Enhancing a Multi-Agent System with Evolving Logic Programs

João Leite and Luís Soares

CENTRIA, Universidade Nova de Lisboa, Portugal
jleite@di.fct.unl.pt   luizsoarez@gmail.com

Abstract. This paper reports on a fertile marriage between madAgents, a Java and Prolog based multi-agent platform, and EVOLP, a logic programming based language to represent and reason about evolving knowledge. The resulting platform was implemented using a combination of Java, XSB Prolog and Smodels. The paper begins with an introductory section and a general description of the platform and supported architecture, to progress with the formal semantic characterisation, a small example and details concerning the implementation, to end with a brief comparison with other related systems and some final conclusions.

1 Introduction

This paper reports on what we believe to be a fertile marriage between madAgents (Multimedia Deductive Agents) [24], a Java based multi-agent platform, and EVOLP [2], a computational logic language to represent and reason about evolving knowledge.

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 [25, 12], 3APL [16, 11], Jason [7], DALI [10], ProSOCS [9], FLUX [26] and ConGolog [15], to name a few. For a survey on some of these systems, as well as others, see [22, 8].

While CL, and Logic Programming in particular, can be seen as a good representation language for static knowledge, if we are to move to a more open and dynamic environment, typical of the agency paradigm, we must consider ways and means of representing and integrating knowledge updates from external as well as internal sources. In fact, an agent should not only comprises knowledge about each state, but also knowledge about the transitions between states. The latter may represent the agent's knowledge about the environment's evolution, coupled to its own behaviour and evolution. 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 systems.

To address this issue the paradigm of Evolving Logic Programming (EVOLP) was introduced in [2]. EVOLP generalizes Answer-set Programming [14] 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, for updates arising from the environment. The resulting language, EVOLP, provides a simpler, and more general, formulation of logic program updating, running close to traditional LP doctrine, setting itself on a firm formal basis in which to express, implement, and reason about dynamic knowledge bases, opening up several interesting research topics.

Indeed, EVOLP can adequately express the semantics resulting from successive updates to logic programs, considered as incremental specifications of agents, and whose effect can be contextual. In contradistinction to other approaches, it automatically and appropriately deals, via its update semantics [19], with the possible contradictions arising from successive specification changes and refinements. Furthermore, the EVOLP language can express self-modifications triggered by the evolution context itself, present or future. Additionally, foreseen updates not yet executed can automatically trigger other updates, and moreover updates can be nested, so as to determine change both in the next state and in other states further down an evolution strand.

In this paper, we report on the enhancement of the Java based Multi-Agent Platform madAgents [24] with EVOLP, aiming at serving two objectives:

1 - provide a *de facto* improvement of madAgents, allowing for the implementation of a richer agent architecture where agents' beliefs and behaviour, as well as their evolution, is 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 consistent worlds; a number of extensions such as preferences, revision, abduction, etc - see [27] for more on this subject) on top of which we add all the specific merits of EVOLP for specifying evolving knowledge.

2 - provide a proof of principle that EVOLP can easily be adopted by existing MAS, to represent an evolving belief base, or also to represent the agent's evolving behaviour.

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

2 Multi-Agent Platform

Our implementation is based on the madAgents [24] 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 this platform is to provide a flexible and customisable basis for the development and deployment of multi-agent systems, so that most effort can be focused on 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 semantics of this platform is simple, 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.

The madAgents platform is used extensively in the current version of the MulE Game Engine [5], an architecture for the development of online multiplayer role-playing games with enhanced realism, to implement several different types of agents, some of them more active in the game (monsters, artificial players) while others are more limited in their actions (non-player characters). It is also used in a video annotation server for personalisation purposes [6]. This server uses madAgents-based agents to perform audiovisual personalisation based on the analysis of the video's semantic data associated with video notes, then suggesting a set of preferential videos to the user.

3 Agent Architecture

In this Section we describe the agent architecture. We start by presenting the language of Evolving Logic Programs (EVOLP), and its semantics, as it constitutes the basis for our architecture. Then, we proceed by showing how the language and its semantics can be used to support an architecture for mental agents whose epistemic specification may dynamically change, without loosing its fine theoretical character.

In a nutshell, EVOLP [2] 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. This is in clear contrast with earlier proposals to this effect (e.g. LUPS [4], EPI [13] and KABUL [19]) since EVOLP was designed, above all, with the desire to add as few new constructs to traditional logic 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).

