INTERACTION-BASED APPROACH FOR GAME AGENTS

Damien Devigne
Philippe Mathieu
Jean-Christophe Routier*
LIFL - CNRS UMR8022
Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq Cédex - FRANCE

KEYWORDS
Cognitive agents, Agent model, Believable behaviour, Simulation design

ABSTRACT

Most often, agents in simulations are based on reactive models. Such systems do not plan actions for the agents and are too limited to express complex realistic behaviour. We propose here an agent model for spatially situated simulations, like computer games are. The involved agents are cognitive (or deliberative) ones: they are able to build plans and to adapt them according to the dynamics of the simulation. Our main goal is to obtain believable behaviours for the agents in simulations.

Thus we propose a generic model for cognitive situated agent in simulations of which video games are a typical example. The proposed ideas promote re-usability from one simulations to another and then favour good software design.

The two main problems can easily (and without surprise) be identified: *how to represent knowledge ?* and *how to build plan using this knowledge*. To solve the first we propose to describe the laws that manage the simulated world in term of interactions that can be performed by some agents and suffered by others. Concerning the second problem, we propose a planning algorithm that is based on the interactions and take into account the facts that agents are situated and that plans must be executed in a situated environment that is in permanent evolution. Thus the plans are actually incrementally built through partial replanning.

INTRODUCTION

The simulation of rational or believable behaviour is, quite from the beginning of Computer Science, one of the major objectives of this field, especially in the purpose of Artificial Intelligence, ie. to reproduce the intellectual abilities of the human being. This issue has been initially addressed from a logical and linguistic viewpoint, which raises huge difficulties. In addition, it appeared rapidly that a large number of AI *applications* did not require a human-like intelligence level.

Now this is not still true. New research fields need a human-like level AI, not in order to *solve complex problems*, but rather to *develop harmonious interactions* with human partners: for instance, social robotics (Brooks and al. 1999), the use of virtual reality in teaching or training, or the large domain of video games (Nareyek 2004, Magerko, Laird, Assanie, Kerfoot, and Stokes 2004) are illustrative examples.

According to J. Laird, the latter constitutes a "Killer Application" for human-level AI (Laird and van Lent 2000). The characters involved in video games have indeed to be perceived as autonomous entities with increasing realistic behaviours. They have to be *convincing*, thus their behaviour must comply with the rational expectations of their partner or opponent human players. They also need to adapt to new situations, acquire additional abilities throughout the game, etc. In addition, team strategies are also often useful. In order to develop such kind of interactions, the agents have to make the human observer thinks that, in order to achieve their goals, they behave in an "intelligent", "rational" way, ie. like the human would have behaved. Our research aims at this goal : modelling believable characters for simulations in general and games in particular. Let us precise at this point that we do not consider here the problem of the simulation of "emotions" (Allbeck and Badler 2003), but consider "simulation" in the sense of "simulation of sequence of actions".

In the case of video games, theses "cognitive" constraints meet additional "economical" ones: the time needed for developing the game. This depends to a large extent on the reusability of previous works. In the case of character's AI, it is often difficult to reuse from one game to another or even, inside a game, from one character to another. This is mainly due to the almost systematic use of scripts whose drawbacks have been many times underlined (Tozour 2002), and that are only partially solved with dynamic scripting (Spronck, Sprinkhuizen-Kuyper, and Postma 2004).

More generally, this domain of modelling believable characters combines difficulties that can be encountered

*This work is supported by the CPER TAC of the *région Nord-Pas de Calais* and the FEDER europeand fund.

in classical AI (knowledge representation), in distributed AI (coordination of agents having most of the time different individual goals), and in Software Engineering (reusability of conceptual and software tools).

Some propositions have been done concerning agents and games (Nareyek 2000), and most of them concern reactive agents (Niederberger and Gross 2003). But reactive agents, while effective in several cases, offer limited behaviours. Indeed their behaviours are "short term directed" and not "goal oriented". Their ability to perform some tasks depends on the immediate surroundings and does not result of wilful acts. Our proposition aims at offering cognitive (or deliberative), driven by goals, proactive agents. Let us precise that we do not consider the interesting problem of behaviour's learning (Ponsen and Spronck 2004).

We promote a generic approach that assumes that a single formalism can be used to design realistic (ie. believable) behaviours in an artificial world in general and in games in particular. Thus from one simulation to another the cognitive behavioural engine stays the same even if the context changes and the behavioural components can be (partially) reused. The main principle is to base the dynamics of the simulations (and the knowledge representation too) on interactions between agents: some agents can perform interactions and other agents can suffer them. Our main goal is then to provide a uniform and generic frame for simulations. Our target is multi-agent spatially situated simulations like most of computer games are. More precisely role-playing games are a privileged target for our work. Agents are situated in an environment provided by an euclidean space. This space has a "geography" (a "map") and notions like "position", "neighbourhood", "move", "distance",... have a meaning. The agents have, at each moment, a partial perception of their environment. This environment is dynamic and concurrent, and thus non monotonic (insofar once an agent knows some data, this knowledge can become wrong - or irrelevant - after some times). The agent's knowledge about the environment is incomplete and can be wrong. The abilities of the agents can differ. Agents may have cognitive abilities. Some of them have objectives (or goals) that direct their actions in the environment. To achieve its goals each cognitive agent has a behavioural engine. This engine chooses at every moment an action to do. This action must allow the agent to fulfil its goals "at best" and rationally. The "rational" notion of a behaviour is rather subjective and is actually evaluated by a jury that is external to the simulation. Thus, we will consider as rational a behaviour if the decision to perform an action could have reasonably been taken by a human which would have had the same information than the agent.

