

From logic programs updates to action description updates ^{*}

J. J. Alferes¹, F. Banti¹, and A. Brogi²

¹ CENTRIA, Universidade Nova de Lisboa, Portugal,
jja|banti@di.fct.unl.pt

² Dipartimento di Informatica, Università di Pisa, Italy,
brogi@di.unipi.it

Abstract. An important branch of investigation in the field agents has been the definition of high level languages for representing effects of actions, the programs written in such languages being usually called action programs. Logic programming is an important area in the field of knowledge representation and some languages for specifying updates of Logic Programs had been defined. In this work we address the problem of establishing relationships between action programs and Logic Programming updates languages, particularly the newly defined Evolp language. Our investigation leads to the definition of a new paradigm for representing actions called Evolp action programs. We provide translations of some of the most known action description languages into Evolp action programs, and underline some peculiar features of this newly defined paradigm. One of such feature is that Evolp action programs can easily express changes in the very rules of the domains, including rules describing changes.

1 Introduction

In the last years the concept of agent had became central in the field of Artificial Intelligence. “*An agent is just something that acts*” [25]. Given the importance of the concept, ways of representing actions and their effects on the environment had been studied. A branch of investigation in this topic has been the definition of high level languages for representing effects of actions [7, 12, 14, 15], the programs written in such languages being usually called *action programs*. Action programs specify which facts (or fluents) change in the environment after the execution of a set of actions. Several works exist on the relation between these action languages and Logic Programming (LP) (e.g. [5, 12, 20]). However, despite the fact that LP has been successfully used as a language for declaratively representing knowledge, the mentioned works basically use LP for providing an operational semantics, and implementation, for action programs. This is so because normal logic programs [11], and most of their extensions, have no in-built

^{*} This work was partially supported by project FLUX (POSI/40958/SRI/2001) and by project SOCS (IST-2001-32530).

means for dealing with changes, something which is quite fundamental for the relation with action languages.

In recent years some effort was devoted to explore the problem of how to update logic programs with new rules [3, 8, 9, 18, 19]. Here, knowledge is conveyed by sequences of programs, where each program in a sequence is an update to the previous ones. For determining the meaning of sequences of logic programs, rules from previous programs are assumed to hold by inertia after the updates (given by subsequent programs) unless rejected by some later rule. LP update languages [2, 4, 10, 18], besides giving meaning to sequences of logic programs, also provide in-built mechanisms for constructing such sequences. In other words, LP update languages extend LP by providing means to specify and reason about rule updates. In [5] the authors show, by examples, a possible use the LP update language LUPS [4] for representing actions. However, the work done does not establish an exact relationship between existing action languages and LP update languages and also the eventual advantages of LP update languages approach to actions are still not clear. The present work tries to clarify these points. Our investigation focuses on the newly defined Evolp language [2].

In section 2 we review some background and notation. In section 3 we show how to use macros defined in Evolp as an action description paradigm. Programs written in such macro language are called Evolp action programs (EAPs). We illustrate the usage of EAPs by an example involving a variant of the classical Yale Shooting Problem. In section 4 we establish the relationship between EAPs and existing approaches by providing simple translations of the action languages \mathcal{A} [12], \mathcal{B} [13] (which is a subset of the language proposed in [14]), and (the definite fragment of) \mathcal{C} [15] into EAPs, thus showing that EAPs are *at least as expressive* as the cited action languages. Coming to this point the next question is what are the possible advantages of EAPs. The underlying idea of action frameworks is to describe dynamic environment. This is usually done by describing rules that specify, given a set of external actions, how the environment evolves. In a dynamic environment, however, not only the facts but also the “rules of the game” can change, in particular *the rules describing the changes*. The capability of describing such kind of *meta level changes* is, in our opinion, an important feature of an action description language. In section 5 we address this topic in the context of EAPs and show EAPs seem, in this sense, more flexible than other paradigms. Evolp provides specific commands that allow for the specification of updates to the initial program but also provides the possibility to specify updates of these updates commands. We show, by successive elaborations of the Yale shooting example defined in section 3.1, how to use this feature to describe successive elaborations of the problem during the evolution of the environment. Finally, in section 6, we conclude and trace a route for future developments.

2 Background and notation

In this section we briefly recall syntax and semantics of *dynamic logic programs* [1] and the syntax and semantics for Evolp[2]. We also recall some basic notions and notation for action description languages.

2.1 Dynamic logic programs and Evolp

The main idea of logic programs updates is to update a logic program by another logic program or by a *sequence* of logic programs, also called *dynamic logic programs* (DLP) the initial program corresponding to the initial knowledge of a given (dynamic) domain, and the subsequent ones to successive updates of the domain. To represent negative information in logic programs and their updates, DLP requires generalized logic programs (GLPs) [21], which allows for default negation *not* A not only in the premises of rules but also in their heads. A language \mathcal{L} is any set of propositional atoms. A literal in \mathcal{L} is either an atom of \mathcal{L} or the negation of such an atom. In general, given any set of atoms \mathcal{F} we denote the by \mathcal{F}_{Lit} the set of literals over \mathcal{F} . Given a literal L , if $L = Q$, where Q is an atom, by *not* L we denote the negative literal *not* Q . Viceversa, if $L = \text{not } Q$, by *not* L we denote the atom Q . A GLP defined over a propositional language \mathcal{L} is a set of rules of the form $L \leftarrow Body$, where L is a literal in \mathcal{L} , and $Body$ is a *set* of literals in \mathcal{L} .³ We say a set of literals $Body$ is true in an interpretation I (or that I satisfies $Body$) iff $Body \subseteq I$. In the paper we will use programs containing variables. As usual when programming within the stable models semantics, a program with variables stands for the propositional program obtained as the set of all possible ground instantiation of the program.

