Objetos Concorrentes em Máquina de Estados Abstratos Distribuída

Marcelo de Almeida Maia [1] Roberto da Silva Bigonha
marcmaia@dcc.ufmg.br bigonha@dcc.ufmg.br

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Resumo

A especificação formal de concorrência, em especial de objetos concorrentes, tem desempenhado um papel importante dado o uso difundido de linguagens orientadas a objetos. Trabalhos importantes têm sido desenvolvidos a partir da álgebra de processos. Porém, nesse trabalho é usada uma outra abordagem baseada em máquinas de estados abstratos distribuídas. Será mostrado como um núcleo de uma linguagem concorrente baseada em objetos pode ser traduzida nas respectivas máquinas.

Abstract

The formal specification of concurrency, specially of concurrent objects, has played an important role because of the widespread use of object-oriented languages. Important work has been developed based on process algebra. However, in this work, it is used another approach based on distributed abstract state machines. It will be shown how a core object based language can be translated into the respective machines.

1 Introdução

Recentemente tem crescido o interesse pela especificação formal de linguagens orientadas a objetos. A origem desse interesse pode ser explicada pela necessidade de sistemas de tipos complexos e de um modelo de execução, quase sempre, mais elaborado do que o das linguagens imperativas estruturadas tradicionais. Em relação à especificação de linguagens orientadas a objetos seqüenciais, a evolução têm sido marcante. Alguns trabalhos importantes que vêm consolidando a área podem ser representados por [**?**][**?**].

[1] **Professor Assistente do Departamento de Computação da Univ. Federal de Ouro Preto**

A tentativa de se propor um modelo orientado a objetos concorrentes não é recente[**?**]. A tentativa de formalização da noção de concorrência é ainda mais antiga[**?**], quando Milner argumentou que funções puras impunham dificuldades para expressar concorrência. O trabalho de Milner vem evoluindo no decorrer das últimas décadas. Um de seus últimos resultados, o π-cálculo[**?**, **?**], tem sido bastante estudado. Em geral, cálculos de processos têm se tornado um veículo popular para pesquisa em fundamentos de linguagens concorrentes orientadas a objetos[**?**],[**?**] e [**?**].

Recentemente, Peter Wegner argumentou que é essencial abrir mão da completeza dos modelos formais que fundamentam os cálculos de processos, para que seja possível inserir um nível de expressividade suficiente para especificar a concorrência verdadeira[**?**]. Os modelos tradicionais normalmente fornecem uma concorrência intermeada. Neste sentido, em [**?**] é mostrado um arcabouço para descrever semântica operacional estruturada onde é possível descrever a semântica do π-cálculo baseada em concorrência verdadeira. Portanto, existe uma forte necessidade de que o modelo para descrição de semântica operacional seja expressivo, elegante e abstrato o suficiente para que a especificação formal de concorrência se torne um trabalho factível.

Em [**?**] é descrito um modelo de especificação chamado de *álgebras evolutivas*[2]*,* que posteriormente vieram se chamar *máquinas de estado abstrato (MEAs)*, sendo também estendidas para incorporar o conceito de agentes, e vindo a se chamarem *máquinas de estado abstrato distribuídas (MEADs)*. Em [**?**] argumenta-se a favor do uso de MEAs para especificação de linguagens de programação e de arquiteturas de computadores, para validação de protocolos distribuídos, para provar resultados de complexidade. Em virtude disso, neste trabalho será especificado o núcleo de uma linguagem concorrente baseada em objetos com o uso de uma MEAD, devido ao fato de elas já possuirem o conceito de concorrência embutido.

Na próxima seção será especificada a linguagem núcleo alvo. Depois será revista a abordagem de MEADs. Em seguida, serão especificadas as regras para tradução de um programa na linguagem núcleo para uma MEAD. Ao final concluímos mostrando a efetividade de uma MEAD para especificação de linguagens concorrentes baseada em objetos.

2 Objetos Concorrentes

Um modelo simplificado de objetos concorrentes deve prover mecanismos para declaração de objetos e descrição da interação (possivelmente concorrente) entre os objetos.

