Desenvolvendo Aplicações Distribuídas em Cm

Celso Gonçalves Jr. *Alexandre Prado Teles*
Rogério Drummond

Relatório Técnico IC 97-07

Maio de 1996

Desenvolvendo Aplicações Distribuídas em Cm•

Celso Gonçalves Jr,¨ Alexandre Teles,¨¨ Rogério Drummond

celso@dcc.unicamp.br

Laboratório A-HAND
IC – Unicamp
Rua Roxo Moreira, 1076
13083-591 Campinas, SP

29 de março de 1996

RESUMO

Cm é uma linguagem baseada em C, com extensões para permitir programação modular e orientada a objetos, oferecendo funcionalidade equivalente a C++. A versão distribuída de Cm, denominada Cm Distribuído, busca estender a linguagem com mecanismos de programação concorrente e distribuída. Basicamente, pretende-se com ela definir um modelo de programação baseado em objetos distribuídos, que podem residir em diferentes pontos de uma rede, interagindo transparentemente através de chamadas de métodos.

Aplicações distribuídas são construídas pela criação e configuração de objetos espalhados por uma rede, servindo como unidades de abstração e distribuição de recursos. Para a transferência de massas de dados para outros objetos, arquivos e periféricos, a linguagem oferece portas de dados, que podem ser tipadas com qualquer tipo complexo especificado na linguagem. Objetos podem ter referências para outros objetos que podem, como as portas de dados, ser configuradas externamente, aumentando sua flexibilidade. Servidores multitarefa podem ser implementados por classes com facilidades de *multi-threading*, onde cada chamada de método causa a criação de uma nova *thread*; o controle de concorrência é feito pelo mecanismo de regiões críticas condicionais estendidas.

ABSTRACT

The Cm programming extends C towards modularity and objects; currently its power compares well with C++. The extension Distributed Cm includes concurrent and distributed programming concepts, thereby defining a programming model based on distributed, networked objects interacting transparently via method calls.

Distributed applications are comprised of objects spreaded accross a network; these work as abstraction/resource distribution units. Data flows between objects, files and peripherals through ports supporting any complex data type. Both ports and reference to objects can be configured outside the object so enhancing its flexibility. Multithreading classes can implement multitasking servers: a method call creates a new thread with concurrency control based on extended conditional critical regions.

INTRODUÇÃO

Este artigo apresenta a especificação da linguagem Cm Distribuído, desenvolvida dentro do grupo de Linguagens de Programação e Sistemas Distribuídos do Laboratório A-HAND. Essa linguagem foi apresentada em [Gonçalves 94], basicamente como uma extensão da linguagem Cm [Drummond 88,

• Este trabalho é parcialmente financiado pelo PROTEM/CNPq, processo número 680089/94–2.

¨ Bolsista DTI/CNPq, processo número 380371/95–2.

¨¨ Bolsista DTI/CNPq, processo número 380377/95–0.

Furuti 91, Teles 93]. O artigo descreve o estado atual desse trabalho, assim como outras linhas de pesquisa em objetos distribuídos no ambiente A-HAND.

O artigo está estruturado da seguinte forma: a combinação da programação orientada a objetos com sistemas distribuídos é explicada com base nas vantagens dessa abordagem e dos problemas que ela põe. Outros trabalhos nessa linha, documentados na literatura, são apresentados. A seguir é explicada a implementação, no ambiente A-HAND, de um modelo de objetos distribuídos, tanto com base em linguagens como no suporte do sistema operacional e *middleware.* As próximas duas seções descrevem as extensões à linguagem Cm para programação distribuída e concorrente; um exemplo de aplicação distribuída complexa é mostrado a seguir. Finalmente, discutimos questões de implementação e apresentamos as conclusões do artigo.

PROGRAMAÇÃO DISTRIBUÍDA ORIENTADA A OBJETOS

A combinação dos conceitos de objetos e sistemas distribuídos é considerada uma abordagem promissora para a construção de sistemas complexos, por tentar complementar as vantagens destes dois conceitos. Há entretanto vários aspectos problemáticos nessa integração, como assinalam [Meyer 93, Gonçalves 94].

O paradigma de objetos

A programação dentro do paradigma de objetos consiste na implementação de programas como sendo conjuntos de classes. As classes são usadas para gerar objetos com estrutura e comportamento comuns; esses objetos irão interagir para produzir os resultados do programa. Classes podem ser agrupadas em estruturas baseadas em herança, que é a definição de novas classes a partir de classes pré-existentes.

A programação orientada a objetos é uma tecnologia testada nos mais diversos domínios de aplicação, e obteve um julgamento consensual quanto às suas vantagens nos processos de desenvolvimento de software. Muitas dessas vantagens derivam da aproximação efetiva que os conceitos do paradigma alcançam para expressar entidades e processos reais. Esse estilo de programação estimula a produção de componentes de software mais flexíveis, extensíveis e robustos.

As linguagens mais representativas do paradigma são Smalltalk [Goldberg 83], C++ [Stroustrup 91] e Eiffel [Meyer 88]. No ambiente A-HAND, foi implementada a linguagem Cm [Furuti 91, Teles 93], descrita em detalhe mais adiante.

Programação em Sistemas Distribuídos

Um sistema distribuído é definido como um sistema que se apresenta aos usuários como se fosse centralizado, quando na verdade ele gerencia recursos espalhados por uma rede de computadores. O conceito fundamental nesses sistemas é a transparência, no sentido de que os recursos são usados através de uma interface uniforme, independentemente de sua localização. Em comparação ao modelo centralizado, baseado em computadores de grande porte (*mainframes*), a alternativa distribuída apresenta vantagens de custo, flexibilidade e confiabilidade.

A programação em sistemas distribuídos é inerentemente complexa. Todos os benefícios relacionados devem ser alcançados com a implementação de um ambiente de programação bastante sofisticado e preciso. Para a construção de tais sistemas, a programação orientada a objetos oferece vantagens bastante desejáveis, como modularidade e extensibilidade.

Trabalhos correlatos

Um grande número de modelos (e implementações) de programação baseada em objetos distribuídos têm sido apresentados nos últimos anos. Com relação às linguagens de programação, duas abordagens distintas são nitidamente definidas: linguagens inteiramente novas, orientadas desde a sua concepção para os requisitos desse modelo de programação; e a adaptação de linguagem existentes, com a adição de mecanismos para distribuição e concorrência. A nossa linha de pesquisa segue essa segunda abordagem: a linguagem Cm foi implementada e seu compilador está operacional. A versão distribuída pretende ser

uma evolução suave para o modelo de objetos distribuídos. Apresentamos a seguir outras propostas semelhantes à nossa.

O projeto COOL (Chorus Object-Oriented Layer) [Lea 93] visa implementar um sistema programação distribuída orientada a objetos utilizando recursos do Chorus. A idéia principal desse ambiente é aproximar as abstrações oferecidas pelo *microkernel* (como atores, *threads*, portas, mensagens e memória virtual) dos mecanismos da programação orientada a objetos, como criação e destruição de objetos, chamada transparente de métodos e objetos persistentes. Esse mapeamento é feito por camadas de software, dentre elas um modelo genérico de objetos, independente de uma linguagem de programação específica; um pré-processador chamado COOL++ traduz código fonte (em uma extensão de C++) para uma notação intermediária baseada nas abstrações oferecidas pelo ambiente.

PANDA [Assenmacher 93] é um ambiente de programação distribuída baseado em C++, oferecendo espaço de endereçamento global, objetos persistentes, migração de objetos e de *threads*. A proposta visa uma alta portabilidade, portanto não impõe modificações na linguagem; um pré-compilador, entretanto, é utilizado para facilitar a codificação. A funcionalidade básica é implementada através de bibliotecas de classes, com os mecanismos de paralelismo, distribuição e persistência. A interação entre objetos remotos pode-se fazer tanto por migração de objetos como de *threads*. Um mecanismo de coleta de lixo distribuída é empregado para evitar os efeitos da perda de referências para objetos.

Separate Entities [Meyer 93] é uma proposta de extensão da linguagem Eiffel, no sentido de modificar uma linguagem orientada a objetos, na menor extensão possível, para atender aos requisitos da programação concorrente e distribuída. Isso deve ser feito preservando a semântica da linguagem e o poder de expressão de técnicas como herança, polimorfismo e classes abstratas. A extensão proposta consiste em inserir na linguagem Eiffel uma nova palavra reservada, separate. Objetos cuja classe é declarada “separada” são executados em paralelo com o objeto que os criar.