Two rules τ and η are *conflicting* (denoted by $\tau \bowtie \eta$) iff the head of τ is the atom A and the head of η is *not* A or viceversa. A dynamic logic program over a language \mathcal{L} is a sequence $P_1 \oplus \dots \oplus P_m$ (also denoted $\oplus P_i^m$) where the P_i s are GLPs defined over \mathcal{L} . The *refined stable model semantics* of DLP defined in [1] assigns to each sequence $P_1 \oplus \dots \oplus P_n$ a set of stable models (that is proven there to coincide with the stable models based semantics defined in [21] when the sequence is formed by a single GLP). The rationale for the definition of a stable model M of a DLP is made in accordance with the **causal rejection principle** [9, 18]: If the body of a rule in a given update is true in M the considered rule rejects all the conflicting rules in previous updates, which means that such rules are ignored in the computation of the stable model. In the refined semantics for DLPs such rule also rejects any conflicting rule in the same update. Moreover, an atom A is assumed false by default if there is no rule, in none of the programs in the sequence, with head A and a true body in M . Formally:

$$\begin{aligned} Default(\oplus P_i^m, M) &= \{not\ A \mid \nexists A \leftarrow Body \in \bigcup P_i \wedge Body \subseteq M\} \\ Rej^S(\oplus P_i^m, M) &= \{\tau \mid \tau \in P_i : \exists \eta \in P_j \ i \leq j, \tau \bowtie \eta \wedge Body(\eta) \subseteq M\} \end{aligned}$$

where M is an interpretation, i.e. any set of literals in \mathcal{L} such that, for each atom A , either $A \in M$ or *not* $A \in M$. If $\oplus P_i^m$ is clear from the context, we omit it as first argument of the above functions.

³ Note that, by defining rule bodies as sets, the order and number of occurrences of literals does not matter.

Definition 1. Let $\oplus P_i^m$ be a DLP over language \mathcal{L} and M a interpretation. M is a refined stable model of $\oplus P_i^m$ iff

$$M = \text{least} \left(\bigcup P_i \setminus \text{Rej}^S(M) \cup \text{Default}(M) \right)$$

where $\text{least}(P)$ denotes the least Herbrand model of the definite program [22] obtained by considering each negative literal not A in P as a new atom.

Having defined the meaning of sequences of programs, we are left with the problem of how to come up with those sequences. This is the subject of LP update languages [2, 4, 10, 18]. Among the existing languages, Evolp [2] uses a particular simple syntax, which extends the usual syntax of GLPs by introducing the special predicate *assert*/1. Given any language \mathcal{L} , the language $\mathcal{L}_{\text{assert}}$ is recursively defined as follows: every atom in \mathcal{L} is also in $\mathcal{L}_{\text{assert}}$; for any rule τ over $\mathcal{L}_{\text{assert}}$, the atom $\text{assert}(\tau)$ is in $\mathcal{L}_{\text{assert}}$; nothing else is in $\mathcal{L}_{\text{assert}}$. An *Evolp program* over \mathcal{L} is any GLP over $\mathcal{L}_{\text{assert}}$. An *Evolp sequence* is a sequence (or DLP) of Evolp programs. The rules of an Evolp program are called *Evolp rules*.

Intuitively an expression $\text{assert}(\tau)$ stands for “update the program with the rule τ ”. Notice the possibility in the language to nest an assert expression in another. The intuition behind the Evolp semantics is quite simple. Starting from the initial Evolp sequence $\oplus P_i^m$ we compute the set, $\mathcal{SM}(\oplus P_i^m)$, of the stable models of $\oplus P_i^m$. Then, for any element M in $\mathcal{SM}(\oplus P_i^m)$, we update the initial sequence with the program P_{m+1} consisting of the set of rules τ such that the atom $\text{assert}(\tau)$ belongs to M . In this way we obtain the sequence $\oplus P_i^m \oplus P_{m+1}$. Since $\mathcal{SM}(\oplus P_i^m)$ contains, in general, several models we may have different lines of evolution. The process continues by obtaining the various $\mathcal{SM}(\oplus P_i^{m+1})$ and, with them, various $\oplus P_i^{m+2}$. Intuitively, the program starts at step 1 already containing the sequence $\oplus P_i^m$. Then it updates itself with the rules asserted at step 1, thus obtaining step 2. Then, again, it updates itself with the rules asserted at this step, and so on. The evolution of any Evolp sequence can also be influenced by external events. An external event is itself an Evolp program. If, at a given step n , the programs receives the external update E_n , the rules in E_n are added to the last self update for the purpose of computing the stable models determining the next evolution but, in the successive step $n+1$ they are no longer considered (that’s why they are called *events*). Formally:

Definition 2. Let n be a natural number. An evolution interpretation of length n , of an evolving logic program $\oplus P_i^m$ with an event sequence $\oplus E_i$ is any finite sequence $\mathcal{M} = M_1, \dots, M_n$ of interpretations over $\mathcal{L}_{\text{assert}}$. The evolution trace $Tr(P)$ associated with an evolution interpretation M_1, \dots, M_n is the sequence $P_1 \oplus \dots \oplus P_{n+m}$ where $P_{m+i} = \{\tau \mid \text{assert}(\tau) \in M_{i-1}\}$ for $m+1 < i \leq n+m$

Definition 3. Let $\oplus P_i^m$ be any Evolp sequence with external events $\oplus E_i^n$ (where n is a natural number), and $\mathcal{M} = M_1, \dots, M_n$ be an evolving interpretation of length n with trace $P_1 \oplus \dots \oplus P_{n+m}$. \mathcal{M} is an evolving stable model of $\oplus P_i^m$ with event sequence $\oplus E_i$ at step n iff M_k is a refined stable model of the program $P_1 \oplus \dots \oplus (P_k \cup E_k)$ for any k with $m+1 \leq k \leq n+m$.

