Evolving Logic Programming based Agents with Temporal Operators

José Júlio Alferes
Universidade Nova de Lisboa
Portugal

Alfredo Gabaldon
Universidade Nova de Lisboa
Portugal

João Leite
Universidade Nova de Lisboa
Portugal

Abstract

Logic Programming Update Languages were proposed as an extension of logic programming, which allow for modelling the dynamics of knowledge bases where both extensional knowledge (facts) as well as intentional knowledge (rules) may change over time due to updates, with important application Multi-Agent Systems (MAS).

Despite their generality, these languages do not provide means to directly access past states of the evolving knowledge. They only allow for so-called Markovian changes i.e. changes determined entirely by the current state. This is a drawback in several situation.

In this paper, after motivating the need for non-Markovian changes, we extend EVOLP – The Logic Programming Update Language at the heart of an existing MAS – with LTL-like temporal operators that allow referring to the history of the evolving agent. We then show that with a suitable introduction of new propositional variables it is possible to embed the extended EVOLP into the original one, thus demonstrating that EVOLP itself can already be used for non-Markovian changes. While showing how to use EVOLP for encoding non-Markovian changes, this embedding sheds light into the relationship between Logic Programming Update Languages and Modal Temporal Logics, of particular importance in MAS.

1 Introduction

With the promise to uniformly integrate the tasks of specifying, programming and verifying Multi-Agent Systems (MAS), Computational Logic (CL) and Logic Programming (LP) have been used as the privileged driving vehicles to describe the informational, motivational and dynamic dimensions of several such systems [25, 11, 18, 9, 7, 17, 21]. For surveys on some of these and others see [23, 8].

Agents must keep beliefs about their goals, intentions, capabilities and the environment in which they are situated. These beliefs must be dynamic, not only because the agent may learn about static features of its environment and new ways to behave, but also because of the intrinsic dynamic character of the environment. This rich dynamic character of MAS called for the development of LP based languages that are capable of dealing with updates that go beyond the simple addition and deletion of fluents. These languages are usually referred to as LP Update Languages and include LUPS [5], EPI [12], KABUL [20] and EVOLP [3].

LP Update Languages are extensions of LP designed to allow the modelling of the dynamics of non-monotonic knowledge represented by logic programs where both their extensional part (set of facts) as well as their intentional part (set of deductive rules) may change over time due to updates. Each defines special types of rules that specify the transition to subsequent states of knowledge through the update of the current one. While LUPS, EPI and KABUL offer a very diverse set of update commands, each specific for one particular kind of update (assertion, retraction, etc), EVOLP – the language we focus on – follows a simpler approach that stays closer to traditional logic programming.

EVOLP (Evolving Logic Programming) generalizes Answer-set Programming [16] to allow for the specification of a program's own evolution, arising both from self (i.e. internal to the program) updating, and external updating originating in the environment. From the syntactical point of view, evolving programs are generalized logic programs, extended with (possibly nested) assertions in either heads or bodies of rules. From the semantical point of view, a model-theoretic characterization is offered of the possible evolutions of programs by means of *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.

EVOLP is at the heart of the MAS presented in [21], where it is used to represent the dynamics of both beliefs and capabilities of agents. The use of EVOLP has also been illustrated in Role Playing Games to represent the dynamic behaviour of Non-Playing Characters [19] and Knowledge Bases to describe their update policies [12]. Furthermore, it was shown that Logic Programming Update Languages are able to capture Action Languages $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, making them suitable for describing domains of actions [1].

Despite their generality concerning the kinds of updates possible, these LP Update Languages do not provide means to directly access past states of the evolving knowledge base i.e. states other than the current one. They were designed for situations where all knowledge updates are Markovian i.e. determined by the current state of affairs.

However, there are many scenarios that require non-Markovian updates i.e. updates that depend on conditions encompassing past states of knowledge.