PROGRAMAÇÃO COM OBJETOS DISTRIBUÍDOS NO AMBIENTE A-HAND

O Ambiente A-HAND [Drummond 87] é o contexto onde se desenvolvem trabalhos em várias áreas da computação, com o objetivo de conceber e implementar um ambiente de desenvolvimento apto a facilitar a construção de grandes sistemas de software. O conceito que fundamenta o ambiente é o uso de *hierarquias de abstrações em níveis diferenciados*: objetos complexos são formados a partir de componentes que, por sua vez, podem ser também objetos compostos por objetos mais simples, e assim por diante.

Na área de linguagens de programação, são oferecidas duas linguagens, Cm e LegoShell. Embora sejam, por construção, bastante relacionadas, cada uma delas tem um objetivo e um estilo de programação diferentes: Cm é derivada de C, com mecanismos de programação modular e orientação a objetos, enquanto que a LegoShell é gráfica, formando programas a partir da conexão de componentes como programas, arquivos e dispositivos periféricos. Está em andamento a especificação e implementação da versão distribuída de Cm (denominada “Cm Distribuído”), conforme descrito neste artigo.

A Linguagem Cm

Cm é derivada de C com relação a comandos, operadores e expressões. A extensão fundamental a C é o tipo classe, que introduz na linguagem abstração de dados, programação modular e orientada a objetos. O compilador da versão original de Cm está descrito em [Furuti 91].

As classes são unidades de compilação em Cm. Um programa executável é gerado pela compilação de uma classe, e a execução corresponde à criação de um objeto dessa classe e a execução do seu construtor. Um exemplo simples de programa Cm está a seguir.

```
class Program<>
import Output;

constructor ()
```

```
{
Output o;

o << “Hello World!”;
}
```

As classes em Cm podem ser parametrizadas, inclusive com tipos. O recurso equivalente em C++ é o *template*; o exemplo seguinte mostra uma classe Cm que implementa pilhas genéricas com relação a tamanho e tipo dos elementos armazenados.

```
class Stack<type T; int SIZE>
T [SIZE] st; int top = 0;

export void push (T x)
{
    st [top++] = x;
}

export T pop ()
{
    return (st [--top]);
}
```

A segunda versão de Cm [Teles 93] incorporou exceções, *overloading* de métodos e operadores, vetores associativos, construtores e destrutores de classes, entre outras mudanças. Falando mais adiante da versão distribuída da linguagem, será importante falar do tratamento de exceções (que é explicado a seguir) num contexto distribuído.

O mecanismo de exceções implementado em Cm [Teles 93] é equivalente ao de C++, e baseia-se nos tipos das expressões usadas na ativação de exceções para encontrar o tratador responsável pelo atendimento daquela exceção. Os tratadores são trechos de código contidos no fim de blocos especiais chamados *comandos protegidos*; cada tratador atenderá exceções do tipo especificado na cláusula when. A classe Stack poderia ser reescrita para gerar exceção quando fosse tentada uma operação inválida na pilha:

```
export void push (T x)
{
    if (top < SIZE)
        st [top++] = x;
    else
        raise “Stack: overflow in push”;
}

// o método pop() gera uma exceção de underflow

// exemplo de uso; o símbolo de abre colchetes abre um bloco de comandos protegidos
[
    st.push (..);
    when char* s:
        // tratador de exceção do tipo char*, valor é colocado em s
]
```

A linguagem LegoShell

A LegoShell [Drummond 89, 96] é uma linguagem visual para a construção de aplicações distribuídas. Os programas que compõem as aplicações são representados por ícones, assim como arquivos, dispositivos de entrada e saída e os elementos conectores, que definirão os canais para a transmissão de dados. Um conjunto de programas, arquivos e dispositivos interligados é chamado *computação*. A seguir temos um exemplo de computação, para o problema de produtores e consumidores.

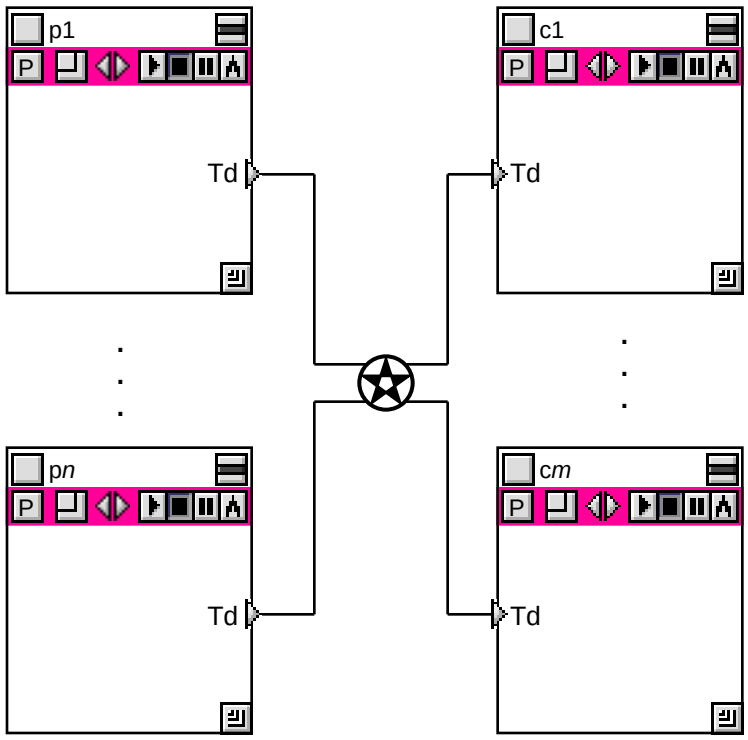


Figura 1— Exemplo de computação LegoShell

Os programas do lado esquerdo são produtores, e disponibilizam seus dados através das portas de nome Td; do lado direito estão os consumidores, que recebem os dados também por suas portas de entrada. As portas são interligadas pelo conector estrela (também chamado *broadcast*), que possui portas virtuais de entrada e saída; esse conector repassa, para suas portas de saída, todos os dados recebidos pelas portas de entrada. Como resultado, todo consumidor recebe os dados gerados por todos os produtores.

As portas de dados são canais para a transmissão de grandes massas de dados entre os componentes de uma computação; em Cm, toda operação de entrada/saída será, tipicamente, efetuada através de portas. A LegoShell oferece vários conectores, que agem como distribuidores de dados entre objetos; eles podem ter semânticas complexas com relação a políticas de distribuição da dados. Atualmente está sendo desenvolvida uma proposta de mestrado [Quadros 95] que utiliza o conceito de conectores especiais para a implementação de objetos com tolerância a falhas.

Um *conector de objetos* é um componente da interface de um objeto, correspondente a uma referência para um servidor que deve ser suprida durante a configuração do objeto. Na LegoShell, isso será feito através da ligação da referência com um objeto servidor da classe esperada; a linguagem Cm oferecerá uma operação equivalente.

Monitoramento de objetos

No ambiente A-HAND esta tarefa está a cargo da ferramenta Object Inspector [Garcia 95], cujas funções estão divididas em dois grupos: a depuração interna e a depuração externa. No primeiro grupo, estão funções típicas dos depuradores de programas seqüenciais, como colocação de *breakpoints*, execução passo a passo, visualização de variáveis, etc.

A depuração externa refere-se ao acompanhamento de toda uma aplicação distribuída, composta de um conjunto de objetos executando paralelamente, onde o foco da atenção está em aspectos como a quantidade de mensagens, tempo gasto na comunicação, carga das máquinas e assim por diante. A metáfora de programação oferecida pela LegoShell a torna apta a operar, em tempo de execução, como interface entre o programador e o Object Inspector.

Servidor de Nomes

O ambiente de utilização das linguagens Cm Distribuído e LegoShell deverá oferecer mecanismos para a criação de espaços de nomes, com funções de registro e recuperação de informações. Essas funções têm aplicação imediata na construção de aplicações distribuídas, pois contribuem para facilitar a disponibilidade e manter a transparência na utilização dos recursos de uma rede. Vários aspectos relativos a esse problemas foram abordados pelo sistema OMNI [Di Cianni 94].