2.2 Action languages

The purpose of an action language is to provide ways of describing how an environment evolves given a set of external actions. A specific environment that can be modified through external actions is called an *action domain*. To any action domain we associate a pair of sets of atoms \mathcal{F} and \mathcal{A} . We call the elements of \mathcal{F} *fluent atoms* or simply *fluents* and the elements of \mathcal{A} *action atoms* or simply *actions*. Basically the fluents are the observable in the environment and the actions are, clearly, the external actions. A *fluent literal* (resp. *action literal*) is an element of \mathcal{F}_{Lit} (resp. an element of \mathcal{A}_{Lit}). In the following, Q will be in general a fluent atom, F a fluent literal and A an action atom. A *state of the world* (or simply a *state*) is any interpretation over \mathcal{F} . We say a fluent literal F is true at a given state s iff F belongs to s .

Each action language provides ways of describing action domains through sets of expression called an *action programs*. Usually, the semantics of an action program is defined in terms of a *transition system* i.e. a function whose argument is any pair (s, K) , where s is a state of the world and K is a subset of \mathcal{A} , and whose value is any set of states of the world. Intuitively, given the current state of the world, a transition system specifies which are the possible resulting states after performing, simultaneously, all the actions in K .

Two kinds of expressions that are common within action description languages are *static and dynamic rules*. The *static rules* basically describe the rule of the domain, while *dynamic rules* describe effects of actions. A dynamic rule has a set of *preconditions*, namely conditions that have to be satisfied in the present state in order to have a particular effect in the future state, and *post-conditions* describing such an effect.

In the following we will consider three existing action languages, namely: \mathcal{A} , \mathcal{B} and \mathcal{C} . The language \mathcal{A} [13] is very simple, allowing only dynamic rules of the form A **causes** F **if** $Cond$ where $Cond$ is a conjunction of fluent literals, such rule intuitively means: performing the action A causes L to be true in the next state if $Cond$ is true in the current state. The language \mathcal{B} [13] is an extension of \mathcal{A} which also considers static rules, i.e. expression of the form F **if** $Body$ where $Body$ is a conjunction of fluent literals which, intuitively, means: if $Body$ is true in the current state, then F is also true in the current state. A fundamental notion in both \mathcal{A} and \mathcal{B} is *fluent inertia* [13]. A fluent F is inertial if its truth value is preserved from a state to another, unless it is changed by the (direct or indirect) effect of an action. For a detailed definition of the semantics of \mathcal{A} and \mathcal{B} see [13].

Static and dynamic rules are also the bricks of the action language \mathcal{C} [16, 15]. Static rules in \mathcal{C} are of the form **caused** J **if** H while dynamic rules are of the form **caused** J **if** H **after** O where J and H are formulae such that any literal in them is a fluent literal and O is any formula such that any literal in it is a fluent or an action literal. The formula O is the precondition of the dynamic rule and the static rule **caused** J **if** H is its postcondition. The semantic of \mathcal{C} is based on *causal theories*[15]. Casual theories are sets of rules of the form **caused** J **if** H meaning: If H is true this is an explanation for J . Within causal theories is that

something is true iff it is caused by something else. Given any action program P , a state s and a set of actions K , we consider the causal theory T given by the static rules of P and the postconditions of the dynamic rules whose preconditions are true in $s \cup K$. Then s' is a possible resulting state iff it is a casual model of T . For a more detailed background on action languages see [12].

3 Evolp action programs

As we have seen, Evolp and action description languages share the idea of a system that evolves. In both, the evolution is influenced by external events (respectively, updates and actions). Evolp is actually a programming language devised for representing any kind of computational problem, while action description languages are devised for the specific purpose of describing actions. A natural idea is then to develop special kind of Evolp sequences for representing actions and then compare such kind of programs with existing action description languages. We will call this kind of programs *Evolp Action Programs* (EAPs).

Following the philosophy of Evolp we use the basic construct *assert* for defining special-purpose macros. As it happens for other action description languages, EAPs are defined over a set of fluents \mathcal{F} and a set of actions \mathcal{A} . A state of the world, in EAPs, is any interpretation over \mathcal{F} . To describe action domains we use an initial Evolp sequence, $I \oplus D$. The Evolp program D contains the description of the environment, while I contains some initial declarations, as it will be clarified later. As in \mathcal{B} and \mathcal{C} , EAPs contain static and dynamic rules.

A *static rule* is simply an Evolp rule of the form $F \leftarrow Body$ where F is a fluent literal and $Body$ is a set of fluent literals.

A *dynamic rule* over $(\mathcal{F}, \mathcal{A})$ is a (macro) expression $\mathbf{effect}(\tau) \leftarrow Cond$ where τ is any static rule $F \leftarrow Body$ and $Cond$ is any set of fluent or action literals. Such an expression simply stand for the following set of Evolp rules:

$$\begin{aligned} F \leftarrow Body, \mathit{event}(L \leftarrow Body) \quad (1) \quad \mathit{assert}(\mathit{event}(F \leftarrow Body)) \leftarrow Cond. \quad (2) \\ \mathit{assert}(\mathit{not event}(F \leftarrow Body)) \leftarrow \mathit{event}(\tau), \mathit{not assert}(\mathit{event}(F \leftarrow Body)) \quad (3) \end{aligned}$$

where $\mathit{event}(F \leftarrow Body)$ is a new literal. The intuitive meaning of such a rule is that the static rule τ has to be considered *only* in those states whose predecessor satisfies condition $Cond$. Since some of the conditions literals in $Cond$ may be action atoms, such a rule may describe the effect of a given set of actions under some conditions. In fact, the above set of rules fits with this intuitive meaning. Rule (1) is not applicable whenever $\mathit{event}(L \leftarrow Body)$ is false. If at some step n the conditions $Cond$ are satisfied, then, by rule (2), $\mathit{event}(L \leftarrow Body)$ becomes true at step $n + 1$. Hence, at step $n + 1$, the rule (1) will play the same role as static rule $F \leftarrow Body$. If at step $n + 1$ $Cond$ is no longer satisfied, then, by rule (3) the literal $\mathit{event}(L \leftarrow Body)$ will become false again and then the rule (1) will be again not effective. The behaviour of **effect** is different from the *assert* command. If we assert τ , it remains by inertia, while with **effect** it lasts for one step only. Moreover, if we assert τ , such rule could reject another rule while a rule inside an **effect** expression does not reject static rules.

