Evolving Logic Programs

José Júlio Alferes[1], Antonio Brogi[2],
João Alexandre Leite[1], and Luís Moniz 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 a simple though quite powerful approach to modelling the updates of knowledge bases expressed by generalized logic programs, by means of a new language, hereby christened EVOLP (after *EVO*lving *L*ogic *P*rograms). The approach was first sparked by a critical analysis of previous efforts and results in this direction [1,2,7,11], and aims to provide a simpler, and at once more general, formulation of logic program updating, which runs closer to traditional logic programming (LP) doctrine. From the syntactical point of view, evolving programs are just generalized logic programs (i.e. normal LPs plus default negation also in rule heads), extended with (possibly nested) assertions, whether in heads or bodies of rules. From the semantics viewpoint, a model-theoretic characterization is offered of the possible evolutions of such programs. These evolutions arise both from self (or internal) updating, and from external updating too, originating in the environment. This formulation sets evolving programs on a firm basis in which to express, implement, and reason about dynamic knowledge bases, and opens up a number of interesting research topics that we brush on.

1 Introduction

Until recently, LP has often been considered less than adequate for modelling the dynamics of knowledge changing over time. To overcome this limitation, languages like LUPS [2] and EPI [7] have been defined. For this purpose, LUPS provides several types of update commands: **assert**, **retract**, **always**, **cancel**, **assert event**, **retract event** and **always event**, all of which having a rule as argument. Such commands can be made conditional on the current state, the conditions being preceded by the keyword **when**. An example of a LUPS command is: **always** $L \leftarrow L_1, \ldots, L_k$ **when** $L_{k+1}, \ldots, L_m$, meaning that, from the moment it is given onwards, whenever all of $L_{k+1}, \ldots, L_m$ are true, the knowledge base should be updated with the rule $L \leftarrow L_1, \ldots, L_k$. A declarative, as well as a procedural semantics for sequences of sets of commands is defined in [2], where its application to several areas is illustrated. EPI [7] proposes two additional main features, both achieved by extensions of LUPS commands: to allow for the specification of commands whose execution depends on the concurrent execution of other commands; and to allow for external events to condition the evolution of the knowledge base.

Both these languages, a bit against the spirit of LP (which, in pure programs, has no keywords), are too verbose. Their verbosity makes them complex, difficult to use, and to prove program properties. Moreover, each keyword encodes a high-level behaviour for the addition of rules. And this constitutes a problem in case one wants to describe a different, unforeseen, high-level behaviour. For instance, one may want: to make the addition of a command dependent upon conditions that may span more than one state; to model changes to update commands themselves; etc. None of these high-level behaviours are achievable by LUPS or EPI commands. Of course, one could further extend these languages. Not only would this make them more and more complex, but also some yet unforeseen but desirable commands would certainly be still missing. We took the opposite approach: instead of extending these languages with new commands, we analyzed what is basic in them, what they offer that is new compared to classical LP, and then minimally add constructs to LP to account for the new capabilities of evolution and updating. The resulting language, EVOLP, provides a simpler, and at once more general, formulation of logic program updating, which runs closer to traditional LP doctrine.

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. When the program semantics affords several program models, branching evolution will occur and several evolution sequences are possible. Branching can be used to specify incomplete information about a situation. 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 program changes and acting on that knowledge before the changes take place.

The ensuing notion of self evolving programs is more basic than the subsequent elaboration of general evolving programs, those that also permit, besides internal or self updates, for updates arising from the outside. So first we define a language of programs able to express changes to its very programs, and study how programs may evolve by themselves. Only afterwards do we analyze and define extensions that cater for the interference from external updates.

The proposed formulation sets EVOLP on a firm formal basis in which to express, implement, and reason about dynamic knowledge bases, and opens up a number of interesting research topics that we brush on. As a consequence of the richness of the new paradigm, our emphasis in this paper will be much more on the appealingness of its foundational precepts and scaffolding, than on exploring specific properties, as these require constraining the general core setting with distinctive further options. Accordingly, in the next two sections we formally define the syntax and semantics of EVOLP. Then we present examples of usage. We end with a section on discussion, brief comparisons with related work, open issues, and themes of future work.

2 Self-evolving Logic Programs

What is required to let logic programs evolve by themselves? To start with, one needs some form of negation in heads of rules, so as to let older rules to be supervened by more recent ones updating them, and thus admit non-monotonicity of rules. Second, one needs a means to state that, under some conditions, some new rule or other is to be added to the program.

