LindaTalk : Combinando Linda com Smalltalk

Márcio Quintaes Marchini[1]
e-mail: mqm@edugraf.ufsc.br
Universidade Federal de Santa Catarina
EDUGRAF - INE - CTC - UFSC
Campus Trindade
Florianópolis SC 88040-900

Sumário

Modelos com suporte à programação paralela orientada a objetos, tais como Emerald, ConcurrentSmalltalk, Act-1, ABCL/1, etc procuram unificar objetos no sentido clássico da orientação a objetos com a noção de processos paralelos 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. Esta metáfora, conforme presente nos atuais sistemas concorrentes orientados a objetos, apresenta diversas fraquezas no que tange a modelagem. Nossa proposta, ao contrário, busca incorporar a uma linguagem orientada a objetos (Smalltalk) um modelo que suporte a programação paralela/distribuída com um maior grau de flexibilidade. Aqui, combina-se o modelo de Espaço de Tuplas de Linda com Smalltalk, o que adiciona uma nova dimensão de programação e modelagem a qualquer implementação existente de Smalltalk, seja ela acadêmica ou comercial.

Apresentamos primeiramente aspectos da troca de mensagens. Em seguida, descrevemos o modelo de Espaço de Tuplas de Linda. Logo após, apresentamos alguns princípios básicos de Smalltalk, e então LindaTalk propriamente dito. Terminamos com uma indicação de propostas de Linda em ambientes orientados a objetos, bem como possíveis direções na continuação deste nosso trabalho.

1 - Introdução

O objetivo de sistemas computacionais é solucionar problemas. Problemas complicados são, em geral, decompostos em sub-problemas. Tal decomposição pode ser a partir de diferentes pontos de vista, tais como procedural, funcional, orientado a objetos, etc. A possibilidade de programar sistemas multiprocessadores e sistemas em redes de computadores, por outro lado, favoreceu as linhas de programação paralela/concorrente/distribuída.

Se por um lado a orientação a objetos clássica[2] 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 [ARCHITECS90], Conic [Kramer83], Ada [Mundie86], etc, permitem preencher esta lacuna. O poder de modelagem e abstração de entidades,

---

[1] Aluno do Programa de Mestrado em Engenharia de Produção da Universidade Federal de Santa Catarina

[2] Usamos o termo "clássica" para a definição dada por [Meyer88], onde objetos são tidos como implementações de tipos abstratos de dados.

entretanto, fica bastante limitado neste tipo de sistemas, levando geralmente à produção de programas difíceis de adaptar, manter e reusar [Meyer88].

Modelos com suporte à programação paralela orientada a objetos, tais como Emerald [Hutchinson87], ConcurrentSmalltalk [Yokote87], Act-1 [Lieberman87], ABCL/1 [Yonezawa87b], etc surgem na tentativa de unificar objetos no sentido clássico de orientação a objetos com a noção de processos paralelos e comunicantes. Contudo, 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.

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 : combinar o modelo de espaço de tuplas de Linda [Gelernter85] com o modelo de objetos de Smalltalk [LaLonde90].

2 - Limitações na Troca de Mensagens

Sistemas baseados em trocas de mensagens surgiram em consequência da programação de sistemas fracamente acoplados, ou seja, aqueles com espaços de endereçamento distintos. Exemplos típicos são sistemas distribuídos sobre redes de computadores. Uma característica central de sistemas baseados em trocas de mensagens é, justamente, a sincronização das mesmas. De forma geral, o envio de mensagens pode ser:

- Síncrono
- Assíncrono
- Ambos

A forma de comunicação, por outro lado, envolve relações entre os emissários e destinatários das mensagens. Dentre elas destacamos:

- 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, [Stemple86] destaca:

- 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 funcional - 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 em sistemas concorrentes orientados a objetos, podemos dizer que:

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

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 ou menos 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.