First section concerns knowledge representation. We first present the environment that models the geography of the simulated world, second the agent model is described. These two points are not sufficient, we must precise how these agents can have an influence on the environment, that is, what the laws that rule the world are. This knowledge representation is crucial since it is used by the agent's behavioural engine in order to act in the environment. We use what we call interactions to achieve this. The following section is dedicated to the agent's behavioural cognitive engine. We present the structure of this engine and more precisely the planning and replanning algorithm.

KNOWLEDGE REPRESENTATION

Simulations consist in *agent*s that evolve in an *environment* and *interact* with the environment and other agents, according to the laws that rule the environment. Therefore, it is essential to describe these different core notions. We will first present the environment that is the basis of the situated side of the simulations. Second, we define the agents involved in our simulations, they are divided in passive (closer to "things") and active agents that are responsible of the dynamics of the simulation. These agents can suffer or perform interactions that are described in the third part. They represent the atomic knowledge beans used by the deliberative agent to act.

Environment

The environment describes the geography of the simulation. It provides the support to situate the agents and then to control the possibility of the realisation of some of their actions, when the notion of neighbourhood has an importance for example. The environment gives a meaning to the notion of *move* for an agent, although it is simple, this notion is full of importance since it impacts a lot on the dynamics, at least the visible one, of the simulation and it makes our concerns different from the pure planning problems. It is the environment too, that is in charge to determine which information can be perceived by an agent.

We represent the environment by a graph where vertices are *places* and edges denote *path* from one place to another.

$$
\begin{array}{lll}
\texttt{environment} & := & \{\texttt{place*,path*}\} \\
\texttt{path} & := & (\texttt{place}_{origin}\texttt{,} \\
 & & \quad \texttt{place}_{destination}\texttt{,} \\
 & & \quad \texttt{condition}) \\
\texttt{condition} & := & \texttt{boolean expression}
\end{array}
$$

A place is a geographical elementary area. The granularity of a place depends on the simulation, the only constraint is that inside a place there is no restriction neither for moves, nor for perception (restrictions due to the other agents, like collision problems, excepted). A place can represent a room, a town or any other part of the environment, and inside a place the position of an agent can be managed discretely or continuously depending on needs.

A path denotes an oriented transition between two places. It is defined by the places that it links, and a condition that must be satisfied if an agent wants to use this path. The edge is oriented and the condition to go from

some place a to a place b is not necessarily the same than the one to go from b to a.

This formalism allows to describe, for example, that a door between two rooms must be opened if we want to go from one room to the other, or that an agent must be able to swim to cross a river between two fields. In this last case, our approach allows, depending on needs, to choose to model or not the river with a place. It depends on whether the crossing of the river has a meaning in the simulation (see figures 1 and 2).

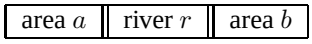

Figure 1: River is modelled. The paths are: (a,r,*"agent can swim"*), (r,a,*true*), (b,r,*"agent can swim"*), (r,b,*true*)

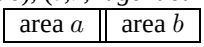

Figure 2: River is not modelled. The paths are:(a,b,*"agent can swim"*),(b,a,*"agent can swim"*).

The environment is the place where the agents are situated. It plays the role of a reference for the agents (then in this context the environment can not itself be an agent). Each agent is located in a place and can not be in a path. If the path must be put into concrete form, this must be done using a place, like we have seen it with the river example. Then a place is mainly characterised by the set of the agents that belongs to it.

The relative position of the agents inside a place (when this has a meaning) will be managed by the place itself, and is a parameter of the simulation.

Agents

The agents involved in the simulations we are interested in, are situated in a place of the environment. It is the environment that is in charge of the creation or removal of an agent in the simulation, even if the decision of these creations or removals is the result of the behaviours of the present agents.

We call *agent*, every entity that has some relevance in a simulation, that is, that can have an influence over the simulation. Among these agents, we distinguish two special classes: the *passive agents* and the *(pro-)active agents*. We use the terms of *inanimate* and *animate* agents too. It is for the latter that the notion of behaviour as a meaning.