Suppose that we want to build agents that control user access for a number of computers at different locations. A login policy for example may say that after the first failed login the 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:

$$sms(U) \leftarrow \Box(not\, sms(U)), fLog(U, IP).$$
$$block(U) \leftarrow \Diamond(sms(U)), fLog(U, IP).$$

where we assume that $fLog(U, IP)$ is an external event representing a failed login by user U from address IP. A feature of EVOLP is the ability to represent such external influence on the contents of a knowledge base and its updates. The symbols $\Diamond$ and $\Box$ represent operators similar to the Past LTL operators where $\Diamond\varphi$ means that there is a state in the past where φ was true and $\Box\varphi$ means that φ was true in all past states. Now suppose that we want to model the updates made by the system administrator. For example, the new policy consists in blocking a user after the first failed login attempt if the user has an IP address from a "bad domain", and not send him an sms. The new policy is represented by the following rules:

$$block(U) \leftarrow fLog(U, IP), dom(IP, D).$$
$$not\, sms(U) \leftarrow fLog(U, IP), dom(IP, D).$$

with D instantiated to the domain in question.

Whether a domain is bad or not, however, depends on the particular agent. In this case, the sys admin may want to send an update to all agents so that the above rules are added to each agent's policy only for domains which are bad according to the agent's current history. The sys admin issues the following update to every agent, saying that the above new rules are to be asserted if the domain has been considered a bad one since the last failed attempt:

$$assert(block(U) \leftarrow fLog(U, IP), dom(IP, D)) \leftarrow$$
$$\quad \mathbf{S}(badDom(D),\ fLog(U2, IP2)), dom(IP2, D).$$
$$assert(not\, sms(U) \leftarrow fLog(U, IP), dom(IP, D)) \leftarrow$$
$$\quad \mathbf{S}(badDom(D),\ fLog(U2, IP2)), dom(IP2, D).$$

where $badDom(D)$ is a predicate defined locally and the symbol $\mathbf{S}$ above represents an operator similar to the Past LTL operator "since". The intuitive meaning of $\mathbf{S}(\psi, \varphi)$ is that 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 which allows one to specify updates to the agent, 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.

The ability to refer to the past is lacking in EVOLP. [12] suggests that LP Languages of Updates need a $prev()$ predicate to access the previous state. In this paper, we go beyond that and introduce LTL-like temporal operators that allow more flexibility in referring to the history of the evolving knowledge base. We proceed by showing that with the introduction of new propositional variables, it is possible to embed the extended EVOLP into the original one, demonstrating that EVOLP itself can already be used for non-Markovian changes. This embedding proves interesting as it shows how to use EVOLP for encoding non-Markovian changes, while shedding light into the relationship between LP Update Languages and modal temporal logics.

2 Preliminaries

EVOLP [3] is a logic programming language extended with the special predicate $assert/1$ used for specifying updates, and with the possibility of having negated rule heads. 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 preceeded by the negation-as-failure operator not) including literals of the $assert/1$ predicate. An atom $assert(R)$ takes a rule R as an argument and intuitively represents that the argument R belongs to the next program in the evolution.

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 can also be captured in EVOLP. This is done by means of a sequence of programs each of which represents external events. The next 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 $\mathcal{L}_{assert}$.*

Nesting of $assert/1$ permits updating the knowledge base with rules that may, in turn, further update it. This language alone is enough to model a knowledge base allowing

for internal updating actions changing it. But EVOLP goes beyond such self-evolution in that it also allows change to be caused by external events, where these may be: observation of facts (or rules) that are perceived at some state; assertion commands directly imparting the assertion of new rules on the evolving program. Both can be represented as EVOLP rules: the former by rules without the assert predicate in the head, and the latter by rules with it. To represent outside influence as a sequence of EVOLP rules:

Definition 2 *Let P be an evolving program over the language L. An* event sequence *over P is a sequence of evolving programs over L.*

Given this syntax, the semantics issue is that of, given an initial EVOLP program and a sequence of EVOLP programs as events, determining what is true and what is false after each of these events. Precisely, the meaning of a sequence of EVOLP programs is given by a set of *evolution stable models*, each of which is a sequence of interpretations or states $\langle I_1, \ldots, I_n \rangle$. Each evolution stable model describes some possible evolution of one initial program after a number n of evolution steps, given the events in the sequence. Each evolution is represented by a sequence of programs $\langle P_1, \ldots, P_n \rangle$, each program corresponding to a knowledge state 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; asserts in bodies are treated as any other predicate literals.

