim,

```
out ("João", 23, 92.5)
```

Listagem 5.4 - Primitiva out em Linda

tem o efeito de adicionar a tupla de três elementos ao Espaço de Tuplas. Esta tupla poderá ser retirada ou lida posteriormente, por um processo. Note que esta primitiva é atômica em relação ao Espaço de Tuplas e não bloqueante em relação ao processo que evoca a primitiva *out*. O processo continua sua execução imediatamente.

5.3.2 - Retirando Tuplas Ordinárias

Simétrica à primitiva *out*, a primitiva *in* permite retirar uma tupla do Espaço de Tuplas. Aqui, contudo, surgem características importantes de Linda. Para se definir a retirada de uma tupla, especifica-se, na verdade, um padrão de tupla que será pesquisado no Espaço de Tuplas. É aqui que surge a utilização de parâmetros formais descritos em 5.2. Por exemplo:

```
VAR
   idade  : CARDINAL;
   nome   : STRING;
  ....

in (? nome, ? idade, 92.5)
```

Listagem 5.5 - Primitiva in em Linda

Neste caso, o processo será bloqueado até que haja, no espaço de tuplas, uma tupla ordinária que case com a tupla acima. Este casamento deve atender uma série de restrições:

i) As tuplas deverão ter a mesma aridade (mesmo número de elementos)

ii) Todos os seus elementos devem *casar*

É preciso, então, descrever quando um elemento de uma tupla ordiária do espaço de tuplas *casa* com um elemento de uma tupla usada na primitiva *in*. Chamando estes dois elementos (parâmetros) de X e Y, respectivamente, temos:

a) Se X e Y são parâmetros reais, ambos devem conter o mesmo valor [46] .

b) Se X é um parâmetro real e Y é um parâmetro formal, X e Y casam somente se X atende à restrição estrutural descrita por Y.

5.3.2.1 - Ilustrando com Parâmetros Reais

Como exemplo do que foi descrito em *a)* , temos que:

```
out ("Continue")
```

por parte de um processo A adicionará esta tupla de um elemento apenas no espaço de tuplas, e que

```
in ("Continue")
```

quando executado por outro processo, B , causará um *casamento* das tuplas. Assim, a primeira tupla será retirada do espaço de tuplas e o processo B continuará em seguida. Caso B evocasse a primitiva *in* antes de A colocar a tupla original através da primitiva *out*, B ficaria bloqueado até que A adicionasse a referida tupla.

---

[46] Aqui utiliza-se semântica de valor. Ser igual significa que as posições de memória alocadas aos objetos contêm os mesmos valores.

Podemos notar que tuplas contendo apenas parâmetros reais servem exclusivamente para sincronizar processos. Aqui, nota-se que não há troca de informação substancial entre os processos A e B, exceto o próprio sincronismo que decorre destas primitivas.

5.3.2.2 - Ilustrando com Parâmetros Formais

Por outro lado, para ilustrar *b)* descrito anteriormente vamos considerar o exemplo:

```
out ("João")
```

Conforme descrito anteriormente, esta tupla de apenas um elemento será adicionada no espaço de tuplas por parte do processo A que evocou *out*.

Um outro processo B, ao evocar *in* conforme segue:

```
VAR
   nome   : STRING;
   ....

in (? nome)
print (nome)
```

Listagem 5.6 - Casamento de Tuplas com Parâmetros Formais em Linda

fará com que o espaço de tuplas seja vasculhado à procura de uma tupla que *case* com a tupla acima. Neste caso, haverá um casamento pois o parâmetro formal *?nome* especifica que quelquer instância de *string* deverá *casar* com este formal. Assim, as duas tuplas *casam*. Aqui, contudo, há troca de informação. Além de ser retirada do espaço de tuplas, a tupla original terá seu parâmetro real transferido para o respectivo parâmetro formal da tupla envolvida na primitiva *in*. Assim, no nosso exemplo acima, *nome* passará a denotar o valor "João", e este será passado à rotina *print* seguinte. Neste caso, novamente, caso *in* fosse evocado por B antes de A adicionar a primeira tupla, B ficaria bloqueado até que uma tupla que casasse a tupla dada pelo *in* fosse adicionada ao espaço de tuplas.

Aqui, notamos que parâmetros formais permitem que os processos não só sincronizem atividades, mas também troquem informações entre si. Note-se que tais informações são descritas na linguagem X em que o modelo foi incorporado. Por isso Linda se preocupa apenas com a coordenação entre os processos. As estruturas de dados propriamente ditas que os processos trocam entre si devem ser expressos na linguagem X que implementa as extensões Linda.

Podemos, é claro combinar na primitiva *in* parâmetros reais e formais. Assim, no exemplo:

```
VAR
   idade   : CARDINAL;
   peso    : REAL;
...
in ("João" , ? idade , ?peso )
print (idade, peso)
```

Listagem 5.7 - Casamento de Tuplas com Parâmetros Reais e Formais em Linda

muitas tuplas que estão no espaço de tuplas poderão *casar* com esta. São exemplos:

```
("João" , 23 , 67.9 )
("João" , 44 , 89.5 )
("João" , 17 , 60.3 )
```

Caso várias tuplas presentes no espaço de tuplas possam casar com a tupla descrita acima na primitiva *in*, uma delas será escolhida. Esta escolha não obedece nenhum critério, não possuindo nenhuma relação com a ordem de inserção das tuplas. Assim, o espaço de tuplas é uma coleção de tuplas sem noção de ordem. É similar a uma memória associativa, onde as tuplas são procuradas baseadas em seus conteúdos. Isso diferencia o modelo Linda de modelos que implementam uma memória global. Numa memória, os elementos são identificados de acordo com sua posição, ou endereço. Em Linda, contudo, eles (tuplas) são identificados única e exclusivamente por seu conteúdo. Isso permite transparência de localização, favorecendo a implementação distribuída do espaço de tuplas sem que os processos envolvidos necessitem conhecer a localização "física" das tuplas no espaço de tuplas.

5.3.3 -Lendo Tuplas Ordinárias

A primitiva *rd* em Linda tem quase o mesmo comportamento que a primitiva *in*. A única diferença, contudo, é que a tupla original que foi adicionada ao espaço de tuplas através de uma primitiva *out* (por exemplo) não é retirada do espaço de tuplas.

Desta forma, uma mesma tupla produzida por um processo A pode ser lida por potencialmente N processos, até ser consumida do espaço de tuplas através da primitiva *in*.

5.3.4 - Adicionando Tuplas Vivas

Uma tupla viva, conforme anteriormente dito, é uma sequência de comandos da linguagem X a serem executados e que se transformam em uma tupla ordinária. Neste ponto, esta tupla ordinária é adicionada ao espaço de tuplas com semântica idêntica ao *out* descrito anteriormente. Tuplas vivas não casam com nenhuma forma de tupla descrita por *in*. Apenas quando uma tupla viva termina sua sequência de computação é que ela poderá ser *casada* com outra.

Em Linda, a forma de expressar uma tupla viva é dada por:

```
eval ( sqrt (225) )
```

Listagem 5.8 - Expressando Tuplas Vivas em Linda

Neste caso, uma tupla viva será criada e adicionada ao espaço de tuplas. Ela representará um processo que calculará a raiz quadrada (sqrt) do número 225. Após terminar de computar o valor, a tupla acima tranforma-se na tupla ordinária dada por:

```
(15.0)
```

e então será adicionada ao espaço de tuplas.

Aqui, também, pode-se ter uma tupla com diversos parâmetros. Assim, o código

```
eval ( sqrt (225) , seno (90) , cosseno (90) )
```

Criará uma tupla viva que computará a raiz quadrada (sqrt) de 255, o seno do ângulo 90 e o cosseno do ângulo 90. Tais computações dão-se, potencialmente, de forma paralela. Ao terminar todas estas computações, será adicionado no espaço de tuplas a seguinte tupla:

```
(15.0 , 1.0 , 0.0 )
```

Note-se também que o modelo não especifica onde tais computações são efetuadas. Assim, em uma implementação distribuída de Linda tais cálculos poderiam ser executados em máquinas distintas, sendo posteriormente reunidos e passando a integrar a tupla final descrita acima.

Com a primitiva *eval* define-se não só a criação de processos, mas também uma forma de sincronização para as computações definidas pelos vários parâmetros da tupla viva.

5.3.5 - Variações

Algumas variações na proposta Linda definem ainda duas outras primitivas não-bloqueantes. São elas:

- Retirar (caso haja) uma tupla ordinária do espaço de tuplas : *inp*
- Ler (caso haja) uma tupla ordinária do espaço de tuplas: *rdp*

Estas primitivas têm, respectivamente, semântica quase igual às suas versões não-bloqueantes (*in* e *rd*, respectivamente). A única diferença é que o processo que as evoca não será bloqueado de forma alguma, independente de haver ou não, no espaço de tuplas, tupla para *casamento*.

5.4 - Modelando Problemas em Linda

Cada modelo de programação paralela possui peculiaridades que, de certa forma, direcionam a forma de modelagem de problemas. Juntamente com o modelo Linda, seus autores também propõem mecanismos para modelagem de programas paralelos em dialetos Linda [Carriero89b].

A grosso modo, segundo [Carriero89b], programas paralelos podem ser codificados de acordo com três princípios gerais: paralelismo de resultado, paralelismo de agenda e paralelismo de especialista. De acordo com o princípio, emprega-se um método diferente de programação: estruturas de dados vivas, estruturas de dados distribuídas, e troca de mensagens, respectivamente.

Desta forma, para modelar um determinado problema em Linda, escolhe-se o método associado à classe conceitual do problema. Caso esta implementação mostre-se inadequada e/ou ineficiente, transforma-se esta implementação, de forma metódica, para outra versão (método).

5.4.1 - Resultado, Agenda e Especialista

Para tornar clara a diferença entre estas principais classes de problemas paralelos, conforme citadas anteriormente, ilustraremos com um problema de conhecimento geral: a construção de uma casa.

5.4.1.1 - Paralelismo de Resultado

Começa-se considerando o produto final, nesta classe de problemas. No nosso exemplo, uma casa. O resultado (casa) pode ser dividido em uma série de componentes: parede, teto, fundação, telhado, etc. Desta forma, inicia-se a construção da casa a partir da construção paralela de todos seus componentes. Ou seja, a cada trabalhador é atribuído um pedaço do resultado final, e estes trabalham independentemente, em paralelo, de acordo com as restrições naturais do problema. A construção de uma casa desta forma se encaixa mais naturalmente com casas pré-fabricadas, onde os componentes são feitos independentemente e então montados em um produto final.

5.4.1.2 - Paralelismo de Agenda

Nesta forma de paralelismo, pensa-se no problema em termos de uma agenda de atividades que precisam ser realizadas. No nosso exemplo, as atividades necessárias para construção da casa. Assim, descreve-se uma agenda seqüencial de atividades onde, para cada atividade, recrutam-se diversos trabalhadores. Assim, os trabalhadores não são especialistas, mas sim capazes de realizar quaiquer das tarefas listadas na agenda. A construção de uma casa desta forma se encaixa mais naturalmente com o esquema de mutirão, onde diversas pessoas de engajam nas diversas atividades sequencialmente, até a construção da(s) casa(s).

5.4.1.3 - Paralelismo de Especialista

Aqui, pensa-se no problema como sendo uma série de trabalhadores que executarão a tarefa. Para a construção de nossa casa, precisamos de trabalhadores com diferentes habilidades: carpinteiro, marceneiro, arquiteto, engenheiro, etc. Assim, montamos uma equipe de trabalhadores, onde cada qual é especialista em determinada tarefa. Todos começam simultaneamente; porém, baseados nas dependências das atividades, muitos deles necessitarão esperar uns pelos outros na fase inicial da solução. Este classe, então, mostra-se mais adequada em tarefas com características de *pipelining*, ou seja, tarefas que requerem a produção ou transformação de uma série de objetos idênticos. A construção de uma casa desta forma se encaixa mais naturalmente com a construção tradicional, onde especialistas são contratados para construção da casa, desde o arquiteto até o marceneiro, uns podendo ser (sub)contratados pelos outros.

Aqui, ilustramos as três classes com um mesmo problema. Em alguns problemas, contudo, a distinção das três pode não se tornar tão clara. Em outros casos, a modelagem do problema em algumas das classes poderá, inclusive, mostrar-se inapropriada.

Trazendo as classes mais perto das linguagens de programação, podemos notar que podemos modelar um aplicação paralela de formas distintas:

a) Planejá-la em torno de suas estruturas de dados, enquanto resultado final. Então, tornamos o modelo paralelo com a computação paralela dos elemtação paralela dos elementos desta

b) Planejar a aplicação em torno de uma agenda de atividades e então associar diversos processos para cada passo da agenda.

c) Planejar a aplicação em torno da combinação de diversos processos especializados, conectados entre si através de uma rede lógica. O paralelismo decorrerá dos nodos desta rede lógica (especialistas) interagindo concorrentemente.

5.4.2 - Identificando o Paralelismo do Problema

Para cada uma das três classes principais de paralelismo descritas anteriormente, propriedades existem que nos permitem identificar, diante de um problema, a mais natural de se usar como forma de modelagem.