In a rather natural and classical approach, agents are defined by a set of properties, a property being a pair (*name*, *value*). However, we will refine this definition (see Figure 3) and precise some particular properties imposed to our agents. We spend no time on the *name* property which allows to have a symbolic reference of the agent, but we rather insist on what characterize the agents: their abilities expressed by interactions.

Our agents (passive or active) are, at first, characterised by the actions (in the following we rather use the term "*interaction*" which denotes the way an action is coded) they can suffer. A *tree* agent could be cut, a *door* agent could be opened or painted, a *sheep* agent could be sheared, etc. We name *can-suffer* this property, the associated value is the list of interactions that the agent can suffer (that is for which he can be a target). The interactions are presented in the next section.

We must now study the particular case of the active agent. It is easy to guess that these agents have the possibility to interact with their environment, that is with the other agents (seen through their "passive" facet). These abilities are expressed by a collection of interactions they can perform, and defined in a property; we name *can-perform* this property.

However, this property remains a declaration of abilities. In order for an active agent to have an impact over the simulation, he must be provided with a behaviour engine that takes at every moment the decision of the action undertaken by the agent, and then of the used interaction. This decision depends on the context. This engine is influenced/directed by the existence of goals for the agent. The section is dedicated to the presentation of this engine.

Confusion must not be made between "active" or "animate" agent and the modelling of "living" entity. Thus, if in a simulation there is a machine which produces regularly some objects o, this must be modelled by an active agent whose goal would be the production of agents corresponding to o and whose behaviour would be the satisfaction of this goal.

We can point out another particular property: the memory of the active agent. It represents the knowledge base for all the information gathered by the agent concerning the environment: the topology of the environment, the other agents (their position and state), etc. This memory is a degraded environment insofar as it corresponds to the data the agent knows about the environment. This knowledge can be incomplete, for instance the agent does not know that others exist. It can even be wrong, for instance because the agent is not necessarily aware when other agents act and modify the state of some entities.

To come back on what we said at the beginning of this paragraph, an element must be considered as being represented by an agent in a simulation (ie. has an influence over the simulation), if and only if either it is active and the list of the interactions it can perform is not empty, and there exists at least one possible target for one of these interactions, or it is passive and the list of the interactions it can suffer from is not empty, and at least one active agent can perform one of these interactions.

It results from this definition that the interactions play an essential role in our simulations. Agents are different because they perform or suffer different interactions. Moreover the interactions define the "laws" of the simulated environment, and then play a central role in knowledge representation. We now define this notion.

Interactions

Interactions are the backbone of our simulation model, we could even speak of *interaction oriented simulations*.

```
agent          :=  passive-agent | active-agent
passive-agent  :=  {  ( "name", Symbol),
                      ("can-suffer", {interaction* }),
                      property*}
active-agent   :=  passive-agent ∪
                    { ("can-perform", {interaction*}),
                    ("goals", goal*),
                    ("memory", (degraded) environment),
                    ("engine", engine)}
```

Figure 3: Definition of an agent

These interactions are the basis of the knowledge representation in the simulations. They define the laws of the modelled world, that is the actions that can be performed in the simulations. They are central since the agent's engine uses them to build the agent's behaviour.

These interactions are the units of knowledge that describe the laws of the simulated world. They represent a declarative knowledge. A consequence is that an interaction must not, very special case excepted, be attached to one simulation but must represent a rather universal knowledge. This constitutes a difficulty in regards with the representation of these interactions, but allows to reuse them from one simulation to another one.

We characterize an interaction by an *actor* and a *target*. The *actor* is instantiated by an active agent who can perform this interaction and the *target* is any agent who can suffer from this interaction.

An interaction is defined in a rather classical way as presented in figure 4. The *name* is a unique identifier. The other three parts are:

- the *condition*, it tests the current context of execution of the interaction and consists mainly of tests on values of target or actor properties.

- the *guard*, it checks general conditions for the interaction applicability, typically it defines that to be fired an interaction requires that the distance between the target and the actor must be less than some given value.
  The guard is separated from the condition since it corresponds to the knowledge due to the geographically situated feature of the simulations. In a non situated context, one would have only the condition and action parts. The guards are at the origin of the moves in the plan, and this is these moves that are indeed specific to situated problems. Thus, we do not express explicitly in an interaction that the agent has a move to do in order to fire it, we want the agent to plan it when required by a guard.

- the *action*, it describes the consequence of the interaction, it can be a change in the state of the actor and/or of the target (ie. a change of the value of a property), and/or the activation of an environment action (like the creation of an agent).

Some interactions does not naturally obey to this schema of interaction between a target and an actor. This is the case, for example, for the "*sleep*" action. However, in this case it suffices to consider that the actor and the target are the same agent: the actor decides to make the target (himself) sleep, and thus he changes the state of the target. Such action can then be represented with the same interaction model.

