PUC

Monografias em Ciência da Computação
nº 24/92

O Uso de Classes como Unidade de Paralelismo

Otávio Pecego Celho (COELHO)
Sérgio E. R. Carvalho

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22454-970
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 24/92
Editor: Carlos J. P. Lucena Julho, 1992

O Uso de Classes como Unidade de Paralelismo *

Otávio Pecego Coelho
Sérgio E. R. Carvalho

* Este trabalho foi patrocinado pela Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

O USO DE CLASSES COMO UNIDADE DE PARALELISMO

Otavio Pecego Coelho e Sergio Carvalho
Departamento de Informática da Pontificia Universidade Católica RJ - Rua Marques de São Vicente 209, Gavea, Rio de Janeiro, RJ

ABSTRACT

An extension of the class mechanism for the handling of parallelism is presented. In extended classes, the behaviour of objects operating in parallel is modeled not only by common synchronous calls (methods), but also by asynchronous events. The use of generic event destinations further increases the encapsulation and abstraction properties of extended classes, allowing the free composition of objects with context-free behaviour.

RESUMO

Uma extensão do conceito de classes para o tratamento de paralelismo é apresentada. Em classes extendidas o comportamento de objetos operando em paralelo é modelado não somente pelas chamadas sincronas usuais (métodos) mas também por eventos assincronos. O uso de destinatários genéricos para eventos aumenta o grau de encapsulamento e abstração de classes extendidas, permitindo a livre composição de objetos de comportamento independente de contexto.

PALAVRAS-CHAVE
Paralelismo, orientação a objetos, eventos.

1 - INTRODUÇAO

Linguagens Orientadas a Objetos (LOO's) têm contribuido em vários aspectos no auxilio à construção de softwares de alto grau de modularidade, propiciando uma maior reutilização e extensibilidade de software. Para tanto, mecanismos de herança, polimorfismo, generalidade, etc, têm sido propostos em grande escala causando um renovado interêsse pela área de linguagens de programação.

Dentre os novos aspectos linguísticos incorporados às linguagens de programação orientadas a objetos, talvez o menos abordado tenha sido o relativo a mecanismos de paralelismo e intercomunicação entre objetos. Apesar do paradigma corrente das LOO's se basear no conceito de mensagem e este estar intimamente ligado às usuais áreas de pesquisa de paralelismo (sistemas operacionais e redes de computadores), poucas foram as incursões ao mundo do paralelismo e quase sempre via a transferência imediata dos mecanismos usuais das linguagens, como por exemplo os mecanismos de corrotina e rendez vous. Os artigos [1, 2] em especial apresentam um conjunto de linguagens e mecanismos utilizados para oferecer paralelismo nas linguagens de computação contemporâneas. Em particular, [1] apresenta uma extensa bibliografia sobre paralelismo e orientação a objetos.

Este artigo pretende abordar a questão do paralelismo em LOO's tendo em vista a manutenção das prerrogativas principais deste paradigma (modularidade e reutilização) via a extensão do conceito de mensagens. Em especial, será apresentada uma proposta de extensão que possibilita o uso de operadores de composição de objetos, de restrição e renomeação de assinaturas conforme encontrados no Cálculo de Processos de Milner [8] - o que proporciona novas formas de compor complexos a partir da aglomeração de objetos definidos em classes, aumentando, em mais um eixo, a dimensão relativa à composição modular de objetos. Além deste, dois novos aspectos não contemplados no modelo de comunicação de Milner, são introduzidos nesta proposta, com o intuito de aumentar o grau de paralelismo, modelar o conceito de

evento, e introduzir um percurso "natural" de mensagens relativas a objetos "familiares". Estes novos aspectos têm se mostrado importantes para uma melhor abstração no tratamento de objetos gráficos em Sistemas de Interface Gráficas (GUI - Graphical User Interface) [4], e se apresentam compatíveis (ortogonais) com os demais mecanismos relativos a mensagens.

