Evolving Logic Programs with Temporal Operators

José Júlio Alferes, Alfredo Gabaldon, and João Leite

CENTRIA – Center for Artificial Intelligence,
Universidade Nova de Lisboa, Portugal

Abstract. Logic Programming Update Languages have been proposed as extensions of logic programming that allow specifying and reasoning about knowledge bases where both extensional knowledge (facts) as well as intentional knowledge (rules) may change over time as a result of updates.

Despite their generality, these languages are limited in that they do not provide a means to directly access past states of the evolving knowledge. They only allow for so-called Markovian change, i.e. change which is entirely determined by the current state of the knowledge base.

After motivating the need for non-Markovian change, we extend the language EVOLP – a Logic Programming Update Language – with Linear Temporal Logic-like operators, which allow referring to the history of an evolving knowledge base. We then show that it is in fact possible to embed the extended EVOLP into the original one, thus demonstrating that EVOLP itself is expressive enough to encode non-Markovian dynamic knowledge bases. This embedding additionally sheds light on the relationship between Logic Programming Update Languages and Modal Temporal Logics. The embedding is also the starting point of our implementation.

1 Introduction

While belief update in the context of classical knowledge bases has traditionally received significant devotion [15], only in the last decade have we witnessed increasing attention to this topic in the context of non-monotonic knowledge bases, notably using logic programming (LP) [25, 5, 24, 10, 11, 31, 35, 30, 27, 3, 6, 2, 34]. Chief among the results of such efforts are several semantics for sequences of logic programs (dynamic logic programs) with different properties [25, 5, 24, 10, 31, 35, 30, 2, 34], and the so-called LP Update Languages: LUPS [6], EPI [9, 11], KABUL [24] and EVOLP [3].

LP Update Languages are extensions of LP designed for modeling dynamic, non-monotonic knowledge bases represented by logic programs. In these knowledge bases, both the extensional part (a set of facts) and the intentional part (a set of rules) may change over time due to updates. In these languages, special types of rules are used to specify updates to the current knowledge base leading to a subsequent knowledge base. LUPS, EPI and KABUL offer a very diverse set

M. Balduccini and T.C. Son (Eds.): Gelfond Festschrift, LNAI 6565, pp. 193–212, 2011.

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.

In a nutshell, *EVOLP* is a simple though quite powerful extension of ASP [18, 17] that allows for the specification of a program's evolution, in a single unified way, by permitting rules to indicate assertive conclusions in the form of program rules. Syntactically, evolving logic programs are just generalized logic programs[1]. But semantically, they permit to reason about updates of the program itself. The language of *EVOLP* contains a special predicate $assert/1$ whose sole argument is a full-blown rule. Whenever an assertion $assert\,(r)$ is true in a model, the program is updated with rule r. The process is then further iterated with the new program. These assertions arise both from self (i.e. internal to the program) updating, and from external updating (e.g. originating in the environment). EVOLP can adequately express the semantics resulting from successive updates to logic programs, considered as incremental knowledge specifications, and whose effect can be contextual. Whenever the program semantics allows for several possible program models, evolution branching occurs, and several evolution sequences are made possible. This branching can be used to specify the evolution of a situation in the presence of incomplete information. Moreover, the ability of *EVOLP* to nest rule assertions within assertions allows rule updates to be themselves updated down the line. Furthermore, the *EVOLP* language can express self-modifications triggered by the evolution context itself, present or future—*assert* literals included in rule bodies allow for looking ahead on some program changes and acting on that knowledge before the changes occur. In contradistinction to other approaches, *EVOLP* also automatically and appropriately deals with the possible contradictions arising from successive specification changes and refinements (via Dynamic Logic Programming[2] [5, 24, 2]).

Update languages have been applied in areas such as Multi-Agent Systems to represent the dynamics of both beliefs and capabilities of agents [23, 21], Legal Reasoning to represent the way rules and regulations change over time and how they update existing ones [29], Role Playing Games to represent the dynamic behavior of Non-Playing Characters [22] and Knowledge Bases to describe their update policies [9]. Furthermore, it was shown that Logic Programming Update Languages are able to capture Action Languages $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ [19], making them suitable for describing domains of actions [1].

[1] Logic programs that allow for rules with default negated literals in their heads.

[2] *Dynamic Logic Programming* determines the semantics of sequences of generalized logic programs representing states of the world at different time periods, i.e. knowledge undergoing successive updates. As individual theories may comprise mutually contradictory as well as overlapping information, the role of *DLP* is to employ the mutual relationships among different states to determine the declarative semantics, at each state, for the combined theory comprised of all individual theories. Intuitively, one can add newer rules at the end of the sequence, leaving to *DLP* the task of ensuring that these rules are in force, and that previous ones are valid (by inertia) only so far as possible, i.e. they are kept for as long as they are not in conflict with newly added ones, these always prevailing.

Despite their generality, none of the update languages mentioned above provides a means to directly access past states of an evolving knowledge base. These languages were designed for domains where updates are Markovian, meaning that their dynamics are entirely determined by the current state of the knowledge base. Although in many cases this is a reasonable assumption, there are also many scenarios where the behavior of the knowledge base is complex and depends on the particular way it has evolved. In order to capture such complex behavior, it is necessary to be able to refer to properties of the knowledge base at previous states in its "history." This is the motivation behind our proposed language, which we call EVOLP$_T$. The following example attempts to illustrate this.

Consider a knowledge base 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 receives a warning by sms, and if there is another failed login attempt the account is blocked. Moreover, if there are two simultaneous failed login attempts from different ip addresses of the same domain, then the domain is considered suspect. This policy could be expressed by the following rules:

$$\begin{aligned} sms(User) &\leftarrow \Box(not\ sms(User)), fLogin(User, IP). \\ assert(block(User)) &\leftarrow \Diamond(sms(User)), fLogin(User, IP). \\ assert(suspect(D)) &\leftarrow fLogin(User1, IP_1), fLogin(User2, IP_2), \\ &\quad domain(D, IP_1), domain(D, IP_2), IP_1 \neq IP_2. \end{aligned}$$

where $fLogin(User, IP)$ represents the external event of a failed login attempt by the user $User$ at the ip IP. The ability to represent such external influence on the contents of a knowledge base is one of the features of EVOLP (and of EVOLP$_T$, which we present below). The symbols $\Diamond$ and $\Box$ represent Past Linear Temporal Logic (Past LTL) like operators. Intuitively, $\Diamond\varphi$ means that there is a past state where φ was true and $\Box\varphi$ means that φ was true in all past states. The *assert* construct is one of the main features of EVOLP. It allows the user to specify updates to a knowledge base by declaring new rules to be asserted without worrying about conflicts with rules already in the knowledge base.

