Operational Semantics of a Timely Bounded Agent Abstract Machine

Abdelkader Gouaich

An operational semantics of a timely bounded agent abstract machine

Abdelkader GOUAICH
gouaich@lirmm.fr

Michael BERGERET
bergeret@lirmm.fr

12/10/2009

Abstract

This paper presents a domain specific programming language dedicated to timely bounded agents. Timely bounded agents are expected to deliver the most appropriate response to a stimuli and this response has to be delivered at the right time, otherwise it is ignored. This modifies assumptions and action models used in most of current agent programming languages and frameworks. We propose a domain specific language by specifying an agent abstract machine and its operational semantics. The dynamics of the agent abstract machine is a sequence of perception-deliberation-influence cycles where an agent: (i) gets its perceptions (that remain unchanged during the cycle) (ii) evaluates its behaviors to generate a set of influences; (iii) commits all influences at once within its environment. An important property is the fact that perceptions are unchanged and side effects are not allowed until the end of a cycle. This property makes possible efficient implementation of the agent abstract machine using parallel evaluations. Simplicity and expressiveness are important features when working on operation semantics. To meet these requirements we have proposed a simple and yet expressive framework inspired from REST (Representation State Transfer) to consider agent actions as manipulation of resources within environments using only a limited set of primitives. This makes both the agent abstract machine and its operational semantics easy to comprehend and the implementation straightforward. Finally, we demonstrate how our proposal has helped building a serious game for upper limb rehabilitation.

1 Introduction

Many efforts have been recently made within multi-agent systems (MAS) community to facilitate building large-scale distributed software systems using agents as building blocks. In fact, starting from foundational works in the 80's MAS

have been considered as an appropriate framework when both the control and data are inherently distributed. Despite this potential, MAS are not considered yet as a mainstream paradigm for software engineering. Our objective is to specify an agent-oriented programming language to facilitate building software systems with the following properties:

- Expressiveness and pragmatism: current works on agent oriented programming are mostly based on logic. These works offer solid theoretical basis to develop theories about agents and reason about their actions. However, the dominant culture of programming is imperative which makes adopting logic based agent-programming languages difficult. It would be interesting to have a hybrid approach using logic to express states of affair and let the control expressed imperatively.

- Target real-time systems: When the environment is dynamic agents are timely bounded to react at a proper time. Few works have tackled this issue and deliberation process of agents are assumed to be unbounded. Our objective is to introduce explicitly the time dimension to program agent based real-time systems.

The rest of the paper is organised as follows: section 2 presents the Agent-REST framework used to unify agent concepts with REST; section 3presents the formal model of an agent abstract machine; section 4 presents the implementation of the agent virtual machine (VM); section 5 presents a simple test case and a benchmark; section 6presents related works; and finally section 7concludes the paper and presents some perspectives.

2 Agent-Rest Framework

REST (REpresentational State Transfer) is an architectural style introduced by T. Fielding [6] to facilitate building networked applications. The central concept of REST is *resource*: a resource is identified using a uniform resource identifiers (uri) [22] and owns an internal state. Access to a resource is performed using HTTP operations. The semantics of these operations is expressed as follow:

- 'put' and 'delete' are used to respectively store and delete resources

- 'post' modifies the state of a resource

- 'get' is used to retreive a representation of the resource's state

REST has gained popularity and is becoming a *de facto* architectural style of Interned based applications. This is explained by the flexibility, scalability and efficiency offered by REST architectures, which are consequences of encapsulation of states within resources rather than in proceses. Thus, it is possible to use coherently multiple stateless threads to handle incoming requests. The other advantage of REST is its simplicity and expressiveness. From a software
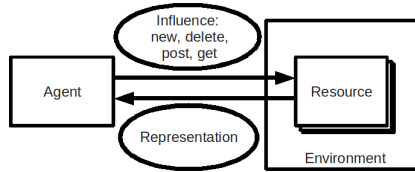
Figure 1: The Agent-REST framework

engineering point of view, simplicity is very useful to diminish complexity and consequently ease maintenance of large scale systems.

Within MAS some architectural styles have been also proposed for building software systems [3, 10]. However, most of these frameworks are centered on the concept of agent or organisation and have neglected persistency and data management.