Definition 3 *An* evolution interpretation *of length n of an evolving program P over L is a finite sequence $\mathcal{I} = \langle I_1, I_2, \ldots, I_n \rangle$ of sets of propositional atoms of $\mathcal{L}_{assert}$. The* evolution trace *associated with an evolution interpretation $\mathcal{I}$ is the sequence of programs $\langle P_1, P_2, \ldots, P_n \rangle$ where $P_1 = P$ and $P_i = \{R \mid assert(R) \in I_{i-1}\}$ for $2 \leq i \leq n$.*

The sequences of programs are then treated as in *dynamic logic programming* [4], 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 is a generalisation of the answer-set semantics of [22] (in the sense that if the sequence consists of a single program, the semantics coincides with answer-sets), and is defined as follows [2]:

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

$$M' = least([\rho_s(\mathcal{P}) - Rej_s(\mathcal{P}, M)] \cup Def_s(\mathcal{P}, M)) \text{ and}$$
$$Def_s(\mathcal{P}, M) = \{not\, A \mid \nexists r \in \rho_s(\mathcal{P}), H(r) = A, M \vDash B(r)\}$$
$$Rej_s(\mathcal{P}, M) = \{r \mid r \in P_i, \exists r' \in P_j, i \leq j \leq s, r \bowtie r', M \vDash B(r')\}$$

where A is an objective literal; $\rho_s(\mathcal{P})$ denotes the multi-set 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); $r \bowtie r'$ (conflicting rules) iff $H(r) = A$ and $H(r') = not\, A$ or $H(r) = not\, A$ and $H(r') = A$; $least(.)$ denotes the least model of the definite program obtained from the argument program by replacing every default literal $not\, A$ by a new atom not_A; and $M' = M \cup \{not\_A \mid A \notin M\}$.

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 modelled. 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 in that state. In fact, a careful analysis of the above definition of the semantics of dynamic logic programs, makes this evident: note that in the definitions of both $Def_s(\mathcal{P}, M)$ and $Rej_s(\mathcal{P}, M)$, rules in previous states are taken into account, but rule bodies are always evaluated with respect to a single model M.

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 [14] where Basic 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 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.

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[1] $\mathcal{G}$ *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[2], 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 $\mathcal{G}$. • 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 generalised 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 generalised logic program rules over $\mathcal{L}_{assertT}$ and $\mathcal{G}$.*

Note that under this definition, e.g. 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.
In contradistinction, e.g. the following rules are not allowed: $assert(\Box(b) \leftarrow a) \leftarrow b.$ $\quad a \leftarrow \Diamond(not\,(a, b)).$ $a \leftarrow not\, not\, b.$

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 or states $\langle I_1, \ldots, I_n \rangle$ (or evolution interpretation). Each interpretation in a sequence stands for the propositional atoms (of $\mathcal{L}_{assertT}$) that are true at the 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 generalised logic program 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.

An evolution trace is then defined as in Def. 3, except t-formulae are eliminated by applying El.

Definition 9 (Evolution Trace) *Let P be an EVOLP$_T$ program and $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ an interpretation. 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, the definition of evolution stable models can be done 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.

[1] Intuitively, b-literal stands for body-literal.
[2] Intuitively, t-formula stands for temporal-formula.

Definition 10 (Evolution Stable Model with Temp. Ops.) *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 given Events) *Let P be an EVOLP$_T$ program over the language 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 SE. 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 rewritten in terms of the others:

Proposition 1 *Let t $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ 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 $\mathcal{I} \models \mathbf{S}(true, G)$*

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.