Pierce e Turner [**?**] argumentam que o paradigma de programação com objetos concorrentes resulta quase inevitavelmente em linguagens onde processos se comunicam trocando dados via canais. Eles delinearam um modelo básico

[2] *evolving algebras*

para expressar objetos como processos em PICT, no qual um objeto é modelado como um conjunto de processos persistentes representando variáveis de instância e métodos. A interface de um objeto é um registro contendo os canais de todas características exportadas. A possibilidade de passar canais como valores entre processos (a característica marcante do π-cálculo) fornece expressividade suficiente para criar topologias dinâmicas de comunicação.

Neste trabalho, será proposto um núcleo de uma linguagem que seja capaz de incorporar o conceito de monitores proposto por Hoare[**?**], e que fundamenta, por exemplo, objetos concorrentes na linguagem de programação JAVA^{TM} [**?**]. O nosso objetivo não é o núcleo em si, mas sim argumentar sobre a boa legibilidade da abordagem de MEADs.

Considere a sintaxe abstrata da linguagem núcleo, que se preocupa apenas com o modelo de interação entre os objetos:

P ::= "classes" $C_1 ... C_c$
"objects" $O_1 ... O_{n_1}:C_1$ ... $O_1 ... O_{n_c}:C_c$
"init" I_1 ... I_i

C ::= "class" Nome
Var_1 ":" Tipo ... Var_v ":" Tipo
M_1 ... M_m

M ::= "method" Nome "(" NomeEntrada ":" Tipo ")" [":" Tipo]
"{"
I_1 ... I_i
"}"

I ::= O.NomeMetodoProc "(" V ")" |
Var ":=" O.NomeMetodoFunc "(" V ")"

V ::= O.NomeMetodoFunc "(" V ")" |
Var |
constantes pré-definidas

Basicamente, um programa consiste em declarações de classes, instâncias das respectivas classes (objetos) e ativação de serviços (métodos) de objetos, além de um conjunto de instruções que iniciam o programa. Uma classe é composta da declaração de suas variáveis de instância e seus métodos. Um método pode receber um valor de entrada e executa seqüencialmente várias instruções e opcionalmente pode retornar um valor. Uma instrução pode ser uma chamada assíncrona de método que não retorna valor (procedimento), onde também pode ser passada uma informação de saída. Pode também ser uma atribuição de um valor a uma variável. Para efeitos de apresentação, será considerado que o valor atribuído a uma variável será o resultado da chamada de um método síncrono. Um valor pode ser uma variável, uma constante pré-definida da linguagem, ou então uma chamada síncrona de método (função) que retorna um valor básico. Os objetos darão origem aos processos, e uma troca de mensagem implica no estabelecimento de um canal.

Foram feitas várias simplificações, como por exemplo, será considerado que todas variáveis e métodos serão exportados, não haverá estado local nos métodos, não haverá preocupação com sistema de tipo, não serão abordadas as estruturas de controle seqüenciais. Estes aspectos podem ser tratados ortogonalmente no futuro. Além disso, a questão do tratamento do não-determinismo imposto por uma condição de disputa[3] será transferida para a questão de como o modelo das MEAD aborda esse aspecto.

3 Máquinas de Estado Abstrato

Nessa seção será apresentada uma breve descrição das MEAs. Uma descrição completa pode ser encontrada em [**?**].

3.1 Estados

Toda MEA tem um *vocabulário (*ou *assinatura*), ou seja, uma coleção finita de nomes de funções, cada uma com aridade fixa. Todo vocabulário contém certos nomes obrigatórios, incluindo nomes de funções nulas[4], **true, false, undef,** bem como os nomes das funções booleanas usuais e o sinal de igualdade.

Um estado S de uma máquina $\mathcal{E}$ com vocabulário Υ é um conjunto não-vazio X, chamado de *superuniverso* de S, juntamente com as interpretações de cada nome de função em Υ sobre X. Em outras palavras, um estado S é uma Υ-álgebra. As interpretações dos nomes nulos **true, false,** e **undef** são singulares em qualquer S. As interpretações dos nomes das funções booleanas comportam-se da maneira usual em {**true,false**} e tem o valor **undef** caso contrário.