3 - Linda

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

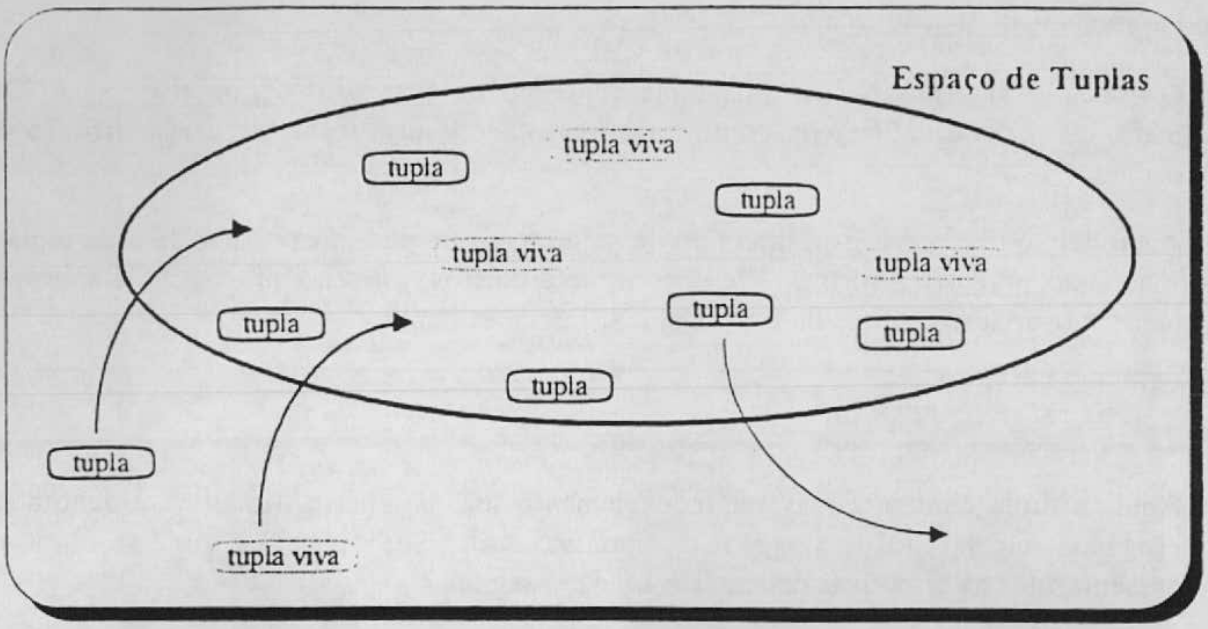
3.1 - O Modelo

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*[3] 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. O processo que produz um determinado dado relevante (chamado de tupla na nomenclatura Linda) simplesmente coloca-o à disposição em uma região chamada *Espaço de Tuplas*. O processo que deseja ter acesso ao dado simplesmente consulta 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, temos:

---

[3] O menor elemento executável dentro de um processo denomina-se *thread* [Wegner89]. É uma estrutura de dados que se torna ativa quando carregada no processador.



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 vivas

Desta forma, Linda permite modelar situações cuja granularidade de paralelismo pode ser pequena, média ou grande [Carriero89b].

3.2 - 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 [Wirth85], 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 )
```

Neste caso, a tupla é uma coleção ordenada de três elementos onde o primeiro deles é um String, o segundo elemento é um número Natural, e o terceiro elemento é um número de ponto flutuante.

Parâmetros Reais e Formais

O exemplo anterior mostra uma tupla contendo os chamados parâmetros reais. São objetos da própria linguagem, conforme apresentado. Outra forma de parâmetro são os 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[4] sobre os objetos propriamente ditos da linguagem X. Segue exemplo:

```
VAR
   idade   : CARDINAL;
```

```
...
("João", ? idade, 92.5)
```

Aqui, a tupla 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.

3.3 - Primitivas do Modelo Linda

Resumidamente, as primitivas deste modelo se destinam a:

- Adicionar uma tupla ordinária ao 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

Adicionando Tuplas Ordinárias

A primitiva *out*, em Linda, permite adicionar uma tupla ao Espaço de Tuplas. Assim, `out("Pedro", 23, 92.5)` 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.

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 citados anteriormente. Por exemplo:

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

```
...

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

[4] No caso de Linda em linguagens tipadas, esta informação é justamente o tipo de cada objeto

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:

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

☑ Todos os seus elementos devem *casar*

É preciso, então, determinar quando um elemento de uma tupla ordinária 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

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

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 apenas um elemento no espaço de tuplas, e que `in("Continue")` quando executado por outro processo, B , acarretará em 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 (ou outro processo) adicionasse a referida tupla.