Besides static and dynamic rules, we still need another brick to complete our construction. As we have seen in the description of the \mathcal{B} language, a notable concept is fluent inertia. This idea is not explicit in Evolp where *the rules* (and not the fluents) are preserved by inertia. Nevertheless, we can show how to obtain fluent inertia using macro programming in Evolp. An *inertial declarations* over $(\mathcal{F}, \mathcal{A})$ is a (macro) expression $\mathbf{inertial}(\mathcal{K})$, where $\mathcal{K} \subseteq F$. The intended intuitive meaning of such expression is that the fluents in \mathcal{K} are inertial. Before defining what this expression stands for, we state that the program I is always of the form $\mathbf{initialize}(\mathcal{F})$, where $\mathbf{initialize}(\mathcal{F})$ stands for the set of rules (where F is any fluent literal in \mathcal{F}_{Lit} , and $prev(F)$ are new atoms not in $\mathcal{F} \cup \mathcal{A}$): $F \leftarrow prev(F)$. The *inertial declaration* $\mathbf{inertial}(\mathcal{K})$ stands for the set (where F ranges over \mathcal{K}):

$$assert(prev(F)) \leftarrow F. \quad assert(not\ prev(F)) \leftarrow not\ F.$$

Let us consider the behaviour of this macro. If we do not declare F as an inertial fluent the rule $F \leftarrow prev(F)$ has no effect. If we declare F as an inertial literal, $prev(F)$ is true in the current state iff in the previous state F was true. Hence in this case F is true in the current state *unless* there is a static or dynamic rule that rejects such assumption. Viceversa, if F was false in the previous state then, F is true in the current one iff it is derived by a static or dynamic rule. We are now ready to formalize the syntax of Evolp action programs.

Definition 4. *Let \mathcal{F} and \mathcal{A} be two disjoint sets of propositional atoms. An Evolp action program (EAP) over $(\mathcal{F}, \mathcal{A})$ is any Evolp sequence $I \oplus D$ where : $I = \mathbf{Initialize}(\mathcal{F})$, and D is any set consisting of static rules, dynamic rules and inertial declarations over $(\mathcal{F}, \mathcal{A})$*

Given an Evolp action program $I \oplus D$, the initial state of the world s (which, as stated above is an interpretation over \mathcal{F}) is passed to the program together with the set K of the actions performed at s , as part of an external event. A resulting state is the last element of any evolving stable model of $I \oplus D$ given the event $s \cup K$ restricted to the set of fluent literals. I.e:

Definition 5. *Let $I \oplus D$ be any EAP over $(\mathcal{F}, \mathcal{A})$ and s a state of the world. Then s' is a resulting state from s given $I \oplus D$ and the set of actions K iff there exists an evolving stable model M_1, M_2 of $I \oplus D$ given the external event $s \cup K$ such that $s' \equiv_{\mathcal{F}} M_2$ (where by $s' \equiv_{\mathcal{F}} M_2$ we simply mean $s' \cap \mathcal{F}_{Lit} = M_2 \cap \mathcal{F}_{Lit}$).*

The definition can be immediately generalized to sequences of set of actions.

Definition 6. *Let $I \oplus D$ be any EAP and s a state of the world. Then s' is a resulting state from s given $I \oplus D$ and the sequence of actions $K_1 \dots, K_n$ iff there exists an evolving stable model M_1, \dots, M_n of $I \oplus D$ given the external event $s \cup K_1, \dots, K_n$ such that $s' \equiv_{\mathcal{F}} M_n$.*

Since EAPs are based on the Evolp semantics, which is an extension of the stable model semantics for normal logic programs, we can easily prove that the complexity of the computation of the two semantics is the same.

Theorem 1. *Let $I \oplus D$ be any EAP over $(\mathcal{F}, \mathcal{A})$, s a state of the world and $K \subseteq \mathcal{A}$. To find a resulting state s' from s given $I \oplus D$ and the set of actions K is an NP-hard problem.*

It is important to notice that, if the initial state s does not satisfies the static rules of the EAP, the correspondent Evolp sequence has no stable model, and hence there will be no successor state. From now onwards we assume that the initial state satisfies the static rules of the domain.

We now show an example of usage of EAPs by elaborating on probably the most famous example of reasoning about actions. The presented elaboration highlights some important features of EAPs: the possibility of handling non-deterministic effects of actions, non-inertial fluents, non-executable actions, and effects of actions lasting for just one state.

3.1 An elaboration of the Yale shooting problem

In the original Yale shooting problem [26], there is a single-shot gun which is initially unloaded, and a turkey which is initially alive. We can load the gun and shoot the turkey. If we shoot, the gun becomes unloaded and the turkey dies. We consider a slightly more complex scenario where there are several turkeys and where the shooting action refers to a specific turkey. Each time we *shoot* a specific turkey, we either hit and kill the bird or miss it. Moreover the gun becomes unloaded and there is be a bang. It is not possible to shoot with an unloaded gun. We also add the property that any turkey moves iff it is not dead.