Referring back to the example, the first rule above specifies that, at any given state, if there is a failed login attempt, $fLogin(User, IP)$, and the user has not received an SMS warning in the past, $\Box(not\ sms(User))$, then the user is sent a warning, $sms(User)$. The second rule specifies that if there is a failed login attempt, $fLogin(User, IP)$, and the user has received an SMS warning sometime in the past, $\Diamond(sms(User))$, then the user is blocked, $assert(block(User))$. Notice that in this case the fact $block(User)$ is asserted while in the case of the SMS warning the atom $sms(User)$ holds in the current state but is not asserted. Intuitively, the atom $sms(User)$ only holds in the state where the rule fires. On the other hand, the fact $block(User)$, which is asserted, persists by inertia until possibly a subsequent update falsifies it.

Now suppose we want to model some updates made later on by the system administrator. Consider for instance that the *sys admin* decides that for those

domains which have remained suspect since the last failed login attempt, from now on, i) immediately block any user failing a login from such a domain and ii) not send sms warnings to users failing logins from such domains.

This may be captured by the rules:

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

The symbol **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 ability to refer to the past, such as referring to an sms never having been sent in the past or that a domain has remained suspect since a failed login occurred, is lacking in EVOLP and other similar update languages. For instance, in [9], Eiter et al. discuss that a possible useful extension of their EPI language would be to add a set of constructs for expressing complex behavior that takes into account the full temporal evolution of a knowledge base, and mention as an example a construct $prev()$ for referring to the previous stage of the knowledge base.

In this paper, we introduce EVOLP$_T$, an extension of EVOLP with LTL-like operators which allow more flexibility in referring to the history of the evolving knowledge base.[3] We proceed by showing that, with a suitable introduction of new propositional variables, it is possible to embed EVOLP$_T$ into the original EVOLP, thus demonstrating that EVOLP is expressive enough to capture non-Markovian knowledge base dynamics. In addition to this expressivity result, the embedding proves interesting also for shedding light into the relationship between Logic Programming Update Languages and modal temporal logics. The embedding is also a starting point for an implementation, which we also describe in this paper, and that is freely available from `http://centria.di.fct.unl.pt/~jja/updates/`.

2 Preliminaries

EVOLP [3] is a logic programming language that includes the special predicate $assert/1$ for specifying updates. An EVOLP program consists of a set of rules of the form

$$L_0 \leftarrow L_1, \ldots, L_n$$

where $L_0, L_1 \ldots, L_n$ are literals, i.e., propositional atoms possibly preceded by the negation-as-failure operator *not* . Notice that EVOLP allows *not* to appear in

[3] EVOLP$_T$ was first introduced in [4], a preliminary shorter version of this paper.

the head of rules. The predicate $assert/1$ takes a rule as an argument. Intuitively, $assert(R)$ means that the current knowledge base will be updated by adding rule R.

An EVOLP program containing rules with *assert* in the head is capable of going through a sequence of changes even without influence from outside. External influence is captured in EVOLP by means of a sequence of programs each of which represents an update due to external events. In other words, using predicate *assert* one can specify self-modifying behavior in an EVOLP knowledge base, while updates issued externally are specified as a sequence of programs in the same language. The following definitions make these intuitions precise.

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 $L_0, \ldots, L_n$ 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 logic program rules over $\mathcal{L}_{assert}$.*

By nesting $assert/1$ one can specify knowledge base updates that may in turn introduce further updates. As mentioned earlier, in addition to these "internal updates," one can specify "external events" intuitively of two types: observation (of facts or rules) events that occur at a given state, and direct assertion commands which specify an update to the knowledge base with new rules. The main difference between the two types of event is that observations only hold in the state where they occur, while rules that are asserted persist in the knowledge base. Syntactically, observations are represented in EVOLP by rules without the assert predicate in the head, and assertions by rules with it. A sequence of external events is represented as a sequence of EVOLP programs:

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 $\mathcal{L}$.*

Given an initial EVOLP program and an event sequence, the semantics dictates what holds and what does not after each of the events in the sequence. 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 $\langle I_1, \ldots, I_n \rangle$. Each evolution stable model describes some possible evolution of an initial program after a number n of evolution steps, given the events in the sequence. Each evolution is represented by a sequence of programs $\langle P_1, \ldots, P_n \rangle$, where each P_i corresponds to a knowledge base constructed 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. Assert literals in the bodies of rules are treated as are any other literals.

Definition 3. *An* evolution interpretation *of length n 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 *of P under $\mathcal{I}$ is the sequence of programs $\langle P_1, P_2, \ldots, P_n \rangle$ where*

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

Sequences of programs are then treated as in *dynamic logic programming* [5], 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. The semantics of dynamic logic programs [2] is a generalization of the answer-set semantics of [26] in the sense that if the sequence consists of a single program, the semantics coincide.