Accordingly, for the syntax of EVOLP we simply adopt that of generalized LPs augmented with the reserved predicate $assert/1$, whether as the rule head or in its body, whose sole argument is itself a full-blown rule, so that arbitrary nesting becomes possible. 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:*

1. *All propositional atoms in $\mathcal{L}$ are propositional atoms in $\mathcal{L}_{assert}$.*
2. *If each of $L_0, \ldots, L_n$ 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 $\mathcal{L}_{assert}$.*
3. *If R is a rule over $\mathcal{L}_{assert}$ then $assert(R)$ is a propositional atom of $\mathcal{L}_{assert}$.*
4. *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 $\mathcal{L}_{assert}$.*

We decided not to include an explicit retract construct in the basic language. Indeed, as we have default negation available also in rule heads, retraction of rules can be encoded in EVOLP. This encoding is, however, left out of this paper.

Self-evolving programs can update their own rules and exhibit a dynamic, non-monotonic behaviour. Their meaning 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 self-evolution of one initial program after a given number n of evolution steps. Each self-evolution is represented by a sequence of programs, each program corresponding to a state.

The sequences of programs are treated as in Dynamic Logic Programs (DLP) [1], where the most recent rules are put in force, and the previous rules are valid (by inertia) as far 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 [8] and generalized programs [12]. Formally, a *dynamic logic program* is a sequence $P_1 \oplus \ldots \oplus P_n$ (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[1]:

Definition 2. *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:*

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

[1] For more details, the reader is referred to [1].

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 3. *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:*

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

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.

The primordial intuitions for the construction of the 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 (SM) 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. Before presenting the definitions that formalize these intuitions, let us show an illustrative example. Consider the self-evolving program P:

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

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. Thus, $assert(not\, a \leftarrow)$ as well as c are also no longer true. In the only stable model of the last sequence both a, b and c are false.

Definition 4. *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:*

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

Definition 5. *Let $M = \langle I_1, I_2, \ldots, I_n\rangle$ be an evolution interpretation of an evolving logic program P, and let $\langle P_1, P_2, \ldots, P_n\rangle$ be its evolution trace. M is an* Evolution Stable Model *of P iff for every i $(1 \le i \le n)$, I_i is a stable model at i of the DLP: $P_1 \oplus P_2 \oplus \ldots \oplus P_i$.*

Being stable models based, it is clear that a self-evolving program may have various evolution models of a given length, as well as no evolution stable models at all. We say that a self-evolving program is inconsistent when it has no evolution stable models of any length, and that it is inconsistent after n-steps if it has at least one stable model after $n-1$ steps and has no stable models thereafter. Note that if a program has no stable models after n steps then, for every $m \geq n$ it has no stable model after m steps. It is also important to observe that the set of all evolution stable models of a program, up to any length n, can be constructed incrementally by means of an operator $\mathcal{T}_P$.

Definition 6. *Let P be an evolving program. The operator $\mathcal{T}_P$ on sets of evolution interpretations is defined by:*

$$\begin{aligned}\mathcal{T}_P(\mathcal{I}) = \{\langle I_1, \ldots, I_i, I_{i+1}\rangle \ \mid \langle I_1, \ldots, I_i\rangle \in \mathcal{I} \ \wedge \\ \wedge\ I_{i+1} \text{ is a stable model of } P_1 \oplus \ldots \oplus P_i \oplus P_{i+1} \text{ at state } i+1\}\end{aligned}$$

where $P_1 = P$ and, for $1 < j \leq i+1$, $P_j = \{R \mid assert(R) \in I_{j-1}\}$.

Theorem 1. *Given an evolving program P, build the sequence of sets of evolution interpretations: $\mathcal{I}_1 = \{\langle I\rangle \mid I$ is a stable model of $P\}$, and $\mathcal{I}_{i+1} = \mathcal{T}_P(\mathcal{I}_i)$. M is an evolution stable model of length n iff M belongs to $\mathcal{I}_n$.*

Each evolution stable model represents one possible evolution of the program, after a given number of steps. However, they do not directly determine a truth relation. What is true (resp. false) after a given number of evolution steps? This is an important question, when one wants to study the behaviour of a self-evolving program. Here, one will be interested in knowing what is guaranteed to be true (or false) after n steps of evolution, and what is unknown (or uncertain). In the example above a is guaranteed true after 1 and 2 steps and false after n steps, for any $n \geq 3$; b and c are true after 2 steps and false after n steps for any $n \neq 2$.