For expressing the non executable the problem we make use of a standard technique used in LP under the stable model semantics. Suppose the used EAP contains dynamic rules of the form **effect**($u \leftarrow not\ u$) \leftarrow *Cond* where u is a literal which does not appear elsewhere. In the following we use, for such rules, the notation **effect**(\perp) \leftarrow *Cond*. This kind of rules means that, if *Cond* is true in the current state, then there is no resulting state. This come from the known fact that programs containing $u \leftarrow not\ u$ has no stable models.

To represent this situation we use the set of fluents: $\{dead(X), moving(X), missed(X), hit(X), loaded, bang\}$ plus the auxiliary fluent u , and the actions *load* and *shoot*(X) (where the X is instantiated with the various turkeys). The fluents *dead* and *loaded* are inertial fluents, since their truth value should remain unchanged until modified by some action effect. The fluents *missed*, *hit* and *bang* are not inertial. Finally, for every turkey t , the fluent *moving*(t) is not declared as inertial. The problem is encoded by the EAP $I \oplus D$, where $I = \mathbf{initialize}(loaded, moving(X), dead\ missed(X), hit(X), u)$, and D is

effect (<i>loaded</i>) \leftarrow <i>load</i> .	<i>moving</i> (X) \leftarrow <i>not</i> <i>dead</i> (X)
effect (\perp) \leftarrow <i>shoot</i> (X), <i>not</i> <i>loaded</i>	effect (<i>not</i> <i>loaded</i> .) \leftarrow <i>shoot</i> (X)
effect (<i>dead</i> (X) \leftarrow <i>hit</i> (X)) \leftarrow <i>shoot</i> (X)	effect (<i>bang</i>) \leftarrow <i>shoot</i> (X)
effect (<i>hit</i> (X) \leftarrow <i>not</i> <i>missed</i> (X)) \leftarrow <i>shoot</i> (X)	inertial (<i>loaded</i>)
effect (<i>missed</i> (X) \leftarrow <i>not</i> <i>hit</i> (X)) \leftarrow <i>shoot</i> (X)	inertial (<i>dead</i> (X))

Let us analyze this EAP. Rule **effect**(\perp) \leftarrow *shoot*(X), *not* *loaded* encodes the impossibility to execute the action *shoot*(X) when the gun is unloaded.

The static rule $moving(X) \leftarrow not\ dead(X)$ implies that, for any turkey tk , $moving(tk)$ is true if $dead(tk)$ is false. Since this is the unique rule for $moving(tk)$ we obtain that $moving(tk)$ is true iff $dead(tk)$ is false. Notice that declaring $moving(tk)$ as inertial, would result, in our description, in the possibility of having a moving dead turkey! In fact, suppose we insert $\mathbf{inertial}(moving(X))$ in the EAP above. Suppose further that $moving(tk)$ is true at state s , that we shoot at tk and kill it. Since $moving(tk)$ is an inertial fluent, in the resulting state $dead(tk)$ is true but $moving(tk)$ also remains true by inertia. Also notable is how effects that last only for one state, like the noise provoked by the shoot are easily encoded. The last three dynamic rules encodes a non deterministic behaviour, each shoot action can either hit and kill a turkey or miss it.

We provide an example of a possible evolution. In the following we adopt the usual convention of the Stable Models semantics where we omit the negative literals belonging to an interpretation, hence any interpretation is represented as a set of atoms. Let us consider the initial state $\{\}$. The state will remain unchanged until we perform some action. If we load the the gun, the program is updated by the external event $\{load\}$. In the unique successor state, the fluent $loaded$ is true and nothing else is changed. The truth value of the fluent remains unchanged (by inertia) until we perform some other action. The same applies for the fluents $dead(t)$ where tk is any turkey. The fluents $bang, missed(tk), hit(tk)$ remains false by default. If we shoot at a specific turkey (let us call the turkey Smith) we update the program with the event $shoot(smith)$. Now several things happen. First, $loaded$ become *false*, and $bang$ becomes true, as an effect of the action. Moreover, the rules $hit(smith) \leftarrow missed(smith)$ and $missed(smith) \leftarrow hit(smith)$ are considered as rules of the domain for one state. As a consequence we can have two possible resulting states. In the first one $missed(smith)$ is true, and all the others fluents are false. In the second one $hit(smith)$ is true, $missed(smith)$ is false and, by the static rule $dead(X) \leftarrow hit(X)$, the fluent $dead(smith)$ becomes true. In both the resulting states, nothing happens to the truth value of $dead(tk)$, $hit(tk)$ and $dead(tk)$ for $tk \neq smith$.

4 Relationship to existing action languages

In this section we show embeddings into EAPs of the action languages \mathcal{B} and (the definite fragment of) \mathcal{C} ⁴. We will assume that the considered initial states are consistent wrt the static rules of the program, i.e. if the body of a static rule is true in the considered state, the head is true as well.

Let us consider first the \mathcal{B} language. The basic ideas of static and dynamic rules of \mathcal{B} and EAPs are very similar. The main difference between the two is that in \mathcal{B} *all* the fluents are considered as inertial, whilst in EAPs only those that are declared as such are inertial. The translation of \mathcal{B} into EAPs is then straightforward: All fluents are declared as inertial and then the syntax of static

⁴ The embedding of language \mathcal{A} is not explicitly exposed here since \mathcal{A} is a (proper) subset of the \mathcal{B} language.

and dynamic rules is adapted. In the following we use, with abuse of notation, *Body* and *Cond* both for conjunctions of literals and for sets of literals.