More generally the consequence of an action is a change in the state of the target or actor. Thus to *open* an object (door, chest, window, etc.) makes it changing from *opened* state to *closed* one. The precise essence of the target is of no importance here, this knowledge must then be represented in a "universal" way by the interaction:

$$\textit{open:} \begin{cases} \texttt{condition} & = & \text{“}\texttt{target.opened} = \texttt{false}\text{”} \\ \texttt{guard} & = & \text{“}\texttt{distance}(\texttt{actor}, \texttt{target}) < 1\text{”} \\ \texttt{action} & = & \text{“}\texttt{target.opened} = \texttt{true}\text{”} \end{cases}$$

Such an action can be used by an engine to generate a plan such that: "*to push a button in the next room I must* open *this obstacle*" (or more precisely the knowledge would be *the obstacle must be opened*, and when this is not satisfied the given plan is produced). Whether this obstacle is a door or a window or anything else that is openable, the plan remains valid.

A problem arises when considering more "specific" agents. For instance, let us consider the case where the obstacle is a *lockable* door. To push the button, the above plan is still valid with respect to the knowledge that must be used and then with respect to the behaviour engine. The difference exists only in the condition for the execution of the action. This lockable door requires that, in its particular context, something like *target.lock = false* must be satisfy too. Then, from an abstract point of view the plan is still valid, but the *open* interaction must be understood as "make the door change from *closed* to *opened* state *when it is unlocked*". The problem is then how the same abstract plan (ie. "open the door to push the button") can receive different solutions (just "open" or "unlock and then open") depending on the target (whether it is lockable or not). To have two different interactions, one named *open-when-lockable* (or anything else) and the other named *open* is not relevant. As a first consequence

```
actor                :=  the agent who performs the interaction
target               :=  the agent who suffers the interaction

interaction          :=  name, condition, guard, action)
name                 :=  Symbol
guard                :=  d op Integer
d                    :=  distance between actor and target
op                   :=  = | > | < | ≤ | ≥
condition            :=  test_property | predicate_primitive(args)
test_property        :=  { actor | target }.property_name op Value
predicate_primitive  :=  primitive
action               :=  affect_property | primitive
affect_property      :=  { actor | target }.property_name = Value
primitive            :=  { actor | target }.primitive_name(args)
primitive_name       :=  Symbol
property_name        :=  Symbol
```

Figure 4: Definition of an interaction

this leads to a multiplication of interactions and implies that the active agents must be finely tuned. Moreover, the agent engine must take into account the different possible cases, although they conceptually represent the same action (*open* here). Thus it is more than probable that we would fall again into one of the major pitfall of the script approach for designing agent's behaviour.

Therefore, our proposition consists in the possibility to specify at the target level (ie. the agents having the considered interaction as a "*can-suffer*" one), the specific process. One can notice that only the nature of the target requires a change in the manipulation, not in the plan. During an interaction between such a target and an actor, the target "tells" to the actor the particular knowledge to be used while interacting with this target. For the actor (active agent), there is still only one generic interaction. Thus, the *can-suffer* property of a *lockable-door* agent contains the *open* interaction, with a specialization of the condition like: *target.locked = true*. Thus, when an actor tries to interact, using *open*, with this door, he gets the full condition "*target.opened = false and target.locked = false*". This leads the actor to (try to) unlock the door before it can open it. This can be seen as a kind of inheritance for interaction. It offers to the game designer the possibility to add new targets that specifies an existing one in order to take into account some particularity of the simulated world (for instance a *lockable door* that specifies a *door*). And this specification does not require the possible actors to be modified, at least their engine must not be changed. This flexibility eases the design of simulations and the ability to reuse interactions and agents from one simulation to another.

THE AGENT ENGINE

This section details the engine of the active agents. We first present the structure of the "mind" of the agent and the dependences between the different elements and second the planning and behaviour engine.

The Agent Structure

The decision cycle applied by the agent is presented in the figure 5. Continuously the agent perceives its environment. The acquired information are forwarded to an update module that can influence the memory and the currently established plan. An action is then chosen and the agent tries to execute it in the environment, and so on. To apply this cycle the agent is provided with a "mind".

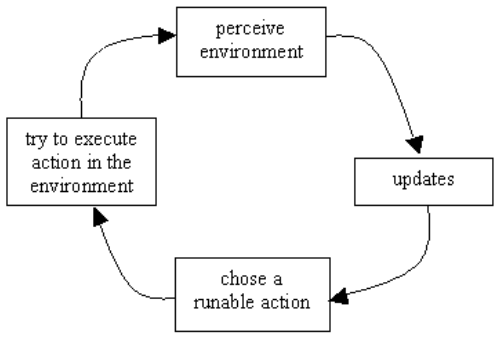

Figure 5: Agent decision cycle.

The "mind" of our active agents is made of several modules: a knowledge or beliefs base, a new information management module, a planning engine, an action selection module and an execution module. The articulation between these parts is illustrated in Figure 6.

The Knowledge Base.
The knowledge of the agent can be divided in two. On one side, the knowledge about the actions the agent can do and on the other side the knowledge about the environment he belongs to.