We present Agent-REST framework (Figure 1) that combines both REST and MAS concepts as follow:

- Environment: an environment represents a container of resources and agents. The environment handles agents' commands (or influences) to either modify resources' state or read their representation using respectively 'post' and 'get' commands

- Resource: a resource encapsulates an internal state and is identified by a uri

- Agent: an agent is an autonomous process that reads resource representations as inputs and submit influences on the environment

- Influence: the term 'influence' is preferred to 'command' because an agent does not modify directly a resource's state. Thus, a resource state modification is always an attempt that has to be validated by the environment.

- Resource representation: as said previously, a resource is an abstraction and agents have access only to their representations. Resource representation are taken as perceptions wihtin the agent's deliberation process.

3 Agent abstract machine

3.1 Modeling influences and representations

Influences and resource representations are expressed as syntaxic terms using the following definitions.

Definition 3.1 *Let N be a countable set of names and V a coutable set of values.*

We build a set of influence terms $S_{Influence}$ as a syntaxic construction over N and V as follows (with $n \in N$, $v \in V$):

$$
\begin{array}{rcl}
S_{Influence} & ::= & \textbf{\textit{get}} \; n \\
& & \textbf{\textit{post}} \; n \, v \\
& & \textbf{\textit{new}} \; n \\
& & \textbf{\textit{delete}} \; n \\
& & \textbf{\textit{0}}
\end{array}
$$

Terms for resource representation is also built syntaxically over N and V as follows (with $n \in N$, $v \in V$):

$$
\begin{array}{rcl}
S_{Representation} & ::= & \textbf{\textit{repr}} \; n \, v \\
& & \textbf{\textit{0}}
\end{array}
$$

3.2 Behavior structure

Current works on agent-oriented operational semantics often describe two levels of abstraction at the same time: (i) an agent level that describes how an agent abstract machine evolves over time and a behavior level describing the decisional process. This makes operational semantics difficult to comprehend as different concerns are addressed at the same time. Our proposal differentiates among these levels and this article is focused on the agent level operational semantics. Still, we need to provide a structure to reify agents' behaviors and exhibit their properties.

Definition 3.2 *Let C be a countable set of* capabilities. *The set of behaviours generated from C is defined by the power set $\mathcal{P}(C)$ equipped with set union as composition law. As a notation, we represent the structure $(\mathcal{P}(C), \cup)$ as $(C, +)$*

Lemma 3.1 *The structure $(C, +)$ is a commutative monoid with $+$ being idempotent.*

Example In the rest of the paper examples are inspired from the educational agent-programming language developed by [21]. Let suppose a turtle agent with the following capabilities:

$$
C = \{\uparrow, \downarrow, \rightarrow, \leftarrow, \text{penup}, \text{pendown}, \text{canMove}\}
$$

These capabilities are interpreted as follows: a turtle is able to move in four directions (up, down, right and left); put down a pen to start drawing, take off the pen to stop drawing, and finally check if the movement is allowed (for instance by checking if the turtle has enough energy)

From this set we are able to create behaviors using the composition law. For instance, $(\rightarrow + \text{pendown}) \in (C, +)$ expresses the willing to turn right and *simultaneously* putting down the pen.

The composition law $+$ is commutative and idempotent. This means that duplication and order of tasks are not important. More formally, this means that $t + t = t$ and $u + v = v + u$ This is useful to express behaviors that involve simultaneously several capabilities.

Definition 3.3 *We denote by $(C, +, .)$ the freely generated monoid having as generator the set $(T, +)$; . (dot) is the formal composition law.*

Example By introducing the . (dot) law, we are able to express a second level of composition that is not commutative. This could be useful for instance to express guards. For instance, one can construct formally the following element: canMove.($\rightarrow$ +pendown) $\in (T, +, .)$ to express the fact that 'canMove' must succeed before turning right and starting the draw mode.

3.3 Capabilities substitution in behaviors

Modularity along with re-usability are crucial in software engineering. These principles are about constructing a complex system as sub-components that can be easily reused, modified and assembled.

The presented behavior structure captures naturally the idea of modularity. In fact, any element can be composed with another using either $+$ or . (dot) laws. Concerning re-usability, we introduce a first level of customization through *capabilities substitution*.

