A Logic Programming System for Evolving Programs with Temporal Operators

José Júlio Alferes[1], Alfredo Gabaldon[1], and João Leite[1]

CENTRIA, Universidade Nova de Lisboa, Portugal

Abstract. Logic Programming Update Languages were proposed as an extension of logic programming that allows modeling the dynamics of knowledge bases where both extensional (facts) and intentional knowledge (rules) may change over time due to updates. Despite their generality, these languages do not provide a means to directly access past states of the evolving knowledge. They are limited to so-called Markovian change, i.e. changes entirely determined by the current state.
We remedy this limitation by extending the Logic Programming Update Language EVOLP with LTL-like temporal operators that allow referring to the history of the evolving knowledge base, and show how this can be implemented in a Logic Programming framework.

1 Introduction

While belief update in the context of classical knowledge bases (KBs) has traditionally received significant devotion [1], only in the last decade have we witnessed increasing attention to this topic in the context of non-monotonic KBs, notably using logic programming (LP) [2–12]. Chief among the results of such efforts are several semantics for sequences of logic programs (dynamic logic programs) with different properties, and the so-called LP Update Languages: LUPS [11], EPI [4], KABUL [3] and EVOLP [10].

LP Update Languages are extensions of LP designed for modeling dynamic, non-monotonic KBs represented by logic programs. In these KBs, both the extensional part (a set of facts) and the intentional part (a set of deductive rules) may change over time due to updates. In these languages, special types of rules are used to specify updates to the current KB leading to a subsequent KB. LUPS, EPI and KABUL offer a very diverse set of update commands, each specific for one particular kind of update (assertion, retraction, etc). On the other hand, EVOLP follows a simpler approach, staying closer to traditional LP.

A generalization of EVOLP with temporal operators, called EVOLP$_T$, was introduced in [13] and its usage illustrated in the context of multi-agent systems. In this paper we present a simplified reformulation of the definition of the semantics (Sec. 2) and introduce an implementation of EVOLP$_T$ programs[1] (Sec. 3).

[1] Freely available from http://centria.di.fct.unl.pt/~ jja/updates/

1.1 Intuitions and Motivating Example

EVOLP (Evolving Logic Programming) generalizes Answer Set Programming [14] by allowing the specification of a program's own evolution, arising both from self (i.e. internal to the program) updating and from external updating originating in the environment. From a syntactical point of view, evolving programs are just generalized logic programs extended with (possibly nested) assertions appearing in the heads and bodies of rules. From a semantic point of view, a model-theoretic characterization is offered of the possible evolutions of such programs by means of the so-called *evolving stable models*, which are sequences of interpretations. Each interpretation in the sequence describes, at the corresponding evolution step, what is true and the possible next-step evolutions.

Despite their generality, none of the update languages available provides means to directly access past states of an evolving KB. These languages were designed for domains where updates are Markovian, that is, entirely determined by the current state. There are many scenarios, however, where the update dynamics are non-Markovian and EVOLP$_T$ was proposed for such cases where non-Markovian control is required. The following example illustrates this.

Consider a KB of user access policies for a number of computers at different locations. A login policy may say, e.g., that after the first failed login a user is warned by sms and if there is another failed login the account is blocked. This policy could be expressed by the following two rules:

$$sms(User) \leftarrow \Box(not\, sms(User)), fLogin(User, IP).$$
$$block(User) \leftarrow \Diamond(sms(User)), fLogin(User, IP).$$

where $fLogin(User, IP)$ represents an external event of a failed login. The ability to represent such external influence on the contents of a KB is one of the features of EVOLP (and of EVOLP$_T$). The symbols $\Diamond$ and $\Box$ represent Past Linear Temporal Logic (Past LTL) like operators. $\Diamond\varphi$ means that there is a past state where φ was true and $\Box\varphi$ means that φ was true in all past states.

