O conteúdo do presente relatório é de única responsabilidade do(s) autor(es).
(The contents of this report are the sole responsibility of the author(s).)

Controle de Concorrência no Cm

Célio N. Targa *Mauro Oliveira Filho*
Celso Gonçalves Jr *Rogério Drummond*

Relatório Técnico IC 97-03

Maio de 1997

Controle de Concorrência no Cm*

Célio Norbiato Targa
Mauro da Silva Oliveira Filho†
Celso Gonçalves Jr‡
Rogério Drummond

{targa, maurosof, celso, rog}@ahand.unicamp.br

Laboratório A-HAND
Instituto de Computação
Unicamp
Rua Roxo Moreira 1076
13083-591 Campinas, SP

(maio de 1996)

RESUMO

Apresentamos um modelo de concorrência a ser utilizado na linguagem de programação distribuída orientada a objetos Cm Distribuído, em desenvolvimento no Laboratório A-HAND.

Objetos em Cm Distribuído podem apresentar concorrência possibilitando a execução de várias *threads* que compartilham dados internos do objeto.

O mecanismo de controle de concorrência a ser utilizado baseia-se no modelo de Regiões Críticas Condicionais [Hoa78] e denomina-se Regiões Críticas Condicionais Estendidas (RCCE). O objetivo das extensões é dar maior poder à linguagem e mais flexibilidade ao programador.

ABSTRACT

Presents a concurrency model to be used in Distributed Cm, an object-oriented programming language for distributed applications. Distributed Cm is also developed by A-HAND Labs.

Distributed Cm objects can be active entities (like servers) running concurrently; also, several running threads can share an object's data.

In Distributed Cm objects can be concurrent, so many *threads* can be ran sharing objects' data.

The proposed concurrency control mechanism — Extended Conditional Critical Regions — enhances the Conditional Critical Regions model providing new expressiveness and flexibility while keeping a concise syntax.

Implementation issues are addressed; in addition, some classical concurrency problems are solved using the model.

---

* Este trabalho é parcialmente financiado pelo PROTEM/CNPq, processo número 680089/94-2.
† Bolsista FAPESP, processo número 96/0882-6.
‡ Bolsista DTI/CNPq, processo número 380371/95-2.

1. INTRODUÇÃO

Um modelo de concorrência está sendo definido para ser utilizado na linguagem de programação básica do ambiente A-HAND, a linguagem Cm Distribuído, com o objetivo de fornecer facilidades para a definição de objetos que encapsulem atividades concorrentes. As principais características de Cm Distribuído são as extensões da semântica de criação de objetos para permitir a criação de objetos remotos, e do mecanismo de ativação de métodos para permitir a interação entre objetos através da chamada de métodos remotos.

Com objetos remotos, obtém-se concorrência entre objetos. Objetos podem ser de natureza passiva, que executam apenas chamadas de métodos, ou ativa, que passam a executar uma seqüência de comandos particular depois de criados. Além disso, objetos podem apresentar concorrência internamente, possibilitando a execução de várias tarefas simultaneamente. Cada pedido de execução do método causa a ativação de uma *thread* para atender esse pedido, que compartilham os dados internos do objeto, necessitando de um mecanismo para manter a coerência do estado interno do objeto, bem como sincronizar a execução das *threads*.

O controle de concorrência a ser implementado tem por base o modelo de Regiões Críticas Condicionais [Hoa78] e denomina-se Regiões Críticas Condicionais Estendidas [Gon94]. Algumas modificações foram feitas no modelo para dar maior poder de expressão à linguagem e mais flexibilidade ao programador.

Inicialmente, dar-se-á uma visão geral sobre *programação distribuída orientada a objetos* e *controle de concorrência*. Logo após, será abordada a *concorrência em Cm Distribuído*. Em seguida, serão levantadas algumas questões a respeito da *implementação* do mecanismo para Cm Distribuído. Por fim, será exemplificado o mecanismo com a resolução de alguns *problemas de concorrência* utilizando-se de Regiões Críticas Condicionais Estendidas.

2. PROGRAMAÇÃO DISTRIBUÍDA ORIENTADA A OBJETOS

A integração de sistemas distribuídos com o paradigma de objetos tornou-se uma crescente fonte de pesquisa e projetos industriais. Um fato a favor para a união destas idéias é, sobretudo, a notável similaridade entre as construções básicas de ambos, destacando-se a analogia que pode ser feita entre objetos e processos [Mey93].