The first is defined by the set of all the interactions that the agent can perform. These interactions represent the absolute knowledge of the agent about an environment independently of any given particular context. They are the

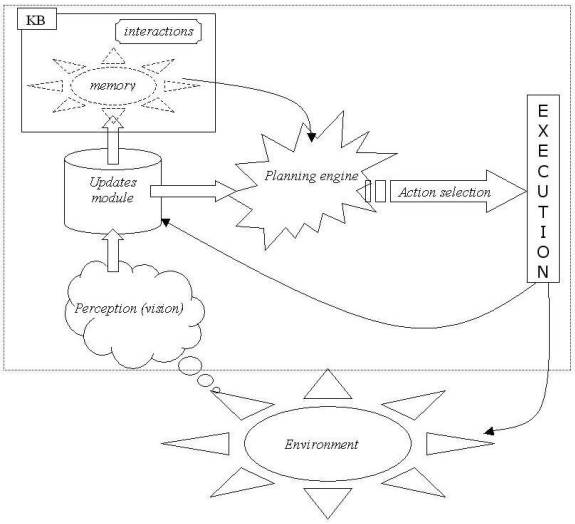

Figure 6: Different elements of agent mind.

basis of the behaviour engine to build plans.

The second corresponds to a base of beliefs and is called the *memory* of the agent. It is a contextual knowledge. It evolves according to the information perceived by the agent. It consists in the knowledge concerning the geography of the environment and in the information about the other agents. The memory is like a degraded environment and corresponds to the perception that the agent has of the environment. In the memory, some of the information can be marked *unknown*. Every known information is timestamped, this helps to estimate a confidence in the data: the older an information concerning the position of a mobile agent is, the less confident it is. The information in the memory are used by the behaviour engine to determine the one among the *can-perform* interactions that must be applied in order to achieve goals. These information are beliefs and not absolute knowledge, consequently when the agent tries to perform an action for which it believes all the conditions are satisfied, it is necessary to check if it is indeed the case in the environment.

Perception.

An innate and absolute knowledge of the environment in which the agent evolves will not produce realistic behaviour. Then it is necessary to provide the agent with a way to perceive new information while he is acting. Actually we content on a simple "visual" perception. The agent perceives the information of the environment that are inside its field of view (whose shape and radius can be changed at will). The perceived information are forwarded to the new information management module that is in charge to manage their influence. Since the perception module is only in charge of the perception and not of the treatment of the new information, it is easy to extend it to new kind of perceptions such as sound.

The update module.

As we previously say, the new information management module is in charge of the new perceived information. It is a kind of short term memory. It operates on two levels: first the memory in order to update the beliefs, and second the planning engine in order to adapt, if needed, the current computed plan through a partial re-planning. This is detailed in the following.

Planning, selection, execution.

These points will be more detailed in the next section. The engine is in charge of the resolution of the objectives of the agent. It uses the *can-perform* interactions of the knowledge base to build a plan of actions according to the memory. The built plan is valid according to the memory but can be wrong in the environment, this is checked at the execution step. This plan determines at every moment which actions the agent can undertake, an action selection strategy is then applied to choose the next effectively fired action. This strategy can be changed from one agent to the other to obtain different behaviours and then different individuality, even if the planning engine is the same. Once the action is chosen, the agent tries to execute it. Either the beliefs of the agent were right and the action is effectively performed in the environment, or they were wrong and the action can not be done and the agent must update its knowledge.

The Planning

The planning algorithm we use is a kind of backward chaining. To fulfil its goal, among all the interactions that it can perform, the (active) agent searches those that can help to achieve it, and then selects one. If the conditions of this action are satisfied (according to the agent memory), the (inter)action can be fired and the plan is done. Otherwise, the non satisfied conditions become new goals that need to be planned.

The goals.

There exist two kinds of goals. First the *interaction-goals*, they correspond to an (inter)action that the agent wants to execute. The target of this interaction can be less or more precisely given: from a named agent to any agent that can suffer the interaction, as shown in the next table:

goal	*type of target*
`eat(apple_12)`	a given precise apple
`eat(an apple)`	any apple
`eat(*)`	any eatable (ie. "who can-suffer from eat") agent

Second, the *condition-goals*, they correspond to a condition that the agent wants to bring to true. For instance:

`actor.energy > 100` "*having actor energy to be greater than 100*"

Planning tree.
In a rather classical way, the plan produced by the backward chaining can be viewed as a tree. The nodes are made of the different goals and subgoals encountered during the resolution. Some are interaction-goals, others are condition-goals. Thus this tree is an AND-OR tree. AND-nodes correspond to condition-nodes (for condition-goals) and OR-goals to interaction-nodes (for interaction-goals).

Condition-nodes and interaction-nodes: A condition-node has sons only if its condition is not satisfied. These sons are interaction-nodes built from the interactions whose action part offers a way to satisfy the condition (or to approach this satisfaction, for example by increasing the energy for the above given condition-goal example). The tree leaves are the satisfied condition-nodes (ie. whose conditions are satisfied).

