Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 130 (2005) 129–150
www.elsevier.com/locate/entcs

What's the Name of the Game? Formal Specification of Artificial Intelligence Games

Vladimir Di Iorio[1]

Departamento de Informática,
Universidade Federal de Viçosa,
Viçosa, Brazil

Roberto S. Bigonha[2], Mariza A. S. Bigonha[3]

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil

Alcione Oliveira[4], Eliseu Miguel[5]

Departamento de Informática,
Universidade Federal de Viçosa,
Viçosa, Brazil

Abstract

Artificial intelligence games are a very interesting tool for teaching Artificial Intelligence techniques. Competitors write programs for agents, which are supposed to complete a given task or fight against other agents. In order to achieve the best performance, programs may have to use advanced Artificial Intelligence methods. In this paper, we present a framework to build artificial intelligence games, using Abstract State Machines (ASM) for the specification of the rules of the games. Choosing ASM, we expect that the competitors will be able to understand clearly the semantics of the rules. The framework includes a compiler for an ASM-based language, allows complete control of the order of execution of agents and easy integration with graphical libraries.

Keywords: Artificial Intelligence, computer games, Abstract State Machines.

doi:10.1016/j.entcs.2005.03.008

1 Introduction

Game playing is one of the oldest areas of research in Artificial Intelligence. The first studied games were board games, like chess. In the past years, research evolved to cover multi-agent games with sophisticated agent interaction, including simulation of human behaviour.

According to Russel and Norvig [22], an *agent* is anything that can be viewed as perceiving its environment through *sensors* and acting upon that environment through *effectors*, or *actions*. In multi-agent artificial intelligence games, a competition is established among agents, which are usually restricted to a small number of actions. The agents are given a task to complete, or they fight against other agents in order to achieve the best performance, according to the rules of the game. In a more complex competition, agents with different behaviour may form teams to maximize their results [24]. When the behaviour of the agents is specified by computer programs, with few or no human interference, they are called *bots* [12]. Some games allow also an interaction among bots and human players.

The main goal of the competitors, in a multi-agent artificial intelligence game, is to write the best program for their bots. To achieve this goal, it is necessary to understand clearly the rules of the game, i. e. what actions the bots are allowed to execute and the consequences of the execution of these actions on the environment.

Abstract State Machines (ASM) [3,8] are an ideal formalism for giving a clear definition of a multi-agent artificial intelligence game. The semantics of an ASM specification is easy to understand and this method has been used for the specification of many distributed systems successfully [2,9].

In this paper, we present a framework for building artificial intelligence games, using ASM for the specification of the rules of the games. The framework includes a compiler from an ASM-based language called Machĭna [28] to C++. It should be interesting if games could be represented graphically, possibly with animation, so the framework provides for easy integration with graphical libraries. The compiled code is efficient enough to produce animation with reasonable speed.

[1] Email: vladimir@dpi.ufv.br
[2] Email: bigonha@dcc.ufmg.br
[3] Email: mariza@dcc.ufmg.br
[4] Email: alcione@dpi.ufv.br
[5] Email: emiguel@dpi.ufv.br

2 Related Work

There is a great number of available commercial games which use artificial intelligence techniques to specify the behaviour of bots [26]. Some of them have bots with a fixed behaviour, designed to fight against human players. Most first-person shooter (FPS) games are included in this category. FPS games are very popular 3D action games where the user moves through different scenes and collect weapons to destroy enemies. One example is *Doom II* [19]. We are interested in another kind of games: the ones in which it is possible for the users to program the behaviour of the bots. They are known as *games with extensible AI* [29].

Some modern FPS games have extensible AI. For example, *Half-Life* [18] offers a bot kit using C++ as the programming language. Some games, although classified as entertainment, have only programmable bots and no human player. One example is *AI Wars* [20], with a programming language mixing specialized commands with basic programming resources. The *Robocode* project [13] implements a robotic battletank. The behaviour of the bots is defined using the Java programming language.

Some frameworks provide visual tools in order to make the programming tasks easier, even for users with no programming languages background. *Stagecast* [6] and *Gamut* [14] are good examples of systems which help users build games and simulations without writing code.

The *Gamebots* project [1,11] has been designed for education and research in artificial intelligence. It has created a test-bed for multi-agent systems using an extension for the commercial *Unreal Tournament* game engine [7]. Unlike other extensible AI games, Gamebots does not define a single benchmark task. The wide variety of predefined tasks and environments can be extended in various ways, using a C++-based scripting language called *UnrealScript* [25]. Communication between the game server and bots are done via sockets, so bots can be programmed in different languages. Examples using Java and Soar [21] are available.

Our work is similar to Gamebots, in the sense that the proposed games have no predefined tasks. But our approach is much more general, since we can define any kind of game. The proposed games are not restricted to a predefined environment or style. In the Gamebots project, games can be extended and even new games created, using Unreal Script, but in a limited way. And in order to understand the rules of the proposed game, it is necessary to know the script language, which was not designed to be a good formal specification method. We believe that Abstract State Machines are a very elegant solution for this problem. The semantics of the rules are clear as long as they are

written as a well designed ASM specification.