Now suppose that we want to model some updates made by the system administrator. For example, adding a new policy consisting in blocking a user after the first failed login attempt if the user has an IP address from a "bad domain", and that an sms is not sent him. This is captured by the rules:

$$block(User) \leftarrow fLogin(User, IP), domain(IP, BadDom).$$
$$not\, sms(User) \leftarrow fLogin(User, IP), domain(IP, BadDom).$$

with *BadDom* instantiated to the domain in question. Whether a domain is bad or not, however, depends on the particular machine. In this case, the sys admin may want to send an update to all machines so that the above rules are added to each machine's policy only for domains which are bad according to the machines' current history. The sys admin issues the following update to every machine, saying that the above new rules are to be asserted if the domain has

been considered a bad one since the last failed attempt:

$$\begin{aligned} assert(\ block(User) \leftarrow fLogin(User, IP), domain(IP, Dom)\ ) \leftarrow \\ \mathbf{S}(badDomain(Dom),\ fLogin(Usr2, IP2)), \\ domain(IP2, Dom). \\ assert(\ not\, sms(User) \leftarrow fLogin(User, IP), domain(IP, BadDom)\ ) \leftarrow \\ \mathbf{S}(badDomain(Dom),\ fLogin(Usr2, IP2)), \\ domain(IP2, Dom). \end{aligned}$$

where $badDomain(Dom)$ is machine specific. The symbol $\mathbf{S}$ represents an operator similar to the Past LTL operator "since". The intuitive meaning of $\mathbf{S}(\psi, \varphi)$ is: at some point in the past φ was true, and ψ has always been true since then. The *assert* construct is one of the main features of EVOLP. It allows one to specify updates to a KB, leaving to its semantics the task of dealing with contradictory rules such as the one that specifies that an sms should be sent if it is the first failure, and the one that specifies otherwise if the domain is bad.

2 Evolving Logic Programs with Temporal Operators

EVOLP [10] is a logic programming language for specifying dynamic KBs. Change is specified through two constructs: a special predicate $assert/1$ for adding new rules to the KB, and negated rule heads which have the effect of invalidating previously added rules in the KB. EVOLP$_T$ extends EVOLP with Past LTL like operators $\bigcirc(G)$, $\Diamond(G)$ $\Box(G)$, and $\mathbf{S}(G_1, G_2)$, which intuitively mean, respectively: G is true in the previous state; there is a state in the past in which G is true; G is always true in the past; and G_2 is true at some state in the past, and since then until the current state G_1 is true.

Arbitrary nesting of these operators as well as negation-as-failure in front of their arguments is allowed. On the other hand, unlike *not*, the temporal operators are not allowed to appear in the head of rules. The only restriction on the body of rules is that negation is allowed to appear in front of atoms and temporal operators only. The formal definition of the language and programs in EVOLP$_T$ is as follows.

Definition 1 (EVOLP$_T$). Let $\mathcal{L}$ be any propositional language not containing symbols *assert*, $\bigcirc$, $\Diamond$, $\mathbf{S}$ and $\Box$. The EVOLP$_T$ *b-literals*, *t-formulae*[2] and rules are inductively defined as follows:

1. Propositional atoms in $\mathcal{L}$ are (atomic) b-literals.
2. If G_1 and G_2 are b-literals then so are $\bigcirc(G_1)$, $\Diamond(G_1)$, $\mathbf{S}(G_1, G_2)$ and $\Box(G_1)$. These b-literals are the t-formulae.
3. If G is a t-formula or an atomic b-literal, then $not\, G$ is a b-literal.
4. If G_1 and G_2 are b-literals, then (G_1, G_2) is a (conjunctive) b-literal.

[2] b-literal stands for "body literal" and t-formula for "temporal-formula".

5. If L_0 is an atomic b-literal, A, or its negation, $not\, A$, and $G_1, \ldots, G_n$ are b-literals, then $L_0 \leftarrow G_1, \ldots, G_n$ is a *rule*.
6. If R is a rule, then $assert(R)$ is an atomic b-literal.
7. Nothing else is a b-literal, t-formula, or rule.

Atomic b-literals will simply be called *atoms*. A b-literal is called *objective* if *not* does not appear in it. An EVOLP$_T$ program is a (possibly infinite) set of rules.

The following is a 'legal' EVOLP$_T$ rule:

$$assert(a \leftarrow not\, \Diamond(b)) \leftarrow not\, \Box(not\, \Diamond(b, not\, assert(c \leftarrow d))).$$

Notice the nesting of temporal operators and the appearance of negation, conjunction and assert under the scope of the temporal operators.

The following are not 'legal' EVOLP$_T$ rules:

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

In the first rule, $\Box(b)$ appears in the argument rule $\Box(b) \leftarrow a$, but temporal operators are not allowed in the head of rules. The second rule applies negation to a conjunctive b-literal, and the third rule has double negation. But negation is only allowed in front of atomic b-literals and t-formulae.