- Em alguns casos, a melhor forma de se projetar um programa paralelo é pensar diretamente na estrutura de dado final (paralelismo de resultado). Exemplos típicos são aqueles onde se deseja produzir uma estrutura de dados de vários elementos (de forma geral, uma coleção). Nestes casos, normalmente é possível especificar exatamente como cada elemento da estrutura resultante depende dos demais elementos e dos dados de entrada. De forma geral este tipo de solução pode ser descrita como: "*Construir uma estrutura de dados X, determinando todos os valores dos componentes desta estrutura de acordo com uma sequência Y de computação. Terminar quando todos tiverem sido calculados*". Em alguns casos, todos os elementos de X podem ser calculados independentemente. Em outros, alguns valores dependerão de computações intermediárias. Um exemplo simples e ilustrativo de paralelismo de resultado é o cálculo da soma de duas matrizes.

- Paralelismo de agenda, por outro lado, envolve uma transformação, ou uma série de transformações, a serem aplicadas a todos elementos de um conjunto, de forma paralela. O modelo mais natural para esta forma de paralelismo é aquele onde se tem um elemento responsável por coordenar as atividades, e diversos trabalhadores que realizarão a tarefa propriamente dita, sendo coordenados. Neste caso, o coordenador inicializa o problema e cria uma série de trabalhadores idênticos, que trabalharão nos diversos passos da computação do problema. O mesmo problema poderá ser modelado com 1, 10, 100, etc trabalhadores. Um exemplo desta forma de paralelismo é quando se tem uma base de dados contendo registros de pessoas. Deseja-se pesquisar esta base e descobrir qual pessoa ganha mais relativamente a sua idade. Suponhamos que, dada uma pessoa P, a função *s(P)* retorna esta proporção, quantizada em termos de um número de ponto flutuante. O problema, então, é simples: "*Aplicar a função s a todos os registros da base de dados. Retornar o registro para o qual s é máximo*". Modelando este problema com paralelismo de agenda, criamos um coordenador que cria um conjunto contendo os registros das pessoas. Os trabalhadores, por sua vez, repetidamente retiram um registro de pessoa do conjunto, computam *s*, e encaminham o resultado ao coordenador, que se encarrega de manter o registro com mais alto *s*. Quando todos os trabalhadores terminam, o coordenador responde com o valor correto.

- Por fim, paralelismo de especialista envolve problemas que podem ser pensados em termos de uma rede lógica (um grafo). Aqui, o problema é modelado como uma rede onde cada nodo é representado por uma sequência de computação relativamente autônoma, e a comunicação entre eles segue caminhos previsíveis. Um simulador de circuitos, por exemplo, pode ser modelado como um programa paralelo onde cada elemento do circuito é um processo (cada qual sendo um nodo da rede lógica). Um outro exemplo, mais complexo, seria de uma empresa de transportes que quisesse efetuar uma série de estimativas de tempo de transportes entre duas localidades. Fatores como clima, tráfego, condições da estrada, etc influenciariam no tempo a ser estimado. Podemos modelar este problema com paralelismo de especialista da seguinte forma: Uma rede lógica, onde cada nodo representa um estado brasileiro. Cada estado é modelado por um processo autônomo independente, cada qual responsável por se manter atualizado sobre condições de estrada, etc características de sua área. Para estimar uma viagem entre Santa Catarina e Bahia, por exemplo, primeiramente planeja-se a rota e inclui-se a representação da mesma em um objeto que represente um caminhão [47]. Este caminhão é "entregue" a Santa Catarina, que calcula o tempo estimado da viagem através de si, e encaminha ao próximo estado da rota. Ao chegar ao último estado da rota, o tempo terá sido estimado. Note que vários caminhões do nosso modelo podem estar trafegando através desta rede de estados. Desta forma, as várias estimativas são computadas potencialmente em paralelo.

5.4.3 - Aplicando o Método mais Natural

Conforme o tipo de paralelismo que mais se encaixa com determinado problema, empregaremos o método correspondente. No caso de paralelismo de resultado, empregamos estruturas de dados vivas. Em Linda, isso significa usar extensivamente a primitiva *eval*. Consequentemente, teremos um paralelismo de alta granularidade, pois cada tupla viva representa uma atividade concorrente. Já em paralelismo de agenda, empregamos estruturas de dados distribuídas. Em Linda, isso significa explorar, de maneira equilibrada, as primitivas *eval* e *out* (com operações *in*/*rd* associadas, claro). Teremos, então, estruturas de dados representadadas sob a forma de tuplas, e distribuídas pelo espaço de tuplas. Um número menor de processos interage através de manipulações destas estruturas de dados, as quais representam tanto o cronograma de atividades bem como os resultados das mesmas (ou seja, uma agenda). Consequentemente, teremos um paralelismo de granularidade média. Finalmente, no paralelismo de especialista, a troca de mensagens é predominante. Aqui, diversos agentes especialistas interagem através de tais trocas de mensagens, as quais são modeladas, em Linda, através das mesmas primitivas *eval* e *out*. Teremos, então, paralelismo de baixa granularidade, pois o número de processos que executam atividades de forma paralela (os especialistas) será potencialmente menor que nos dois casos anteriores.

A aplicação de um método que inicialmente parece mais natural, pode levar a soluções ineficientes, ou até mesmo difíceis de adaptar, etc. Nestes casos, aconselha-se transformar a

---

[47] Note que em linguagens tradicionais tal caminhão seria representado de maneira pobre, dada a dificuldade de modelar abstrações. Em um dialeto Linda com suporte para objetos, certamente tal representação seria mais rica. Objetos, por encapsularem representação e comportamento em uma única entidade, permitiriam uma melhor abstração de tal caminhão. Mudanças futuras no sistema permitiriam incorporar subclasses da anterior (caminhão), refinando e adaptando tanto comportamento quanto representação de um caminhão, sem comprometer a modularidade.

representação do problema de um método para o outro. [Carriero89b] apresenta de forma detalhada os passos a seguir para cada uma das seis transformações possíveis.

5.4.4 - Modelando Estruturas de Dados no Espaço de Tuplas

Estruturas de dados modeladas sobre o Espaço de Tuplas podem ser classificadas, de forma geral, em:

1) Estruturas com elementos idênticos ou não-distinguíveis.

2) Estruturas cujos elementos são distinguíveis pelo nome

3) Estruturas cujos elementos são distinguíveis por posição. Esta pode ser dubdividida em:

a) Aquelas onde os elementos podem ser acessados aleatoriamente

b) Aquelas onde os elementos são acessados segundo alguma ordem

5.4.4.1 - Estruturas com Elementos Idênticos

Estruturas cujos elementos não são distinguíveis surgem, normalmente, na forma de *Bags*. Esta estrutura está associada a duas operações principais: adicionar e retirar um elemento. Estas estruturas surgem principalmente no uso do paralelismo de agenda. Neste caso, o coordenador das atividades adicionará ao *Bag* um descritor de tarefa da seguinte forma:

```
out ("tarefa" , DescritorDeTarefa )
```

Listagem 5.9 - Adicionando Tarefas a um Bag em Linda

Os trabalhadores, por outro lado, retirarão do *Bag* descritores de tarefa a serem realizadas da seguinte forma:

```
in ("tarefa" , ? NovaTarefa )
```

Listagem 5.10 - Retirando Tarefas de um Bag em Linda

5.4.4.2 - Estruturas com Elementos Distintos por Nome

Estruturas deste tipo se assemelham a registros de linguagens como Pascal e C. São utilizadas nas diversas formas de paralelismo, sendo modeladas geralmente em tuplas na forma *(nome , valor)*. Para ler um determinado campo da estrutura, executa-se:

```
rd ( nome, ? valor )
```

Listagem 5.11 - Acessando Estruturas Identificadas por Nome em Linda

E para alterá-lo, executa-se a sequência:

```
in( nome, ? valor )
out ( nome,  valor )
```

Listagem 5.12 - Alterando Estruturas Identificadas por Nome em Linda

5.4.4.3 - Estruturas com Elementos Distintos por Posição

Usadas principalmente (mas não exclusivamente) em paralelismo de resultado e agenda, estruturas deste tipo, implementadas no espaço de tuplas, resultam em estruturas de dados distribuídas. O exemplo mais típico seria o de vetores (e matrizes). No espaço de tuplas, vetores podem ser representados da seguinte forma: *(nome-do-vetor,posiçao-do-elemento,elemento)*. Para ler um determinado elemento da estrutura, executa-se:

```
rd ( nome, indice, ? valor )
```

Listagem 5.13 - Acessando Estruturas Identificadas por Posição em Linda

E para alterá-lo, executa-se a sequência:

```
in( nome, indice, ? valor )
out ( nome,  indice, novoValor )
```

Listagem 5.14 - Alterando Estruturas Identificadas por Posição em Linda

Usando este tipo de representação, paralelismo de resultado para exemplos como soma de matrizes fica bastante simplificado (maiores detalhes no exemplo a ser apresentado mais adiante) .

Por outro lado, em algumas estruturas os elementos são acessados por posição, mas não aleatoriamente como no caso de vetores e matrizes. São exemplos típicos listas, grafos, árvores, *streams*, etc. Dessas, *streams* são as estruturas mais comuns de surgirem na modelagem de problemas paralelos.

Streams são, basicamente, uma coleção ordenada de elementos. Processos podem ler/consumir o próximo elemento do *stream* ou adicionar mais um elemento ao fim do *stream*. Associados ao *stream* de elementos temos, então, a indicação do próximo elemento a retirar/ler e também onde adicionar o próximo elemento. Em Linda, *streams* são representados por uma série numerada de tuplas, como:

```
("stream" , 1 , valor1)
( "stream" , 2 , valor2 )
...
```

Listagem 5.15 - Implementando *Streams* em Linda

A indicação do próximo elemento a ler/retirar pode-se dar com uma tupla na forma:

```
( "stream" , "inicio" , 1 )
```

De forma análogo, a indicação de onde inserir o próximo elemento se dá na forma:

```
( "stream" , "fim" , 3 )
```

Para adicionar um novo elemento ao *stream*, um processo executa:

```
in ( "stream", "fim" , ? indice )
out ("stream" , "fim" , indice + 1)      ; atualização da posição
out ("stream" , indice + 1, NovoElemento ) ; inserção do elemento
```

Listagem 5.16 - Adicionando Elementos a um *Stream* em Linda

Para retirar o próximo elemento do *stream*, um processo executa:

```
in ( "stream", "inicio" , ? indice )
ou ("stream" , "inicio" , indice + 1)   ; atualização da posição
in ("stream" , indice , ? ProximoElemento ) ; obtenção do elemento
```

Listagem 5.17 - Retirando Elementos de um *Stream* em Linda

Um caso especial de *stream* é aquele onde os processos lêem o próximo elemento apenas, sem retirá-lo. Nestes casos, não há necessidade de se manter um marcador de início de tupla. Cada processo pode armazenar localmente (sem usar o espaço de tuplas) seu próprio índice relativo ao próximo elemento a ler do *stream*. Tais processos podem ser representados por:

```
indice := 1
loop
    rd ("stream" , indice , ? ProximoElemento )
    indice := indice + 1
    ...
```

Listagem 5.18 - *Streams* de Leitura em Linda

Esta mesma técnica pode ser aplicada em casos onde apenas um processo adiciona novos elementos ao fim do *stream*.

Streams são usualmente utilizados quando se deseja troca de mensagens entre processos. Nestes casos, utiliza-se *eval* para criar os vários processos que comporão a rede lógica de especialistas. A comunicação entre os mesmos é feita através da leitura/escrita de mensagens em *streams* de mensagens.

5.4.5 - Exemplo

Ilustraremos aqui um problema simples: achar todos os números primos entre 1 e N. Retirado de [Carriero89b], este problema é ilustrativo por ser simples, porém modelável naturalmente nas várias formas de paralelismo sugeridas aqui.

5.4.5.1 - Paralelismo de Resultado

Neste tipo de abordagem, podemos pensar no resultado com sendo um vetor de N elementos booleanos. Caso o elemento *V[j]* seja VERDADEIRO, indica que *j* é primo. Caso seja FALSO, indica que *j* não é primo. Precisamos, então, definir na linguagem em que se inseriu o paradigma de Linda uma função que, dado um número qualquer *j*, retorna um valor booleano indicando se *j* é ou não primo. Chamaremos esta função de *éPrimo*. Nao detalharemos aqui sua implementação mas, de forma geral, um número *j* é primo se e somente se não há nenhum primo que o antecede, menor ou igual a raiz quadrada de *j*, que divide *j* sem deixar resto. Por definição, 1 é primo.

Assim o laço de repetição

```
FOR j := 2 to N DO
        eval ("primos" , j , éPrimo (j)  );
END;
```

Listagem 5.19 - Paralelismo de Resultado em Linda

resolve o problema. O acesso a qualquer elemento do vetor resultante pode, consequentemente, ser feito por:

```
rd ("primos" , j , ? ok )
```

Tal leitura pode ser usada tanto para imprimir o vetor resultante bem como pela própria função *éPrimo*, ao desejar saber, para um dado *j*, se os elementos que o antecedem são primos, de acordo com a idéia do algoritmo apresentada acima.

5.4.5.2 - Paralelismo de Agenda

A modelagem anterior é concisa, elegante e fácil de desenvolver. A granularidade dos processos é pequena - um grande número de processos com uma pequena sequência de computação. Porém, esta abordagem pode-se mostrar ineficiente (em termos de tempo de execução) dependendo da plataforma Linda em que é implementada. Nestes casos, podemos transformar o modelo para paralelismo de agenda, onde teremos um menor número de atividades paralelas processando uma maior sequência de atividades.