3 Abstract State Machines

In this section, we present a brief introduction to the formalism of Abstract State Machines, concentrating in the sequential model. Agents and concurrent execution will be addressed later. A more formal and complete presentation can be obtained in [3,8,27].

Vocabulary, functions and states

The *vocabulary* or *signature* of a sequential ASM $\mathcal{A}$ is a finite collection of function names, each with a fixed arity. A *state* of $\mathcal{A}$ is a nonempty set called *superuniverse*, together with interpretations of the vocabulary names on functions over the superuniverse. These interpretations are designated *basic functions*. Different states have different interpretations for the vocabulary names, but the superuniverse always remains unchanged.

Formally, in a superuniverse X, a basic function with arity r is a $X^r \rightarrow X$ function. When $r = 0$, a function is called *distinct element*. The superuniverse always contains the distinct elements *true*, *false* and *undef*, defined as *logical constants*. The element *undef* is used for representing partial functions, for example, $f(\overline{a}) = undef$ means that function f is undefined for tuple $\overline{a}$. A *r-ary relation* over X can be represented by a $X^r \rightarrow \{true,false\}$ function. An *universe* U is a special basic function: an unary relation identified by the set of elements x such that $U(x) = true$, i.e., $\{x : U(x)\}$.

Programs

A *program* of $\mathcal{A}$ is a transition rule, specifying transformations over states, generating new states. A transition rule is composed by *basic* and *non-basic* rules. The *basic rules* are: *update rule*, *block constructor* and *conditional constructor*.

Update rule

An *update rule* is an expression $f(\overline{t}) := t_0$, where f is the name of a function on the vocabulary of $\mathcal{A}$, $\overline{t}$ is a tuple of terms whose length equals the arity of f and t_0 is another term. Terms have no free variables and are recursively built using names of distinct elements and application of function names to other terms. The semantics of the update rule is: the tuple $\overline{t}$ is evaluated and the value of the basic function f applied to the evaluation of $\overline{t}$ is updated with the evaluation of the term t_0. In other words, the name f receives a new interpretation.

Conditional constructor

A *conditional constructor* is an expression with the following format:

$$\textbf{if } g_0 \textbf{ then } R_0 \quad \textbf{elseif } g_1 \textbf{ then } R_1 \ \ldots \ \textbf{elseif } g_k \textbf{ then } R_k \quad \textbf{endif}$$

The semantics is: the rule R_i, $0 \leq i \leq k$, will be executed if the boolean terms $g_0, ..., g_{i-1}$ evaluate to *false* and g_i evaluates to *true*.

Block constructor

A *block constructor* is a set of rules:

$$R_0, \qquad R_1, \qquad \ldots, \qquad R_k$$

with the following semantics: rules R_0, R_1, ..., R_k are executed in parallel. If this execution produces inconsistent updates, an unpredictable result is generated.

Non-basic rules

Non-basic rules use bound variables. They increase the power of expression of the language allowing, for example, the introduction of non-determinism and the extension of universes, creating new elements. One example is the *var rule*:

$$\textbf{var } v \textbf{ ranges over } U \quad R_0 \quad \textbf{endvar}$$

where v is a variable, U is a finite universe and R_0 is a rule. An instance of rule R_0 is created for each element belonging to universe U, associating the variable v to each of these elements. Then, all rules are executed in parallel.

Runs

A *run* of a program of $\mathcal{A}$ is a sequence of states. Each state is generated by the execution of the transition rule over the previous state. If a run is not affected by the external environment, it ends when no update is produced by the execution of the transition rule. Most implementations of ASM define also a special command `stop`, which indicates an explicit end for a run.

External functions

In order to allow an interface with the external environment, *external functions* are defined in ASM. An external function may return different results, when called with the same arguments, in different steps of a run.

4 The Language Machĭna

Machĭna is a strongly typed ASM-based programming language, with special structures for modularity, visibility control and information hiding. In this section, we present only the main concepts of the language. Complete information about Machĭna is available in [28]. Examples are presented in sections 6 and 7.

4.1 Modules

The main syntactic structure of Machĭna programs is a *module*. A module contains a transition rule and declarations of types, *actions* and ASM functions. Only declarations qualified as *public* are visible outside the module.

To execute the transition rule of a module, it is necessary to dynamically create an agent based on this module. The exception is the *Main module*, for which an agent is automatically defined. When an agent executes the transition rule of a module, the function name `self` is interpreted as the current agent.

The first section of a module is the *import section*, where public names from other modules can be imported. Next, the *declaration section* defines *types*, *functions* and *actions*. Following the declarations, an *initial state section* may be defined, which is an ASM rule executed only once, before the execution of the transition rule of any module. Finally, the transition rule of the module can be defined. This rule is executed every time an agent associated with the module becomes active. Elements present in the declaration section of Machĭna modules are described bellow.