Definition 7. *Let P be an evolving program over the language $\mathcal{L}$. We say that a set of propositional atoms M over $\mathcal{L}_{assert}$ is a* stable model of P after n steps *iff there exist $I_1, \ldots, I_{n-1}$ such that $\langle I_1, \ldots, I_{n-1}, M\rangle$ is an evolution stable model of P. We say that propositional atom A of $\mathcal{L}$ is:* true after n steps *iff all stable models of P after n steps contain A;* false after n steps *iff no stable model of P after n steps contains A;* unknown after n steps *otherwise (iff some stable models of P after n steps contain A, and some do not).*

3 Evolving Logic Programs

With this alone, evolving programs are autistic: there is no way one can control or influence their evolution after initialization. To allow for control and influence from the outside, full-fledged evolving programs consider, besides the self-evolution of a program, that new rules may arrive from an outside environment. This influence from the outside may be, at each moment, of various kinds. Notably: observation of facts (or rules) that are perceived at some state; assertion orders directly imposing the assertion of new rules on the evolving program. Both can be represent as EVOLP rules: the former by rules without the assert predicate, and the latter by rules with it. Consequently, we shall represent outside influence as a sequence of EVOLP rules:

Definition 8. *Let P be an evolving program over the language $\mathcal{L}$. An* event sequence *over P is a sequence of evolving programs over $\mathcal{L}$.*

The rules coming from the outside, be they observations or assertion orders, are to be understood as events given at a state, that 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 is not to be assumed by inertia from then onward. With this understanding of event sequence, it is easy to define the evolution stable model of an evolving program influenced by a sequence of events. Basically, a sequence of interpretations is a stable model of a program given a sequence of events, if each I_i in the sequence is a stable model at state i of the trace plus the events added at state i.

Definition 9. *An evolution interpretation $\langle I_1, I_2, \ldots, I_n\rangle$, with evolution trace $\langle P_1, P_2, \ldots, P_n\rangle$, is an* evolution stable model *of P given $\langle E_1, E_2, \ldots, E_k\rangle$ iff for every i ($1 \leq i \leq n$), I_i is a stable model at state i of $P_1 \oplus P_2 \ldots \oplus (P_i \cup E_i)$.*

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}.

Clearly this definition generalizes the one of self evolving programs (if all the sets of events in a sequence are empty, this definition collapses to Definition 5).

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 $\langle E_1, \ldots, E_n, E_{n+1}, \ldots, E_m\rangle$.*

In fact, the evolution stable models of a program given a sequence of events can also be constructed incrementally by means of an operator $\mathcal{T}_P^{ev}$.

Theorem 2. *Let P be an evolving program, and let $\mathcal{I}$ be a set of evolution interpretations of P given the event sequence $\langle E_1, \ldots, E_i\rangle$. The operator $\mathcal{T}_P^{ev}$ is, where $P_1 = P$ and, for $1 < j \leq i+1$, $P_j = \{R \mid assert(R) \in I_{j-1}\}$:*

$$\begin{aligned}\mathcal{T}_P^{ev}(\mathcal{I}, E_{i+1}) = \{ \langle I_1, \ldots, I_i, I_{i+1}\rangle \mid{} & \langle I_1, \ldots, I_i\rangle \in \mathcal{I} \wedge \\ & \wedge\ I_{i+1} \textit{ is a stable model of } P_1 \oplus \ldots \oplus P_i \oplus (P_{i+1} \cup E_{i+1})\}\end{aligned}$$

Consider the sequence: $\mathcal{I}_1 = \{\langle I\rangle \mid I$ is a stable model of $P \cup E_1\}$, and $\mathcal{I}_{i+1} = \mathcal{T}_P^{ev}(\mathcal{I}_i, E_{i+1})$. M is an evolution stable model of length n given the event sequence $\langle E_1, \ldots, E_n\rangle$ iff M belongs to $\mathcal{I}_n$.

