Environment-aware computations via program updates

J. J. Alferes[1], A. Brogi[2], J. A. Leite[1], and L. M. Pereira[1]

[1] Centro de Inteligência Artificial, Universidade Nova de Lisboa, Portugal
[2] Department of Computer Science, University of Pisa, Italy

Abstract. We show how different behaviours of environment-aware agents can be naturally specified in terms of logic program updates. The obtained declarative semantics sets a firm basis for the development and the analysis of environment-aware agents.

1 Introduction

The paradigm shift from stand-alone isolated computing to environment-aware computing clearly indicates that the ability of reacting to changes occurring in the external environment is a crucial capability of software agents. Reactivity must however be suitably reconciled with rationality. Indeed the ability of an agent to reason on available information is as important as its ability to promptly react to sudden changes occurring in the external environment.

The way in which an agent combines rationality and reactivity determines the quality of the services offered by the agent. Consider for instance a software agent whose task is to recommend investments based on the analysis of trends in the stock market [6]. A scarcely reactive behaviour may generate well-evaluated recommendations based on outdated information, while a scarcely rational behaviour may quickly generate recommendations based only on the most recently acquired information.

While developing environment-aware agents, the environment in which the agents will operate is at least partially unknown. Typically, even if the set of possible observable behaviours of the environment is known, the precise dynamic behaviour of the environment is not predictable at software development time. On the other hand, the availability of a well-founded description of the possible behaviours of environment-aware programs is crucial for performing tasks such as verification and analysis before putting the program at work with the external environment.

The goal of this paper is to provide a formal characterization of the behaviours of environment-aware agents. Our approach can be summarized as follows:

- We consider an agent to be *environment-aware* if it is capable of reacting to changes occurring in the external environment. As the environment may dynamically change while the agent is performing its computations, such changes may influence the agent behaviour.

- Agents have a partial representation of the external environment, represented by their *perceptions* of the environment. The type of such perceptions of course depends on the sensing capabilities owned by the agent. We will focus on the way in which the behaviour of an agent may be influenced by its perceptions, rather than on the way in which the agent will get such perceptions. For instance, we will abstract from the way in which a software agent accesses some piece of information available in the external environment (e.g., by receiving a message, by downloading a file, or by getting data from physical sensors). Formally, if we denote by $percs(P)$ the set of possible perceptions of an agent P, the set $\mathcal{E}$ of all possible environment configurations can be defined as $\mathcal{E} \subseteq \mathcal{P}(percs(P))$, that is, as the set of all possible sets of perceptions of the environment that P may (simultaneously) get.
- We choose *logic programming* as the specification language of environment-aware agents. We show that the computation of a program P that reacts to a sequence of environment configurations $\langle E_1, E_2, \ldots E_n \rangle$ can be naturally modelled by means of a *Dynamic Logic Program* (DLP)[1], that is, by a sequence $Q_0 \oplus Q_1 \oplus Q_2 \oplus \ldots \oplus Q_n$ of (generalized) logic programs [1] whose semantics defines the effects of first updating Q_0 with Q_1, then updating the result with Q_2, and so on.
- From a programming perspective, we show that the environment-aware behaviours of a program reacting to sequences of environment configurations can be specified by a set of LUPS [2] rules that program the way in which the knowledge of the program will be updated by a sequence of environment configurations. More precisely, the behaviour of a program P that reacts to the sequence of environment configurations $\langle E_1, E_2, \ldots E_n \rangle$ is described by the sequence of LUPS updates: $P \otimes E_1 \otimes E_2 \otimes \ldots \otimes E_n$ where each E_i is a set of (temporary) updates representing an environment configuration.
- The formal semantics of LUPS (with the modification of [19]) is defined in terms of a program transformation, by first transforming a sequence of LUPS updates into a DLP, and this DLP into a generalized logic program:

$$\begin{aligned} Sem(P \otimes E_1 \otimes E_2 \otimes \ldots \otimes E_n) &= SM(\tau(\Upsilon(P \otimes E_1 \otimes E_2 \otimes \ldots \otimes E_n))) \\ &= SM(\tau(Q_0 \oplus Q_1 \oplus Q_2 \oplus \ldots \oplus Q_n)) \\ &= SM(G) \end{aligned}$$

  where Υ is the mapping from LUPS to DLP, τ is the mapping from DLP to generalized logic programs, and where SM denotes the set of stable models [11] of a generalized logic program.