Before we define this semantics, we need to introduce some notation. Let $\rho_s(P_1, ..., P_n)$ denote the multiset of all rules appearing in the programs $P_1, ..., P_s$. If r is a rule of the form $L_0 \leftarrow L_1, \ldots, L_n$, 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). Two rules r, r' are said to be *conflicting rules*, denoted by $r \bowtie r'$, iff $H(r) = A$ and $H(r') = not\, A$ or $H(r) = not\, A$ and $H(r') = A$. By $least(P)$ we denote the least model of the definite program obtained from the argument program P by replacing every default literal $not\, A$ by a new atom not_A. For every set of propositional atoms M in $\mathcal{L}_{assert}$, we define $M' = M \cup \{not\_A \mid A \notin M\}$. Finally, we need to define the following two sets. Given a program sequence $\mathcal{P}$ and a set of atoms M, the first set denotes the set of negated atoms $not\, A$ that hold wrt a state s:

$$Def_s(\mathcal{P}, M) \stackrel{\text{def}}{=} \{not\, A \mid \nexists r \in \rho_s(\mathcal{P}), H(r) = A, M \vDash B(r)\}\,.$$

The second is the set of rules that are "rejected" because of the appearance of a conflicting rule later in the sequence and whose body is satisfied by M:

$$Rej_s(\mathcal{P}, M) \stackrel{\text{def}}{=} \{r \mid r \in P_i, \exists r' \in P_j, i \leq j \leq s, r \bowtie r', M \vDash B(r')\}\,.$$

Definition 4. *Let $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ be a sequence of programs over $\mathcal{L}$. A set, M, of propositional atoms in $\mathcal{L}_{assert}$ is a dynamic stable model of $\mathcal{P}$ at state s, $1 \leq s \leq n$, iff*

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

Whenever $s = n$, we simply say that M is a stable model of $\mathcal{P}$.

Going back to EVOLP, the events received at each state must be added to the corresponding program of the trace before testing the stability condition of stable models of the evolution interpretation.

Definition 5 (Evolution Stable Model). *Let $\mathcal{I} = \langle I_1, ..., I_n \rangle$ be an evolution interpretation of an EVOLP program P and $\langle P_1, P_2, \ldots, P_n \rangle$ be the corresponding execution 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 i $(1 \leq i \leq n)$, I_i is a stable model of $\langle P_1, P_2, \ldots, (P_i \cup E_i) \rangle$.*

3 EVOLP with Temporal Operators

EVOLP programs have the limitation that rules cannot refer to past states in the evolution of a program. In other words, they do not allow one to specify behavior that is conditional on the full evolution of the system being modeled. Despite the fact that the whole evolution is available as a sequence of evolving programs, the body of a rule at any state is always evaluated only with respect to that state. In fact, a careful look at the above definition of the semantics of dynamic logic programs makes this evident: in the definitions of both $Def_s(\mathcal{P}, M)$ and $Rej_s(\mathcal{P}, M)$, rules in previous states are indeed taken into account, but the rule bodies are always evaluated with respect to model M which is the model defined for the knowledge base at state s.

Our goal here is to extend the syntax and semantics of EVOLP to overcome this limitation, defining a new language called EVOLP$_T$. Our approach is similar to the approach in [12, 14] where action theories in the Situation Calculus are generalized with non-Markovian control. In particular, we extend the syntax of EVOLP with Past LTL modalities $\bigcirc(G)$, $\Diamond(G)$ $\Box(G)$, and $\mathbf{S}(G_1, G_2)$, which intuitively mean, respectively: G holds in the previous state; G holds in some past state; G holds in all past states; and G_1 holds since G_2 holds.

Moreover, we allow arbitrary nesting of these operators as well as negation-as-failure in front of their arguments. Unlike *not* , however, temporal operators are not allowed 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 6 (EVOLP with Temporal Operators). *Let $\mathcal{L}$ be any propositional language (not containing the predicates assert/1, $\bigcirc$/1, $\Diamond$/1, $\mathbf{S}$/2 and $\Box$/1). The* extended temporal language $\mathcal{L}_{assertT}$ *and the* set of b-literals[4] $\mathcal{G}$ *are defined inductively as follows:*

- *All propositional atoms in $\mathcal{L}$ are propositional atoms in $\mathcal{L}_{assertT}$ and b-literals in $\mathcal{G}$.*
- *If G_1 and G_2 are b-literals in $\mathcal{G}$ then $\bigcirc(G_1)$, $\Diamond(G_1)$, $\mathbf{S}(G_1, G_2)$ and $\Box(G_1)$ are t-formulae[5], and are also b-literals in $\mathcal{G}$.*
- *If G is a t-formula or an atom in $\mathcal{L}_{assertT}$ then not G is a b-literal in $\mathcal{G}$.*
- *If G_1 and G_2 are b-literals in $\mathcal{G}$, then (G_1, G_2) is a b-literal in $\mathcal{G}$.*

[4] Intuitively, b-literal stands for body-literal.
[5] Intuitively, t-formula stands for temporal-formula.

- *If L_0 is a propositional atom A in $\mathcal{L}_{assertT}$ or its default negation not A, and each of $G_1, \ldots G_n$ is a b-literal, then $L_0 \leftarrow G_1, \ldots, G_n$ is a generalized logic program rule over $\mathcal{L}_{assertT}$ and $\mathcal{G}$.*
- *If R is a rule over $\mathcal{L}_{assertT}$ then $assert(R)$ is a propositional atom of $\mathcal{L}_{assertT}$.*
- *Nothing else is a propositional atom in $\mathcal{L}_{assert}$ or a b-literal in $\mathcal{G}$.*

An evolving logic program with temporal operators *over a language $\mathcal{L}$ is a (possibly infinite) set of generalized logic program rules over $\mathcal{L}_{assertT}$ and $\mathcal{G}$.*

For example, under this definition 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, which is all allowed.

On the other hand, the following are examples of rules that are not legal according to the definition:

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

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 atoms and t-formulae.

As in EVOLP, the definition of the semantics is based on sequences of interpretations $\langle I_1, \ldots, I_n \rangle$ (evolution interpretations). Each interpretation in a sequence stands for the propositional atoms (of $\mathcal{L}_{assertT}$) that are true at the corresponding state, and a sequence stands for a possible evolution of an initial program after a given number n of evolution steps. However, whereas in the original EVOLP language the satisfiability of rule bodies in one such interpretation I_i can easily be defined in terms of set inclusion—all the positive atoms must be included in I_i, all the negative ones excluded—in EVOLP$_T$ satisfiability is more elaborate as it must account for the Past LTL modalities.