Definition 3.4 *A capabilities substitution from set X to Y,$\sigma : X \longrightarrow Y$ is a function $\sigma : (X, +, .) \to (Y, +, .)$ mapping an element of X with an element of Y where all capabilities, c, are replaced by their image $\sigma(c)$*

X_σ denotes the set of generated behaviors from X with substitution σ

Definition 3.5 *We define the category of behaviors $\mathbf{Cat}(beh)$ as follows:*

- *Objects are behavior structures $(C, +, .)$*

- *Morphisms are capabilities substitutions:$f : (X, +, .) \to (Y, +, .)$*

- *Identity morphism is the identity function*

- *Morphism composition if simply function composition: $f.g = f \circ g$*

Capabilities substitution are introduced to express an elementary level of behavior customization by changing some capabilities with others. Although theoretically this is simply expressed as term substitution, in practice this is an interesting feature to construct specialization relationship among behaviors. In fact, starting from an elementary behavior, another behavior can be derived by changing capabilities. This is similar to overriding methods in object-oriented programming.

Example In this example a mirror turtle is built from an existing one. This is simply performed by providing the following substitution:

$$\sigma_1(\rightarrow) = \leftarrow; \sigma_1(\leftarrow) = \rightarrow; \sigma_1(\downarrow) = \uparrow; \sigma_1(\downarrow) = \uparrow \; ;$$

For instance, when σ_1 is applied to a behavior this gives:

$$\begin{aligned} x &= \text{canMove.}(\rightarrow +\text{pendown}) \\ \sigma_1(x) &= \text{canMove.}(\leftarrow +\text{pendown}) \end{aligned} \qquad (1)$$

Whenever the base turtle wants to go in a direction, the new turtle goes in the opposite direction.

Now suppose that we want to define a safer implementation of canMove capability:

$\sigma_2(\text{canMove}) = \text{SaferCanMove}$

The fact that capabilities substitution builds a category [2] **Cat**(beh) simply says that substitutions are composable. In fact, one can build a mirror turtle with the new version of canMove capability by taking $\sigma_1 \circ \sigma_2$ as substitution. So the example presented in (1) becomes:

$\sigma_1 \circ \sigma_2(x) = \text{SafercanMove.}(\leftarrow +\text{pendown})$

3.4 Taskgram structure

To be evaluated, a behavior needs input representing agent's perceptions. To represent this unit of evaluation comprising a behavior and its associated set of perceptions we introduce the concept of *taskgram*.

Definition 3.6 *A taskgram is defined as a couple (t, i) element of $(B, +, .) \times \mathcal{P}(S_{\text{Perception}})$ where t represents a behavior and i is its associated input data.*

3.4.1 Taskgram evaluation

As an execution unit a taskgram is evaluated *as a whole* using an evaluation device to produce a set of influences:

Definition 3.7 *An evaluation device M is considered as a black-box capable of producing a mapping between a taskgram and a set of influences. Formally, this device is modelled as a partial function:*

$$\begin{aligned} M : (B, +, .) \times \mathcal{P}(S_{\text{Perception}}) &\rightarrow \mathcal{P}(S_{\text{Influence}}) \\ (t, i) &\mapsto M(t, i) \end{aligned}$$

with the following properties:

$$M(\{l\}, i) \quad = \quad eval_M(l, i) \tag{2}$$

$$M(\emptyset, i) \quad = \quad \emptyset \tag{3}$$

$$M(u + v, i) \quad = \quad M(u, i) \cup M(v, i) \tag{4}$$

$$M(u.v, i) \quad = \quad \begin{cases} M(u, i) \cup M(v, i) & \text{iff } M(u, i) \neq \emptyset \\ \emptyset & \text{otherwise.} \end{cases} \tag{5}$$

Lemma 3.2 *M is a monoid homomorphism from* $(B, +)$ *to* $\mathcal{P}(S_{\text{Influence}})$

Equation (2) indicates that any singleton containing a capability label is directly evaluated by the device; (4) indicates that the evaluation process is linear on + law. (3) and (4) are necessary to prove lemma 3.2. (5) defines an evaluation of behaviors of the form $u.v$: whenever the evaluation of the first operand is an empty-set then the overall expression is evaluated as an empty-set. This models a conditional evaluation of task v according to u.

3.4.2 Timely bounded evaluation

In the last section we have defined a theoretical evaluation device that is able to map any taskgram to a set of influences. However this raises the classical decidability problem: given a set of inputs, does the evaluation of a taskgram terminate?

To solve this problem, we suggest using evaluation devices with a timeout. Whenever the timeout is exceeded, the evaluation device returns an empty-set as result.