The definition of the semantics of EVOLP$_T$ is based on sequences of interpretations (sets of atoms), $\langle I_1, \ldots, I_n \rangle$, called *evolution interpretation*. Each interpretation I_i contains the atoms that hold at state i of the evolution, and a sequence represents a possible evolution of an initial program after the given n evolution steps.

Satisfaction of b-literals by an evolution interpretation is defined as follows.

Definition 2 (Satisfaction of b-literals). Let $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ be an evolution interpretation of a program P and let G and G' be b-literals. Then

$$\begin{array}{lll}
\langle I_1, \ldots, I_n \rangle \models A & \text{iff} & A \in I_n \text{ and } A \text{ is an atom.} \\
\langle I_1, \ldots, I_n \rangle \models not\, G & \text{iff} & \langle I_1, \ldots, I_n \rangle \not\models G. \\
\langle I_1, \ldots, I_n \rangle \models G, G' & \text{iff} & \langle I_1, \ldots, I_n \rangle \models G \text{ and } \langle I_1, \ldots, I_n \rangle \models G'. \\
\langle I_1, \ldots, I_n \rangle \models \bigcirc(G) & \text{iff} & n \geq 2 \text{ and } \langle I_1, \ldots, I_{n-1} \rangle \models G. \\
\langle I_1, \ldots, I_n \rangle \models \Diamond(G) & \text{iff} & n \geq 2 \text{ and } \exists i < n : \langle I_1, \ldots, I_i \rangle \models G. \\
\langle I_1, \ldots, I_n \rangle \models \mathbf{S}(G, G') & \text{iff} & n > 2,\ \exists i < n : \langle I_1, \ldots, I_i \rangle \models G' \text{ and} \\
 & & \forall i < j < n : \langle I_1, \ldots, I_j \rangle \models G. \\
\langle I_1, \ldots, I_n \rangle \models \Box(G) & \text{iff} & \forall i < n : \langle I_1, \ldots, I_i \rangle \models G.
\end{array}$$

Given an evolution interpretation, an *evolution trace* represents one of the possible evolutions of the KB.

Definition 3. The *evolution trace* associated with an evolution interpretation $\mathcal{I}$ of a program P is the sequence of programs $\langle P_1, P_2, \ldots, P_n \rangle$ where:

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

Our first step towards defining the semantics of EVOLP$_T$ programs consists in defining an operator that eliminates t-formulae from a program by evaluating them against an evolution interpretation.

Definition 4 (Elimination of Temporal Operators). Let $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ be an evolution interpretation and $L_0 \leftarrow G_1, \ldots, G_n$ a rule. The rule resulting from the elimination of temporal operators given $\mathcal{I}$,

$$El(\mathcal{I}, L_0 \leftarrow G_1, \ldots, G_n)$$

is obtained by replacing by *true* every t-formula G_t in the body such $\mathcal{I} \models G_t$ and by replacing all remaining t-formulae by $false$, where constants *true* and $false$ are defined, as usual, such that the former is true in every interpretation and the latter is not true in any interpretation.

The program resulting from the elimination of temporal operators given $\mathcal{I}$, $El(\mathcal{I}, P)$ is obtained by applying El to each of the program's rules.

Before we define the models of a program, we need to introduce some notation. Let $\mathcal{P} = \langle P_1, ..., P_n \rangle$ be a sequence of programs. For $1 \leq s \leq n$, $\rho_s(\mathcal{P})$ denotes the multiset of all the rules appearing in the subsequence $P_1, ..., P_s$. For a set atoms I, $\hat{I} = I \cup \{not\_A \mid A \notin I\}$. If r is a rule $L_0 \leftarrow L_1, \ldots, L_n$, then $H(r) = L_0$ (dubbed the head of the rule) and $B(r) = L_1, \ldots, L_n$ (dubbed the body of the rule). We say r and r' are *conflicting rules*, denoted by $r \bowtie r'$, iff $H(r) = A$ and $H(r') = not\, A$ or $H(r) = not\, A$ and $H(r') = A$. Let $\mathcal{P}$ be a program, I an interpretation, and $1 \leq s \leq n$. The following sets originate form the semantics of Dynamic Logic Programs [2].

$$\begin{aligned} Def_s(\mathcal{P}, I) = \{&not\, A \mid A \text{ is an objective atom,} \\ &\nexists r \in \rho_s(\mathcal{P}), H(r) = A, I \vDash B(r)\}. \\ Rej_s(\mathcal{P}, I) = \{&r \mid r \in P_i, \exists r' \in P_j, i \leq j \leq s, r \bowtie r', I \vDash B(r')\}. \end{aligned}$$

Intuitively, $Def_s(\mathcal{P}, I)$ is the set of 'default' b-literals, $not\, A$, such that A has never held at any state up to s. $Rej_s(\mathcal{P}, I)$ is the set of rules that are 'rejected' (i.e. disabled) at state s because a conflicting rule whose body is satisfied was added in a more recent or the same state than the state where the rejected rule was added.