O *Paradigma de Objetos* descreve um estilo de programação baseado na noção de classes. Uma *classe* descreve um conjunto de objetos com a mesma estrutura e mesmo comportamento. Um objeto (*instância* de uma classe) é composto de dados e de operações (*métodos*) sobre estes dados. Ao receber uma mensagem, o objeto deve executar o método solicitado na mensagem, através de computações sobre seus próprios dados e, opcionalmente, requisitar execução de métodos de outros objetos. Vários aspectos caracterizam o paradigma de objetos, dentre eles: abstração, encapsulamento, modularidade e herança [Boo94].

A programação com objetos depende de uma linguagem que suporte o paradigma e, opcionalmente, de um ambiente de programação que ofereça objetos como recursos básicos do sistema. Se, além destas características, tal sistema oferecer um ambiente distribuído para construção de programas, então podemos denominá-lo *Sistema de Programação Distribuída Orientada a Objetos*.

Embora seja viável a integração de sistemas distribuídos à programação orientada a objetos, deve-se considerar os vários problemas envolvidos. Vários aspectos devem ser abordados, tais como: transparência, ativação de métodos, objetos passivos e ativos, persistência, controle de concorrência, tolerância a falhas e desempenho [Hor89].

3. CONTROLE DE CONCORRÊNCIA

O paralelismo está presente em vários tipos de problemas computacionais. Tais problemas têm solução mais natural quando representados por atividades paralelas, atuando de forma cooperativa ou competitiva, que colaboram cada uma com uma parcela da solução final. Para isso, são necessários mecanismos que controlem o acesso aos dados compartilhados e as dependências entre resultados intermediários.

3.1 Conceitos Básicos

As seguintes definições estão conforme [And83]. Um *programa seqüencial* especifica uma lista de comandos que devem ser executados de forma seqüencial. Tal programa, quando executado, é denominado *processo*. Um *programa concorrente* especifica dois ou mais programas seqüenciais que podem ser executados concorrentemente como *processos paralelos*.

Para que haja cooperação, deve haver comunicação e sincronização entre os processos que executam concorrentemente. A comunicação permite que a execução de um processo influencie na execução do outro e pode ser efetuada por mecanismos baseados em compartilhamento de memória (que é o nosso interesse nesse trabalho) ou troca de mensagens.

Para sincronizar processos que usam variáveis compartilhadas na comunicação, podemos fazer uso de regiões críticas ou condições de sincronização. Uma *região crítica* é uma seqüência de comandos que deve ser executada de maneira indivisível. A execução de regiões críticas de forma livre de interferência de outras regiões críticas é denominada *exclusão mútua*. A *condição de sincronização* é feita de forma que um processo, que está tentando executar uma operação, fique bloqueado até que uma determinada condição seja verdadeira.

Existem vários mecanismos utilizados para garantir a sincronização entre processos, tais como: *Busy Waiting*, Semáforos, Monitores e Regiões Críticas Condicionais. Dentre esses, estamos interessados em Regiões Críticas Condicionais, que serve como base para o modelo a ser implementado em Cm Distribuído.

3.2 Regiões Críticas Condicionais

Para o uso de regiões críticas condicionais [Hoa72], as variáveis compartilhadas são associadas na declaração, a um *recurso*, de forma que só poderão ser utilizadas dentro de regiões críticas condicionais relativas a esse recurso.

Uma *região crítica condicional* consiste de um trecho de código que protege os recursos, sendo executado em exclusão mútua com os demais processos em relação a tais recursos. Além disso, uma expressão lógica estabelece uma condição a ser satisfeita para que a região possa ser executada. A execução do comando da região crítica condicional é suspensa até que a condição se torne verdadeira.

Um recurso *r* contendo variáveis $v_1, v_2, \dots, v_n$ pode ser declarado como:

`resource r :` $v_1, v_2, \dots, v_n$

A notação original de uma região crítica condicional usa o comando `with` [Hoa78], cuja sintaxe é:

```
with r when B do C
```

onde:

- *r* representa o recurso;
- *B* é a expressão lógica que controla o acesso;

- *C* é a região crítica propriamente dita.

A expressão lógica é formada apenas por variáveis vinculadas ao recurso *r*, e o comando *C* só pode alterar as variáveis da região. A avaliação de *B* e a execução de *C* não são interrompidas por comandos de outras regiões críticas condicionais, que usam o mesmo recurso *r*. Assim, quando a região crítica *C* é executada, a condição de sincronização *B* é garantidamente verdadeira.

