Utilização de Máquinas de Estado Abstratas em Aplicações de Inteligência Artificial e Jogos

Vladimir O. Di Iorio, Alcione P. Oliveira

Universidade Federal de Viçosa,
Departamento de Informática,
Viçosa, Brasil, 36570-000
{vladimir,alcione}@dpi.ufv.br
and
Eliseu C. Miguel

FAGOC - Faculdade Ubaense Ozanan Coelho,
Departamento de Informática,
Ubá, Brasil, 36500-000
emiguel@fagoc.br
and
Roberto S. Bigonha, Mariza A. S. Bigonha

Universidade Federal de Minas Gerais,
Departamento de Ciência da Computação
Belo Horizonte, Brasil, 30123-970
{bigonha,mariza}@dcc.ufmg.br

Abstract

The *Abstract State Machines* (ASM, for short) are a formalism created to model algorithms at its natural abstraction level. In this article, we discuss the use of ASM for the specification of agents in Artificial Intelligence applications, presenting examples which are graphically represented in two dimensions with animation. The text also shows how to create simple computer games written in an ASM-based language.

Key words: Abstract State Machines, Artifical Intelligence, intelligent agents, computer games.

Resumo

As *Máquinas de Estado Abstratas* (ASM, do inglês *Abstract State Machines*) são um formalismo criado com o objetivo de simular algoritmos em uma maneira direta, no nível de abstração desejado. Por meio de exemplos, este artigo discute a adequação do formalismo ASM para a especificação do comportamento de agentes em aplicações de Inteligência Artificial. Para facilitar o entendimento, as especificações são representadas utilizando-se gráficos animados em duas dimensões. O texto explora ainda a possibilidade de utilizar ASM na formulação de jogos simples de computador.

Palavras-Chave: Máquinas de Estado Abstratas, Inteligência Artificial, agentes inteligentes, jogos.

1 Introdução

No ensino de Inteligência Artificial, muitas vezes são propostos problemas aos alunos cuja solução deve ser apresentada na forma de um agente que reage e atua sobre o ambiente, de acordo com regras pré-estabelecidas, de modo a resolver o problema. Mesmo sendo simplificados, os problemas podem servir como interessantes pontos de partida para o entendimento de conceitos como a formalização de modelos e técnicas de Inteligência Artificial. O paradigma lógico é o mais utilizado na especificação do comportamento dos agentes.

A utilização de representações gráficas pode auxiliar o entendimento do funcionamento dos algoritmos formulados para especificar o comportamento de agentes em Inteligência Artificial. Esse recurso pode servir também como estímulo para o aprendizado, uma vez que assim é possível visualizar os resultados, no lugar de apenas imaginar um modelo abstrato. Um dos recursos apresentados neste artigo é uma biblioteca de funções que permite uma fácil especificação de modelos gráficos animados em duas dimensões. Essas funções podem ser utilizadas para representar visualmente a evolução de modelos de Inteligência Artificial e também para criar jogos simples de computador, com animação.

As *Máquinas de Estado Abstratas* (*ASM*, do inglês *Abstract State Machines*) são um formalismo criado por Yuri Gurevich [5], utilizado na descrição da semântica de linguagens de programação [3, 6], arquiteturas de sistemas [2], protocolos distribuídos [7] etc. O estado (ou álgebra) de um sistema é descrito por um conjunto de funções. Regras simples definem atualizações para essas funções, especificando como novos estados são gerados, produzindo uma evolução do sistema.

Acreditamos que o formalismo das Máquinas de Estado Abstratas pode ser usado com sucesso para especificar o comportamento de agentes em aplicações de Inteligência Artificial, no lugar de linguagens lógicas. Neste artigo, vamos explorar essa possibilidade usando exemplos significativos. Para maximizar o entendimento dos algoritmos formulados e aumentar a motivação para o estudo, os modelos terão uma representação animada, usando as facilidades oferecidas pela biblioteca gráfica descrita anteriormente. Com esses mesmos recursos, mostraremos que ASM pode ser usado com sucesso também para criar jogos de computador simples. Finalmente, vamos discutir alguns recursos que uma linguagem baseada em ASM deve oferecer para se adequar melhor ao domínio explorado, podendo ser usada para construir ambientes de definição de jogos envolvendo agentes inteligentes.

