Evolving Characters in Role Playing Games

João Leite, Luís Soares
CENTRIA
New University of Lisbon, Portugal
jleite@di.fct.unl.pt, luizsoarez@gmail.com

Abstract

In this paper, we report on the enhancement of the Java based Multi-Agent Platform madAgents with Evolving Logic Programming (EVOLP), and its application to Massive Multi-player Online Role-Playing Games. The resulting system has been implemented using a combination of Java, XSB Prolog and Smodels, and allows for the specification of non-playing characters where beliefs and behaviours are specifiable in EVOLP, thus evolvable.

1 Introduction

A Role-Playing Game is essentially a story. As in a regular story, it has characters. Some of these characters have the ability to directly intervene in the story, affecting its course by executing actions such as interacting with other characters, buying and selling items, learning skills, using magic, etc. They are autonomous and have their own agenda in the story, just like the main characters in a movie. They are called Player Characters (PCs) and can either be artificial or human. There are also some characters that have no capacity to directly interfere with the story. They are shopkeepers and other characters that may hold certain key items or information for the players, and whose interaction with the players is in the form of conversations. These are called Non-Player Characters and are always artificial.

Non-Player Characters are usually dumb reactive agents, often implemented by simple finite-state machines that, although being able to play a relevant role in the story (the NPC who has the weapon the player needs, or the NPC who knows a secret password) aren't viewed as key characters. They are however, as almost every aspect in a role-playing game, essential to the added realism of the game. In these games, realism and immersion are the main strengths the game developer has to play with, since the focus resides not only on completing goals but also on providing the player with an experience of his character's life as if it were real. In this aspect, a non-player character who gives players the feeling it has an unpredictable complex behaviour provides an advantage in terms of realism. A key aspect to this realism lies in the evolving capabilities demonstrated by the characters populating the game, yet another aspect where computer games can be seen as a killer application for AI [Laird and van Lent, 2000].

The MulE Game Engine (Multi-player Evolutive Game Engine) [Assunção *et al.*, 2005] is an open-source system implemented to develop realistic Massive Multi-player Online Role-Playing Games (MMORPGs). Its main role is to provide help in developing MMORPGs, and free the game developers to concentrate more on playability details and game design issues, than on low-level implementation. It is composed of several modules, one of which, the AI Server, being responsible for managing artificial players (both non-player and player characters). The AI Server is, essentially, a multi-agent platform providing support for the specification and execution of all such characters, now being implemented with madAgents (Multimedia Deductive Agents) [Soares *et al.*, submitted], a Java and XSB-Prolog based multi-agent platform. The extent to which the programmer can specify such characters and their behaviour is key characteristic of the overall realism of the game, thus its success.

In this paper we borrow recent advances from the area of Computational Logics for Multi-Agent Systems to enhance the madAgents platform, thus the MulE Game Engine, with features that will allow for the design of NPCs with richer behaviours that evolve as the game progresses.

We have recently seen an increase in the cross fertilisation between the areas of Multi-Agent Systems (MAS) and Computational Logic (CL). On the one hand, CL provides a rigorous, general, integrative and encompassing framework for systematically studying computation, be it syntax, semantics, procedures, or implementations, tools, and standards. On the other hand, MAS provides a rich and demanding environment populated with problems that challenge Computational Logic. Examples of this cross fertilisation include CL based MAS such as IMPACT [Subrahmanian *et al.*, 2000], 3APL [Dastani *et al.*, 2005], Jason [Bordini *et al.*, 2005a] and ProSOCS [Bracciali *et al.*, 2006], just to name a few. For a survey on some of these systems, as well as other, see [Bordini *et al.*, 2005b].

While CL, and Logic Programming in particular, can be seen as a good representation language for static knowledge, if we are to use it in scenarios where the evolution of logical specifications is a key issue, such as MMORPGs, we must consider ways and means of representing and integrating the evolution of knowledge in the form of updates, from external as well as internal sources (or self updates).

In fact, an agent should not only comprise knowledge about its behavior, but also about the way such behaviour

should evolve. The lack of rich mechanisms to represent and reason about dynamic knowledge and agents i.e. represent and reason about environments where not only some facts about it change, but also the rules that govern it, and where the behaviours of agents also change, is common to the above mentioned MAS.

