Logic Programming for Evolving Agents

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

[1] CENTRIA, Universidade Nova de Lisboa, Portugal
[2] Dipartimento di Informatica, Università di Pisa, Italy

Abstract. Logic programming has often been considered less than adequate for modelling the dynamics of knowledge changing over time. In this paper we describe Evolving Logic Programs (EVOLP), a simple though quite powerful extension of logic programming, which allows for modelling the dynamics of knowledge bases expressed by programs, and illustrate its usage in modelling agents whose specifications may dynamically change. From the syntactical point of view, evolving programs are just generalized logic programs (i.e. normal LPs plus default negation in rule heads too), extended with (possibly nested) assertions, whether in heads or bodies of rules. 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 concomitant external updating originating in the environment. To illustrate the usage and power of EVOLP, and its ability to model agents' specifications, we elaborate on variations in the modelling of a Personal Assistant Agent for e-mail management.

1 Introduction

The agent paradigm, commonly implemented by means of imperative languages mainly for reasons of efficiency, has recently increased its influence in the research and development of computational logic based systems (see e.g. [15]).

Since efficiency is not always the crucial issue, but clear specification and correctness are, *Logic Programming* and *Non-monotonic Reasoning* have been brought back into the spotlight. Indeed, the *Logic Programming (LP)* paradigm provides a well-defined, general, integrative, encompassing, and rigorous framework for systematically studying computation, be it syntax, semantics, procedures, or attending implementations, environments, tools, and standards. Furthermore, *LP* approaches problems, and provides solutions, at a sufficient level of abstraction so that they generalize from problem domain to problem domain. This is afforded by the very nature of its foundations in logic, both in substance and method, and constitutes one of its major assets. To this accrues the recent significant improvements in the efficiency of *Logic Programming* implementations for *Non-monotonic Reasoning* [12, 13, 16].

While *LP* 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 sources, but also inner source knowledge updates (or self updates). In fact, an agent 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.

To address this concern, the authors, with others, first introduced *Dynamic Logic Programming (DLP)* [2]. There, they studied and defined the declarative and operational semantics of sequences of logic programs (or dynamic logic programs). [6] addressed similar concerns. According to DLP, knowledge is conveyed by a sequence of theories (encoded as generalized logic programs) representing different states of the world. Each of the states may contain mutually contradictory and overlapping information. The role of DLP is to take into account the mutual relationships extant between different states to precisely determine the declarative and the procedural semantics of the combined theory comprised of all individual theories and the way they relate.

Now, since logic programs can describe well knowledge states and, we have just mentioned above, also sequences of updating knowledge states, it's only fit that logic programs be utilized to describe the transitions between knowledge states as well. This can be achieved by associating with each state a set of transition rules to obtain the next state. However, till recently, *LP* had sometimes been considered less than adequate for modelling the dynamics of knowledge change over time. To overcome this limitation, languages like LUPS [3], EPI [5] and KABUL [9] were defined. But both these languages, a bit against the spirit of *LP* (whose pure programs have no keywords), are too verbose, and that makes them complex and difficult to prove properties of their programs. Moreover, each keyword encodes a high-level behaviour for the addition of rules, this constituting a problem in case one wants to describe a different, unforeseen, high-level behaviour.

In a previous work [1] we took the opposite approach towards the design of a new language, EVOLP. There, we analyze what was basic in the aforementioned languages, what they did offer that was new compared to classical *LP*, and then opted to start anew, and only minimally add constructs to *LP* so as to account for the newly required abilities needed for evolution through external updating and internal self updating. EVOLP provides a simpler and at once more general formulation of logic program updating, and runs closer to traditional LP doctrine. And it does so in a precise way, with a well-defined semantics of possible program evolutions through updates. We now move to show how EVOLP can be utilized to model the specification of evolving agents.

Indeed, EVOLP generalizes *LP* to allow 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 they belong to a model of the program P, can be employed to generate an updated version of P. This process can then be iterated on the basis of the new program. Whenever the program semantics affords several possible program models, evolution branching will occur, and several evolution sequences are made possible. This branching can be used to specify incomplete information about a situation. Moreover, the ability of EVOLP to nest rule assertions within assertions allows rule updates to be themselves updated down the line, conditional on each evolution strand. The ability to include assertive literals in rule bodies allows for looking ahead on some program changes and acting on that knowledge before the changes occur.

In all, 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, 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.

It is the goal of this paper to show that the attending formulation of EVOLP provides a good firm formal basis in which to express, implement, and reason about dynamic knowledge bases of evolving agents, and to show that it goes beyond some of the limitations of other approaches. To do this, in the ensuing section we present, with motivating examples, the formal syntax and semantics of EVOLP. Immediately afterwards we make our case by presenting a detailed and protracted application example of EVOLP usage, employing it to define an e-mail Personal Assistant Agent, whose executable specification evolves by means of external and of internal dynamic updates, both of which can be made contingent on the evolution context in which they occur. We end with a section comprising discussion, comparisons with related application work, open issues, and themes of future developments.

2 Evolving logic programs

2.1 Language

As mentioned in the Introduction, we are interested in a logic programming language, EVOLP, that caters for the evolution of an agent's knowledge, be it caused by external events, or by internal requirements for change. Above all, we desire to do so by adding as few new constructs to traditional logic programming as possible.

What is required to let logic programs evolve? For a start, one needs 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 negation not just in rule bodies, as in normal logic programming, but in rule heads as well[1].