As regiões críticas condicionais combinam, em um único mecanismo, os dois aspectos de sincronização entre processos: a exclusão mútua (execução exclusiva da região crítica) e a condição de sincronização (avaliação da condição lógica).

4. CONCORRÊNCIA EM CM DISTRIBUÍDO

A programação em Cm Distribuído está baseada no conceito de *objetos remotos* [Gon94], de forma a trazer para o escopo da linguagem Cm o conceito de transparência de localização e acesso. Nessa abordagem, os objetos utilizam recursos e interagem com outros objetos sem precisar saber que tais recursos e objetos estão espalhados pela rede. Pretende-se dessa forma produzir programas distribuídos sem muitas diferenças sintáticas e semânticas da programação sequencial.

Um objeto é dito *remoto* se ele é criado fora do contexto de execução do objeto que o criou. É uma unidade autônoma de execução que interage com outros objetos através de chamadas remotas de métodos. Isto tem duas conseqüências importantes: um objeto remoto pode ser executado em paralelo com o objeto que o criou e pode ser compartilhado por outros objetos. Dessa forma, objetos remotos podem ser executados de forma concorrente.

Um objeto remoto pode estar localizado em qualquer ponto da rede, de forma que um objeto que ofereça serviços pode estar disponível para qualquer outro objeto que queira utilizá-lo. Assim, num determinado momento, vários objetos clientes podem estar fazendo requisições a um mesmo objeto servidor, levantando a possibilidade de concorrência dentro dos objetos.

A chamada de métodos de objetos remotos é feita de forma síncrona, bloqueando o objeto "chamador" até que venha uma resposta do objeto remoto ou seja detectado um *time-out*. Entretanto, o uso de chamadas síncronas não implica necessariamente em perda de concorrência, pois dentro de um objeto podem existir múltiplos fluxos de controle (*threads*), provendo concorrência internamente ao objeto. Sendo assim, quando uma chamada a um método de um objeto remoto é feita por um objeto cliente, apenas a *thread* responsável pela chamada é bloqueada. No objeto servidor cada requisição será atendida por uma *thread* exclusiva (Figura 1).

Dessa forma, podemos distinguir em Cm dois tipos de objetos remotos: *objetos mono-threaded*, que possuem apenas um fluxo de execução e cujas requisições para execução dos métodos são serializadas; e objetos *multi-threaded*, que permite a execução paralela de seus métodos.

Em Cm Distribuído um objeto *multi-threaded* tem estrutura dinâmica, ou seja, o número de *threads* que executam dentro do objeto pode variar. O escalonamento das *threads* é feito pelo sistema e o acesso sincronizado às variáveis compartilhadas é feito pelo programador.

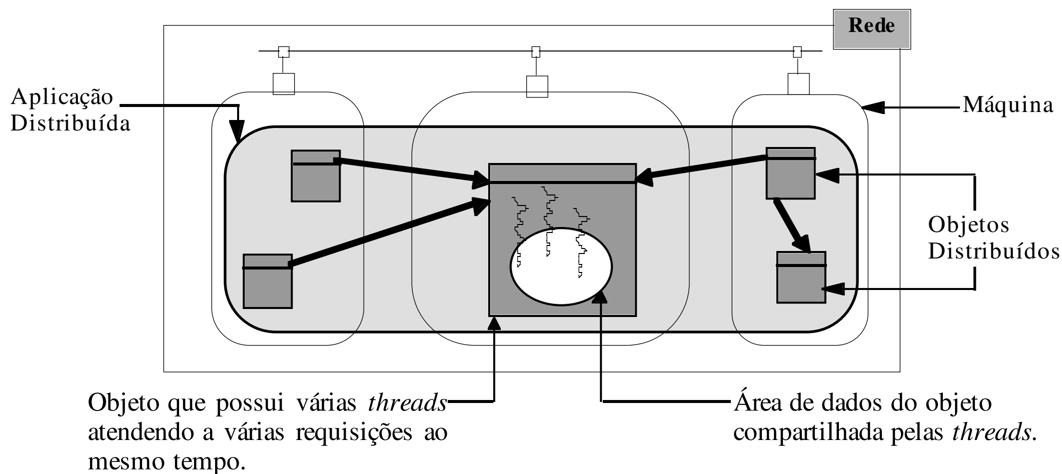

Figura 1: Exemplo de aplicação distribuída.