Types

New types can be created, using the predefined types and composition. Composed types are: lists, sets, tuples and agents. The type "`Agent`" is a generic agent and the type "`Agent of M`" defines an agent based on a module with name "`M`". A *functional* type is defined by the syntactic construct `(`T_1 `->` T_2`)`, where T_1 and T_2 are types.

Functions

Functions can be qualified as *static*, *dynamic*, *derived* and *external*. *Static functions* are ASM functions that cannot be updated, *dynamic functions* are ASM functions that can be updated, *derived functions* receive parameters and return values and *external functions* are defined outside the system, possibly written in another programming language.

Actions

Actions are abstractions for ASM transition rules, discussed in detail in Section 4.3. Actions can also be qualified as external, when defined outside the system.

4.2 Transition Rules

The basic ASM transition rules, *update*, *conditional* and *block rule*, are implemented in Machĭna with the usual ASM semantics. Other rules are available:

- ***choose***: Non-basic rule for non-deterministic choices.
- ***forall***: Equivalent to the *var rule*, described in Section 3.
- ***let***: For local definitions.
- ***stop***: Interrupts the execution of a program and kills all agents.
- ***create***: Used for the creation of agents. When an agent is created, this rule indicates the program code which will be executed, associating the agent with a Machĭna module.

4.3 Actions

Actions are an important Machĭna feature, implementing abstractions for transition rules. Actions may receive parameters and execute a transtition rule with bound variables, using values instantiated at execution time. All values in Machĭna are dynamically allocated, so when parameters are passed to actions in an *action call*, references are used, implementing a call-by-reference protocol. Examples of Machĭna actions are shown in Section 7.

The semantics of an action call is the following: all updates are collected, and then fired in parallel with other updates of the block where the action call is placed. In other words, updates produced by an action call in a state of a run only affect values in the next state of the same run.

There is also a special kind of action designated *loop action*. In this case, the transition rule of the action is repeatedly executed until a *return* command is executed. But a call to a loop action has the same semantics of a call to a simple, non-loop action: all updates are collected and fired in parallel, affecting only the next state of the run.

5 Multi-Agent Artificial Intelligence Games

In multi-agent artificial intelligence games, we can identify two kind of programs: programs designed by the creator of the game, defining the rules of

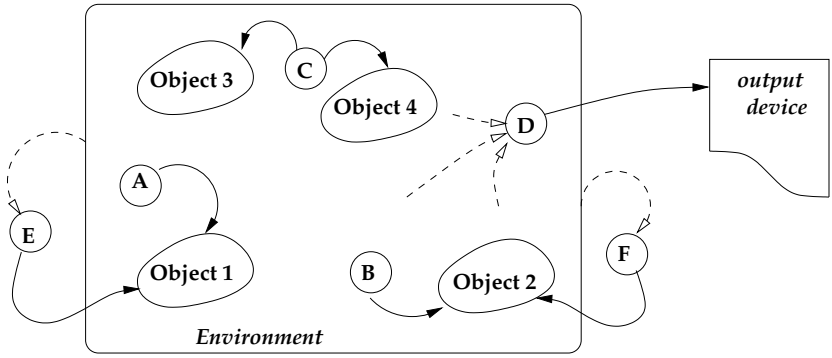

Fig. 1. Scheme of a multi-agent artificial intelligence game.

the game, and programs created by users, simulating the behaviour of competitors. In this section, we discuss these programs and other components of a multi-agent artificial intelligence game in detail. It is important to understand the relationship between these components before we present the framework we have built, in Section 6.

5.1 Main Components

In Figure 1, *environment* represents the current state of a game. The environment consists of a set of objects. The attributes of these objects may be concurrently updated by agents, denoted in Figure 1 by A, B, C, D, E and F. Dashed arrows represent agents reading information about the environment, and solid arrows represent agents updating environment information.

Some agents are designed by the creator of the game, defining the rules of the proposed game. We will call them *internal agents*, represented in Figure 1 by A, B, C and D. Figure 1 shows also user-controlled agents, as the ones represented by E and F, which we will call *user agents*. User agents may receive information from the environment and try to act upon the objects.

Internal agents are always controlled by computer programs. User agents, on the other hand, may also be controlled by humans. But in this work, we will suppose that all user agents in an artificial intelligence game are controlled by computer programs. When the behaviour of agents is specified by computer programs, working intelligently without depending upon any human interaction, they may be called *bots* [12]. So the agents of Figure 1 will be classified as *internal bots* (the programs defining the rules of the game) and *user bots* (which are user-defined programs, representing the players in a multi-agent artificial intelligence game).

If graphical representation is necessary, it may be interesting to have an extra internal bot designed specifically to produce visual information. In Fig-

ure 1, it is represented by D, which reads the current state of the environment and displays it in a suitable way, in an output device.