2 Máquinas de Estado Abstratas

Nesta seção, apresentaremos uma rápida introdução ao formalismo das Máquinas de Estado Abstratas, nos concentrando no modelo seqüencial. Agentes e execução concorrente serão abordados mais adiante. As definições visam tornar o artigo autocontido, mas uma abordagem mais completa e formal sobre ASM pode ser encontrada em [5], [4] e [11].

2.1 ASM Seqüenciais

A *assinatura* de uma ASM seqüencial $\mathcal{A}$ é uma coleção finita de nomes de funções, cada uma com uma aridade fixa. Um *estado* de $\mathcal{A}$ é um conjunto não vazio, o *superuniverso*,

junto com interpretações dos nomes da assinatura em funções sobre os elementos do superuniverso. As interpretações são designadas *funções básicas* do estado. À medida que $\mathcal{A}$ muda de estado, as interpretações dos nomes podem ser alteradas, mas o superuniverso se mantém inalterado. Funções básicas que não se alteram são chamadas de *funções estáticas*, enquanto as demais são chamadas *funções dinâmicas*.

Formalmente, supondo um superuniverso X, uma função básica de aridade r é uma função $X^r \rightarrow X$. Quando $r = 0$, a função é designada *elemento distinto*. O superuniverso sempre contém os elementos distintos *true*, *false* e *undef*, definidos como *constantes lógicas*. O elemento *undef* é utilizado para representar funções parciais, por exemplo, $f(\overline{a}) = \mathit{undef}$ significa que f é indefinida para a tupla $\overline{a}$. Uma relação *r-ária* sobre X pode ser vista como uma função $X^r \rightarrow \{\mathit{true}, \mathit{false}\}$. Um *universo* U é um tipo especial de função básica: uma relação unária geralmente identificada pelo conjunto dos elementos x tais que $U(x) = \mathit{true}$, i.e., $\{x : U(x)\}$.

Um programa de $\mathcal{A}$ é uma regra de transição, que pode ser composta por regras básicas e não básicas. As regras básicas são: *regra de atualização*, *construtor de bloco* e *construtor condicional*.

Uma regra de atualização tem o formato $f(\overline{t})$:= t_0, onde f é o nome de uma função da assinatura de $\mathcal{A}$, $\overline{t}$ é uma tupla de termos cujo tamanho é igual à aridade de f e t_0 é outro termo. Os termos não possuem variáveis livres e são construídos recursivamente usando-se nomes de elementos distintos e aplicação do nome de uma função a outros termos. De maneira informal, a semântica é a seguinte: a tupla $\overline{t}$ é avaliada, e o valor da função básica f aplicada à tupla é alterado para o valor da avaliação de t_0. Ou seja, o nome f passa a ter uma nova interpretação.

Um construtor condicional tem genericamente a forma

if g_0 **then** R_0 **elseif** g_1 **then** R_1 ... **elseif** g_k **then** R_k **endif**

Sua semântica é a seguinte: uma regra R_i, $0 \leq i \leq k$, será executada se os termos booleanos $g_0, ..., g_{i-1}$ são avaliados para *false* e g_i é avaliado para *true*.

Um construtor de bloco é um conjunto de regras. Sua semântica é a seguinte: todas as possíveis atualizações das regras contidas no bloco são disparadas em paralelo. Se uma atualização contradiz outra, uma escolha não determinística é realizada.

Regras não básicas utilizam variáveis. Permitem um maior poder de expressão possibilitando, por exemplo, estender universos com a importação de elementos ou introduzir não-determinismo. Um exemplo é a regra *var*:

var v **ranges over** U R_0 **endvar**

onde v é uma variável, U é um universo finito e R_0 é uma regra. O efeito dessa regra é criar uma instância de R_0 para cada elemento pertencente ao universo U. Em cada instância de R_0, a variável v está associada ao elemento correspondente de U. Após criadas as instâncias, todas são executadas em paralelo.