A notion of truth after a number of steps given an event sequence can be defined similarly to that provided for self-evolving programs:

Definition 10. *Let P be an evolving program over the language $\mathcal{L}$. We say that a set of propositional atoms M over $\mathcal{L}_{assert}$ is a* stable model of P after n steps given the sequence of events SE *iff there exist $I_1, \ldots, I_{n-1}$ such that $\langle I_1, \ldots, I_{n-1}, M\rangle$ is an evolution stable model of P given SE. We say that propositional atom A of $\mathcal{L}$ is:* true after n steps given SE *iff all stable models after n steps contain A;* false after n steps given SE *iff no stable model after n steps contains A;* unknown after n steps given SE *otherwise.*

4 Examples of Usage

Having formally presented EVOLP, we next show examples of usage. For lack of space, we do not elaborate on how EVOLP actually provides the results shown.

EVOLP was developed as a language capable of undergoing changes in a knowledge base, both by self-evolution as well as imposed from the outside. One immediate application area is that of modelling systems (or agents) that evolve over time and are influenced and/or controlled by the environment.

Example 1. Consider an agent in charge of controlling a lift that receives from outside signals of the form $push(N)$, when somebody pushes the button for going to floor N, or $floor$, when the lift reaches a new 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 the rule: $assert(request(F)) \leftarrow push(F)$[2]. Mark the difference between this rule and the rule $request(F) \leftarrow push(F)$. When the button F is pushed, with the latter rule $request(F)$ is true only at that moment, while with the former $request(F)$ is asserted to the evolving program so that it remains inertially true (until its truth is possibly deleted afterwards).

Based on the pending requests at each moment, the agent must prefer where to go. This could be modelled in the evolving program by the rules:

$$going(F) \leftarrow request(F), not\ unpref(F). \qquad better(F1, F2) \leftarrow at(F),$$
$$unpref(F) \leftarrow request(F2), better(F2, F). \qquad \mid F1 - F \mid < \mid F2 - F \mid.$$

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, $at(F)$ must be incremented/decremented i.e., in EVOLP:

$$\begin{array}{l}
assert(at(F+1)) \leftarrow floor, at(F), going(G),\ G > F.\\
assert(not\,at(F)) \leftarrow floor, at(F), going(G),\ G > F.\\
assert(at(F-1)) \leftarrow floor, at(F), going(G),\ G < F.\\
assert(not\,at(F)) \leftarrow floor, at(F), going(G),\ G < F.
\end{array}$$

When the lift reaches the floor to which it was going, it must open the door. After that, it must remove the pending request for going to that floor:

$$open(F) \leftarrow going(F), at(F). \qquad assert(not\,request(F)) \leftarrow going(F), at(F).$$

Note that there is no need to remove the facts $going(F)$: by removing the $request(F)$, $going(F)$ will no longer be concluded for that F. To illustrate the

[2] In the examples, rules with variables simply stand for all their ground instances.

behaviour of this evolving program, consider that initially the lift is at the 5th floor (i.e. $at(5)$ also belongs to the program), and that the agent receives the sequence $\langle \{push(10), push(2)\}, \{floor\}, \{push(3)\}, \{floor\} \rangle$. This program with this event sequence has, for every n, a single stable model after n steps which, the reader can check, gives the desired results modelling the intuitive behaviour of the lift. Apart from the (auxiliary) predicates to determine the preferred request, for the first 6 steps the stable models are (with the obvious abbreviations):

1 *step* : $\{at(5), push(10), push(2), as(req(10)), as(req(2))\}$
2 *steps* : $\{at(5), req(10), req(2), going(2), floor, as(at(4)), as(not\, at(5))\}$
3 *steps* : $\{at(4), req(10), req(2), going(2), push(3), as(req(3)))\}$
4 *steps* : $\{at(4), req(10), req(2), req(3), going(3), floor, as(at(3)), as(not\, at(4))\}$
5 *steps* : $\{at(3), req(10), req(2), req(3), going(3), open, as(not\, req(3))\}$
6 *steps* : $\{at(3), req(10), req(2), going(2)\}$

The first rule above specifies the effect of the action of pushing a button. Actions specify changes to an evolving knowledge base, and so they can be represented by EVOLP rules specifying those changes. In general, an action, *action*, with preconditions *preconds* and effect *effect* can be represented by: $assert(effect) \leftarrow preconds, action$. Moreover, unlike existing languages for describing actions or updates, EVOLP allows for changes in the rules that specify action effects. Suppose the lift becomes old and sometimes the buttons get stuck and don't work. The knowledge base can be updated accordingly by sending to it an event: $assert(not\, assert(request(F)) \leftarrow stuck(F))$. The effect of this event rule is that, from the moment it is given onward, $request(L)$ is only asserted if there is a $push(F)$ action and $stuck(L)$ is false at the moment the push occurs.