Definition 3.8 *A* timed evaluation device *is an evaluation device* M_δ *with a timeout* $\delta \in \mathbb{N}$ *such that:*

$$M_\delta(t, i) \quad = \quad \begin{cases} M(t, i) & \text{, when evaluation time} \leq \delta \\ \emptyset & \text{, otherwise.} \end{cases}$$

Within real-time systems, agents have to take the correct decision at the right time. This is a consequence of to the dynamic nature of their surrounding environment [18]. Consequently, any deliberation process has to be timely bounded. Whenever the timeout is exceeded the deliberation process is halted and an empty-set is returned as result.

This does not mean that lengthy deliberation processes cannot be defined. In fact, several mechanisms such as continuation style passing can be used to halt the deliberation process, store the current state before exceeding the timeout, restore the stored state to resume the deliberation process when a new cycle is started.

3.5 Fluents

Fluents have been introduced by situation and fluent calculus to represent facts about situations [8]. We use a simpler notion of fluents to express conditions that trigger behaviors. Conditions are expressed on resource representations using selectors. A selector is a pattern used to match some uri and select the value of an attribute. For conciseness, the full syntax of fluents is not presented; only notations illustrate how they are express and evaluated.

Definition 3.9 Fluents *are expressed through syntactic constructs as follows:*

$$
\begin{array}{llll}
P & ::= & exp & \qquad exp \quad ::= \quad uri\_selector \\
  & | & \rho_i \underbrace{(P, ...)}_{i\ elements} & \qquad \qquad \quad | \quad \Omega_i \underbrace{(exp, ...)}_{i\ elements}
\end{array}
$$

With ρ_i being usual logic operators: $\{\top, \bot, \neg, \wedge, \vee\}$ *and Ω_i standard comparison operators such as* $=, \neq, \leq, ...$

Example Let us consider a turtle with the following perception set:

uri	attributes
grid://turtles/turtle#10	(color,red)
agent://self	(color,blue);(mode,peace)

The following fluent checks that the observed turtle is not an enemy:

$$
(grid://turtles/\{x\}[color] \ != \ agent://self[color])
$$

$$
\wedge (agent://self[mode]=war)
$$

The current evaluation of this fluent is false. However, when a context making this fluent true arrives a special behavior (flee for instance) can be trigged.

3.6 Environment

The presented framework follows ideas developed in [9] to expose environments explicitly as a first-class entity within a multi-agent system. From an agent perspective, an environment is considered as a container of resources and interaction among agents holds through common resources. In fact, to exchange data agents have to perform simply post and get operations on common resources shared through uri. This unifies several interaction mechanisms such as asynchronous messages and generative communication [7]

In the theoretical model, an environment is considered as a black-box with two primitives to: (i) send an influence to a resource, (ii) and get a resource representation. These operations are noted respectively as follow: $E(r)$ and $E \leftarrow p$ where r represents an influence of type 'get' and p a request of type 'post'.
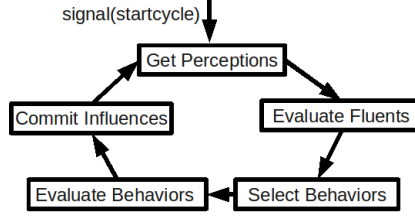
Figure 2: Agent abstract machine phases

3.7 Agent Abstract Machine

3.7.1 Agent structure

An agent program is given by describing a set of behaviours; a set of fluents and a set of triggers that relate fluents to behaviours.

Definition 3.10 *Let C be a set of capabilities; an C-*agent *program is given by a 3-uplets (B, F, G) where:*

- $B : \{b_i \in (C, +, .)\}$ *is a set containing behaviors*

- $F : \{f_i \in F\}$ *is a set containing fluents*

- $G \subseteq F \times B$ *is a trigger relation between fluents and behaviours*

3.7.2 Agent abstract machine configuration

The operational semantics methodology suggests to specify formally all states or configurations of the abstract machine. The dynamics of this machines is then given as evolution rules linking configurations.

Definition 3.11 *A configuration of the agent abstract machine is given by the 5-uplet (s, b, i, o, n) where:*

- $s \in \mathbb{Z}/5\mathbb{Z}$ *represents the state of the machine; 5 stages are needed for the complete evaluation cycle.*

- $b \subseteq B$ *represents the set of active behaviors*

- $i \in \mathcal{P}(S_{\text{Perception}})$ *represents the set of current perceptions*

- $o \in \mathcal{P}(S_{\text{Influence}})$ *represents the set of current influences*

- $n \in \mathbb{N}$ *represents the cycle number.*

3.7.3   Agent abstract machine dynamics

The agent abstract machine dynamics is characterized by five stages :

1. Get perceptions phase: this is the starting point of each cycle where agent's perceptions are gathered by retrieving requested resource representations

2. Evaluate fluents phase: during this phase all fluents are evaluated to determine their truth values

3. Select behaviors phase: the last phase has calculated the truth values of fluents which permit to select triggered behaviors

4. Evaluate behaviors phase: all triggered behaviors along with perceptions are used to construct taskgram that are submitted to the evaluation device

5. Commit influences phase: the evaluation of taskgrams produces a set of influences that are submitted to the environment. This ends the current cycle and a new cycle is started when a heartbeat signal is received.