A Linguagem Cm Distribuído

A linguagem Cm foi estendida através da implementação de mecanismos de programação concorrente e distribuída, dando origem ao Cm Distribuído. Os programas escritos nessa nova linguagem tipicamente contêm elementos que estão sendo executados em pontos distintos de uma rede; esses elementos são objetos, que comunicam-se transparentemente através de chamadas de métodos. Para o compartilhamento de objetos são oferecidos mecanismos de controle de concorrência, permitindo a construção de servidores multi-tarefa.

A implementação do Cm Distribuído tem duas frentes: as alterações (sintáticas e semânticas) na linguagem e a construção de uma camada responsável pelo oferecimento de abstrações de recursos de programação concorrente e distribuída (como representação de objetos distribuídos, chamada remota de métodos, primitivas de sincronização, etc). Essa camada é o Sistema de suporte à execução [Oliva 95] (também chamado *Run-time system*).

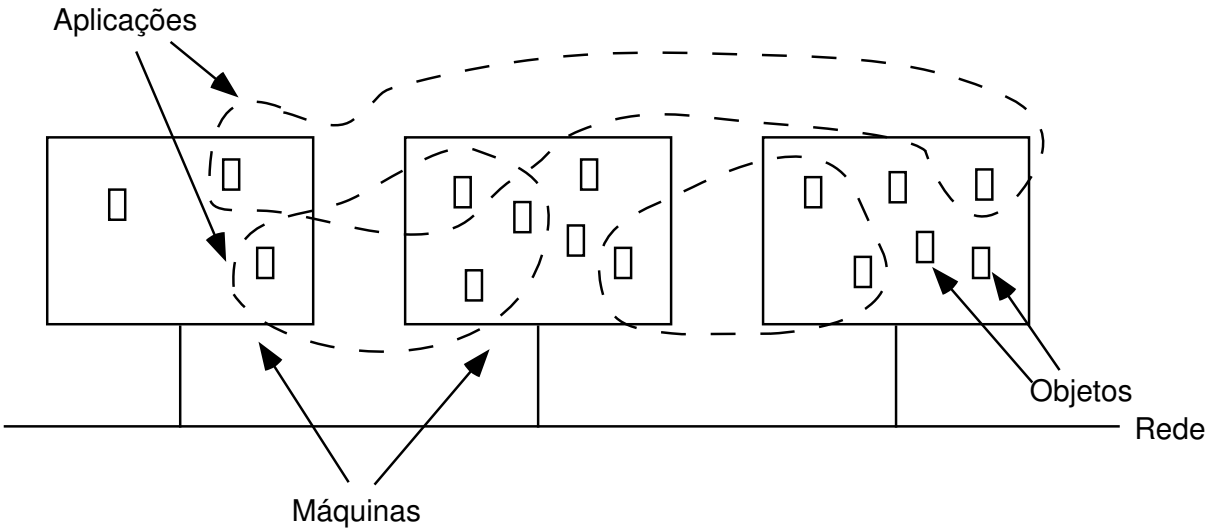


Figura 2 — Aplicações distribuídas baseadas em objetos

Uma aplicação distribuída é tipicamente um agregado de objetos cooperantes, localizados em pontos distintos da rede. Um mesmo objeto pode participar de várias aplicações, desde que sua localização seja conhecida pelos objetos clientes interessados. Uma aplicação começa pela criação de um objeto "principal", que depois passa a criar os outros objetos da aplicação, aproveitar objetos já existentes (caso típico de servidores) e configurar as referências para objetos e as conexões entre portas de dados.

A concepção de Cm Distribuído buscou estender os conceitos do paradigma de objetos de maneira natural e uniforme para o contexto distribuído. A criação e destruição de objetos é feita através de construtores e destrutores, e a comunicação entre objetos é feita com transparência de localização, cabendo ao sistema de suporte à execução transmitir via rede as chamadas de método entre objetos em diferentes espaços de endereçamento.

EXTENSÕES PARA PROGRAMAÇÃO DISTRIBUÍDA EM CM

A seguir apresentamos as definições introduzidas pelo Cm Distribuído, a maior parte delas descritas em [Gonçalves 94]; termos aproveitados de outros trabalhos da área trazem as referências respectivas.

Objetos remotos

Um programa Cm é resultado da compilação de uma classe (e das classes das quais ela eventualmente dependa) e sua ativação (por exemplo, em uma *shell* do Unix). Cada programa é representado por um código executável, resultado da compilação e ligação da classe que dá nome ao programa. Não existe a função `main()` como em C++: o programa começa e termina com a execução do construtor e do destrutor de um objeto dessa classe. Um exemplo simples escrito em Cm é dado por:

```
class UseStack<>
import Stack, Output;

Stack<int, 500> my_st;  Output o;

constructor()
{
    o << "Hello World!";
    my_st.push(10);
}
```

A classe `Stack` é a mostrada anteriormente; a classe `Output` é pré-definida na linguagem. A execução desse programa, depois da compilada a classe `UseStack`, está a seguir:

```
% UseStack
Hello World!
%
```

No exemplo mostrado, o objeto da classe `Stack` está contido dentro do objeto da classe `UseStack`; isso se reflete na sua visibilidade e no seu tempo de vida. Definimos como *contexto* o conjunto de informações relativas à execução de um objeto, como por exemplo o nome do usuário que o ativou, espaço de endereçamento e, a informação mais importante, o nó da rede onde o objeto está sendo executado.

Definimos um *objeto local* como sendo um objeto que é executado dentro do contexto do objeto que o criou. No exemplo, o objeto da classe Stack é local quando considerado em relação ao objeto da classe `UseStack`. A separação de contextos de execução dos objetos de uma aplicação é o conceito chave do Cm Distribuído.

Um objeto é *remoto* se ele é executado em um contexto diferente do objeto que o criou; este último é denominado *objeto pai*. Normalmente essa diferença será nos pontos da rede onde eles estão, mas não necessariamente: dois objetos podem ser remotos mesmo executando em uma mesma máquina. A diferença essencial é que a aplicação, da qual eles fazem parte, é distribuída lógica e fisicamente. Utilizando a mesma classe Stack do exemplo anterior:

```
class UseRemoteStack<>
import Stack, Output;


Remote<Stack<int, 500>> my_remote_st;
Output o;

constructor()
{
    o << "Hello World!";
    my_remote_st.push(7);
}
```

A execução desse programa, depois de compilada a classe `UseRemoteStack`, está a seguir:

```
% UseRemoteStack
Hello World!
%
```

A classe pré-definida `Remote<>`, ao ser parametrizada, gera instâncias que são objetos remotos, que além dos métodos definidos na sua classe poderão também executar métodos definidos na própria classe `Remote<>`, úteis para funções que ilustraremos a seguir. Na proposta original de extensão de Cm [Gonçalves 94], os objetos remotos eram criados com a palavra reservada `remote`, que atuava como construtor de tipos. Acreditamos que essa nova alternativa simplifica a extensão da linguagem enquanto mantém a semântica e o poder de expressão da proposta original.

Objetos remotos podem ser criados dinamicamente, através da operação `new`. A utilização do objeto remoto se faz da mesma forma descrita anteriormente, e passa a ser de atribuição do programador destruir o objeto, através de uma chamada de `release`. Caso o objeto remoto não seja destruído, o efeito colateral é a permanência de um objeto, em outro contexto, para o qual não há mais referência, o que poderia exigir um mecanismo de coleta de lixo; voltaremos a esse assunto ao falar de escopo e tempo de vida.

Chamadas remotas de métodos

Uma *chamada remota* é uma chamada de método de um objeto remoto. Essas chamadas não diferem de uma chamada de método de um objeto local; de fato, a única distinção entre esses objetos é feita no momento da sua declaração. Há um caso especial de chamada remota que não é válida: no caso de parâmetros passados por referência, essa informação não faria sentido no objeto remoto que executará o método, já que ele está num contexto diferente. Nesses casos, o compilador mostrará uma mensagem de advertência e a passagem dos parâmetros será por cópia.

O mecanismo de tratamento de exceções de Cm será estendido para a correspondente variante distribuída, isto é, exceções podem ser propagadas em resposta a uma chamada remota de método; basta para isso que ela não tenha sido tratada no objeto remoto, devendo então seguir para o objeto chamador. Isso é fundamental para preservar a transparência de uso de objetos remotos, já que em termos funcionais uma chamada de método terá o mesmo resultado independentemente da localização do objeto chamado.