It is important to observe that the LUPS language features the possibility of programming different types of updates. We will show how this very feature can be actually exploited to specify different environment-aware behaviours of programs. We will also show how the declarative semantics of LUPS programs provides a formal characterization of environment-aware behaviours, which can be exploited for resource-bounded analyses of the possible behaviours of a program w.r.t. a set $\mathcal{E}$ of possible environment configurations.

2 LUPS: A language for dynamic updates

In this section we briefly present Dynamic Logic Programming (DLP) [1], and the update command language LUPS [2][1].

The idea of Dynamic Logic Programming is simple and quite fundamental. Suppose that we are given a sequence of generalized logic program (i.e., programs possibly with default negation in rule heads) modules $P_1 \oplus \cdots \oplus P_n$. Each program P_s $(1 \leq s \leq n)$ contains knowledge that is given as valid at state s. Different states may represent different time instants or different sets of knowledge priority or perhaps even different viewpoints. Consequently, the individual program modules may contain mutually contradictory as well as overlapping information. The role of DLP is to use the mutual relationships existing between different sequentialized states to precisely determine, at any given state s, the *declarative* as well as the *procedural* semantics of the combined program, composed of all modules. The declarative semantics at some state is determined by the stable models of the program that consists of all those rules that are "valid" in that state. Intuitively a rule is "valid" in a state if either it belongs to the state or belongs to some previous state in the sequence and is not rejected (i.e., it is inherited by a form of non-monotonic inertia). A rule r from a prior state is rejected if there is another conflicting rule (i.e., a rule with a true body whose head is the complement of the head of r) in a subsequent state. A transformational semantics into generalized program, that directly provides a means for DLP implementation, has also been defined.

LUPS [2] is a logic programming command language for specifying logic program updates. It can be viewed as a language that declaratively specifies how to construct a Dynamic Logic Program. A sentence U in LUPS is a set of simultaneous update commands that, given a pre-existing sequence of logic programs, whose semantics corresponds to our knowledge at a given state, produces a new DLP with one more program, corresponding to the knowledge that results from the previous sequence after performing all the simultaneous update commands.

A program in LUPS is a sequence of such sentences, and its semantics is defined by means of a dynamic logic program generated by the sequence of commands. In [2], a translation of a LUPS program into a generalized logic program is also presented, where stable models exactly correspond to the semantics of the original LUPS program.

LUPS update commands specify assertions or retractions to the current program. In LUPS a simple assertion is represented by the command:

$$\textbf{assert } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{1}$$

meaning that if $L_{k+1}, \ldots, L_m$ is true in the current program, then the rule $L \leftarrow L_1, \ldots, L_k$ is added to the new program (and persists by inertia, until possibly retracted or overridden by some future update command, by the addition of a

[1] All the formal definitions can be found in [1, 2]. Both papers, together with the implementations of DLP and LUPS, and the Lift Controller example below, are available from `http://centria.di.fct.unl.pt/~jja/updates/`

rule with complementary head and true body). To represent rules and facts that do not persist by inertia, i.e. that are one-state only persistent, LUPS includes the modified form of assertion:

$$\textbf{assert event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{2}$$

The retraction of rules is performed with the two update commands:

$$\textbf{retract } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{3}$$
$$\textbf{retract event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{4}$$

meaning that, subject to precondition $L_{k+1}, \ldots, L_m$ (verified at the current program) rule $L \leftarrow L_1, \ldots, L_k$ is either retracted from its successor state onwards, or just temporarily retracted in the successor state (if governed by **event**).

Normally assertions represent newly incoming information. Although its effects may persist by inertia (until contravened or retracted), the assert command itself does not persist. However, some update commands may desirably persist in the successive consecutive updates. This is the case of, e.g., laws which subject to preconditions are always valid, rules describing the effects of an action, or, as we shall see, rules describing behaviours of environment-aware programs. For example, in the description of the effects of actions, the specification of the effects must be added to all sets of updates, to guarantee that, whenever the action takes place, its effects are enforced. To specify such persistent update commands, LUPS introduces the commands:

$$\textbf{always } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{5}$$
$$\textbf{always event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{6}$$
$$\textbf{cancel } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{7}$$

The first two commands state that, in addition to any new set of arriving update commands, the persistent update command keeps executing along with them too. The first case without, and the second case with, the **event** keyword. The third statement cancels the execution of this persistent update, once the conditions for cancellation are met.