Definition 7 (Satisfiability of b-literals). *Let $\mathcal{I} = \langle I_1, ..., I_n \rangle$ be an evolution interpretation of length n of a program P over $\mathcal{L}_{assertT}$, and let G and G' be any b-literals in $\mathcal{G}$. The satisfiability relation is defined as:*

$\mathcal{I} \models A$ *iff* $A \in I_n \wedge A \in \mathcal{L}_{assertT}$
$\mathcal{I} \models not\, G$ *iff* $\langle I_1, ..., I_n \rangle \not\models G$
$\mathcal{I} \models G, G'$ *iff* $\langle I_1, ..., I_n \rangle \models G \wedge \langle I_1, ..., I_n \rangle \models G'$
$\mathcal{I} \models \bigcirc(G)$ *iff* $n \geq 2 \wedge \langle I_1, ..., I_{n-1} \rangle \models G$
$\mathcal{I} \models \Diamond(G)$ *iff* $n \geq 2 \wedge \exists i < n : \langle I_1, ..., I_i \rangle \models G$
$\mathcal{I} \models \mathbf{S}(G, G')$ *iff* $n > 2 \wedge \exists i < n : \langle I_1, ..., I_i \rangle \models G' \wedge \forall i < k < n : \langle I_1, ..., I_k \rangle \models G$
$\mathcal{I} \models \Box(G)$ *iff* $\forall i < n : \langle I_1, ..., I_i \rangle \models G$

Given an evolution interpretation, an *evolution trace* (defined below) represents one of the possible evolutions of the knowledge base. In EVOLP$_T$, whether an evolution trace is one of these possible evolutions additionally depends on the satisfaction of the t-formulae that appear in rules. Towards formally defining evolution traces, we first define an elimination procedure which evaluates satisfiability of t-formulae and replaces them with a corresponding truth constant.

Definition 8 (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 generalized logic program rule. The rule resulting from the elimination of temporal operators given $\mathcal{I}$, denoted by $El(\mathcal{I}, L_0 \leftarrow G_1, \ldots, G_n)$, 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 $\mathcal{I}$, denoted by $El(\mathcal{I}, P)$, is obtained by applying $El(\mathcal{I}, r)$ to each rule r of P.

Evolution traces are defined as in Def. 3 except that t-formulae are eliminated by applying *El*:

Definition 9 (Evolution Trace of EVOLP$_T$ Programs). *Let $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ be an evolution interpretation of an EVOLP$_T$ program P. The* evolution trace *of P under $\mathcal{I}$ is the sequence of programs $\langle P_1, P_2, \ldots, P_n \rangle$ where:*

- *$P_1 = El(\langle I_1 \rangle, P)$ and*
- *$P_i = El(\langle I_1, \ldots, I_i \rangle, \{R \mid assert(R) \in I_{i-1}\})$ for $2 \leq i \leq n$.*

Since the programs in an evolution trace do not mention t-formulae, evolution stable models can be defined in a similar way as in Def. 5, only taking into account that the temporal operators must also be tested for satisfiability, and eliminated accordingly, from the evolution trace and also from the external events. Here, events are simply sequences of EVOLP$_T$ programs.

Definition 10 (Evolution Stable Models of EVOLP$_T$ Programs). *Let $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ be an evolution interpretation of an EVOLP$_T$ program P and $\langle P_1, P_2, \ldots, P_n \rangle$ be the corresponding execution trace. Then $\mathcal{I}$ is an* evolution stable model *of P given event sequence $\langle E_1, E_2, \ldots, E_n \rangle$ iff I_i is a stable model of*

$$\langle P_1, P_2, \ldots, (P_i \cup E_i^*) \rangle$$

for every i ($1 \leq i \leq n$), where $E_i^* = \{El(\langle I_1, \ldots, I_i \rangle, r) \mid r \in E_i\}$.

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 11 (Stable Models after n Steps of EVOLP$_T$ Programs). *Let P be an EVOLP$_T$ program over the language $\mathcal{L}$. We say that a set of propositional atoms M over $\mathcal{L}_{assertT}$ 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 propositional atom A of $\mathcal{L}_{assertT}$ 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 express in terms of other operators:

Proposition 1. *Let $\mathcal{I} = \langle I_1, \ldots, I_n\rangle$ be an evolution stable model of an EVOLP$_T$ program given a sequence of events $\mathcal{E}$. Then, for every $G \in \mathcal{G}$:*

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

Moreover, it should also be noted that EVOLP$_T$ is an extension 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 the original EVOLP. As an immediate consequence of this fact, it can also be noted that EVOLP$_T$ coincides with answer-sets when, moreover, the sequence of events is empty and predicate $assert/1$ does not occur in the program.

We end this section by illustrating the results of EVOLP$_T$ semantics on the motivating example of the Introduction.

Example 1. The initial program P is made of the rules specifying the initial policy plus, for the example, some facts about ips and domains:

$$sms(User) \leftarrow \Box(not\, sms(User)), fLogin(User, IP).$$
$$assert(block(User)) \leftarrow \Diamond(sms(User)), fLogin(User, IP).$$
$$assert(suspect(D)) \leftarrow fLogin(User1, IP_1), fLogin(User2, IP_2),$$
$$domain(D, IP_1), domain(D, IP_2), IP_1 \neq IP_2.$$
$$domain(ip_1, d_1). \quad domain(ip_2, d_2). \quad domain(ip_3, d_2). \quad domain(ip_4, d_2).$$

At this initial state, assuming an empty E_1, the only evolution stable model is $\langle I_1\rangle$ with I_1 only including the $domain/2$ facts.

Now suppose that there is an event of a failed login from John at ip_1, represented by the program E_2 with the single fact $fLogin(john, ip_1)$.