Finally, $least(P)$ denotes the least model of the definite program obtained from a program P without t-formulae by replacing every b-literal $not\, A$, where A is an atom, by a new atom not_A.

Definition 5 (Dynamic Stable Models). Let $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ be a sequence of EVOLP$_T$ programs and I an interpretation. I is a *dynamic stable model* of $\mathcal{P}$ at state s, $1 \leq s \leq n$ iff

$$\hat{I} = least\left([\rho_s(\mathcal{P}) - Rej_s(\mathcal{P}, I)] \cup Def_s(\mathcal{P}, I)\right).$$

Definition 6 (Evolution Stable Models). Let $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ be an evolution interpretation of an EVOLP$_T$ program P and $\mathcal{P}$ be the corresponding evolution trace. Then $\mathcal{I}$ is an *evolution stable model* of P iff for every $1 \leq s \leq n$, I_s is a dynamic stable model of $El(\mathcal{I}, \mathcal{P})$ at s.

In addition to allowing the specification of updates in the KB itself, EVOLP$_T$ also allows, as does its predecessor EVOLP, incorporating changes caused by external events. These events may be facts/rules representing observations made at some stage of the evolution or assertion commands specifying new update directives to be added to the KB. Both types of event can be represented as EVOLP$_T$ rules: the former by rules without the assert predicate in the head, and the latter by rules with it. Formally, a sequence of such events is just a sequence of programs over the same language:

Definition 7. Let L be the language of an evolving program P. An *event sequence* over P is a sequence of evolving programs over L.

Definition 8 (Evolution Stable Models with Events). Let $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ be an evolution interpretation of an EVOLP$_T$ program P and $\langle P_1, \ldots, P_n \rangle$ be the corresponding evolution trace. Then $\mathcal{I}$ is an *evolution stable model* of P given event sequence $\langle E_1, E_2, \ldots, E_n \rangle$ iff for every s, $1 \leq s \leq n$, I_s is a dynamic stable model of $El(\mathcal{I}, \langle P_1, P_2, \ldots, (P_s \cup E_s) \rangle)$ at s.

Since multiple different evolutions of the same length may exist, evolution stable models alone do not determine a truth relation. But one such truth relation can be defined, in the usual way, based on the intersection of models:

Definition 9 (Stable Models after n Steps given Events). Let P be an EVOLP$_T$ program. We say that a set of atoms M is a *stable model of P after n steps given the sequence of events $\mathcal{E}$* 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 $\mathcal{E}$.
We say that an atom A is:

- *true after n steps given $\mathcal{E}$* iff all stable models after n steps contain A;
- *false after n steps given $\mathcal{E}$* iff no stable model after n steps contains A;
- *unknown after n steps given $\mathcal{E}$* otherwise.

It is worth noting that basic properties of Past LTL operators carry over to EVOLP$_T$. In particular, in EVOLP$_T$, as in LTL, some of the operators are not strictly needed, since they can be rewritten in terms of the others:

Proposition 1. *Let $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ be an evolution stable model over a set L of b-literals. Then, for every $G \in L$:*

- $\mathcal{I} \models \Box(G)$ *iff* $\mathcal{I} \models not\, \Diamond(not\, G)$,
- $\mathcal{I} \models \Diamond(G)$ *iff* $\mathcal{I} \models \mathbf{S}(true, G)$.

Moreover, it should also be noted that EVOLP$_T$ is a generalization of EVOLP in the sense that when no temporal operators appear in the program and in the sequence of events, then evolution stable models coincide with those of EVOLP. A further, immediate consequence of this fact is that if the sequence of events is empty and predicate $assert/1$ does not occur in the program, evolution stable models as defined for EVOLP$_T$ coincide with answer-sets.

3 EVOLP$_T$ Implementations

We have developed two implementations of EVOLP$_T$. One follows the evolution stable models semantics defined above, while the second one computes answers to existential queries under the well-founded semantics [15]. The implementations rely on two consecutive program transformations: the first transforms an EVOLP$_T$ program into an EVOLP one, i.e. temporal operators are eliminated. The second transformation, which is based on previous work [16], takes the result of the first and generates a normal logic program.