2 - INSUFICIENCIAS DO MECANISMO DE MENSAGEM EM LOO's CONVENCIONAIS

Mensagem é o mecanismo usual de comunicação em LOO's. Na maior parte destas linguagens, objetos, ao receberem uma mensagem, realizam uma chamada a um de seus métodos via ligação tardia ("late binding") ou ligação estática no momento da compilação ("early binding"). Com a introdução de paralelismo em LOO's é quase natural a extensão do mecanismo de mensagens já que este foi desenvolvido exatamente por áreas da ciência da computação que há muito trabalham com multiprogramação. Hoje é comum encontrar mecanismos de processos, monitores, etc, atrelados ao conjunto de mecanismos relativos a classes e objetos, mas poucas são as linguagens em que objetos, definidos à partir de classes, podem ser considerados como unidades de paralelismo [1, 3]. Mesmo nestas, quase sempre o esforço se resume ao de incorporar mecanismos de mensagens síncronas entre objetos concorrentes, deixando de satizfazer alguns pontos relevantes como:

1) Adiamento da Composição de Objetos em Paralelo: objetos que trabalham em paralelo e que se comunicam mutuamente formam complexos de objetos (análogos aos objetos que, aglomerados na área de representação de uma classe, compõe um novo objeto). No paradigma de mensagens, no entanto, tipicamente o destinatário de uma mensagem é pré-definido no comando de envio da mensagem, impossibilitando a posterior composição de novos complexos de objetos pela falta de generalidade do mecanismo de destinação de mensagens. Este aspecto é abordado no item 3 deste artigo segundo uma solução baseada na teoria de Cálculo de Processos.

2) Eventos : o sincronismo entre o código transmissor da mensagem e o código receptor é tipicamente muito forte, acontecendo mesmo em linguagens em que objetos são unidades de paralelismo. A simplicidade desta sincronização proporciona uma sintaxe simples e é respaldada por técnicas de especificação formais como o Cálculo de Processos. Porém, a necessidade de um grau maior de paralelismo e da melhor modelagem relativa a softwares que exigem o conceito de eventos parece ser suficiente para que eventos não sejam negligenciados nas novas LOO's.

3) Bidirecionabilidade na troca de mensagens : objetos que fazem parte na composição de um outro objeto podem, tipicamente, receber mensagens deste último e, em resposta, produzem resultados, de modo análogo à chamada convencional de procedimentos. A composição serve assim como mecanismo de hierarquização de objetos segundo um modelo senhor/escravo, onde os objetos abaixo na hierarquia respondem mas não evocam

mensagens em seus superiores imediatos. No entanto, a composição de objetos que trabalham em paralelo pode exigir a comunicação de eventos iniciada pelos objetos inferiores na hierarquia. Um objeto janela de um Sistema de Interface Gráfica, por exemplo, pode conter (por composição) vários botões que, ao serem pressionados, podem indicar, via eventos, a ação a ser tomada pelo objeto janela, sem que haja necessidade de uma política especial de "polling" ou bloqueio específico à espera de eventos como estes.

Dentro do espectro destas críticas, será apresentado no item que se segue o modelo linguístico proposto por Milner em seu Cálculo de Processos. Este modelo, já utilizado em algumas linguagens de programação [3, 5], propõe soluções para o aspecto 1) acima citado, mas que, como será visto, não satisfaz os aspectos 2) e 3).

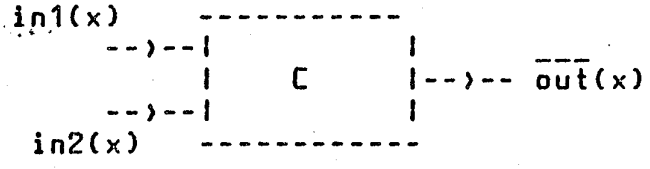
3 - CÁLCULO DE PROCESSOS - UM BOM PARADIGMA

Está fora do escopo deste trabalho explicar o Cálculo de Processos proposto por Milner. Porém alguns dos aspectos deste cálculo devem ser citados para a melhor compreensão do caminho percorrido na inclusão de mecanismos que solucionem a inclusão de paralelismo nas linguagens orientadas a objetos.