To address this issue, Evolving Logic Programming (EVOLP) was introduced in [Alferes *et al.*, 2002]. EVOLP generalises Answer-set Programming [Gelfond and Lifschitz, 1990] to allow for the specification of a program's own evolution, in a single unified way, by permitting rules to indicate assertive conclusions in the form of program rules. Such assertions, whenever belonging to a model of the program P, can be employed to generate an updated version of P. Furthermore, EVOLP also permits, besides internal or self updates, updates arising from the environment. The resulting language, EVOLP, provides a simpler and more general formulation of logic program updating, setting itself on a firm formal basis in which to express, implement, and reason about dynamic knowledge bases.

EVOLP can adequately express the semantics resulting from successive updates to logic programs. These updates can be considered as incremental specifications of agents, and whose effect can be contextual. Unlike other approaches, it automatically and appropriately deals, via its update semantics [Leite, 2003], with the possible contradictions arising from successive specification changes and refinements. Furthermore, EVOLP can express self-modifications triggered by the evolution context itself, present or future. Additionally, foreseen updates not yet executed can automatically trigger other updates. Updates can also be nested, so as to determine change both in the next state and in other states further down the evolution.

In this paper, we report on the enhancement of the Java based Multi-Agent Platform madAgents with EVOLP, and its application to MMORPGs. This constitutes a *de facto* improvement of madAgents as a provider of the MulE Game Engine's AI Server, allowing for the implementation of a richer agent architecture where NPCs' beliefs and behaviour, as well as their evolution, are specifiable in EVOLP. This allows for the inheritance of all merits of Answer Set Programming (e.g., default negation for reasoning about incomplete knowledge; semantics based on multiple answer-sets for reasoning about several possible worlds; a number of extensions such as preferences, revision, abduction, etc.) on top of which we add all the specific merits of EVOLP for specifying evolving knowledge.

The paper is organised as follows: in Section 2 we briefly overview the MulE Game Engine and the Multi-Agent Platform madAgents; in Section 3 we define the EVOLP based agent architecture and its semantics; in Section 4 we elaborate on some aspects related to the implementation; in Section 5 we briefly illustrate with a small example, to draw some final remarks in Section 6.

2 Game Engine and Multi-Agent Platform

The MulE Game Engine [Assunção *et al.*, 2005] is an architecture designed to serve as the base to develop online role-playing games. It consists of a set of modules which implement the basic functions in a RPG, leaving the content creation tasks to the game developer. In a nutshell, it includes the following main server modules:

Player management server, which includes the player's database with each player's profile, and manages the game's login/logout process.

World server, which maintains the representation of the world and includes the management of time and its effects on players and items, the management of actions which handles all interactions between characters and their actions, and the broadcasting of what each player is perceiving at any given time.

AI server, an instance of the madAgents architecture, described below, which handles all NPCs in the game, as well as all artificial player characters.

Additionally, in order to facilitate the interaction with the game, the game developers have a Client Base, a set of methods and functionalities that all players will need during the game.

In this paper we will focus on the extension of the madAgents platform for logic-based agents, that serves as a base to the development of multi-agent systems, freeing the developer from tasks specific to the distributed systems areas. The main purpose of madAgents is to provide a flexible and customisable basis for developing and deploying multi-agent systems, so that most effort can be devoted to the artificial intelligence dimension of the problem.

The platform is implemented with Java (`http://java.sun.com`) and XSB Prolog (`http://xsb.sourceforge.net`), and includes a set of classes and Prolog files from which to extend the application and build the agents. Prolog is used to handle all Knowledge Representation and Reasoning, connecting to Java modules through InterProlog (`http://www.declarativa.com/InterProlog`) interface. The platform is based on a simple semantics, defined by a set of interactions between agents and other software modules. This way the platform supports an agent architecture that can be easily extended to integrate other modules or functionalities, while the meaning of the resulting application is only dependent on the nature of the modules, the actions of the agents and their reasoning rules. The platform's main functionalities include:

• A communication system built over TCP/IP, allowing agents to control the sending and receiving of messages. Message handling is automatically managed by the system which checks for messages arriving at the inbox and sends messages added to the outbox. Message preprocessing can also be handled by this module, which can be tuned to any specific agent communication language or a developer-defined language. Furthermore, the final destination of the messages, i.e., if they call Java methods or if they are asserted to the agent's knowledge base, is also handled by this module;

• Support for local and RMI-based actions. Actions, can be more than just sending messages and, in some cases, executing a method can be more expressive than requesting such in a message. Local method execution was already possible because of the InterProlog interface. But it could also be the case that an agent could provide methods for other agents to call remotely. This can be easily accomplished with the use of Java's Remote Method Invocation capabilities. The madAgents architecture already provides templates of RMI-based actions that can be called remotely

from other agents, thus helping in the development of applications based on this technology.