Moreover, one needs a means to state that, under some conditions, some new rule or other is to be added to the program. We do so in EVOLP simply 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 $\mathcal{L}$ be any propositional language (not containing the predicate $assert/1$). The* extended language $\mathcal{L}_{assert}$ *is defined inductively as follows: – All propositional atoms in $\mathcal{L}$ are propositional atoms in $\mathcal{L}_{assert}$; – If each of $L_0, \ldots, L_n$*

[1] A well known extension to normal logic programs [10].

is a literal in $\mathcal{L}_{assert}$ (i.e. a propositional atom A or its default negation $not\, A$), then $L_0 \leftarrow L_1, \ldots, L_n$ is a generalized logic program rule over $\mathcal{L}_{assert}$; – If R is a rule over $\mathcal{L}_{assert}$ then $assert(R)$ is a propositional atom of $\mathcal{L}_{assert}$; – Nothing else is a propositional atom in $\mathcal{L}_{assert}$.

An evolving logic program *over a language $\mathcal{L}$ is a (possibly infinite) set of generalized logic program rules over $\mathcal{L}_{assert}$.*

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 changing 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 $\mathcal{L}$. An* event sequence *over P is a sequence of evolving programs over $\mathcal{L}$.*

2.2 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 some forms of its own evolution. Besides this, we consider sequences of events representing observation and commands arising from the outside. 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 knowledg 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 then treated as in DLP, where 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 stable models of normal [7] and generalized programs [10]. Formally, a *dynamic logic program* is a sequence $P_1 \oplus \cdots \oplus P_n$ (also denoted $\bigoplus \mathcal{P}$, where $\mathcal{P}$ is a set of generalized logic programs indexed by $1, \ldots, n$), and its semantic is determined by[2]:

Definition 3. *Let $\bigoplus\{P_i : i \in S\}$ be a dynamic logic program over language $\mathcal{L}$, let $s \in S$, and let M be a set of propositional atoms of $\mathcal{L}$. Then:*

$$Default_s(M) = \{not\, A \leftarrow . \mid \nexists A \leftarrow Body \in P_i (1 \leq i \leq s) : M \models Body\}$$
$$Reject_s(M) = \{L_0 \leftarrow Body \in P_i \mid \exists\ not\, L_0 \leftarrow Body' \in P_j, i < j \leq s\ \wedge M \models Body'\}$$

where A is an atom, $not\, L_0$ denotes the complement w.r.t. default negation of the literal L_0, and both $Body$ and $Body'$ are conjunctions of literals.

Definition 4. *Let $\mathcal{P} = \bigoplus\{P_i : i \in S\}$ be a dynamic logic program over language $\mathcal{L}$. A set M of propositional atoms of $\mathcal{L}$ is a stable model of $\mathcal{P}$ at state $s \in S$ iff:*

$$M' = least\left(\left[\bigcup\nolimits_{i \leq s} P_i - Reject_s(M)\right] \cup Default_s(M)\right)$$

where $M' = M \cup \{not\_A \mid A \notin M\}$, and $least(.)$ denotes the least model of the definite program obtained from the argument program by replacing every default negated literal $not\, A$ by a new atom not_A.

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

Example 2. Consider an initial program P, and that all the events are empty EVOLP programs:

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

The (only) stable model of P is $I = \{a, assert(b \leftarrow a)\}$ and it conveys 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. in P_2. In the only stable model I_2 of the new program $P \oplus P_2$, atom b is true as well as atom $assert(not\, a \leftarrow)$ and also c, meaning that $P \oplus P_2$ is ready to evolve into a new program $P \oplus P_2 \oplus P_3$ by adding rule $(not\, a \leftarrow)$ at the next step, i.e. in P_3. Now, the (negative) fact in P_3 conflicts with the fact in P, and so this older fact 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 stable model of the last sequence both a, b, and c are false.

[2] For more details, the reader is referred to [2].

This examples simplifies the problem of defining the semantics in that it does not consider the influence of events from the outside. In fact, as stated above, all those events are empty. How to treat outside events? The idea is very simple: add the rules that came in the i-th event to the program of state i, and calculate as in the example above.

Example 3. In the example above, suppose that at state 2 there is an event from the outside 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 should evolve into the new program obtained by updating P not only with the rule $b \leftarrow a$ but also with those rules, i.e. $P \oplus \{b \leftarrow a;\ assert(d \leftarrow b) \leftarrow a;\ e \leftarrow\}$. The only stable model I_2 of this program is now $\{b, assert(not\, a \leftarrow), assert(d \leftarrow b), e\}$.

If we keep with the evolution of this program (e.g. via some additional subsequent empty events), in the definition of the semantics we will have to decide what we want 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, but which are not to persist by inertia. I.e. if R belongs to some set S_i of an event sequence, this means that R was perceived or given 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, of course). In other words, when constructing subsequent states, the rules coming from events in state 2 should no longer be available and considered.