Two or more bots may be running the same program code. In this case, they will present the same behaviour. A bot may update several objects, and an object may be updated concurrently by several bots. When there is just one internal bot in the system, objects have a centralized control. In Robocode [13], for example, the environment is controlled by a single agent, responsible also for producing visual representation. When objects are updated by different internal bots, the rules of the game are defined by a distributed program. This multi-agent definition for the environment is used in the framework presented in Section 6, and represents a more general approach than the one adopted in systems like Robocode. No assumption is made for the relative execution speed of internal and user bot programs.

5.2 An Example: the Snake Game

To better illustrate the concepts presented in Section 5.1, we now show a simple example that uses all the main components of a general multi-agent artificial intelligence game. The example involves a game designated as the *snake game*, which is also used in the following sections.

In the snake game, several snakes move in a two-dimensional space, trying to reach an object designated as the *vitamin*. A snake is composed by a special first cell called the *head*, possibly followed by several other cells, known as the *body*. If there is at least one cell in the body, the last cell of the sequence is the *tail*. The movement is controlled by the head, and the other cells follow their respective previous neighbour in the sequence. When the head of a snake reachs the vitamin, we say that this snake "eats" the vitamin. In this case, the body of that snake grows one cell, and the vitamin appears somewhere else in the space. The goal is to make a snake longer, until reaching a predefined length. We have added some other rules in order to make the competition more interesting. We will present these rules in Section 7.

Using the snake game in an artificial intelligence context, we may have all the components shown in Figure 1, as described below.

Environment:

The environment is represented by the position of the vitamin and the position of the cells of each snake. In addition, we may have an attribute associated with each snake, representing the direction of its movement (*north*, *south*, *east* or *west*).

Graphical Representation:

A special bot may read information from the environment and define a visual representation. Figure 2 shows an example of a simple graphical representation for the snake game. A snake head is represented by an image that indicates its direction of movement. Animation is generated by the sequence of state changes produced by the execution of the bot programs.

User Bots:

User bots may control the snake actions indirectly, changing their direction of movement. If two snakes have the same behaviour, they may be controlled by user bots that run the same program code. But each snake may be controlled by a single user bot.

Internal Bots:

Internal bots change the position of the snakes, according to the current direction of movement. For example, if the direction of movement of a snake is *north*, the associated internal bot will execute transformations in the environment in order to move the snake head one position to north, and move all other cells to the previous position of their neighbour in the sequence. There is exactly one internal bot associated with each snake. We say that *an internal bot A is controlled by an user bot B* when B controls the snake to which A is associated. We may suppose that all snakes follow the same rules for movement, so all internal bots run the same program code.

Among other tasks, programs written for internal bots must have code for:

- generating movement for a snake, according to the current direction of its head;
- detecting collision of a snake's head with the vitamin;
- increasing a snake's length when a collision with the vitamin is detected.

Notice that these items represent exactly the rules for the snake game. So, understanding the program code built for internal agents is crucial for understanding exactly the rules of the game.

6 A Framework for Multi-Agent Artificial Intelligence Games

The framework we have developed is composed by a compiler from the ASM-based language Machĭna to C++, together with facilities to control the concurrent execution of the agents and to produce graphical representation for

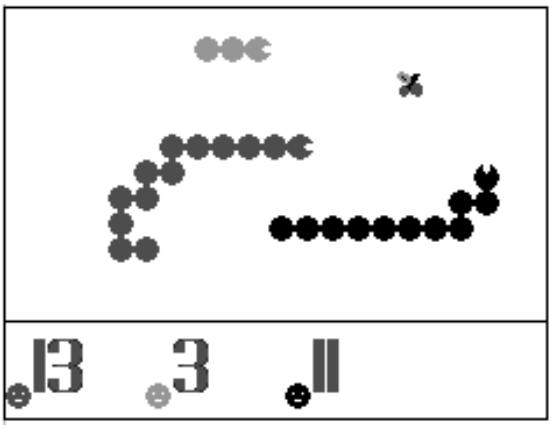

Fig. 2. Snapshot of the snake game.

games. The framework provides an implementation for all the components discussed in Section 5.

6.1 The Specification Language

We have chosen Machĭna as the specification language for our framework. The use of an ASM-based language for the specification of the rules of the games provides a precise definition for them. If the specifications are carefully designed, the rules will be easily understood. Another advantage of using Machĭna is that the specifications can be executed, simulating animation at reasonable speed.

Important features present in Machĭna are: suitable data structures for the description of the environment in artificial intelligence games, possibility of specification of distributed programs, control of visibility of data and actions, efficient compilation to C++. The use of these features in the specification of multi-agent artificial intelligence games is discussed below.

Environment Specification:

When defining an artificial intelligence game, the first task is the specification of the *environment*. The environment is a collection of attributes that describe the current state of a game. Using an ASM-based language, these attributes may be represented by functions, defining an ASM state. Machĭna offers suitable data structures to represent these functions.

Programs executed by bots:

The possible transformations that the environment may suffer, together with the environment specification, define the rules of an artificial intelligence game. These transformations are carried out by internal and user bots. Distributed ASM programs may provide an implementation for bot programs, which execute concurrently. These programs can be defined using Machĭna *modules*, and executed by Machĭna *agents*.