Definition 7. Let P be any action program in \mathcal{B} over the fluent language \mathcal{F} . The translation $B(P, \mathcal{F})$ is the couple $(I^B \oplus D^{BP}, \mathcal{F}^B)$ where: $\mathcal{F}^B \equiv \mathcal{F}$, $I^B = \mathbf{initialize}(\mathcal{F})$ and D^{BP} contains exactly the following rules:

- **inertial**(F) for each fluent $F \in \mathcal{F}$
- a rule $L \leftarrow \mathbf{Body}$ for any static rule $L \mathbf{if} \mathbf{Body}$ in P .
- a rule **effect**(L) $\leftarrow A, \mathbf{Cond}$. for any dynamic rule $A \mathbf{causes} L \mathbf{if} \mathbf{Cond}$ in P .

Theorem 2. Let P be any \mathcal{B} action program over \mathcal{F} , $(I^B \oplus D^{BP}, \mathcal{F})$ its translation, s a state and K any set of actions. Then s' is a resulting state from s given P and the set of actions K iff it is a resulting state from s given $I^B \oplus D^{BP}$ and the set of actions K .

Let us consider now the action language \mathcal{C} . It is known that the computation of the possible resulting states in the full \mathcal{C} language is \sum_P^2 -hard, [15]. So, this language belongs to a category with higher complexity than EAPs which are NP-hard. However, only a fragment of \mathcal{C} is implemented and the complexity of such fragment is NP. This fragment is known as the *definite fragment* of \mathcal{C} [15]. In such fragment static rules are expressions of the form **caused** F **if** \mathbf{Body} where F is a fluent literal and \mathbf{Body} is a conjunction of fluent literals, while dynamic rules are expressions of the form **caused not** F **if** \mathbf{Body} **after** \mathbf{Cond} where \mathbf{Cond} is a conjunction of fluent or action literals⁵. For this fragment it is possible to provide a translation into EAPs.

The main problem of the translation of \mathcal{C} into EAPs lies the simulation of causal reasoning with stable model semantics. The approach followed here to encode causal reasoning with stable models is in line with the one proposed in [20]. We need to introduce some auxiliary predicates and define a syntactic transformation of rules. Let \mathcal{F} be a set of fluents, by \mathcal{F}^C we denote the set of fluents $\mathcal{F} \cup \{F_N \mid F \in \mathcal{F}\}$. We add, for each $F \in \mathcal{F}$, the constraints:

$$\leftarrow \mathbf{not} F, \mathbf{not} F_N. \quad \leftarrow F, F_N. \quad (2)$$

Let F be a fluent and $\mathbf{Body} = F_1, \dots, F_n$ a conjunction of fluent literals. We will use the following notation: $\overline{F} = \mathbf{not} F_N$, $\overline{\mathbf{not} F} = \mathbf{not} F$ and $\overline{\mathbf{Body}} = \overline{F_1}, \dots, \overline{F_n}$

Definition 8. Let P be any action program in \mathcal{C} over the fluent language \mathcal{F} . The translation $C(P, \mathcal{F})$ is the couple $(I^C \oplus D^{CP}, \mathcal{F}^C)$ where: \mathcal{F}^C is defined as above, $I^C \equiv \mathbf{initialize}(\mathcal{F}^C)$ and D^{CP} consists exactly of the following rules:

- a rule **effect**($F \leftarrow \overline{\mathbf{Body}}$) $\leftarrow \mathbf{Cond}$, for any dynamic rule in P of the form **caused** F **if** \mathbf{Body} **after** \mathbf{Cond} ;

⁵ The definite fragment defined in [15] is (apparently) more general, allowing \mathbf{Cond} and \mathbf{Body} to be arbitrary formulae. However, it is easy to prove that such kind of expressions are equivalent to a set of expressions of the form described above

- a rule $\mathbf{effect}(F_N \leftarrow \overline{Body}) \leftarrow Cond$, for any dynamic rule in P of the form **caused not F if $Body$ after $Cond$** ;
- a rule $F \leftarrow \overline{Body}$, for any static rule in P of the form **caused F if $Body$** ;
- a rule $F_N \leftarrow \overline{Body}$, for any static rule in P of the form **caused not F if $Body$** ;
- The rules (2) for each fluent in \mathcal{F} .

For this translation we obtain a result similar to the one obtained for the translations of the \mathcal{B} language. In this case:

Theorem 3. *Let P be any \mathcal{C} action program over \mathcal{F} , $(I^C \oplus D^{CP}, \mathcal{F}^C)$ its translation, s a state, s^C the interpretation over \mathcal{F}^C defined as follows:*

$$s^C = s \cup \{\overline{Q} \mid Q \in s\} \cup \{\text{not } \overline{Q} \mid \text{not } Q \in s\}$$

and K any set of actions. Then s^* is a resulting state from s^C given $I^C \oplus D^{CP}$ and the set of actions K iff there exists s' such that s' is a resulting state from s , given P and the set of actions K .

By showing translations of the action languages \mathcal{B} and \mathcal{C} into EAPs, we proved that EAPs are *at least as expressive* as such languages. Moreover the provided translations are quite simple (basically one EAP static or dynamic rule for each static or dynamic rule in the other languages). The next natural question is: Are they *more expressive*?

5 Updates of action domains

Action description languages describe the rules governing a domain where actions are performed. In practical situations, it may happen that the very rules of the domain change with time too. EAPs are just a particular kind of Evolp sequences. So, as in general Evolp sequences they can be updated by external events.

When we want to update the existing rules by the rule τ , we just add the fact $\mathit{assert}(\tau)$ as an external event. This way, the rule τ is asserted and the existing Evolp sequence is updated. Following this line, we extend EAPs, allowing the external events updating an EAP to contain facts of the form $\mathit{assert}(\tau)$ where τ is an Evolp rule and show how they can be used to express updates to EAPs.

To illustrate how to update an EAP, we come back to the example of section 3.1. Let $I \oplus D$ be the EAP defined in that section. Let us now consider that after some shots, and dead turkeys, we acquire rubber bullets. We can now either load the gun with normal bullets or with a rubber bullet, but not with both. If we shoot with a rubber loaded gun, we never kill a turkey.