4 Embed 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$_T$, and this way to easily and directly express non-Markovian conditions over evolving logic programs[3].

Similar transformations have been used for theories of action in the situation calculus [15] and for temporal queries in databases [10]. They replace t-formulae with new propositional atoms and rules that encode the dynamics of the temporal operators. First we introduce the target language.

Definition 12 (Transformation Language) *Let $\mathcal{L}$ be any propositional language, and let $\mathcal{L}_{assertT}$ and $\mathcal{G}$ be, respectively, the extended language and the set of b-literals given $\mathcal{L}$. The transformed propositional language $\mathcal{L}^*$ is $\mathcal{L}$ augmented with a propositional variable $'G'$ for each b-literal G in $\mathcal{G}$. Here by $'G'$ we mean a propositional variable whose name is the (atomic) string of characters that compose the formula G. Furthermore, it is assumed that none of these new propositional variables already occur in $\mathcal{L}$.*

In fact, as it will become clear in the sequel, the transformation language could be made simpler by adding new propositional variables only for those b-literals that appear in either the program or the sequence of events. Indeed, the aforementioned implementation uses the shorter transformation language, only adding the rules of the transformation (below) for those b-literals that appear in the program, and further adding then to those appearing in the events, along with the arrival of events. However, since in this paper we do not focus on the complexity of the transformation, in order to keep the definition as simple as possible we opted for not further restricting the transformation language.

Definition 13 (Transformed EVOLP program) *Let P be an EVOLP program with temporal operators over $\mathcal{L}$. Then $Tr(P)$ is an EVOLP program (without temporal operators) in language $\mathcal{L}^*$ obtained from P by replacing every t-formula G in the body of rules by the new propositional variable $'G'$ and adding the rules:*
- *$assert('\bigcirc(G)') \leftarrow' G'$. and*
  $assert(not'\bigcirc(G)') \leftarrow not\,'G'$.
  for every b-literal $\bigcirc(G)$ appearing in P;
- *$assert('\Diamond(G)') \leftarrow' G'$.*
  for every b-literal $\Diamond(G)$ appearing in P;
- *$assert('\mathbf{S}(G_1, G_2)') \leftarrow' G_1', \bigcirc(G_2)'$. and*
  $assert(assert(not'S(G_1, G_2)') \leftarrow not\,'G_1') \leftarrow assert('S(G_1, G_2)')$.
  for every b-literal $\mathbf{S}(G_1, G_2)$ appearing in P;
- *$'\Box(G)' \leftarrow' G',\ not\ \bigcirc true$. and*
  $assert(not'\Box(G)') \leftarrow not\,'G'$.
  for every b-literal $\Box(G)$ appearing in P;
- *$'not\, G' \leftarrow not\,'G'$.*
  for every b-literal $not\, G$ appearing in P;
- *$'G_1, G_2' \leftarrow' G_1', G_2'$. and*
  $'G_2, G_1' \leftarrow' G_1', G_2'$.

[3] Implementation available at http://centria.fct.unl.pt/~jja/updates/

for every pair of b-literals G_1 and G_2 appearing in P.

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 added in the transformation, and no rule head in the EVOLP$_T$ program contains $\Diamond(G)$, from then onwards $'\Diamond(G)'$ is true. The first rule for $\bigcirc(G)$ is similar to the one for $\Diamond(G)$. But the second 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 dualisation of the $\Diamond(G)$ operator. More interesting are the rules for $\mathbf{S}(G_1, G_2)$. The first simply caters for the condition for which $'\mathbf{S}(G_1, G_2)'$ starts to be 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 some rule for $not\,'\mathbf{S}(G_1, G_2)'$. We want this to happen until a state immediately after one in which $'G_1'$ becomes false. This effect is obtained with the second rule by adding, along with the fact $'\mathbf{S}(G_1, G_2)'$, a rule stating that the falsity of $'G_1'$ leads to the assertion of $not\,'\mathbf{S}(G_1, G_2)'$.