Nomes de funções podem ser rotulados como **external**; a idéia é que nomes de funções externas têm seus valores determinados fora do controle de uma MEA. Um nome de função é chamado de **static** se ele tem a mesma interpretação em qualquer estado S de uma execução particular de uma MEA. Os nomes das funções nulas e dos operadores booleanos são estáticos.

O nome nulo **undef** é usado para representar funções parciais: uma função parcial f retorna o valor **undef** para tuplas (argumentos) fora do domínio. Relações são representadas como funções booleanas. Uma função booleana unária *U(x)* pode ser vista representando um conjunto: $\{x : U(x) = \textbf{true}\}$. Nesse caso, chamamos U de *universo*. Por exemplo, o vocabulário de toda MEA contém o nome de universo *Bool*, e em todo estado de uma máquina, o universo *Bool* contém dois elementos, **true** e **false.**

[3] *race condition*

[4] *nullary functions* = função de aridade zero

3.2 Regras de Transição

Regras de transição descrevem como estados de uma MEA se modificam no decorrer do tempo. Uma *instrução de atualização* é o tipo de regra de transição mais simples e tem a forma $f(\overline{x}) := v$ onde f é um nome de função não-estática, $\overline{x}$ é uma tupla de termos de tamanho apropriado, e v é um termo. Executar tal instrução tem o seguinte resultado: se $\overline{a}$ e b são, respectivamente, os valores de $\overline{x}$ e v no estado corrente, então $f(\overline{a}) = b$ no próximo estado.

Um *bloco de regras* é uma regra de transição, e é simplesmente uma seqüência de regras de transição. Para executar um bloco de regras, execute cada uma das regras na seqüência *simultaneamente*. Conflitos entre regras não são permitidos.

Uma *instrução condicional* é uma regra de transição e tem a forma:

if g_0 **then** R_0
elseif g_1 **then** R_1
$\vdots$
elseif g_n **then** R_n
endif

onde g_i são termos booleanos de primeira ordem e R_i são regras de transição.

Para executar uma regra de transição dessa forma no estado S, determine k, tal que g_k é o primeiro g_i $(1 \leq i \leq n)$ que retorna **true**; se existir esse $k \leq n$ então execute R_k, senão não execute nada.

Seja Υ o vocabulário da álgebra $\mathcal{E}$. Seja Υ^- o conjunto de todos nomes de funções internas de $\mathcal{E}$, chamado de *vocabulário interno*. Um *estado interno* é definido como uma Υ^--álgebra estática. Se S é o estado de $\mathcal{E}$, então S^- denota seu estado interno correspondente. Uma execução de uma MEA seqüencial é uma seqüência de estados. Se S_i e S_{i+1} são estados consecutivos em uma execução, então o estado interno S^-_{i+1} é o resultado de executar todas atualizações habilitadas de $\mathcal{E}$ em S_i.

3.3 Paralelismo e Variáveis Livres

O uso de variáveis livres também é permitido em MEAs. Uma regra de declaração de variável é da forma

var x **ranges over** U
R

onde x é uma variável livre dentro da regra R, e U é o nome do universo. Para executar uma regra de declaração, execute a regra R com x tomando todos os valores possíveis em U. Assim, se U contém n elementos em um dado estado, executar tal regra causa a execução simultânea de n cópias de R, cada uma com um valor diferente para x.

3.4 Máquinas de Estado Abstrato Distribuídas

Um programa seqüencial pode ser visualizado como sendo executado por uma entidade (chamada de *agente*) que em cada passo na execução do programa, avalia as regras do programa e executa as atualizações habilitadas. Um programa distribuído pode ser visualizado como um conjunto de tais agentes, cada um executando independentemente seu próprio programa seqüencial.

Mais formalmente, uma máquina de estados abstratos distribuída contém um vocabulário Υ, um conjunto M de programas seqüenciais chamados *módulos*, e um conjunto I de estados iniciais. Em Υ está incluído o nome de uma função unária *Mod*, cujo domínio é o conjunto de agentes e cuja imagem é o conjunto de nomes de módulos. A função *Mod* mapeia um agente para o programa seqüencial (módulo) que ele executa.