Another interesting aspect which, unlike existing languages, is easy to deal with in EVOLP, is that of incomplete knowledge of the outside environment. Suppose that, in the sequence of events above, instead of the last event the system receives a signal that may (or may not) be a $floor$ signal. This can easily be coded by replacing the corresponding fact $floor$ by the two rules:

$$floor \leftarrow not\, no\_signal. \qquad\qquad no\_signal \leftarrow not\, floor.$$

With this, after 4 steps (and from then onwards), there are 2 evolution stable models: one corresponding to the evolution in case the floor signal is considered; the other, in case it isn't. The truth relation can here be used to determine what is certain despite the undefinedness of the events received. E.g. after 5 steps, it is true that the lift is going to the 3rd floor, has requests for floors 2, 3 and 10, it is unknown whether the lift is at floor 3 or 4 (though it is false that the lift is at any other floor), and unknown whether the door is open.

This example does not exploit the full power of EVOLP. In particular the assertions made are of simple facts only. For illustrating how an assertion of a rule can be used, consider the following example from legal reasoning:

Example 2. Whenever some law is proposed, and before it is made valid it must first be voted by parliament ($v\,(.)$) and then approved by the president ($a\,(.)$).

Further consider a scenario where a law stating that abortions are punishable with jail is proposed (r_1). This can be added to an evolving program as an event:

$$assert\,(assert\,(assert\,(jail\,(X) \leftarrow abort\,(X)) \leftarrow a\,(r_1)) \leftarrow v\,(r_1))$$

Subsequently, after the events $v\,(r_1)$ and $a\,(r_1)$ are observed, the rule $jail\,(X) \leftarrow abort\,(X)$ is in force and anyone who performs an abortion goes to jail. For example, if the knowledge base contains $abort\,(mary)$, then it also entails $jail\,(mary)$. Now suppose that a law stating that abortions are not punishable with jail if there is life danger for the pregnant is proposed (r_2). This rule can be added as:

$$assert\,(assert\,(assert\,(not\,jail\,(X) \leftarrow abort\,(X)\,, danger\,(X)) \leftarrow a\,(r_2)) \leftarrow v\,(r_2))$$

After $v\,(r_2)$ and $a\,(r_2)$ are observed, rule $not\,jail\,(X) \leftarrow abort\,(X)\,, danger\,(X)$ is in force, and anyone who performs an abortion and is in danger would not go to jail. If the knowledge base also contains, e.g., the facts $abort\,(lisa)$ and $danger\,(lisa)$ then it would not entail $jail\,(lisa)$. Note that before the events $v\,(r_2)$ and $a\,(r_2)$ are observed Lisa would go to jail, and even after those events, Mary still goes to jail because $danger\,(mary)$ is not true.

Lack of space prevents us from showing here more examples from other potential application areas of EVOLP. Instead of doing so, we end this section by schematically showing how to code in EVOLP some high level behavioural changes to knowledge bases. Starting with a simple one, which happens to correspond to the behaviour of a LUPS **always** command, suppose we want to state that whenever some conditions *Cond* are met then rule *Rule* is to be asserted. This can be coded by $assert(Rule) \leftarrow Cond$. If such a behaviour is to be included not from the beginning of the program evolution, but rather only after some time, instead of adding that rule in the evolving program one can send, at the appropriate time, an outside event $assert(assert(Rule) \leftarrow Cond)$.

One may also want the rule to be in force after some other condition is met, e.g. if $Cond_1$ then add a rule imposing that whenever $Cond_2$ then *Rule* is asserted. This is not possible to code directly in LUPS, but is easy in EVOLP, due to availability of nested asserts: $assert(assert(Rule) \leftarrow Cond_2) \leftarrow Cond_1$.