O Cálculo de Processos permite caracterizar um conjunto de entidades que operam em paralelo, denominadas AGENTES, capazes de trocar mensagens entre si de modo sincronizado. Este modelo de

concorrência e comunicação, comumente identificado com os mecanismos encontrados na linguagem CSP [5] e com o trabalho de C.A.R.Hoare [7], permite a especificação formal do comportamento de um sistema, a partir da especificação dos agentes que o compõe, via o uso de combinadores.

Dentre os combinadores, existem os combinadores dinâmicos que permitem descrever a transição dos estados dos agentes à partir da recepção e envio de mensagens destes. Neste caso, os combinadores dinâmicos mais relevantes são o de Soma e o de Prefixação. Por exemplo, um buffer com duas entradas pode ser descrito da forma :



$$C = in1(x).C'(x) + in2(x).C'(x)$$

$$C'(x) = \overline{out}(x).C$$

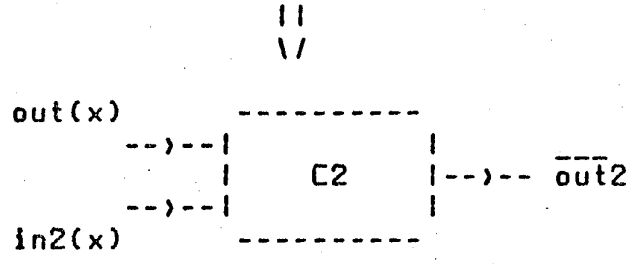
indicando que o agente C , ao receber uma mensagem (indicado pela prefixação não sublinhada) de nome in1 ou in2, vai ao estado C' até que, neste estado, envie a mensagem (indicado pela prefixação sublinhada) $\overline{out}$ retornando ao estado inicial. Nesta expressão, é o combinador Soma ('+') o indicador da opção na recepção das mensagens in1 ou in2.

O envio da mensagem $\overline{out}$, no exemplo acima, poderá ser feito à partir do momento em que outro agente conectado a C possa recebe-la, caracterizando assim o sincronismo da comunicação. Para especificar esta conexão entre agentes existem os combinadores estáticos: Composição, Restrição e Renomeação. O

buffer C do exemplo acima pode ser conectado a um outro buffer identico, formando um complexo constituido de dois buffers, via os seguintes passos:

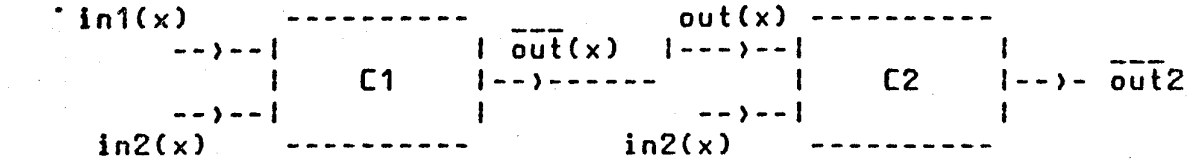
Definir C1 ::= C;
Definir C2 como uma renomeação de C :

$$C2 ::= C\ [\ in1/out,\ \overline{out}/\ \overline{out2}\ ]$$



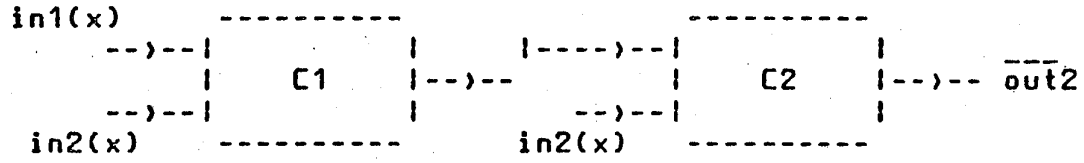
e unir C1 e C2 definindo um complexo D via a composição:

$$D ::= (\ C1 \mid C2\ )$$



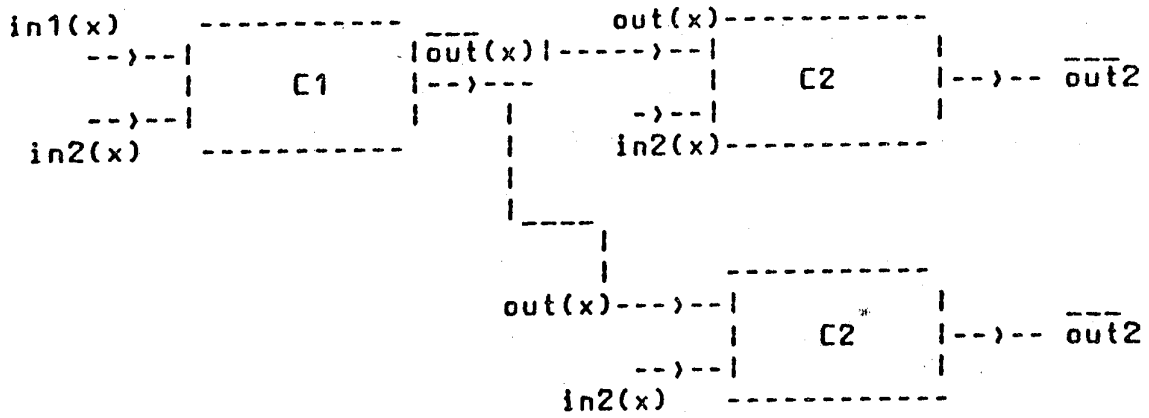
O combinador Restrição permite esconder os nomes internos para que não sejam usados em novas composições da forma:

$$D ::= (\ C1 \mid C2\ )\backslash\{out\}$$



De outro modo, out e $\overline{out}$ estariam disponiveis para nova composição como abaixo:

$$D2 ::= (\ C1 \mid C2 \mid C2\ ) ::= (\ C1 \mid C2\ ) \mid C2$$
$$::= D1 \mid C2$$



A partir destes combinadores, complexos de agentes podem ser construidos indicando os benefícios relativos a modularidade e reusabilidade dos agentes conseguido através do adiamento da ligação das assinaturas envolvidas numa composição.

Este esquema não é implementado nas LOO's correntes apesar dos benefícios supracitados. Para possibilitar a realização desta combinação estática de objetos em uma LOO (via os combinadores propostos por Milner) é preciso relaxar a definição do objeto destinatário de uma mensagem permitindo o envio de mensagens para um conjunto de objetos passíveis de recepção. Isto pode ser conseguido via o uso de filas de mensagens associadas à cada assinatura de mensagem e a especificação de um destinatário genérico ANY. Este destinatário genérico permitirá uma ligação posterior com a respectiva fila partilhada por vários objetos de acordo com as definições do usuário via combinadores estáticos . Feito isto, o objeto (agente - nos termos de Milner) tem informações suficientes para colocar sua mensagem na respectiva fila (definida para cada assinatura) e aguardar para que um outro objeto, que tenha também acesso a esta fila, receba a mensagem, realizando o

denominado "rendez vous".

Nos itens que se seguem, serão apresentadas as consequências do relaxamento do sincronismo deste modelo e a possibilidade da introdução de um caminho especifico de troca de mensagens entre um objeto e seus componentes paralelos.

4 - A NOÇAO DE EVENTO

Eventos são, de modo análogo a datagramas confiáveis em redes de computadores, mensagens assincronas - isto é: mensagens que não exigem bloqueio tanto por parte do transmissor quanto do receptor, possibilitando, por isto mesmo, um melhor uso do paralelismo. Evento é também uma abstração necessária e corrente nas várias áreas da computação onde há interação frequente e não sequencial entre vários objetos. O exemplo mais corrente seria o de softwares para Sistemas de Interfaces Gráficas que permitem que o usuário opere com vários objetos em paralelo, organizados muitas vezes de forma hierárquica, provocando ações segundo um leque muito grande de possibilidades.