Neste nosso exemplo, ao invés de construir um vetor vivo no espaço de tuplas, podemos usar um vetor passivo e criar processos "trabalhadores". A faixa 1-N de números a serem testados (se são primos ou não) pode ser dividida em sub-faixas de tamanho adequado para cada processo trabalhador. Assim, trabalhadores pegam, no espaço de tuplas, uma nova sub-faixa a testar e prosseguem. Os vários processos trabalhadores apresentam o mesmo comportamento, parando somente quando todas as sub-faixas tiverem sidos testadas.

Trabalhadores podem ser expressos da forma:

```
loop
    in ("proxima tarefa" , ? inicio )
    out ("proxima tarefa" , inicio + GRANULARIDADE)
    ...acha os primos na faixa inicio..inicio + GRANULARIDADE
```

Listagem 5.20 - Processos em Paralelismo de Agenda em Linda

Desta forma, os trabalhadores pegam sub-faixas em sequência, onde cada trabalhador ocioso obtém a próxima sub-faixa que ele deve computar, e atualiza para o próximo trabalhador que se tornar ocioso. A condição de parada pode ser definida, por exemplo, por um valor especial tal como *("proxima tarefa",-1)*. O coordenador simplesmente cria os trabalhadores, da seguinte forma:

```
tr := numeroDeTrabalhadores (N, GRANULARIDADE)
FOR i := 1 TO tr DO
          eval ("trabalhador" , i )
```

Listagem 5.21 - Paralelismo de Agenda em Linda

Uma modificação possível no algoritmo é, ao invés de produzir um vetor booleano de tamanho N, produzir uma coleção apenas com os números primos encontrados. Este refinamento, bem como detalhes de performance em diferentes plataformas Linda podem ser encontrados em [Carriero89b].

5.4.5.3 - Paralelismo de Especialista

Um algoritmo para determinação de números primos que se encaixa com o paralelismo de especialista é o crivo de Eratosthenes [Carriero89b]. Conceitualmente, o algoritmo pode ser pensado com uma coleção (*stream*) de números inteiros, os quais passam sob uma série de crivos. O crivo "2" remove desta coleção os múltiplos de "2". O crivo "3" remove desta coleção os múltiplos de "3", e assim por diante. Quando o último crivo acha um número primo, ele adiciona este novo crivo à sequência de crivos já existentes. Assim, crivos funcionam como filtros, onde para cada novo primo encontrado, um novo crivo (filtro) é adicionado (acrescentando, assim, mais um elemento ao *pipeline* de processos-crivo).

Uma maneira de iniciar este problema é com dois processos. Um produz o *stream* de números, enquanto o outro representa o último crivo. Cada vez que este acha um novo primo, ele transforma-se no crivo deste novo número, e insere um novo processo-crivo em seu lugar.

Podemos usar um *stream* de um único produtor e um único consumidor. Elementos deste *stream* aparecerão no espaço de tuplas na seguinte forma:

```
("seg", <destino> , <indice-no-stream> , <integer> )
```

Onde <destino> identifica o próximo segmento do "pipe". Ou seja, um segmento de "pipe" que remova múltiplos de "3" aparece na forma:

```
("seg", 3 , <indice-no-stream> , <integer> )
```

Ao ser encontrado um novo primo, o crivo que o encontrou cria um novo crivo em seu lugar , através de *eval*. Para isso, o crivo que descobriu o novo primo desacopla-se do *stream* de números a acopla-o ao novo crivo criado. A saída deste novo crivo criado torna-se, então, o novo *stream* de entrada do processo que evocou *eval*. Ao fim, teremos no espaço de tuplas algo como:

```
("produtor" , 1 , 2 )          ; produzida pelo gerador do
                                 stream de entrada
("segmento do pipe" , 2 , 3 )      ; produzido pelo crivo "2"
("segmento do pipe" , 3 , 5 )      ; ...
("segmento do pipe" , 4 , 7 )
...
("ultimo" , MaiorIndice , MaiorPrimo )
; produzido pelo crivo que dispara os evals
```

Listagem 5.22 - Paralelismo de Especialista em Linda

Esta estrutura pode, então, ser percorrida e o resultado ser apresentado. A listagem completa da solução é descrita em [Carriero89b].

Esta abordagem oferece menos paralelismo que a anterior. No paralelismo de agenda, era possível testar, simultaneamente, a existência de primos entre k e k^2 para cada novo primo k. Já no paralelismo de especialista apresentado aqui, esta busca não pode se dar de forma paralela, mas sim um por vez. O *pipelining* permite uma rápida sequência de descobertas, mas estas são feitas sequencialmente no tempo. Detalhes sobre a ineficiência desta abordagem em implementações Linda podem ser encontrados em [Carriero89b].

5.5 - Considerações

Linda apresentou algumas alterações desde suas propostas iniciais [Gelernter85]. Destacamos o surgimento das primitivas *eval*, *inp* e *rdp*.Outra expansão foi o surgimento de múltiplos espaços de tuplas, e não um só.

O mecanismo simples de espaço de tuplas permite que sua implementação seja distribuída [Ahuja86], propiciando o desenvolvimento deste tipo de aplicações.

A possibilidade de representar abstrações, por outro lado, não é responsabilidade de Linda. Os tipos que possam surgir nos campos das tuplas são limitados exclusivamente pela linguagem em que se insere o modelo. Isso deixa em aberto a potencialidade de expressar e extender os tipos de objetos presentes nas tuplas. Com isso em mente é que procuramos ver os benefícios decorrentes da combinação de linguagens orientadas a objetos clássicas com Linda, conforme segue.

6 - Linda e Objetos

A filosofia Linda tem sido incorporada em diferentes ambientes, dando origem a dialetos dos mesmos, conforme descrito anteriormente. Algumas destas experiências foram feitas com linguagens sem o suporte para a orientação a objetos, como C e Modula-2. Outras, contudo, foram incorporadas em linguagens com tal suporte, tais como C++, Eiffel e Smalltalk [48].

6.1 - Propostas Existentes

A existência de parâmetros formais nas tuplas do modelo Linda requer que informação sobre tipagem possa ser definida na linguagem em que o modelo está sendo inserido. A inexistência de tal facilidade em linguagens como Modula-2, C, etc faz com que haja necessidade de se implementar um pré-compilador para que as operações Linda possam ser introduzidas em determinada linguagem. Este é o caso, por exemplo, com C-Linda.

Vamos nos ater aqui a implementações onde isto não se fez necessário. As que destacamos são implementações em Eiffel [Jellinghaus90] e Smalltalk [Matsuoka88], justamente duas linguagens orientadas a objetos. O contraste das duas vale a pena devido ao fato de uma ser fortemente tipada, enquanto a outra não.

6.1.1 - Eiffel Linda

[Jellinghaus90] apresenta uma experiência onde se incorporam as primitivas Linda em Eiffel. O modelo implementa a classe TUPLE, onde são definidos os equivalentes das primitivas Linda *in*, *rd* e *eval*. Assim, tuplas são objetos da linguagem, e as primitivas Linda são métodos de instância em Eiffel.

A noção de classe, em Eiffel, deixa de existir em tempo de execução. Segundo [Meyer88], classes têm existência em tempo de compilação, e instâncias em tempo de execução. Justamente a impossibilidade de expressar informações sobre classes dificulta a definição de formais em Linda. Aqui, o autor optou por identificar os formais através de *void* [49]. Sendo um formal um identificador de algo a ser aceito (uma instância de determinada classe) no *casamento*, uma referência *void* poderia retornar justamente a instância a ser aceita após o *casamento* das tuplas. Como os tipos INTEGER, etc[50] não são referências em Eiffel, eles nunca provocarão *casamento* com o formal *void* (que é uma referência vazia). Isso faz com que tais objetos de "tipos primitivos" somente *casem* com outro exatamente igual (casamento de parâmetro real com parâmetro real). Para poder possibilitar formal para os chamados tipos primitivos, é necessário definir, em Eiffel, uma classe para cada um dos tipos primitivos. No caso de INTEGER, por exemplo, seria necessário uma classe CLASSE_INTEGER. Referências *void* desta classe seriam consideradas formais que casariam com instâncias desta classe. Assim, cada INTEGER deveria ser transformado em instância de CLASSE_INTEGER para depois ser inserido em tuplas para termos a potencialidade de expressar formais para tais tipos.

---

[48] Um levantamento aprofundado de plataformas que suportam Linda pode ser encontrado em [Carriero89a].

[49] *Void* é equivalente a um pointeiro nulo - *NIL* em Modula-2.

[50] Tipos "primitivos"em outras linguagens.

A otimização das tuplas para posterior *casamento*, como é usualmente feito pelo pré-compilador C-Linda, não se faz necessário em Eiffel Linda. Aqui, através de asserções [Meyer88] e o mecanismo de classificação, a condição de casamento pode ser garantida. A forte tipagem de Eiffel continua sendo válida para as novas classes que formam o modelo Linda.

A possibilidade de definir tuplas, cujos campos podem conter virtualmente qualquer objeto da linguagem, traz problemas conhecidos no modelo Linda. Um campo de uma tupla poderia conter, por exemplo, uma estrutura de dados (árvore, por exemplo).A estruturá poderá conter objetos compartilhados por outras estruturas, conforme a figura que segue:

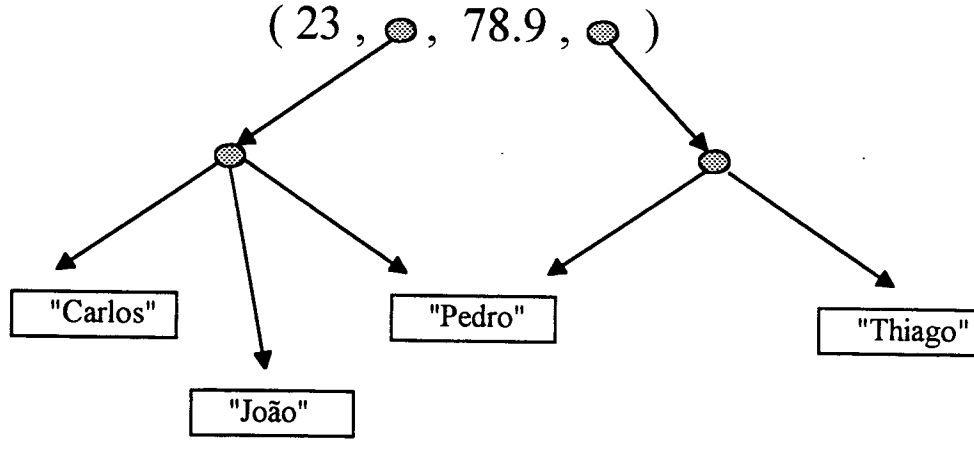

Figura 6.1 - Objetos Compartilhados em Tuplas

Ao ser colocada no espaço de tuplas, esta tupla deve carregar os objetos que contém, propriamente falando, e não apenas as referências. Isso se deve ao fato de que referências são usualmente ponteiros (*pointers*), que têm sentido apenas em um único espaço de endereçamento. No caso de Linda, vários processos podem estar cooperando, cada qual em um espaço de endereçamento distinto. Esta possibilidade de distribuição torna crítico o uso de ponteiros no modelo Linda.

A técnica adotada aqui é copiar toda a estrutura da referência, através da funcionalidade da classe STORABLE de Eiffel. Desta forma, todo o grafo de referências de cada um dos sub-objetos é copiado para o espaço de tuplas. Comunicação através do espaço de tuplas têm, então, a semântica de cópia. Um objeto de uma tupla produzida com *out*, por exemplo, será copiado para o contexto de outro processo que evoque *in* ou *rd*. Agora dois parâmetros reais apenas *casam* se as suas estruturas são completamente iguais, conforme armazenado através da funcionalidade de STORABLE. Ciclos no grafo de referência são detectados apropriadamente pela implementação.

A sintaxe das operações fica um pouco diferente, por exemplo, de C-Linda. Seja *x* um objeto Eiffel que represente uma tupla, as "primitivas" Linda surgiriam da seguinte forma:

```
x.out
x.in
```

Listagem 6.1 - Sintaxe Eiffel Linda

Neste último caso, seguindo a filosofia Eiffel, a operação não apresenta efeitos colaterais. Desta forma, o programador precisa acessar explicitamente os campos com novos objetos resultantes do *casamento*, uma vez que não há alteração das variáveis locais que integram tuplas, conforme ocorre com C-Linda.