Uma execução de um programa de $\mathcal{A}$ é uma seqüência de estados, onde o estado seguinte é obtido a partir do anterior através da execução da regra de transição do programa. A maioria das implementações de ASM determina o final da execução quando o

```
var i rangesover [2..tam]
  posX(i) := posX(i-1)
  posY(i) := posY(i-1)
endvar,
if mov = CIMA then posY(1) = posY(1) + 1
elseif mov = BAIXO then posY(1) = posY(1) - 1
elseif mov = ESQ then posX(1) = posX(1) - 1
elseif mov = DIR then posX(1) = posX(1) + 1
endif
```

(a) Movimento da serpente.

```
if (posX(1) > MAXX)
   or (posX(1) < MINX)
   or (posY(1) > MAXY)
   or (posY(1) < MINY) then
  stop
endif,
var i rangesover [2..tam]
  if (posX(i) = posX(1))
     and (posY(i) = posY(1)) then
    stop
  endif
endvar
```

(b) Verifica posição da cabeça.

```
if tecla(KEYUP) then
  mov := CIMA
elseif tecla(KEYDOWN) then
  mov := BAIXO
elseif tecla(KEYLEFT) then
  mov := ESQ
elseif tecla(KEYRIGHT) then
  mov := DIR
endif
```

(c) Sentido de deslocamento.

Figura 1: Regras para o "jogo da serpente".

disparo da regra de transição não produz nenhuma atualização. Ou então um comando especial, por exemplo, `stop`, indica explicitamente o término da execução.

Para permitir uma interface com o mundo externo, o modelo ASM oferece *funções externas*. Uma função externa não precisa retornar necessariamente um mesmo valor para chamadas com os mesmos parâmetros, se essas forem disparadas em passos diferentes de uma execução.

2.2 Exemplo de ASM Seqüencial

No exemplo a seguir, usaremos boa parte dos conceitos apresentados na Seção 2.1. O exemplo consiste na formalização simplificada de um jogo conhecido como "jogo da serpente". Uma serpente é formada por um determinado número de células, que se deslocam seguindo o caminho traçado pela cabeça, que é a primeira célula da seqüência. Ao se deslocar sobre um espaço de duas dimensões, a cabeça não pode ultrapassar limites pré-determinados na horizontal e vertical, e além disso não pode ocupar uma posição que esteja sendo ocupada no momento por alguma outra célula da serpente.

Na formalização a seguir, os elementos distintos estáticos `CIMA`, `BAIXO`, `ESQ` e `DIR` são usados para indicar o sentido de deslocamento da serpente. O nome `mov` é interpretado como um desses elementos. Os nomes `tam`, `MINX`, `MAXX`, `MINY` e `MAXY` são interpretados como elementos distintos estáticos inteiros. Finalmente, as funções dinâmicas `posX` e `posY` possuem aridade 1 e mapeiam cada célula da serpente, identificada por um número entre 1 e `tam`, em posições dentro do espaço bidimensional.

Para executar um movimento da serpente, pode-se usar um bloco de regras como o exibido na Figura 1(a). A atualização dos valores das funções `posX` e `posY` é feita simultaneamente, para $i = 2, \ldots$ `tam`. A cabeça é a célula cuja posição está representada em `posX(1)` e `posY(1)`. A Figura 1(b) exibe um bloco de regras para verificar se a posição da cabeça é válida. Na Figura 1(c), é usada uma função externa `tecla` para alterar o sentido do movimento da serpente. Essa função deve ser interpretada como *true* se uma determinada tecla foi pressionada.

A partir de um estado inicial adequado, as regras da Figura 1 podem ser executadas em paralelo, ou seja, agrupadas em um único bloco. Outra possibilidade seria usar uma "variável" de controle para executar as regras (a), (b) e (c) em passos seqüenciais.

3 Gráficos Animados em Duas Dimensões

Para demonstrar graficamente as transformações de modelos criados usando ASM, desenvolvemos um conjunto de funções que permitem uma rápida elaboração de gráficos animados em duas dimensões. As funções foram desenvolvidas sobre a biblioteca gráfica Clanlib [1]. Essa biblioteca de código aberto para desenvolvimento de jogos foi escrita na linguagem C++ e roda em plataformas Windows e Linux.

As funções oferecidas podem ser acessadas, dentro de uma especificação ASM, como se fossem funções externas. Um exemplo disso é a função `tecla`, usada no código da Figura 1(c). Imagens e outros recursos como som são especificados usando-se arquivos de recursos, com um formato definido pela biblioteca Clanlib. Após a criação de um objeto animado, seus atributos são armazenados pelo sistema: imagem, posição, camada de exibição e outros. Para produzir uma animação, um usuário deve modificar esses atributos e executar uma função externa para repintar a tela.