3 Programming environment-aware behaviours

We will now show how different environment-aware behaviours can be programmed in LUPS. We will start by considering the environment-aware behaviours that have been analysed in [6]. Therein different environment-aware behaviours are formally defined and compared to one another. Agents are specified by definite logic programs, and perceptions are (positive) atoms. Namely an environment configuration is simply a set of (positive) atoms. Environment-aware behaviours are defined by extending the standard bottom-up semantics of definite logic programs, defined in terms of the immediate consequence operator $T(P)$ [9]. The idea is to model the environment-aware behaviour of a definite program P by means of an operator $\varphi(P)(I,E)$ which given a Herbrand interpretation I (representing the partial conclusions of the program so far) and a Herbrand interpretation E (representing one set of environment perceptions) returns the new set of conclusions that the program P is able to draw. Different definitions of $\varphi(P)$ are analysed and compared to one another in [6]:

$$\begin{array}{rlr} \tau_i(P)(I,E) = & I \cup T(P)(I \cup E) & \textit{(uncontrolled) inflationary} \\ \tau_{\omega i}(P)(I,E) = & \tau_i^{\omega}(P)(I,E) & \textit{controlled inflationary} \\ \tau_n(P)(I,E) = & T(P)(I \cup E) & \textit{(uncontrolled) non-inflationary} \\ \tau_{\omega n}(P)(I,E) = & \tau_n^{\omega}(P)(I,E) & \textit{controlled non-inflationary} \end{array}$$

The behaviour expressed by $\tau_i(P)$ is called *inflationary* as the $\tau_i(P)$ operator is inflationary on its first argument I. Intuitively speaking, every previously reached conclusion is credulously maintained by $\tau_i(P)$. The operator $\tau_{\omega i}(P)$ expresses a *controlled* behaviour as the program P reacts to the changes occurred in the external environment only after terminating the internal computation triggered by the previous perceptions. The operators $\tau_n(P)$ and $\tau_{\omega n}(P)$ model the corresponding behaviours for the non-inflationary case. These different definitions of $\varphi(P)$ model environment-aware behaviours which differ from one another in the way they combine rationality and reactivity aspects. More precisely, as shown in [6], they differ from one another in their degree of credulousness (or skepticism) w.r.t. the validity of the information perceived from the environment and in the conclusions derived thereafter.

In this paper, we show how persistent LUPS updates can be used to naturally program different forms of environment-aware behaviours. In this setting, environment evolution is described by updates asserting new events that state that perceptions (non-persistently) become true or false:

$$\textbf{assert event } E_i$$

where E_i is a literal. Roughly speaking, the four environment-aware behaviours considered in [6] can be programmed by the following LUPS updates:

$$\begin{array}{r} (\tau_i)\ \textbf{always } L \textbf{ when } \overline{L}_P, \overline{L}_E \\ (\tau_{\omega i})\ \textbf{always } L \leftarrow \overline{L}_P \textbf{ when } \overline{L}_E \\ (\tau_n)\ \textbf{always event } L \textbf{ when } \overline{L}_P, \overline{L}_E \\ (\tau_{\omega n})\ \textbf{always event } L \leftarrow \overline{L}_P \textbf{ when } \overline{L}_E \end{array}$$

where $\overline{L}_E$ denotes a conjunction of environment perceptions, and $\overline{L}_P$ denotes a (possibly empty) conjunctions of program-defined literals.

The rule for (τ_i) states that, whenever the events $\overline{L}_E$ occur, if the literals in $\overline{L}_P$ were already true then L is added as a fact and remains true by inertia until overridden, thus modelling an inflationary behaviour. By having the **event** keyword, in rule (τ_n) L is added only in the following state, and then removed. Consequently, the behaviour modelled by this rule is non-inflationary as the truth of L does not remain by inertia. In $(\tau_{\omega i})$ rather than testing for the truth of $\overline{L}_P$ in the previous state and adding L, the logic program rule $L \leftarrow \overline{L}_P$ is asserted. This allows the conclusion L, and any other conclusion depending on

L via other rules with the same behaviour, to be reached in the same single state after the occurrence of $\overline{L}_E$. This way all the conclusions are obtained before any other change in the external environment is analysed, as desired in the controlled behaviour. The behaviour modelled by $(\tau_{\omega n})$ is similar, but the rule is only added in the following state and then removed, so as to model a non-inflationary behaviour. Also note that all these behaviours are modelled via persistent update commands. Indeed, e.g. in (τ_i), we want L to be added whenever the pre-conditions are met, and not only tested once, as it would happen if an **assert** command would be introduced instead.