The only evolution stable model of P given $\langle E_1, E_2\rangle$ is the interpretation $\langle I_1, I_2\rangle$ where I_1 includes the $domain/2$ facts and I_2 includes, besides the $domain/2$ facts, $sms(john)$, which intuitively means that upon receiving the

event of the failed login from John, an sms is sent to him. Indeed, the evolution trace is $\langle P_1, P_2 \rangle$ where P_2 is empty (since I_1 has no $domain/1$ predicates) and $P_1 = El(\langle I_1, I_2 \rangle, P)$ is:

$$
\begin{aligned}
sms(User) &\leftarrow \mathbf{true}, fLogin(User, IP).\\
assert(block(User)) &\leftarrow \mathbf{false}, fLogin(User, IP).\\
assert(suspect(D)) &\leftarrow fLogin(User1, IP_1), fLogin(User2, IP_2),\\
&\qquad domain(D, IP_1), domain(D, IP_2), IP_1 \neq IP_2.\\
domain(ip_1, d_1). &\quad domain(ip_2, d_1). \quad domain(ip_3, d_2). \quad domain(ip_4, d_2).
\end{aligned}
$$

It is easy to verify that I_1 is a stable model of $\langle P_1 \cup E_1^* \rangle\rangle$ and I_2 a stable model of $\langle P_1, (P_2 \cup E_2^*) \rangle\rangle$.

Continuing the example, suppose that there is an event of another failed login from John, this time at ip_3, and of a failed login from Eva at ip_4 (represented by the program E_3 with the facts $fLogin(john, ip_3)$ and $fLogin(eva, ip_4)$). Now the evolution stable model is $\langle I_1, I_2, I_3 \rangle$ were I_1 and I_2 are as above, I_3 includes $sms(eva)$, and also $assert(block(john))$ and $assert(suspect(d_2))$. As such, $block(john)$ and $suspect(d_2)$ will be added to the evolution trace in the subsequent step.

Suppose now that the system administrator issues the update with the rules shown in the Introduction, including them in E_4, and subsequently, in E_5, there are failed logins from Carl at ip_4 and from Vic at ip_1. In this case, after these 5 steps, $assert(block(carl))$ becomes true (blocking Carl's access), and no sms is sent to him because the more recent rule $not\, sms(User) \leftarrow fLogin(User, IP), domain(IP, d_2)$ (belonging to P_5 because of the assertion in E_4) overrides the first rule in the initial program. The same does not happen for Vic, since this rule is not asserted for domain d_1, as it was not suspect.

4 Embedding Temporal Operators in EVOLP

In this section we show that it is possible to transform EVOLP$_T$ programs into regular EVOLP programs. This transformation is important for at least two reasons. On one hand, it shows that EVOLP is expressive enough to deal with non-Markovian conditions, although not immediately nor easily. On the other hand, given the existing implementations of EVOLP, the transformation readily provides a means to implement EVOLP$_T$, as we describe below.

Transforming EVOLP$_T$ programs and sequences of events into EVOLP mainly amounts to eliminating the t-formulae by introducing new propositional atoms that will represent the t-formulae, and rules that will model how the truth value of the t-formulae changes as the knowledge base evolves. We start by defining the target language of the resulting EVOLP programs.

Definition 12 (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

$$\mathcal{L}_E(\mathcal{L}, P, \mathcal{E}) \stackrel{\text{def}}{=} \mathcal{L} \cup \{'L' \mid L \in \mathcal{G}(P, \mathcal{E})\}$$

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 behavior of the operators.

Definition 13 (Transformation). 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 the language $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, defined as follows:[6]

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:

$$assert('\bigcirc(G)') \leftarrow 'G'.$$
$$assert(not\,'\bigcirc(G)') \leftarrow not\,'G'.$$

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:

$$'\Box(G)' \leftarrow 'G',\ not \bigcirc \mathbf{true}.$$
$$assert(not\,'\Box(G)') \leftarrow not\,'G'.$$

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:

$$'not\,G' \leftarrow not\,'G'$$
$$'G_1, G_2' \leftarrow 'G_1', 'G_2'.$$
$$'G_1, G_2' \leftarrow 'G_2', 'G_1'.$$

[6] When G is an atom, $'G'$ stands for the same atom G.

Before establishing the correctness of this transformation with respect to EVOLP$_T$, it is worth giving some intuition on how the rules added in the transformed program indeed capture the meaning of the temporal operators.

The rule for $\Diamond(G)$ guarantees that whenever $'G'$ is true at some state, the fact $'\Diamond(G)'$ is added to the subsequent program. So, since no rule for $not\,'\Diamond(G)'$ is ever added in the transformation and rules with $\Diamond(G)$ in the head are not allowed in EVOLP$_T$ programs, $'\Diamond(G)'$ will continue to hold in succeeding states. The first rule for $\bigcirc(G)$ is similar to the one for $\Diamond(G)$. But the second one adds the fact $not\ \bigcirc\ (G)$ in case $not\,G$ is true. So $'\bigcirc\ (G)'$ will be true in the state after the one in which $'G'$ is true, and will become false in a state immediately after one in which $'G'$ is false, as desired.

The rules for $\Box(G)$ are also easy to explain, and in fact result from the dualization of the $\Diamond(G)$ operator. More interesting are the rules for $\mathbf{S}(G_1, G_2)$. The first simply captures the condition where $'\mathbf{S}(G_1, G_2)'$ first becomes true: it adds a fact for it, in the state immediately after one in which $'G_1'$ is true and which is preceded by one in which $'G_2'$ is true. With the addition of this fact, according to the semantics of EVOLP, $'\mathbf{S}(G_1, G_2)'$ will remain true by inertia in subsequent states until a rule that asserts $not\,'\mathbf{S}(G_1, G_2)'$ fires. We want this to happen only until a state immediately after one in which $'G_1'$ becomes false. This effect is obtained with the second rule by adding, at the same time as the fact $'\mathbf{S}(G_1, G_2)'$ is asserted, a rule stating that the falsity of $'G_1'$ leads to the assertion of $not\,'\mathbf{S}(G_1, G_2)'$.

These intuitions form the basis of the proof of the following theorem, which we sketch below.