A princípio, uma *thread* é criada dinamicamente pela invocação de um método do objeto, e destruída ao fim da execução do método. Planejamos incorporar um mecanismo de reflexão para que, antes de uma *thread* ser criada e ligada a um método, o objeto possa controlar quando e para quais mensagens deve-se criar as *threads*.

Como em Cm todas as chamadas de funções são síncronas, bloqueando o chamador até a chegada do resultado, existe um mecanismo que pode ser utilizado por métodos que não devolvem resultados (métodos do tipo *void* e que não têm parâmetros por referência), de forma a não bloquear o método do objeto chamador. Tal mecanismo usa a palavra reservada *thread* na declaração do método e é muito útil na implementação de objetos ativos.

Para garantir a consistência do estado interno de um objeto, tratando problemas de exclusão mútua e sincronização entre *threads*, será incluído na linguagem um mecanismo baseado em regiões críticas condicionais, denominado *regiões críticas condicionais estendidas* [Gon94].

4.1 Regiões Críticas Condicionais Estendidas

O mecanismo de controle de concorrência interna a objetos, em Cm, é baseado no mecanismo de regiões críticas condicionais [Hoa72], sendo mais genérico e menos restritivo que a proposta original. As mudanças buscam um equilíbrio entre flexibilidade e segurança das aplicações. Este mecanismo é denominado Regiões Críticas Condicionais Estendidas (RCCE) [Gon94].

Nesta nova proposta, os dados compartilhados pelas *threads* são protegidos de acessos indevidos, quando utilizados dentro de regiões críticas guardadas por *variáveis de região*, que não são explicitamente associadas a estes dados. Assim, as *variáveis de região* são utilizadas para limitar o número máximo de *threads* que podem estar executando concorrentemente dentro de uma região crítica.

Uma *variável de região* é definida com o tipo *region*. Seu valor inicial é chamado de *cardinalidade* da exclusão mútua associada a essa variável, ou seja, o número máximo de *threads* permitidas dentro de regiões críticas protegidas por essa variável. Para que um objeto permita a execução de múltiplas *threads* ao mesmo tempo, sua classe deve ser declarada com o termo *threaded*.

A notação de uma regiões crítica condicional estendida é a seguinte:

```
with ( <expr-reg> [ ; <cond-sinc> ] ) <comando>
```

onde:

- *<expr-reg>* é uma expressão de região;
- *<cond-sinc>* é uma condição de sincronização;
- *<comando>* é a região crítica propriamente dita, cuja execução depende da avaliação das expressões <expr-reg> e <cond-sinc>.

Definimos a *guarda* da região como o par *<expr-reg>* e *<cond-sinc>*. A expressão de região envolve variáveis de região, constantes inteiras e operadores (&&, =) e sua avaliação é feita de forma atômica, garantindo o acesso a todas variáveis de uma vez, ou não aterando nenhuma delas. O uso dos operadores "&&" e "=" se dá da seguinte forma:

- && combina, em uma expressão de região, mais de uma variável de região. É útil para evitar a construção de comandos `with` aninhados ou quando o acesso aos dados compartilhados não puderem ser desmembrados, já que outras threads poderiam causar interferências;
- = atribui temporariamente uma nova cardinalidade a uma variável de região, durante a execução da região crítica.

A *condição de sincronização* é usada para testar se a região crítica pode ser executada pela *thread* avaliando a guarda naquele momento. Se o resultado da expressão é *true*, então é permitida àquela *thread* a execução do comando; se o resultado é *false*, a *thread* é bloqueada à espera de uma nova oportunidade para testar a guarda da região e eventualmente executá-la. As condições de sincronização envolvem variáveis que não estão necessariamente protegidas por regiões críticas, podendo mudar de valor tanto dentro como fora das regiões.

Variáveis de região podem ser declaradas da seguinte maneira:

```
region r4 = 4;        // até quatro threads por vez
region r0 = 0;        // número ilimitado de threads
region r1;            // default - uma thread por vez
region* apr;          // apontador para variável de região
region[N] vetr;       // vetor de variáveis de região
```

e como exemplo de regiões críticas condicionais estendidas, podemos ter:

```
with (r1) ...
```

executa a região crítica quando não houver nenhuma *thread* em regiões guardadas por r_1

```
with (r4 ; i == j) ...
```

executa a região crítica quando houver menos de quatro *threads* em regiões críticas guardadas por r_4 e i for igual a j

```
with (r1 && r4 = 3 ; i != j) ...
```

executa a região crítica se não houver nenhuma *thread* em regiões guardadas por r_1 e se houver menos de três *threads* em regiões guardadas por r_4; além disso, i deve ser diferente de j

```
with (vetr[k]) ...
```

executa a região crítica quando não houver nenhuma *thread* executando em regiões críticas guardadas pelo k-ésimo componente do vetor

```
        with (*apr = 1) ...
```

acesso à região crítica é garantido por *apr, com cardinalidade igual a 1.