Um problema adicional, que não existia no caso de exceções propagadas entre chamadas locais de métodos, é a necessidade de verificação de tipos em tempo de execução. Um objeto pode estar utilizando um servidor remoto, desenvolvido por outro programador, e compilado dentro de um espaço de nomes diferente. A verificação de tipos, em termos dos parâmetros e resultados dos serviços, é feita em tempo de compilação, mas durante a execução das chamadas de métodos as exceções serão tratadas com base em informações de tipo. Essa questão está sendo estudada em [Oliveira 95].

Estamos especificando uma classe Exception, que poderá ser usada para tratamento mais elaborado de exceções em Cm Distribuído. A idéia é oferecer um conjunto mínimo de informações que poderiam ser proveitosas para tratadores genéricos; isso é particularmente necessário no caso distribuído, porque saber onde ocorreu a exceção pode ser tão importante quanto saber qual o valor que ela traz. Um objeto da classe `Exception` conteria informações como o nó da rede onde a exceção ocorreu, a identificação do objeto remoto que a propagou, qual o método onde ela foi ativada, etc. No caso de exceções específicas da aplicação, o programador poderia derivar novas classe por herança; o próprio ambiente poderia oferecer outras classes pré-definidas, como exceções causadas por *time-out*, falhas de *hardware*, etc.

Objetos configurados externamente

Há casos em que um objeto utilizará serviços de um objeto remoto que já exista: isso será comum no caso de servidores de recursos em rede. Isso pode ser feito de duas maneiras: pelo próprio objeto cliente, ou externamente. No primeiro caso, o objeto declara um apontador para um objeto remoto do tipo do servidor, e coloca no apontador uma referência para o servidor através de uma consulta ao Servidor de Nomes. O exemplo seguinte mostra esse mecanismo:

```
class UseGlobalStack<>
import Stack, Document;
```

```
Remote<Stack<Document, 100>>*  myDocStack;

...
    myDocStack->attach("Staff.Documents.Stack");
    myDocStack->push(..);
...
```

O método `attach()` é oferecido pela classe `Remote<>`, e faz uma consulta ao Servidor de Nomes para encontrar o objeto chamado de "Staff.Documents.Stack"; encontrado esse objeto, o valor do apontador é preenchido com essa referência, e a partir daí a variável `myDocStack` pode ser usada para fazer chamadas de métodos.

Na outra maneira a associação é feita externamente, sem a participação do objeto cliente, caso em que essa informação (servidor externo) faria parte da configuração do objeto. Nesse caso, a classe que utilizará o objeto remoto instanciado externamente deverá ser alterada para mostrar essa dependência.

```
class UseExternalStack<>
import Stack, Document;

URemote<Stack<Document, 100>>  myDocStack;

...
    // myDocStack já deve ter sido instanciado por quem criou este objeto
    .. = myDocStack.pop();
...
```

A classe `URemote<>` é usada para declarar uma referência a um objeto remoto que deve ser suprida externamente, durante a configuração do objeto. Um objeto da classe `UseExternalStack` deve, ao ser criado, ter a variável `myDocStack` associada a um objeto do seu tipo; essa associação deverá ser feita pelo objeto que o criou. Essa forma de construção de aplicações distribuídas será extensivamente usada na LegoShell [Drummond 96].

Escopo e tempo de vida

No caso geral, o tempo de vida de um objeto é determinado pelo escopo em que ele é conhecido. A ativação dos construtores e destrutores corresponde às fronteiras dos blocos, no caso de execução normal, de desvio por comandos como `break` e `return`, e ainda no caso de propagação de exceções. Como resultado, um objeto remoto existirá por um tempo menor ou igual que o tempo do objeto pai, como se fosse um objeto local. O exemplo abaixo ilustra isso:

```
class UseTmpRemoteStack<>
import Stack<>;
...
void useTmpStack ()
{
Remote<Stack> tmpStack;  //  ativa o construtor de tmpStack e cria o objeto remoto
...
    return;// ativa o destrutor de tmpStack e destrói o objeto remoto
...
} // ativa o destrutor de tmpStack e destrói o objeto remoto
```

Dados dois objetos remotos, um criado a partir do outro, dizemos que o objeto criado está *vinculado* se o seu tempo de vida for limitado pelo tempo de vida do objeto pai. Essa relação não é invariável: um objeto pode ser desvinculado do objeto pai, para que o seu tempo de vida prolongue-se pelo tempo em que for útil para outros objetos. Se um objeto for desvinculado do seu pai, fica implícito que outros objetos, possivelmente de outras aplicações, têm uma referência para ele (o objeto desvinculado), pois a referência contida no pai pode desaparecer.

A desvinculação de um objeto é feita por um chamada do método `unbind()`; o efeito é uma dissociação entre o objeto e o seu pai, seguida de uma vinculação do objeto ao gerenciador de objetos do sistema, que é o criador (direto ou indireto) de todos os objetos distribuídos.

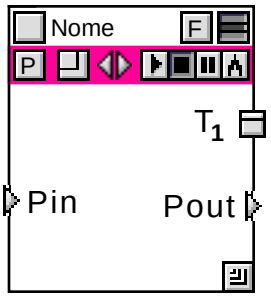
```
class CreateUnboundStack<>
import Stack, Output;

Remote<Stack> unb_st;

constructor() {
    ...
    unb_st.unbind();  // tempo de vida desse objeto não está mais vinculado
}
```

Após a execução o objeto remoto da classe `Stack` continuará existindo, já que ele foi desvinculado do seu objeto pai. Ele permanecerá nesse estado até que ele seja explicitamente destruído pelo seu novo pai, o gerenciador de objetos. Outros objetos podem ser clientes dessa pilha, desde que tenham recebido, do objeto da classe `CreateUnboundStack`, uma referência para ela, ou que ela tenha sido registrada no Servidor de Nomes.

No caso de objetos remotos criados dinamicamente, através da operação `new`, não há um escopo bem definido para a validade da referência para o objeto. Se não for executado um `release` para esse apontador, o objeto remoto sobreviverá ao término do escopo onde foi executado o `new`; mas a vinculação com o seu pai permanece. Portanto, todos os objetos remotos são, por *default*, vinculados ao objeto remoto pai, e a desvinculação será feita através de chamada de método `unbind()`.

Integração com a LegoShell

Um dos objetivos estabelecidos pelo nosso grupo é que os programas distribuídos escritos em Cm possam ser visualizados em uma computação LegoShell equivalente, do ponto de vista funcional e estrutural. A recíproca é verdadeira: o mapeamento de uma computação LegoShell em um programa Cm Distribuído é direto. Classes em Cm Distribuído poderiam inclusive partir de uma interface definida na LegoShell, ficando o trabalho de codificação restrito ao corpo dos métodos e às estruturas de dados encapsuladas na classe.




```
class Nome<>
import T1, T2;

URemote<T1> ...;
UPort<T2, IN> Pin;
UPort<T2, OUT> Pout;
...
```

Figura 3 Ícone de programa da LegoShell e classe Cm correspondente

Na LegoShell objetos Cm podem ser inseridos em uma computação, usando para isso portas de dados ou conectores de objetos (Para mais detalhes vide [Drummond 96]). Em tempo de "edição" das computações, os objetos não existem, mesmo que informações de parametrização sejam fornecidas; esses dados serão usados quando a execução da computação for iniciada: então seus objetos serão criados conforme configurados.

Uma aplicação distribuída em Cm, iniciada a partir de um objeto que cria (direta ou indiretamente) todos os demais objetos, remotos ou não, têm uma correspondência direta com uma computação LegoShell. As regras de vinculação entre objetos podem ser uniformes nas duas linguagens, já que pretende-se chegar a uma situação em que o uso de uma ou de outra linguagem para a montagem das aplicações será questão de conveniência.

Para que as computações LegoShell possam ser diretamente traduzidas em programas Cm, resta criar em Cm objetos como arquivos, dispositivos e conectores, além das portas virtuais das abstrações (para mais detalhes ver [Drummond 96]). Todos esses elementos serão oferecidos através de classes Cm pré-definidas.

EXTENSÕES PARA PROGRAMAÇÃO CONCORRENTE EM CM

Aplicações distribuídas em Cm podem compartilhar objetos para a execução de serviços, criando a necessidade de mecanismos de programação concorrente para regular essa utilização comum. O cenário típico é a criação de um servidor que irá atender a um grande número de clientes, todos executando de forma paralela. Pedidos para a execução de serviços, ocorrendo de forma não-determinística, exigem do servidor a capacidade de sincronizar o atendimento dessas requisições. A seguir descrevemos esse tipo de concorrência e quais os mecanismos implementados no Cm Distribuído para atender a essas necessidades.