Theorem 1 (Correctness). *Let P be an evolving logic program with temporal operators over language $\mathcal{L}$, and let $Tr(P)$ be the transformed evolving logic program over language $\mathcal{L}_E(\mathcal{L}, P, \langle\rangle)$. Then $M = \langle I_1, \ldots, I_n\rangle$ is an evolving stable model of P iff there exists an evolving stable model $M' = \langle I'_1, \ldots, I'_n\rangle$ of $Tr(P)$ such that*

$$I_1 = (I'_1 \cap \mathcal{L}_{assert}), \ldots, I_n = (I'_n \cap \mathcal{L}_{assert}).$$

Proof. (Sketch) The proof 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. However, since this nesting can be combined with conjunction and negation, as per the definition of the syntax of EVOLP$_T$ (Def. 6), care must be taken with the new propositional variables that stand for those conjunctions and negations. The last rules of the transformation guarantee that a new atom for a conjunction is true in case the b-literals in the conjunction are true, and that a new atom for the negation of a b-literal is true in case the negation of the b-literal is true. □

Since events are also EVOLP$_T$ programs, we can easily deal with events by applying the same transformation. First, when transforming the main program P, we take into account the t-formulae in the event sequence. Then the transformation is applied to the events themselves.

Definition 14 (Transformation of EVOLP$_T$ with Event Sequence). *Let P be an evolving program with temporal operators and $\mathcal{E} = \langle E_1, \ldots, E_k \rangle$ be an event sequence, both over $\mathcal{L}$. Then $Tr(P, \langle E_1, \ldots, E_k \rangle)$ is an EVOLP program (without temporal operators) over language $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$ obtained from P by applying exactly the same procedure as in Def. 13, only replacing* "appearing in P" *by* "either appearing in P or in any of the E_i's".

Theorem 2. *Let P be an evolving logic program with temporal operators over language $\mathcal{L}$, and let $Tr(P)$ be the transformed evolving logic program over language $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$.*

Then $M = \langle I_1, \ldots, I_n \rangle$ is an evolving stable model of P given $\langle E_1, \ldots, E_k \rangle$ iff there exists an evolving stable model $M' = \langle I'_1, \ldots, I'_n \rangle$ of $Tr(P, \langle E_1, E_2, \ldots, E_k \rangle)$ given the sequence $\langle Tr(E_1), \ldots, Tr(E_k) \rangle$ such that

$$I_1 = (I'_1 \cap \mathcal{L}_{assert}), \ldots, I_n = (I'_n \cap \mathcal{L}_{assert}).$$

5 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 [16], i.e., it is sound, though not complete, with respect to the truth relation of Def. 11. The implementations rely on two consecutive program transformations: the first is the transformation from EVOLP$_T$ into EVOLP described above which eliminates temporal operators. The second transformation is based on previous work [33] and takes the result of the first and generates a normal logic program.

5.1 Evolution SM Semantics Implementation

Here we discuss some of the intuitions behind the second transformation and then describe the actual implementation and its usage. We then describe our implementation of query answering under the well-founded semantics.

The implementation proceeds by using the transformation presented in [33] which converts the resulting EVOLP program and sequence of events into a normal logic program over an extended language whose stable models have a one-to-one correspondence with the evolution stable models of the original EVOLP program and sequence of events.

As described in [33], the extended language over which the resulting program is constructed is defined as follows:

$$\begin{aligned}\mathcal{L}_{\text{trans}} \stackrel{\text{def}}{=} & \left\{ A^j, A^j_{\text{neg}} \;\middle|\; A \in \mathcal{L}_{\text{assert}} \wedge 1 \leq j \leq n \right\} \\ & \cup \left\{ \text{rej}(A^j, i), \text{rej}(A^j_{\text{neg}}, i) \;\middle|\; A \in \mathcal{L}_{\text{assert}} \wedge 1 \leq j \leq n \wedge 0 \leq i \leq j \right\} \\ & \cup \left\{ u \right\} \quad .\end{aligned}$$

Atoms of the form A^j and A^j_{neg} in the extended language allow us to compress the whole evolution interpretation (consisting of n interpretations of $\mathcal{L}_{\text{assert}}$, see Def. 3) into just one interpretation of $\mathcal{L}_{\text{trans}}$. The superscript j then encodes the index of these interpretations. Atoms of the form $\text{rej}(A^j, i)$ and $\text{rej}(A^j_{\text{neg}}, i)$ are needed for simulating rule rejection. Roughly, an atom $\text{rej}(A^j, i)$ intuitively means that, at step j, rules with head A that were added at the earlier step i have been rejected and thus cannot be used to derive A. Similarly for $\text{rej}(A^j_{\text{neg}}, i)$. The atom u will serve to formulate constraints needed to eliminate some unwanted models of the resulting program.

To simplify the notation in the transformation's definition, we will use the following conventions: Let L be a literal over $\mathcal{L}_{\text{assert}}$, *Body* a set of literals over $\mathcal{L}_{\text{assert}}$ and j a natural number. Then:

- If L is an atom A, then L^j is A^j and L^j_{neg} is A^j_{neg}.
- If L is a default literal **not** A, then L^j is A^j_{neg} and L^j_{neg} is A^j.
- $Body^j = \{ L^j \mid L \in Body \}$.

Definition 15. *[33] Let P be an evolving logic program and $\mathcal{E} = (E_1, E_2, \ldots, E_n)$ an event sequence. By a* transformational equivalent *of P given $\mathcal{E}$ we mean the normal logic program $P_{\mathcal{E}} = P^1_{\mathcal{E}} \cup P^2_{\mathcal{E}} \cup \ldots \cup P^n_{\mathcal{E}}$ over $\mathcal{L}_{\text{trans}}$, where each $P^j_{\mathcal{E}}$ consists of these six groups of rules:*