• Synchronous or asynchronous agent execution. Agents can be synchronised with the platform manager, making their execution cycle dependent on a certain message from the platform manager. Only after receiving this message can the agent go on with its reasoning cycle (deciding which actions to execute and executing them). But agents can also be implemented so as to function like an asynchronous distributed system, where there is no global clock and the reasoning speed of the agents may differ. In this case, the multi-agent application has a non-deterministic execution. To further improve this sense of non-determinism, the agents' reasoning speed can also be defined, providing the application with agents of different reactive nature.

3 Agent Architecture

In this Section we present our proposed architecture, based on EVOLP, which supports mental agents whose epistemic specification may dynamically change, without loosing its theoretical character.

3.1 Evolving Logic Programs

In a nutshell, EVOLP [Alferes *et al.*, 2002] is a simple though quite powerful extension of logic programming, which allows for modeling the dynamics of knowledge bases expressed by programs, be it caused by external events, or by internal requirements for change. From the syntactical point of view, evolving programs are just generalized logic programs (i.e. extended LPs plus default negation in rule heads), extended with (possibly nested) assertions, whether in heads or bodies of rules. EVOLP was designed, above all, with the desire to add as few new constructs to traditional logic programming (or answer set programming) as possible. From the semantical point of view, a model-theoretic characterization is offered of the possible evolutions of such programs. These evolutions arise both from self (i.e. internal to the agent) updating, and from external updating originating in the environment (including other agents). For sake of self containment of this paper, we include the main definitions concerning EVOLP. Further details can be found in [Alferes *et al.*, 2002].

We start with the usual preliminaries. Let $\mathcal{A}$ be a set of propositional atoms. An objective literal is either an atom A or a strongly negated atom $\neg A$. A default literal is an objective literal preceded by not. A literal is either an objective literal or a default literal. A rule r is an ordered pair $H(r) \leftarrow B(r)$ where $H(r)$ (dubbed the head of the rule) is a literal and $B(r)$ (dubbed the body of the rule) is a finite set of literals. A rule with $H(r) = L_0$ and $B(r) = \{L_1, \ldots, L_n\}$ will simply be written as $L_0 \leftarrow L_1, \ldots, L_n$. A generalized logic program (*GLP*) P, in $\mathcal{A}$, is a finite or infinite set of rules. If $H(r) = A$ (resp. $H(r) = not\, A$) then $not\, H(r) = not\, A$ (resp. $not\, H(r) = A$). If $H(r) = \neg A$, then $\neg H(r) = A$. By the expanded generalized logic program corresponding to the GLP P, denoted by $\mathbf{P}$, we mean the GLP obtained by augmenting P with a rule of the form $not\, \neg H(r) \leftarrow B(r)$ for every rule, in P, of the form $H(r) \leftarrow B(r)$, where $H(r)$ is an objective literal. Two rules r and r' are conflicting, denoted by $r \bowtie r'$, iff $H(r) = not\, H(r')$. An interpretation M of $\mathcal{A}$ is a set of objective literals that is consistent i.e., M does not contain both A and $\neg A$. An objective literal L is true in M, denoted by $M \models L$, iff $L \in M$, and false otherwise. A default literal $not\, L$ is true in M, denoted by $M \models not\, L$, iff $L \notin M$, and false otherwise. A set of literals B is true in M, denoted by $M \models B$, iff each literal in B is true in M.