The nesting of temporal operators is dealt with in the transformation by adding 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. For this, the last rules of the transformation are added, guaranteeing 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.

These intuitions form the basis to prove the next theorem. The proof, that cannot be added 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.

Theorem 2 (Embedding of Temporal Operator) *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 $\mathcal{L}^*$. 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 $I_1 = (I'_1 \cap \mathcal{L}_{assert}), \ldots, I_n = (I'_n \cap \mathcal{L}_{assert})$.*

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 (Transformed EVOLP and Event Sequence) *Let P be an evolving program with temporal operators and $\langle E_1, \ldots, E_k\rangle$ 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) in language $\mathcal{L}^*$ obtained from P by applying exactly the same procedure as in Definition 13, only replacing* "appearing in P" *by* "either appearing in P or in any of the E_i's".

Theorem 3 (Embedding of Temp. Op. with Events) *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 $\mathcal{L}^*$. 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 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. [13] 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 [13] 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, [24] 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, [14] 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 [7]. 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 [6] the authors propose a non-monotonic extension of LTL with the purpose of specifying agent's goals. Whereas [6] 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 [6] is given by limited explicit exceptions to rules, thus appearing to be less general than our proposal.

References

[1] J. J. Alferes, F. Banti, and A. Brogi. From logic programs updates to action description updates. In *CLIMA V*, volume 3487 of *LNAI*. Springer, 2004.

[2] J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1):7–32, 2005.

[3] J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In *JELIA'02*, volume 2424 of *LNAI*, pages 50–61. Springer, 2002.

[4] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1-3):43–70, September/October 2000.

[5] J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. LUPS – a language for updating logic programs. *Artificial Intelligence*, 138(1&2), June 2002.

[6] C. Baral and J. Zhao. Non-monotonic temporal logics for goal specification. In *IJCAI'07*, pages 236–242, 2007.

[7] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. Metatem: A framework for programming in temporal logic. In *Procs of REX Workshop 1989*, volume 430 of *LNCS*, pages 94–129, 1990.

[8] R. H. Bordini, L. Braubach, M. Dastani, A. E. F. Seghrouchni, J. J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(1):33–44, 2006.

[9] R. H. Bordini, M. Wooldridge, and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[10] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

[11] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3), 2008.

[12] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specifications in logic programs. In *IJCAI'01*, pages 649–654, 2001.

[13] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Reasoning about evolving nonmonotonic knowledge bases. *ACM Trans. Comput. Log.*, 6(2):389–440, 2005.

[14] A. Gabaldon. Non-markovian control in the situation calculus. In *AAAI'02*, pages 519–524. AAAI Press, 2002.

[15] A. Gabaldon. Compiling control knowledge into preconditions for planning in the situation calculus. In *IJCAI'03*, pages 1061–1066, 2003.

[16] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *ICLP'90*, pages 579–597, 1990.

[17] G. D. Giacomo, Y. Lesprance, and H. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[18] K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Int. J. of Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[19] J. Leite and L. Soares. Evolving characters in role-playing games. In *EMCSR'06*, volume 2, pages 515–520, 2006.

[20] J. A. Leite. *Evolving Knowledge Bases*. IOS Press, 2003.

[21] J. A. Leite and L. Soares. Adding evolving abilities to a multi-agent system. In *CLIMA VII*, volume 4371 of *LNAI*, pages 246–265. Springer, 2007.

[22] V. Lifschitz and T. Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In *KR'92*, 1992.

[23] V. Mascardi, M. Martelli, and L. Sterling. Logic-based specification languages for intelligent software agents. *Theory and Practice of Logic Programming*, 4(4):429–494, 2004.

[24] G. Mendez, J. Lobo, J. Llopis, and C. Baral. Temporal logic and reasoning about actions. In *3rd Symp. on Logical Formalizations of Commonsense Reasoning*, 1996.

[25] V. S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press, 2000.