Como um exemplo de classe de que permite a execução concorrente de seus métodos, temos a seguinte implementação de uma pilha:

```
threaded class Stack <type T; int SIZE>

T[SIZE] st;
int top = 0;
region r = 0;        // permite a execução de qualquer número de threads

export void push(T x)
{
        with (r = 1; top < SIZE)
                st[top++] = x;
}

export T pop()
{
        with (r = 1; top >= 0)
                return (st[--top]);
}

export int is_full()
{
        with (r)
                return (top == (SIZE - 1));
}
```

Um ponto forte da proposta de regiões críticas condicionais estendidas é a sua flexibilidade, que provém principalmente do fato de não haver vinculação explícita entre as variáveis de região e os dados que devem ser protegidos por elas. Tal relação deve ser implicitamente estabelecida pelo programador, alterando os dados de forma consistente dentro das regiões críticas protegidas pelas variáveis de região. Dessa forma, é possível declarar uma variável de região como componente de um vetor ou como um apontador.

Além da cardinalidade definida na declaração de uma variável de região, é preciso levar em consideração a cardinalidade corrente da variável, que surge devido a possibilidade de redefinição. Dessa forma, a cardinalidade corrente será a menor das redefinições feitas pelas RCCEs dentro de um certo contexto. Vale dizer que o valor temporário da cardinalidade só tem efeito se representar restrição de concorrência, ou seja, o novo valor deve ser menor que o atribuído na declaração.

No mecanismo de controle de concorrência há trabalho na especificação da semântica do mecanismo proposto, o que inclui o estudo de axiomas para prova formal de algoritmos que usam as regiões críticas condicionais estendidas.

5. IMPLEMENTAÇÃO

A nossa proposta parte da definição de regiões críticas condicionais estendidas e o objetivo é implementá-la como parte do ambiente de suporte a execução da linguagem, e integrá-la às demais partes desenvolvidas para a linguagem Cm Distribuído. Em tal implementação deverão ser

resolvidos alguns problemas que surgem ao tornar o mecanismo de controle de concorrência mais flexível.

A definição das estruturas de dados a serem utilizadas pelo mecanismo de concorrência é um aspecto muito importante do projeto. Tais estruturas devem levar em conta, em primeiro lugar, a execução eficiente dos procedimentos de teste das guardas e escalonamento de *threads* para execução.

Como tópicos importantes a serem tratados na integração de regiões críticas condicionais com Cm Distribuído, podemos destacar:

- tratamento de eventuais efeitos colaterais localizados na condição de sincronização;
- análise estática das regiões críticas condicionais feita pelo compilador de forma a otimizar o trabalho a ser feito pelo sistema de suporte à execução da linguagem Cm Distribuído (*Run Time System*), e oferecendo informações úteis para os procedimentos de escalonamento de *threads* e testes das guardas de CCRs;
- integração do mecanismo de exceções com regiões críticas condicionais estendidas, já que exceções podem ocorrer na execução das regiões críticas, ou até mesmo nas avaliações das guardas; da mesma forma, uma *thread* bloqueada por mais tempo que um valor pré-determinado pode gerar uma exceção;
- terminação anormal de um região crítica, isto é, casos em que o fluxo de execução é desviado para fora da região crítica (*break*, *continue*, *return*, etc.);
- a garantia de que as *threads* concorram em igualdade de condições, respeitadas as prioridades de cada uma;
- interferências indevidas que podem ocorrer durante a execução de regiões que utilizam dados globais em comum mas têm acesso garantido por variáveis de região diferentes;
- não interferência do mecanismo de controle de concorrência com o mecanismo de herança;
- definição e integração de um mecanismo de *time-out* com o mecanismo de controle de concorrência e tratamento de exceções.

Com relação às estratégias de teste das guardas, as alternativas mais naturais são:

- quando uma *thread* atinge uma CCR;
- quando uma *thread* deixa uma CCR.

Quando uma *thread* deixa uma CCR as *threads* que estão bloqueadas em variáveis de região da guarda dessa CCR são candidatas naturais ao teste da guarda, porém a ocupação da vaga não é certo, já que a *thread* pode estar bloqueada em outras variáveis de região. Na escolha das *threads* para a execução, o tempo de espera das *threads* bloqueadas deve ser levado em consideração.