3.1 Language

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 $H\left(r\right) \leftarrow B\left(r\right)$ where $H\left(r\right)$ (dubbed the head of the rule) is a literal and $B\left(r\right)$ (dubbed the body of the rule) is a finite set of literals. A rule with $H\left(r\right) = L_0$ and $B\left(r\right) = \{L_1, \ldots, L_n\}$ 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 $H\left(r\right) = \neg A$, then $\neg H\left(r\right) = A$. 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 $not\, \neg H\left(r\right) \leftarrow B\left(r\right)$ for every rule, in P, of the form $H\left(r\right) \leftarrow B\left(r\right)$, where $H\left(r\right)$ 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 $M \vDash L$, iff $L \in M$, and false otherwise. A default literal $not\, L$ is true in M, denoted by $M \vDash not\, L$, iff $L \notin M$, and false otherwise. A set of literals B is true in M, denoted by $M \vDash B$, iff each literal in B is true in M. An interpretation M of $\mathcal{A}$ is an answer set of a GLP P iff $M' = least\left(\mathbf{P} \cup \{not\, A \mid A \notin M\}\right)$, where $M' = M \cup \{not\_A \mid A \notin M\}$, A is an objective literal, 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.

In order to allow for logic programs to evolve, we first need some mechanism for letting older rules be supervened by more recent ones. That is, we must include a mechanism for deletion of previous knowledge along the agent's knowledge evolution. This can be achieved by permitting default negation not just in rule bodies, as in extended logic programming, but in rule heads as well[21]. Furthermore, we need a way to state that, under some conditions, some new rule should be asserted in the knowledge base[1]. In EVOLP this is achieved by augmenting the language with a reserved predicate $assert/1$, whose sole argument is itself a full-blown rule, so that arbitrary nesting becomes possible. This predicate can appear both as rule head (to impose internal assertions of rules) as well as in rule bodies (to test for assertion of rules). Formally:

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 $\mathcal{A}$[2] is a (possibly infinite) set of generalized logic program rules over $\mathcal{A}_{assert}$.*

Example 1. Examples of EVOLP rules are:

$$assert(not\, a \leftarrow b) \leftarrow not\, c. \qquad a \leftarrow assert(b \leftarrow).$$
$$assert(assert(a \leftarrow) \leftarrow assert(b \leftarrow not\, c), d) \leftarrow not\, e.$$

Intuitively, the first rule states that, if c is false, then the rule $not\, a \leftarrow b$ must be asserted in the agent's knowledge base; the 2nd that, if the fact $b \leftarrow$ is going to be asserted in the agent's knowledge base, then a is true; the last states that, if e is false, then a rule must be asserted stating that, if d is true and the rule $b \leftarrow not\, c$ is going to be asserted then the fact $a \leftarrow$ must be asserted.

This language alone is enough to model the agent's knowledge base, and to cater, within it, for internal updating actions that change it. But self-evolution of a knowledge base is not enough for our purposes. We also want the agent to be aware of events that happen outside itself, and desire the possibility too of giving the agent update "commands" for changing its specification. In other words, we wish a language that allows for influence from the outside, where this influence may be: observation of facts (or rules) that are perceived at some state; assertion commands directly imparting the assertion of new rules on the evolving program. Both can be represented as EVOLP rules: the former by rules without the assert predicate in the head, and the latter by rules with it. Consequently, we shall represent outside influence as a sequence of EVOLP rules:

Definition 2. *Let P be an evolving program over the language $\mathcal{A}$. An* event sequence *over P is a sequence of evolving programs over $\mathcal{A}$.*

3.2 Semantics

In general, we have an EVOLP program describing an agent's initial knowledge base. This knowledge base may already contain rules (with asserts in heads) that describe

[1] Note that asserting a rule in a knowledge base does not mean that the rule is simply added to it, but rather that the rule is used to update the existing knowledge base according to some update semantics, as will be seen below.

[2] Here we extend EVOLP for LPs with both strong and default negation, unlike in [2] where only LP's without strong negation were used.

some forms of its own evolution. Besides this, we consider sequences of events representing observation and messages arising from the environment. Each of these events in the sequence are themselves sets of EVOLP rules, i.e. EVOLP programs. The semantics issue is thus that of, given an initial EVOLP program and a sequence of EVOLP programs as events, to determine what is true and what is false after each of those events.

More precisely, the meaning of a sequence of EVOLP programs is given by a set of *evolution stable models*, each of which is a sequence of interpretations or states. The basic idea is that each evolution stable model describes some possible evolution of one initial program after a given number n of evolution steps, given the events in the sequence. Each evolution is represented by a sequence of programs, each program corresponding to a knowledge state.

The primordial intuitions for the construction of these program sequences are as follows: regarding head asserts, whenever the atom $assert(Rule)$ belongs to an interpretation in a sequence, i.e. belongs to a model according to the stable model semantics of the current program, then $Rule$ must belong to the program in the next state; asserts in bodies are treated as any other predicate literals.