Concorrência dentro de objetos

Servidores utilizados por um número arbitrário de clientes, distribuídos pela rede, devem preferencialmente atender requisições de serviços de forma concorrente, para não limitar o paralelismo potencial das aplicações; nesse caso são chamados *servidores multi-tarefa*. Em alguns casos, a concorrência é uma necessidade, quando objetos utilizam um recurso comum de forma sincronizada, através de um *lock*: o bloqueio em um objeto pressupõe a eventual liberação posterior do recurso por outro objeto.

Nos servidores multi-tarefa, cada requisição cria uma nova *thread*, que é responsável pelo atendimento a essa requisição; a *thread* que fez a chamada, no objeto cliente, está bloqueada à espera do resultado do método. Conceitualmente, as duas compõem uma única "*thread* virtual", abstraindo a separação em diferentes espaços de endereçamento; essa relação é importante para manter a uniformidade da execução de chamadas remotas de métodos.

As variáveis compartilhadas pela *threads* serão protegidas de acessos indevidos pelo mecanismo de *regiões críticas condicionais estendidas*. Por esse mecanismo, uma região crítica é associada a objetos especiais que devem ser consultados para garantir o acesso disciplinado aos dados protegidos pela região. A completa especificação desse mecanismo e a sua implementação são trabalhos relacionados em [Targa 95].

Classes threaded

Objetos que possuam a capacidade de atender simultaneamente várias chamadas de métodos devem utilizar os mecanismos da linguagem para disciplinar o acesso concorrente às suas variáveis internas, que são a memória compartilhada pelas *threads* correspondentes às requisições. Essa capacidade deve estar prevista na classe do objeto, através da cláusula `threaded`.

Objetos cuja classe não for declarada como *threaded* poderão executar apenas uma *thread* em qualquer momento; portanto, se forem compartilhados por vários objetos, deverão atender às chamadas de métodos de forma estritamente seqüencial. Pode ser o caso que, dentro de um objeto cuja classe é threaded, o paralelismo seja limitado em função do acesso a objetos seqüenciais.

Regiões críticas condicionais estendidas

O controle de concorrência em Cm Distribuído é feito por uma extensão do mecanismo de região crítica condicional, proposto originalmente em [Hoare 72]. Em Cm Distribuído, essas regiões são

implementadas através do comando `with`, que tem duas expressões: uma expressão de região e uma condição de sincronização. Esse comando é explicado com o exemplo a seguir:

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

T [SIZE] st;
int top = 0;
Region r = 1;

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

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

Essa classe implementa pilhas que podem atender a chamadas simultâneas de métodos. A variável `r` é uma *variável de região*, cujo tipo é a classe pré-definida `Region`. Ela será usada para proteger dados de acesso indevido; no caso, esses dados são a própria pilha e o apontador para o topo. Os métodos `push()` e `pop()` usam o comando `with`, e nos dois casos a expressão de região é apenas a variável `r`, que foi declarada com o valor 1. Isso indica que, nas regiões críticas protegidas por essa variável, a execução é de, no máximo, uma *thread* por vez. As variáveis de região podem ser criadas com outros valores para permitir níveis arbitrários de concorrência dentro das regiões críticas. Esse valor é chamado *cardinalidade* da variável de região; o valor *default* para a cardinalidade de uma variável de região é 1.

O comando `with` ainda contém uma condição de sincronização, que deve ser verdadeira para que o corpo do comando seja executado. No caso do método `push()`, a condição para que um novo dado seja empilhado é que a pilha não esteja cheia (condição `top < SIZE`); e no método `pop()`, a pilha não pode estar vazia (condição `top >= 0`). Uma chamada de `pop()` para uma pilha vazia não é um erro: a *thread* ficará bloqueada até que uma chamada de `push()` coloque um dado na pilha.

As expressões de região podem ser mais complexas, combinando variáveis de região através do operador `&&`. Isso significa que uma expressão de região como "`r1 && r2`" exige que o acesso só pode ser feito respeitadas as restrições impostas pelas duas variáveis de região. Essa possibilidade elimina a necessidade de comandos `with` encadeados, com a vantagem de não bloquear outras *threads* enquanto não consegue acesso por todas as variáveis de região.

Objetos passivos e objetos ativos

Essa distinção não pertence à linguagem, mas sim à própria natureza dos objetos [Gonçalves 94]. Uma estrutura de dados, por exemplo, normalmente fará algum processamento para atender a uma chamada de método, para inserção ou retirada de dados; nos intervalos entre as chamadas, ela permanece ociosa porque não há trabalho a fazer. Por outro lado, há programas que estão continuamente executando, mesmo que não haja demanda num dado momento, porque o seu estado interno pode variar com o tempo. Um exemplo é um simulador de vôo: mesmo que não haja interação com o "piloto" por algum período, o estado do avião simulado (direção, altura, combustível, etc.) muda em função do tempo, devendo então ser computado continuamente.

Do ponto de vista de programação, a diferença entre esses dois tipos de objetos é que, em um objeto passivo, toda execução começa e termina por chamada de método pelos clientes, ou seja, uma *thread* encarregada da execução de um método é sempre uma extensão da *thread* que fez a chamada, no cliente. Nos objetos ativos, podem existir *threads* funcionando desde o momento da construção do objeto, mantendo a coerência do seu estado interno; no caso de chamadas de métodos de clientes, as novas *threads* executarão em paralelo com aquelas que já estiverem ativas.

Para implementar objetos ativos é necessário um mecanismo adicional de chamada de métodos, já que em Cm todas as chamadas são síncronas, bloqueando o chamador até a chegada do resultado. O mecanismo adotado é a declaração de métodos que não devolvem resultado (com tipo `void` e sem parâmetros por referência) e que não bloqueiam o chamador. Métodos assim são declarados com o modificador `thread`; para detalhes vide o exemplo de aplicação mostrado mais adiante.

EXEMPLO DE APLICAÇÃO DISTRIBUÍDA

Nesta seção vamos ilustrar o uso das facilidades do Cm Distribuído através do esboço de uma aplicação distribuída complexa. O exemplo é o TeamSim, uma simulação multi-usuário feita no Laboratório A-HAND [Furuti 94].

Nesse jogo, cada usuário tem em sua *workstation* uma instância do programa, que apresenta uma interface para a condução de um veículo de guerra (tanque, helicóptero ou avião). Há um cenário (uma cidade) virtual comum a todos os jogadores, que interagem com seus veículos e vêem essas interações refletidas no cenário. Vários jogadores podem participar ao mesmo tempo, e também entrar em simulações em andamento. A figura abaixo mostra um conjunto de jogadores participando do TeamSim.