Visibility Control:

User bots may affect the game environment in a restricted way, following the rules of the game. The communication between user bots and the game environment can be implemented via ASM functions. With Machĭna, it is also possible to define this communication with *actions*, abstractions for ASM rules. Not all functions representing the environment state and not all defined actions may be visible for user bots. In Machĭna, it is possbile to define restrictions on the visibility of data structures and actions. This visibility control is important in preventing "cheating", i. e. competitors are supposed to follow strictly the rules of the game.

Graphical Representation:

Visual representation of a game may be provided by an extra agent that reads information from the environment, as discussed in Section 5. This agent may have access to a graphical library via ASM *external functions*. Machĭna has also *external actions*, abstractions for ASM transition rules which may be defined elsewhere in the system, even using another programming language. We have developed examples written in C++, using the open source multiplatform game development library *Clanlib* [17].

Using ASM and Machĭna, the rules of an artificial intelligence game are represented by the environment specification and by distributed programs which will be executed by bots. We hope that, reading these specifications, a competitor is able to understand clearly the rules of the game and is ready to write programs which will compete in the proposed environment. ASM allows the creation of specifications at a natural level of abstraction, and this quality is inherited by Machĭna, which has the advantage of an efficient compilation to C++.

6.2 Controlling the Execution of Agents

In principle, no assumption is made for the relative execution speed of Machĭna agents. It means that an agent can execute several moves, before any other

agent has the chance to proceed. But sometimes it is interesting to define a more restrictive order of execution. For example, consider the snake game of Section 5.2. A special internal agent produces a visual representation for the environment. Every time an internal bot executes a move on a snake, a visual change should be produced. In order to create good animation, the agent that provides visual representation should be executed immediately after the execution of any other internal bot of the system.

Our framework has added to Machĭna the concept of *active agents*, similar to that presented in [8] for Distributed Algebras, using the additional unary relation name `machActive`. Only agents satisfying this relation are *active* and can make moves. But in our framework, every time an agent executes its transition rule, it also automatically executes the following update: `machActive(self) := false`. It means that the agent becomes idle until the framework changes its state to active again. There is also an additional relation `machAllAgents`, which is *true* for any agent, either active or not.

As presented in Section 4.1, Machĭna defines a special module named *Main module*, for which an agent is automatically defined. In our framework, this agent is designated the *Main agent*, and it is always active. The *Main agent* is allowed to create other agents and update the function `machActive`, building any desired policy for the execution of all other agents in the system.

Figures 3 and 4 show an example of a *Main module* that implements the policy discussed in the beginning of Section 6.2. The dynamic functions involved are:

- `curAg`: Represents the current active agent.
- `nextAgents`: Set from which an agent is picked up to be activated.
- `exib`: Represents an agent that produces a visual representation for the game.
- `userBots`: The set of user bots.

In this case, the set `machAllAgents` is formed by agent `exib`, together with the union of set `userBots` and the set of internal agents which produce movement for the snakes.

Suppose that a module named `Exibition` defines rules that produce a visual representation for a game. In the *initial state* section of the *Main module* of Figure 3, an agent that executes transition rules defined by module `Exibition` is created. This agent is then associated with the function name `exib`. Other agents may be created in this section, representing the internal and user bots of the game, as shown in Section 7.4.

In Figure 4, the transition rules received line numbers in order to make the following explanation easier. Lines 1-3 initialize the set `nextAgents` with

```
dynamic functions:

    curAg, exib : Agent;
    nextAgents : set of Agent;

 initial state:

    curAg := undef,
    nextAgents := { },  /* empty set */
    create x : Agent of Exibition do
        exib := x
    end
```

Fig. 3. Declarations of functions and initial state in a *Main module*.

all internal bots that move the snakes (all agents of the system, except `exib` and the user bots). Lines 4-9 choose, non-deterministically, an agent from the `nextAgents` set. This agent is activated, removed from the `nextAgents` set and defined as the current active agent. Lines 11-12 will only be executed after `curAg` has executed its transition rule, when it is automatically deactivated. Then, agent `exib` is activated to show the possible changes produced by `curAg`. Line 14 will only be executed after `exib` has executed its transition rule. A new agent is selected from the `nextAgents` set, in lines 4-9, until this set becomes empty. Then the process starts again in lines 1-3 with the set of all bots, except agent `exib`. Rules of lines 16-18 assures that all user bots will always be active, so the transition rules of the user bots may be executed at any time.

The semantics of the transition rules of Figure 4 assures that visual information is provided immediately after any agent moves. Besides this, it implements a policy which is free from *starvation*.

The execution order of agents may be programmed directly in the code of the internal bots. But separating the rules for the scheduling policy from the rules of the game can make the specification more clear. It can also make proofs of properties related to the scheduling policy easier.

Our framework has the advantage that even the order of execution of agents is defined by a specification written in ASM.