É possível que um módulo seja executado por vários agentes. Isso corresponde à situação da vida real onde um mesmo trecho de código é executado por vários processos. Deve haver um meio para que um agente distingua ele próprio dos outros agentes executando o mesmo módulo. Para tal, é introduzida uma função especial *Self*. Todas funções em Υ são interpretadas identicamente por todos agentes com exceção da função *Self* - cada agente interpreta *Self* para ele mesmo.

Como no caso das álgebras seqüenciais, duas regras disparadas por um mesmo agente não podem conflitar. A definição de uma execução garante que conflitos entre agentes não ocorrerão.

Uma execução de uma MEAD é uma seqüência de estados globais (semântica de ordem parcial para uma execução também pode ser definida). Se S_i e S_{i+1} são estados consecutivos numa execução seqüencial de uma MEAD então S_{i+1}^- é o resultado de executar as atualizações habilitadas para algum subconjunto não conflitante de agentes em S_i.

4 Objetos Concorrentes numa Máquina de Estados Abstratos Distribuída

Nessa seção mostraremos como pode ser feita a tradução de um programa baseado em objetos concorrentes para uma MEAD. Usaremos os esquemas de compilação *U, V, C, M, I, D* para definir o processo de tradução. Os esquemas serão definidos a seguir.

4.1 Definição de Universos e Funções

Para modelar os diversos objetos na MEAD é necessário a criação de um universo que enumere todos os objetos que foram declarados no programa.

U["objects" O_1 ... O_{n_1}: C_1 ... O_1 ... O_{n_c}: C_c] =
universe *Objects* **is** *static, finite*

subset of *Integer* **enum** *1,* n_1+...+ n_c
eval [_] *: Boolean* **expr** *(($1 ≥ 1) And ($1 <* n_1+...+ n_c*))*
universe *Objects*C_1 **is** *static, finite*
subset of *Objects* **enum** *1,* n_1
eval [_] *: Boolean* **expr** *(($1 ≥ 1) And ($1 <* n_1*))*
...
universe *Objects*C_c **is** *static, finite*
subset of *Objects* **enum** $n_1 + ... + n_{c-1} + 1$, $n_1 + ... + n_{c-1} + n_c$
eval [_] *: Boolean* **expr** *(($1 ≥* n_1+...+n_{c-1}+1) *And* ($1 <* n_1+...+n_c*))*

onde n_i é o número de objetos da classe C_i.

A definição acima especifica a criação de universos de objetos. Cada objeto correspondente a um inteiro único. A definição é feita por *enumeração* dos valores que pertencem ao universo. A declaração **eval** define a função característica do universo.

Como o mecanismo de execução de uma MEAD é feito através da execução de atualizações do estado, a chamada de um método será interpretada como uma modificação do estado para sinalizar a possibilidade de execução das instruções correspondentes àquele método. Assim, será criada a tabela de contadores de programas (TPC) de todos os métodos de cada objeto. Portanto deve ser gerada a assinatura de uma função de aridade 2 que retorna o contador de programa - PC - de um dado método de um dado objeto. Por *default* todos contadores começam com o valor igual a zero, o que significa que todos métodos estão inativos. Para iniciar um método basta atualizar seu PC correspondente com um. A geração escreve o seguinte:

function TPC [2] **default** 0;

Também será necessária uma função de aridade 2 que retorna a fase do protocolo de invocação de um método síncrono. A fase pode ser 1 (envio da invocação) ou 2 (recebimento da resposta).

function Fase [2] **default** 1;

A definição das variáveis de instância de uma classe também deve ser traduzida para funções globais, uma para cada variável de cada objeto. Basta gerar apenas a assinatura, pois as atualizações acontecerão no decorrer do programa. Como não assumimos a existência de um sistema de tipos, o *Tipo* que aparece na declaração deve ser um *universo* pré-definido do interpretador em questão utilizado para executar as MEADs. A função *TU* é a responsável pelo mapeamento do *Tipo* no *universo* correspondente. A tradução de cada variável de um dado *Objeto* pode ser feita com a função:

V[Var : Tipo] Objeto =
function *Var* [1] **eval** *TU*[Tipo];