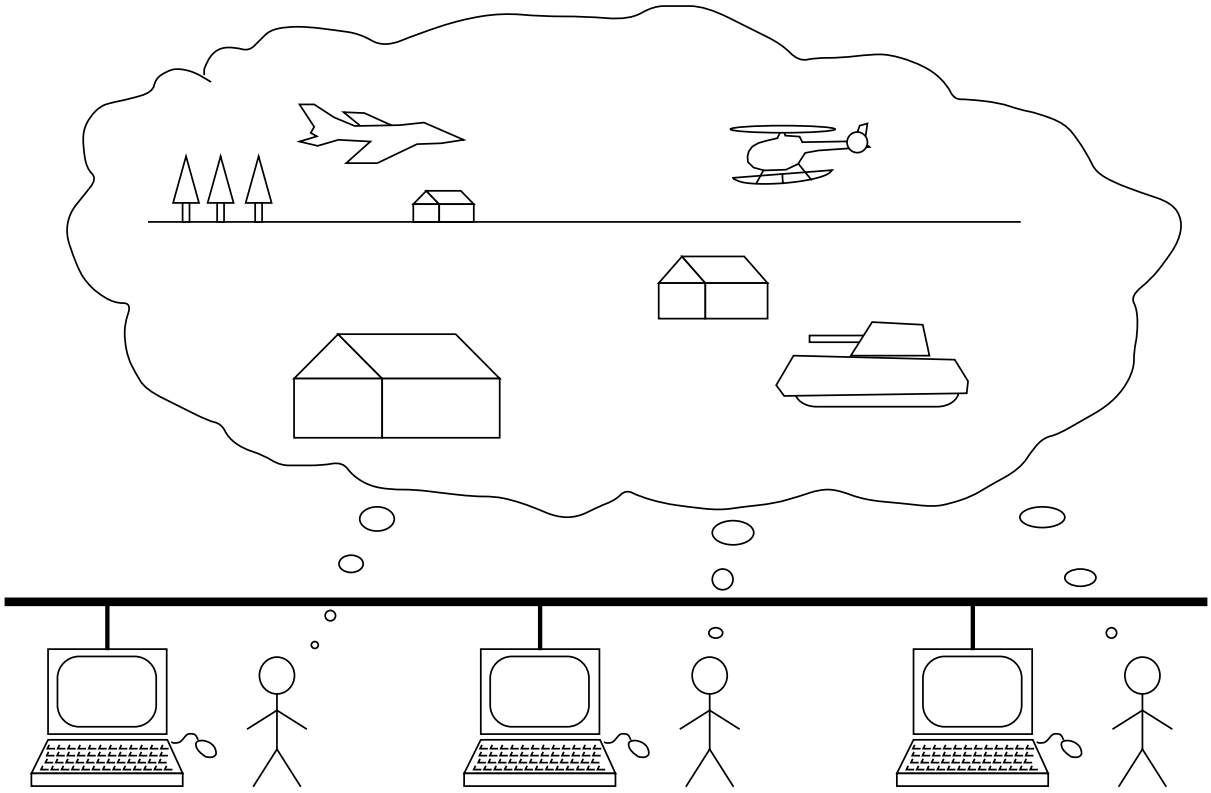


Figura 4 — TeamSim, simulação distribuída

A implementação atual utiliza a biblioteca Astra [Furuti 93] para a construção das interfaces gráficas e o sistema OMNI [Cianni 94] para criação e comunicação de processos distribuídos. O jogo é implementado de forma distribuída, já que cada instância do programa realiza localmente todos os cálculos de simulação; uma alternativa (com desempenho muito pior) seria um programa centralizado que apenas distribuísse as interfaces dos veículos. Uma versão utilizando Cm Distribuído teria a seguinte estrutura: os veículos seriam implementados como objetos ativos, que reagiriam aos comandos emitidos na sua

interface; objetos como mísseis poderiam ser criados dinamicamente; um objeto que representasse o jogo controlaria a entrada de jogadores e a criação de objetos.

Uma boa maneira de visualizar essa estrutura é utilizar a notação da LegoShell, como mostrado no desenho a seguir. O objeto da classe `Game` é uma instância de um jogo, reunindo um conjunto de jogadores; a cada jogador está associada uma interface, pela qual ele controle o seu veículo. Os veículos usam uma porta de dados para informar aos outros componentes as mudanças de seu estado (velocidade, posição, orientação, etc.) que sejam relevantes para cálculo do cenário. Essas alterações são enviadas, através de um conector *broadcast*, para as interfaces (que procederão à atualização do cenário e apresentação ao usuário) e para o objeto controlador do jogo (que manterá o estado corrente de todos os participantes).

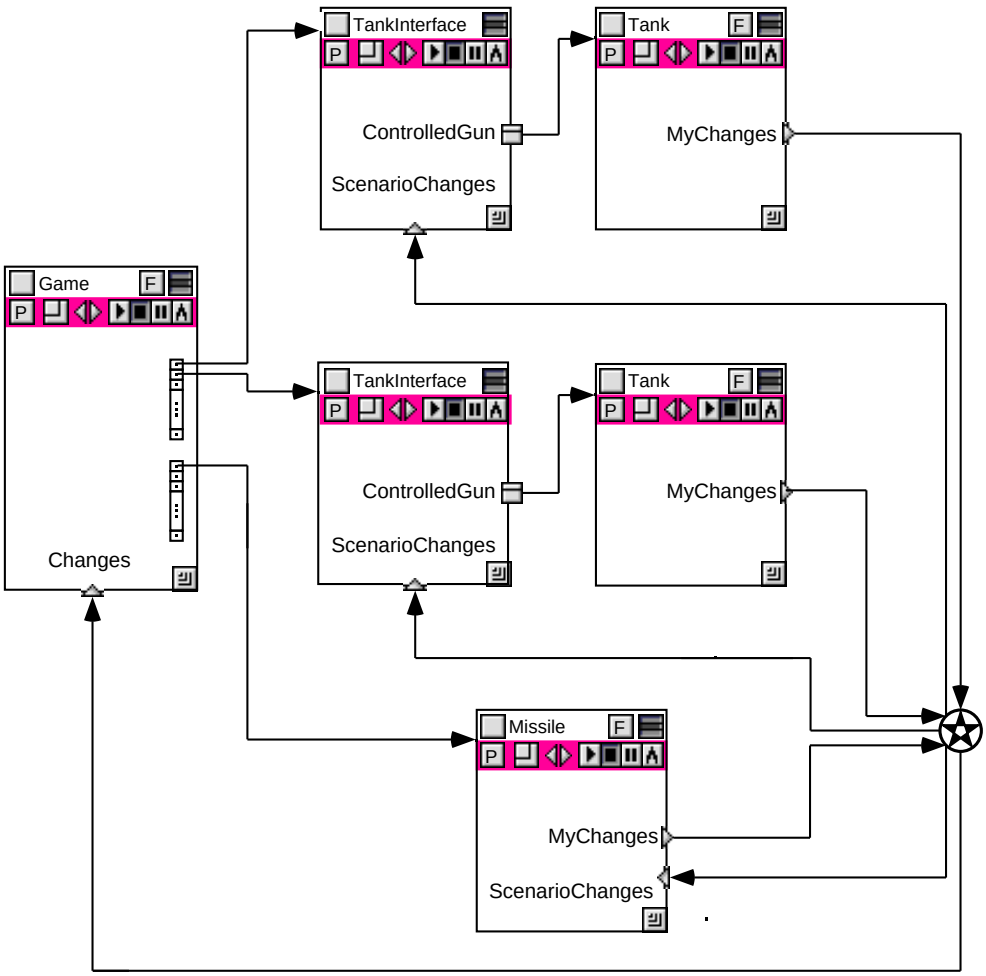


Figura 5 — Visualização da estrutura do TeamSim usando a LegoShell

A classe `Game` é responsável por iniciar um jogo, incluir novos jogadores, criar interfaces, veículos e outros objetos. A seguir mostramos um trecho de uma possível implementação dessa classe.

```
class Game<>
import Vehicle, Tank, Plane, Helicopter, Missile, Bullet, ...
       Interface, TankInterface, PlaneInterface, HelicopterInterface,
       BroadCast, ...;

type PlayerNode = struct {   //  dados de cada jogador
    int code; char* name, site;
```

```
    Remote<Interface>* interf;
    Remote<Vehicle>* vehic;
    };

PlayerNode [..] Players;
Remote<BroadCast> GameBroadCast;
UPort<..> Changes;

void AddPlayer (char* n, s; vehicleType v; int i) {
    ...
    switch (v) {
    ...
    case TANK:
        Players[i].interf = new (Remote<TankInterface> /* parâmetros construtor */);
        Players[i].vehic = new (Remote<Tank> /* parâmetros construtor */);
        attach(Players[i].interf->ControledVehicle, Players[i].vehic);
        connect(Players[i].vehic->MyChanges, GameBroadCast);
        connect(GameBroadCast, Players[i].vehic);
        break;
...
}
```

A inclusão de um jogador consiste da atribuição de um código único a cada um (esse código irá identificar a origem das mensagens disseminadas pelo conector *broadcast*), a criação de uma interface para o veículo escolhido e o veículo propriamente dito. A função `attach()` é usada para configurar o objeto remoto que será usado através da referência `ControledVehicle`, e a função `connect()` é usada para conectar as portas de dados com o conector *broadcast*.