We start by defining the first program transformation, then give some intuitions on the second transformation. That is followed by a description of the actual implementation and its usage and then some details about the other implementation.

3.1 Program transformations

The transformation of EVOLP$_T$ programs and sequences of events into EVOLP mainly consists in eliminating the t-formulae by introducing new propositional atoms and rules that encode the dynamics of the temporal operators. We start by defining the target language of the resulting EVOLP programs.

Let P be an EVOLP$_T$ program and $\mathcal{E}$ a sequence of events in a propositional language $\mathcal{L}$. The target language is obtained from $\mathcal{L}$ by adding a new propositional variable for every non-atomic b-literal that appears in P and $\mathcal{E}$. Recall that atomic b-literals are the propositional atoms and atoms of the form $assert(R)$. Formally:

Definition 10 (EVOLP Target language). Let P and $\mathcal{E}$ be an EVOLP$_T$ program and a sequence of events, respectively, in a propositional language $\mathcal{L}$. Let $\mathcal{G}(P, \mathcal{E})$ be the set of all non-atomic b-literals that appear in P or $\mathcal{E}$.

The EVOLP target language is $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E}) = \mathcal{L} \cup \{'L' \mid L \in \mathcal{G}(P, \mathcal{E})\}$, where by $'L'$ we mean a propositional variable whose name is the (atomic) string of characters that compose the formula L (which is assumed not to occur in $\mathcal{L}$).

The transformation takes the rules in both program and events, and replaces all occurrences of t-formulas and conjunctions in their bodies by the corresponding new propositional variables in the target language. Moreover, extra rules are added to the program for encoding the behaviour of the operators.

Definition 11 (Transformed EVOLP program). Let P be an EVOLP$_T$ program and $\mathcal{E} = \langle E_1, E_2, \ldots, E_n \rangle$ be a sequence of events in a propositional language $\mathcal{L}$. Then $Tr_E(P, \mathcal{E}) = (T_P, \langle T_{E_1}, \ldots, T_{E_n} \rangle)$ is a pair consisting of an EVOLP program (i.e., without temporal operators) and a sequence of events, both in language $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, defined as follows:

1. **Rewritten program rules.** For every rule r in P (resp. each of the E_i), T_P (resp. T_{E_i}) contains a rule obtained from r by replacing every t-formula G in its body by the new propositional variable $'G'$;

2. **Previous-operator rules.** For every propositional variable of the form $'\bigcirc(G)'$, appearing in $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, T_P contains:
   $assert('\bigcirc(G)') \leftarrow' G'.$ $\qquad$ $assert(not\,'\bigcirc(G)') \leftarrow not\,'G'.$

3. **Sometimes-operator rule.** For every propositional variable of the form $'\Diamond(G)'$, appearing in $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, T_P contains:
   $assert('\Diamond(G)') \leftarrow' G'.$

4. **Since-operator rules.** For every propositional variable of the form $'\mathbf{S}(G_1, G_2)'$, appearing in $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, T_P contains:
   $assert('\mathbf{S}(G_1, G_2)') \leftarrow' G_1',' \bigcirc(G_2)'.$
   $assert(assert(not'\mathbf{S}(G_1, G_2)') \leftarrow not\,'G_1') \leftarrow assert('\mathbf{S}(G_1, G_2)').$

5. **Always-operator rules.** For every propositional variable of the form $'\Box(G)'$, appearing in $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, T_P contains:
   $'\Box(G)' \leftarrow' G',\ not \bigcirc true.$ $\qquad$ $assert(not\,'\Box(G)') \leftarrow not\,'G'.$

6. **Conjunction and negation rules.** For every propositional variables of the form $'not\,G'$, or of the form $'G_1, G_2'$ appearing in $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, T_P contains, respectively:
   $'not\,G' \leftarrow not\,'G'$
   $'G_1, G_2' \leftarrow' G_1',' G_2'.$ $\qquad$ $'G_1, G_2' \leftarrow' G_2',' G_1'.$

The correctness of this transformation is established by the theorem:

Theorem 1 (Correctness of the EVOLP transformation). Let P and $\mathcal{E} = \langle E_1, \ldots, E_k \rangle$ be, respectively, an EVOLP$_T$ program and a sequence of events in a propositional language $\mathcal{L}$, and let $Tr_E(P, \mathcal{E}) = (T_P, \langle T_{E_1}, \ldots, T_{E_k} \rangle)$. Then $M = \langle I_1, \ldots, I_n \rangle$ is an evolution stable model of P given $\mathcal{E}$ iff there exists an evolution stable model $M' = \langle I_1', \ldots, I_n' \rangle$ of T_P given $\mathcal{E}$ such that $I_1 = (I_1' \cap \mathrm{L}), \ldots, I_n = (I_n' \cap \mathrm{L})$.