Because of space limitations, rather than providing a deeper analysis of each of the behaviours specified by the rules above, including the proof on the equivalence to the behaviours specified in [6], we will present a single detailed example where all these behaviours occur. The example also illustrates a (limited) use of default negation, in that a single stable model exists for each state.

3.1 Example: A lift controller

Consider an agent that is in charge of controlling a lift. The available perceptions are sets made up from the predicates $push(N)$ and $floor$. Intuitively, $floor$ means that the agent receives a signal from the lift indicating that a new floor has been reached. $push(N)$ signifies that a lift button to go to floor N was just pushed, whether the one inside the lift or the one at the floor.

Upon receipt of a $push(N)$ signal, the lift records that a request for going to floor N is pending. This can easily be modelled by an inflationary (uncontrolled) rule. It is inflationary because the request remains registered in subsequent states (until served). It is uncontrolled because, in our example, the request is not handled immediately. In the LUPS language[2]:

$$\textbf{always } \ request(F) \textbf{ when } push(F) \tag{8}$$

Based on the pending requests at each moment, the agent must prefer where it is going:

$$\textbf{always } \ going(F) \leftarrow preferredReq(F) \tag{9}$$

$$\textbf{always } \ preferredReq(F) \leftarrow request(F), not\ unpreferred(F) \tag{10}$$

$$\textbf{always } \ unpreferred(F) \leftarrow request(F2), better(F2, F) \tag{11}$$

Note that these rules reflect a controlled inflationary behaviour. This is so because the decision of where to go must be made immediately, i.e., before other perceptions take place. The predicate $better/2$ can be programmed with rules with a controlled inflationary behaviour, according to some preference criterion. For example, if one wants to say that the preferred request is the one for going to the closest floor, one may write:

$$\textbf{always } \ better(F1, F2) \leftarrow at(F),\ \mid F1 - F \mid < \mid F2 - F \mid \tag{12}$$

[2] In the sequel all rules with variables stand for their ground instances, and operations (sums and subtractions) simply restrict those instances.

The internal predicate $at/1$ stores, at each moment, the number of the floor where the lift is. Thus, if a $floor$ signal is received, depending on where the lift is going, the $at(F)$ must be incremented/decremented[3]:

$$\textbf{always event}\ \ at(F+1)\ \textbf{when}\ floor, at(F), going(G),\ G > F \qquad (13)$$

$$\textbf{always event}\ \ at(F-1)\ \textbf{when}\ floor, at(F), going(G),\ G < F \qquad (14)$$

Since the floor in which the lift is at changes whenever new $floor$ signals come in, these rules are modelled with a non-inflationary behaviour. To guarantee that the floor in which the lift is at does not change unless a floor signal is received, the following non-inflationary rule is needed:

$$\textbf{always event}\ \ at(F)\ \textbf{when}\ not\ floor, at(F) \qquad (15)$$

When the lift reaches the floor to which it was going, it must open the door (with a non-inflationary behaviour, to avoid tragedies). After opening the door, it must remove the pending request for going to that floor:

$$\textbf{always event}\ \ opendoor(F)\ \textbf{when}\ going(F), at(F) \qquad (16)$$

$$\textbf{always}\ \ not\ request(F)\ \textbf{when}\ going(F), at(F) \qquad (17)$$

To illustrate the behaviour of this program, consider now that initially the lift is at the 5th floor, and that the agent receives a sequence of perceptions, starting with $\{push(10), push(2)\}$, and followed by $\{floor\}$, $\{push(3)\}$, $\{floor\}$. This is modelled by the LUPS program $(P \cup E_0) \otimes E_1 \otimes E_2 \otimes E_3 \otimes E_4$, where P is the set of commands (8)-(17), E_0 comprises the single command **assert event** $at(5)$ (for the initial situation), and each other E_i contains an assert-event command for each of the elements in the corresponding perception.