Definition 5. *An* evolution interpretation *of length n of an evolving program P over $\mathcal{L}$ is a finite sequence $\mathcal{I} = \langle I_1, I_2, \ldots, I_n \rangle$ of sets of propositional atoms of $\mathcal{L}_{assert}$. The* evolution trace *associated with an evolution interpretation $\mathcal{I}$ is the sequence of programs $\langle P_1, P_2, \ldots, P_n \rangle$ where:*

$$P_1{=}P \text{ and } P_i{=}\{R \mid assert(R) \in I_{i-1}\} \quad for\ each\ 2 \leq i \leq n.$$

Definition 6. *An evolution interpretation of length n, $\langle I_1, I_2, \ldots, I_n \rangle$, with evolution trace $\langle P_1, P_2, \ldots, P_n \rangle$, is an* evolution stable model *of P given $\langle E_1, E_2, \ldots, E_k \rangle$, with $n \leq k$, iff for every i ($1 \leq i \leq n$), I_i is a stable model at state i of $P_1 \oplus P_2 \cdots \oplus (P_i \cup E_i)$.*

Notice that the rules coming from the outside indeed 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 $M = \langle I_1, \ldots, I_n \rangle$ be an evolution stable model of P given an event sequence $\langle E_1, \ldots, E_n \rangle$. Then, for any $m > n$ and any sets of events*

$E_{n+1}, \ldots, E_m$, M is also an evolution stable model of P given an event sequence $\langle E_1, \ldots, E_n, E_{n+1}, \ldots, E_m \rangle$.

Being based on stable models, it is clear that EVOLP programs may have various evolution models of a given length, as well as no evolution stable models at all:

Example 4. Consider P, 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:

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

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

Definition 7. *Let P be an EVOLP program and $\mathcal{E}$ be an event sequence of length n, both over the language $\mathcal{L}$. A set of propositional atoms M over $\mathcal{L}_{assert}$ is a* Stable Model of P given $\mathcal{E}$ *iff there exists an evolution stable model of P given $\mathcal{E}$ with length n, where the last interpretation is M.*

We say that propositional atom A of $\mathcal{L}$ is: true given $\mathcal{E}$ *iff all stable models of P given $\mathcal{E}$ have A;* false given $\mathcal{E}$ *iff no stable model of P given $\mathcal{E}$ has A;* unknown given $\mathcal{E}$ *otherwise.*

With this definition of stable models given event sequences, we can now state that the semantics of EVOLP is, in fact, a proper generalization of the stable models semantics of normal and generalized logic programs, in the following sense:

Proposition 2. *Let P be a generalized logic program (without predicate $assert/1$) over a language $\mathcal{L}$, 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 $\mathcal{L}$ is a stable model of P (in the sense of [7, 10]).*

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. However, as we shall better discuss later, to implement agents that execute actions it might be important to guarantee that, for any given event sequence, no "branching" occurs, i.e. a single stable model exists.

Definition 8. *An EVOLP program P is* deterministic *given event sequence $\mathcal{E}$ iff there exists only one stable model of P given $\mathcal{E}$.*

The definition of conditions over programs and sequences to guarantee determinism is beyond the scope of this paper. Also beyond the scope is the issue of how to implement EVOLP, though an implementation already exists, available from http://centria.fct.unl.pt/~jja/updates/

3 E-mail Agent

Forthwith,EVOLP is employed to specify several features of a Personal Assistant agent for e-mail management, able to perform a few basic actions such as sending, receiving, and deleting messages, as well as moving them between folders, and to perform tasks such as filtering spam messages, storing messages in the appropriate folders, sending automatic replies, notifying the user and/or automatically forwarding specific messages, all of which dependant on user specified criteria. Some existing commercial systems already provide basic mechanisms to specify such tasks (e.g.[14, 8, 11]). If we expect the user to specify once and for all a consistent set of policies that trigger those actions then, such commercial systems would be all that is needed. But reality tells us otherwise: one observes that the user, every now and then, will discover new conditions under which incoming messages should be deleted, and under which messages now being deleted should not. If we allow the user to specify both the positive instances of such policies (e.g. *should be deleted*) and negative ones (e.g. *should not be deleted*), soon the union of all such rules becomes inconsistent, and we cannot expect the user to debug the set of rules so as to invalidate all the old rules that should no longer be used due to more recent ones that countervene them. We should allow the user to simply state whatever new is to be done, and let the agent automatically determine which of the old rules may persist and which not. We are not presupposing the user is contradictory, but just that he has updated his profile, something reasonable. For example, suppose he is tired of receiving spam messages advertising credit and tells the agent to delete all incoming messages whose subject contains the word *credit*. Later he finds out that important messages from his accountant are being deleted because the subject mentions *credit*. He should simply tell the agent not to delete such incoming messages from his accountant, and the agent should automatically determine that such messages are not to be deleted, in spite of the previous rule. But if we just evaluate the union of all specified policies, we obtain a contradiction. Next we show how EVOLP deals with these contradictions and automatically solves them with clear and precise semantics.

It would be important for the personal e-mail assistant agent to allow the user to specify tasks not as simple as just performing actions whenever some conditions are met. Suppose one is organizing a conference and wants to automate part of the communication with referees and authors. Basic tasks include automatic replies to authors whenever abstracts are submitted, etc. But more complex tasks can be conceived that we wish the agent to take care of, such as: waiting for messages from referees accepting to review a paper and, once the message arrives, forwarding to him a message with the paper if it has arrived, otherwise waiting till it arrives and forwarding it then; having different policies to deal with papers before and after the deadline; permitting the specification of extensions to the deadline on a case by case manner, and dealing with each of those papers differently; updating the initial specification for those policies; etc.