In EVOLP the rules coming from the outside are time point events, i.e. do not persist by inertia, and rules in the evolving program are subject to inertia. But one may want to specify, as part of the self evolution of the program, a non inertial rule. For example: "whenever *Cond* is met add $L \leftarrow Body$ in the next state as a time point event, i.e. add it in the next state and then remove it from subsequent states (unless in the following states something else adds the rule again)". The coding of such a rule in EVOLP is not that direct. First one may make the rule $L \leftarrow Body$ conditional on the truth of say $event(L \leftarrow Body)$, and add both the rule and a fact for $event(L \leftarrow Body)$ whenever *Cond* is met:

$$assert(event(L \leftarrow Body)) \leftarrow Cond.$$
$$assert((L \leftarrow Body, event(L \leftarrow Body))) \leftarrow Cond.$$

Now, if at any state, for some rule R, $event(R)$ is true then we must make it false at the next state. But only unless the rule is not going being added as an event, for some other reason, at that next state, i.e. unless $assert(event(R))$ is not itself true. And this can be coded in EVOLP by:

$$assert(not\, event(R)) \leftarrow event(R), not\, assert(event(R))$$

This rule illustrates one usage of *assert* in rule bodies. It is needed here to state that some rule is to be asserted unless some other rule is not going to be asserted. It is easy to think of examples where such constructs are needed. E.g. in chain updates to databases where one might want to state that if something is going to be asserted, then something else must be asserted, etc. This kind of chain updates, difficult to model in LUPS, are easy to model in EVOLP.

5 Concluding Remarks

We have presented the foundational framework of evolving logic programs, to express and reason about dynamic knowledge bases. Somewhat related approaches can be found in the literature, and some words on comparison are in order.

Stemming from the very initial motivation for setting forth this framework, it is mandatory that we begin with comments on the relation with LUPS [2] and EPI [7]. Even though their semantics are based on a notion of stable models, where a sequence of programs can have several stable models, the conditions of the LUPS and EPI commands are evaluated against a single valuation based on the union (in [2]) or intersection (in [11,7]) of all stable models. As a result, each state transition, unlike for EVOLP, produces a single program i.e., the evolution does not involve branching. This prevents a direct embedding of LUPS and EPI in EVOLP, but it is possible to informally assume a modified version of these languages where the conditions would be evaluated at each individual stable model, for the sake of comparison. Conversely, EVOLP could be easily modified in the opposite way so as to not have branching, by evaluating the rules bodies at the union (or intersection) of the stable models to construct the next program in the sequence. With this caveat, we are ready to state that LUPS and EPI commands are all specifiable within the language of EVOLP. In the previous section, we have already shown how to specify in EVOLP a couple of LUPS commands. This alone is a major achievement inasmuch one of our motivations was to simplify the rather complex extant languages of updates. But EVOLP endorses more than LUPS or EPI, e.g.: by employing arbitrary nesting of assertions, one can for example specify updates whose conditions span more than one state, e.g. assertions that depend on a sequence of events as in Example 2; since persistent commands in LUPS and in EPI can be encoded as plain rules in EVOLP, they can be the subject of updates by other rules, as illustrated in Example 1. Finally, unlike in LUPS and EPI, where update commands and object level rules are written in different languages, by unifying states and state transitions in the same language and semantics, EVOLP constitutes a simple framework not just to understand and write specifications in, but to study the general properties of updates as well .

A logic-based modelling of simple updates is described in [5], where a deductive database approach to planning is presented. In [5], STRIPS-like plans are modelled by means of Datalog$_{1s}$ programs [6] which have a stable model semantics amenable to efficient implementation. Updates are used to model state changes determined by executing planning actions. This approach shares with ours the choice of the stable model semantics to provide a logical characterization of updates. However, [5] models only updates of positive literals, while our approach accounts for (possibly nested) updates of arbitrary rules. Moreover, while [5] syntactically hacks program evolution inside programs by using stage arguments, EVOLP models program evolution at the semantics level, without adding arguments to predicates.

Action languages [9] and Event Calculus [10] have been proposed to describe and reason about the effects of actions and events. Intuitively, while such frameworks and EVOLP are both concerned with modelling changes, the former focus on the notions of causality, fluents and events, while EVOLP focusses its features on the declarative evolution of a general knowledge base.