Proof. (Sketch) The proof, that can only be sketched here due to lack of space, proceeds by induction on the length of the sequence of interpretations, showing that the transformed atoms corresponding to t-formulae satisfied in each state, and some additional assert-literals guarantying the assertion of t-formulae, belong to the interpretation state.

For the induction step, the rules of the transformation are used to guarantee that the new propositional variables belong to the interpretation of the transformed program whenever the corresponding temporal b-literals belong to the interpretation of the original EVOLP$_T$ program. For example, if some $G \in I_i$ then, according to the EVOLP$_T$ semantics, for every $j > i$ $\Diamond(G) \in I_j$; by induction hypothesis, $'G' \in I_i'$ and the "sometime-operator rule" guarantees that $'\Diamond(G)'$ is added to the subsequent program and so, since no rule for $not\,'\Diamond(G)'$ is added in the transformation, for every $j > i$ $'\Diamond(G') \in I_j'$. As another example, note that the first "previous-operator" rule is similar to the "sometime-operator rule" and the second adds the fact $not\,'\bigcirc(G)'$ in case $not\,'G'$ is true; so, as in

the EVOLP$_T$ semantics, $' \bigcirc (G)' \in I'_{i+i}$. A similar reasoning is applied also for the since-operator and always-operator rules.

To account for the nesting of temporal operators first note that the transformation adds the above rules for all possible nestings. Since this nesting can be combined with conjunction and negation, as per the definition of the syntax of EVOLP$_T$ (Def. 1), care must be taken with the new propositional variables that stand for those conjunctions and negations. This is taken care the the conjunction and negation rules, which guarantee that a new atom with a conjunction is true in case the b-literals in the conjunction are true, and that a new atom with the negation of a b-literal is true in case the negation of the b-literal is true. □

As explained above, the implementation proceeds by transforming the resulting EVOLP program into a normal logic program. This is done by using the results of [16]. In a nutshell[3], for a program P and a sequence of events $\langle E_1, \ldots, E_n \rangle$ in $\mathcal{L}$, the language is first extended with, for every proposition $A \in \mathcal{L}$, new propositional variables A^j and A^j_{neg} for every $j \leq n$, and $rej(A^j, i)$ and $rej(A^j_{neg}, i)$ for every $j \leq n$ and $i \leq j$. Intuitively, these new variables stand for, respectively: the truth (resp. falsity) of A in P_j, and the rejection of rules with head A at I_j due to the existence of a rule in P_i.

All rules $L \leftarrow Body$ in P and in each E_i are transformed in this extended language into, resp., $L^j \leftarrow Body^j, not\, rej(L^j, 1)$ for every $j \leq n$, and $L^i \leftarrow Body^i, not\, rej(L^j, i)$. Further rules are added to account for the semantics of EVOLP. For example, for every rule $r = (L \leftarrow Body)$ and all $1 < i \leq j$ s.t. $(assert(r))^{i-1}$ is the head of some transformed rule, add also the rule

$$L^j \leftarrow Body^j, (assert(r))^{i-1}, not\, rej(L^j, i)$$

As another example, to account for default negation, for every $j \leq n$ and for every A, add $A^j_{neg} \leftarrow not\, rej(A^j_{neg}, 0)$. Other rules are added (see [16]) to account for rejection, and for guaranteeing that in every interpretation I^j either A^j or A^j_{neg} belong to it (and not both).

Given the results of [16], proving a one to one relation between the stable models of the so transformed normal program and the stable models of the original EVOLP program, and the result of Theorem 1, it is clear that the composition of these two transformation is correct, i.e. the stable models of the obtained normal logic program correspond to the stable models of the original EVOLP$_T$ program plus events.

3.2 Implementations and usage

Our implementation relies on the above described composed transformation. More precisely, the basic implementation takes an EVOLP$_T$ program and a sequence of events and preprocesses it into a normal logic program.