Throughout the remainder of this Section, we illustrate some features of EVOLP for these tasks. Instead of exploring all the basic features of the agent, many of which can be found in agents of the kind in the literature, we concentrate on those features directly concerned with the evolving specification of the agent, namely the representation of the dynamic user profile, and of the dynamic specification of its actions and their effects. The way to specify other common simple tasks can easily be inferred from the exposition. We also abstract from the way actions are actually executed. Often we address some of the issues in what does not seem like the most natural way, solely with the purpose of illustrating features of EVOLP, because it is difficult to show all of its capabilities in a single example. For lack of space, we do not show wholly the stable models, but rather single out their main characteristics for our purposes.

We start with a program that contains the initial specification of our agent. It consists of the rules r_1 through r_{10}, i.e. $P = \{\langle r_1\rangle, \langle r_2\rangle, \ldots, \langle r_{10}\rangle\}$.

$r_1 : time\,(1) \leftarrow \qquad r_2 : assert\,(time\,(T+1)) \leftarrow time\,(T)$
$r_3 : assert\,(not\,time\,(T)) \leftarrow time\,(T)$
$r_4 : assert\,(msg\,(M, F, S, B, T)) \leftarrow newmsg\,(M, F, S, B)\,, time\,(T)\,, not\,delete\,(M)$
$r_5 : assert\,(in\,(M, inbox)) \leftarrow newmsg\,(M, \_, \_, \_)\,, not\,move\,(M, F)\,, not\,delete\,(M)$
$r_6 : assert\,(in\,(M, F_{to})) \leftarrow newmsg\,(M, \_, \_, \_)\,, move\,(M, F_{to})$
$r_7 : assert\,(in\,(M, F_{to})) \leftarrow move\,(M, F_{from}, F_{to})\,, in\,(M, F_{from})$
$r_8 : assert\,(not\,in\,(M, F_{from})) \leftarrow move\,(M, F_{from}, F_{to})\,, not\,in\,(M, F_{to})$
$r_9 : assert\,(not\,in\,(M, F)) \leftarrow delete\,(M)\,, in\,(M, F)$
$r_{10} : assert\,(sent\,(To, S, B, T)) \leftarrow send\,(To, S, B)\,, time(T)$

The first three encode a clock which for now will be used to time-stamp all incoming messages. Note such time-stamping is not really required, but we thought it useful to show how a clock can be encoded in EVOLP. Rule r_4 specifies that all incoming messages, represented by atom $newmsg\,(MsgId, From, Subject, Body)$, if not specified to be deleted, represented by literal $not\,delete\,(MsgId)$, should be time-stamped and asserted as a fact of the form $msg\,(MsgId, From, Subject, Body, Time)$. Rule r_5 specifies that all incoming messages, if not specified to be deleted, and not specified to be moved to a specific folder represented by $not\,move\,(MsgId, Folder)$, should be stored in the folder inbox. We use $in\,(MsgId, Folder)$ to represent that the message $MsgId$ is in folder $Folder$. Rule r_6 specifies the effect of moving an incoming message to a specific folder. Rule r_7 and r_8 encode the effect of moving a message between folders, represented by $move\,(MsgId, Folder_{from}, Folder_{to})$. Note no problem exists with specifying that a message is to be moved between the same folder. Rule r_9 specifies the effect of the action delete, represented by $delete\,(MsgId)$. This action causes the message to be removed from its current folder. Finally, rule r_{10} encodes that sending a message, represented by the atom $send\,(To, Sbject, Body)$, causes the message to be sent, hereby represented by the assertion of the fact $sent\,(To, Subject, Body, Time)$.

At this initial state, the stable model only contains $time\,(1)$. With this initial specification, since we do not yet have any rules to specify which incoming messages are to be deleted and which are to be moved, every message received is moved to folder *inbox*. Also, at every state transition, the clock increases its value. Suppose we receive an update containing three messages i.e. an event E_1 with the facts:

$newmsg\,(1,$ "a@a", "credit", "some spam text"$)$
$newmsg\,(2,$ "accountant@c", "hello", "some text"$)$
$newmsg\,(3,$ "b@d", "free credit", "more spam"$)$

After this update, the stable model contains:

$assert\,(msg\,(1, \text{“a@a”}, \text{“credit”}, \text{“some spam text”}, 1))\,, assert\,(in\,(1, inbox))$
$assert\,(msg\,(2, \text{“accountant@c”}, \text{“hello”}, \text{“some text”}, 1))\,, assert\,(in\,(2, inbox))$
$assert\,(msg\,(3, \text{“b@d”}, \text{“free credit”}, \text{“more spam”}, 1))\,, assert\,(in\,(3, inbox))$
$time\,(1)\,, assert\,(not\,time\,(1))\,, assert\,(time\,(2))$

With this, we construct P_2 containing the facts:

$msg\,(1, \text{“a@a”}, \text{“credit”}, \text{“some spam text”}, 1)\,, in\,(1, inbox)\,, not\,time\,(1)$
$msg\,(2, \text{“accountant@c”}, \text{“hello”}, \text{“some text”}, 1)\,, in\,(2, inbox)$
$msg\,(3, \text{“b@d”}, \text{“free credit”}, \text{“more spam”}, 1)\,, in\,(3, inbox)\,, time\,(2)$

indicating that the agent's knowledge base has been updated so as to store all messages, properly time-stamped, in folder *inbox*. Moreover the clock was updated to its new value.