Finalmente, a operação *eval* é definida também na classe TUPLE. A princípio, esta operação não faz praticamente nada. O programador deve adicionar subclasses de TUPLE, redefinindo a operação *eval* para cada nova classe adicionada. Ao ser terminada a computação de *eval* definida na nova subclasse, a tupla é automaticamente [51] adicionada ao espaço de tuplas. É importante notar a grande perda de expressão em relação ao modelo Linda original. Por exemplo, em C-Linda é possível:

```
eval ( find(letfTree, 55) , find (rightTree, 99)
```

Aqui, serão executados em paralelo as duas operações de pesquisa, cada qual em uma sub-árvore. Haverá, ainda, uma sincronização das duas atividades, uma vez que somente após as duas operações terem sido terminadas é que a tupla viva transforma-se em tupla ordinária, sendo adicionada ao espaço de tuplas. Na implementação Eiffel Linda, a redefinição de *eval* em uma nova subclasse forçaria o programador a:

1) Definir uma subclasse de TUPLE somente para efetuar tal operação *eval*

2) Escolher uma dentre duas formas de avaliar as duas operações de pesquisa:

a) Efetuá-las uma de cada vez, perdendo a oportunidade de paralelizar tal busca através da potencialidade do modelo.

b) Através de um outro recurso de Eiffel, disparar dois sub-processos, um para cada pesquisa, e implementar alguma forma de sincronização entre os mesmos. Somente após terminadas as duas operações de busca o processo original (*eval* da nova classe definida) poderá continuar, para que a tupla seja inserida no espaço de tuplas. Esta abordagem contraria justamente a elegância que o modelo propõe, forçando o programador a ter que utilizar um segundo método ainda para disparar e sincronizar processos.

A utilização de objetos para implementar tuplas, por outro lado, permite que tuplas possam ser atribuídas a variáveis, passadas como parâmetro, etc, ou seja, tratadas de forma uniforme na linguagem. Mais além, sendo os parâmetros formais objetos da linguagem, eles podem também ser tratados da mesma forma, podendo também haver casamento entre formais.

Objetos distribuídos podem ser implementados através de novas classes em Eiffel. Por exemplo, ARRAY_DISTRIBUIDO , LISTA_DISTRIBUIDA, etc podem ser implementados sobre as tuplas Linda, cuja distribuição faz parte do modelo. Esta é uma potencialidade a ser

[51] Graças ao código *eval* da superclasse TUPLE .

explorada, que depende da implementação realmente distribuída do espaço de tuplas. O autor aponta alguns problemas desta abordagem em relação à forte tipagem de Eiffel, aliado ao fato de a igualdade escolhida para efeitos de casamento ser estrutural. O autor sugere o método *match* a ser definido para instâncias de tuplas, mas não optou por esta implementação devido à sua maior complexidade e perda de performance.

Eiffel Linda foi originalmente implementado sobre uma rede de estações de trabalho SUN, utilizando o kernel Linda TSnet [Arango89]. Sua intenção não era provar eficiência do modelo Linda, o que já é demonstrado por C-Linda, mas sim investigar as potencialidades/limitações da combinação do modelo de objetos de uma linguagem tipada como Eiffel e Linda. O autor ressalta ainda algumas limitações na distribuição de executáveis Eiffel distintos que precisem interagir através do espaço de tuplas. Isso ocorre devido ao identificador interno que Eiffel associa a cada objeto, que deve ser uniforme caso tais objetos sejam espalhados através de uma rede de estações de trabalho. A primitiva *eval*, por outro lado, ainda não havia sido implementada até a data da publicação do trabalho.

6.1.2 - Smalltalk Linda

[Matsuoka88] propõe a inserção do modelo Linda em Smalltalk [52]. Primeiramente o autor propõe adaptações ao modelo de espaço de tuplas conforme apresentado originalmente, de forma a se adequar na combinação com uma linguagem orientada a objetos como Smalltalk.

6.1.2.1 - Adaptações do Modelo

As principais adaptações do modelo Linda para combinação com Smalltalk são:

- Tuplas e Espaço de Tuplas como Objetos

Aqui, o autor propõe que tanto as tuplas quanto o espaço de tuplas de Linda sejam tratados como objetos em Smalltalk, podendo ser passados como parâmetros, atribuídos a variáveis, etc. Algumas das potencialidades decorrentes são:

i) Um objeto pode reusar a tupla em novas operações

ii) Tuplas são sujeitas à coleta de lixo do sistema

- Ortogonalidade entre Emissor e Receptor

No modelo Linda original, processos que adicionam tuplas ao espaço de tuplas descrevem a estrutura da tupla em si. Processos que irão ler/retirar tal tupla, especificam a estrutura da tupla. Segundo o autor, tornar tuplas objetos do sistema viola o princípio de ortogonalidade. Desta forma, tanto as tuplas que são inseridas no espaço quanto as especificações de tuplas para casamento são consideradas simplesmente tuplas. Desta forma, elas podem ser usadas e reusadas entre vários processos emissores/receptores igualitariamente. Mais além, o autor propõe uma mudança de interpretação das operações Linda. Tando as primitivas *out* quando *rd* e *in* adicionam uma tupla ao espaço de tuplas. Caso haja o casamento os formais da tupla associada ao rd/in serão instanciados com os parâmetros reais da tupla produzida pelo *out*.

---

[52] Na verdade, o artigo relata um protótipo implementado sobre ConcurrentSmalltalk.

- Múltiplos Espaços de Tuplas

O autor propõe a existência de diversos espaços de tuplas, qualquer um deles podendo ser usado na comunicação entre os processos. Tal característica visa eliminar deficiências do modelo inicialmente proposto por Linda, com um só espaço de tuplas, tais como:

i) Ineficiência. Um único espaço de tuplas, onde todas as operações acontecem, tende a ser um gargalo do sistema

ii) Falta de Escopo. Quando grupos disjuntos de processos se comunicam através de um único espaço de tuplas faz-se necessária a utilização de convenções para identificar que tuplas são associadas a quais grupos de processos. Este tipo de informação propaga-se pelo sistema todo, violando a modularidade através da falta de escopo.

iii) Falta de Segurança. Da deficiência acima decorre que qualquer descuido cometido pelo programador pode acarretar em tuplas de grupos distintos serem "confundidas", originando erros difíceis de se detectar.

Além de superar as deficiências acima, múltiplos espaços de tuplas possibilitam:

i) Criação dinâmica de novos espaços de tuplas. Espaços não utilizados podem ser coletados como lixo de forma automática pelo sistema.

ii) Espaços de Tuplas, por serem objetos do sistema, podem ser passados como parâmetro, atribuídos a variáveis e até mesmo integrarem um campo de uma tupla usada para comunicação em outro espaço de tuplas.

6.1.2.2 - Classes Adicionadas

Dada estas adaptações ao modelo original, a proposta adiciona um conjunto de classes e métodos em Smalltalk para implementação do modelo Linda. Dentre eles, destacam-se:

- Tuple

A classe Tupla (Tuple) define os objetos que são usados na comunicação entre processos. São similares a coleções ordenadas, no sentido de que podem ter tamanho variável e têm seus elementos acessados de forma indexada. Seus elementos são separados por espaços, todos delimitados por parênteses, tal como: *(#foo 123 bar)*.

- Formal

A classe Formal, por outro lado, serve para definir um parâmetro formal de uma tupla. Basicamente formais armazenam a informação de qual classe se deseja que o formal represente. Por exemplo, *Formal ofClass: Point*.

- TS

A classe TS (Tuple Space) define os espaços de tuplas que serão criados no modelo, implementando a funcionalidade Linda propriamente dita.

6.1.2.3 - O *Casamento*

Quatro combinações são possíveis no casamento de parâmetros reais e formais de tuplas. São elas:

i) real x formal : Ocorre casamento caso o objeto que forma o parâmetro real seja instância da classe que forma o parâmetro formal ou qualquer uma de suas subclasses.

ii) real x real : Aqui, os objetos devem ter métodos que permitam dizer se eles são equivalentes. Mais ainda, um dos objetos pode-se considerar equivalente ao outro, enquanto a recíproca não é verdadeira. De forma objetiva, é necessário que esta relação seja simétrica. Outro ponto a ressaltar é que a relação de equivalência pode existir sem que os dois objetos pertençam necessariamente a uma mesma classe.

iii) formal x formal : Não há casamento

iv) formal x real : Ocorre casamento caso o objeto que forma o parâmetro real seja instância da classe que forma o parâmetro formal ou qualquer uma de suas subclasses.

6.1.2.4 - Métodos que Definem a Funcionalidade do Modelo

A classe TS implementa efetivamente as operações do modelo Linda. Os principais métodos de instância para tal são:

- <Espaço> put: <Tupla>

Operação idêntica a *out* do modelo Linda. A tupla é adicionada ao espaço de tuplas.

- <Espaço> get: <Tupla>

Operação idêntica a *in* do modelo Linda.A tupla instanciada pelo casamento é retirada e retornada.

- <Espaço> read: <Tupla>

Operação idêntica a *rd* do modelo Linda. Uma cópia da tupla instanciada pelo casamento é retornada.

- <Espaço> set: <Tupla>

Esta operação é novidade para o modelo Linda original. Similar a *put:* (*out*) , *set:* se diferencia por bloquear o processo que evoca este método até que haja casamento desta tupla.

6.1.2.5 - Extensões ao Modelo

Algumas primitivas são sugeridas para manipulação de diversas tuplas de uma só vez ou manipulação de diversos espaços de tuplas simultaneamente.

Múltiplas Tuplas

- <Espaço> put: <Tupla> and: <Tupla> ... and: <Tupla>

Estas operações, análogas ao *out* de Linda, permitem que várias tuplas sejam adicionadas ao espaço de tuplas simultaneamente.

- <Espaço> set: <Tupla> and: <Tupla> ... and: <Tupla>

Estas operações permitem que várias tuplas sejam adicionadas ao espaço de tuplas simultaneamente. O processo fica bloqueado até que todas tuplas inseridas provoquem um casamento .

- <Espaço> set: <Tupla> or: <Tupla> ... or: <Tupla>

Estas operações permitem que várias tuplas sejam adicionadas ao espaço de tuplas simultaneamente. O processo fica bloqueado até que uma das tuplas inseridas provoque um casamento .

- <Espaço> get: <Tupla> and: <Tupla> ... and: <Tupla>

Estas operações, análogas ao *in* de Linda, permitem que várias tuplas sejam instanciadas a partir do espaço de tuplas. O processo é bloqueado até ocorrer casamento em todas as tuplas. O resultado é uma coleção ordenada (*OrderedCollection*) de tuplas instanciadas.

- <Espaço> get: <Tupla> or: <Tupla> ... or: <Tupla>

Estas operações, análogas ao *in* de Linda, permitem tentar instanciar uma dentre várias tuplas a partir do espaço de tuplas. O processo é bloqueado até ocorrer casamento de uma delas. O resultado é justamente a tupla resultado da instanciação.

- <Espaço> read: <Tupla> and: <Tupla> ... and: <Tupla>

Estas operações, análogas ao *rd* de Linda, permitem que várias tuplas sejam lidas a partir do espaço de tuplas. O processo é bloqueado até ocorrer casamento em todas as tuplas. O resultado é uma coleção ordenada (*OrderedCollection*) de tuplas instanciadas.

- <Espaço> read: <Tupla> or: <Tupla> ... or: <Tupla>

Estas operações, análogas ao *rd* de Linda, permitem tentar ler uma dentre várias tuplas a partir do espaço de tuplas. O processo é bloqueado até ocorrer casamento de uma delas. O resultado é justamente a tupla resultado da instanciação.

Múltiplos Espaços de Tuplas

- <Espaço> set: <Tupla> andInTS: <Espaço> set: <Tupla> ... andInTS: <Espaço> set: <Tupla>

Estas operações permitem que várias tuplas sejam adicionadas aos vários espaços de tuplas simultaneamente. O processo fica bloqueado até que todas tuplas inseridas provoquem um casamento .

- <Espaço> get: <Tupla> andInTS: <Espaço> get: <Tupla> ... andInTS: <Espaço> get: <Tupla>

Estas operações, análogas ao *in* de Linda, permitem que várias tuplas sejam instanciadas a partir de diferentes espaços de tuplas. O processo é bloqueado até ocorrer casamento em todas as tuplas. O resultado é uma coleção ordenada (*OrderedCollection*) de tuplas instanciadas.

- <Espaço> get: <Tupla> orInTS: <Espaço> get: <Tupla> ... orInTS: <Espaço> get: <Tupla>

Estas operações, análogas ao *in* de Linda, permitem tentar instanciar uma dentre várias tuplas a partir de diferentes espaços de tuplas. O processo é bloqueado até ocorrer casamento de uma delas. O resultado é justamente a tupla resultado da instanciação.

- <Espaço> read: <Tupla> andInTS: <Espaço> read: <Tupla> ... andInTS: <Espaço> read: <Tupla>

Estas operações, análogas ao *rd* de Linda, permitem que várias tuplas sejam lidas a partir de diferentes espaços de tuplas. O processo é bloqueado até ocorrer casamento em todas as tuplas. O resultado é uma coleção ordenada (*OrderedCollection*) de tuplas instanciadas.

- <Espaço> read: <Tupla> orInTS: <Espaço> read: <Tupla> ... orInTS: <Espaço> read: <Tupla>

Estas operações, análogas ao *rd* de Linda, permitem tentar ler uma dentre várias tuplas a partir de diferentes espaços de tuplas. O processo é bloqueado até ocorrer casamento de uma delas. O resultado é justamente a tupla resultado da instanciação.

6.1.2.6 - *Eval*

A implementação sugerida pelo autor não menciona a primitiva *eval*. Tal fato se deve, provavelmente, ao fato de que o modelo Linda ter sofrido modificações desde sua introdução, sendo uma delas o surgimento de *eval*. Sem tal primitiva, contudo, fica difícil imaginar como os processos são criados nesta implementação. Os espaços de tuplas servem para comunicação e sincronização, mas a criação de processos não fica definida. Provavelmente o autor espera que tais processos decorram do fato da utilização de objetos paralelos de ConcurrentSmalltalk, onde o modelo foi implementado. Para versões ordinárias de Smalltalk, provavelmente o programador se veria forçado a usar o método de instância *fork* para blocos (Context).