These phases are described more formally by evolution rules described hereafter. E and M_δ represents respectively an environment and an evaluation device.

get perceptions rule

$$\frac{i' = E(\pi_{get}(o))}{(0, b, i, o, n) \rightarrow (1, b, i', o, n)} \tag{6}$$

This rule states that a new perception set, i', is calculated by selecting only influences of type 'get' from the set of influences using the projection function π_{get} These influences are submitted to the environment that replies with representations of resources.

fluents evaluation rule

$$\frac{b' = \{t \in B \,|\, (f, t) \in G \,\wedge\, Hold(f)\}}{(1, b, i, o, n) \rightarrow (2, b', i, o, n)} \tag{7}$$

This rule constructs a new set, b', of behaviors that have been triggered by holding fluents.

behaviors evaluation rule

$$\frac{o' = \bigcup_{x \in b} M_\delta(x, i)}{(2, b, i, o, n) \rightarrow (3, b, i, o', n)} \tag{8}$$

Behaviors are used to construct takgrams using perceptions as an input. These taskgrams are submitted to the evaluation device to calculate influences. It is worth noting that side effects are not allowed until all influences are returned from the evaluation device. This is an interesting proterty that will be used to enable parallel evaluations of behaviors using a multi-threaded evaluation device.

submit influences rule

$$\frac{E \leftarrow \pi_{\mathrm{post,new,delete}}(o)}{(3,b,i,o,n) \rightarrow (4,b,i,o,n+1)} \tag{9}$$

This rules submit influences of type 'post', 'new' and 'delete' to the environment. Since influences are submitted to the environment the agent's cycle number is increased.

restart cycle rule

$$\frac{\mathrm{signal(heartbeat)}}{(4,b,i,o,n) \rightarrow (0,b,i,o,n)} \tag{10}$$

This rule is used to restart a new cycle for an agent when receiving a heartbeat signal.

3.8 From agents to multi-agents system

A multi-agent systems is an aggregation of individual agents that collectively behave as a whole and create system level functionality. However this informal definition of a MAS does not specify the semantics of the aggregation. Thanks to the algebraic structures of our agent framework we are able to suggest two semantics for the aggregation of agents: (i) holonic semantics that considers a MAS as a bigger agent, and (ii) reductionist semantics that differentiates between the system level and the sub-system composed of autonomous agents.

3.8.1 Holonic MAS

Definition 3.12 *Let $A_i : (B_i, F_i, G_i)$ be a family set of agent structures; An* holonic multi-agent system *built on A_i is an agent defined as follows:*

$$\mathcal{H}_{A_i} = (\bigcup B_i, \bigcup F_i, \bigcup G_i)$$

The system $\mathcal{H}_{A_i}$ is still an agent built by making a union of all behaviours, fluents, and triggers.

3.8.2 Reductionistic MAS

Definition 3.13 *Let $A_i : (B_i, F_i, G_i)$ be a family set of agent structures; An* reductionistic multi-agent system $\mathcal{R}_{A_i}$ *is defined by three set (B,F,G) representing* disjoint unions *of agents:*

$$\mathcal{R}_{A_i} = (\oplus B_i, \oplus F_i, \oplus G_i)$$

Since, the union is disjoint we can identify each part of the union. We define projection function $\pi_i(\mathcal{R})$ to isolates the i-th agent from the system $\mathcal{R}$.
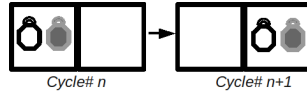
Figure 3: Example of a holonic MAS composed of two turtle agents (while and grey). The evolution of such as system is atomic from cycle n to $n+1$.
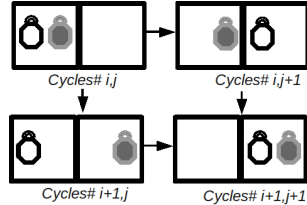

Figure 4: Example of a reductionistic MAS composed of two turtle agents. Each agent has its cycle number and the evolution of the system is reduced to the evolution of its agents.