The sequences of programs are treated as in Dynamic Logic Programming [19, 3, 1], a framework for specifying updates of logic programs where knowledge is given by a sequence of logic programs whose semantics is based on the fact that the most recent rules are set in force, and previous rules are valid (by inertia) insofar as possible, i.e. they are kept for as long as they do not conflict with more recent ones. In DLP, default negation is treated as in answer-set programming [14]. Formally, a *dynamic logic program* is a sequence $\mathcal{P} = (P_1, \ldots, P_n)$ of generalized logic programs and its semantic is determined by (c.f. [19, 1] for more details):

Definition 3. *Let $\mathcal{P} = (P_1, \ldots, P_n)$ be a dynamic logic program over language $\mathcal{A}$. An interpretation M is a (refined) dynamic stable model of $\mathcal{P}$ at state s, $1 \leq s \leq n$ iff*

$$M' = least\left(\left[\rho_s\left(\mathcal{P}\right) - Rej_s(M)\right] \cup Def_s(M)\right) \textit{ where:}$$
$$Def_s(M) = \{not\, A \mid \nexists r \in \rho(\mathcal{P}), H(r) = A, M \vDash B(r)\}$$
$$Rej_s(M) = \{r \mid r \in \mathbf{P}_i, \exists r' \in \mathbf{P}_j, i \leq j \leq s, r \bowtie r', M \vDash B(r')\}$$

and A is an objective literal, $\rho_s\left(\mathcal{P}\right)$ denotes the multiset of all rules appearing in the programs $\mathbf{P}_1, ..., \mathbf{P}_s$, and M' and $least(.)$ are as before.

Before presenting the definitions that formalize the above intuitions, let us show some illustrative examples.

Example 2. Consider an initial program P containing the rules $a.$, $assert(not\, a \leftarrow) \leftarrow b.$, $c \leftarrow assert(not\, a \leftarrow).$ and $assert(b \leftarrow a) \leftarrow not\, c.$, and that all the events are empty EVOLP programs. The (only) answer set of P is $M = \{a, assert(b \leftarrow a)\}$ and conveying the information that program P is ready to evolve into a new program $P \oplus P_2$ by adding rule $(b \leftarrow a)$ at the next step, i.e. to P_2. In the only dynamic stable model M_2 of the new program (P, P_2), atom b is true as well as atom $assert(not\, a \leftarrow)$ and also c, meaning that (P, P_2) evolves into a new program (P, P_2, P_3) by adding rule $(not\, a \leftarrow)$ at the next step, i.e. in P_3. This negative fact in P_3 conflicts with the fact in P, and the older is rejected. The rule added in P_2 remains valid, but is no longer useful to conclude b, since a is no longer valid. So, $assert(not\, a \leftarrow)$ and c are also no longer true. In the only dynamic stable model of the last sequence both a, b, and c are false.

This examples does not address external events. The rules that belong to the i-th event should be added to the program of state i, and proceed as in the example above.

Example 3. In the example above, suppose that at state 2 there is an external event with the rules, r_1 and r_2, $assert(d \leftarrow b) \leftarrow a$ and $e \leftarrow$. Since the only stable model of P is $I = \{a, assert(b \leftarrow a)\}$ and there is an outside event at state 2 with r_1 and r_2, the program evolves into the new program obtained by updating P not only with the rule $b \leftarrow a$ but also with those rules, i.e. $(P, \{b \leftarrow a;\ assert(d \leftarrow b) \leftarrow a;\ e \leftarrow\})$. The only dynamic stable model M_2 of this program is $\{b, assert(not\, a \leftarrow), assert(d \leftarrow b), e\}$.

If we keep with the evolution of this program (e.g. by subsequent empty events), we have to decide what to do, in these subsequent states, about the event received at state 2. Intuitively, we want the rules coming from the outside, be they observations or assertion commands, to be understood as events given at a state, that are not to persist by inertia. I.e. if rule r belongs to some set E_i of an event sequence, this means that r was perceived, or received, after $i-1$ evolution steps of the program, and that this perception event is not to be assumed by inertia from then onward. In the example, it means that if we have perceived e at state 2, then e and all its possible consequences should be true at that state. But the truth of e should not persist into the subsequent state (unless e is yet again perceived from the outside). In other words, when constructing subsequent states, the rules coming from events in state 2 should no longer be available and considered. As will become clear below, making these events persistent can be specified in EVOLP.

Definition 4. *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 $\mathcal{A}_{assert}$. 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 5. *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 $(P_1, P_2 \ldots, (P_i \cup E_i))$.*

Notice that the rules coming from the outside do not persist by inertia. At any given step i, the rules from E_i are added and the (possibly various) I_i obtained. This determines the programs P_{i+1} of the trace, which are then added to E_{i+1} to determine the models I_{i+1}. The definition assumes the whole sequence of events given a priori. In fact this need not be so because the events at any given step n only influence the models in the evolution interpretation from n onward:

Proposition 1. *Let $M = (M_1, \ldots, M_n)$ be an evolution stable model of P given a sequence of events $(E_1, E_2, \ldots, E_n)$. Then, for any sets of events $E_{n+1}, \ldots, E_m$ ($m > n$), M is also an evolution stable model of P given $(E_1, \ldots, E_n, E_{n+1}, \ldots, E_m)$.*

EVOLP programs may have various evolution models of given length, or none:

Example 4. Consider P with the following two rules, and 3 empty events:

$$assert(a \leftarrow) \leftarrow not\, assert(b \leftarrow), not\, b. \qquad assert(b \leftarrow) \leftarrow not\, assert(a \leftarrow), not\, a.$$

The reader can check that there are 2 evolution stable models of length 3, each representing one possible evolution of the program after those empty events:

$$M_1 = \langle \{assert(a \leftarrow)\}, \{a, assert(a \leftarrow)\}, \{a, assert(a \leftarrow)\} \rangle$$
$$M_2 = \langle \{assert(b \leftarrow)\}, \{b, assert(b \leftarrow)\}, \{b, assert(b \leftarrow)\} \rangle$$

Since various evolutions may exist for a given length, evolution stable models alone do not determine a truth relation. A truth relation can be defined, as usual, based on the intersection of models:

Definition 6. *Let P be an evolving program, $\mathcal{E}$ an event sequence of length n, both over the language $\mathcal{A}$, and M an interpretation over $\mathcal{A}_{assert}$. M is a* Stable Model of P given $\mathcal{E}$ *iff $(M_1, \ldots, M_{n-1}, M)$ is an evolution stable model of P given $\mathcal{E}$ with length n, for some interpretations $M_1, \ldots, M_{n-1}$. We say that propositional atom A of $\mathcal{A}$ is:* true given $\mathcal{E}$ *iff A belongs to all stable models of P given $\mathcal{E}$;* false given $\mathcal{E}$ *iff A does not belong to any stable models of P given $\mathcal{E}$;* unknown given $\mathcal{E}$ *otherwise.*

A consequence of the above definitions is that the semantics of EVOLP is, in fact, a proper generalization of the answer-set semantics, in the following sense:

Proposition 2. *Let P be a generalized (extended) logic program (without predicate $assert/1$) over a language $\mathcal{A}$, and $\mathcal{E}$ be any sequence with $n \geq 0$ of empty EVOLP programs. Then, M is a stable model of P given $\mathcal{E}$ iff the restriction of M to $\mathcal{A}$ is an answer set of P (in the sense of [14, 21]).*

The possibility of having various stable models after an event sequence is of special interest for using EVOLP as a language for reasoning about possible evolutions of an agent's knowledge base. Like for answer-set programs, we define the notion of categorical programs as those such that, for any given event sequence, no "branching" occurs, i.e. a single stable model exists[3].

Definition 7. *An EVOLP program P is* categorical *given event sequence $\mathcal{E}$ iff there exists only one stable model of P given $\mathcal{E}$.*

3.3 Agent Architecture

We now turn our attention to the agent architecture. Each agent is conceptually divided into three layers as depicted in Figure 1. The Platform Layer deals with all the necessary platform specific protocols such as registration, coordination, control, etc. The Physical Layer is 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. The Mental Layer is 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 presenting details concerning the other two layers.

[3] The definition of conditions over programs and sequences ensuring that a program is categorical is beyond the scope of this paper.

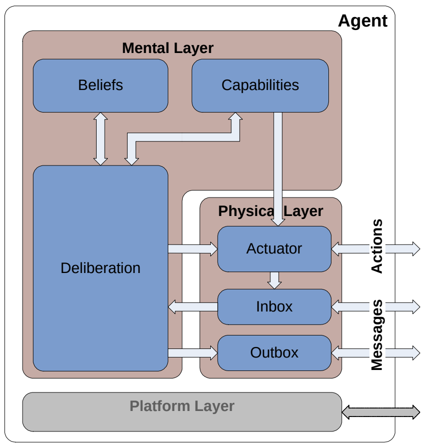


Fig. 1. Agent Architecture

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, 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 [17], implementing some specific semantics. Formally, an agent initial specification is defined as follows:

Definition 8. *Let $\mathcal{N} = \{\alpha_1, ..., \alpha_n\}$ be a set of agent names and $\mathcal{A}$ a propositional language*[4]*. 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 dynamic stable model given a dynamic logic program;*
- *$\mathcal{P}_1 = (P_1)$ 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 definition:

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

[4] Without loss of generality, we are assuming that all agents share a common language.

Definition 9. *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*[5] *$\mathcal{P}_{s+1} = (\mathcal{P}_s, P_{s+1})$, $P_{s+1} = \{r \mid assert(r) \in M_s\}$, and $M_s = Sel(\mathcal{P}_{s-1}, (P_s \cup E_s))$.*

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. The following result relates both views of EVOLP:

Theorem 1. *Let $\langle \alpha_i, \mathcal{C}, Sel(\cdot), (P_1, P_2, \ldots, P_s) \rangle$ be agent α_i at state s resulting from the initial specification $\langle \alpha_i, \mathcal{C}, Sel(\cdot), (P_1) \rangle$ and the sequence of events $(E_1, E_2, \ldots, E_s)$. Let M_s be an interpretation. Then: — If M_s is a dynamic stable model of $(P_1, P_2, \ldots, P_s)$ then M_s is a stable model of P_1 given $(E_1, E_2, \ldots, E_s)$; — If P_1 is* categorical *given the event sequence $(E_1, E_2, \ldots, E_s)$, then, M_s is a stable model of P_1 given $(E_1, E_2, \ldots, E_s)$ iff M_s is a dynamic stable model of $(P_1, P_2, \ldots, P_s)$.*

3.4 Illustrative Example

In this section we present a small illustrative example. In this example we consider two agents, $teacher$ and $student$. The student will want to know how to get a PhD, and will ask the teacher for as along as he doesn't know how. He knows that people that are not smart do not get PhDs. The teacher will tell the student to study, in case the student hasn't told the teacher he has already studied. The student, however, when requested to study, may chose between studying or not. When the student finally tells the teacher that he has studied, then the behaviour of the teacher will be updated, and he will, from then onwards, reply to the student's subsequent requests by teaching him one of two rules, both regarding ways to get a PhD (either by having a good advisor or doing good work). Teaching the rule amounts to sending it as an assertion event to the student agent, which will be used to update the student's beliefs. The tuple $\langle teacher, (\mathcal{A}^t_{\mathcal{C}}, \mathcal{E}^t_{\mathcal{F}}), Sel^t(\cdot), (P^t_1) \rangle$ is the teacher's initial specification, where:

$$
\begin{aligned}
P^t_1 = \{ & tell\_study \leftarrow ask, not\, studied, not\, study. \qquad assert(studied) \leftarrow study.\\
& assert(teach\_r1 \leftarrow not\, teach\_r2,\ ask.) \leftarrow study.\\
& assert(teach\_r2 \leftarrow not\, teach\_r1,\ ask.) \leftarrow study. \}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}^t_{\mathcal{C}} &= \{tell\_study, teach\_r1, teach\_r2\}\\
\mathcal{E}^t_{\mathcal{F}} &= \{ \mathit{effects}(teach\_r1, student, \{assert(phd \leftarrow good\_advisor.)\})\\
&\quad \mathit{effects}(teach\_r2, student, \{assert(phd \leftarrow good\_work.)\})\\
&\quad \mathit{effects}(tell\_study, student, \{tell\_study.\})\}
\end{aligned}
$$

[5] If $\mathcal{P} = (P_1, ..., P_s)$ is a DLP and P a GLP, by $(\mathcal{P}, P)$ we mean the DLP $(P_1, ..., P_s, P)$.

and a non-deterministic random choice function $Sel^t(\cdot)$. The first rule in P_1^t specifies that any event $study$ should make $tell\_study$ true in case there is no evidence that the student has studied (in the past or present). When the teacher receives an event E_i with $study$, $assert(studied)$ is part of the model and $studied$ is used to update the beliefs by becoming part of P_{i+1}^t so that it becomes persistent. Furthermore, the assertions specified by the last two rules also cause an update, rendering their argument rules "active". Any subsequent event E_i with ask is replied to with either $teach\_r1$ or $teach\_r2$ as it generates two stable models, each of which with one of those actions. The initial program P_1^t only defines the mental dimension of actions, i.e., it defines which action to perform but not its concrete effects. This amounts to the clear separation between reasoning about actions and action execution. Note that we can include a rule of the form *effect* $\leftarrow$ $action$ in the program where *effect* is the epistemic effect of deciding to perform $action$. For example, to have the teacher only performs actions $teach\_r1$ and $teach\_r2$ once, we could include the rules $assert(teached\_r1) \leftarrow teach\_r1.$ and $assert(teached\_r2) \leftarrow teach\_r2.$ and add $not\,teached\_r1$ and $not\,teached\_r2$, respectively, to the bodies of the two rules asserted, as follows:

$$teach\_r1 \leftarrow not\,teach\_r2, not\,teached\_r1,\ ask.$$
$$teach\_r2 \leftarrow not\,teach\_r1, not\,teached\_r2,\ ask.$$

Let us now pay attention to the student. $\langle student, (\mathcal{A}_{\mathcal{C}}^s, \mathcal{E}_{\mathcal{F}}^s), Sel^s(\cdot), (P_1^s)\rangle$ is it's initial specification, where $Sel^s(\cdot)$ is a non-deterministic random choice function and

$$\begin{aligned}
P_1^s &= \{study \leftarrow not\,\neg study, tell\_study. \qquad \neg study \leftarrow not\,study, tell\_study.\\
&\qquad not\,phd \leftarrow not\,smart. \qquad ask \leftarrow not\,phd. \qquad good\_advisor.\}\\
\mathcal{A}_{\mathcal{C}}^s &= \{ask, study\}\\
\mathcal{E}_{\mathcal{F}}^s &= \{effects(ask, teacher, \{ask.\}), effects(study, teacher, \{study.\})\}
\end{aligned}$$