The interaction-node's sons are built from the conditions that can be found in the condition and guard parts of the interaction: from these, condition-nodes are built. These sons are always built. An interaction-goal is said to be satisfied when all its sons are satisfied, the associated interaction is then declared runnable.

These correspond to the general cases, however since the simulations take place in situated environment, moves must be taken into account. They must receive particular considerations as discussed in (Devigne, Mathieu, and Routier 2004), this leads to introduce *move-nodes*

Move-nodes and exploration-nodes: *To move* or *to explore* the environment correspond for the agent to execution of interactions. The associated nodes must then be present in the planning tree as particular cases of interaction-nodes.

The exploration case can be reduced to the move case. To explore the agent must indeed make move towards a chosen location. The existence of the exploration-nodes are justified by the need to choose the targeted position before making the move. The agent must then apply its own exploration strategy to make its choice. Thus in the following we will only concentrate on the move case.

One problem is: what are the condition-nodes sons of a move-node? This problem amounts to ask what are the conditions that must be satisfied to make a move possible. To a move corresponds a computed path that is a sequence of elementary paths presented in the "Environment" section. With these elementary paths come conditions. A move is possible if these conditions are satisfied. With these conditions we create condition-nodes that become the sons of the considered move-node.

A classical backward chaining.
The planning tree is built according to the algorithm presented (in broad lines) in Figure 7. Every calculus are based on the memory (ie. beliefs base) of the agent. It is in particular the case when the agents checks a condition or computes a path for a move. Therefore, the computed plan is valid according to the agent memory, but can be wrong once it faces up to the reality of the environment.

For the exploration-nodes, the principle is roughly the same once the exploration strategy has provided a place to reach.

Every details are of course not presented in this algorithm. In particular if the same (sub)goal occurs more than one time during the planning, the corresponding node is not expanded twice, it is shared by its fathers. The tree is then an oriented graph.

But, actually, our algorithm builds the plan in an incremental way as we will see in section on replanning. Indeed, according to perceived information, the plan is adapted: the plan's tree is locally modified and not fully rebuilt.

A small example To illustrate the different points described in the previous paragraph we will consider a very small and simple example (for instance, we do not consider the *open* interaction). We consider a world with two places/rooms separated by a door d (see figure 8), the path between these two rooms has the condition "d.*locked=false*". Four interactions define the laws: *unlock*, *take*, *move*, *push* (see Table 1). In the world are an active agent a that can perform these 4 interactions, and three passive agents, the door d that can suffer *open*, a key k that can suffer from *take* (and can be used to unlock d) and a button b that can suffer from *push*. The goal of the agent is to push on b. The figure 8 presents the planning in two different situations.

unlock:	*condition*	=	"`target.locked = true`"
			"`actor.own(target.key)`"
	guard	=	"`distance(actor, target) < 1`"
	action	=	"`target.locked = false`"
take:	*condition*	=	`true`
	guard	=	"`distance(actor, target) < 1`"
	action	=	"`actor.own(target) = true`"
push:	*condition*	=	`true`
	guard	=	"`distance(actor, target) < 1`"
	action	=	"`target.pushed = true`"
move:	*condition*	=	`conditionsfoundinpath`
	guard	=	
	action	=	"`distance(actor, target) < 1`"

Table 1: Definitions of the interactions (adapted - but without distortion - to shorten the example)

As one can see, different plans are obtained depending on the context. Moreover in this case, this is because the agents are situated in the environment that two different plans exist. Indeed, it is because the actor must move near the button in order to push it that the state of the door is of importance. It results that the notions of neighbourhood and distance have a direct influence on the produced plan.

A partial replanning.
Active agents evolve in a dynamic environment. They establish a plan according to their knowledge, that can prove to be incorrect and then they can then be brought to adapt the computed plan according to new perceived information. These information can be of several types:

```
actor = the agent that builds the plan

this = interaction-node
expand()
  for each condition c in this.condition and this.guard
    newNode = createConditionNode(c)
    this.sons.add(newNode)
    newNode.expand()

this = condition-node
expand()
  If this.condition.isSatisfied()
    then finished
    else
              // gets the action that allows to solve condition
      BA = this.condition.getBackwardAction()
              // the list of can-perform candidates for BA
      LI = actor.getCanPerform(BA)
      for each I in LI
        if I is "moveTo"
          then this.sons.add(createMoveNode())
        else
              // list of known agents that can suffer I
          lAgents = actor.getKnownAgents(I)
          for each a in lAgents
            if actor.execute(I,a) satisfied this.condition
              newNode = createInteractionNode(I,a)
              this.sons.add(newNode)
              newNode.expand()
            end if
        end if
        if this.sons.isEmpty()
          then this.sons.add(create-exploration-node)

this = move-node
expand()
  path = actor.computePath()
  If path = null            // no path found
    then this.sons.add(createConditionNode("false")
    else
              // list of conditions≠"true" on paths
      conditionsList = path.getPathsConditions()
      If conditionsList.isEmpty()
        then this.sons.add(createConditionNode("true")
        else for each condition c in conditionsList
          newNode = createConditionNode(c)
          this.sons.add(newNode)
          newNode.expand()
```

Figure 7: Node's expansion algorithm.