3.8.3 Dynamics of holonic and reductionistic MAS

As a holistic MAS can be considered as a single agent which dynamics follows what have been described in §3.7.3.

The dynamics of the a reductionistic MAS R is *reduced* to the dynamics of its individual agents. Each agent $\pi_i(\mathcal{R})$ is considered independently and evaluated as described in §3.7.3.

Consequently, agents of a holistic system are always synchronised on their cycle number. By contrast agents of a reductionistic MAS evolve independently and each agent owns its own cycle number.

Example Let us consider two turtle agents a_1, a_2 with a simple behavior: at each cycle they move to the right cell.

When building a holonic MAS $\mathcal{H}_{\{a_1,a_2\}}$ the system behaves a single agent. Figure 3 illustrates for instance the dynamics of this system by moving both turtles in a single atomic cycle.

By contrast, a reductionistic MAS $\mathcal{R}_{\{a_1,a_2\}}$ keeps information about each agent and the dynamics of the system is defined by the proper dynamics of each agent. Figure 4 illustrates this situation by showing all possible evolutions to reach the final situation where both turtles have moved to the right cell.

4 Implementation of the abstract machine

We have followed principles developped in [1] for the development of the DSL by distinguishing between different phases (front-end and back-end) and establishing an intermediary representation (IR). The agent IR specifies an agent program as a set of behaviours, fluents and triggers. Different front-ends can be
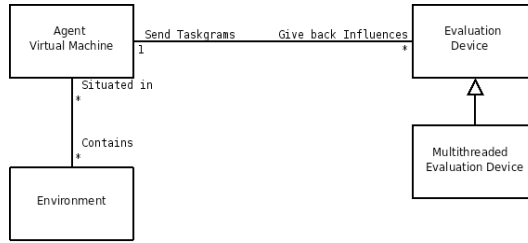
Figure 5: Architecture of the MAS interpreter

used, such as graphical programming, to generate agent IRs that are interpreted by the agent virtual machine. The architecture of the MAS interpreter is described in Figure 5. Environments are organized as a hierarchy and responsible for: (i) updating resource states when getting new, delete and post influences and (ii) retrieving resource representations in case of a get influence. The agent virtual machine implements the abstract agent machine as described in §3.7.3. The evaluation device is responsible for computing taskgrams to produce influences. The current implementaion of MAS interpreter takes advantage from taskgram properties: In fact, as demonstrated by functional programming [17], computations that are bounded to a static data set and that avoid side effects are efficiently parallelized. Taskgrams exhibit such property so a multi-threaded evaluation device has been used to enhance performances.

5 Test case: Agent Serious Game

To experiment with the presented framework an application has been developed whithin the context of a health oriented serious game project for upper limb reeducation (the name and reference to the project have been removed to comply with the blind review process) Within this project, patients play movement based games to improve their gesture. Developed games are required to be intelligent enough to adjust gameplay to patient's skills. Movements are captured using motion capture devices and the game system must react at real-time. To develop such flexible serious games, timely bounded agents are used to control all game elements (sprites and non playing characters) Although several game scenarios are investigated, we will present one specific game based on prey-predator principle. The scenario of this game is as follows: the patient controls a predator and wants to catch several moving preys controlled by software agents. Obviously, the game has to pay attention to player's frustration by adjusting the difficulty of catching preys. This is performed by changing prey agents' cycle duration. The more this cycle is long, the slower prey agents will react decreasing by the way the difficulty of the game. When the player starts succeeding in catching preys the cycle duration is decreased to make agents more reactive until reaching a threshold when the player is no longer able to make catches. Again the agents' cycle duration is adjusted since the game has

13

exceeded patient's skills. The rest of subsection shows how this game has been modelled and implemened.

5.1 Resources

Three types of resources are defined for the game:

Resource	Description	URI
Arena	This is a 2D grid composed of cells. Each cell has a position (x,y) and can hold an agent	/arena/cell
Prey	Represents a prey agent. Attributes are: location, velocity, angle and cyle.	/prey
Predator	Represents mouse pointer location. Attributes are: location.	/predator

5.2 Programming prey agents

5.2.1 Capabilities:

- move_safe : according to the near position of the ball, move the object

- move_unsafe : according to the very near position of the ball, accelerate and move the object

- restore : no predator in sight, don't move and restore from being tired

- do_nothing

5.2.2 Fluents:

- predatornear : the predator is near

- predatorverynear : the predator is very close in the catch range

- wallnear : a wall is in the move range of th object

- tired : the prey agent is tired, and his velocity isn't in his maximal power

- adrenaline : adrenaline cooldown is complete. This is used to make brief acceleration.