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 deste tipo de casamento.

Ilustrando com Parâmetros Formais

Por outro lado, para ilustrar *b)* descrito anteriormente vamos considerar o exemplo `out("João")`. Conforme descrito anteriormente, esta tupla de um elemento apenas 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).
```

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 qualquer 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 lugar do 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* com 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 dos processos. As estruturas de dados propriamente ditas que os processos trocam entre si devem ser expressos na linguagem X que implementa o modelo Linda. Podemos, é claro combinar na primitiva *in* parâmetros reais e formais. Assim, no exemplo:

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

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)`, etc. Caso várias tuplas presentes no espaço de tuplas possam casar com a tupla descrita acima na primitiva *in*, uma delas será escolhida, de forma não-determinística. Ou seja, esta escolha não possui 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 convencional, 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.

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* 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*.

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, transformando-se em tupla ordinária, é que ela poderá ser *casada* com outra.

Em Linda, a forma de expressar uma tupla viva é com a primitiva *eval*. Por exemplo: `eval(sqrt(225))`. 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 anterior tranforma-se na tupla ordinária dada por `(15.0)`, sendo então 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 os resultados das computações 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.

3.4 - 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

A única diferença em relação às suas versões bloqueantes (*in* e *rd*) 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*.

4 - 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 tanto de 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 foi apresentado ao grande público como Smalltalk-80, em [BYTE81]. Hoje o sistema já é sucesso de mercado, sendo utilizado em aplicações variadas, desde sistemas de controle de processos até aplicações financeiras. Esse sucesso se deve à grande potencialidade oferecida pelo sistema, que apresentaremos a seguir.

4.1 - O Ambiente

Ao contrário de linguagens de programação como Modula-2, etc, Smalltalk[5] é um sistema integrado. Engloba uma interface gráfica manipulável através de um dispositivo

[5] Estas caracterisucas aqui são do projeto inicial. Diversas caracteristicas mudaram, como

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 apresenta todas as ferramentas num ambiente só. O editor de textos/programas, o compilador/gerador de código e a própria máquina virtual (Smalltalk gera um código intermediário, os chamados *bytecodes* [Krasner81]).

4.2 - O Modelo

Smalltalk apresenta uma série de conceitos que lhe são peculiares, alguns deles sendo inspirados por 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.

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. 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, objetos em Smalltalk pertencem a uma classe. Esta, por sua vez, pode possuir uma superclasse[6] e uma ou mais subclasses. Tal classificação não é feita a esmo, e sim seguindo um conjunto de princípios de modelagem [Rumbaugh91].

4.3 - Objetos e Mensagens

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

Embora a idéia de objetos que se comunicam leve intuitivamente à idéia de paralelismo, isto não ocorre no modelo de objetos "tradicional". Aqui, existe a noção de apenas um *thread* 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 Modula-2, C, etc.

Para expressar paralelismo, Smalltalk provê objetos que representam distintos *threads* de execução [Deutsch81]. Desta forma, para expressar paralelismo na interação entre os objetos que compõem um determinado programa em Smalltalk, é necessário criar

---

interfaceamento com GUIs já existentes como Microsoft Windows, OpenWindows, etc.

[6] A proposta original é de herança simples. Em Smalltalk, pode-se configurar uma classe para não possuir superclasse (nil). Esta característica é normalmente usada para implementar objetos procuradores (*proxy*) [LaLonde90].

objetos processo, os quais contêm 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 linha de Smalltalk levou a linguagens com suporte a programação orientada a objetos com uma ênfase para este sentido clássico: objetos são (grosseiramente falando) 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+@ , Modula-3, Eiffel , etc.

Por outro lado, contudo, esta limitação levou à busca de linguagens onde os objetos seriam paralelos, autônomos. Nesta classe de pesquisa enquadram-se Emerald, ConcurrentSmalltalk, ABCL/1, Act-1, etc. Aqui, objetos são (grosseiramente falando) como processos, respondendo às mensagens recebidas, potencialmente de forma paralela.

4.4 - 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 terão os objetos que a ela pertencem, serão as instâncias que efetivamente irão armazenar os 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 é 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 listados também na classe.

4.5 - Herança, Tipagem e Acoplamento Dinâmico

Em Smalltalk, assim como em 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 mesma. 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.

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 [Tadao88] provido pelos objetos, é a herança de atributos e métodos que favorece enormemente a reusabilidade de código. Mais ainda, a herança permite que sistemas complexos sejam refinados e adaptados mais rapidamente, pois comunalidades 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 [Tadao88]. Esta "tipagem" 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` denota o envio da mensagem *print* ao objeto *obj*. Olhando-se este código, 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 sentido ascendente na cadeia de herança, até achar onde tal método foi definido. Caso nenhuma das superclasses defina tal método, ocorre um "erro".

4.6 - Gerenciamento de Memória

Durante a execução do "programa" em Smalltalk, diversos objetos são criados. Tais instâncias podem deixar de existir enquanto o programa é executado. 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 (os programadores vêem-se livres da gerência manual de memória em cada uma de sua aplicações).

5 - LindaTalk

Nossa proposta é incorporar o modelo Linda à Smalltalk. O modelo de espaço de tuplas de Linda é 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 (um processo X pode comunicar-se com outro Y, que nem existe ainda, através do espaço de tuplas). Outra possibilidade é um processo B obter informações de outro A, que já deixou de existir.

Para implementar nosso modelo, duas classes principais formam criadas. Uma delas, a classe Tupla, torna as tuplas de Linda objetos do ambiente Smalltalk. Ou seja, podem ser criadas, atribuídas a variáveis, passadas como parâmetros e até mesmo integrar campos de outras tuplas. A segunda classe adicionada a Smalltalk foi Espaço (de tuplas), que permite também a instanciação de múltiplos espaços de tuplas, e não apenas um espaço global [Marchini94].

5.1 Tupla

Tuplas são identificadas através de uma lista de objetos que as compõem, separadas por "||" [7]. Assim, uma possível tupla no nosso modelo seria:

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

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' ).
```