To describe this change in the domain, we introduce a new inertial fluent representing the gun being loaded with rubber bullets. We have to express that, if the gun is rubber-loaded, we can not kill the turkey. For this purpose we introduce the new macro: $\mathbf{not\ effect}(F \leftarrow Body) \leftarrow Cond$ where F , is a fluent literal, $Body$ is a set of fluents literals and $Cond$ is a set of fluent or action literals. We refer to such expressions as *effects inhibitions*. This macro simply stands for the rule $\mathit{assert}(\mathbf{not\ event}(F \leftarrow Body)) \leftarrow Cond$ where $\mathit{event}(F \leftarrow Body)$ is a

new atom. The intuitive meaning is that, if the condition $Cond$ is true in the current state, any dynamic rule whose effect is the rule $F \leftarrow Body$ is ignored.

To encode the changes described above, we update the EAP with the external event E_1 consisting of the facts $assert(I_1)$ where $I_1 = (\mathbf{initialize}(rubber_loaded))$. Then, in the subsequent state, we update the program with the external update $E_2 = assert(D_1)$ where D_1 is the set of rules⁶

$$\begin{aligned} \mathbf{effect}(\perp) &\leftarrow rubber_loaded, load. \\ \mathbf{inertial}(rubber_loaded) & \\ \mathbf{effect}(\perp) &\leftarrow loaded, rubber_load. \\ \mathbf{not\ effect}(dead(X) \leftarrow hit(X)) &\leftarrow rubber_loaded. \end{aligned}$$

Let us analyze the proposed updates. First, the fluent $rubber_loaded$ is initialized. It is important to initialize any fluent before starting to use it. The newly introduced fluent is declared as inertial and two dynamic rules are added specifying that load actions are not executable when the gun is already loaded in a different way. Finally we use the new command to specify that the effect $dead(X) \leftarrow hit(X)$ does not occur if, in the previous state, the gun was loaded with rubber bullets. Since this update is more recent than the original rule $\mathbf{effect}(dead(X) \leftarrow hit(X)) \leftarrow shoot(X)$, such dynamic rule is updated.

It is also possible to update static rules and the descriptions of effects of an action. Suppose the cylinder of the gun becomes dirty and, whenever one shoots, the gun may either work properly or fail. If the gun fails, the action $shoot$ has no effect. We introduce two new fluents in the program with the event $assert(I_2)$ where $I_2 = \mathbf{initialize}(fails, work)$. Then, we assert the event $E_2 = assert(D_2)$ where D_2 is the following EAP

$$\begin{aligned} \mathbf{effect}(fails \leftarrow not\ work) &\leftarrow shoot(X). & \mathbf{effect}(work \leftarrow not\ fails) &\leftarrow shoot(X). \\ not\ bang &\leftarrow fails. & not\ unloaded &\leftarrow fails. \\ not\ missed &\leftarrow fails. & not\ missed &\leftarrow fails. \end{aligned}$$

This last example is important since it shows how to update the effects of a dynamic rule via a new static rule. It is also possible to update the effects of a dynamic rule via another dynamic rule. We show a possible evolution of the updated system. Suppose currently the gun is not loaded. We load the gun with a rubber bullet and then we shoot to the turkey named Trevor. The initial state is $\{\}$. The first set of actions is $\{rubber_load\}$. The resulting state after this action is $s' \equiv \{rubber_loaded\}$. Suppose we perform the action $load$. Since the EAP is updated with the dynamic rule $\mathbf{effect}(\perp) \leftarrow rubber_loaded, load$, there is no resulting state. This happens because we have performed a non executable action. Suppose, instead, the second set of actions is $\{shoot(trevor)\}$. There are three possible resulting states. In one the gun fails. In this case, the resulting state is, again, s' . In the second, the gun works but the bullet misses Trevor. In this case, the resulting state is $s''_1 \equiv \{missed(trevor)\}$. Finally, the gun works

⁶ In the remainder we use the notation $assert(U)$, where U is a set of macros (which are themselves sets of Evolp rules) for the set of all facts $assert(\tau)$ such that τ is a rule used in: there exists a macro η in U with $\tau \in \eta$.

and the bullet hits Trevor. Since the bullet is a rubber bullet, Trevor is still alive. In this case the resulting state is $s_2'' \equiv \{hit(trevor)\}$.

The events introduced changes in the behaviour of the original EAP. This opens a new problem. In classical action languages we do not care about the previous *history* of the world: If the current state of the world is s , the computation of the resulting states is not affected by the states before s . In the case of EAPs the situation is different, since external updates can change the behaviour of the considered EAP. Fortunately, we do not have to care about the *whole* history of the world, but just about those events containing new initializations, inertial declarations, effects inhibitions, and static and dynamic rules.

It is possible to have a compact description of an EAP that is updated several times via external events. For that we need to further extend the original definition of EAPs.