At this point, the user becomes upset with all the spam messages being received and decides to start deleting them on arrival. For this he updates the agent by asserting a general rule specifying that spam messages should be deleted, encoded as the assertion of rule r_{11}, and he also updates the agent with a definition of what should be considered a spam message, in this case those whose subject contains the word "credit", encoded by the assertion of rule r_{12}.

$$r_{11} : delete\,(M) \leftarrow newmsg\,(M, F, S, B)\,, spam\,(F, S, B)$$
$$r_{12} : spam\,(F, S, B) \leftarrow contains\,(S, \text{“credit”})$$

Throughout, consider the literal $contains\,(S, T)$ true whenever T is contained in S, whose specification we omit for brevity. The assertion of these two rules, together with an update so as to delete messages 1 and 3, constitutes event E_2:

$$E_2 = \{assert\,(\langle r_{11}\rangle)\,, assert\,(\langle r_{12}\rangle)\,, delete\,(1)\,, delete\,(3)\}$$

After this update, the stable model contains:

$assert\,(\langle r_{11}\rangle)\,, assert\,(\langle r_{12}\rangle)\,, delete\,(1)\,, delete\,(3)\,, assert\,(not\,in\,(1, inbox))$
$assert\,(not\,in\,(3, inbox))\,, assert\,(time\,(3))\,, assert\,(not\,time\,(2))$

together with those propositions of the form $msg/5$, $time/1$, $in/2$, representing the existing messages, their locations, and the current internal time[3].

From this model we construct program P_3, which contains r_{11}, r_{12}, together with the facts $time\,(3)$, $not\,time\,(2)$, $not\,in\,(1, inbox)$ and $not\,in\,(3, inbox)$.

Suppose we receive an update containing three messages, i.e. an event E_3 with the facts:

$newmsg\,(4, \text{“d@a”}, \text{“free credit card”}, \text{“spam spam spam”})$
$newmsg\,(5, \text{“accountant@c”}, \text{“credit”}, \text{“got your credit”})$
$newmsg\,(6, \text{“girlfriend@d”}, \text{“hi”}, \text{“theater tonight?”})$

After this update, the stable model contains:

$spam\,(F, \text{“free credit card”}, B)\,, assert\,(in\,(3, inbox))\,, delete\,(4)\,, delete\,(5)\,,$
$assert\,(msg\,(6, \text{“girlfriend@d”}, \text{“hi”}, \text{“theater tonight?”}, 3))\,, spam\,(F, \text{“credit”}, B)$

[3] From now on, we omit all those propositions and assertions concerning the clock, unless relevant for the presentation.

Since messages 4 and 5 are considered spam messages, they are both set for deletion and thus are not asserted. Only message 6 is asserted. From this model we construct the program P_4 which contains facts $not\,time\,(3)$, $in\,(6, inbox)$, $time\,(4)$, and $msg\,(6,$ "girlfriend@d", "hi", "theater tonight?", $3)$.

Next we receive an update containing a single message i.e. an event E_4 with the fact[4]:

$$newmsg\,(7, \text{"accountant@c"}, \text{"are you there?"}, \text{"..."})$$

This message made the user aware that previous messages from his accountant had been deleted as spam. The user then decides to update the definition of spam, stating that messages from his accountant are not spam. He does this by asserting rule r_{13} (below). Note this rule is contradictory with rule r_{12}, for messages from the accountant with subject containing the word "credit". But EVOLP automatically detects such contradictions and removes them by taking the newer rule to be an update of any previously existing ones, and we thus expect such messages not to be deleted. Now the user is appointed conference chair and decides to program the agent to perform some attending tasks. Henceforth, messages with the subject "abstract" should be moved to folder *abstracts*, encoded by rule r_{14}, those containing the word "cfp" in their subjects should be moved to folder cfp (r_{15}). Furthermore, as the user is accustomed to only looking at his inbox folder, he wishes to be notified whenever an incoming message is immediately stored at a folder other than *inbox*. This is accomplished with rule r_{16}, which renders $notify\,(M)$ true in such cases. Mark that $notify/1$ represents an action with no internal effect on the agent's knowledge base. The agent must also send a message acknowledging receipt of every abstract (r_{17}). And since the user will be away from his computer, he decides to forward urgent mail to his new temporary address. This could be accomplished by simply stating that those messages should be sent to his new address. But he decides to create a new internal action, represented by $forward\,(MsgId, To)$, whose effect is to forward the newly incoming message $MsgId$ to the address To, thus making it easier to specify future forwarding options. The specification of this action is achieved by asserting rule r_{18}. Then, based on this action, he can specify that all urgent messages be forwarded to his new address, by asserting rule r_{29}. Finally, the user realizes that the messages that have been deleted are not being effectively deleted, but rather only removed from their folders, i.e. $msg\,(M, F, S, B, T)$ is still true, except that there is no $in\,(M, \_)$ that is true. He then decides to create another internal action, *purge*, whose effect is that of making false all those messages that have been previously removed from all folders by the action *delete*. The specification of this action is obtained by asserting rule r_{20}.