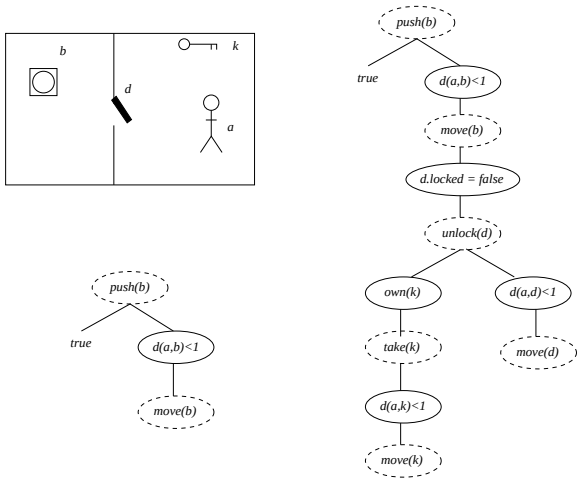

Figure 8: An active agent a is situated in an environment where are also 3 passive agents: a door d, a button b and a key k. The goal of a is to push on b. a must build a plan to achieve it. A plan can be drawn as a tree, nodes due interaction-goals are drawn with dashed lines and nodes due to condition-goals with solid lines. Depending on the execution context, different plans can be obtained. Left is the tree obtained when d is not locked and right is the case where d is locked. a must adapt its plan to the context.

- *a new information*: the agent learns that a so far unknown information exists. It is the case when the agent sees a new place, meets another agent for the first time, gets a new goal, discovers a condition on a path of the graph, etc.
- *a modification of an existing information*: it concerns mainly modifications about the state of known agents, a property value change or a position change. The position change information covers three situations:

known→known:	we thought agent at a position and we see it at another
known→unknown:	we thought agent at a viewed position and it is not there
unknown→known:	we did not know where the agent were and we now see it

To each of these situations is associated an event that describes what must be modified (changed or added) in the agent memory. The table 2 lists these events, one can easily guess what they are according to the previous paragraphs.

These events are transmitted to the update module who is in charge to take them into account and to consider their impact. First, the memory of the agent must be modified: either a new knowledge is added, or an existing data must be corrected. Second, because of the changes, it is possible that the currently established plan must be adapted. It is the new information management module that is in charge of forwarding these information to the planning engine.

new information	NewGoal, NewPlace, NewPathCondition, NewAgent, NewInteraction
modification	AgentModified, AgentMovedKK, AgentMovedKU, AgentMovedUK

Table 2: List of events for new information

However, a new information concerns only a portion (even none) of the planning tree. Therefore, it is neither reasonable, nor efficient, to rebuild a new plan for every new information. Indeed, even if it is established that, in theory, no efficiency gain can be guaranteed while using plan reuse rather than new plan generation (Nebel and Koehler 1993), in practice improvements can be expected. Indeed, our context of dynamic simulations corresponds to the case where the agent perceives very frequently slight changes of its knowledge base.

In particular, this is due to the fact that the agent engine uses uncertain information: the planning is based on the information that are in the memory. But, since simulations occur in open dynamic environments, then the built plans are correct with respect to the memory of the agent, but can be wrong once confronted with the real environment, at the execution step. Therefore only partial and local adaptations can be expected in most of cases. Our experiments confirm that.

Our approach is then to top-down propagate events from root to leaves in the planning tree. Each node checks if it is concerned by each event, first because it is the good type and second because additional conditions are satisfied. Checking these conditions is very fast, therefore propagation costs not too much time and in particular less than a replanning when events have no impact. Thus, only the appropriate nodes are updated (collapsed, re-unfolded, adding or removal of subnodes, etc.).

It would probably be a bit tedious to enumerate all the cases and conditions for which a node is affected by an event. Thus we give only two examples. For instance, a new information that affects the graph topology or agent positions can (but not systematically) have an impact on move-nodes. Indeed a new path can “appear” or at the opposite a computed path can become blocked, in these cases, the move-node should be re-expanded. In a similar way, a new met agent can affect a condition-nodes. For instance this can be the case if this agent can be the target of an interaction that helps to solve the condition (that is one of the interactions stored in the *LI* list seen in the algorithm presented in Figure 7). In this case a new interaction-subnode must be added and unfolded. Other cases are similar.

Using this principle, there is no need to recompute the plan at each step. It is indeed probable that only few new information occur each step, and this is not necessarily of importance for the agent. But this principle helps the agent to remains reactive too and to adapts as soon as it

is necessary and relevant. Moreover this partial and local tree modification leads the plan to be incrementally built, change after change.