6.2 - Considerações

A combinação de Linda com linguagens orientadas a objetos clássicas permitem também adicionar o poder de expressar atividades paralelas na orientação a objetos. O objeto que provê tal unificação, entretanto, é a tupla (mais especificamente tuplas ordinárias e tuplas vivas), caracterizando uma forma alternativa de programação concorrente orientada a objetos.

A possibilidade de incorporar o modelo a implementações comerciais de linguagens orientadas a objetos faz desta abordagem uma opção prática, que propicia a reusabilidade de velhos programas nas plataformas em que se incorpora o Linda. A possibilidade de expressar novos objetos adicionados ao sistema (classes) faz com que o modelo seja extensível e promova a abstração de dados, reusabilidade e modularidade [Meyer88].

A possibilidade de distribuição do espaço de tuplas através de espaços de endereçamento disjuntos faz desta forma de combinação um promissor modelo na programação distribuída e orientada a objetos. Aqui, objetos podem existir tendo seus componentes (variáveis de instância) particionados em espaços de endereçamento distintos. Isso pode ser obtido, por exemplo, mapeando variáveis de instâncias de objetos ordinários a tuplas distintas no espaço de tuplas. Estas teriam, então, existência independente nas dimensões espaço e tempo, o que é enfatizado por Linda.

A possibilidade de acessar meta-informação sobre a própria implementação [LaLonde90] faz de Smalltalk um forte candidato a Linda. Pode-se pensar inclusive em alterar a forma de o sistema armazenar os dicionários de métodos, provendo a Smalltalk uma forma distribuída e compartilhada de assimilar mudanças incrementais (como adição de classes ou adição de métodos a uma classe). Buscando esta potencialidade, buscamos esta combinação, a ser vista a seguir.

7 - LindaTalk

Na solução de problemas, um dos atributos mais atrativos da orientação a objetos, conforme vimos no início deste trabalho, é permitir uma modelagem natural de entidades no domínio do problema. Desta forma, entidades tais como carros, pessoas ou até mesmo conceitos abstratos podem ser modelados.

Na busca por modelos que suportem a cooperação de forma paralela entre entidades, surgiram as linguagens paralelas, como Occam, Conic, Ada [Mundie86], etc. Muitas destas, entretanto, negligenciaram a modelagem natural das entidades do domínio do problema (justamente o ponto forte da orientação a objetos) conforme vimos na segunda parte deste trabalho.

Por outro lado, a programação paralela orientada a objetos, conforme vimos na terceira parte deste trabalho, surge no sentido de tentar a unificação de objetos no sentido clássico de orientação a objetos com a noção de processos paralelos. Tanto nesta abordagem quanto na programação orientada a objetos clássica, contudo, a metáfora de interação entre objetos é a mesma: troca de mensagens. Uma característica central desta forma de interação é a sincronização das mensagens. O envio de mensagens pode ser:

- Síncrono
- Assíncrono

A forma de comunicação, por outro lado, abrange a relação entre os emissários e destinatários das mensagens. Dentre elas destacamos [Andrews83] [Stemple86] [Matsuoka88]:

- Simetria de Comunicação - Simétrica ou Assimétrica (mestre-escravo)
- Correspondência de Comunicação - 1 para 1, 1 para N, N para 1 ou N para N
- Escopo de Comunicação - Para quem uma mensagem pode ser enviada

Uma questão que afeta enormemente a correspondência e o escopo da comunicação é o esquema de referenciamento (identificação) de processos. Dentre estes esquemas, destacamos:

- Endereçamento Implícito - Permite que um processo se comunique com um único processo pai
- Endereçamento Explícito - Requer que o processo referencie seu destinatário explicitamente. Requer o conhecimento global de uma fonte que contém os identificadores de todos os processos no sistema.
- Endereçamento Global - Associa nomes globais a caixas postais locais.
- Endereçamento por Serviço - Estabelece conexões baseadas na necessidade de servir ou requisitar um serviço. A rota de um serviço tal como uma porta identifica o processo no outro extremo.

Caracterizando o envio de mensagens conforme presente em sistemas orientados a objetos são, podemos dizer que:

- Podem ser síncronos e/ou assíncronos
- São simétricos, em geral
- Têm, na grande maioria, correspondência 1 para 1
- Apresentam escopo global
- Adotam o endereçamento explícito

A referência explícita de identificadores de processos/objetos paralelos é restritiva, conforme enfatizam [Andrews83] [Stemple86]. Na interação de entidades paralelas surgem formas variadas de comunicação, que não a troca de mensagens, e que são mais naturais conforme o problema. Mensagens de difusão, por exemplo, são uma das formas características onde o envio de mensagens com endereçamento explícito 1 para 1 se mostra inadequado. Outro ponto fraco é quando um processo precisa interagir com outros N. Conhecer o valor exato de N invariavelmente causará a modelagem do primeiro processo altamente suscetível a mudanças. Envio de mensagens, na grande maioria dos casos, não consegue manter esta informação transparente para os processos que interagem.

Acreditamos que muitos dos problemas encontrados na utilização efetiva, mais precisamente distribuída, de sistemas orientados a objetos reside na forma restritiva de interação de seus objetos. Neste sentido, tentamos apresentar um modelo onde a forma de interação entre as entidades paralelas seja distinta do paradigma usual de troca de mensagens. Esta é a intenção de LindaTalk.

Nossa proposta, similar a [Matsuoka88], é incorporar o modelo Linda a Smalltalk. O modelo de espaço de tuplas de Linda, acreditamos, fornece uma forma alternativa de comunicação e sincronização das atividades paralelas, funcionando como uma linguagem de coordenação. Linda possibilita comunicações, por exemplo, 1 para N, desacoplando os processos que interagem até mesmo no tempo, dada a característica assíncrona permitida em Linda.

Ao contrário de [Matsuoka88], contudo, buscamos implementar [53] todas as primitivas do modelo citadas na bibliografia, incluindo *inp*, *rdp* e *eval*, como veremos a seguir.

7.1 - Classes Adicionadas

Para implementar nosso modelo, duas classes principais foram criadas. Uma delas, a classe Tupla, torna as tuplas de Linda objetos do sistema. Ou seja, podem ser criadas, atribuídas a variáveis, passadas como parâmetros e até mesmo integrar campos de outras tuplas. A segunda classe adicionada foi Espaço (de tuplas), a qual permite, além da mesma potencialidade descrita acima (objetos do sistema), a instanciação de múltiplos espaços de tuplas, e não apenas um espaço global.

---

[53] Por força das circunstâncias, nossa primeira implementação foi feita na plataforma Smalltalk/V 286, da Digitalk [Digitalk-Win] [Digitalk-286]. Sua portabilidade para outras implementações de Smalltalk, tais como ObjectWorks\Smalltalk, parece-nos trivial.

7.1.1 - Tupla

Tuplas são identificadas através de uma lista de objetos que as compõem, separadas por "||". Assim, uma possível tupla no nosso modelo seria:

```
( 'maria' || 45.7 || $r )
```

Listagem 7.1 - Tuplas em LindaTalk

Elementos que sejam resultados de expressões devem ser incluídos entre parênteses, como segue:

```
( (5 @ 6) || ( 4 >= 8 ) || ( 6 fatorial ) )
```

As expressões contidas são avalidas imediatamente, e a tupla é criada com os objetos resultantes. A tupla acima seria, no fim, a tupla que segue:

```
( (5 @ 6) || false || 720 )
```

Tais tuplas podem ser, por exemplo, atribuídas a variáveis ou compor outras tuplas, tal como segue:

```
| tuplaA tuplaB |

tuplaA := ( 'maria' || 45.7 || $r ) .
tuplaB := ( 76 || tuplaA || 'pedro' ).
```

Listagem 7.2 - Objetos que Integram Tuplas em LindaTalk

Optamos por "||"ao invés de "," , como é usual em dialetos como C-Linda devido ao fato de "," já ter o significado de concatenação de coleções em Smalltalk. Logo, a expressão abaixo

```
( 'maria' , 'pedro' )
```

não denotaria uma tupla válida, mas sim a concatenação de dois objetos *string*. Teríamos que ou mudar a implementação de "," para coleções (o que violaria toda a biblioteca de classes do sistema, que assume que tal mensagem tem o comportamento de concatenação) ou não permitir que qualquer coleção (*Strings*, *Arrays*, etc) fossem o primeiro elemento de uma tupla. Como este último caso é bastante frequente nos exemplos Linda, optamos por usar "||".

7.1.1.1 - Manipulando Elementos

A classe Tupla é subclasse de *OrderedCollection*, Como tal, provê métodos que permitem alterar/obter os elementos de uma tupla, bem como adicionar/remover objetos. Por exemplo, podemos querer saber qual o terceiro elemento de uma tupla, ou substituir o segundo elemento de uma tupla por um outro objeto. Os exemplos a seguir demonstram como

```
| tupla ponto char |

tupla := ( 'maria' || (3@5) || $t ).
ponto := tupla at: 2.

"Primeira forma"
char := tupla @ 3.

"Segunda forma"
tupla at: 3 put: 'novoElemento'.
```

Listagem 7.3 - Manipulando Tuplas em LindaTalk

Uma tupla pode ter elementos acrescentados ou removidos. O exemplo abaixo mostra a remoção do segundo elemento de uma tupla, e posterior inserção de um elemento como sendo o primeiro.

```
| tupla tupla2 |

tupla := ( 'maria' || (3@5) || $t ).
tupla removeElement: 2. " Primeira forma -  ( 'maria' || $t )  "
tupla2 := tupla - 1. " Segunda forma - ( $t )  "
tupla addFirst: 78.   " (78 || 'maria' || $t )  "
```

Listagem 7.4 - Adicionando e Removendo Elementos de Tuplas em LindaTalk

Além das operações de manipulação da estrutura de dados propriamente dita, a classe implementa as operações do modelo Linda, conforme veremos mais adiante.

7.1.1.2 - Expressando Formais

Seguinto o exemplo de C-Linda, onde os parâmetros potenciais representam os tipos da linguagem (no caso, tipos primitivos de C) optamos por usar classes de Smalltalk para denotar formais. Uma possível tupla a ser usada para casamento seria:

```
( Integer || Point ) in.
```

Listagem 7.5 - Formais em LindaTalk

Esta tupla causaria match, por exemplo, com a tupla:

```
( -45 || (5@6) ) out.
```

Devido ao fato de *-45* pertencer à classe *Integer*, há casamento , assim como o objeto *5@6* , por pertencer à classe *Point*. Uma classe presente em um campo de tupla usando em primitivas como *in, inp, rd, rdp* é interpretada como parâmetro formal, e causará casamento caso o objeto correspondente da tupla produzida com *out, outi ou eval* satisfaça a relação *isKindOf:* de Smalltalk.

7.1.1.3 - *Casamento*

Para haver casamento de duas tuplas, é necessário que elas tenham o mesmo número de elementos. Além disso, deverá haver casamento para cada um dos objetos que ocupam as mesmas posições nas tuplas. Acima vimos como um parâmetro formal causa casamento em relação a um parâmetro real em nossa implementação. Falta-nos, ainda, abordar as outras possíveis combinações.

Uma delas é justamente caso sejam dois parâmetros formais. Devido ao fato de classes em Smalltalk serem também objetos do sistema, optamos por permitir que haja casamento caso as classes sejam as mesmas. Assim, as tuplas:

```
( Point || 73      ) out.
( Point || Integer) in.
```

Listagem 7.6 - Casamento de Tuplas em LindaTalk

satisfazem a condição de casamento . O objeto *73* é um *Integer*, o que cai no exemplo citado anteriormente. *Point*, por outro lado, aparece nas duas tuplas. Podemos, então, interpretar de duas formas:

a) São dois formais idênticos. De forma análoga a reais idênticos, é natural que causem casamento .

b) São dois objetos idênticos. É um caso de casamento de dois reais, conforme veremos a seguir. Também para esta interpretação parece-nos natural que deve haver o casamento. .

Outra combinação possível é justamente dois objetos reais. Aqui o casamento é baseado na relação "=" de Smalltalk. Assim, as tuplas:

```
(34 || (3@2) || $t) out.
(34 || (3@2) || $t) in.
```

Listagem 7.7 - Casamento com Parâmetros Reais em LindaTalk

causam casamento pois os objetos satisfazem (retornam true para) a relação [54] "=".

A terceira combinação possível é aquela onde na tupla produzida por um com *out*, *outi* ou *eval* seja justamente um parâmetro potencial (uma classe) , e na tupla a ser usada com *in*, *inp*, *rd*, *rdp* apareça um parâmetro real. Um exemplo seria:

```
( Point || Integer ) out.
( (5@4) || 45 ) in.
```

Listagem 7.8 - Casamento com Parâmetros Formais em LindaTalk

Neste caso, haverá casamento, mas sem transferência de informação (parâmetros).

---

[54] Método, na verdade.

Obviamente aparecerão tuplas com diversos elementos, envolvendo possivelmente todas as possibilidades de casamento listadas aqui. Satisfeitas todas uma-a-uma, haverá casamento da tupla como um todo.