$r_{13} : not\,spam\,(F, S, B) \leftarrow contains\,(F, \text{"accountant"})$
$r_{14} : move\,(M, abstracts) \leftarrow newmsg\,(M, F, S, B)\,, contains\,(S, \text{"abstract"})$
$r_{15} : move\,(M, cfp) \leftarrow newmsg\,(M, F, S, B)\,, contains\,(S, \text{"cfp"})$
$r_{16} : notify\,(M) \leftarrow newmsg\,(M, F, S, B)\,, not\,assert\,(in\,(M, inbox))\,, assert\,(in\,(M, Fldr))$
$r_{17} : send\,(From, S, \text{"Thanks"}) \leftarrow newmsg\,(M, F, S, B)\,, contains\,(S, \text{"abstract"})$
$r_{18} : send\,(To, S, B) \leftarrow forward\,(M, To)\,, newmsg\,(M, F, S, B)$
$r_{19} : forward\,(M, \text{"b@domain"}) \leftarrow newmsg\,(M, \_, \text{"urgent"}, \_)$
$r_{20} : assert\,(not\,msg\,(M, F, S, B, T)) \leftarrow purge, msg\,(M, F, S, B, T)\,, not\,in\,(M, \_)$

[4] At this state we omit the model and update.

The assertion of all these rules constitutes event $E_5 = \{assert(\langle r_{13}\rangle), assert(\langle r_{14}\rangle), assert(\langle r_{15}\rangle), assert(\langle r_{16}\rangle), assert(\langle r_{17}\rangle), assert(\langle r_{18}\rangle), assert(\langle r_{19}\rangle), assert(\langle r_{20}\rangle)\}$

At the subsequent update the agent receives more messages, performs a *purge*, moves message 6 to the private folder, and deletes message 6, encoded by the following facts belonging to E_6:

$$\begin{array}{l} newmsg\,(9, \text{“a2@e”}, \text{“abstract”}, \text{“abs...”})\,, newmsg\,(10, \text{“a3@e”}, \text{“abstract”}, \text{“abs...”}) \\ newmsg\,(13, \text{“accountant@c”}, \text{“fwd:credit”}, \text{“...”})\,, newmsg\,(11, \text{“x@d”}, \text{“urgent”}, \text{“...”}) \\ move\,(6, inbox, private)\,, delete\,(6)\,, purge, newmsg\,(8, \text{“a1@e”}, \text{“abstract”}, \text{“abs...”}) \\ newmsg\,(12, \text{“accountant@c”}, \text{“fwd:credit”}, \text{“...”}) \end{array}$$

After this update, the stable model contains, for messages 1 and 3, as a result of the *purge*, $assert\,(not\,msg\,(M, F, S, B, T))$; $assert\,(in\,(M, abstracts))$ for messages 8, 9 and 10, $forward\,(11, \text{“b@domain”})$ and the corresponding send action, i.e. $send\,(\text{“b@domain”}, \text{“urgent”}, \text{“...”})$, and, concerning message 6, the stable model contains $assert\,(in\,(6, private))$ and $delete\,(6)$. There are also notifications for messages 8, 9, 10 and 14. Next the user decides that whenever a message is both deleted and moved, the deletion action prevails, i.e. it should not be asserted into the folder specified by the move action. This is encoded by the assertion of rule r_{21} (below). Furthermore, the user decides to update his spam rules to avoid all the spam his accountant has been forwarding to him (r_{22}). Finally, because he wants the agent to deal with communication with the referees, he sets up the assignments between referees and submitted papers ($r_{23} - r_{28}$). The rules and event E_7 are:

$$\begin{array}{ll} \multicolumn{2}{l}{r_{21} : not\,assert\,(in\,(M, F_{to})) \leftarrow move\,(M, F_{from}, F_{to})\,, delete\,(M)} \\ \multicolumn{2}{l}{r_{22} : spam\,(F, S, B) \leftarrow contains\,(S, \text{“credit”})\,, contains\,(S, \text{“Fwd”})} \\ r_{23} : assign\,(\text{“paper1”}, \text{“ref2@b”}) & r_{24} : assign\,(\text{“paper2”}, \text{“ref2@b”}) \\ r_{25} : assign\,(\text{“paper2”}, \text{“ref3@c”}) & r_{26} : assign\,(\text{“paper3”}, \text{“ref3@c”}) \\ r_{27} : assign\,(\text{“paper3”}, \text{“ref1@a”}) & r_{28} : assign\,(\text{“paper1”}, \text{“ref1@a”}) \end{array}$$

$$E_7 = \left\{ \begin{array}{l} assert\,(\langle r_{21}\rangle)\,, assert\,(\langle r_{22}\rangle)\,, assert\,(\langle r_{23}\rangle)\,, assert\,(\langle r_{24}\rangle) \\ assert\,(\langle r_{25}\rangle)\,, assert\,(\langle r_{26}\rangle)\,, assert\,(\langle r_{27}\rangle)\,, assert\,(\langle r_{28}\rangle) \end{array} \right\}$$

After all these rules have been asserted, and at the subsequent update, the agent receives a spam message from the accountant, performs a move and a delete of message 12 to test if the new rule is working, and sends messages to the referees inviting them to review the corresponding papers, encoded by the following facts and rules that belong to E_8:

$$\begin{array}{ll} \multicolumn{2}{l}{newmsg\,(15, \text{“accountant@c”}, \text{“fwd:credit”}, \text{“...”})\,, move\,(12, inbox, folder1)} \\ delete\,(12) & send\,(R, PId, \text{“invitation to review”}) \leftarrow assign\,(PId, R) \end{array}$$

At this point, we invite the reader to check that message 15 was rejected, and that message 12 was indeed deleted. It is important to note that the messages to the referees are only sent once. This is so because the rule belonging to E_8 is not an assertion and thus never becomes part of the agent's knowledge base. It is only used to determine the stable model at this state, and never used again.