Definition 9. An updated *Evolp* action program over $(\mathcal{F}, \mathcal{A})$ is any sequence $I \oplus D_1 \oplus \dots \oplus D_n$ where I is *initialize*(\mathcal{F}), and the various D_k are sets consisting of static rules, dynamic rules, inertial declarations and effects inhibitions such that any fluent appearing in D_k belongs to \mathcal{F}

In general, if we update an *Evolp* action program $I \oplus D$ with the subsequent events $assert(I_1)$, $assert(D_1)$, where $I_1 \oplus D_1$ is another EAP, we obtain the equivalent updated *Evolp* action program $(I \cup I_1) \oplus D \oplus D_1$ Formally:

Theorem 4. Let $I \oplus D_1 \oplus \dots \oplus D_k$ be any update EAP over $(\mathcal{F}, \mathcal{A})$. Let $\bigoplus E_i^n$ be a sequence of events such that: $E_1 = K_1 \cup s$ where s is any state of the world and K is any set of actions and the others E_i s are: any set of actions K_α , or any set *initialize*(\mathcal{F}_β) where $\mathcal{F}_\beta \subseteq \mathcal{F}$, or any D_i with $1 \leq i \leq k$. Then s' is a resulting state from s given $I \oplus D_1 \oplus \dots \oplus D_k$ and the sequence of sets of actions $\bigoplus K_\alpha$ iff there exists an evolving stable model M_1, \dots, M_n of $I \oplus D$ with event sequence $\bigoplus E_i^n$ such that $M_n \equiv_{\mathcal{F}} s$

For instance, the updates to the original EAP of the example in this section are equivalent to the updated EAP is $I_{sum} \oplus D \oplus D_1 \oplus D_2$ such that $I_{sum} \equiv I \cup I_1 \cup I_2$ where I and D are as in example of section 3.1 and the I_i s and D_i s are as in the description above.

Yet one more possibility opened by updated *Evolp* action programs is to cater for successive elaborations of a program. Consider an initial problem described by an EAP $I \oplus D$. If we want to describe an elaboration of the program, instead of *rewriting* $I \oplus D$ we can simply *update* it with new rules. This gives a new answer to the problem of elaboration tolerance [24] and also open the new possibility of *automatically update* action programs by other action programs.

The possibility to elaborate an action program is also discussed in [15] in the context of the $\mathcal{C}+$ language. The solution proposed is to consider $\mathcal{C}+$ programs whose rules have one extra fluent atom in their bodies that are assumed false by default. The elaboration of an action program P is the program $P \cup U$ where U is a new action program. The rules in U can defeat the rules in P by changing the truth value of the extra literals in their bodies. An advantage of EAP is that, in this framework, the possibility of updating rules is a built-in feature rather

then a programming technique involving manipulation of rules and introduction of new fluents. Moreover, in EAPs we can simply encode the new behaviours of the domain by new rules and then let these new rules update the previous ones.

6 Conclusions and future work

In this paper we have explored the possibility of using logic programs updates languages as action description languages. In particular we have focused our attention on the Evolp language. As a first point, we have defined a new action language paradigm, christened Evolp action programs, defined as a macro language over Evolp. We have provided an example of usage of such language. We have compared Evolp action programs with action languages \mathcal{A} , \mathcal{B} , \mathcal{C} and provided simple translations into Evolp of these languages. Moreover we have shown, both by theoretical argumentation and practical examples, how some expressive capabilities of EAPs seem to be not replicable in these languages. Finally we have also shown and argued about the capability of $(\oplus P_i^m, \oplus E_i)$ to handle changes in the domain during the execution of actions.

Several important topics are not touched here, and will be subject of future work. An important field of research is how to deal, in the Evolp context, with the problem of planning [23]. Yet another topic involves the possibility of concurrent execution of actions. *EAPs* allow this possibility, nevertheless, we have not fully explored this topic, and confronted the results with extant works [6, 17]. Finally EAPs have to be implemented and tested in real and complex contexts.

References

1. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. Semantics for dynamic logic programming: a principled based approach. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, volume 1730 of *LNAI*, Berlin, 2004. Springer.
2. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, volume 2424 of *LNAI*, pages 50–61. Springer-Verlag, 2002.
3. 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.
4. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002.
5. J. J. Alferes, L. M. Pereira, T. Przymusinski, H. Przymusinska, and P. Quaresma. Preliminary exploration on actions as updates. In M. C. Meo and M. V. Ferro, editors, *Proceedings of the 1999 Joint Conference on Declarative Programming (AGP-99)*, 1999.
6. C. Baral and M. Gelfond. Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31:85–118, 1997.

7. C. Baral, M. Gelfond, and Alessandro Provetti. Representing actions: Laws, observations and hypotheses. *Journal of Logic Programming*, 31, April–June 1997.
8. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming (ICLP-99)*, Cambridge, November 1999. MIT Press.
9. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of semantics based on causal rejection. *Theory and Practice of Logic Programming*, 2:711–767, November 2002.
10. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specifications in logic programs. In Bernhard Nebel, editor, *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, pages 649–654, San Francisco, CA, 2001. Morgan Kaufmann Publishers, Inc.
11. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
12. M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.
13. M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 16, 1998.
14. E. Giunchiglia, J. Lee, V. Lifschitz, N. Mc Cain, and H. Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.
15. E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 2003.
16. E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI'98*, pages 623–630, 1998.
17. J. Lee and V. Lifschitz. Describing additive fluents in action language C+. In William Nebel, Bernhard Rich, Charles Swartout, editor, *Proc. IJCAI-03*, pages 1079–1084, Cambridge, MA, 2003. To Appear.
18. J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
19. J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. In *LPKR'97: workshop on Logic Programming and Knowledge Representation*, 1997.
20. V. Lifschitz. *The Logic Programming Paradigm: a 25-Year Perspective*, chapter Action languages, answer sets and planning, pages 357–373. Springer Verlag, 1999.
21. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the 3th International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*. Morgan-Kaufmann, 1992.
22. John Wylie Lloyd. *Foundations of Logic Programming*. Springer, Berlin, Heidelberg, New York, 1987.
23. J. McCarthy. Programs with commons sense. In *Proceedings of Teddington Conference on The Mechanization of Thought Process*, pages 75–91, 1959.
24. J. McCarthy. *Mathematical logic in artificial intelligence*, pages 297–311. Daedalus, 1988.
25. S. Russel and P. Norvig. *Artificial Intelligence A Modern Approach*, page 4. Artificial Intelligence. Prentice Hall, 1995.
26. D. McDermott S. Hanks. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33:379–412, (1987).