7.1.1.4 - Variações das Primitivas Linda

Para compatibilidade com programas em dialetos Linda com apenas um espaço de tuplas, foram implementadas duas variações para cada primitiva Linda. No caso da primitiva *out*, por exemplo, podemos ter:

- <Tupla> out. Primitiva *out* sobre o espaço "default"
- <Tupla> out: <Espaço>. Primitiva *out* sobre um determinado espaço

Desta forma, pode optar-se por trabalhar com um único espaço de tuplas, de acordo com as primeiras propostas do modelo, ou com diversas instâncias distintas de espaços de tuplas. Para todas as primitivas, a tupla simplesmente delega para o espaço correspondente o tratamento da operação. Desta forma, as primitivas propriamente ditas são implementadas na classe Espaço.

7.1.2 - Espaço

Embora na classe tupla haja a implementação das duas variações das primitivas citadas acima, elas evocam a mesma primitiva correspondente na classe Espaço. Por exemplo:

```
( 34 || 'verde') out.
```

transforma-se em:

```
Espaço default out: ( 34 || 'verde').
```

Alternativamente, no exemplo:

```
( 34 || 'verde') out: umEspaço.
```

transforma-se em:

```
umEspaço out: ( 34 || 'verde').
```

Desta forma, existe, na verdade, apenas uma implementação de cada primitiva na classe Espaço. A seguir, descreveremos as primitivas.

- out: <Tupla>

Este método na classe Espaço implementa a primitiva *out* de Linda, conforme apresentado anteriormente. Haverá ou não casamento baseado nas condições já apresentadas.

- outi: <Tupla>

Este método na classe Espaço implementa uma variação proposta por nós para a primitiva *out* de Linda. Embora as condições de casamento sejam as mesmas, existe uma peculiaridade quanto à persistência da tupla no espaço de tuplas. Enquanto *out* permanece no espaço até que seja consumida por um *in* ou *inp*, conforme veremos, *outi* simplesmente não tem efeito caso não haja, *a priori*, uma tentativa de leitura/consumo de tupla. Mesmo no caso onde haja uma tentativa *a priori* de leitura da tupla através de um *rd*, após tal leitura a tupla associada ao *outi* não permanecerá no espaço de tuplas.

- eval: <Tupla>

Este método na classe Espaço implementa a primitiva *eval* de Linda, conforme apresentado anteriormente. A tupla transforma-se em uma tupla viva, que iniciará uma sequência de computação. Quando terminada, tal tupla transforma-se em uma tupla ordinária, que é adicionada ao espaço de forma análoga a um *out*. Para tal, precisamos de uma forma de expressar uma sequência de computação. Aproveitamos os objetos *Context* de Smalltalk, denotados por *"[...]"*. Assim, um exemplo de tupla usada com *eval* seria:

```
( (5 @ 6) || ( 4 >= 8 ) || [ 6 fatorial ] ) eval.
```

Listagem 7.9 - Expressando Tuplas Vivas em LindaTalk

Note a diferença sutil para o exemplo usado anteriormente, dado por:

```
( (5 @ 6) || ( 4 >= 8 ) || [ 6 fatorial ] ) .
```

Enquanto neste último caso o fatorial é calculado *a priori*, à medida que se vai construindo a tupla, o exemplo com *eval* adia a computação real para mais tarde, a qual é executada em paralelo. Assim, logo após o *eval*, o processo que o evocou continua em paralelo com o cálculo do fatorial. Ao terminar a computação, a tupla acima transfroma-se, dentro do espaço de tuplas, na tupla ordinária:

```
( (5 @ 6) || false || 720 )
```

e poderá causar casamento a partir deste instante.

Eval permite que computações paralelas sejam sincronizadas entre si[55], pois a tupla somente será tranformada em tupla ordinária quando todos seus campos que contenham computações terminarem. Assim, no exemplo:

```
( [obj1 updateDatabase] || [obj2 printReport] )
```

um processo que aguarde a leitura (por exemplo) desta tupla terá a garantia que ambas as computações descritas foram feitas. Note que ambas são, potencialmente, feitas em paralelo, e sincronizadas graças ao comportamento de *eval*.

---

[55] Uma espécie de "join".

- in: <Tupla>

Este método na classe Espaço implementa a primitiva *in* de Linda, conforme apresentado anteriormente. Haverá ou não casamento baseado nas condições já apresentadas. O processo ficará bloqueado até que haja uma tupla para casamento , a qual será retirada do espaço de tuplas. Retorna-se a tupla resultante do casamento .

- inp: <Tupla>

Este método na classe Espaço implementa a primitiva *inp* de Linda, conforme apresentado anteriormente. Haverá ou não casamento baseado nas condições já apresentadas. Distinto de *in*, no caso de *inp* o processo não ficará bloqueado caso não haja uma tupla para casamento . Neste caso, retorna-se *nil*, e não a tupla resultante do casamento .

- rd: <Tupla>

Este método na classe Espaço implementa a primitiva *rd* de Linda, conforme apresentado anteriormente. Haverá ou não casamento baseado nas condições já apresentadas. O processo ficará bloqueado até que haja uma tupla para casamento , a qual não será retirada do espaço de tuplas, mas apenas "lida". Retorna-se a tupla resultante do casamento .

- rdp: <Tupla>

Este método na classe Espaço implementa a primitiva *rdp* de Linda, conforme apresentado anteriormente. Haverá ou não casamento baseado nas condições já apresentadas. Distinto de *rd*, no caso de *rdp* o processo não ficará bloqueado caso não haja uma tupla para casamento . Neste caso, retorna-se *nil*, e não a tupla resultante do casamento .

7.2 - Exemplo

Um dos problemas clássicos apresentados na bibliografia para testar a potencialidade na modelagem de problemas paralelos é o jantar dos filósofos [Ben-Ari90] [Andrews91]. A modelagem deste problema em C-Linda pode ser encontrada em [Carriero89a], enquanto [Matsuoka88] apresenta outra possibilidade. Este último, no entanto, não mostra claramente como os processos são criados, talvez pela própria inexistência da primitiva *eval*. Optamos, então, por traduzir a implementação em C-Linda apresentada em [Carriero89a] para LindaTalk. Segue o código real, conforme consta na nossa implementação.

```
exemploFilosofos
"Exemplo do uso de Linda. Implementou-se aqui o exemplo dos
filosofos, listado em Linda in Context; Communications of the ACM,
April 1989, Vol 32, Number 4.
-----------------------------------------------------
"    | numFilosofos |

    numFilosofos := 5.            "Numero de filosofos do problema"
    ('numeroDeFilosofos' || numFilosofos ) out.     "Constantes sao
                                                  colocadas no Espaco"
    1 to: numFilosofos do: [ :i |
            ('garfo' || i ) out.              "Cria o iesimo garfo"
            ('filosofo' || i ) out.       "Afixa o identificador do
                                                    filosofo"
            ('filosofo' || [self filosofo] ) eval. "Cria a tupla
                                viva, com o comportamento do filosofo"
            i < numFilosofos
                ifTrue: [ #( 'ticket') comoTupla out.
                                    "Forma alternativa de criacao de
                                tupla (a partir de uma colecao).
                                Distribui os tickets para sentar à
                                mesa"
                        ]
    ].
```

Listagem 7.10 - Exemplo de Jantar dos Filósofos em LindaTalk

```
filosofo
"   Este metodo implementa a vida do filosofo propriamente dita.
Veja metodo  exemploFilosofos.
-----------------------------------------------------
"    | posicao fils numFilosofos meuId eu|
fils := ('numeroDeFilosofos' || Integer ) rd.   "Obtem numero de
filosofos"
numFilosofos := fils at: 2.
eu := ('filosofo' || Integer ) in.  "Obtem meu identificador"
meuId := eu @ 2.  "Forma alternativa de acessar um elemento"
posicao := (meuId - 1 * 80) @ 0. "Posicao na tela para
                                    mostrar meu estado"
[true] whileTrue: [                                 "Repete sempre"
   8000 timesRepeat: [].

   'pensando  ' displayAt: posicao.  "Ficou um tempo pensando"
   8000 timesRepeat: [].
   #( 'ticket') comoTupla in.         "Obtenho ticket para comer"
   ('garfo' || meuId ) in.            "Pego garfo da esquerda"
   ('garfo' || ((meuId \\ numFilosofos) + 1) ) in. "Pego garfo da
                                                    direita"
   'comendo  ' displayAt: posicao.    "Ficou um tempo comendo"
   8000 timesRepeat: [].

   ('garfo' || meuId) out.            "Largo garfo da esquerda"
   ('garfo' || ((meuId \\ numFilosofos) + 1)) out. "Largo garfo da
                                                   direita"
   #( 'ticket') comoTupla out.        "Devolvo ticket para comer"

]
```

Listagem 7.11 - Comportamento de um Filósofo em LindaTalk

7.3 - Conclusões Finais sobre LindaTalk

A combinação de objetos no sentido clássico, como existentes em Smalltalk, e processos coordenados e sincronizados pelo modelo Linda parece-nos uma abordagem bastante interessante. Se por um lado a programação orientada a objetos clássica negligencia a modelagem de atividades paralelas, Linda vem justamente suprir esta necessidade. O mesmo pode-se dizer quanto à programação paralela. A ausência de facilidades para modelar abstrações em tais ambientes, tais como Occam, Conic, etc, faz de Smalltalk um poderoso paradigma. A combinação das duas potencialidades parece-nos preencher uma grande lacuna na modelagem/implementação de tais sistemas.

Já modelos como ConcurrentSmalltalk e ABCL/1 buscam prover um mecanismo uniforme para representar abstrações, no sentido de objetos, mas que são paralelas entre si, de forma análoga a processos. Tal unificação certamente livra o programador da dicotomia de expressar seu modelo em dois níveis distintos, um a nível de entidades e outro a nível de atividades, onde os modelos não parecem se combinar a contento. Nos ambientes que suportam programação paralela orientada a objetos, tais conceitos são unificados buscando simplicidade. É importante notar, entretanto, que tais modelos têm ainda difícil aceitação na utilização de projetos reais. Tais ferramentas precisam ainda ser desenvolvidas de tal forma que possam ser utilizadas em situações de campo assim como, por exemplo, as implementações comerciais de Smalltalk já o são. LindaTalk, por extender implementações Smalltalk já existentes, permite explorar este tipo de perspectiva em projetos reais.

A caracterísca fundamental de LindaTalk, no entanto, é a proposta de uma forma alternativa de interação de atividades paralelas, que não o envio direto de mensagens. A tentativa, aqui, é justamente procurar superar as limitações do envio de mensagens.

Embora a implementação atual de LindaTalk não seja distribuída[56], temo-la utilizado como núcleo para a implementação de Ágora[57] [Melgarejo92] [Marchini93], com resultados interessantes.

Um caminho promissor na continuação do trabalho de LindaTalk é a sua implementação de forma distribuída. Na data da publicação deste trabalho, LindaTalk começa a ser portado para ObjectWorks\Smalltalk, com suporte para TCP/IP [Comer91]. Espera-se implementar espaços de tuplas distribuídos sobre redes locais em TCP/IP, o que deve aumentar em muito as potencialidades de utilização do modelo Linda em Smalltalk. Uma possível linha de investigação neste sentido é a implementação de objetos Smalltalk de forma distribuída, de forma análoga a estruturas de dados distribuídas em C-Linda.

A formalização do modelo nos parece outra linha interessante de pesquisa neste trabalho. Neste caso, acreditamos serem necessários dois níveis de formalismo. Um deles seria para descrever o modelo de computação da linguagem em que Linda é implementada (no nosso caso, Smalltalk). Conformidade de "tipos", aqui , seria um tema crucial, dada a sua importância no casamento de tuplas. Um outro formalismo necessário seria a descrição do comportamento das primitivas, de forma a deixar precisa a sua interpretação.

---

[56] Ver 8.2.4.

[57] Ágora é um ambiente educacional que, através de micromundos [Papert80], promove a modelagem distribuída de sistemas. Embora o modelo de comunicação de processos (atores na nomenclatura Ágora) não seja o mesmo de LindaTalk, o modelo empregado em Ágora é fruto das experiências com LindaTalk.

Sendo LindaTalk uma combinação de Smalltalk com Linda, acreditamos que LindaTalk "herde" os benefícios apontados tanto pela orientação a objetos (e Smalltalk em última instância) detalhada no capítulo 2 quanto pela programação paralela e distribuída com Linda, detalhada no capítulo 5.

8 - Implementação de LindaTalk

Apresentamos, aqui, a implementação propriamente dita de LindaTalk. Ela é feita em Smalltalk/V 286, e divide-se em duas partes. A primeira delas implementa um escalonador de processos por fatia de tempo em Smalltalk/V 286, para que os processos criados a partir do *eval* sejam executados "em paralelo". A segunda parte é a implementação do conjunto de classes e métodos que formam o LindaTalk propriamente dito.

Para se portar o LindaTalk para outras plataformas Smalltalk, acreditamos ser necessária apenas a segunda parte, uma vez que a maioria das implementações Smalltalk apresenta um mecanismo de escalonamento de processos por fatia de tempo.

8.1 - Escalonador de Processos

Smalltalk[58] permite que se criem vários processos (instâncias da classe Process). Conforme visto anteriormente neste trabalho (ver listagem 2.2), um processo é (criado e) disparado enviando-se a mensagem fork a um bloco.