CONCLUSION

The design of simulations of behaviours that will be perceived and evaluated as believable by a human external observer is not an easy problem. The solutions proposed by the reactive systems are not generic and reusable enough.

We endeavour to propose a general model for the simulation of behaviours. The dynamics of this model relies on the description of interactions that can be performed or suffered by some agents that are situated in a dynamic environment. An individual generic behavioural engine uses these interactions to propose a plan of actions to the deliberative agents.

A natural application of this work are computer games. Targeted games could be action and role-playing games, where one needs complex believable non-player characters to increase the quality of the simulated world and the interactions of the player with it.

Our current work concerns the application of this model to team of agents (Devigne, Mathieu, and Routier 2005). It is clear that it is a challenge for simulations, and games in particular, to be able to design groups of characters that act together to fulfil common objectives.

AUTHOR BIOGRAPHIES

DAMIEN DEVIGNE is Ph.D. student at the University of Lille, in the SMAC research team (`http://www.lifl.fr/SMAC`) LIFL (UMR CNRS 8022) laboratory.He works on simulation of teams of cognitive agents in a geographical environment. E-mail: `devigne@lifl.fr`, personal web-page: `http://www.lifl.fr/~devigne`.

PHILIPPE MATHIEU is professor at the University of Lille. He receives his Ph D. in 1991. He is leader of the SMAC research team of the LIFL laboratory. He is specialized in multi-agent systems, modelization of behaviours and game theory. E-mail: `mathieu@lifl.fr`, personal web-page: `http://www.lifl.fr/~mathieu`.

JEAN-CHRISTOPHE ROUTIER is assistant professor at the University of Lille. He receives his Ph D. in 1994. He is member of the SMAC research team of the LIFL laboratory. His research centers of interest cover agent platforms, agent-oriented software engineering, and recently the design of believable behaviours for simulations (like games). E-mail: `routier@lifl.fr`, personal web-page: `http://www.lifl.fr/~routier`.

REFERENCES

Allbeck, J.; and N. Badler. 2003. "Representing and Parameterizing Agent Behaviors," in *Life-like Characters: Tools, Affective Functions and Applications.*, ed. by H. Prendinger, and M. Ishizuka, 19–39.

Brooks, R. A.; and al.. 1999. "The COG Project: Building a Humanoid Robot," in *Computation for Metaphors, Analogy, and Agents*, vol. 1562 of *LNCS*, 52–87. Springer.

Devigne, D.; P. Mathieu; and J.-C. Routier. 2004. "Planning for Spatially Situated Agents in Simulations," in *Proceedings of the 2004 IEEE/WIC/ACM International Joint Conference IAT'04*, 385–388.

Devigne, D.; P. Mathieu; and J.-C. Routier. 2005. "Team of cognitive agents with leader : how to let them autonomy," in *Proceedings of 2005 IEEE Symposium on Computational Intelligence and Games*.

Laird, J. E.; and M. van Lent. 2000. "Human-level AI's Killer Application: Interactive Computer Games," in *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence.*, 1171 – 1178. AAAI.

Magerko, B.; J. Laird; M. Assanie; A. Kerfoot; and D. Stokes. 2004. "AI Characters and Directors for Interactive Computer Games," in *Proceedings of the 2004 Innovative Applications of Artificial Intelligence Conference*, ed. by A. Press.

Nareyek, A.. 2000. "Intelligent Agents for Computer Games," in *Computers and Games, Second International Conference, CG 2000. LNCS 2063.*, 414–422.

Nareyek, A.. 2004. "Artificial Intelligence in Computer Games - State of the Art and Future Directions," *ACM Queue*, 1(10), 58–65.

Nebel, B.; and J. Koehler. 1993. "Plan modification versus plan generation: A complexity-theoretic perspective," in *Proceedings of of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, 1436–1441.

Niederberger, C.; and M. Gross. 2003. "Hierarchical and Heterogenous Reactive Agents for Real-Time Applications," in *Proceedings of the Eurographics 2003, Computer Graphics Forum, Conference Issue*, ed. by P. Brunet, and D. Fellner, 323–331. Blackwell Publishing, UK.

Ponsen, M.; and P. Spronck. 2004. "Improving Adaptive Game AI with Evolutionary Learning.," in *Proceedings of Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)*, ed. by S. N. Quasim Mehdi, Norman Gough, and D. Al-Dabass, 389–396.

Spronck, P.; I. Sprinkhuizen-Kuyper; and E. Postma. 2004. "Enhancing the Performance of Dynamic Scripting in Computer Games," in *Entertainment Computing - ICEC 2004*, ed. by M. Rauterberg, no. 3166 in Lecture Notes in Computer Science, 296–307. Springer-Verlag.

Tozour, P.. 2002. "The Perils of AI Scripting," in *AI Game Programming Wisdom*, ed. by S. Rabin, 541–547. Charles River Media.