The preprocessor that is available at http://centria.di.fct.unl.pt/~ jja/updates/ is implemented in Prolog. It takes a file that starts with an EVOLP$_T$ program,

[3] For details see [16].

where the syntax is just as described in Def. 1, except that the rule symbol is `<-` and the temporal operators are `previous/1`, `sometime/1`, `since/2`, `always/1` standing for the temporal operators $\bigcirc(G)$, $\Diamond(G)$, $\mathbf{S}(G_1, G_2)$ and $\Box(G)$, respectively. The EVOLP$_T$ program is ended by a fact `newEvents`. The rules after this fact constitute the first set of events, and is again ended by a fact `newEvents`, after which the rules for the second set of events follow, etc.

The preprocessing is done by combining the two transformation in a single step, rather than having them in sequence as described in the previous subsection. This is done for efficiency of the preprocessors, and the combination simply combines in sequences each of the rules in both transformations. Moreover, instead of creating new atomic names, as done in both transformations, the preprocessor uses Prolog terms for the new propositions accounting for the b-literal (e.g. it uses a term `sometime(a)` instead of $'\Diamond(p)'$ or $'sometime(p)'$), which eases the processing of nested temporal operators; it adds new arguments to predicates $A(j)$, instead of creating new propositions of the form A^j, as in the second transformation.

The programs obtained by the Prolog preprocessor can then run in any answer-set solver to obtain the set of the stable models of the original EVOLP$_T$ program and events. We have tested the implementation using the lparse grounder and the smodels solver(http://www.tcs.hut.fi/Software/smodels/). The implementation can also take advantage of the implementation of EVOLP and interface described in [17]. For this, we provide a version that one performs the first transformation, that is to be fed to the (java-based) implementation of [17] which, in turn, performs the second transformation and computes the stable models (using smodels).

Instead of computing the stable models of the resulting normal program, one may compute its well-founded model [15]. This provides a (3-valued) model which is sound, though not complete, w.r.t. the intersection of all stable models. I.e. if an atom $A(n)$ belongs to the wf-model then A is true in all stable models of the program and events after n steps; if $not\, A(n)$ belongs to the wf-model then A belongs to no stable models after n steps; if neither $A(n)$ nor $not\, A(n)$ belong to the wf-model, then nothing can be concluded.

Despite the incompleteness, the wf-model has the advantage of having polynomial complexity and allowing for (top-down) query-driven procedures. With this in mind, we have done another implementation, also available online, that besides the preprocessor also includes a meta-interpreter that answers existential queries under the well founded semantics. The meta-interpreter is implemented in XSB-Prolog, and relies on its tabling mechanisms for computing the wf-model. For top-down querying we provide a top goal predicate `G after I in (N1,N2)` which, given a goal `G` and two integers `N1` and `N2`, returns in `I` all integers between `N1` and `N2` such that in all stable models after `I` steps, `G` is true. This XSB-Prolog implementation also allows for a more interactive usage, e.g. allowing to add events as they occur (and adjusting the transformation on the fly), separately from the initial programs, and querying the current program (after as many steps as the number of events given).

4 Related Work and Conclusions

The language EVOLP$_T$ generalizes its predecessor EVOLP by providing rules that, via Past LTL like modalities, may refer to past states in the evolution of a KB. Here, in addition to a simplified reformulation of its semantics, we have presented a provably correct transformation of EVOLP$_T$ programs and two implementations based on it.

The use of temporal logic in computer science is widespread. Here we would like to mention some of the most closely related work. Eiter et al. [5] present a very general framework for reasoning about evolving knowledge bases. This abstract framework allows the study of different approaches to logic programming knowledge base update, including those specified in LUPS, EPI, and KABUL. For the purpose of verifying properties of evolving knowledge bases in this language, they define a syntax and semantics for Computational Tree Logic (CTL), a branching temporal logic, modalities. While in [5] temporal logic is only used for verifying meta-level properties, in EVOLP$_T$ temporal operators are used in the object language to specify the behavior of an evolving knowledge base.

In the area of reasoning about actions, [18] describes an extension of the action language $\mathcal{A}$ with Past LTL operators, which allows formalizing actions whose effects depend on the evolution of the described domain. On a similar vein but in the more expressive situation calculus, [19] shows a generalization of Reiter's Basic Action Theories for systems with non-Markovian dynamics. Both of these formalisms provide languages that can refer to past states in the evolution of a dynamic system. However, the focus of these formalisms is on solving the *projection problem*, i.e., reasoning about what will be true in the resulting state after executing a sequence of actions. On the other hand, the focus in the EVOLP$_T$ language is specifying updates to the system's knowledge base itself due to internal or external influence. For example, a system formalized in EVOLP$_T$ would be able to modify the description of its own behavior, which is not possible in $\mathcal{A}$ or in Basic Action Theories.