1. ***Rewritten program rules.*** *For every rule $(L \leftarrow Body.) \in P$, $P^j_{\mathcal{E}}$ contains the rule*
$$L^j \leftarrow Body^j, \textbf{not}\, \text{rej}(L^j, 1).$$
2. ***Rewritten event rules.*** *For every rule $(L \leftarrow Body.) \in E_j$, $P^j_{\mathcal{E}}$ contains the rule*
$$L^j \leftarrow Body^j, \textbf{not}\, \text{rej}(L^j, j).$$
3. ***Assertable rules.*** *For every rule $r = (L \leftarrow Body.)$ over $\mathcal{L}_{\text{assert}}$ and all i, $1 < i \leq j$, such that $(\text{assert}(r))^{i-1}$ is in the head of some rule of $P^{i-1}_{\mathcal{E}}$, $P^j_{\mathcal{E}}$ contains the rule*
$$L^j \leftarrow Body^j, (\text{assert}(r))^{i-1}, \textbf{not}\, \text{rej}(L^j, i).$$

4. ***Default assumptions.*** *For every atom* $A \in \mathcal{L}_{\text{assert}}$ *such that* A^j *or* A^j_{neg} *appears in some rule of* $P^j_{\mathcal{E}}$ *(from the previous groups of rules),* $P^j_{\mathcal{E}}$ *also contains the rule*

$$A^j_{\text{neg}} \leftarrow \mathbf{not}\, \text{rej}(A^j_{\text{neg}}, 0).$$

5. ***Rejection rules.*** *For every rule of* $P^j_{\mathcal{E}}$ *of the form*

$$L^j \leftarrow \mathit{Body}, \mathbf{not}\, \text{rej}(L^j, i).^7$$

$P^j_{\mathcal{E}}$ also contains the rules

$$\text{rej}(L^j_{\text{neg}}, p) \leftarrow \mathit{Body}. \quad (1)$$

$$\text{rej}(L^j, q) \leftarrow \text{rej}(L^j, i). \quad (2)$$

where:

(a) $p \leq i$ *is the largest index such that* $P^j_{\mathcal{E}}$ *contains a rule with the literal* $\mathbf{not}\, \text{rej}(L^j_{\text{neg}}, p)$ *in its body. If no such* p *exists, then* (1) *is not in* $P^j_{\mathcal{E}}$.

(b) $q < i$ *is the largest index such that* $P^j_{\mathcal{E}}$ *contains a rule with the literal* $\mathbf{not}\, \text{rej}(L^j, q)$ *in its body. If no such* q *exists, then* (2) *is not in* $P^j_{\mathcal{E}}$.

6. ***Totality constraints.*** *For all* $i \in \{1, 2, \ldots, j\}$ *and every atom* $A \in \mathcal{L}_{\text{assert}}$ *such that* $P^j_{\mathcal{E}}$ *contains rules of the form*

$$A^j \leftarrow \mathit{Body}_p, \mathbf{not}\, \text{rej}(A^j, i).$$

$$A^j_{\text{neg}} \leftarrow \mathit{Body}_n, \mathbf{not}\, \text{rej}(A^j_{\text{neg}}, i).$$

$P^j_{\mathcal{E}}$ also contains the constraint

$$u \leftarrow \mathbf{not}\, u, \mathbf{not}\, A^j, \mathbf{not}\, A^j_{\text{neg}}.$$

A thorough explanation of this transformation can be found in [33].

Also in [33] it is proved that there is a one-to-one relation between the stable models of the so transformed normal program and the stable models of the original EVOLP program. From that result and Theorem 2, it follows that the composition of these two transformations is correct, i.e. that the stable models of the resulting normal logic program correspond to the stable models of the original EVOLP$_T$ program plus events.

Our implementation relies on the composed transformation described above. 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 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, where the syntax is just as described in Def. 6, except that the rule

[7] It can be a rewritten program rule, a rewritten event rule or an assertable rule (default assumptions never satisfy the further conditions). The set *Body* contains all literals from the rule's body except the $\mathbf{not}\, \text{rej}(L^j, i)$ literal.

symbol is `<-` and the symbols `previous/1`, `sometime/1`, `since/2`, `always/1` are used instead of $\bigcirc(G)$, $\Diamond(G)$, $\mathbf{S}(G_1, G_2)$ and $\Box(G)$, respectively. The EVOLP$_T$ program is ended by a fact `newEvents`. The rules after this fact constitute the first event, which is again ended by a fact `newEvents`, after which the rules for the second event follow, etc.

For efficiency reasons, the preprocessor applies both transformations simultaneously, rather than applying them in sequence as described in the previous subsection. This is done by a simple combination of the sequences of steps from each of the 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-literals (e.g. it uses a term `sometime(p)` instead of $'\Diamond(p)'$ or $'sometime(p)'$), which eases the processing of nested temporal operators. Similarly, it adds an argument j to atoms, 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 [32]. For this, we provide a version that starts by applying the first transformation, then feeds the result to the (java-based) implementation from [32] which in turn performs the second transformation and computes the stable models (using smodels).

5.2 Query-Answering under WF Semantics Implementation

Instead of computing the stable models of the resulting normal program, one may compute its well-founded model [16]. This provides a (3-valued) model which is sound, though not complete, w.r.t. the intersection of all stable models and thus also with respect to the truth relation of Def. 11. That is, for programs with stable models, if an atom $A(n)$ belongs to the well founded model then A is true in all stable models of the program and events after n steps; if *not* $A(n)$ belongs to the well founded model then A belongs to no stable models after n steps; if neither $A(n)$ nor *not* $A(n)$ belong to the well founded model, then nothing can be concluded.

Despite the incompleteness, the well founded 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 well founded 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).

6 Related Work and Conclusions

We have introduced the language EVOLP$_T$ for representing and reasoning about evolving knowledge bases with non-Markovian dynamics. The language generalizes its predecessor EVOLP by providing rules that may refer to the past states in a knowledge base evolution through Past LTL modalities. In addition to defining a syntax and semantics for the new language, we show, through a syntactic transformation, that an evolving logic program in EVOLP$_T$ can be compiled into a regular program in EVOLP. The latter is thus proved to be expressive enough to capture non-Markovian evolving knowledge bases as defined above.