Subsequently the user decides to specify the way the agent should deal with communication with authors and reviewers. Forthwith, we show how some of these tasks could be programmed. Upon receipt of a message from a reviewer accepting to review a given paper, the paper should be sent to the referee once it arrives. This could be specified by rule r_{29} (below) which specifies the assertion of a rule that sends the paper to the referee, but this assertion should only take place after the referee accepts the task. If the paper has already been received when the reviewer accepts the task, then it should be sent immediately (r_{30}). Of course, if papers are received after some deadline, and unless some extension was given for a particular paper, then they should be rejected and the author so notified. This is encoded by rules r_{31} and r_{32} which are asserted when the deadline is reached, even though it has not been set yet. Rule r_{31} sends a message to the author while rule r_{32} prevents the paper from being sent to the referee. Finally, the user asserts two rules to deal with deadline extensions on a paper by paper basis. Whenever the user includes an event of the form $dline\,(PId, Dur)$ in an update, he is giving an extension of the deadline concerning paper PId and with duration Dur. This immediately causes $ext\,(PId)$ to be asserted, preventing the paper from being rejected. Concurrently, by means of rule r_{34}, a rule is asserted that will render $ext\,(PId)$ false once the deadline plus the extension is reached, after which the paper is rejected.

$$
\begin{aligned}
&r_{29} : assert\,(send\,(R, S, B) \leftarrow newmsg\,(M, F, S, B)\,, contains\,(S, PId)\,, assign\,(PId, R)) \leftarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad newmsg\,(M, R, PId, B)\,, contains\,(B, \text{“accept”}) \\
&r_{30} : send\,(R, PId, B) \leftarrow newmsg\,(M, R, PId, B_1)\,, contains\,(B_1, \text{“accept”})\,, msg\,(M_1, F, PId, B, T) \\
&r_{31} : assert\,(send\,(F, S, \text{“too late”}) \leftarrow newmsg\,(M, F, S, B)\,, contains\,(S, PId)\,, not\,ext\,(PId)) \leftarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad time\,(T)\,, deadline\,(T) \\
&r_{32} : assert\,(not\,send\,(Referee, S, B) \leftarrow newmsg\,(M, F, S, B)\,, contains\,(S, PId)\,, not\,ext\,(PId)) \leftarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad time\,(T)\,, deadline\,(T) \\
&r_{33} : assert\,(ext\,(PId)) \leftarrow dline\,(PId, Dur) \\
&r_{34} : assert\,(assert\,(not\,ext\,(PId)) \leftarrow time\,(Dur + T)\,, deadline\,(T)) \leftarrow dline\,(PId, Dur)
\end{aligned}
$$

The event that encodes this update is: $E_9 = \{assert\,(\langle r_{29}\rangle),\ assert\,(\langle r_{30}\rangle),\ assert\,(\langle r_{31}\rangle),\ assert\,(\langle r_{32}\rangle),\ assert\,(\langle r_{33}\rangle),\ assert\,(\langle r_{34}\rangle)\}$

Subsequently the user sets the deadline by asserting the fact $deadline\,(14)$[5], i.e. the event E_{10} contains the fact $assert\,(deadline\,(14))$.

The remainder of the story goes as follows: at event E_{11} the agent receives both acceptance messages from referee 1; at event E_{12} it receives paper 2; the user grants deadline extensions of two time units to papers 1 and 3, encoded in event E_{13}; at event E_{14} it receives the acceptance messages from referee 2; at event E_{15}, i.e. after the deadline but before the extension, it receives paper 1; at event E_{16} it receives the acceptance messages from referee 3; at event E_{17}, i.e. after the extension has expired, it receives paper 3. Lack of space prevents us from elaborating further on what happens after all these events, but we invite the reader to check that: after event E_{14} paper 2 is sent to referee 2; after event E_{15} paper 1 is sent both to referees 1 and 2; after event E_{16} paper 2 is sent to referee 3; since paper 3 arrives after the deadline extension it is rejected, a message is sent to the author, and the paper is not sent to any referee.

$$
\begin{aligned}
&E_{11} = \{newmsg\,(16, \text{“ref1@a”}, \text{“paper1”}, \text{“accept”})\,, newmsg\,(17, \text{“ref1@a”}, \text{“paper3”}, \text{“accept”})\} \\
&E_{12} = \{newmsg\,(18, \text{“a2@e”}, \text{“paper2”}, \text{“the paper”})\}\ E_{13} = \{dline\,(\text{“paper3”}, 2)\,, dline\,(\text{“paper1”}, 2)\} \\
&E_{14} = \{newmsg\,(19, \text{“ref2@b”}, \text{“paper1”}, \text{“accept”})\,, newmsg\,(20, \text{“ref2@b”}, \text{“paper2”}, \text{“accept”})\} \\
&E_{15} = \{newmsg\,(21, \text{“a1@e”}, \text{“paper1”}, \text{“the paper”})\}\ E_{17} = \{newmsg\,(24, \text{“a3@e”}, \text{“paper3”}, \text{“the paper”})\} \\
&E_{16} = \{newmsg\,(22, \text{“ref3@c”}, \text{“paper2”}, \text{“accept”})\,, newmsg\,(23, \text{“ref3@c”}, \text{“paper3”}, \text{“accept”})\}
\end{aligned}
$$

[5] Dealing with the synchronisation of external and internal times is outside the scope of this paper. Here, we set the deadline as a value that refers to the agent's internal time.