5.2.3 Triggers

$ballnear \land \neg wallnear \land \neg ballverynear \implies movesafe$

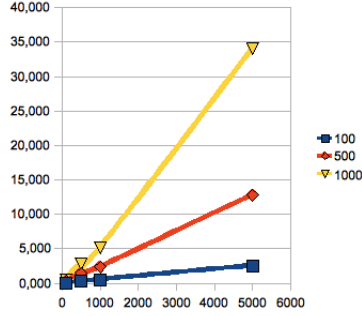$ballverynear \land \neg wallnear \land adrenaline \implies moveunsafe$

#agents\\#iteration	100	500	1000
100	0,097 s	0,311 s	0,549 s
500	0,272 s	1,491 s	2,817 s
1000	0,565 s	2,407 s	5,179 s
5000	2,565 s	12,828 s	34,022 s

Figure 6: Prey-Predator benchmark results (Y duration, X number of agents)

#agents\\#iteration	100	500	1000
100	0,00097	0,00062	0,00055
500	0,00272	0,00298	0,00282
1000	0,00565	0,00481	0,00518
5000	0,02565	0,02566	0,03402

Table 1: Stability of the ratio time/number of iterations

$ballverynear \land \lnot wallnear \land \lnot adrenaline \implies movesafe$

$\lnot ballnear \land tired \implies restore$

$\lnot ballnear \land \lnot tired \implies donothing$

5.2.4 Implementation of the game

This case study was implemented and tested using the stackless python. Stackless Python is an enhanced version of the Python programming language. It allows programmers to reap the benefits of thread-based programming without the performance and complexity problems associated with conventional threads. The microthreads that Stackless adds to Python are a cheap and lightweight convenience. Some usefull fonctionnality are also provided like channel for communication and task scheduling. For this scenario, we consider the stackless scheduling device as a multi-threaded evaluation device.

5.3 Benchmarks

The game described in the last section was also considered as a benchmark to test performances of the agent VM. In this case, the palyer is replaced by a random movement of the predator. Results[1] of benchmark are presented in Figure 6. They show that despite the large number of agents that have been used, the implementation gives interesting performances: the duration time progresses linearly with the increase of agents. This is a pretty behaviour that demonstrates that the implementation scales with the number of agents. This is due to the use of parallel evaluations and stackless Python library to evaluate taskgrams. Table 1 shows another interesting result: one can observe a stability of the ratio execution time/number of iteration for a given number of agents. This indicates that the dynamics of MAS has reached a steady regime in which the time to compute a complete iteration making all agents deliberating is stable. This can be explained by decomposition of the activities as taskgrams and the use of efficient and stable multithreading library.

6 Related works

Since Y. Shoham [19] has introduced agent-oriented programming, many works have contributed in creating an agent style of thinking, designing and programming software systems. As mentioned in [4] works on agent programming fits in the classical decomposition of programming languages being either declarative or imperative. Declarative approaches are affiliated to the AI heritage of knowledge representation and reasoning. In this area we could find languages based upon Prolog and BDI, like AgentSpeak [15] or 3APL [11], which are great to deduce knowledge from knowledge, by using predifined rules. However, these approaches often suffer from practices of software engineering and complexity of real world applications that cannot be captured at once as a set of first-order logical expressions. For imperative style one can mention works like Jack or CLAIMS [5] to model mobile agents. These works belong the process algebra tradition initiated by R. Milner in the beginning of the 80's. This approach is purely syntactic and all properties have to be given as rewriting rules. As consequence, the operational semantics is a big collection of rules that are not easy to comprehend. To avoid this, we have adopted an algebraic semantics approach that uses algebraic properties of structures. This makes the structure of the abstract machine more readable and its dynamics easy to express and comprehend. Peter Novak has proposed the Behaviour State Machine [14] framework as a way to manage knowledge representation by using an external KR module. To some extent, we follow the same idea by considering a more general approach using REST framework to encode knowledge within resources.