4.2 Tradução de Classes

Para traduzir uma classe será utilizado o conceito de módulo de uma MEAD. Para criar os objetos, os módulos serão parametrizados de acordo com os universos correspondentes aos objetos de cada classe. Serão usadas as funções auxiliares *metodo* e *nr_metodo*, supondo a existência de suas definições. A tradução de uma classe pode ser feita com a seguinte função:

C[“classes” C_1...C_c “objects” O_1...O_{n_1}:C_1 ... O_1...O_{n_c}:C_c] =
module C_1 [o : ObjectsC_1]
if *(*TPC*(*o, 1*) > 0*) **then**
M[metodo(*1,*C_1)] o 1
endif
...
if (TPC(o, *nr_metodos*(1)) > 0) **then**
M[metodo(*nr_metodos(1),*C_1)] o *nr_metodos(1)*
endif
endmodule
...
module C_c [o : ObjectsC_c]
if *(*TPC*(*o, 1*) > 0*) **then**
M[metodo(*1,*C_c)] o 1
endif
...
if (TPC(o, *nr_metodos*(c)) > 0) **then**
M[metodo(*nr_metodos(c),*C_c)] o *nr_metodos(c)*
endif
endmodule

O esquema de compilação acima cria um módulo para cada classe. Cada módulo é parametrizado pelos objetos que pertencem ao universo correspondente à respectiva classe. Assim, cada módulo dará origem a todos os objetos (concorrentes) de uma classe. Cada regra de transição de uma classe dá origem a um método. O esquema M que compila os métodos serão definidos a seguir.

4.3 Tradução de Métodos

Agora será especificada a tradução da declaração dos métodos através do esquema M. Em seguida, será mostrado como traduzir um comando que corresponda à chamada assíncrona de um método (procedimento) e como traduzir um valor que corresponda à chamada síncrona de um método (função). Antes será mostrado como funcionarão os protocolos de chamada síncrona e assíncrona.

Será criada uma função para modelar toda a comunicação entre os objetos. Essa função é um estado global e representa um *pool* de mensagens trocadas entre os objetos. Será suposta a existência dos seguintes procedimentos/funções de

manipulação do *pool*: *Insertpool* - insere uma mensagem no *pool* guardando como resultado um identificador único de mensagem, *Getpool* - dado um identificador de mensagem e um rótulo para seleção de um componente da mensagem, retorna o valor de um componente da mensagem, *Deletepool* - dado um identificador de mensagem, remove essa mensagem do *pool*. Uma mensagem é representada por uma lista (tupla) de oito elementos, respectivamente, o objeto destinatário, o método que está sendo invocado, o ponto de entrada no método, o objeto remetente, o método de onde saiu a mensagem, o ponto de retorno da chamada, o conteúdo da mensagem, e o ambiente (representado pelo identificador da mensagem que ativou o método atual).

O ambiente de execução de um determinado método de um objeto será representado por uma função de aridade 2 (*Ambiente(o,m)*) cujo resultado é o identicador de uma mensagem no *pool*, já que o *pool* concentra todas as informações para composição do ambiente. Quem modifica o ambiente de execução de um método é o módulo *Despachante* a ser definido posteriormente.

O protocolo de chamada síncrona funciona em três fases:

1. Deve ser colocada no *pool* uma mensagem com todas as informações preenchidas de acordo com o contexto atual. O contador de programa do método que enviou a mensagem deve ser zerado para que ele possa esperar o resultado chegar. O contador de programas do método invocado deve ser atualizado para o valor um.

2. No corpo do método (função) destinatária podem haver acessos ao conteúdo da mensagem no *pool*. Ao final do processamento do método: a mensagem deve ser retirada do *pool;* o contador de programas do método atual é zerado; o contador de programa do método remetente deve ser atualizado com o valor do ponto de retorno; o ambiente do método remetente deve ser recuperado; e deve ser colocada uma mensagem no *pool* cujo destinário é o objeto e método remetentes e cujo valor é aquele que foi calculado no decorrer da execução do método. O objeto de origem e o ponto de retorno não são relevantes na mensagem de resposta.