Definition 1 *Let $\mathcal{A}$ be a set of propositional atoms (not containing $assert/1$). The* extended language *$\mathcal{A}_{assert}$ is defined inductively as follows: – All propositional atoms in $\mathcal{A}$ are propositional atoms in $\mathcal{A}_{assert}$; – If r is a rule over $\mathcal{A}_{assert}$ then $assert(r)$ is a propositional atom of $\mathcal{A}_{assert}$; – Nothing else is a propositional atom in $\mathcal{A}_{assert}$.*

An evolving logic program *over a language $\mathcal{A}$ is a GLP over $\mathcal{A}_{assert}$. An* event sequence *over $\mathcal{A}$ is a sequence of evolving logic programs over $\mathcal{A}$.*

Definition 2 *Let $\mathcal{P} = (P_1, \ldots, P_s)$ be a sequence of GLPs (or a dynamic logic program) over language $\mathcal{A}$. An interpretation M is a (refined) dynamic stable model of $\mathcal{P}$ iff $M' = least\left([\rho(\mathcal{P}) - Rej(M)] \cup Def(M)\right)$ where $Def(M) = \{not\, A | \nexists r \in \rho(\mathcal{P}), H(r) = A, M \models B(r)\}$, $Rej(M) = \{r | r \in \mathbf{P}_i, \exists r' \in \mathbf{P}_j, i \leq j, r \bowtie r', M \models B(r')\}$ and A is an objective literal, $\rho(\mathcal{P})$ denotes the multiset of the rules in the programs $\mathbf{P}_1, ..., \mathbf{P}_s$, $M' = M \cup \{not\_A \mid A \notin M\}$, and $least(.)$ denotes the least model of the definite program obtained from the argument program by replacing every default literal $not\, A$ by a new atom not_A. By $DSM(\mathcal{P})$ we mean the set of all dynamic stable models of $\mathcal{P}$.*

Definition 3 *An* evolution interpretation *of length n of an evolving program P over $\mathcal{A}$ is a finite sequence $\mathcal{I} = (I_1, I_2, \ldots, I_n)$ of interpretations of $\mathcal{A}_{assert}$. The* evolution trace *associated with evolution interpretation $\mathcal{I}$ is the sequence of programs $(P_1, P_2, \ldots, P_n)$ where: $P_1 = P$ and $P_i = \{r \mid assert(r) \in I_{i-1}\}$, for each $2 \leq i \leq n$.*

Definition 4 *An evolution interpretation $(I_1, I_2, \ldots, I_n)$, of length n, with evolution trace $(P_1, P_2, \ldots, P_n)$ is an* evolution stable model *of an evolving program P given a sequence of events $(E_1, E_2, \ldots, E_k)$, with $n \leq k$, iff for every i ($1 \leq i \leq n$), I_i is a dynamic stable model at state i of $(P_1, P_2 \ldots, (P_i \cup E_i))$.*

3.2 Architecture

We now turn our attention to the agent architecture. Each agent is conceptually divided into three distinct layers as depicted in Figure 1. A Platform Layer that deals with the necessary platform specific protocols such as registration, coordination, control, etc. A Physical Layer responsible for interfacing the agent with the environment, providing an actuator responsible for executing actions in the environment as well as an inbox and outbox to process the all incoming and outgoing events. A Mental Layer responsible for maintaining the agent's beliefs, behaviour and capabilities and deliberation processes. In this Section we will describe in greater detail the Mental Layer. Lack of space prevents us from further detailing the other two layers.

The Mental Layer can be seen as being divided into three main components, namely the Beliefs, Capabilities and Deliberation, even though each may provide for more than one aspect. The Beliefs module is specified in EVOLP,

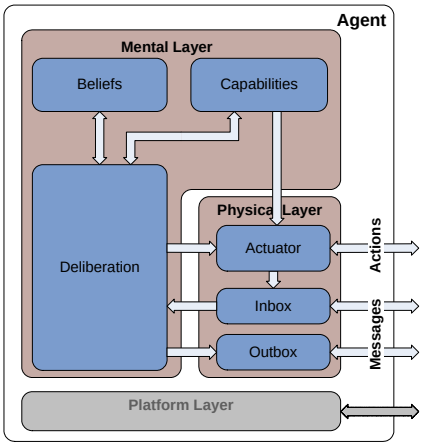

Figure 1: Agent Architecture

thus specifying not only the beliefs of the agent but also its behaviour and evolution. The Capabilities module contains information regarding the actions that the agent can perform, together with their epistemic effects, in the form of updates both to the agent's own Beliefs as well as to other agent's beliefs. The Deliberation Module executes the common observe-think-act cycle, implementing some specific semantics. Formally, an agent initial specification is defined as follows:

Definition 5 *Let $\mathcal{N} = \{\alpha_1, ..., \alpha_n\}$ be a set of agent names and $\mathcal{A}$ a propositional language*[1]*. An agent (α_i) initial specification, or an agent α_i at state 1, is a tuple $\langle \alpha_i, \mathcal{C}, Sel(\cdot), \mathcal{P}_1 \rangle$ where:*

- *$\mathcal{C}$ is a pair $(\mathcal{A}_\mathcal{C}, \mathcal{E}_\mathcal{F})$ representing the capabilities of the agent where $\mathcal{A}_\mathcal{C}$ is a set of propositions, each representing an action that agent α_i is capable of preforming and $\mathcal{E}_\mathcal{F}$ is a set of predicates of the form effects$(Action, \alpha_j, Effect)$ with $Action \in \mathcal{A}_\mathcal{C}$, $\alpha_j \in \mathcal{N}$ and Effect is an EVOLP program over language $\mathcal{A} \cup \mathcal{A}_\mathcal{C}^{\alpha_j}$ (where $\mathcal{A}_\mathcal{C}^{\alpha_j}$ is the set of actions that agent α_j is capable of preforming) representing the epistemic effects, on α_j, of agent α_i performing action $Action$ i.e., after $Action$ is performed, Effect should be added to agent's j next event;*
- *$Sel(\cdot)$ is some selection function that selects one of a set of dynamic stable models;*
- *$\mathcal{P}_1 = (P_1)$ is a dynamic logic program consisting of just one evolving logic program, P_1, over language $\mathcal{A} \cup \mathcal{A}_\mathcal{C}$, representing the agent's initial beliefs and behaviour specification.*

According to the execution mode specified in the Platform Layer (e.g. synchronous, asynchronous, etc.) an agent evolves into the next state as per the following observe-think-act cycle and the following definition:

$cycle(s, \langle \alpha_i, (\mathcal{A}_\mathcal{C}, \mathcal{E}_\mathcal{F}), Sel(\cdot), \mathcal{P}_s \rangle)$
  $observe$ (perceive E_s from inbox)
  $think$ $(M_s = Sel(DSM(\mathcal{P}_{s-1}, (P_s \cup E_s))))$
  act (execute actions $M_s \cap \mathcal{A}_\mathcal{C}$)
  $cycle(s+1, \langle \alpha_i, (\mathcal{A}_\mathcal{C}, \mathcal{E}_\mathcal{F}), Sel(\cdot), \mathcal{P}_{s+1} \rangle)$

---

[1]Without loss of generality, we assume that all agents share a common language.

Definition 6 *Let $\langle \alpha_i, \mathcal{C}, Sel(\cdot), \mathcal{P}_s \rangle$ be agent α_i at state s, and E_s the events perceived by agent α_i at state s. Agent α_i at state $s+1$ is $\langle \alpha_i, \mathcal{C}, Sel(\cdot), \mathcal{P}_{s+1} \rangle$ where*[2] *$\mathcal{P}_{s+1} = (\mathcal{P}_s, P_{s+1})$, and $P_{s+1} = \{r \mid assert(r) \in M_s\}$, and $M_s = Sel(DSM(\mathcal{P}_{s-1}, (P_s \cup E_s)))$.*

In the previous definition, we assume the capabilities to be fixed. However, it needs not be so as we can easily allow for the specification of updates to this component if we wish to have the agent, for example, learn new capabilities.

Unlike its original inductive definition, we employ a constructive view of EVOLP. The main difference relates to the existence of the selection function that trims some possible evolutions when selecting one stable model, corresponding to the commitment made when some set of actions is executed, and not another. This commitment is obtained by fixing the trace, instead of simply storing the initial program and sequence of events.