Na Figura 2(a), pode-se ver um trecho de um arquivo de configuração, simplificado, para o jogo da serpente. Esse arquivo define as imagens que serão usadas. Observe que a serpente utiliza cinco imagens diferentes. As imagens estão definidas em um arquivo "serpente.pcx", que pode ser visto na Figura 2(b). Na Figura 2(c), é exibido um *snapshot* do jogo, onde foram incluídas outras regras para sortear aleatoriamente "vitaminas", que quando "ingeridas" fazem a serpente crescer no comprimento. O objetivo do jogo é atingir o maior comprimento sem violar as condições estabelecidas.

Para realizar testes como o apresentado nesta seção, utilizamos a linguagem Machĭna[10], baseada no modelo ASM. O código das especificações foi traduzido para C e integrado com as funções externas da biblioteca gráfica, escritas em C++. Um arquivo executável do jogo da serpente pode ser obtido no endereço eletrônico `www.dpi.ufv.br/~vladimir/asm/serpente.zip`.

4 Aplicações em Inteligência Artificial

Ao longo dos anos, a área de Inteligência Artificial (IA) desenvolveu inúmeras técnicas de resolução de problemas e representação do conhecimento no intuito de solucionar problemas para os quais não existe uma solução algorítmica ou, pelo menos, uma solução algorítmica de complexidade computacional aceitável. No entanto, faltava, até recentemente, um arcabouço (*framework*) que agrupasse todas essas técnicas, permitindo uma

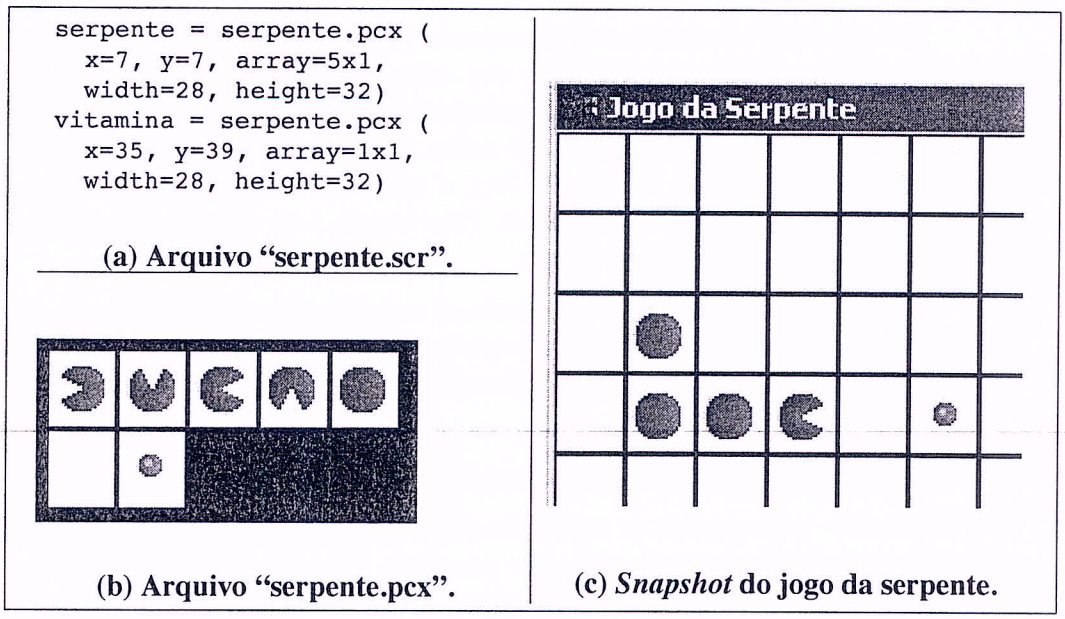

Figura 2: Definindo imagens para os objetos animados.

forma organizada de projeto e análise das soluções. O arcabouço surgido para preencher essa lacuna foi a definição de soluções usando *agentes* ou *sistemas de agentes*.

4.1 Agentes em Inteligência Artificial

Agentes possuem diversas definições dentro de IA. Umas das mais abrangentes é a proposta por Russel e Norvig [9] que diz que "um agente é qualquer coisa que pode ser vista como percebendo seu ambiente por meio de sensores e atuando sobre esse ambiente por meio de *atuadores*". Com essa definição é possível enquadrar um grande número de entidades como um agente.

Contudo, segundo os mesmos autores, a IA está interessada em um tipo particular de agentes, capazes de realizar ações racionais, denominados *agentes racionais ideais*. Eles definem um agente racional ideal da seguinte forma: *para cada possível seqüência de percepções, um agente racional ideal deve fazer a ação que espera-se maximizar sua medida de desempenho, com base nas evidências fornecidas pela seqüência de percepções e pelo conhecimento embutido no agente* (tradução dos autores).