We will now illustrate a possible run of this system, assuming a synchronous mode of operation. Initially, the teacher's only stable model is $M_1^t = \{\}$. The student has one dynamic stable model, namely $M_1^s = \{ask, good\_advisor\}$, which has the effect of sending the event $\{ask\}$ to the teacher agent, i.e. $E_2^t = \{ask\}$. At state 2, the teacher will have one dynamic stable model $M_2^t = \{ask, tell\_study.\}$, as he doesn't know the student has studied. This will send the event $\{tell\_study\}$, i.e. $E_3^s = \{tell\_study\}$. Meanwhile, since at state 2 the student still doesn't know how to obtain a PhD, he asks again (we can't say he's very polite), sending the appropriate event to the teacher i.e. $E_3^t = \{ask\}$. At state 3, the teacher will have one dynamic stable model $M_3^t = \{ask, tell\_study.\}$, as he still doesn't know the student has studied, and thus $E_4^s = \{tell\_study\}$. Meanwhile, the student has just received the $tell\_study$ event and has two dynamic stable models, $M_3^s = \{tell\_study, ask, study\}$ and $M_3^{s\prime} = \{tell\_study, ask, \neg study\}$. If the student chooses $M_3^{s\prime}$, this story will just go on as before. Eventually, the student will choose M_3^s, sending the events $E_4^t = \{study, ask\}$ to the teacher. At state 4, the teacher has one dynamic stable model $M_4^t = \{ask, study, assert(studied), assert(teach\_r2 \leftarrow not\,teach\_r1,\ ask.), assert(teach\_r1 \leftarrow not\,teach\_r2,\ ask.)\}$. This will make the rule $teach\_rule1 \leftarrow not\,teach\_r2,\ ask.$, the fact $studied.$, and the rule $teach\_rule2 \leftarrow not\,teach\_r1,\ ask.$ belong to P_5^t, and will not send any events to the student since none of the elements of M_4^t is an action of the teacher.

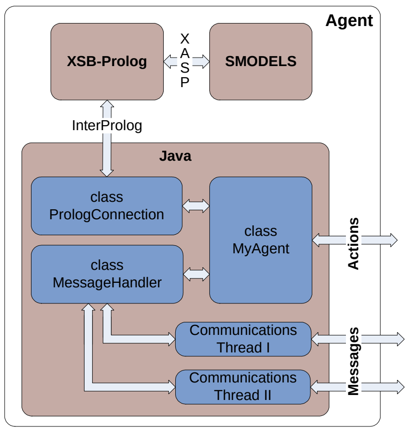
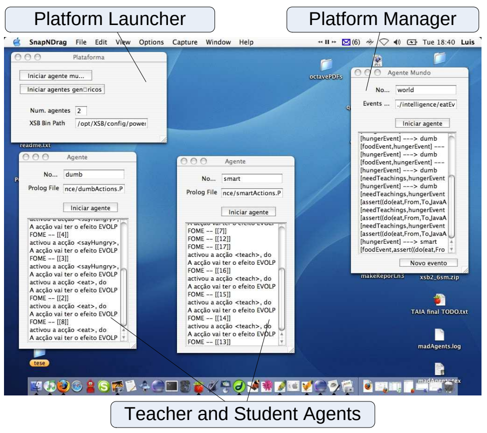


Fig. 2. Agent Implementation Architecture



Num. agentes 2
XSB Bin Path /opt/XSB/config/powe

No... dumb
Prolog File nce/dumbActions.P

```
A acção vai ter o efeito EVOLP
FOME -- [[4]]
activou a acção <sayHungry>,
A acção vai ter o efeito EVOLP
FOME -- [[3]]
activou a acção <sayHungry>,
A acção vai ter o efeito EVOLP
activou a acção <eat>, do
A acção vai ter o efeito EVOLP
FOME -- [[2]]
activou a acção <eat>, do
A acção vai ter o efeito EVOLP
FOME -- [[8]]
activou a acção <eat>, do
A acção vai ter o efeito EVOLP
```

No... smart
Prolog File nce/smartActions.P

```
FOME -- [[7]]
FOME -- [[12]]
FOME -- [[17]]
activou a acção <teach>, do
A acção vai ter o efeito EVOLP
FOME -- [[16]]
activou a acção <teach>, do
A acção vai ter o efeito EVOLP
FOME -- [[15]]
activou a acção <teach>, do
A acção vai ter o efeito EVOLP
FOME -- [[14]]
activou a acção <teach>, do
A acção vai ter o efeito EVOLP
FOME -- [[13]]
```

No... world
Events ... ./intelligence/eatEv

```
[hungerEvent] ---> dumb
[foodEvent,hungerEvent] ---
[hungerEvent] ---> dumb
[foodEvent,hungerEvent] ---
[hungerEvent] ---> dumb
[needTeachings,hungerEvent
[hungerEvent] ---> dumb
[needTeachings,hungerEvent
[assert((do(eat,From,To,JavaA
[needTeachings,hungerEvent
[assert((do(eat,From,To,JavaA
[needTeachings,hungerEvent
[assert((do(eat,From,To,JavaA
[hungerEvent] ---> smart
[foodEvent,assert((do(eat,Fro
```