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. [11] 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 [11] 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, [28] and [20] describe extensions 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, [12, 14] shows a generalization of Reiter's Basic Action Theories for systems with non-Markovian dynamics and [13] introduces a transformation, somewhat similar to ours, of non-Markovian Basic Action Theories into traditional Markovian ones. Both the action language and the situation calculus based 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 the multi-agent language MetateM [8]. 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 [7] the authors propose a non-monotonic extension of LTL with the purpose of specifying agent's goals. Whereas [7] share with our work the the use of LTL operators and non-monotonicity, like MetateM it provides future operators, but the non-monotonic character in [7] is given by limited explicit exceptions to rules, thus appearing to be less general than our proposal.

References

1. Alferes, J.J., Banti, F., Brogi, A.: From logic programs updates to action description updates. In: Leite, J., Torroni, P. (eds.) CLIMA 2004. LNCS (LNAI), vol. 3487, pp. 52–77. Springer, Heidelberg (2005)
2. Alferes, J.J., Banti, F., Brogi, A., Leite, J.A.: The refined extension principle for semantics of dynamic logic programming. Studia Logica 79(1), 7–32 (2005)
3. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 50–61. Springer, Heidelberg (2002)
4. Alferes, J.J., Gabaldon, A., Leite, J.: Evolving logic programming based agents with temporal operators. In: IEEE/WIC/ACM Int'l Conf. on Intelligent Agent Technology (2008)
5. Alferes, J.J., Leite, J.A., Pereira, L.M., Przymusinska, H., Przymusinski, T.C.: Dynamic updates of non-monotonic knowledge bases. The Journal of Logic Programming 45(1-3), 43–70 (2000)
6. 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)
7. Baral, C., Zhao, J.: Non-monotonic temporal logics for goal specification. In: IJCAI 2007, pp. 236–242 (2007)
8. Barringer, H., Fisher, M., Gabbay, D., Gough, G., Owens, R.: Metatem: A framework for programming in temporal logic. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1989. LNCS, vol. 430, pp. 94–129. Springer, Heidelberg (1990)
9. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: A framework for declarative update specifications in logic programs. In: IJCAI 2001, pp. 649–654 (2001)
10. 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)
11. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: Reasoning about evolving nonmonotonic knowledge bases. ACM Trans. Comput. Log. 6(2), 389–440 (2005)
12. Gabaldon, A.: Non-markovian control in the situation calculus. In: AAAI 2002, pp. 519–524. AAAI Press, Menlo Park (2002)
13. Gabaldon, A.: Compiling control knowledge into preconditions for planning in the situation calculus. In: IJCAI 2003, pp. 1061–1066 (2003)
14. Gabaldon, A.: Non-Markovian Control in the Situation Calculus. Artificial Intelligence 175(1), 25–48 (2011)
15. Gabbay, D., Smets, P. (eds.): Handbook of Defeasible Reasoning and Uncertainty Management Systems. Belief Change, vol. 3. Kluwer, Dordrecht (1998)
16. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)

17. Gelfond, M.: Answer sets. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, ch. 7, pp. 285–316. Elsevier, Amsterdam (2008)
18. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: 7th Int'l Conf. on Logic Programming (1990)
19. Gelfond, M., Lifschitz, V.: Action languages. Electronic Transactions on Artificial Intelligence 3, 195–210 (1998)
20. Gonzalez, G., Baral, C., Gelfond, M.: Alan: An action language for non-markovian domains. In: NonMon. Reasoning, Action and Change Workshop (2003)
21. Leite, J.: Playing with rules. In: Baldoni, M., Bentahar, J., van Riemsdijk, M.B., Lloyd, J. (eds.) DALT 2009. LNCS, vol. 5948, pp. 1–19. Springer, Heidelberg (2010)
22. Leite, J., Soares, L.: Evolving characters in role-playing games. In: EMCSR 2006, vol. 2, pp. 515–520 (2006)
23. Leite, J., Soares, L.: Adding evolving abilities to a multi-agent system. In: Inoue, K., Satoh, K., Toni, F. (eds.) CLIMA 2006. LNCS (LNAI), vol. 4371, pp. 246–265. Springer, Heidelberg (2007)
24. Leite, J.A.: Evolving Knowledge Bases. IOS Press, Amsterdam (2003)
25. Leite, J., Moniz Pereira, L.: Generalizing updates: From models to programs. In: Dix, J., Moniz Pereira, L., Przymusinski, T.C. (eds.) LPKR 1997. LNCS (LNAI), vol. 1471, p. 224. Springer, Heidelberg (1998)
26. Lifschitz, V., Woo, T.: Answer sets in general nonmonotonic reasoning (preliminary report). In: KR 1992 (1992)
27. Marek, V., Truszczynski, M.: Revision programming. Theor. Comput. Sci. 190(2), 241–277 (1998)
28. Mendez, G., Lobo, J., Llopis, J., Baral, C.: Temporal logic and reasoning about actions. In: 3rd Symp. on Logical Formalizations of Commonsense Reasoning (1996)
29. Saias, J., Quaresma, P.: A methodology to create legal ontologies in a lp based web information retrieval system. Artif. Intell. Law 12(4), 397–417 (2004)
30. Sakama, C., Inoue, K.: Updating extended logic programs through abduction. In: Gelfond, M., Leone, N., Pfeifer, G. (eds.) LPNMR 1999. LNCS (LNAI), vol. 1730, p. 147. Springer, Heidelberg (1999)
31. Šefránek, J.: Irrelevant updates and nonmonotonic assumptions. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 426–438. Springer, Heidelberg (2006)
32. Slota, M., Leite, J.: Evolp: An implementation. In: Sadri, F., Satoh, K. (eds.) CLIMA VIII 2007. LNCS (LNAI), vol. 5056, pp. 288–298. Springer, Heidelberg (2008)
33. Slota, M., Leite, J.: Evolp: Tranformation-based semantics. In: Sadri, F., Satoh, K. (eds.) CLIMA VIII 2007. LNCS (LNAI), vol. 5056, pp. 117–136. Springer, Heidelberg (2008)
34. Slota, M., Leite, J.: On semantic update operators for answer-set programs. In: ECAI 2010, pp. 957–962. IOS Press, Amsterdam (2010)
35. Zhang, Y., Foo, N.Y.: Updating logic programs. In: ECAI 1998. John Wiley & Sons, Chichester (1998)