Management of realtime agents has not been widely addressed. In fact, works on situation culculus [13], event calculus [16], IMPACT [20] and Temporal Agent Reasoning [12], consider time as an additional dimension of a logical problem but are not bounded to it. For instance, the time consumed by the agent to make

---

[1] hardware and software configuration: Intel(R) Core(TM)2 Duo CPU E7300 @ 2.66GHz running under Linux Ubuntu 9.04

inferences using Prolog is not taken into accound. This makes these approaches very useful for making analysis about specification of realtime systems but not suitable to actually run realtime systems since they cannot be bound to wall clock.

Our model defines formally an operational semantics to run timely bounded agents by defining a timely bounded evaluation of taskgrams. By saying that we can introduce a paradigm shift in building agent-based real-time systems. In fact, since computation time is finite, we will always face the problem of bounded evaluation of taskgrams. Consequently what matters in an agent based real-time systems is not only defining intelligent behaviors but a strategy to adopt to execute some behaviors while delaying or cancelling others. In fact, the most complex and intelligent behavior could be useless if it get delayed by other activities and consequently fails in meeting its time boundaries.

7   Conclusion

The framework presented in this paper defines a simple but yet powerful way to program agent oriented application. Our proposition uses the REST architecture for interaction between agents, and also with the environment. That guarantees many important properties in AOP programming, like agent autonomy because all information passing from an entity to another is considered as representations of a resource. The dynamics of the abstract machine expresses a well specifies agent's cycle, with clear transition from a step to another. The evaluation device ensures computation of influences within a timely-bounded period. The framework has been implemented and tested in a real test case, and the benchmarking shows some interesting and promising results. Future works will consider using the proposed framework to design multi-agent system in two application domains. The first one is the serious gaming to define real-time games used in reeducation of upper limb. The second domain is embedded system, where the use of this framework will be efficient because of the consideration of the time. An interesting extension of the presented framework is about introducing special events such to notify urgency and cancelation of behaviors during a cycle. In fact, we have considered that an agent is completely isolated from the external world during the evaluation of a cycle of perception-deliberation-influence. It could be interesting to consider some exceptions to handle urgency and cancellation events. Execution of multiple parallel cycles is an interesting approach to investigate.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2 edition, August 2006.

[2] M. Barr and C. Wells. *Category Theory for Computing Science*. Centre de recherches mathématiques, Montreal, Canada, third edition, August 1999.

[3] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[4] R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni. *Multi-Agent Programming: Languages, Tools and Applications*. Springer Publishing Company, Incorporated, 2009.

[5] A. E. Fallah-Seghrouchni and A. Suna. Claim and sympa: A programming environment for intelligent and mobile agents. In *Multi-Agent Programming*, pages 95–122. 2005.

[6] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000.

[7] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[8] G. D. Giacomo and H. J. Levesque. An incremental interpreter for high-level programs with sensing. In H. J. Levesque and F. Pirri, editors, *Logical Foundation for cognitive agents: contributions in honor of Ray Reiter*, pages 86–102. Springer, Berlin, 1999.

[9] A. Gouaïch and F. Michel. Towards a unified view of the environment(s) within multi-agent systems. *Informatica International Journal*, 29(4):423–432, 2005.

[10] O. Gutknecht and J. Ferber. The madkit agent platform architecture. In *Revised Papers from the International Workshop on Infrastructure for Multi-Agent Systems*, pages 48–55, London, UK, 2001. Springer-Verlag.

[11] K. V. Hindriks and F. S. D. Boer. Agent programming in 3apl. *AAMAS Journal*, 2:357–401, 1999.

[12] S. K. Jurgen Dix and V. Subrahmanian. Temporal agent reasoning. 127(1):87-135, 2001.

[13] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90.

[14] P. Novák. Tbehavioural state machines: Programming modular agents. *Papers from the AAAI Spring Symposium, Technical Report SS-08-02: Architectures for Intelligent Theory-Based Agents, AITA'08,AAAI Press*, 2008.

[15] A. S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. pages 42–55. Springer-Verlag, 1996.

[16] M. S. R.Kowalski. A logic-based calculus of events. 6795, 1986.

[17] P. Roe. *Parallel Programming using Functional Languages.* PhD thesis, Glasgow University, 1991.

[18] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[19] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[20] V. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems: Theory and Implementation.* MIT Press, 2000.

[21] S. Tisue and U. Wilensky. Netlogo: A simple environment for modeling complexity. 2004.

[22] W3C. Rfc 1630, universal resource indentifiers in www. Technical report, W3C, 1994.