According to LUPS semantics, at E_1 both $push(10)$ and $push(2)$ are true. So at the moment of E_2 both requests (for 2 and 10) become true, and immediately (i.e., before receiving any further perception) $going(2)$ is determined (by rules (9)-(12)). Note the importance of these rules having a controlled behaviour for guaranteeing that $going(2)$ is determined before another external perception is accepted. At the moment of E_3, $at(4)$ becomes true by rule (14), since $floor$ was true at E_2, and, given that the rules for the at predicate are all non-inflationary, the previous $at(5)$ is no longer true. Both request facts remain true, as they where introduced by the inflationary rule (8). Since $push(3)$ is true at E_3, another request (for 3) is next also true. Accordingly, by rules (9)-(12), $going(3)$ becomes then true (while $going(2)$ is no longer true). Moreover, by rule (14), $at(3)$ is next true. It is easy to check that given the $floor$ signal at E_4, in the subsequent state $request(3)$ becomes false (by rule (17)), and $opendoor(3)$ becomes true and in any next state becomes false again (since rule (16) is non-inflationary). Note

[3] Because of space limitations, we do not give here the full specification of the program, where lift movements are limited within a top and a ground floor. This can however be done by suitably constraining the rules defining $at/1$ by means of a $topFloor/1$ and a $groundFloor/1$ predicates.

that the falsity of $request(3)$ causes $going(2)$ to be true again, so that the lift will continue its run.

Finally, note that the behaviour of the lift controller agent can be extended so as to take into account emergency situations. For instance, the update rule:

$$\textbf{always event } opendoor(F) \leftarrow at(F) \textbf{ when } firealarm \tag{18}$$

tells the agent to open the door when there is a fire alarm (perception) independently of whether the current floor was the planned destination. Note that this rule reflects a controlled non-inflationary behaviour and refines the agent behaviour defined by rule (16) only in the case of an emergency.

4 Reasoning about environment-aware behaviours

The declarative semantics of LUPS provides a formal characterization of environment-aware behaviours, which can be exploited for resource-bounded analyses of the possible behaviours of a program w.r.t. a set $\mathcal{E}$ of possible environment configurations. For instance the states that a program P may reach after reacting to a sequence of n environment configurations are formally characterized by:

$$\Phi(P, \mathcal{E}, n) = \bigcup_{E_{i_k} \in \mathcal{E}} Sem(P \otimes E_{i_1} \otimes E_{i_2} \otimes \ldots \otimes E_{i_n})$$

Namely, $\Phi(P, \mathcal{E}, n)$ is a set of stable models that denotes all the possible states that P can reach after reacting n times to the external environment.

We define the notion of *beliefs* of an environment-aware program as the largest set of (positive and negative) conclusions that the program will be certainly able to draw after n steps of computation, for whatever sequence of environment configurations[4]:

$$\mathcal{B}(P, \mathcal{E}, n) = \left(\bigcap_{M \in \Phi(P, \mathcal{E}, n)} M^+ \right) \cup \left(\bigcap_{M \in \Phi(P, \mathcal{E}, n)} M^- \right)$$

Notice that in general $\mathcal{B}(P, \mathcal{E}, n)$ is a partial interpretation.

We now introduce the notion of *invariant* for environment-aware programs. The invariant of a conventional program defines the properties that hold at each stage of the program computation. Analogously, the invariant of an environment-aware program defines the largest set of conclusions that the program will be able to draw at any time in any environment. A resource-bounded characterization of the invariant of an environment-aware program after n steps of computation can be formalized as follows, where in general $\mathcal{I}(P, \mathcal{E}, n)$ is again partial:

$$\mathcal{I}(P, \mathcal{E}, n) = \left(\bigcap_{M \in \mathcal{B}(P, \mathcal{E}, i), i \in [1,n]} M^+ \right) \cup \left(\bigcap_{M \in \mathcal{B}(P, \mathcal{E}, i), i \in [1,n]} M^- \right)$$

[4] We denote by M^+ and M^- the positive part and the negative part of a (possibly partial) interpretation M. Formally: $M^+ = M \cap HB$ and $M^- = \{not\ A \mid A \in M\}$.

5 Related work

The use of computational logic for modelling single and multi-agent systems has been widely investigated (e.g., see [21] for a quite recent roadmap). Two approaches stand closer to our own. The agent-based architecture described in [18] aims at reconciling rationality and reactivity. Agents are logic programs which continuously perform an "observe-think-act" cycle, and their behaviour is defined via a proof procedure which exploits iff-definitions and integrity constraints. Different action languages [12, 13] have been proposed to describe and reason on the effects of actions. Intuitively, while action languages and LUPS are both concerned with modelling changes, action languages focus on the notions of causality and fluents, while LUPS focusses its features on declarative updates for general knowledge bases (see [2] for a thorough discussion of the relation between the two approaches).