7 The Snake Game Implemented in Machĭna

In this section, we show part of an implementation of the snake game in Machĭna. The first task is to develop an abstract data type for the representation of snakes. Using the operations of this abstract data type, we build

```
transition rule:

1   if nextAgents = { } then
2       nextAgents := machAllAgents - { exib } - userBots,
3       curAg := undef
4   elseif curAg = undef then
5       choose a in nextAgents do
6           machActive(a) := true,
7           nextAgents(a) := false,
8           curAg := a
9       end
10  elseif not machActive(curAg) and curAg != exib then
11      curAg := exib,
12      machActive(exib) := true
13  elseif not machActive(curAg) and curAg = exib then
14      curAg := undef
15  end,
16  forall a in userBots do
17      machActive(a) := true
18  end
```

Fig. 4. Transition rules for a *Main module*.

the rules of game, defining the code executed by internal bots, which produce movements for the snakes. Competitors may write user bots to control the direction of movement of the snakes. Finally, a *Main module* creates instances of the user bots.

7.1 Representing Snakes

Figure 5 shows part of the module `SnakeData`, that implements an abstract data type for the representation of snakes. Some basic structures, like `Pos2D` (position in a 2-dimensional space) and `Direction` (direction of movement) are imported from other modules.

The public type `SnakeDescriptor` is defined as a 3-tuple. A snake is represented by its length, the position of each cell and the current direction of movement. The position of each cell is given by a function that associates the sequential number of the cells (the head is numbered as "1") with positions in the 2-dimensional space.

Several public *derived functions* and *actions* implement operations on the abstract data type. The symbol "`:=`" (usually used as function update) is

```
module SnakeData
import:
    Position (Pos2D, Direction, ...);
type
    public SnakeDescriptor is (
        length : Int;
        posCell : Int -> Pos2D;
        direction : Direction
    )

derived:
    public getLength (s : SnakeDescriptor) : Int :=
        s.length
    public buildSnake (p : Pos2D; d : Direction)
    : SnakeDescriptor :=
        SnakeDescriptor (1, {1 -> p}, d)

actions:
    moveToPosition (s : SnakeDescriptor; p : Pos2D) :=
        s.posCell(1) := p,
        forall i in 2 ..  getLength(s) do
            s.posCell(i) := s.posCell(i-1)  /* in parallel */
        end
```

Fig. 5. Module `SnakeData` : abstract data type representing a snake.

used here for the definition of the body of derived functions and actions. For example, given a snake descriptor, function `getLength` returns its length. Suppose that `p` is a position neighbour to the head of a snake described by a `SnakeDescriptor s`. Then action `moveToPosition` changes the position of the snake head to `p`, moving also the cells of the snake body. Notice that call-by-reference parameter passing is important for this action to work properly.

The name of a named tuple may also be used in Machĭna as a *constructor*. An example is shown by the public derived function `buildSnake`, which creates a new snake descriptor with just one cell, given an initial position and direction of movement.

7.2 The Rules of the Game

In figures 6 and 7, part of the code of module `SnakeMove` is presented. It implements the rules that define the movements for snakes. These are in fact the

rules of the game.

As previously discussed in Section 5.2, the snake game involves two kind of agents. Internal bots produce movement for snakes, according to the current direction of movement, so they are agents based on module `SnakeMove`. User bots, on the other hand, are agents based on modules written by competitors. According to the rules informally presented in Section 5.2, each internal bot has a single user bot that controls it. So, in order to prevent "cheating", the framework must provide resources to ensure that each `SnakeMove` agent will be controlled by a single user agent.

In Figure 6, the game environment is represented by the dynamic functions `vitaminPos`, the vitamin position, `snakeControl`, which maps an user bot to the `SnakeMove` agent that it controls, and `snakeInfo`, which maps a `SnakeMove` agent to the descriptor of the snake that it moves. An example of use of `snakeInfo` is shown by the derived function `getLength`. Given a `SnakeMove` agent, this function returns the length of the associated snake, using a derived function with the same name from module `SnakeData` (see also Figure 5).

The movement of all snakes follow the same rules, so it is natural that agents which move snakes execute the same code. Then, all internal bots run the same program code. Each internal bot access the attributes of its associated snake using the function name `self`, which has a different interpretation when the code is executed by different agents. In module `SnakeMove`, `self` is interpreted as the current `SnakeMove` agent in execution. Examples are shown in the transition rule of Figure 7, where line numbers are added in order to make the following explanation easier.