Fig. 3. Graphical User Interface

To cut this story short, after the next ask by the student, the teacher will have two dynamic stable models (of the DLP $(P_1^t, ..., P_5^t)$ where $P_2^t = P_3^t = P_4^t = \emptyset$), one with $teach\_rule1$ true and another with $teach\_rule2$ true. The teacher's selection function chooses one of them and either executes $teach\_rule1$ or $teach\_rule2$, causing either of $assert\,(phd \leftarrow good\_advisor.)$ and $assert\,(phd \leftarrow good\_work.)$ to be sent to the student. If the student receives the event $assert\,(phd \leftarrow good\_advisor.)$, he will stop asking because he can conclude phd, even though $not\,smart$ is still true, since the newly accquired rule will serve, as per the DLP semantics, to reject the rule $not\,phd \leftarrow not\,smart$. As long as the teacher keeps sending the events $assert\,(phd \leftarrow good\_work.)$, since the student cannot prove $good\_work$ (besides impolite he is also lazy), he will keep asking the teacher until the teacher eventually selects the action $teach\_rule1$, sends $assert\,(phd \leftarrow good\_advisor.)$, and the student will stop bothering the teacher.

4 Implementation

In this Section we briefly elaborate on some issues related to the implementation of a generic agent. As depicted in Figure 2, 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. Interfacing between the Java and Prolog components is handled by InterProlog. The Java component also provides a Graphical User Interface, shown in Figure 3. 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 agent's, in the platform manager or encode outside events. Communication is implemented so as to provide a transparent message exchange mechanism to the agents, where they only have to call methods to add messages to their inbox or outbox and the `MessageHandler` class does 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 [19]; 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;

5. Determine the next program in the trace and cycle again. Since the transformation mentioned above is incremental, the agent effectively keeps the transformed program adding just the rules corresponding to the new part of the trace. The required new methods added to the platform were encapsulated in the `PrologConnection` class. This class now includes methods for setting up the Prolog environment, calling non-deterministic goals, and calculating the new EVOLP trace and program transformation.

5 Related Work

Other systems exist that, to some extent, are related to the one we presented. Each has some specific characteristics that make them more appropriate for some applications, although none has the rich update characteristics of EVOLP.

3APL [16, 11] is a logic based programming language for implementing agents with beliefs, goals, and plans as mental attitudes, that can generate and revise plans to achieve goals, and are capable of interaction. Actions can be external, communication, or internal mental ones, resembling actions and their effects in our system. 3APL supports the integration of Prolog and Java. 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 or, as explored elsewhere in [23], by means of DLP to obtain extra expressivity. 3APL lacks the ability to deal with updates such as our system.

Jason [7] is a Java written interpreter for an extended version of AgentSpeak(L), a BDI-styled logic-based agent-oriented programming language. It provides features such as speech-act based communication, customisable selection and trust functions, and overall agent architecture (perception, belief-revision, inter-agent communication, and acting), and easy extensibility through user-defined "internal actions". A detailed comparison with Jason is subject for future work. A first impression shows that both systems use default negation for knowledge representation and, as others, Jason does not provide the ability to deal with updates such as our system.

IMPACT [25, 12] is a system developed with the main purpose of providing a framework to build agents on top of heterogeneous sources of knowledge. It provides the notion of an agent program written over a language of so-called code-calls, encapsulations of whatever the legacy code is. Code-calls are used in clauses, that form agent programs, determining constraints on the actions that are to be taken by agents. Agent programs and their semantics resemble logic programs extended with deontic modalities. The semantics is given by the notion of a rational status sets, which are similar to the notion of stable models used in our system. While IMPACT is at a far more advanced stage of development than our system, we believe that our system, namely due to its use of EVOLP, has the potential to meet IMPACT standards, while providing an added value in what concerns the ability to represent evolutions.

MINERVA [20, 19] uses KABUL (Knowledge and Behaviour Update Language) to specify agents' behaviour and their updates. MINERVA is conceptually close to our system concerning its knowledge representation mechanism. One difference 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.

DALI [10] is an Active Logic Programming Language, somehow related to MINERVA and our present system, designed for executable specification of logical agents. It uses plain Horn Clauses and its semantics is based on Least Herbrand Models. Three main differences are noticed when comparing our system with DALI: our semantics is based on ASP, with default negation while DALI uses plain Horn Clauses and the semantics is based on Least Herbrand Models; DALI is implemented in PROLOG while our system uses a combination of Java, XSB and Smodels; DALI doesn't allow for evolving specifications such as the ones provided by EVOLP.

6 Concluding Remarks

In this paper we presented a platform and architecture that resulted from combining madAgents with EVOLP.