AgentSpeak(L) [20] is a logical language for programming Belief-Desire-Intention (BDI) agents, originally designed by abstracting the main features of the PRS and dMARS systems [17]. Our approach shares with AgentSpeak(L) the objective of using a simple logical specification language to model the execution of an agent, rather than employing modal operators. On the other hand, while AgentSpeak(L) programs are described by means of a proof-theoretic operational semantics, our approach provides a declarative, model-theoretic characterization of environment-aware agents. The relation of our approach with the agent language 3APL [16] (which has been shown to embed AgentSpeak(L)) is similar inasmuch as 3APL is provided only with an operational characterization.

MetateM (and Concurrent MetateM) [3, 10] is a programming language based on the notion of direct execution of temporal formulae, primarily used to specify reactive behaviours of agents. It shares similarities with the LUPS language inasmuch as both use rules to represent a relation between the past and the future, i.e. each rule in MetateM and in LUPS consists of conditions about the past (present) and a conclusion about the future. While the use of temporal logics in MetateM allows for the specification of rather elaborate temporal conditions, something for which LUPS was not designed, the underlying DLP semantics of LUPS allows the specification of agents capable of being deployed in dynamic environments where the governing laws change over time. While, for example, the temporal connectives $\Box$ and $\bigcirc$ of MetateM can be used to model the inflationary and non-inflationary behaviours, respectively, obtained when only considering definite logic programs, if we move to the more general class of logic programs with non-monotonic default negation both in the premisses and conclusions of clauses, LUPS and DLP directly provide an update semantics needed to resolve the contradictions naturally arising from conflicting rules acquired at different time instants, something apparently not possible in MetateM. This partially amounts to the difference between updating theories represented in classical logic and those represented by non-monotonic logic programs (cf. [1, 8]).

Related to the problem of environment-aware agents are also (real-time) reactive systems [14], that constantly interact with a given physical environment (e.g. automatic control and monitoring systems). In these real-time systems safety is often a critical issue, and so the existence of programming languages that allow programs to be easily designed and validated is crucial. With this purpose, Synchronous Declarative Languages have been designed (e.g. LUSTRE [7] and SIGNAL [4]). Such languages provide idealized primitives allowing users to think of their programs as reacting instantaneously to external events, and variables are functions of multiform time each having an associated clock defining the sequence of instants where the variable takes its values. Our approach shares with these the declarativity and the ability to deal with changing environments. However, their very underlying assumption that a program is always able to react to an event before any other event occurs, goes against the situations we want to model. As stated in the introduction, we are interested in modelling situations where rationality and reactivity are combined, where one cannot assume that the results are obtained before other events occur. On the contrary, with our approach (e.g., in a (uncontrolled) inflationary behaviour) external events may occur before all the conclusions reachable from previous events have been determined. Being based on LUPS, our approach also allows for modelling environments where the governing laws change over time, and where it is possible to reason with incomplete information (via nonmonotonic default negation). Both these aspects are out of the scope of the synchronous declarative languages. On the other hand, the ability of these languages to deal with various clocks, and the synchronization primitives, cannot be handled by our approach. The usefulness of these features to the problems we want to model, as well as the possibility of incorporating them in our approach, are subjects of current work.

Our representation of the possible environment-aware behaviours somehow resembles the possible world semantics of modal logics [15]. More precisely, the beliefs and the invariant of an environment-aware program (introduced in Sect. 4) are resource-bounded approximations of the set of formulae that are *possibly* true (i.e., true in a possible world) and *necessarily* true (i.e., true in every possible word). While the scope of our logic programming based characterization is narrower than a full-fledged modal logic, the former is simpler than the latter and it accounts for an effective prototyping of environment-aware agents.

6 Concluding remarks

We have shown how different environment-aware behaviours of agents can be naturally expressed in LUPS. The LUPS specification provides a formal declarative characterization of such behaviours, that can be exploited for performing resource-bounded analyses and verifications as illustrated in Sect. 4. Moreover, the available LUPS implementation can be exploited for experimenting and prototyping different specifications of environment-aware agents.

In Section 3 we have shown how the four main environment-aware behaviours analysed in [6] can be expressed in LUPS. It is important to observe that the use of LUPS to model environment-aware behaviours extends the approach of [6] in several ways: (1) different behaviours can be associated with different rules, while in [6] all the rules in a program must have the same behaviour; (2) LUPS allows for negative literals both in clause bodies and in clause heads of programs, while in [6] only definite programs are considered; (3) environment perceptions can be both positive and negative literals (rather than just positive literals).