Russel e Norvig também classificam os agentes segundo sua arquitetura nos seguintes tipos, listados em ordem de complexidade de implementação:

agente reflexivo ou reativo: é o mais simples de todos e simplesmente reage às alterações do ambiente segundo suas regras internas;

agente com estado: é um agente reflexivo que leva em consideração o histórico das percepções;

agente com objetivo: possui característica adicional de selecionar a regra que mais o aproxima de cumprir um determinado objetivo;

agente baseado em utilidade: procura maximizar sua medida de utilidade.

Neste artigo, estamos interessados na especificação de agentes reativos e agentes com estado. Apesar de sua simplicidade, esses tipos de agentes possuem uma ampla gama de

aplicações, principalmente em ambientes dinâmicos, que demandam respostas rápidas, como é o caso de ambientes de jogos e simuladores.

4.2 Exemplo: *Mundo Wumpus*

Um bom exemplo para utilização de agentes racionais ideais é o problema conhecido como *Mundo Wumpus* [9]. Na versão simplificada desse problema abordada nesta seção, um agente situa-se em um ambiente definido por um quadrado de duas dimensões, cercado por paredes. O agente pode se movimentar por esse espaço, caminhando uma célula de cada vez, e receber informações que definem sua percepção do ambiente. Uma das células contém um *Wumpus*, monstro que deve ser evitado, e outra célula contém um pote de ouro, que deve ser encontrado.

As ações que o agente pode executar são: **Turn(Right)** (virar à direita), **Turn(Left)** (virar à esquerda), **Walk** (caminhar para frente) e **Grab** (pegar objeto que esteja na mesma célula). As percepções que o agente tem do ambiente são: **Stench** (cheiro do Wumpus, percebido nas quatro células vizinhas ao monstro), **Bump** (percebido quando "tromba" em uma parede) e **Glitter** (brilho percebido na célula que contém o pote de ouro). A definição original do Mundo Wumpus inclui ainda buracos que devem ser evitados e uma arma que pode ser usada contra o monstro.

4.3 Um Agente Inteligente Usando ASM

Nesta seção, vamos usar o modelo ASM para formalizar o comportamento de um agente que navega dentro do Mundo Wumpus. A interface do agente com o ambiente será formalizada por meio de funções externas. Na Seção 4.4, mostraremos que essas funções externas podem ser implementadas usando-se o próprio modelo ASM.

Na formalização a seguir, as percepções são definidas por meio de funções externas que retornam um valor *booleano*. Se o agente se movimentar em um passo da execução, no passo seguinte as funções `encontrou_parede`, `encontrou_cheiro` e `encontrou_brilho`, de significados óbvios, indicarão se uma percepção é verdadeira. Por exemplo, se caminhar para frente e trombar em uma parede, no passo seguinte teremos `encontrou_parede` = *true*.

A posição do agente é definida pelas funções `posx` e `posy`, que indicam as coordenadas do mesmo no espaço bidimensional. Vamos considerar que o agente não conhece, inicialmente, sua posição real no ambiente. Ele poderia supor que os valores iniciais de `posx` e `posy` são ambos iguais a zero e considerar seu deslocamento **relativo** a essa posição inicial. A mesma suposição pode ser feita em relação ao sentido de deslocamento do agente. A função `sentido` será interpretada como um dos elementos distintos `NORTE`, `LESTE`, `SUL` e `OESTE`, e poderá não corresponder ao real sentido de deslocamento.

Vamos especificar um agente com estado, que acumula conhecimento do ambiente à medida que se desloca. Para isso, vamos utilizar uma função `mapa` que associa a posição de cada célula (coordenadas x e y) a informações que representam o conhecimento do agente sobre o ambiente. Por exemplo, o valor `LIVRE` indica que a célula já foi visitada e é segura, pois não contém o monstro. Outras funções são utilizadas para armazenar o conhecimento do agente, denotando os limites do espaço bidimensional explorados até um momento, e indicando as coordenadas relativas das paredes, quando encontradas.