O acréscimo de eventos aos mecanismos de mensagem é, além de conveniente, fácil de ser realizado via a extensão do modelo discutido no item anterior. Para isto, sintaticamente é necessária uma diferenciação entre mensagens sincronas e mensagens assincronas, possibilitando a geração diferenciada de código de envio de mensagens. O mesmo esquema apresentado no item anterior, de uma fila para cada assinatura partilhada, pode

ser usado. A diferença básica seria, no caso de mensagens sincronas, a exigência de uma resposta (REPLY) por parte do objeto que recebe a mensagem e a politica de espera (bloqueio) pelo rendez vous. Eventos, em oposição, não exigem respostas nem bloqueios.

5 - HIERARQUIA DE OBJETOS PARALELOS

Sistemas de Interface Gráficas costumam definir uma hierarquia de objetos que se comunicam em paralelo via eventos. Uma janela, tipicamente, é composta de menus, scrollbars, botões, subjanelas de edição, etc. Um menu, por sua vez, é composto de um conjunto de subjanelas para cada item, denominado item de menu, e assim por diante. Cabe ao código de cada objeto coordenar os eventos gerados pelos seus subobjetos requisitando destes novas ações, como mudança de cor, estado, tamanho, etc.

É usual que a ligação destes objetos componentes ("filhos") com o objeto hierarquicamente superior a eles seja feita explicitamente pelo programador através da passagem da referência do objeto "pai" como argumento no procedimento de criação de cada instância "filho". Porém, um acoplamento mais seguro e abstrato pode ser realizado via a "herança" de um nome destinatário genérico, PARENT, para uso no envio de eventos (ou mesmo mensagens) por objetos que se destinam a ser "filhos" de outros via composição. Este adiamento é similar ao que ocorre com o destinatário genérico ANY apresentado no item 3 e contribui, como este, para a modularidade e generalidade do esquema de

comunicação. Assim, abre-se um caminho "natural" de comunicação entre objetos "pais" e seus respectivos "filhos" possibilitando o projeto de objetos (como botões, caldeiras, etc) que tratam localmente de seus eventos e que informam para seus superiores (janelas, fábricas, etc) apenas os eventos que a estes interessam. Isto segue os preceitos de modularidade, abstração e reutilização de software.

6 - RESUMO DAS PROPOSTAS

Um conjunto de propostas para incorporação de mecanismos de comunicação e paralelismo foi sugerido nos itens acima. Deste conjunto podemos resumir:

a) Associar classes ao mecanismo de paralelismo. Objetos, por exemplo, podem ser definidos em classes onde um atributo especial indica seu comportamento análogamente ao que se faz com processos em sistemas de multiprogramação [2].

b) Incorporar os comandos de envio de mensagens síncronas e assíncronas (eventos) com a possibilidade de dois destinatários genéricos (além do específico usual): ANY para composição via combinadores estáticos e PARENT para composição via encapsulamento.

c) Possibilitar a criação de novas classes (ou complexos de classes) via o uso dos combinadores estáticos propostos por Milner em seu Cálculo de Processos.

d) Permitir a composição hierárquica de objetos que trabalham em paralelo, definidos via os mecanismos de composição e encapsulamento correntes na definição de classes convencionais.

Além destes, podemos indicar um conjunto de mecanismos extras que podem facilitar a arquitetura de programas, como por exemplo: 1) a necessidade de diferenciar os métodos a serem tratados assincronamente dos métodos síncronos; 2) a definição de filtros de mensagens permitindo o bloqueio à espera exclusiva de um evento e/ou mensagem, ou um conjunto destes; 3) a possibilidade de definir métodos atrelados ao nome dos possíveis objetos transmissores de mensagem, afim de evitar código de testes para realizar tratamentos segundo o transmissor (neste caso, o nome genérico ANY também pode ter validade para tratamentos identicos para um conjunto as mensagens enviadas por instâncias de objetos diferentes).