8.1.1 - Escalonamento Preemptivo

A existência de apenas uma CPU para execução de diversos processos levou Smalltalk a apresentar um modelo de processos preemptivos baseados em prioridades. Caso um processo esteja sendo executado pela CPU e um novo processo torne-se apto a ser executado (a ser visto adiante), aquele que roda poderá ou não ser interrompido (caracterizando a preempção). Caso a prioridade do novo processo seja maior ou igual à prioridade do processo que roda, haverá preempção; caso contrário, não.

Este mecanismo permite, por exemplo, tratar eventos assíncronos gerados por periféricos como *mouse*, teclado, etc. Na verdade, esta é a forma como o modelo de interface de Smalltalk funciona. Um processo de prioridade médiana fica constantemente esperando por novos eventos, para tratá-los. Esta espera não é feita por *busy waiting* [Andrews83], mas sim através de um mecanismo combinado de geração de interrupção e semáforos, a ser visto adiante.

Então existe, em Smalltalk/V 286, uma forma alternativa de criar e disparar processos: especificando em que prioridade irão ser executados. Um exemplo segue abaixo.

```
[ [true]
    whileTrue: [Time now printString displayAt: 0@0]
] forkAt: 2.
```

Listagem 8.1 - Prioridades de Processos em Smalltalk

É interessante notar que o processo de mais baixa prioridade aparentemente é bloqueado quando, por exemplo, mexe-se o mouse do computador. Por que isso ocorre ?

A princípio, Smalltalk executa um processo ocioso (*idle*). Ele tem a menor prioridade possível, e é executado somente quando não existe nenhum outro processo a ser executado.

[58] Usaremos o termo Smalltalk aqui sempre para denotar Smalltalk/V 286, da Digitalk.

Ao ser gerado um evento qualquer (geralmente decorrente de uma interrupção direcionada à máquina virtual de Smalltalk) entra o mecanismo preemptivo de Smalltalk. Como o processo que cuida da interface (que responde a eventos de teclado, mouse, etc) tem prioridade superior ao processo ocioso, este último é retirado da CPU, e o processo interface pode tratar o evento ocorrido. Após faze-lo, o processo interface volta a bloquear-se, à espera de novos eventos, e a CPU é passada ao próximo processo a executar (ocioso, no nosso caso). No exemplo anterior, foi disparado um processo de baixa prioridade (menor que a prioridade da interface) que fica, constantemente, mostrando a hora na tela. Neste caso o processo ocioso não está rodando, pois há um processo a ser executado. Ao ocorrer um evento de interface (movimento do mouse) nosso "processo-relógio" será interrompido e retirado da CPU. Somente ao cessarem os eventos de interface é que ele será novamente carregado na CPU, continuando a mostrar as horas. Isso dá a impressão que o processo é bloqueado ao movimentar-se o mouse, por exemplo.

8.1.2 - Fatias de Tempo

No nosso caso, gostaríamos de executar vários processos a uma mesma prioridade, e que eles se revezassem na CPU em fatias igualitárias de tempo. Isso nos permitiria ter a ilusão de paralelismo, mesmo com uma CPU apenas. Desta forma, a implementação de LindaTalk apresenta um maior realismo, mesmo não sendo distribuída ainda. Para tal, é necessário compreender o mecanismo de tratamento de interrupção de Smalltalk.

8.1.3 - Interrupções em Smalltalk

Interrupções são o mecanismo usado para notificar a ocorrência de eventos externos, assíncronos, à máquina virtual de Smalltalk. São exemplos de tais eventos: pressionamento de uma tecla, movimento do mouse, pressionamento de botão de mouse, eventos de tempo[59], etc.

O modelo de interrupção de Smalltalk corresponde a arquiteturas típicas de *hardware*. Quando uma interrupção ocorre e as interrupções estão habilitadas, estas são desabilitadas e a mensagem *#vmInterrupt:* é enviada ao objeto do topo da pilha do processo atual[60]. O argumento passado a*#vmInterrupt:* é justamente outro seletor de mensagem, caracterizando que tipo de interrupção ocorreu. Por exemplo, para interrupção de teclado, é passado o seletor *#keyboardInterrupt*. Para eventos de tempo, é passado *#timerInterrupt*. Demais tratadores podem ser encontrados em [Digitalk-286].

Como nenhum objeto, a princípio, reimplementa *#vmInterrupt:*, é utilizado o tratador *default*: envia-se o seletor que veio como parâmetro à classe Process. Assim, tratadores de interrupção são definidos através de um nome de seletor, e implementados na forma de método de classe de Process.

---

[59] São justamente estes eventos de tempo (*clock ticks*) que nos permitirão implementar o escalonador baseado em fatias de tempo.

[60] Ao enviar uma mensagem, um objeto sempre espera imediatamente pela resposta. Esta situação corresponde a uma linguagem procedural, onde a qualquer ponto no tempo existe uma pilha de chamadas não completadas ainda de procedimentos, e há um único procedimento ativo. Em Smalltalk, existe uma pilha de chamadas incompletas de métodos para cada processo, e um único método ativo em cada processo.

Um tratamento típico é simplesmente sinalizar um semáforo, o que acordará um processo para tratamento da interrupção ocorrida. Segue exemplo.

```
keyboardInterrupt
    "Trata eventos de teclado"

KeyboardSemaphore signal.
```

Listagem 8.2 - Tratamento de Interrupção de Teclado em Smalltalk

8.1.4 - Usando Interrupções de Tempo para Escalonamento

A forma de se implementar um escalonador de processos por fatia de tempo, então, é implementar um tratador para *#timerInterrupt*. Basta, então, implementar o método *#timerInterrupt* na classe Process adequadamente.

A idéia consiste em criar um processo de mais alta prioridade no sistema[61], o qual irá provocar o escalonamento adequado dos processos.Este processo fica sendo executado sempre, bloqueando-se simplesmente. Ele tem a forma:

```
[
  [true] whileTrue: [
          TimerSemaphore wait.
          ]
] forkAt: 7.
```

Listagem 8.3 - Um Escalonador de Processos Simplificado em Smalltalk

Assim, entrando e saindo constantemente da CPU, tal processo provoca o "giro" da fila de processos com prioridade menor ou igual a ele, provocando escalonamento. Este processo é acordado, então, pela interrupção de tempo. Assim, a implementação de *#timerInterrupt* deve ser algo como:

```
timerInterrupt
    "Trata eventos de tempo"

TimerSemaphore signal.
```

Listagem 8.4 - Tratamento de Interrupção de Relógio em Smalltalk

Note-se que as interrupções de tempo devem ser ligadas[62] sempre que se carrega o imagem de Smalltalk. Isso pode ser feito, por exemplo, colocando tal comando no arquivo GO. Apesar de simples, esta é basicamente a idéia por trás de nosso escalonador de processos.

---

[61] Na verdade, com prioridade maior do que a prioridade mínima que deverá apresentar comportamento por fatias de tempo. Por exemplo, se desejamos que da prioridade 1 a 6 os processos sejam escalonados por fatia de tempo, mas acima de 6 tenham o comportamento original de Smalltalk, deveríamos criar o processo escalonador com prioridade 6 também.

[62] Olhe métodos de classe na classe Time. Lá surgem métodos tanto para habilitar quanto desabilitar interrupções de tempo.

8.2 - LindaTalk

De posse de um mecanismos de criação e execução de processos baseados em fatias de tempo e de semáforos, a implementação não-distribuída de LindaTalk é trivial.

8.2.1 - Espaços de Tuplas

Na nossa implementação, os espaços de tuplas são implementados na forma de dicionários. A chave do dicionário é um inteiro, representando a aridade da tupla[63]. O valor associado, por sua vez, é uma partição do espaço de tuplas onde todas as tuplas possuem o mesmo número de elementos. Embora simples, este esquema tem-se mostrado bastante efetivo, suportando o Ágora [Melgarejo92] [Marchini93] por exemplo. Um campo em aberto aqui é justamente um mecanismo de *hashing* eficiente que respeite as relações de subclasses/superclasses possíveis.

O casamento de tuplas, por sua vez, baseia-se nas aridades das tuplas e nas relações "=" e "isKindOf:" de Smalltalk.

8.2.2 - Eval

A implementação da primitiva *eval* é bastante simples. Ao ser evocado *eval*, cria-se um processo raiz. Este é responsável por identificar, dentro da tupla, que parâmetros são blocos (e, consequentemente, originarão novos processos) e disparará-los na forma de processos paralelos. Para cada sub-processo disparado, cria-se um semáforo que será sinalizado quando o sub-processo terminar. O processo raiz bloqueia-se em todos estes semáforos, só acordando, então, quando todos os sub-processos do *eval* tiverem terminado. Neste ponto, a tupla transforma-se em tupla ordinária, e o processo raiz adiciona-a no espaço de tuplas e então termina.

8.2.3 - Semântica de Cópia e de Referência

Em Smalltalk, objetos seguem a semântica de referência. Isso permite, por exemplo, que um mesmo objeto possa fazer parte (na agregação) de dois objetos distintos, sendo compartilhado. Isso faz sentido em sistemas fortemente acoplados, como é o caso de Smalltalk. Na nossa implementação atual, objetos são tratados desta forma, sem maiores problemas.

Para a implementação distribuída, contudo, é importante notar que objetos migrarão de nodo para nodo da rede, constituindo semântica de cópia para objetos que satisfaçam casamentos de tuplas. Nestes casos, é preciso implementar um método que seja capaz de serializar um objeto, e vice-versa. Tal método deve ser capaz de detectar cilcos no grafo de referência dos objetos, para posterior reconstrução adequada.

Implementamos tal característica em LindaTalk, embora não a tenhamos utilizado ainda. Basicamente, a idéia consiste em percorrer o grafo de referências de um objeto, transformando sua representação em uma sequência de *bytes*, para possibilitar sua transmissão em qualquer stream de dados (disco, soquete TCP/IP, etc). Utilizando um IdentityDictionary, pode-se

[63] Poder-se-ia usar Array, mas como normalmente um espaço conterá valores discretode de aridade (ex. 1, 7 12) optou-se por esta representação mais compacta.

detectar estruturas recursivas, armazenando-se então apenas um indicador (flag) de um objeto que já foi armazenado. Ao ser carregado, tal indicador (flag) indica que o objeto a ser carregado na verdade já existe (foi carregado previamente) e deve ser compartilhado. Esta funcionalidade é fundamental ao portar-se LindaTalk de forma distribuída sobre TCP/IP, por exemplo. Dependendo da plataforma Smalltalk, contudo, este tipo de utilitário já está presente (BOSS de ObjectWorks\Smalltalk, por exemplo).

Uma abordagem alternativa é tentar manter a semântica de referência. Aqui, objetos manteriam suas identidades, migrando de nodo para nodo. Esta funcionalidade assemelha-se ao mecanismo provido por Emerald [Hutchinson87], e pode ser implementada através de *proxys*[64] em Smalltalk [LaLonde90].

8.2.4 - Eficiência e Distribuição: Presente e Futuro

LindaTalk não foi implementado para mostrar que Linda pode ser implementada eficientemente. Para esta finalidade existem as implementações comerciais de C-Linda [Gelernter86] [Bogoch90]. Nem para mostrar que Linda pode ser implementada de forma distribuída, o que é demonstrado em [Ahuja86] [Gelernter90] [Leler90].

LindaTalk busca mostrar que um paradigma simples de programação paralela e distribuída pode ser incorporado a implementações existentes de Smalltalk. De certa forma este trabalho assemelha-se com DistributedSmalltalk [FAQa], uma implementação comercial eficiente para suporte à programação distribuída em Smalltalk, mas com um modelo de comunicação diferente do modelo de espaço de tuplas empregado em LindaTalk.

Sabendo que Linda pode ser implementada eficientemente e de forma distribuída, bem como Smalltalk, parece-nos claro que LindaTalk pode ser implementada eficientemente e de forma distribuída também. Este tipo de preocupação, no entanto, está fora do escopo deste trabalho. Deve, contudo, ser levado em conta caso se deseje transformar LindaTalk em produto comercial (o que não é o caso na data de publicação dester trabalho).

A não-disponibilidade de facilidades de rede em Smalltalk/V 286[65] nos impossibilitou de levar adiante uma implementação-protótipo de forma distribuída. Assim como DistributedSmalltalk, acreditamos que LindaTalk devesse ser implementado em ObjectWorks\Smalltalk[66]. Dessa forma, varias classes poderiam ser reusadas, e nem mesmo o mecanismo de serialização de objetos na forma de *bytes*, para transmissão via rede, precisaria ter sido implementado (veja 8.2.3). A disponibilidade de primitivas para soquetes (TCP/IP) em ObjectWorks\Smalltalk, associada ao mecanismo de Binary Object Storage System permitiria implementar LindaTalk de forma distribuída praticamente da mesma forma que o trabalho apresentado em [Gelernter90]. É nesta linha que pretendemos levar adiante o trabalho com LindaTalk.

---

[64] Objetos que funcionam como procuradores de terceiros. Eles assumem a identidade do objeto original, delegando a estes (via rede, no caso) as mensagens recebidas.

[65] Ambiente onde LindaTalk foi originalmente desenvolvido.

[66] Na época não disponível a nós.