Being a paper that describes the theory of the underlying the system, its implementation, and some examples, it necessarily lacks some deeper explanations concerning other important issues. Some of the issues that were left out concern open problems and a deeper comparison with the other related systems, while others refer to issues that we addressed, often representing non-trivial challenges, such as for example the details concerning the implementation of the transformation into a program to be sent to SMODELS because of its restricted use of variables. However, we had to leave something out and this is the result of our choice. Before we end, in this Final Section, we re-elaborate on some of the main points of our system.

Using default negation allows for reasoning with incomplete information, important in open environments. In this paper, we additionally extended EVOLP to also allow for strong negation. By combining both forms of negation, the programmer can obtain the full expressivity of answer-set programming.

The stable model based semantics, assigning a set of models at each state, allows for reasoning about possible worlds. 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.

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

The architecture effectively separates the notions of reasoning about actions and 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, as well as incorporating different coordination policies, specifiable in logic programming.

The specification language EVOLP, being based on logic programs with the stable model semantics (aka. answer set programming), and properly extending these, directly supports the representation, specification and reuse of the grand number of results produced by those working in answer-set programming for knowledge representation and reasoning (see [27] for references).

Most of all, we believe to have achieved both main goals i.e. to provide an improvement of madAgents, allowing for the implementation of a richer agent architecture where agents' beliefs and behaviour, as well as their evolution, is specifiable in EVOLP, and to provide a proof of principle that EVOLP can easily be adopted by existing MAS, to represent an evolving belief base, or also to represent the agent's evolving behaviour.

Researchers working in computational logic for multi-agent systems, have 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.

We are currently working on several directions which, for lack of space, we cannot elaborate on. However, on the application side, we would like to mention that we are re-designing the MulE Game Engine [5] to benefit from our enhanced system. Initial results can be found in [18]. On the development side, this are just the initial results of a constantly evolving ongoing project which we believe to be very promising.

References

1. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1), 2005.

2. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In *JELIA'02*, volume 2424 of *LNAI*. Springer, 2002.
3. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 45(1-3), 2000.
4. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS : A language for updating logic programs. *Artificial Intelligence*, 138(1-2), 2002.
5. P. Assunção, L. Soares, J. Luz, and R. Viegas. The mule game engine - extending online role-playing games. In *ACM-ITiCSE05*, 2005.
6. M. Boavida, S. Cabaço, N. Folgôa, F. Mourato, F. Sampayo, and A. Trabuco. Video based tools for sports training. In *IACSS'2005*, 2005.
7. R. Bordini, J. Hübner, and R. Vieira. Jason and the Golden Fleece of agent-oriented programming. In Bordini et al. [8], chapter 1.
8. 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.
9. 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.
10. S. Costantini and A. Tocchio. A logic programming language for multi-agent systems. In *JELIA'02*, volume 2424 of *LNAI*. Springer, 2002.
11. M. Dastani, M. B. van Riemsdijk, and J.-J. Ch. Meyer. Programming multi-agent systems in 3APL. In Bordini et al. [8], chapter 2.
12. J. Dix and Y. Zhang. IMPACT: a multi-agent framework with declarative semantics. In Bordini et al. [8], chapter 3.
13. T. Eiter, M. Fink, G. Sabbatini, and H Tompits. A framework for declarative update specifications in logic programs. In *IJCAI'01*. Morgan-Kaufmann, 2001.
14. M. Gelfond and V. Lifschitz. Logic Program with Classical Negation. In *ICLP'90*. MIT, 1990.
15. G. De Giacomo, Y. Lesprance, and H.J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(109-169), 2000.
16. K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Int. J. of Autonomous Agents and Multi-Agent Systems*, 2(4), 1999.
17. R. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):391–419, 1999.
18. J. Leite and L. Soares. Evolving characters in role playing games. In *AT2AI@EMCSR'06*, 2006. to appear.
19. J. A. Leite. *Evolving Knowledge Bases*. IOS Press, 2003.
20. J. A. Leite, J. J. Alferes, and L. M. Pereira. Minerva - a dynamic logic programming agent architecture. In *Intelligent Agents VIII*, volume 2333 of *LNAI*. Springer, 2002.
21. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In *KR'92*. Morgan-Kaufmann, 1992.
22. V. Mascardi, M. Martelli, and L. Sterling. Logic-based specification languages for intelligent software agents. *Theory and Practice of Logic Programming*, 4(4), 2004.
23. V. Nigam and J. Leite. Using dynamic logic programming to obtain agents with declarative goals. submitted, 2006.
24. L. Soares, P. Assunção, J. Luz, and R. Viegas. Madagents - an architecture for logic-based agents. submitted, 2006.
25. V. S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press, 2000.
26. M. Thielscher. *Reasoning Robots: The Art and Science of Programming Robotic Agents*. Springer, 2005.
27. Working group on Answer-Set Programming. http://wasp.unime.it.