3. No objeto remetente, o método original volta a ficar ativo a partir do ponto de retorno, e pode ser feito acesso ao *pool* de mensagens para acessar o resultado da função. Após o acesso, a mensagem deve ser retirada do *pool* e o ambiente original deve ser recuperado.

O protocolo de chamada assíncrona é mais simples e funciona em duas fases:

1. Deve ser colocada no *pool* uma mensagem com as informações preenchidas de acordo com o contexto atual. Não precisam ser colocadas na mensagem os dados referentes à origem (objeto, método, e contador de retorno). O contador de programa do método que enviou a mensagem é incrementado

normalmente. O contador de programa do método invocado deve ser atualizado para o valor um.

2. No corpo do método (função) destinatário podem haver acessos ao conteúdo da mensagem no *pool*. Ao final do processamento do método: a mensagem deve ser retirada do *pool,* o contador de programa do método atual é zerado.

É importante notar que a atribuição de um contador de programa para iniciar a execução de um método, assim como a atribuição do respectivo ambiente para a execução será controlada pelo módulo *Despachante.* Esse controle garante a ausência de condições de disputa na máquina.

Seja o esquema de compilação da declaração de um método síncrono:

M[“method” Nome “(” Entrada “:” Tipo “) :” Tipo “{” I_1...I_n “}”] o m =

```
    if (TPC(o, m) = 1) then
        I[I1] o m;
    elseif ...
    elseif (TPC(o,m) = n) then
        I[In] o m;
    elseif (TPC(o,m) = n+1) then
        Deletepool( Ambiente(o,m));
        TPC(o,m) := 0;
        Insertpool(MakeList8(
            Getpool(Ambiente(o,m),“obj_rem”),
            Getpool(Ambiente(o,m),“met_rem”),
            Getpool(Ambiente(o,m),“end_ret”), _,_,_,
            ReturnValue(o,m), Getpool(Ambiente(o,m),“ambiente”)));
    endif
```

Seja o esquema de compilação da declaração de um método assíncrono:

M[“method” Nome “(” Entrada “:” Tipo “) {” I_1...I_n “}”] o m =

```
    if (TPC(o, m) = 1) then
        I[I1] o m
    elseif ...
    elseif (TPC(o,m) = n) then
        I[In] o m
    elseif (TPC(o,m) = n+1) then
        Deletepool( Ambiente(o,m));
        TPC(o,m) := 0;
    endif
```

A compilação de um método gera uma regra de transição que executa, em um dado passo, somente a instrução indicada pelo contador de programa corrente daquele método. Ao final da execução de todas as instruções, o contador de programas do método é zerado e as devidas atualizações são processadas. A seguir será mostrado o esquema de compilação de uma instrução.

Seja o esquema de compilação para uma invocação de um método assíncrono. Pressupõe-se a existência de um esquema de compilação V para todo tipo de valor.

I[O.M "(" V ")"] o m =
    Insertpool(
      MakeList8(*nr_obj* [O], *nr_met* [O.M], 1,_, _, _, *V*[V], _));
    TPC(o, m) := TPC(o, m) + 1;

O esquema acima simplesmente coloca a chamada no *pool* e passa a executar a próxima instrução do método.

Seja o esquema de compilação para uma invocação de um método síncrono com a atribuição do resultado a uma variável de instância:

I[Var ":=" O.M "(" V ")"] o m =
    if (Fase(o, m) = 1) **then**
        Insertpool(MakeList8(*nr_obj* [O], *nr_met* [O.M], 1,
                o, m, TPC(o,m), *V*[V], Ambiente(o,m));
        TPC(o, m) := 0;
        Fase(o, m) := 2;
    elseif (Fase(o, m) = 2) **then**
        Var(o) := Getpool(Ambiente(o,m),"val_msg")
        Deletepool(Ambiente(o,m));
        TPC(o, m) := TPC(o,m) + 1;
        Fase(o, m) := 1;
        Ambiente(o,m) := Getpool(Ambiente(o,m), "ambiente");
    endif

No esquema acima, tem-se que o método na fase 1 insere a chamada no *pool* e suspende sua execução, esperando ser ativado na fase 2, quando o método invocado encerrar a sua execução. Na fase 2, o valor recebido é recuperado do *pool*, a atribuição é feita, e em seguida são feitas as atualizações para encerrar a instrução.