4 Implementation

The implementation employs five different technologies, namely Java, XSB Prolog, Interprolog, XASP (package provided by XSB), Smodels (`http://www.tcs.hut.fi/Software/smodels`). In the madAgents platform, the agents' execution is controlled from the Java component, where each agent is represented by a Java Thread. Each agent has an instance of a `MessageHandler` class, which handles all communication and a `PrologConnection` class which interfaces with XSB-Prolog through InterProlog. The Java component also provides a Graphical User Interface. The agent's execution cycle proceeds as follows:

1. Wait for the agent's reasoning delay to exhaust, or wait for a synchronisation message to arrive from the platform manager;
2. Check for new messages and process them. Message processing can be the assertion of the message in the agent's knowledge base or the direct execution of a certain method. Messages may originate in other agents, in the platform manager or encode outside events. Communication is implemented so as to provide a transparent message exchange mechanism through an inbox and an outbox, the `MessageHandler` class doing all the rest. This class also contains a simple "address book", allowing the agent to know the host and port where to contact a certain agent;
3. Select one dynamic stable model which will encode the actions to execute. Determining the set of dynamic stable models is performed in two steps: first the DLP is syntactically transformed into a single logic program, written in an extended language, whose answer-sets are in a one-to-one correspondence with the dynamic stable models of the initial DLP, following the results in [Leite, 2003]; subsequently, Smodels is invoked from within XSB, using the interface XASP, to determine the stable models of the transformed program. Finally, the Selection function is used to select one of these models. If the selection function chooses randomly, we can have Smodels determine one model only, thus saving computational time;
4. Execute the chosen actions. Action execution may include method calling or message sending;

[2]If $\mathcal{P} = (P_1, ..., P_s)$ is a DLP and P an evolving logic program, by $(\mathcal{P}, P)$ we mean the DLP $(P_1, ..., P_s, P)$.

5. Determine the next program in the trace and cycle again. Since the transformation mentioned above is incremental, the agent keeps the transformed program and adds to it the rules corresponding to the new part of the trace.

5 A shopkeeper with an attitude

In this Section we illustrate some of the capabilities provided by our system. Lack of space prevents us from providing a full specification of a NPC. Instead, we will appeal to the reader's intuitions and describe some key aspects. For this, we ask the reader enter Tolkien's universe, the Lord of the Rings, and to consider one of the most common characters in role-playing games, the shopkeeper. It's role is to run a store and sell items to customers. However, it does not treat all customers alike, and is a bit of a racist when it comes to Orcs and Elfs since it does not trust them. It does trust Hobbits. This can be represented by the facts, in the shopkeeper's initial program: $\neg trust\,(orc)$, $\neg trust\,(elf)$ and $trust\,(hobbit)$. In general, the shopkeeper will trust a player if it is of a trusted race, and will not trust a player if it is of an untrusted race:

$$trust\,(P) \leftarrow player\,(P,R)\,, trust\,(R)\,. \qquad (1)$$
$$\neg trust\,(P) \leftarrow player\,(P,R)\,, \neg trust\,(R)\,. \qquad (2)$$

In our story we will have the following players: $player\,(aragorn, human)$, $player\,(gandalf, human)$, $player\,(legolas, elf)\,, player\,(frodo, hobbit)$. Furthermore we will consider two items, lembas bread, a special food made by the elves, and wine.

The shopkeeper's general policy is to sell non deteriorated items to trusted players, whenever they want to buy them. The buying request will come as an event of the form $buy\,(Item, Player)$, and the previous selling policy could be encoded as

$$sell\,(I,P) \leftarrow trust\,(P)\,, not\, deter\,(I)\,, buy\,(I,P)\,.$$

When it comes to untrusted players, the shopkeeper will not sell good items:

$$\neg sell\,(I,P) \leftarrow \neg trust\,(P)\,, not\, deter\,(I)\,, buy\,(I,P)\,.$$

He will, however, sell deteriorated items to these players:

$$sell\,(I,P) \leftarrow \neg trust\,(P)\,, deter\,(I)\,, buy\,(I,P)\,.$$

When it comes to players that are neither known to be trusted nor untrusted, the shopkeeper will either sell the items, or not, non-deterministically. This can be encoded with the usual even cycle through default negation, generating two stable models, one to be non-deterministically chosen by the selection function. The rules are:

$$\begin{aligned} sell\,(I,P) \leftarrow & not\, trust\,(P)\,, not\, \neg trust\,(P)\,, \\ & buy\,(I,P)\,, not\, \neg sell\,(I,P)\,. \\ \neg sell\,(I,P) \leftarrow & not\, trust\,(P)\,, not\, \neg trust\,(P)\,, \\ & buy\,(I,P)\,, not\, sell\,(I,P)\,. \end{aligned}$$

With this set of rules, any request from Frodo for either bread or wine will be attended, since none of the items is known to be deteriorated and Frodo is trusted (because Hobbits are trusted). Requests by Legolas would not be attended. All requests by either Aragorn and Gandalf are either attended for , or not, in a non-deterministic way. The scenario so far has illustrated the use of both default and strong negations, as well as the generation of different possible courses of action (stable models) that can be used to introduce non-determinism in our character's behaviour, a very important feature of these kind of games.