4 Discussion and Conclusions

Though permitted by EVOLP, the example does not involve any branching of evolutions, i.e. there is always a single stable model of the program, given any of the example's events. In fact, that program given those events is deterministic (in the sense of Definition 8). Non-stratified rules for assertions (as in Example 4) can be used to model alternative updates to the agent's knowledge base, e.g. for stating that, under certain conditions, either move a message to a folder or delete it, but not both. Non-stratification can also be used to model uncertainty in the external observations. In both these cases, EVOLP semantics provides several evolution stable models, upon which reasoning can be made, concerning what happens in case one or other action is chosen. On the other hand, by having various models, EVOLP can no longer be used to actually perform the actions, unless some mechanism for selecting models is introduced. For (static) logic programs, this issue of selecting among stable models has already been extensively studied: either by defining more skeptical semantic that always provide a unique model or by preferring among stable models based on some priority ordering on rules. The introduction of such mechanisms in EVOLP too is the subject of current and future work by the authors.

Another issue, not illustrated in the example, and not (yet) addressed by EVOLP, is that of synchronisation of external and internal times. In the example, this problem does not even appear, until the moment where we want to set the deadline. For expressing the deadline in terms of the internal time we can assume, for example, that an event is given to the agent (albeit empty) after every fixed amount of external time (say 5 minutes). This way, we could express the external time and date of the deadline as the time-stamp of an internal state. Another possibility would be to assume that every event comes with a fact $etime(T)$ stating at which moment T (in term of external time) the event occurred, and then compare the last T with the deadline. For this example, both these solutions would be enough, since synchronisation is not a crucial issue. But in general, synchronisation is an issue that deserves further attention, and is the subject of future work by the authors.

A large number of software products is nowadays available to perform email monitoring and filtering. One example is Spam Agent [14], a recently deployed email monitoring and filtering tool which features a comprehensive set of filters (over 1500) to block spam and unwanted emails. Email monitoring and filtering rules can be defined in terms of message sender, recipient, subject, body, and arbitrary combinations of them. The SpreadMsg software email filtering and forwarding agent [8] provides unattended data capture, scanning, and extraction from a wide variety of data. User rule sets are applied to data and, when criteria are met, data are further parsed and turned into messages that can be delivered to email addresses, text pagers, digital cellular or GSM mobile phones, laptops, PDA's etc.. The SuperScout Email Filter [11], besides supporting similar capabilities to the previoulsy mentioned agents, features a Virtual Learning Agent (VLA). The VLA is a content development tool that can be trained to understand and recognise specific proprietary content in order to protect confidential and business critical information from the security risks arising from accidental or malicious leakage. Widely used commercial email filtering systems provide SPAM filters and, in some cases, a more general set of email handling rules. Outlook, Netscape, and Hotmail, for instance, all provide means to define email filtering rules. Some systems require the user to write the filtering rules, while others employ learning algorithms or try to extract patterns from examples.

An interesting, very recent proposal is the Personal Email Assistant (PEA) [4], which is intended to provide a customizable, machine learning based environment to support email processing. An aspect of PEA is that it relies on combining available open source components for information retrieval, machine learning, and agents.

Lack of space prevents us from mentioning other (out of many) email monitoring and filtering agents available. It is worth observing however that, to the best of our knowledge, none of the available agents enjoys the ability of autonomously and dynamically updating its own filtering policies in a way as general as the EVOLP specifications illustrated in the present work.

We began this article by extolling the virtue and promise of Logic Programming, for adumbrating the issues and solutions relative to the (internally and externally) updatable executable specification of agents, and to the study of their evolution by means of a precisely defined declarative semantics with well-defined properties. We have shown, in an example, how EVOLP can be used, not only to specify the agent's policies, but also to dynamically change that specification, and make those changes dependent on some external conditions or events.

References

1. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In *JELIA'02*, volume 2424 of *LNAI*. Springer, 2002.
2. 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.
3. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS : A language for updating logic programs. *Artificial Intelligence*, 138(1-2), 2002.
4. R. Bergman, M. Griss, and C. Staelin. A personal email assistant. Technical Report HPL-2002-236, HP Labs Palo Alto, 2002.
5. T. Eiter, M. Fink, G. Sabbatini, and H Tompits. A framework for declarative update specifications in logic programs. In *IJCAI'01*. Morgan-Kaufmann, 2001.
6. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming*, 2002. to appear.
7. M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In *ICLP'88*. MIT Press, 1988.
8. Compuquest Inc. Spreadmsg. www.compuquestinc.com.
9. J. A. Leite. *Evolving Knowledge Bases*. IOS Press, 2003.
10. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In *KR'92*. Morgan-Kaufmann, 1992.
11. Caudex Services Ltd. Superscout email filter. www.caudexservices.co.uk.
12. I. Niemelä and P. Simons. Smodels: An implementation of the stable and well-founded semantics for normal LP. In *LPNMR'97*, volume 1265 of *LNAI*. Springer, 1997.
13. D. De Schreye, M. Hermenegildo, and L. M. Pereira. Paving the roadmaps: Enabling and integration technologies, 2000. Available from www.compulog.org.
14. Spam-Filtering-Software.com. Spam agent. www.spam-filtering-software.com.
15. V. S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Ozcan, and Robert Ross. *Heterogeneous Agent Systems*. MIT Press, 2000.
16. XSB-Prolog. The XSB logic programming system, version 2.0, 1999. www.cs.sunysb.edu/ sbprolog.