4.4 Controlando a Execução dos Métodos

Por fim, devemos garantir que somente existe uma incarnação de um método executando em um determinado instante. Essa exigência deve ser obedecida pois essa foi a semântica proposta na linguagem núcleo, onde haveria um processo para cada objeto. Em princípio, pode haver paralelismo na execução de métodos diferentes de um dado objeto, porém a responsabilidade para evitar condições de disputa ficará a cargo do programador.

A idéia básica para satisfazer essa restrição é selecionar qual invocação de um determinado método será ativada através do *pool*. Portanto, existirá um módulo *Despachante* que seleciona uma mensagem do *pool* e a ativa de acordo com o ponto de entrada - PC. Essa seleção só ocorre quando o método está desativado, caso contrário a regra não dispara a seleção. Será suposta a existência da função

Selectmeth(o,m) que retorna o PC para execução do método (o PC pode ser diferente de um nos retornos de chamada de função. Também será suposta a existência da função que retorna o número único do objeto *nr_obj*.

D["objects" O_1...O_{n_1}:C_1 ... O_{1_c}...O_{n_c}:C_c] =

module Despachante
if *(*TPC*(*nr_obj[O_1] , 1*) = 0)* **then**
TPC(nr_obj[O_1], 1) := SelectMeth(nr_obj[O_1] ,1);
endif
...
if (TPC(nr_obj[O_{n1}] , *nr_metodos*(nr_obj[O_{n1}])) = 0) **then**
TPC(nr_obj[O_{n1}],1) :=
SelectMeth(nr_obj[O_{n1}],*nr_metodos(*nr_obj[O_{n1}] *));*
endif
...
if *(*TPC*(*nr_obj[O_{1c}] , 1*) = 0)* **then**
TPC(nr_obj[O_{1c}], 1) := SelectMeth(nr_obj[O_{1c}] ,1);
endif
...
if (TPC(nr_obj[O_{nc}] , *nr_metodos*(nr_obj[O_{nc}])) = 0) **then**
TPC(nr_obj[O_{nc}],1) :=
SelectMeth(nr_obj[O_{nc}],*nr_metodos(*nr_obj[O_{nc}] *));*
endif
endmodule

Repare que o módulo Despachante dará origem a um único agente, pois o módulo não é parametrizado. Esse agente é composto por regras de transição que checam quais métodos estão inativos, selecionam alguma ativação para tais métodos, e atualizam o contador de programa dos respectivos métodos de acordo com a informação que está no *pool*.

5 Considerações Finais

Nesse trabalho foi proposta uma linguagem núcleo para expressar objetos concorrentes. A granularidade de concorrência (aquilo que gera um processo) foi colocada no nível dos objetos. O modelo é simples, porém impõe dificuldades de sincronização suficientes para demonstrar o poder de especificação das MEADs. Uma grande vantagem foi o tratamento automático de agentes.

Uma outra alternativa semântica mais audaciosa seria colocar a granularidade de concorrência no nível de chamada de métodos. Essa abordagem seria interessante no sentido que modelaria mais facilmente a noção de processos leves [**?**]. Além disso, pode ser explorada construções de sincronização para facilitar a escrita de programas sem condições de disputa. Com isso, aparecerá a necessidade de fornecer mecanismos para a verificação do protocolo de comunicação e

sincronização dos objetos. Essa é uma importante área de continuação do trabalho, pois vai requerer do modelo das MEADs uma fácil interação com algum mecanismo de prova de propriedades adequado.

Depois da apresentação da tradução, espera-se que os problemas que aparecem na definição de um protocolo tenham se tornado mais claros, assim como, as respectivas soluções tenham se tornado fáceis de se compreender. Isso porque o uso de MEADs, apesar de especificar a semântica operacional de um sistema, possibilita um nível de abstração suficiente para se atentar somente aos problemas relevantes da especificação. Achamos que o objetivo foi atingido pois as propostas de novas extensões à linguagem núcleo apareceram naturalmente.