Let us now elaborate on some other features provided by our system, that are usually not present in other approaches. We start with the possibility to have updates to the shopkeeper's knowledge base (KB). For this, let's suppose that the shopkeeper is willing to learn new rules so as long as they are provided by the player with the ring. If rules are sent as events of the form $rule\,(Player, Rule)$, the desired behaviour could be encoded as:

$$assert\,(R) \leftarrow rule\,(P,R)\,, has\_ring\,(P)\,.$$

Let's suppose that Frodo has the ring, represented by $has\_ring\,(frodo)$, and it sends the event $rule\,(frodo, `trust\,(legolas) \leftarrow has\_ring\,(frodo)\,')$. Then, the rule $trust\,(legolas) \leftarrow has\_ring\,(frodo)$ would be asserted in the shopkeeper's KB and, as long as Frodo has the ring, the shopkeeper will trust Legolas. Note that even though the conclusion of this rule contradicts the one of rule (2) above, because Legolas is an Elf, this new rule overrides the previous one and Legolas is in fact trusted. However, if at a later stage $has\_ring\,(frodo)$ becomes false, then the body of this rule is no longer true and $\neg trust\,(legolas)$ becomes true again. Frodo can also say that an entire race should become trusted, for example by sending the event $rule\,(frodo, `trust\,(human)\,')$. Similar requests from other players would be ignored by the shopkeeper. Simultaneous requests from different players can be easily dealt with, just by making them both part of the set of events. Consequently, we can specify effects that depend on such concurrent events.

Some updates can also be internal, just waiting for the right conditions to be met. For example, the shopkeeper could have a behaviour according to which a player would become trusted if it says the right secret word. This could be encoded by the rule (assuming that $msg/2$ are events):

$$assert\,(trust\,(P)) \leftarrow msg\,(P, `secretword')\,.$$

We can even have such assertion be dependent not on a secret word, but on the right sequence of two words:

$$\begin{aligned} assert\,(assert\,(trust\,(P)) \leftarrow msg\,(P, `word2')) \leftarrow \\ msg\,(P, `word1')\,. \end{aligned}$$

Or have the effect of one word be delayed for some time (e.g. three steps):

$$assert\,(assert\,(assert\,(trust\,(P)))) \leftarrow msg\,(P, `w1')\,.$$

It could be important to have a timer to allow for updates to depend on it. Such timer can be encoded in EVOLP with the rules $assert\,(time\,(T+1)) \leftarrow time\,(T)\,.$ and $assert\,(not\, time\,(T)) \leftarrow time\,(T)\,.$,together with the fact $time\,(0)$ in the initial KB. Frodo could tell the shopkeeper that the lembas bread will become deteriorated at time 100 by means of event

$$rule\,(frodo, `assert\,(deter\,(bread)) \leftarrow time\,(100)\,')\,.$$

Note the difference between this event and $rule\,(frodo, `deter\,(bread) \leftarrow time\,(100)\,')\,.$: the

first updates the KB with the fact $deter\,(bread)$ at time 100, causing it to be true from that time onwards, until rejected by a subsequent update, while the second renders $deter\,(bread)$ true at time 100, but false again at time 101.

6 Concluding Remarks

To conclude, we recap some of the features of our system. The use of two forms of negation allows for reasoning with open and closed world assumptions, i.e. the full expressive power of answer-set programming. The stable model based semantics, assigning a set of models at each state, allows for reasoning about possible worlds and allows for the introduction of non-determinism, a very important feature in role-playing games. The inclusion of the selection function allows for meta-reasoning about these possible worlds using any paradigm chosen by the programmer. However, preference based semantics, specifiable in logic programming, can be directly programmed in the agent's EVOLP program. But most important, the evolving features provided by EVOLP allow for the specification of richer NPC in a very simple and declarative way.