Para simplificar, vamos supor que o agente esteja em uma posição inicial não vizinha ao monstro. A regra da Figura 3(a) deve ser executada em um passo imediatamente após o agente caminhar para frente (`Walk`). Propositalmente, apresentamos uma especificação em um nível de abstração bem alto, contrastando com o código exibido na Figura 1. A técnica de refinamentos sucessivos é constantemente utilizada quando se usa o modelo ASM, para aumentar a clareza de uma especificação, como nesse exemplo da Figura 3(a). O texto deve ser interpretado da forma a seguir: se `encontrou_parede` = *false*, o movimento deve realmente ser realizado, pois não havia uma parede à frente. Nesse caso, o agente pode atualizar suas coordenadas de posição, de acordo com o sentido do movimento. Essa atualização é realizada pela *ação* `EXECUTAMOV`, que é detalhada na Figura 3(b). Uma ação é uma abstração, possivelmente parametrizada, de uma regra de transição [10]. Se uma parede foi encontrada, o movimento não ocorreu, mas o agente pode acumular conhecimento sobre a posição de uma parede. Essa acumulação de conhecimento é implementada pela ação `DEFPAREDES`, cujo código não é mostrado na Figura 3.

A regra da Figura 3(c) deve ser disparada em um passo seguinte à execução da regra da Figura 3(a). De acordo com uma possível nova posição, o conhecimento sobre os limites do espaço bidimensional podem ser ampliados, executando-se as regras associadas à ação `NOVOLIMITE`. Esse conhecimento é representado pelas funções `minX`, `maxX`, `minY`, `maxY`, que aparecem no detalhamento da ação `NOVOLIMITE`, na Figura 3(d).

Na Figura 3(c), pode-se ver que a célula da posição corrente é marcada como livre, na função `mapa`. Além de indicar que o monstro não ocupa essa célula, essa informação auxilia o processo de caminhamento exaustivo no espaço bidimensional. Se a célula da posição corrente contiver o pote de ouro, ele deve ser recolhido pelo agente. O possível encerramento da execução é verificado e executado na ação `Grab`. Se a célula corrente for vizinha à posição do monstro (`encontrou_cheiro` = *true*), todas as células vizinhas são marcadas com uma informação de "perigo", na ação `MARCAWUMPUS`. Caso contrário, todas as células vizinhas à célula corrente que contiverem "perigo" são marcadas com uma informação que indica "célula não visitada", na ação `ANULAWUMPUS`. Isso indica que o monstro não pode estar nessas células.

Por falta de espaço, não podemos mostrar todas as regras desenvolvidas, que incluem manobras para contornar a célula onde está o monstro e uma pesquisa exaustiva do espaço em busca do pote de ouro. Testes foram realizados adicionando-se regras para produzir animação, seguindo o esquema apresentado na Seção 3.

4.4 Ambiente do Mundo Wumpus em ASM

Como anunciamos anteriormente, as funções externas usadas na Seção 4.3 podem ser implementadas usando-se o próprio modelo ASM. Uma possibilidade é desenvolver a implementação usando um segundo agente, que mantém uma representação própria do estado do ambiente, inclusive com a real localização do primeiro agente, do monstro Wumpus e do pote de ouro. No texto a seguir, vamos chamar esse segundo agente de Agente2, e o agente da Seção 4.3, de Agente1. O Agente2 deve fornecer ao Agente1 os serviços necessários para sua navegação no ambiente, por meio de funções externas.

A álgebra do Agente2 não será a mesma utilizada pelo Agente1. Entretanto, a linguagem utilizada deverá prover algum mecanismo para que algumas funções definidas

```
if not encontrou_parede then
  EXECUTAMOV
else
  DEFPAREDES
endif
```

(a) Bloco de Regras 1.

```
EXECUTAMOV =
  if sentido = NORTE then
    posY := posY + 1
  elseif sentido = SUL then
    posY := posY - 1
  elseif sentido = LESTE then
    posX := posX + 1
  else // sentido = OESTE
    posX := posX - 1
  endif
```

(b) Ação **EXECUTAMOV**.

```
NOVOLIMITE(posX,posY),
mapa(posX,posY) := LIVRE,
if encontrou_brilho then
  Grab
elseif encontrou_cheiro then
  MARCAWUMPUS(posX,posY)
else
  ANULAWUMPUS(posX,posY)
endif
```

(c) Bloco de Regras 2.

```
NOVOLIMITE(x,y:Int) =
  minX := minimo(minX,x),
  maxX := maximo(maxX,x),
  minY := minimo(minY,y),
  maxY := maximo(maxY,y)
```

(d) Ação **NOVOLIMITE**.

Figura 3: Regras para Agente no Mundo Wumpus.

no Agente2 sejam interpretadas como funções externas no Agente1.