Se nenhuma *thread* puder executar, deve-se analisar as *threads* que estão bloqueadas em outras variáveis de região, já que o bloqueio de um *thread* pode se dá exclusivamente devido à condição de sincronização, que podem mudar de estado em qualquer ponto do programa.

Para cada variável de região será definida uma lista de *threads* executando regiões críticas guardadas pela variável, e uma lista de *threads* bloqueadas. As listas estão ordenadas por tempo de espera, o que implicaria numa política FIFO no caso de guardas sem condição de sincronização.

Com o objetivo de otimizar o trabalho feito pelo sistema de suporte, com relação às políticas de escalonamento e detecção de problemas de concorrência, estamos estudando uma forma de organizar as guardas das regiões em um grafo que represente as relações de dependência entre as variáveis de região.

Um conceito importante que estamos estudando para acrescentá-lo ao projeto é Reflexão Computacional [Mae87], que nos ajudará a dar uma solução mais elegante para alguns problemas.

O mecanismo será implementado utilizando o pacote de *threads* fornecido pelo sistema operacional Solaris, que fornece suporte para criação, gerenciamento e destruição de *threads*. Atualmente estamos trabalhando na definição das estruturas de dados utilizadas internamente pelo *Run Time System* e na interface que este deverá fornecer ao compilador de Cm, no que diz respeito a controle de concorrência. Já foram definidos, em alto nível, os procedimentos que devem ser executados quando uma *thread* tenta entrar numa região crítica e quando sai desta.

6. PROBLEMAS DE CONCORRÊNCIA

Na literatura sobre concorrência existem vários problemas interessantes que têm sido amplamente discutidos e analisados. Alguns desses problemas serão mostrados nesta seção como exemplos.

6.1 Leitores e Escritores

O problema dos leitores e escritores modela o acesso a uma base de dados. Em uma grande base de dados, como um sistema de reserva de passagens aéreas, é necessário ter-se vários processos tentando ler ou escrever ao mesmo tempo. É possível ter vários processos (leitores) lendo a base de dados ao mesmo tempo, mas somente um único processo (escritor) pode escrever na base de dados em um determinado momento; mesmo leitores não podem acessar nesse momento. A questão é como programar leitores e escritores.

Na solução do problema utilizando Regiões críticas condicionais estendidas temos o seguinte:

```
region r = 0;
/* região r de cardinalidade infinita será implicitamente vinculada
   à base de dados
*/

void Writer()
{
   while(TRUE) {               // laço infinito
      Think_up_Data();         // processa o dado a ser escrito
      with(r=1)                // acesso exclusivo à base de dados
         Write_Data_Base();    // acessa a base (região crítica)
   }
}

void Reader()
{
   while(TRUE) {
      with(r)                  // acesso para leitura
         Read_Data_Base();     // acessa dados (região crítica)
      Use_Data_Read();         // processa o dado lido
   }
}
```

No exemplo fica claro o uso da cardinalidade definida na declaração da variável de região, de forma a permitir vários leitores acessarem a base de dados num mesmo instante. A implementação do procedimento *Reader()* fica bem simplificada e, se houver necessidade de se limitar o número de leitores, basta definir a cardinalidade da variável de região como o número máximo de leitores desejados.

No procedimento *Writer()* utilizamos o recurso de redefinição temporária da cardinalidade, fornecida pelo mecanismo de regiões críticas condicionais estendidas, para restringir o número de escritores que podem ter acesso à base de dados.

Na solução, não é preciso ter controle explícito sobre o número de leitores que acessam a região crítica, como é o caso da solução utilizando semáforos apresentada em [Tan92], onde é preciso gerenciar dois semáforos.

Do jeito que a solução foi apresentada um leitor tem preferência sobre o escritor, já que um leitor poderá executar a região crítica sempre que um existirem outros leitores executando, e um escritor ficará bloqueado até que não tenha nenhum leitor ou outro escritor na região.