Transaction Logic (TL) [3] is also a LP language for updates. TL is concerned with finding, given a goal, the appropriate transactions (or updates) to an underlying knowledge base in order to satisfy the goal. Contrary to EVOLP, the rules that specify changes are separate from the knowledge base itself, making it impossible to change them dynamically. Moreover the whole process in TL is query driven. On the contrary, in EVOLP goals are not needed to start an update process: an EVOLP knowledge base evolves by itself. Also, we have no notion of finding updates to satisfy a goal: the updates are given, and EVOLP is concerned with determining the meaning of the KB after those given updates. Determining such updates amount to a form of abduction over EVOLP programs, driven by goals or integrity constraints. Similar arguments apply when comparing EVOLP to Prolog asserts, though in this case there is the additional argument of there not being a clear semantics (with attending problems when asserts are subject to backtracking).

Space limitations prevent us discussing quite a number of other interesting issues and directions for further work. Nevertheless, we briefly touch upon some of them here.

A *transformational definition* of the semantics of EVOLP programs has already been defined. Informally, the idea is to label program literals with a state argument so the truth of L^n denotes the truth of literal L after n evolution steps. The interest of the transformational definition is two-fold. On the one hand, it provides an alternative characterization of the semantics of EVOLP programs in terms of the standard semantics of generalized programs. On the other hand, it is the basis on which we have developed a query-based implementation of the EVOLP language (available from http://centria.fct.unl.pt/~jja/updates/).

The proffered semantics models *consistent* evolutions of programs. Indeed, according to the definition of evolution stable model, a program is considered to be inconsistent after n evolution steps if it has at least one evolution stable model of length $(n-1)$ but no evolution stable model of length n. An important direction for future work is to extend the basic semantics to allow programs to continue their computations even if they reach some inconsistency. Different forms of dealing with contradiction can be considered ranging from naive removal of directly contradicting rules, to paraconsistency, to forms of belief revision.

A very interesting perspective to explore is the use of the EVOLP language to specify evolving *agents* that interact with the external environment. While the semantics of evolving programs accounts for rules and updates that may arise in the external environment, we have not addressed here the issues concerning the overall software architecture of an evolving agent. For instance, the agent architecture will be in charge of suitably filtering perceptions and requests that arrive from the outside, as well as of mastering the asynchronicity between the agent and the external environment. Another important aspect to be explored concerns the "acting" abilities of agents. These may be naturally expressed by means of outgoing updates (directed towards other agents or the external environment). Yet another direction discerns between sets of controlled and of uncontrolled events in the environment.

Finally, the availability of a description of the possible behaviours of evolving programs is a firm ground for performing static *program analysis* before putting agents to work with the environment. It is of interest to develop resource-bounded analyses, quantitative and qualitative, of evolving programs along the lines of [4] so as to statically determine both their possible and invariant beliefs.

Acknowledgements

This work was partially supported by a bilateral project ICCTI-CNR, and by project FLUX (POSI/40958/SRI/2001).

References

1. 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):43–70, 2000.
2. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS : A language for updating logic programs. *Artificial Intelligence*, 138(1-2), 2002. A short version appeared in M. Gelfond et al., LPNMR-99, LNAI 1730, Springer.
3. A. Bonner and M. Kifer. Transaction logic programming. In David S. Warren, editor, *ICLP-93*, pages 257–279. The MIT Press, 1993.
4. A. Brogi. Probabilistic behaviours of reactive agents. *Electronic Notes in Theoretical Computer Science*, 48, 2001.
5. A. Brogi, V.S. Subrahmanian, and C. Zaniolo. The logic of totally and partially ordered plans: A deductive database approach. *Annals of Mathematics and Artificial Intelligence*, 19((1,2)):27–58, 1997.
6. J. Chomicki. Polynomial-time computable queries in temporal deductive databases. In *PODS'90*, 1990.
7. T. Eiter, M. Fink, G. Sabbatini, and H Tompits. A framework for declarative update specifications in logic programs. In *IJCAI'01*, pages 649–654. Morgan-Kaufmann, 2001.
8. M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In R. Kowalski and K. Bowen, editors, *ICLP'88*, pages 1070–1080. MIT Press, 1988.
9. M. Gelfond and V. Lifschitz. Action languages. *Linkoping Electronic Articles in Computer and Information Science*, 3(16), 1998.
10. R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

11. J. A. Leite. A modified semantics for LUPS. In P. Brazdil and A. Jorge, editors, *EPIA-01*, volume 2258 of *LNAI*, pages 261–275. Springer, 2001.
12. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *KR'92*. Morgan-Kaufmann, 1992.