Para exemplificar a formalização do ambiente do Mundo Wumpus pelo Agente2, vamos analisar a ação `Walk`. A Figura 4 apresenta a regra associada a essa ação, que é utilizada no Agente1, mas implementada pelo Agente2. O trecho de código mostra como a posição real do Agente1 é alterada, juntamente com as percepções do ambiente, quando a ação `Walk` é disparada. A ação `MOVE` é detalhada na mesma figura e utiliza ainda outras ações, das quais apenas `verif_monstro` é exibida em detalhe. As funções `posXReal`, `posYReal` e `sentidoReal` são usadas com propósito similar ao do Agente1. Mas, nesse caso, elas representam a realidade do ambiente, e não uma percepção interna, como no Agente1. As funções `wposX` e `wposY` representam a posição (real) do Wumpus.

O fato de termos a implementação das ações externas do Agente1 dentro do Agente2 não acarreta problemas de sincronismo. A separação na implementação, nesse caso, visa apenas ressaltar as restrições que o Agente1 possui com relação às percepções do ambiente. Por exemplo, o Agente1 não sabe sua **real** posição no espaço bidimensional. Mas quando o Agente1 dispara uma ação como `Walk`, por exemplo, todas as atualizações dessa ação são disparadas **em paralelo** com as demais atualizações do Agente1.

5 Um Ambiente para Jogos de Inteligência Artificial

Um ambiente para desenvolvimento de jogos envolvendo agentes racionais poderia ter os seguintes componentes:

1. Uma linguagem para se descrever tanto as regras dos jogos, quanto o comportamento dos agentes que participam dos jogos.
2. Mecanismos para assegurar que os agentes irão obedecer as regras descritas.
3. Uma interface gráfica animada para exibir as transformações associadas aos jogos.

```
if sentidoReal = NORTE then MOVE (posXReal, posYReal+1)
elseif sentidoReal = LESTE then MOVE (posXReal+1, posYReal)
elseif sentidoReal = SUL then MOVE (posXReal, posYReal-1)
else /* sentidoReal = OESTE */ MOVE (posXReal-1, posYReal) endif
```

```
MOVE (x, y :  Int) =
  if verif_parede(x,y) then // encontrou parede na posição (x,y)
    encontrou_parede := true
  else
    encontrou_parede := false,
    encontrou_brilho := verif_brilho(x,y),
    encontrou_cheiro := verif_cheiro(x,y),
    if verif_monstro(x,y) then stop endif, // encontrou monstro
    posXReal := x, posYReal := y
  endif
```

```
verif_monstro (x, y :  Int) =
  return (x = wposX) and (y = wposY)
```

Figura 4: Regras para a ação `Walk`.

Na Seção 4.3, mostramos como o modelo ASM pode ser utilizado para formalizar o comportamento de um agente inteligente que navega no ambiente do Mundo Wumpus. As regras para descrever o próprio ambiente (regras do jogo) são exemplificadas na Seção 4.4. Esse conjunto satisfaz o Item 1 acima.

Para assegurar que os agentes irão seguir as regras estabelecidas, pode-se usar mecanismos de encapsulamento e controle de visibilidade. A linguagem Machĭna, implementação do modelo ASM que usamos em nossos testes, oferece recursos que muitas vezes satisfazem as exigências do Item 2. Um desses recursos é a utilização de *módulos* que podem definir objetos e serviços públicos e privados. Em uma formalização como a exibida na Seção 4.4, por exemplo, um módulo descrevendo as regras do jogo deveria manter uma representação interna do ambiente, não acessível externamente, e oferecer como serviços públicos as ações `Turn(Right)`, `Turn(Left)`, `Walk`, e `Grab`, e as percepções necessárias.

Na linguagem Machĭna, é possível especificar que a regra de transição associada a um agente tem *execução intercalada*[10]. Isso significa que, entre dois passos da execução dessa regra, pode acontecer a execução das regras de outros agentes. Voltando ao exemplo do Mundo Wumpus, uma execução intercalada seria interessante para que o Agente2 pudesse exibir o estado graficamente, a cada alteração promovida pela regra do Agente1. Infelizmente, para nossos propósitos, na linguagem Machĭna, a modalidade de execução intercalada é definida dentro do próprio módulo que especifica o comportamento de cada agente. O ideal seria que o módulo que especifica as regras do jogo pudesse definir que a execução do Agente1 seria obrigatoriamente intercalada.