As interfaces dos veículos são objetos separados dos veículos que elas controlam. Isso é conveniente para permitir, por exemplo, que um veículo destruído seja "abandonado" no cenário e que a interface seja reconfigurada para outro veículo recém criado. Outro caso interessante são veículos "inteligentes", que atuam no jogo seguindo uma lógica implementada por programa e que dispensa um jogador. Podemos criar uma classe abstrata para descrever uma interface de veículo, e depois derivar classes específicas para cada interface, já que a visão que o jogador tem do jogo depende do tipo de veículo que ele conduz.

```
threaded class Interface<class V>

URemote<V> ControledVehicle;
UPort<..> ScenarioChanges;

export virtual thread DrawInterface (..);
export virtual thread GetEvents (..);

thread CalculateScenario () {
    for (;;) {
        ScenarioChanges >> ..;  // lê ações dos veículos
        ...
    }
}

constructor(..) {
    ...
    CalculateScenario ();
    DrawInterface()
    GetEvents();
}
```

A classe `Interface` é declarada como `threaded` porque seus objetos (as instâncias de interface para cada jogador) têm concorrência interna. As variáveis `ControledVehicle` e `ScenarioChanges` serão configuradas externamente (como já visto na classe `Game`). O método `CalculateScenario` é um *loop* infinito que lê da porta de dados da interface as ações originadas nos objetos do jogo (inclusive do próprio veículo sendo controlado) e calcula as conseqüências no cenário do jogo. Os outros dois métodos, `DrawInterface()` e `GetEvents()`, também são *loops* infinitos, pois são computações

contínuas (desenho do cenário e captura de eventos de *mouse* e teclado). Como todos esses métodos são declarados como `thread`, a chamada a eles não bloqueia: uma nova *thread* é criada para a sua execução. Os três métodos podem compartilhar dados, usando as regiões críticas condicionais estendidas: um exemplo seria um *display file*, que é alterado pelo cálculo do cenário e lido periodicamente para mudar a visão do jogo na tela. O construtor da classe recebe como parâmetros a situação atual do jogo como armazenada no objeto `Game`, e termina sua execução após disparar a execução dos métodos concorrentes.

Uma interface específica para um veículo é uma subclasse de `Interface`, como mostrado a seguir.

```
threaded class TankInterface<>
import Tank;
inherit Interface<Tank>;

export thread DrawInterface () { for (;;) ...}
export thread GetEvents () { for (;;) ... }
```

O método `DrawInterface()` exibe continuamente a visão do jogo e o método `GetEvents()` interpreta as ações do jogador (através do *mouse* ou teclado) sobre a interface de controle do veículo e envia os comandos correspondentes para o veículo controlado.

Os veículos do jogo podem ser agrupados em uma classe abstrata, que descreva um comportamento e estrutura comuns.

```
threaded class Vehicle<>

UPort<..> MyChanges;

export virtual void IWasHit (..);
export virtual thread Move (..);

constructor (..) {
    ...
    Move ();
}
```

A classe é declarada como `threaded` porque os veículos são objetos essencialmente ativos: deve haver um cálculo contínuo da sua posição (em função de velocidade e direção, por exemplo) ao mesmo tempo em que os comandos vindos da interface irão alterar o estado do veículo. Além disso, há interação com outros veículos, como levar um tiro ou ser perseguido por um míssil. Todo veículo tem uma porta por onde envia alterações do seu estado que possam influir no cenário ou em outros veículos: mudança de posição, velocidade ou orientação; dano visível; disparo contra outros veículos; e assim por diante. O método `IWasHit()` serve para o veículo calcular mudanças no seu estado em função de ser atingido (os dados serão passados por parâmetro); um bom exemplo é o tanque: quando ele é atingido por tiros, sua "couraça" vai sendo danificada até o veículo ser destruído. O método `Move()` é um *loop* infinito que calcula o estado do veículo em função de todos esses eventos. O construtor recebe como parâmetros a posição no cenário onde o veículo "entra" no jogo.

```
threaded class Plane<>
inherit Vehicle;

export void Accelerate (..) {..}
export void Turn (..) {..}
export void FireMissile (..) {..}
export void Shoot (..) {..}
export void IWasHit (..) {..}

export thread Move (..) { for (;;) ... }
```

A classe `Plane` é uma especialização de veículo, acrescentando métodos específicos para um avião. Veículos mais especializados podem ser construídos, com novos métodos (ou métodos redefinidos) e interfaces especializadas de forma correspondente.

IMPLEMENTAÇÃO

A disponibilização do Cm Distribuído é um trabalho com duas frentes: uma nova versão do compilador e o ambiente de suporte à execução da linguagem, construído sobre o sistema operacional. Estamos trabalhando simultaneamente nas duas áreas, em função das alterações ilustradas neste artigo e nos outros trabalhos na área [Oliveira 95, Quadros 95, Targa 95].

Compilador para a versão distribuída

O resultado apresentado é bastante simples em termos de sintaxe: três novas palavras reservadas. Todas as demais diferenças serão implementadas através de classes pré-definidas, propiciando essencialmente duas coisas: o rápido desenvolvimento de aplicações distribuídas, opcionalmente com uso da linguagem LegoShell; e o uso recursos distribuídos através de uma interface orientada a objetos, encapsulando a heterogeneidade desses recursos.

Com a declaração de objetos remotos simplificada com a classe pré-definida `Remote<>`, as alterações na sintaxe da linguagem Cm ficam limitadas às construções de controle de concorrência: a cláusula `threaded` (para classes), `thread` (para métodos) e o comando `with`.

O símbolo terminal `threaded` deverá ser acrescentado à gramática do Cm, e a regra que será alterada será a que contém a palavra reservada `class`, que aparece somente como o primeiro terminal dos arquivos fonte Cm. O símbolo terminal `thread` é um modificador de tipos, que aparecerá nas declarações de métodos juntamente com as classes de armazenamento.

O comando `with` será mais uma alternativa para o símbolo não-terminal que reconhece comandos em Cm (como `if`, `while`, `switch`, etc.). A sintaxe do comando é a seguinte:

<*with_stm*> ::= **with (** <*opt_region_expr*> **;** <*opt_sync_cond*> **)** <*statment*>

A expressão de região e a condição de sincronização são ambas opcionais. A expressão de região pode, conforme exemplificado, combinar variáveis de região com o operador `&&`, e também restrições temporárias de cardinalidade. A condição de sincronização é, em princípio, qualquer expressão válida da linguagem; uma detalhada discussão sobre restrições sobre essa condição (a ausência de efeitos colaterais, por exemplo) está em [Gonçalves 94].

Sistema de suporte à execução

A implementação do Cm Distribuído e da LegoShell depende de um sistema que preencha o *gap* semântico entre o nível de abstração de objetos e os serviços oferecidos pelos sistemas operacionais. A construção desse sistema de suporte à execução (também denominado *Run-time System*) está descrito em [Oliva 95].

O sistema de suporte à execução está dividido em três níveis: encapsulamento do sistema operacional, serviços oferecidos para suporte à execução, e serviços oferecidos pela biblioteca padrão de classes. Essa arquitetura visa, além das vantagens naturais de encaspulamento e modularidade, acomodar a inclusão de novas extensões e facilidades.

O ambiente A-HAND foi projetado e está sendo desenvolvido sobre Unix (sem prejuízo da adoção de outras possibilidades no futuro), com uma série de questões de portabilidade entre as suas diferentes implementações, o que irá diminuir com a adoção de tecnologias como o DCE [OSF 90] e o CORBA [OMG 92]. A camada de encapsulamento do sistema operacional serve para oferecer uma interface uniforme para serviços básicos oferecidos de forma diferente pelas diversas plataformas. Entre esses serviços podemos citar alocação de memória, comunicação e *multi-threading*.

Na camada seguinte, de serviços oferecidos pelo sistema de suporte, incorpora-se a abstração de objetos, com serviços que serão elaborados na camada superior. Essa camada baseia-se em serviços básicos de serviço de nomes e verificação de tipos; a noção de objetos é implementada por operações de criação e destruição e chamada remota de métodos. Aqui é importante salientar que essa camada deve ser implementada de forma independente de linguagem, cabendo a outras camadas de software e ao

compilador implementar o mapeamento adequado. Essa é a mesma abordagem seguida por COOL [Lea 93] e Amadeus [Cahill 93].

Na última camada, o suporte é feito pela biblioteca padrão de classes, num nível de abstração já bastante próxima ao da linguagem Cm. Exemplos desses serviços são portas e conectores, variáveis de região, temporizadores e toda a funcionalidade de objetos remotos.

O sistema de suporte a execução vai usar técnicas de reflexão [Maes 87] para adaptar sua funcionalidade com relação a uma série de aspectos. Objetos gerenciados pelo sistema terão meta-objetos *default*, que poderão ser substituídos conforme seja necessário. Casos típicos para a utilização de reflexão são: a criação (transparente) de objetos replicados, linearização de dados, escalonamento de chamadas de métodos, controle de concorrência, entre outros.