9 - Referências Bibliográficas

[Agha87] Concurrent Programming Using Actors; Agha, Gul; Hewitt, Carl; in [Yonezawa87a]

[Ahuja86] Linda and Friends; Ahuja, Sudhir; Gelernter, David; IEEE Computer, August 1986; pages 26-34

[Andrews83] Concepts and Notations for Concurrent Programming; Andrews, Gregory R.; Schneider, Fred B.; Computing Surveys; Vol. 15; No 1; March 1983; pages 3-43

[Andrews91] Concurrent Programming; Andrews, Gregory R.; The Benjamin/Cummings Publishing Company, Inc; 1991 ; ISBN 0-8053-0086-4

[Arango89] TSnet: A Linda Implementation for Networks of Unix-based Computers; Arango, M.; Research Report YALE/DCS/RR-739, Yale University, August 1989

[Ben-Ari90] Principles of Concurrent and Distributed Programming; Ben-Ari, M.; Prentice Hall; 1990; ISBN 0-13-711821-X

[Bogoch90] Supercomputers Get Personal; Bogoch, Sam; Bason, Iain; Williams, Jeff; Russel, Mike; BYTE Magazine, Maio 1990; pages 231-237

[Booch88] Object Oriented Design with Applications; Booch, Grady; The Benjamin/Cummings Publishing Company, Inc; 1988

[Business93] Finally, The Buzz is About Smalltalk; Business Week; April 19, 1993

[BYTE81] BYTE Magazine, Special Issue on Smalltalk; August 1981; Vol 6; No. 8

[Carriero89a] Linda in Context; Carriero, Nicholas; Gelernter, David; Communications of the ACM; April 1989,Volume 32, Number 4, pages 444-458

[Carriero89b] How to Write Parallel Programs: A Guide to the Perplexed; Carriero, David; Gelernter, David; ACM Computing Surveys; Vol 21; No. 3; Semptember 1989; pages 323-355

[Comer91] Internetworking with TCP/IP. Vol I: Principles, Protocols and Architecture; Comer, Douglas E.; Second Edition; Prentice-Hall International 1991

[Date89] Introdução a Sistemas de Bancos de dados; Date, C. J.; ISBN 85-7001-392-2; Editora Campus Ltda 1989

[Deutsch81] Building Control Structures in the Smalltalk-80 System; Deutsch, L. Peter; in [BYTE81]

[Diederich87] Experimental Prototyping in Smalltalk; Diederich, Jim; Milton, Jack; IEEE Software; May 1987; pp 50-64

[Digitalk-286] Smalltalk/V 286 - Object-Oriented Systems; Tutorial and Programming Handbook; Digitalk Inc.

[Digitalk-Win] Smalltalk/V Windows - Object-Oriented Systems; Tutorial and Programming Handbook; Digitalk Inc.

[FAQa] Frequently Asked Questions; comp.object Newsgroup

[FAQb] Frequently Asked Questions; comp.lang.smalltalk Newsgroup

[Fikes85] The Role of Frame-Based Representation in Reasoning; Fikes, Richard; Kehler, Tom; Communications of the ACM; Vol 28, No 9; pp 904-920

[Fleming93] The C+@ Programming Language; Fleming, Jim; Dr Dobb's Journal; October 1993; pp 24-32

[Gelernter85] Generative Communication in Linda; Gelernter, David; ACM Transactions on Programming Languages and Systems; Vol 7, No 1, January 1985, Pages 80-112

[Gelernter86] Domesticating Paralellism; Gelernter, David; IEEE Computer, August 1986; pages 12-16

[Gelernter90] Spending your Free Time; Gelernter, David; Philbin, James; BYTE Magazine, Maio 1990; pages 213-219

[Goldberg81] Introducing the Smalltalk-80 System; Goldberg, Adele; in [BYTE81]

[Goldberg87] Programmer as Reader; Goldberg, Adele; IEEE Software; 1987; pp 62-70

[Goldberg90] Taming Object-Oriented; Goldberg, Adele; Computer Language; October 1990; pp 34-45

[Harbinson92] Safe Programming with Modula-3; Harbinson, Sam; Dr Dobb's Journal; October 1992; pp 88-96

[Hoare85] Communicating Sequential Processes; Hoare, C. A.; ISBN 0-13-153271-5; Prentice-Hall International 1985

[Hutchinson87] Emerald: An Object-Based Language for Distributed Programming; Hutchinson, Norman C.; Department of Computer Science, University of Washington, Seattle, WA 98195, USA, Technical Report 87-01-01

[Ingalls81] Design Principles Behind Smalltalk; Daniel H Ingalls ; in [BYTE81]

[Jellinghaus90] Eiffel Linda: An Object-Oriented Linda Dialect; Jellinghaus, Robert; ACM SIGPLAN Notices; Vol 25, No. 12, December 1990; pages 70-84

[Jensen75] Pascal User Manual report; Jensen; K.; Wirth, N.; ISBN 0-387-90144-2; Springer-Werlag 1975

[Kaehler81] Virtual Memory for an Object-Oriented Language; Kaehler, Ted; in [BYTE81]

[Kaehler81] Virtual Memory for an Object-Oriented Language; Kaehler, Ted; in [BYTE81]

[Kay90] User Interface: A Personal View; Kay, Alan; in [Laurel90].

[Kent79] Limitations of Record-Based Information Models; Kent, William; ACM Transactions on Database Systems; Vol 4; No 1; March 1979; pp 107-131

[Kirkerud90] Object-Oriented Programming with Simula; Kirkerud, Bjorn; ISBN 0-201-17574-6; Addison-Wesley 1990

[Kramer83] CONIC: An Integrated Approach to Distributed Computer Control Systems; Kramer, J.; Magee, J.; Sloman, M.; Lister, A.; IEEE Proceedings; Vol. 130; No 1; January 1983

[Krasner81] The Smalltalk-80 Virtual Machine; Glenn Krasner; in [BYTE81]

[LaLonde90] Inside Smalltalk; LaLonde, Wilf R.; Pugh, John R,; Volume I; Prentice Hall; 1990; ISBN 0-13- 468414-1

[Laurel90] The Art of Human-Computer Interface Design; Laurel, Brenda; Addison-Wesley Publishing Company, Inc.; 1990; ISBN 0-201-51797-3

[Leler90] Linda Meets Unix; Leler, Wm; IEEE Computer; February 1990; pages 43-54

[Lieberman87] Concurrent Object-Oriented Programming in Act 1; Lieberman, Henry; in [Yonezawa87a]

[Maier86] Development of an Object-Oriented DBMS; Maier, Maier; Stein, Jacob; Otis; Allen; Purdy, Alan; OOPSLA'86 Proceedings; pp 472-482

[Marchini93] Programação Paralela: Reflexões sobre Ágora; Marchini, Marcio Q. ; Melgarejo, Luiz F. B.; Anais do IV Simpósio Brasuleiro de Informática na Educação (a ser publicado); Recife, PE, 8 a 10 de dezembro de 1993

[Matsuoka88] Using Tuple Space Communication in Distributed Object-Oriented Languages; Matsuoka,Satoshi; Kawai, Satoru; Proceedings of Object Oriented Programming Systems Languages and Applications; 1988;pages 276-284

[Mattos91] An Approach to Knowledge Base Management; Mattos, N. M.; ISBN 3-540-54268-X; Springer-Werlag 1991

[Melgarejo92] Micromundos: Paralelismo e Comunicação entre Agentes; Melgarejo, Luiz F. B.;Marchini, Marcio Q. ; Anais do III Simpósio Brasuleiro de Informática na Educação; pp 28-37; Rio de Janeiro, RJ, Outubro de 1992

[Meyer88] Object-Oriented Software Construction; Meyer, Bertrand; Prentice-Hall Int,; 1988

[Mundie86] Parallel Processing in Ada; Mundie, David A. ; Fisher, David A.; IEEE Computer, August 1986; pages 20-25

[OTI-ENVY] ENVY/Smalltalk Embedded System Development; Folder; Object Technology International Inc.

[Papert80] LOGO: Computadores e Educação; Papert, Seymour; Editora Brasiliense; 1980

[Parcplace] Designing for Change; Folder; ParcPlace Systems

[ParcPlace92a] ObjectWorks\Smalltalk User's Guide; ParcPlace Systems; 1992

[ParcPlace92b] VisualWorks User's Guide; ParcPlace Systems; 1992

[Peckam88] Semantic Data Models; Peckam, Joan; Maryanski, Fred; ACM Computing Surveys; Vol 20; No 3; September 1988; pp 153- 189

[Pountain87] A Tutorial Introduction to Occam; Pountain, D.; May, D.; McGraw-Hill, New York, 1987

[Rich88] Inteligência Artificial; Rich, Elaine; McGraw-Hill, 1988

[Robson81] Object-Oriented Software Systems; David Robson; in [BYTE81]

[Rocha87] Análise e Projeto Estruturado de Sistemas; Rocha, Ana R. C.; ISBN 85-7001-431-7; Editora Campus Ltda 1987

[Rumbaugh91] Object-Oriented Modeling and Design; Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; Lorensen, William; Prentice Hall; 1991; ISBN 0-13-629841-9

[Shibayama87] Distributed Computing in ABCL/1 ; Shibayama, Etsuya;Yonezawa, Akinori; in [Yonezawa87a]

[Sloman86a] Flexible Communication Structure for Distributed Embedded Systems; Sloman, M.; Kramer, J.; Magee, J.; Twidle, K.; IEEE Proceedings; Vol. 133; Pt. E; No. 4; July 1986

[Sloman86b] The Conic Toolkit for Building Distributes Systems; Sloman, M.; Kramer, J.; Magee, J.; Anais do 4º Simpósio Brasileiro de Redes de Computadores; Março 1986; pages 148-157

[Stemple86] Functional Addressing in Gutenberg: Interprocess Communication without Process Identifiers; Stemple, David A; Vinter, Stephen T.; Ramamritham, Krithivasan; IEEE Transactions on Software Engineering, Vol. SE-12, No. 11, November 1986, pages 1056-1066

[Stroustrup86] The C++ Programming Language; Stroustrup, Bjarne; ISBN 0-201-12078-X; Addison-Wesley 1986

[Takahashi88] Introdução à Programação Orientada a Objetos; Takahashi, Tadao; III Escola Brasileiro-Argentina de Informática, 1988

[Tesler81] The Smalltalk Environment; Larry Tesler; in [BYTE81]

[Tozer87] Prototyping as a System Development Methodology: Opportunities and Pitfals; Tozer, Jane A.; Information and Software Technology; Vol 29, No 5, June 1987; pp 265-269

[Ungar87] Self: The Power of Simplicity; Ungar, David; Smith, Randall B.; Proceedings of Object Oriented Programming Systems Languages and Applications; 1987; pages 227-242

[Wegner89] Learning the Language; Wegner, Peter; BYTE Magazine; March 1989

[Wegner89] Introduction to the Special Issue on Programming Language Paradigms; Wegner; Peter; ACM Computing Surveys; Vol 21; No. 3; Semptember 1989; pages 253-258

[Wirth85] Programming in Modula-2; Wirth, Niklaus; Springer-Verlag; 1985; ISBN 0-387-15078-1

[Xerox81] The Smalltalk-80 System; The Xerox Learning Reasearch Group; in [BYTE81]

[Yokote87] Concurrent Programming in Concuurent Smalltalk; Yokote, Yasuhiko; Tokoro, Mario, in [Yonezawa87a].

[Yonezawa87a] Object-Oriented Concurrent Programming; Yonezawa, Akinori; Tokoro, Mario; MIT Press; 1987; ISBN 0-262-24-26-2

[Yonezawa87b] Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1; Yonezawa, Akinori; Shibayama, Etsuya; Takada, Toshihiro; Honda, Yasuaki; in [Yonezawa87a]

[Yonezawa87c] Object-Oriented Concurrent Programming: An Introduction; Yonezawa, Akinori;Tokoro, Mario, in [Yonezawa87a].

[Yonezawa90a] ABCL - An Object-Oriented Concurrent System; Yonezawa, Akinori; MIT Press; 1990; ISBN 0-262-24029-7

Bibliografia

Programming Languages for Distributed Computing Systems; Bal, Henry E. ; Steiner, Jennifer G.; Tanembaum; Andrews S.; ACM Computing Surveys; Vol 21; No. 3; Semptember 1989; pp 261-322

Conception, Evolution and Application of Functional Programming Languages; Hudak, Paul; ACM Computing Surveys; Vol 21; No. 3; Semptember 1989; pp 359-411

Object Oriented Programming in the Beta Programming Language; Kristensen, Bent Bruun; Madsen, Ole Lehrmann; Moller-Pedersen, Birger; Draft Version, September 1992

Distributed Application Support: Survey and Synthesis of Existing Approaches; Schill, A; Information and Software Technology; Vol 32; No 8; October 1990; pp 545-558

High-Level Programming of Real-Time Systems with Asynchronous Communication; Serbedzija, N. B. ; Information and Software Technology; Vol 32; No 7; September 1990; pages 497-505

The Family of Concurrent Logic Programming Languages; Shapiro, Ehud; ACM Computing Surveys; Vol 21; No. 3; Semptember 1989; p 412

When Can Knowledge-Based Systems be Called Agents? ; Sichman, Jaime S.; Demazeau, Yves; Boissier, Olivier; Anais do IX Simpósio Brasileiro de Inteligência Artificial; Outubro de 1992; Rio de Janeiro ; pp 172-185