Also designed for specifying dynamic systems using temporal logic is MetateM [20]. A program in this language consists of rules of the form $P \Rightarrow F$, where P is a Past LTL formula and F is a Future LTL formula. Intuitively, such a rule evaluated in a state specifies that if the evolution of the system up to this state satisfies P, then the system must proceed in such a way that F be satisfied. EVOLP$_T$ does not include Future LTL connectives (our future work) so MetateM is more expressive in that sense. On the other hand, MetateM does not have a construct for updates and it is monotonic, unlike EVOLP$_T$. In [21] the authors propose a non-monotonic extension of LTL with the purpose of specifying agent's goals. Whereas [21] share with our work the use of LTL operators and non-monotonicity, like MetateM it provides future operators, but the non-monotonic character in [21] is given by limited explicit exceptions to rules, thus appearing to be less general than EVOLP$_T$.

References

1. Gabbay, D., Smets, P., eds.: Handbook of Defeasible Reasoning and Uncertainty Management Systems, Volume 3: Belief Change. Kluwer (1998)
2. Alferes, J.J., Leite, J.A., Pereira, L.M., Przymusinska, H., Przymusinski, T.C.: Dynamic updates of non-monotonic knowledge bases. Journal of Logic Programming **45**(1-3) (September/October 2000) 43–70
3. Leite, J.A.: Evolving Knowledge Bases. IOS Press (2003)
4. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: On properties of update sequences based on causal rejection. Theory and Practice of Logic Programming **2**(6) (2002)
5. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: Reasoning about evolving nonmonotonic knowledge bases. ACM Trans. Comput. Log. **6**(2) (2005) 389–440
6. Sefránek, J.: Irrelevant updates and nonmonotonic assumptions. In: Procs. of JELIA'06. Volume 4160 of LNAI., Springer (2006) 426–438
7. Zhang, Y., Foo, N.Y.: Updating logic programs. In: Procs. ECAI. (1998)
8. Sakama, C., Inoue, K.: Updating extended logic programs through abduction. In: Procs. of LPNMR'99. Volume 1730 of LNAI., Springer (1999)
9. Marek, V., Truszczynski, M.: Revision programming. Theor. Comput. Sci. **190**(2) (1998) 241–277
10. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In: Procs. of JELIA'02. Volume 2424 of LNAI., Springer (2002) 50–61
11. Alferes, J.J., Pereira, L.M., Przymusinska, H., Przymusinski, T.C.: LUPS – a language for updating logic programs. Artificial Intelligence **138**(1&2) (June 2002)
12. Alferes, J.J., Banti, F., Brogi, A., Leite, J.A.: The refined extension principle for semantics of dynamic logic programming. Studia Logica **79**(1) (2005) 7–32
13. Alferes, J.J., Gabaldon, A., Leite, J.A.: Evolving logic programming based agents with temporal operators. In: IEEE/WIC/ACM Int'l Conf. on Intelligent Agent Technology. (2008)
14. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: Procs. of ICLP'90. (1990) 579–597
15. Gelder, A.V., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM **38**(3) (1991) 620–650
16. Slota, M., Leite, J.: Evolp: Tranformation-based semantics. In Sadri, F., Satoh, K., eds.: CLIMA VIII. Volume 5056 of LNCS., Springer (2008) 117–136
17. Slota, M., Leite, J.: Evolp: An implementation. In Sadri, F., Satoh, K., eds.: CLIMA VIII. Volume 5056 of LNCS., Springer (2008) 288–298
18. Mendez, G., Lobo, J., Llopis, J., Baral, C.: Temporal logic and reasoning about actions. In: 3rd Symp. Logical Formalizations of Commonsense Reasoning. (1996)
19. Gabaldon, A.: Non-markovian control in the situation calculus. In: Procs. AAAI, AAAI Press (2002) 519–524
20. Barringer, H., Fisher, M., Gabbay, D., Gough, G., Owens, R.: Metatem: An introduction. Formal Aspects of Computing **7**(5) (1995) 533–549
21. Baral, C., Zhao, J.: Non-monotonic temporal logics for goal specification. In: Procs. IJCAI. (2007) 236–242