AVALIAÇÃO E CONCLUSÕES

A extensão da linguagem Cm para uma versão distribuída é uma proposta alinhada, quanto à forma de abordagem, com uma série de outros trabalhos atualmente em desenvolvimento. A efetiva implementação dessa versão da linguagem será importante para conferirmos o acerto de algumas decisões de projeto.

Quanto à implementação, a nossa proposta tem vários pontos que a distinguem de outras abordagens presentes na literatura. Enquanto alguns sistemas são construídos sobre arquiteturas bastante elaboradas (como por exemplo COOL [Lea 93], baseado em Chorus), pretendemos implementar a noção de objetos distribuídos sobre plataforma Unix comum. Embora os recursos oferecidos sejam mais limitados, o resultado deve facilitar imensamente o desenvolvimento de aplicações distribuídas em ambientes como o que temos disponível.

A idéia básica do nosso trabalho está bem próxima do conceito de *separate entities* [Meyer 93], proposto para a extensão da linguagem Eiffel; o conceito de objeto remoto, inclusive, surgiu na mesma época em que tomamos conhecimento dessa proposta. Uma diferença importante, entretanto, é que em Cm Distribuído os objetos podem ser concorrentes internamente, ao passo que a extensão sugerida para Eiffel descarta essa possibilidade, argumentando que um mecanismo de controle de concorrência aumentaria muito a complexidade da linguagem. A nossa proposta de região crítica condicional estendida parte do pressuposto que a concorrência pela execução simultânea de métodos compensa essa complexidade adicional; todavia, sempre buscamos alcançar um equilíbrio entre poder de expressão e facilidade de utilização.

Um outro aspecto importante do nosso trabalho, e para o qual não encontramos equivalentes na revisão da bibliografia, é a integração entre as linguagens do ambiente, de uma forma direta e intuitiva. No presente trabalho o material apresentado é insuficiente para avaliar adequadamente essa integração, pois o assunto é mais complexo e merece desenvolvimento em outro artigo. A correspondência entre Cm Distribuído e LegoShell é a nível de programa fonte, não sendo necessário o uso de pré-processadores ou bibliotecas para passar de uma notação a outra. Essa integração depende não apenas de decisões de projeto relativas às linguagens em si, mas também do sistema subjacente, que deverá oferecer o recurso básico de objetos distribuídos.

Vários aspectos podem ser citados como direções futuras do nosso trabalho. Um objetivo já fixado é estabelecer uma correspondência, entre programas em Cm Distribuído e computações LegoShell; em um sentido (LegoShell para Cm) isso é simples, dado que os componentes de uma computação serão objetos remotos gerados segundo classes pré-definidas, que incorporam todas as funções de conexão e configuração. No sentido contrário (de Cm para LegoShell), deve-se buscar um estilo mais declarativo na construção de programas distribuídos em Cm, pois nesse caso pode-se determinar a computação correspondente sem que seja necessário analisar comandos, informação disponível apenas durante a execução.

O Laboratório A-HAND é integrante do Projeto ASAP, projeto de pesquisa da Chamada de Projetos 94/95 do ProTeM–CC e destinado a implementar um ambiente de desenvolvimento de aplicações distribuídas, baseado em objetos; participam também do ASAP o LCMI/UFSC, CAA/UFF e LARC/UFC. O traço mais característico da proposta desse grupo é a construção de um ambiente compatível com padrões industriais (OSF/DCE [OSF 90] e OMG/CORBA [OMG 92]), com uma interface aberta de

utilização; há também contribuições importantes para programação em tempo real e tolerância a falhas. O trabalho exposto neste artigo é parte integrante do ASAP e a sua evolução deverá seguir as diretrizes estabelecidas no projeto.

REFERÊNCIAS

[Assenmacher 93] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, R. Schwarz. Panda – Supporting Distributed Programming in C++. In *Proceedings ECOOP'93, LNCS 707*.

[Cahill 93] Vinny Cahill, Seán Baker, Clovis Horn and Gradimir Starovic. The Amadeus GRT – Generic Runtime Support for Distributed Persistent Programming. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications* (1993), pp. 144-161.

[Di Cianni 94] Di Cianni, C. *OMNI – Sistema de Suporte a Aplicações Distribuídas*. Tese de Mestrado. DCC, Unicamp (setembro 1994).

[Drummond 87] Drummond, R. e Liesenberg, H. A_HAND: Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados. *Anais do IV Encontro de Trabalhos do Projeto ETHOS*, Petrópolis, RJ (abril 1987), pp. 313-322.

[Drummond 88] Drummond, R. e da Silva, F. Linguagem C_m: Manual de Referência. *Anais da IV Reunião de Trabalho do Projeto ESTRA*, SID Infomática (outubro 1988), pp. 175-210. Publicado também como Relatório Interno DCC, Unicamp (maio 1988).

[Drummond 89] Drummond, R. LegoShell: Linguagem de Computações. *Anais do III Simpósio Brasileiro de Engenharia de Software*, Recife, PE (setembro 1989), pp. 2-16.

[Drummond 96] Drummond, R., Gonçalves Jr, C. LegoShell Linguagem Visual de Programação Distribuída. *Artigo aceito para publicação no XIV Simpósio Brasileiro de Redes de Computadores*, Fortaleza CE (maio 1996).

[Furuti 91] Furuti, C.A. *Um compilador para uma linguagem de Programação Orientada a Objetos.* Tese de Mestrado. DCC, Unicamp (julho 1991).

[Furuti 93] Furuti, C. A. The ASTRA User Interface Library. *Anais do VII Simpósio Brasileiro de Engenharia de Software*, Rio de Janeiro, RJ, 1993.

[Furuti 94] Furuti, C.A. TeamSim – Uma demonstração Astra-OMNI. *Caderno de Ferramentas do VIII Simpósio Brasileiro de Engenharia de Software*. Curitiba, PR. 1994.

[Garcia 95] Garcia, I. e Drummond, R. Object Inspector. *I Workshop do Projeto ASAP*, Florianópolis SC (outubro 1995).

[Goldberg 83] Goldberg, A. e Robson, D. *Smalltalk–80: The Language and Its Implementation*. Addison-Wesley, Reading (MA), 1983.

[Gonçalves 94] Gonçalves Jr, C. *Objetos Distribuídos.* Tese de Mestrado. DCC, Unicamp (setembro 1994).

[Gonçalves 95] Gonçalves Jr, C., Teles, A. e Drummond, R. Programação Distribuída na Linguagem Cm. *I Workshop do Projeto ASAP*, Florianópolis SC (outubro 1995).

[Hoare 72] Hoare, C. Towards a Theory of Parallel Programming. In *Operating Systems Techniques*, Academic Press, New York (NY), 1972.

[Lea 93] Lea, R., Jacquemot, C. e Pillvesse, E. COOL: System Support for Distributed Programming. *Communications of the CACM*, 36 (9), pp 37–46 (setembro 1993).

[Maes 87] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications* (1987). pp. 147-155.

[Meyer 88] Meyer, B. *Object-Oriented Software Construction*. Prentice-Hall, New York (NY). 1988.

[Meyer 93] Meyer, B. Systematic Concurrent Object-Oriented Programming. *Communications of the CACM*, 36 (9), pp 56–80 (setembro 1993).

[Oliva 95] Oliva, A. e Drummond, R. Sistema de Suporte à Execução para Cm Distribuído. *I Workshop do Projeto ASAP*, Florianópolis SC (outubro 1995).

[Oliveira 95] Oliveira, M. *Resolução de Tipos e Exceções entre Objetos Remotos*. Proposta de tese de mestrado, DCC/Unicamp (setembro 1995).

[OMG 92] *Object Management Architecture Guide*, OMG TC Document 92.11.1. John Wiley & Sons Inc., New York (NY), 1992.

[OSF 90] *OSF Distributed Computing Environment Rationale*, Open Software Foundation (maio 1990).

[Quadros 95] Quadros, E. *Implementação de Objetos Resilientes*. Proposta de tese de mestrado, DCC/Unicamp (setembro 1995).

[Stroustrup 91] Stroustrup, B. *The C++ Programming Language, 2nd. Edition*. Addison-Wesley, Reading (MA), 1991.

[Targa 95] Targa, C. *Controle de Concorrência no Cm Distribuído*. Proposta de tese de mestrado, DCC/Unicamp (setembro 1995).

[Teles 93] Teles, A. *A Linguagem de Programação Cm*. Tese de Mestrado. DCC, Unicamp (dezembro 1993).