5.1.1 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

---

[7] Optamos por "||" ao invés de "," , como é usual em dialetos como C-Linda [Carriero89] devido ao fato de "," já ter o significado de concatenação de coleções em Smalltalk. Logo, a expressão `('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 "||".

```
| tupla ponto char |

tupla := ( 'maria' || (3@5) || $t ).
ponto := tupla at: 2.          "Primeira forma"
char := tupla @ 3.             "Segunda forma"
tupla at: 3 put: 'novoElemento'.
```

Uma tupla pode ter elementos acrescentados ou removidos. O exemplo a seguir 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 )  "
```

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

5.1.2 Expressando Formais

Seguindo o exemplo de C-Linda, onde os parâmetros formais 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`. Esta tupla poderia casar, por exemplo, com a tupla: `(-45||(5@6))out` . Devido ao fato de -45 pertencer à classe Integer, há casamento, bem como com o objeto 5@6 , por pertencer à classe Point. Uma classe presente em um campo de tupla usado em primitivas como *in, inp, rd, rdp* é interpretada como parâmetro formal, e acarretará casamento se o objeto correspondente da tupla produzida com *out, outi* ou *eval* satisfaça a relação *isKindOf:* [8] de Smalltalk.

5.1.3 Casamento de Tuplas

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. Anteriormente vimos como um parâmetro formal casa com um parâmetro real em nossa implementação. Falta-nos, ainda, abordar as outras possíveis combinações.

Uma delas é justamente caso sejam dois 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` e `(Point||Integer)in` 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:

---

[8] Esta relação, implementada na forma de método de instância, deve retornar verdadeiro (true) caso o objeto se considere instância da classe dada, ou de uma de suas subclasses.

☑ São dois formais idênticos. De forma análoga a reais idênticos, é natural que casem.

☑ São dois objetos idênticos. É um caso de casamento de dois reais, conforme veremos a seguir. Também para esta interpretação parece-nos natural que devam casar.

Outra combinação possível é justamente dois objetos reais. Aqui o casamento é baseado na relação "=" [9] de Smalltalk. Assim, as tuplas: `(34||(3@2)||$t)out` e `(34||(3@2)||$t)in` casam pois os objetos satisfazem (retornam true para) a relação (método, na verdade) "=".

A terceira combinação possível é aquela onde na tupla (produzida por um com *out, outi* ou *eval)* esteja presente justamente um parâmetro formal (uma classe) , e na tupla a ser usada com *in, inp, rd, rdp* apareça um parâmetro real. Um exemplo seria: `(Point||Integer)out` e `((5@4)||45)in`. Neste caso, há casamento caso o objeto da tupla associada ao *in* pertença à classe descrita na tupla *out* (ou a qualquer uma de suas superclasses). Neste caso, contudo, não há transferência de valores entre as tuplas; apenas ocorre o casamento.

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

5.1.4 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
```