Para que um escritor tenha sempre precedência sobre os leitores, basta fazer uso da condição de sincronização, como segue:

```
region r = 0;
region r_write;
int want_write;

void writer()
{
    while(TRUE) {              // laço infinito
        Think_up_Data();       // processa o dado a ser escrito
        with (r_write; want_write == 0)
            want_write = 1;
        with(r=1) {            // acesso exclusivo à base de dados
            Write_Data_Base(); // acessa a base (região crítica)
            want_write = 0;
        }
    }
}

void reader()
{
    while(TRUE) {
        with(r; want_write == 0)   // acesso para leitura
            Read_Data_Base();  // acessa dados (região crítica)
        Use_Data_Read();       // processa o dado lido
    }
}
```

Como se pode ver, a variável *want_write*, que participa da condição de sincronização relativa à variável de região *r*, mudou de valor fora de uma região crítica guardada por *r*. Tal facilidade não é possível na proposta original.

Na solução apresentada, embora a "prioridade" seja dos escritores, os leitores não ficarão em *starvation*, pois quando um escritor sai da região crítica *with(r=1)...*, as *threads*, tanto de leitores como de escritores, estarão competindo em igualdade de condições.

6.2 Barbeiro Dorminhoco

Uma barbearia tem um barbeiro, uma cadeira de barbeiro e *n* cadeiras de espera para clientes. Caso não exista nenhum cliente, o barbeiro senta na sua cadeira e dorme; quando um cliente chega, ele acorda o barbeiro. Se outro cliente chegar enquanto o barbeiro estiver trabalhando, ele senta-se em uma cadeira de espera, se alguma estiver vaga, ou deixa a barbearia se não houver cadeiras vagas. O problema é programar o barbeiro e os clientes sem que eles entrem em conflitos.

A solução com Regiões Críticas Condicionais Estendidas é a seguinte:

```
const int CHAIRS=5;            // número de cadeiras de espera

region mx;                     // exclusão mútua
region barbers = N_BARBERS;    // acesso ao barbeiro
int waiting=0;                 // clientes que estão sentados nas
                                    // cadeiras de espera

void Barber()
{
   while(TRUE) {
      with(mx; waiting >0 )    // dorme se número de clientes é 0
         waiting--;            // decrementa o contador de espera
      Cut_Hair();              // corta cabelo fora da região crítica
   }
}

void Customer()
{
   with(mx) {
      if (waiting > CHAIRS)    // não existe cadeiras de espera vagas
         return;               // barbearia cheia; vai embora
      else                     // existe cadeiras vagas
         waiting++;            // aumenta o nº de clientes à espera
   }
   with(barbers) {             // espera um barbeiro livre
      Get_Haircut();           // vai cortar o cabelo
   }
}
```

No procedimento *Customer()* usamos a variável de região *mx*, com o objetivo único de garantir a exclusão mútua na variável *waiting*. O teste *waiting* > *CHAIRS* não pode ser feito como condição de sincronização, porque de acordo com a semântica de regiões críticas, o cliente ficaria bloqueado, esperando uma cadeira desocupar, o que não é compatível com a definição do problema.

Na solução pode-se observar que, caso o barbeiro esteja dormindo e chegue algum cliente, não há necessidade de acordar o barbeiro explicitamente, como em uma solução usando semáforos [Tan92].

6.3 Jantar dos Filósofos

Dijkstra propôs e resolveu um problema de sincronização chamado de Problema do Jantar dos Filósofos. Desde então, a cada nova primitiva de sincronização inventada, tem-se tentado

demonstrar o poder da nova primitiva apresentando uma solução elegante para o Jantar dos Filósofos.

Cinco filósofos estão sentados em volta de uma mesa na qual cada filósofo tem um prato de espaguete e entre os pratos existe um garfo. Para comer o espaguete, o filósofo necessita de dois garfos. A vida de um filósofo consiste em comer e pensar de forma que, quando um filósofo fica com fome, ele tenta pegar o garfo da esquerda e o da direita, um por vez, em qualquer ordem. Se conseguir pegar os dois garfos, ele come por algum tempo, coloca os garfos de volta e volta a pensar. O problema é programar cada filósofo de tal forma que ele nunca fique sempre tentando comer sem conseguir.

Nossa solução é a seguinte:

```
const int N=5;                // número de filósofos

region fork[N];
/* vetor de variáveis de região, cada variável esta associada a um
   garfo implicitamente
*/

void philosopher(int i)       // i indica qual filósofo 1 a N
{
   while(TRUE) {
      Think(i);               // filósofo i está pensando
      with(fork[(i+1)%N] &&   // filósofo i está com fome; ou pega
            fork[i]) {        // os dois garfos ou bloqueia
         Eat(i);              // filósofo i está comendo
      }
   }
}
```

Aqui demonstramos o uso de vetor de variáveis de região e da combinação de algumas variáveis de região em uma única expressão de região. No caso dos filósofos tal expressão nos dá uma solução bem clara, já que um filósofo só pode comer se conseguir pegar as garfos da direita e da esquerda, ou seja, se conseguir acesso às variáveis de região que guardam os respectivos garfos. Dessa forma, evitamos situações de *deadlock*, pois um filósofo nunca segura um garfo e fica tentando pegar o outro.