If a snake achieves a predefined length, it is considered the winner of the game. Line 1 of the transition rule executes the Machĭna rule `stop`, which kills all agents, if the current agent has achieved the desired length. Otherwise, `p` is calculated as the new position of the snake head, according to the currrent direction of movement. Lines 5-6 are executed if the snake eats the vitamin. In this case, action calls make the snake grow one cell and make a new position to the vitamin be chosen. If `p` is a free cell, an action call in line 8 makes the snake move to that position (observe the definition of the action `moveToPosition` in figures 6 and 5). Lines 10-13 are executed when the snake head eats another snakes tail. In this case, the current snake grows one cell and the other loses one cell. If none of the above conditions are true, it means that `p` is the position of a snake cell, but not a tail. In this case, the action call in line 15 is executed and the snake loses one cell. The rules of the game are much more sophisticated than the ones presented in Section 5.2, but, using Machĭna, they are very clear and easy to understand. They define when a snake can move, grow and shrink.

```
module SnakeMove
dynamic
    vitaminPos :  Pos2D;
    snakeControl : Agent -> Agent of SnakeMove;
    snakeInfo : Agent of SnakeMove -> SnakeDescriptor;

derived
    public getLength (s : Agent of SnakeMove) : Int :=
        SnakeData.getLength (snakeInfo(s))

actions:
    moveToPosition (s : SnakeAgent; p : Pos2D) :=
        SnakeData.moveToPosition (snakeInfo(s), p)
    login (a : Agent; x, y : Int; d : Direction) :=
        userBots(a) := true,
        create s : Agent of SnakeMove do
            snakeControl(a) := s,
            snakeInfo(s) := buildSnake (buildPos2D(x,y), d)
        end
    public changeDirection (d : Direction) :=
        SnakeData.changeDirection (
         snakeInfo(SnakeControl(self)), d)
```

Fig. 6. Declarations for the rules of the snake game (Module `SnakeMove`).

7.3 User Bots

User bots are based on modules written by competitors. The only public action from module `SnakeMove` they can execute is `changeDirection` (in Figure 6). It is executed by user bots in order to control the movement of a snake.

An user bot has access to its own agent using function `self`, but it has no access to other agents. The framework gives the permission of this kind of access only to the *Main module*. This restriction ensures that an user bot can only affect the direction of movement of the internal bot associated to it by function `snakeControl`.

In order to win the game, a competitor may write code that moves a snake toward the position of the vitamin. It may also try to eat the tail of other snakes. The code may be written in Machĭna or C++. We do not show an example here because of lack of space, and because our main goal is the specification of the rules of games. The complete code of all modules, including examples of user bots and visual animation using the graphical library *Clanlib*

```
transition rule:
1   if getLength(self) = target then stop
2   else let p = newPosition (
3   getHead(self), getDirection(self)) in
4       if p = vitaminPos then
5           growToPosition(self,p),
6           defNewVitaminPos()
7       elseif isFreeOfSnakes(p) then
8           moveToPosition(self,p)
9       elseif isSnakeTail(p) and (getTail(self) != p) then
10          growToPosition(self,p),
11          forall s in setOfSnakes do
12              if getTail(s) = p then shrink(s) end
13          end
14      else
15          shrink(self)
16      end
17  end
```

Fig. 7. Transition rules of the snake game (Module `SnakeMove`).

[17] can be found in [5].

7.4 Creating Instances of User Bots

The *Main module* creates instances of the user bots. Association to internal bots is performed by action `login`, defined in module `SnakeMove`, in Figure 6. This action is not public, so it cannot be executed by another module, except the *Main module*.

The action `login` receives an user bot as a parameter. It creates a new agent, an internal bot, to move a snake. The new internal bot is associated to the user bot using function `snakeControl`. A descriptor is created to store information about the new snake. This descriptor is associated to the new internal bot using function `snakeInfo`. Using action `login` is the only way new snakes are inserted into the game.

For example, suppose that a competitor writes a module named `Player1`. The code below creates an user bot and its associated snake, with initial position (10,30) and initial direction of movement to north. This code should be placed in the specification of the initial state of the *Main module* (see

Figure 3):

```
create x : Agent of Player1 do
    login (x, 10, 30, NORTH)
end
```

8 Conclusions and Future Work

Artificial intelligence games are usually defined using logic languages like Prolog. Examples of simple games created with Prolog can be found in [23]. A much more complex Prolog game is described in [15,16]. But problems with efficiency of generated code prevent logic languages of being used on games with visual animation.

Some systems like the *Gamebots* project [1,11] allow the definition of different scenarios for sophisticated 3D games, but the designer must be an expert on programming on the *Unreal Script* [25] language. And there is a worse problem: in order to understand the rules of a proposed game, either the competitors are also fluent in *Unreal Script*, or they must rely on a textual, non-precise description of the rules.

In this work, we have shown that ASM and Machĭna are a good alternative for the definition of artificial intelligence games. The advantages are:

- ASM are a precise formal specification method. If a specification is carefully designed, competitors can understand clearly the rules of the proposed game.
- Machĭna provides an efficient implementation for ASM specifications. Generated code is efficient enough to produce animation at a reasonable speed. Besides this, Machĭna includes features for visibility control of data and actions, and complete control of the order of execution of agents. These features are important for preventing "cheating" by the competitors.

Other languages based on ASM are also available. Perhaps the most important is *ASML* [10]. For the purpose of this work, the main disadvantage of ASML is that the language does not implement yet distributed Abstract State Machines. Multi-agent systems can still be defined using ASML, but the specifications are not so elegant and clear as they can be with Machĭna.