Desta forma, pode-se optar 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.

5.2 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, o exemplo: `( 34 || 'verde') out: umEspaço` transforma-se em: `umEspaço out: ( 34 || 'verde')`. Desta forma existe, na verdade, apenas uma

---

[9] Esta relação, implementada na forma de método de instância, deve retornar verdadeiro (true) caso o objeto se considere igual ao outro. Note que esta implementação pode tornar a relação assimétrica.

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 anteriormente.

outi: <tupla>

Este método na classe Espaço implementa a variação aqui proposta 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* faz com que a tupla permaneça 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. 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. Para a implementação utilizamos os objetos da classe Context de Smalltalk, denotados por "[...]". Assim, um exemplo de tupla usada com *eval* seria: `((5@6)||(4>=8)||[6 fatorial])eval.` 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, sendo 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)`.

Eval permite que computações paralelas sejam sincronizadas, pois a tupla somente será tranformada em tupla ordinária quando todos seus campos que contenham computações terminarem. Assim, no exemplo: `([obj1 updateDatabase]||[obj2 printReport])eval`, um processo que aguarde a leitura (por exemplo) desta tupla terá a garantia que ambas as computações descritas foram feitas.

in: <tupla>

Este método na classe Espaço implementa a primitiva *in* de Linda, conforme apresentado anteriormente. 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. Distinto de *in*, com *inp* o processo não ficará bloqueado caso não haja uma tupla para casamento. Não havendo casamento, retorna-se nil (objeto nulo).

rd: <tupla>

Este método na classe Espaço implementa a primitiva *rd* de Linda, conforme apresentado anteriormente. 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. Distinto de *rd*, no caso de *rdp* o processo não ficará bloqueado caso não haja uma tupla para casamento. Não havendo casamento, retorna-se nil.