Future work will be devoted to investigate further the greater expressive power featured by LUPS updates. For instance, we can model environment evolution by general LUPS updates and hence represent the environment itself as a dynamically evolving program (rather just as a sequence of perceptions). Another interesting direction for future work is to extend the stable semantics of LUPS programs in order to support an incremental, state-based characterization of environment-aware behaviours of programs in the style of [6]. A further interesting extension is to introduce quantitative aspects in the analysis of environment-aware behaviours, by associating probability distributions to the possible environment configurations along the lines of [5].

Acknowledgements

This work was partially supported by the bilateral Italian-Portuguese project "Rational and Reactive Agents", funded jointly by ICCTI and CNR, and by POCTI project FLUX. João Leite is partially supported by PRAXIS XXI scholarship no. BD/13514/97. Thanks are due to the anonymous referees for their remarks which helped us in improving the paper.

References

1. J.J. Alferes, J.A. Leite, L.M. Pereira, H. Przymusinska, and T. Przymusinsky. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 45(1-3):43-70, 2000.
2. J.J. Alferes, L.M. Pereira, H. Przymusinska, and T. Przymusinsky. LUPS: A language for updating logic programs. *Artificial Intelligence*, 2001. To appear. [A shorter version appeared in M. Gelfond, N. Leone and G. Pfeifer (eds), LPNMR'99, LNAI 1730, Springer-Verlag.]
3. H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A framework for programming in temporal logic. In *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (LNCS 430).* Springer-Verlag, 1989.
4. A. Benveniste, P Le Guernic and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103-149, 1991.
5. A. Brogi. Probabilistic behaviours of reactive agents. *Electronic Notes in Theoretical Computer Science*, 48, 2001.
6. A. Brogi, S. Contiero, and F. Turini. On the interplay between reactivity and computation. In F. Sadri and K. Satoh (editors) "CL2000 Workshop on Computational Logic in Multi-Agent Systems", London, July 2000.
7. P. Caspi, D. Pilaud, N. Halbwachs andJ. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, pages 178-188. ACM, 1987.

8. T. Eiter, M. Fink, G. Sabbatini and H. Tompits. On Properties of Update Sequences Based on Causal Rejection. *Theory and Practice of Logic Programming*, 2001. To appear.
9. A. van Emden, R.A. Kowalski. The semantics of predicate logic as a programming language. *J.ACM* 23:733-742, 1976.
10. M. Fisher. A survey of Concurrent METATEM - the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic - Proceedings of the First International Conference (LNCS 827)*, pages 102-117. Springer-Verlag, 1995.
11. M. Gelfond, V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K.A. Bowen, editors, *ICLP'88*. MIT Press, 1988.
12. M. Gelfond, V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17:301-322, 1993.
13. M. Gelfond, V. Lifschitz. Action languages. *Linkoping Electronic Articles in Computer and Information Science*, 3(16), 1998.
14. D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*. Springer, 1985.
15. G. Hughes and M.J. Cresswell. *A new introduction to modal logic*. RoutLedge, 1996.
16. K.V. Hindriks, F.S. de Boer, W. van der Hoek, and J-J. Ch. Meyer. A Formal Embedding of AgentSpeak(L) in 3APL. In G. Antoniou and J. Slaney, editors, *Advanced Topics in Artificial Intelligence (LNAI 1502)*, pages 155–166. Springer-Verlag, 1998.
17. F.F. Ingrand, M.P. Georgeff, and A.S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), 1992.
18. R. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality and reactivity. In C. Zaniolo and D. Pedreschi (editors) *Logic in Databases*, LNCS 1154, 137-150. Springer-Verlag, 1996.
19. J. A. Leite. A modified semantics for LUPS. In P. Brazdil and A. Jorge (eds.) *Progress in Artificial Intelligence, 10th Portuguese International Conference on Artificial Intelligence (EPIA'01)*. Springer LNAI 2285, 2001 (to appear).
20. A. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. *Lecture Notes in Computer Science*, 1038, pages 42-55, 1996.
21. F. Sadri and F. Toni. *Computational Logic and Multiagent Systems: a Roadmap*. 1999. Available at `http://www2.ags.uni-sb.de/net/Forum/Supportdocs.html`