Other systems exist that, to some extent, are related to madAgents, such as 3APL [Dastani *et al.*, 2005], Jason [Bordini *et al.*, 2005a] and IMPACT [Subrahmanian *et al.*, 2000]. Each has some specific characteristics that make them more appropriate for some applications, although none has the rich update characteristics of EVOLP and, to our knowledge, none has been applied in the context of role playing games. The notion of goals and plans plays a central role in these systems. MadAgents does not have an explicit goal base nor plan library, although answer set programming can be used for planning purposes, and goals can be represented by integrity constraints without requiring a language extension. Anyway, for the application here described, where MMORPG characters are essentially reactive, this is not such a relevant issue.

MINERVA [Leite, 2003] uses KABUL (Knowledge and Behaviour Update Language) to specify agents' behaviour and their updates. MINERVA is conceptually close to our system in what concerns its knowledge representation mechanism. One of the differences is that MINERVA uses KABUL whereas our system uses EVOLP. Even though KABUL has some extra features that allow for more elaborate forms of social knowledge, EVOLP is a simpler and more elegant language that has all the features required for this application. Furthermore madAgents provides with a full fledged multi-agent platform.

Even though we only presented the stable model based semantics, the architecture allows for the use of a well founded, a three valued, semantics that allows for more efficient, though less expressive, top down proof procedures.

MadAgents effectively separates the notion of reasoning about action from the notion of acting, even though this was not exemplified in detail in this paper. The agent reasons about actions, choosing which ones to execute. These will be executed, resulting (or not) in changes to the environment which can be monitored and may produce new inputs to the agent in the form of events. In parallel, the execution of each action will have an epistemic effect reflected by the (self)-update of the agent's knowledge, in the form of a set of events to be included in the agent's input, which can be seen as the effects of knowing that the action was executed, different from those of the action itself.

The possibility of having several different modes of execution, namely time-driven, event driven, synchronous and asynchronous, allows for the specification of different types of agents. Traditionally, NPC are event-driven. However, as we have briefly illustrated with the example of the passing time, having NPCs that evolve independently of the interaction with other players can bring added value to the game. It provides players with an added sensation that the world is evolving even as they sit still.

The research community working in Computational Logic for Multi-Agent Systems has often been criticized for not carrying their theoretical results to the implementation stage. Even though our platform and architecture are evolving proposals and in constant development, they are fully implemented while enjoying a well defined formal semantics.

References

[Alferes *et al.*, 2002] J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In *JELIA'02*, volume 2424 of *LNAI*. Springer, 2002.

[Assunção *et al.*, 2005] P. Assunção, L. Soares, J. Luz, and R. Viegas. The mule game engine - extending online role-playing games. In *ACM-ITiCSE05*, 2005.

[Bordini *et al.*, 2005a] R. Bordini, J. Hübner, and R. Vieira. Jason and the Golden Fleece of agent-oriented programming. In Bordini et al. [2005b], chapter 1.

[Bordini *et al.*, 2005b] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, 2005.

[Bracciali *et al.*, 2006] A. Bracciali, N. Demetriou, U. Endriss, A. Kakas, W. Lu, and K. Stathis. Crafting the mind of a prosocs agent. *Applied Artificial Intelligence*, 20(4-5), 2006.

[Dastani *et al.*, 2005] M. Dastani, M. B. van Riemsdijk, and J.-J. Ch. Meyer. Programming multi-agent systems in 3APL. In Bordini et al. [2005b], chapter 2.

[Gelfond and Lifschitz, 1990] M. Gelfond and V. Lifschitz. Logic Program with Classical Negation. In *ICLP'90*. MIT, 1990.

[Laird and van Lent, 2000] J. E. Laird and M. van Lent. Human-level AI's killer application: Interactive computer games. In *AAAI'00*. AAAI Press / The MIT Press, 2000.

[Leite, 2003] J. A. Leite. *Evolving Knowledge Bases*. IOS Press, 2003.

[Soares *et al.*, submitted] L. Soares, P. Assunção, J. Luz, and R. Viegas. madAgents - an architecture for logic-based agents, submitted.

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