Outra questão relevante se refere ao mecanismo de herança relacionado com os mecanismos aqui citados. Consideramos o conceito de herança como ortogonal ao conceito de paralelismo, em oposição ao sugerido em [1]. As regras relativas a herança de métodos e componentes de uma superclasse, em ambiente de monoprogramação, não interferem, ao nosso ver, com a introdução da extensão do mecanismo de mensagens para realização de multiprogramação. As críticas levantadas em [1] não parecem levar em conta que herança e composição são mecanismos com funções linguísticas diferentes e que, em particular, composição é um mecanismo que permite partilhar código sem partilhar subtipos.

O mesmo não se deve dizer do conceito de acesso e atribuição a objetos e componentes neste ambiente de paralelismo. Mensagens surgiram historicamente neste tipo de ambiente, para desacoplar áreas de dados entre processos, dispensando o uso de memória compartilhada (permitindo uso em ambientes distribuidos e evitando bloqueios desnecessários à espera do recurso memória). Permitir o acesso e atribuição à representação (componentes) de objetos trabalhando em paralelo seria ignorar as vantagens deste encapsulamento rigido, que não permite acesso a memória compartilhada.

7 - EXEMPLO DE SINTAXE E USO

As propostas apontadas no item anterior são feitas com base na nossa experiência no desenvolvimento de uma linguagem especifica para programação em Sistemas de Interface Gráfica. Afim de ilustrar aquelas propostas, apresentamos a seguir dois exemplos, em linguagem semelhante à sendo desenvolvida.

7.1 - EXEMPLO 1

O primeiro exemplo trata dos aspectos de familiaridade entre objetos de uma composição e seu respectivo "pai". Para isto, apresentamos uma classe que define um botão que indicará, a um objeto "pai", que foi pressionado para o modo on ou modo off. Neste caso, supomos uma classe paralela primitiva da biblioteca (button) que desenha a imagem na tela via o método show e avisa ao "pai" que o botão foi acionado pelo mouse ao definir, no bloco

de MESSAGES da classe, a assinatura pressed. O atributo PARALLEL indica que a classe button_on_off define objetos sujeitos a paralelismo, instanciáveis via comando NEW, e capazes de receber/enviar mensagens assincronas. O texto seria:

```
PARALLEL CLASS button_on_off

     REP
        button b1;
        BOOLEAN on = FALSE;    ! estado atual do botão
                               ! FALSE na criação
     END REP

     ! metodos sincronos/assincronos têm assinaturas definidas
     !      no bloco de MESSAGES
     MESSAGES
          EVENT on;
          EVENT off;
     END MESSAGES

     ! tratamento do evento gerado pelo button b1
     EVENT HANDLER FOR pressed FROM b1
          IF on
             THEN
               ! pressionou de on para off
               on := FALSE
               PARENT <- off   ! acorda pai indicando mudança
             ELSE
               ! pressionou de off para on
               on := TRUE
               PARENT <- on    ! acorda pai indicando mudança
          END IF
     END HANDLER

     ! método sincrono para iniciar e mostrar botão
     METHOD creator( IN INTEGER xposicao,yposicao,
                                xtamanho,ytamanho
                   )
         ! aloca e inicia botão
         b1 <- NEW;
         b1 <- show( xposicao, yposicao, xtamanho, ytamanho );
     END METHOD

     ! ... outros métodos e tratadores

END CLASS
```