6 - Exemplo

Um dos problemas clássicos apresentados na bibliografia para testar a potencialidade na modelagem de problemas paralelos é o jantar dos filósofos. A modelagem deste problema em C-Linda pode ser encontrada em [Carriero89]. Segue o código em LindaTalk, conforme [Marchini94].

```
exemploFilosofos
"Exemplo do uso de Linda. Implementou-se aqui o exemplo dos
filosofos"
| numFilosofos |

numFilosofos := 5.                    "Numero de filosofos do problema"
('numeroDeFilosofos' || numFilosofos ) out.  "Constantes sao colocada
                                              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"
                    ]    ].
```

```
filosofo
"    Este metodo implementa a vida do filosofo propriamente dita. Veja
     metodo  exemploFilosofos."
     | 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"

     [true] whileTrue: [          "Repete sempre"
     self pensa.                  "Fico um tempo pensando"

     #( 'ticket') comoTupla in.     "Obtenho ticket para comer"
     ('garfo' || meuId ) in.        "Pego garfo da esquerda"
     ('garfo' || ((meuId \\ numFilosofos) + 1) ) in.  "Pego garfo da
                                                       direita"
     self come.                  "Fico um tempo comendo"

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

Este exemplo, embora clássico, não explora completamente a possibilidade de utilização de tuplas vivas em Linda. Uma metodologia de codificação de programas paralelos em dialetos Linda é descrita em [Carriero89b].

7 - Trabalhos Relacionados

A filosofia Linda tem sido incorporada em diferentes ambientes, dando origem a dialetos destes, conforme descrito aqui. 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.

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.

Por outro lado, existem implementações Linda em linguagens orientadas a objetos. Destacamos Eiffel [Jellinghaus90] e Smalltalk [Matsuoka88]. Seu estudo vale a pena devido ao fato de uma (Eiffel) ser fortemente tipada, enquanto a outra (Smalltalk) não.

8 - O Futuro de LindaTalk

A implementação atual de LindaTalk , embora não seja distribuída, não invalida sua utilização. Temos utilizado o LindaTalk, por exemplo, como substrato para o Ágora [Melgarejo92] [Marchini93].

Um possível caminho na continuação do trabalho em LindaTalk é a sua implementação de forma distribuída. Na data deste trabalho, LindaTalk começa a ser portado para Smalltalk/V Windows[10] [Digitalk-Win], com suporte para TCP/IP [Comer91]. Espera-se implementar espaços de tuplas distribuídos (sobre 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.

Uma segunda área de estudo para LindaTalk seria a formalização do modelo. Neste caso, acreditamos serem necessários dois níveis de formalismo. Um deles 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 Linda, de forma a deixar precisa a sua interpretação.

9 - Conclusão

A combinação de objetos no sentido clássico, como existentes em Smalltalk, e processos coordenados e sincronizados como no 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 sistemas para programação paralela [Marchini94] faz de Smalltalk um poderoso aliado. A combinação dos dois paradigmas parece-nos preencher uma grande lacuna na modelagem/implementação de tais sistemas.

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. Nos ambientes que suportam programação paralela orientada a objetos, tais conceitos são unificados buscando simplicidade. A utilização do paradigma de troca de mensagens como forma de interação, contudo, parece-nos bastante limitada para diversas aplicações.

A proposta fundamental de LindaTalk é ser justamente 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 através da utilização do modelo de Espaço de Tuplas de Linda.

LindaTalk representa uma experiência rica. Sua utilização têm sido constante, e esperamos que seu aperfeiçoamento seja sempre continuado, em qualquer uma das linhas descritas acima ou outras que venham a surgir.

Agradecemos a todos aqueles que contribuíram de alguma forma com este projeto. Em especial ao pessoal do Laboratório de Software Educacional da UFSC (EDUGRAF), onde todo nosso trabalho foi desenvolvido.

---

[10] A versão atual roda em Smalltalk/V 286 [Digitalk-286].

10 - Bibliografia

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

[ARCHITECS90] A Tutorial Introduction to Occam Programming; Computer Systems Architecs 1990

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

[Carriero89] 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

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

[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.

[Gelernter85] Generative Communication in Linda; Gelernter, David; ACM Transactions on Programming Languages and Systems; Vol 7, No 1, January 1985, Pages 80-112

[Goldberg81] Introducing the Smalltalk-80 System; Goldberg, Adele; in [BYTE81]

[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

[Jellinghaus90] Eiffel Linda: An Object-Oriented Linda Dialect; Jellinghaus, Robert; ACM SIGPLAN Notices; Vol 25, No. 12, December 1990; pages 70-84

[Kay90] User Interface: A Personal View; Kay, Alan; in [Laurel90].

[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

[Lieberman87] Concurrent Object-Oriented Programming in Act 1; Lieberman, Henry; in [Yonezawa87]

[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

[Marchini94] LindaTalk: Suporte Distribuído à Programação Concorrente Orientada a Objetos; Marchini, Márcio Q.; Dissertação de Mestrado EPS-CTC-UFSC; fase de conclusão.

[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

[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

[Robson81] Object-Oriented Software Systems; David Robson; in [BYTE81]

[Rumbaugh91] Object-Oriented Modeling and Design; Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; Lorensen, William; Prentice Hall; 1991; ISBN 0-13-629841-9

[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

[Tadao88] Introdução à Programação Orientada a Objetos; Takahashi, Tadao; III Escola Brasileiro-Argentina de Informática, 1988

[Wegner89] Learning the Language; Wegner, Peter; BYTE Magazine; March 1989

[Wirth85] Programming in Modula-2; Wirth, Niklaus; Springer-Verlag; 1985; ISBN 0-387-15078-1

[Yokote87] Concurrent Programming in Concuurent Smalltalk; Yokote, Yasuhiko; Tokoro, Mario, in [Yonezawa87].

[Yonezawa87] 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 [Yonezawa87]