The framewok presented in Section 6 can be used for the definition of any distributed system with animated visual representation. In this paper, we have concentrated on artificial intelligence games, but simulation of other distributed systems can be carried out without difficulty.

The framework is being prepared to be used in teaching AI for undergraduate students. Interesting, animated multi-agent games can be specified. A

competition will be established among students, who are supposed to write programs for bots representing the competitors in the proposed games. Advanced artificial intelligence techniques may be used in order to produce the best programs. We expect that this competition will make students feel also motivated for learning formal specification methods like ASM.

Our future plans include the definition of more sophisticated games, using also 3D animation. A current project is the implementation of a classic artificial intelligence problem known as *Wumpus World* [22], whose ASM rules were first presented in [4].

References

[1] R. Adobbati, A. N. Marshall, A. Scholer, S. Tejada, G. Kaminka, S. Schaffer, and C. Sollitto. Gamebots: A 3d virtual world test-bed for multi-agent research. In *Proceedings of the International Conference on Autonomous Agents (Agents-2001) - Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001.

[2] E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[3] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[4] Vladimir O. Di Iorio, Alcione P. Oliveira, Eliseu C. Miguel, Roberto S. Bigonha, and Mariza A. S. Bigonha. Utilização de Máquinas de Estado Abstratas em Aplicações de Inteligência Artificial e Jogos. In *Proceedings of XXIX CLEI - Conferencia Latino Americana de Informatica*, La Paz, 2003.

[5] Vladimir Oliveira Di Iorio. Abstract State Machines and Artificial Intelligence Games. On-line documentation, 2004. (retrieved 2 July, 2004, from http://www.dpi.ufv.br/~vladimir/asm/asmANDgames.htm).

[6] P. Fleisher. Stagecast - software review. Technology & Learning Magazine, 2003.

[7] J. Gerstmann. Unreal Tournament : Action game of the year. Gamespot, 1999. (retrieved 8 July, 2004, from www.gamespot.com/features/1999/p3_01a.html).

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

[9] 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.

[10] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic Essence of Asml. Technical Report MSR-TR-2004-27, Microsoft Research, March 2004.

[11] G. A. Kaminka, M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshal, A. S. Scholer, and S. Tejada. Gamebots: the ever-challenging multi-agent research test-bed. *Communications of the ACM*, January 2002.

[12] A. Leonard. *Bots: The Origin of New Species*. Hardwired, 1997.

[13] S. Li. Robocode: Advanced Robot Battle Simulation Engine. IBM developerWorks, 2002. (retrieved 19 July, 2004, from http://www-106.ibm.com/developerworks/java/library/j-robocode/index.html).

[14] Richard G. McDaniel and Brad A. Myers. Building Applications Using Only Demonstration. In *Proceedings of IUI'98: 1998 International Conference On Intelligent User Interfaces*, pages 109–116. ACM Press, January 1998.

[15] Dennis Merritt. *Adventure in Prolog.* Springer Verlag, 1990.

[16] Dennis Merritt. Exploring Prolog: Adventures, Objects, Animals, and Taxes. *PC AI Magazine*, 7.5, September/October 1993.

[17] M. Norddahl and Kenneth Gangstoe. Clanlib, a multi-platform game development library., 2004. (retrieved 7 July, 2004, from http://www.clanlib.org/intro.html).

[18] J. Ocampo. Half-Life - game review. Gamespot, 2004. (retrieved 22 November, 2004, from http://www.gamespot.com/pc/action/halflife2/review.html).

[19] F. Provo. Doom II - game review. Gamespot, 2002. (retrieved 21 July, 2004, from http://www.gamespot.com/gba/action/doom2/review.html).

[20] J. Reder. AI Wars (the insect mind), 2004. (retrieved 22 July, 2004, from http://www.tacticalneuronics.com).

[21] P. Rosenbloom, J. Laird, and A. Newell. *The Soar Papers.* MIT Press, 1993.

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

[23] Patrick Saint-Dizier. *An Introduction to Programming in Prolog.* Springer Verlag, 1990.

[24] N. Schurr, S. Okamoto, R. Maheswaran, P. Scerri, and M. Tambe. *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, chapter "Evolution of a Teamwork Model". Cambridge University Press, 2004.

[25] T. Sweeney and A. Moise. UnrealScript Language Reference. Epic MegaGames, Inc., 1998. (retrieved 9 July, 2004, from http://unreal.epicgames.com/UnrealScript.htm).

[26] P. Sweetser. Current AI in Games: A Review, 2002. (retrieved 15 July, 2004, from http://www.itee.uq.edu.au/~penny/Game%20AI%20Review.pdf).

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

[28] Fbio Tirelo, Roberto S. Bigonha, Marcelo A. Maia, and Vladimir O. Di Iorio. Machĩna: An ASM-based Specification Language (in portuguese). Technical Report 08/1999, Laboratório de Linguagens de Programação, Universidade Federal de Minas Gerais, 1999.

[29] S. Woodcock. Games with Extensible AI, 2004. (retrieved 15 July, 2004, from http://www.gameai.com/exaigames.html).