Finalmente, para satisfazer o Item 3 descrito no início desta seção, podemos utilizar as facilidades descritas na Seção 3. Um ambiente com as características descritas nesta seção pode ser utilizado na especificação e teste de inúmeras situações envolvendo jogos com agentes inteligentes, sendo uma ferramenta muito interessante no estudo de especificação formal de algoritmos e técnicas de inteligência artifical.

6 Conclusões e Trabalhos Futuros

Neste artigo, procuramos mostrar a utilização do modelo ASM na formalização do comportamento de agentes inteligentes em aplicações de Inteligência Artificial, um domínio ainda não explorado por esse formalismo. Além do exemplo simplificado do *Mundo Wumpus*, iniciamos o desenvolvimento de outras aplicações, como jogos simples de tabuleiro, guerras de robôs e outras. Acreditamos que o uso do modelo será bem aceito, como uma alternativa a linguagens lógicas ou então com o uso associado a essas linguagens. Consideramos que uma vantagem é a noção clara de alterações de estado, bastante intuitiva para usuários de linguagens de programação de paradigma imperativo.

Com base nas idéias expostas neste artigo, estamos construindo uma ferramenta que utiliza o modelo ASM para desenvolvimento de jogos envolvendo agentes inteligentes. Nessa ferramenta, as regras de um jogo são especificadas usando-se o modelo ASM, como no exemplo da Seção 4.4. Em um ambiente de ensino de Inteligência Artificial, caberia aos alunos desenvolverem a especificação dos agentes inteligentes que iriam participar do jogo. Mecanismos oferecidos pela linguagem utilizada, que pode ser a linguagem Machĭna, irão garantir a obediência às regras pré-estabelecidas. Estamos estudando a proposição de modificações na linguagem Machĭna, para melhorar esses mecanismos.

Podemos enumerar três benefícios imediatos da utilização do modelo ASM e uma linguagem como Machĭna na descrição das regras de um jogo envolvendo agentes inteligentes. Primeiro, tem-se um ambiente onde é possível testar a execução do código de agentes. Segundo, não é necessário produzir um documento explicando as regras do jogo, uma vez que a semântica estará toda descrita em ASM. O usuário que for especificar o comportamento de um agente para participar do jogo deverá conhecer o modelo ASM e os mecanismos de encapsulamento de Machĭna, para entender as regras, as possíveis percepções e as ações que pode executar. O terceiro benefício é a facilidade para a construção de demostrações de propriedades sobre o comportamento dos agentes, usando indução finita, como demonstrado em [11] e [8].

Um efeito colateral interessante que pudemos observar, com a utilização de gráficos animados, foi uma maior facilidade para entendimento do próprio modelo ASM. O conceito de atualização de funções pode ser novidade para uma boa parte de estudantes, mas o uso de exemplos como o da Seção 2.2 parece ter facilitado o entendimento, uma vez que é possível visualizar a evolução do modelo.

References

[1] Clanlib, a multi-platform game development library. http://www.clanlib.org/intro.html.

[2] E. Börger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. In J. Bowen, M. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer, 1997.

[3] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[4] G. Del Castillo, I. Durdanović, and U. Glässer. An Evolving Algebra Abstract Machine. In H. K. Büning, editor, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95), volume 1092 of *LNCS*, pages 191–214. Springer, 1996.

[5] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[6] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[7] Y. Gurevich and J. Huggins. The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In *Proceedings of CSL'95 (Computer Science Logic)*, volume 1092 of *LNCS*, pages 266–290. Springer, 1996.

[8] V. Iorio. *Avaliação Parcial em Máquinas de Estados Abstratas (in portuguese)*. PhD thesis, Universidade Federal de Minas Gerais, 2001.

[9] S. Russel and P. Norvig. *Artifical Intelligence: A Modern Approach*. Prentice Hall, New Jersey, 1995.

[10] F. Tirelo, R. Bigonha, M. A. Maia, and V. Iorio. Machĭna: A Linguagem de Especificação de ASM (in portuguese). Technical Report 08/1999, Laboratório de Linguagens de Programação, Universidade Federal de Minas Gerais, 1999.

[11] F. Tirelo, R. Bigonha, M. A. Maia, and V. Iorio. Tutorial em Máquinas de Estado Abstratas. In *Anais do III Simpósio Brasileiro de Linguagens de Programação*, Porto Alegre, Maio 1999.