A solução alcança o máximo de paralelismo do problema, de forma que, com cinco filósofos, dois deles poderão estar comendo ao mesmo tempo.

Sem a extensão definida em RCCE, poderíamos usar comandos `with` aninhados, o que leva à possibilidade de *deadlock*, ou teríamos que controlar o número de garfos disponíveis para cada filósofo, como a solução proposta em [Hoa72].

Não é preciso manter explicitamente o estado em que os filósofos se encontram (faminto, comendo ou pensando), como é o caso da solução apresentada em [Tan92] utilizando semáforos. Com o mecanismo de regiões críticas condicionais estendidas é preciso se preocupar apenas em apanhar os dois garfos, não importando o estado dos vizinhos.

6.4 Ponte de Mão Única

Dada uma ponte com uma única pista numa estrada de mão dupla e duas pistas. Na ponte cabem três carros no máximo. Um funcionário público é responsável por alternar o sentido do tráfego na ponte. O problema consiste de implementar as seguintes funções que devem ser executadas para que não haja acidentes.

A solução com Regiões Críticas Condicionais Estendidas é a seguinte:

```
// carro no sentido norte->sul
ns();
sai();

// carro no sentido sul->norte
sn();
sai()

// funcionário público
void funcionario()
{
   while(TRUE) {
      dorme();
      muda_ponte();
   }
}
```

```
enum {norte, sul} dir=norte;
region ponte=3;

void ns()
{
   with(ponte; dir==sul);
}

void sn()
{
   with(ponte; dir==norte);
}

void muda_ponte()
{
   with(ponte=1) {
      if(dir==norte)
         dir=sul;
      else
         dir=norte;
   }
}
```

Neste caso, o fato da RCCE permitir várias *threads* executarem regiões críticas e permitir a redefinição temporária da cardinalidade, torna a solução mais natural e fácil de programar.

7. REFERÊNCIAS BIBLIOGRÁFICAS

[And83] Concepts and Notations for Concurrent Programming
Gregory R. Andrews e Fred B. Schneider
ACM Computing Surveys, 15(1), pp. 3-43, março 1983.

[Boo94] Object Oriented Design with Applications, 2nd edition.
Grady Booch
The Benjamin/Cummings, Redwood City (CA), 1994.

[Gon96] Desenvolvendo Aplicações Distribuídas em Cm.
Gonçalves, C., Teles, A. e Drummond, R.
Anais do XIV Simpósio Brasileiro de Redes de Computadores. UFC, Fortaleza (March 1996).

[Gon94] Objetos Distribuídos
Celso Gonçalves Jr.
Tese de mestrado, DCC-IMECC-UNICAMP, agosto 1994.

[Han78] Distributed processes, a concurrent programming concept
Per Brinch Hansen
Communicatios of the ACM, 21 (11), pp 934-941, novembro 1978.

[Hoa72] Towards a Theory of Parallel Programing
Charles A. R. Hoare
Operating Systems Techniques, Academic Press, New York (NY), 1972

[Hoa78] Communicating sequential processes
Charles A. R. Hoare
Communications of the ACM, 21 (8), pp 666-677, agosto 1978.

[Hor89] Is object orientation a good thing for distributed systems?
Chris Horn
Lecture Notes on Computer Science n 433, pp 60-74, Springer-Verlag, Berlin.

[Mae87] Concepts and Experiments in Computational Reflection
Pattie Maes
OOPSLA'87 , pp. 147-155

[Mey93] Systematic concurrent object-oriented programing
Bertrand Meyer
Communications of the ACM, 36 (9), pp 56-80, setembro 1993.

[Tan92] Modern operating systems
Andrew S. Tanenbaum
Prentice-Hall, Englewood Cliffs (NJ), 1992.

[Tel93] A linguagem de programação Cm (versão 2.0x)
Alexandre P. Teles
Tese de mestrado, DCC-IMECC-UNICAMP, outubro 1993.

[Weg87] Dimensions of Object-Based Language Design
Peter Wegner
ACM SIGPLAN Notices OOPSLA'87 Proceedings, pp. 168-182, 1987