7.2 - EXEMPLO 2

O segundo exemplo aborda a composição via combinadores estáticos. Para isto, duas classes são definidas: a classe dicionário e a classe pesquisador. Num esquema produtor/consumidor, um objeto da classe pesquisador faz consultas aos objetos da classe dicionário que respondem (neste caso sincronamente) com o texto referente à palavra a ser pesquisada. Nesta pseudo-linguagem teríamos:

```
! classe do dicionário
PARALLEL CLASS dicionário

     REP
          ! representação interna
          ! ...
     END REP

     ! Não existem mensagens neste servidor mas existem
     !       respostas a mensagem a ser tratada

     MESSAGES
          REPLY  answer_consult RETURNS TEXT texto_resposta
                                TO consult;
     END MESSAGES

     SYNC HANDLER FOR consult( WORD word_to_consult )FROM ANY
        ! responde consulta com o resultado da procura
        CALLER<-answer_consult( internal_search(word_to_consult)
                              );
     END HANDLER

     ! Métodos internos ...
     METHOD creator
          ! corpo do método
          !   ...
     END METHOD

     METHOD internal_search ( WORD word_to_consult )
                    RETURNS TEXT
          ! corpo do método
          !   ...
     END METHOD
     ...

END CLASS
```

```
! classe do pesquisador

PARALLEL CLASS pesquisador

     REP
          ! representação interna
          ! ...
     END REP

     ! pesquisador chama quem souber resolver consultas
     MESSAGES
          SYNC consult( WORD word_in_research );
     END MESSAGES


     ! Métodos internos ...
     METHOD creator

          ! inicialização ...
          ! ... loop de pesquisa
          WHILE ( has_word_to_research )
            ! chama dicionário sincronamente
            text := ANY(-consult( word_in_research );
            ! trata texto referente a palavra em questão
            ! ...

          END WHILE
          !   ...
     END METHOD
     ...
     .
END CLASS
```

Pesquisador e dicionário são classes que definem objetos que podem, ao trabalhar em paralelo, utilizar os mecanismos de combinação estática para implementar um maior desempenho via uma relação de muitos dicionários (servidores) para muitos pesquisadores (consumidores). Um complexo de 3 dicionários e 10 pesquisadores pode ser definido via a seguinte sintaxe:

```
COMPLEX research_dictionary ::=
              ( pesquisador[10] | dicionario[3] )\ ( consult ) ;
```

Nesta linguagem exemplo, um complexo funciona como a

definição de uma classe. Complexos de objetos podem ser declarados e instanciados via a mesma sintaxe de objetos de classe. No momento da criação, cada objeto do complexo será instânciado e terá seu metodo de criação chamado para que comecem a trabalhar em conjunto.

8- CONCLUSAO

Parafraseando Hoare em [6], " foi descrito neste artigo um número de técnicas bem conhecidas em programação e novas notações para expressá-las" - neste caso via o paradigma de orientação a objetos. Tendo em vista que este paradigma pode ajudar a reutilização de código porque, entre outros, permite um alto grau de desacoplamento entre suas entidades (objetos), não se pode pensar em introduzir novos aspectos sem tentar manter em perspectiva a continuidade destes objetivos.

A extensão aqui proposta parece cumprir com estes objetivos já que, mais do que introduzir a multiprogramação em LOO's, introduz também um conjunto de facilidades linguisticas que ajudam no desacoplamento de objetos sujeitos a paralelismo, permitindo um nivel mais flexivel de definição de coleção de objetos como complexos de objetos, hierárquicos ou não.

AGRADECIMENTOS

Este trabalho não poderia ter sido realizado sem o importante apoio da SPA - SISTEMAS, PLANEJAMENTO E ANALISE S/A e em especial dos colegas W.Vervloet, A.G.Ayres e N.Gorini, com os

quais algumas das idéias aqui apresentadas surgiram e foram discutidas.

REFERENCIAS :

[1] P.AMERICA
Issues in the Design of a Parallel Object Oriented Language
Formal Aspects of Computing - 1989 1:366-411

[2] H.E.BAL,J.G.STEINER,A.S.TANENBAUM
Programming Languages for Distributed Computing Systems
ACM Computing Surveys Vol.21 Num. 3 Sept. 1989

[3] J.V.DEN BOS, C.LAFFRA
PROCOL - A Parallel Object Language with Protocols
OOPSLA'89 Proceedings October 1-6, pags. 95-102, 1989

[4] R.GREHAN
In Any Event
BYTE - Vol. 15 Num. 5 May 1990

[5] C.A.R.HOARE
Communicating Sequential Processes
Communications of the ACM Vol.21 Num. 8 Aug. 1978

[6] C.A.R.HOARE
Recursive Data Structures
International Journal of Computer and Information Sciences
Vol.4 Num.2 1975

[7] C.A.R.HOARE
Communicating Sequential Processes
Prentice-Hall International, 1985

[8] R.MILNER
Communication and Concurrency
Prentice-Hall International, 1989