LUPS – a language for updating logic programs

José Júlio Alferes[1,2], Luís Moniz Pereira[2],
Halina Przymusinska[3,4], and Teodor C. Przymusinski[4]

[1] Dept. de Matemática, Univ. Évora,
Rua Romão Ramalho, 59, P-7000 Évora, Portugal, jja@dmat.uevora.pt
[2] Centro de Inteligência Artificial, Fac. Ciências e Tecnologia, Univ. Nova de Lisboa,
P-2825-114 Caparica, Portugal, lmp@di.fct.unl.pt,
Voice:+351 1 294 8533, Fax: +351 1 294 8541
[3] Dept. Computer Science, California State Polytechnic Univ.
Pomona, CA 91768, USA, halina@cs.ucr.edu
[4] Dept. Computer Science, Univ. of California
Riverside, CA 92521, USA, teodor@cs.ucr.edu

Abstract. Most of the work conducted so far in the field of logic programming has focused on representing static knowledge, i.e. knowledge that does not evolve with time. To overcome this limitation, in a recent paper, the authors introduced the concept of *dynamic logic programming*. There, they studied and defined the declarative and operational semantics of sequences of logic programs (or dynamic logic programs), $P_0 \oplus \ldots \oplus P_n$. Each such program contains knowledge about some given state, where different states may, e.g., represent different time periods or different sets of priorities. The role of dynamic logic programming is to employ relationships existing between the possibly mutually contradictory sequence of programs to precisely determine, at any given state, the declarative and procedural semantics of their combination.

But how, in concrete situations, is a sequence of logic programs built? For instance, in the domain of actions, what are the appropriate sequences of programs that represent the performed actions and their effects? Whereas dynamic logic programming provides a way for determining what should follow, given the sequence of programs, it does not provide a good practical language for the specification of updates or changes in the knowledge represented by successive logic programs.

In this paper we define a language designed for specifying changes to logic programs (LUPS – "Language for dynamic updates"). Given an initial knowledge base (in the form of a logic program) LUPS provides a way for sequentially updating it. The declarative meaning of a sequence of sets of update actions in LUPS is defined using the semantics of the dynamic logic program generated by those actions. We also provide a translation of the sequence of update statements sets into a single generalized logic program written in a meta-language, so that the stable models of the resulting program correspond to the previously defined declarative semantics. This meta-language is used in the actual implementation, although this is not the subject of this paper. Finally we briefly mention related work (lack of space prevents us from presenting more detailed comparisons).

1 Introduction

Several authors [9, 10, 2] have addressed the issue of updates of logic programs and deductive databases, most of them following the so called *"interpretation update"* approach. This approach, proposed in [11, 7], is based on the idea of reducing the problem of finding an update of a knowledge base DB by another knowledge base U to the problem of finding updates of its individual interpretations or models. More precisely, a knowledge base DB' is considered to be the update of a knowledge base DB by U if the set of models of DB' coincides with the set of updated models of DB. As pointed out in [1], such an approach suffers from several important drawbacks: first, it requires the computation of all models of DB, before computing the update; second, the resulting knowledge base DB' is only indirectly characterized (as one whose models are all the updated models of the original DB) – no direct definition of DB' is provided; last, and most importantly, it leads to counterintuitive results when the intensional part of the knowledge base (i.e. the set of rules) changes. In [2] the authors eliminated the first two drawbacks by showing how to, given a program P, construct another program P' whose models are exactly the interpretation updates of the models of P. However the last, and most important, drawback still remained: no method to update logic programs consisting of rules, not just extensional facts, was provided.

Example 1. Consider the logic program:

$$\begin{aligned} free &\leftarrow not\, jail \\ jail &\leftarrow abortion \end{aligned}$$

whose only stable model is $M = \{free\}$. Suppose now that the update U states that *abortion* becomes true, i.e. $U = \{abortion \leftarrow\}$. According to the interpretation approach to updating, we would obtain $\{free, abortion\}$ as the only update of M by U. However, by inspecting the initial program and the update, we are likely to conclude that, since $free$ was true only because $jail$ could be assumed false, and that was the case because *abortion* was false, now that *abortion* became true $jail$ should also have become true, and $free$ should be removed from the conclusions.

Suppose now that the law changes, so that abortion no longer implies jail. That could, for example, be described by the new (update) program $U_2 = \{not\, jail \leftarrow abortion\}$. We should now expect $jail$ to become false and, consequently, $free$ to become true (again).

This example suggests that the principle of inertia should be applied not just to individual literals but rather to the whole rules of the knowledge base. It also suggests that the update of a knowledge base by another one should not just depend on their semantics, it should also depend on their syntax. It also illustrates the need for some way of representing negative conclusions.

In [1], the authors investigated the problem of updating knowledge bases represented by generalized logic programs[1] and proposed a new approach to this problem that eliminates the drawbacks of previously proposed solutions. It starts by defining the *update* of a generalized program P by another generalized program U, $P \oplus U$. The semantics of $P \oplus U$ avoids the above mentioned problems by applying the inertia principle not just to atoms but to entire program rules. This notion of updates is then extended to sequences of programs, thereby defining the so-called *dynamic logic programming*. A dynamic logic program is a (finite or infinite) sequence $P_0 \oplus \ldots \oplus P_n \oplus \ldots$, representing consecutive updates of logic programs by logic programs. The semantics defined in [1] assigns meaning to such sequences.

However, dynamic logic programming does not by itself provide a proper language for specifying (or programming) changes of logic programs. If knowledge is already represented by logic programs, dynamic programs simply represent the evolution of knowledge. But how is that evolving knowledge specified? What makes knowledge evolve? Since logic programs describe knowledge states, it's only fit that logic programs describe transitions of knowledge states as well. It is natural to associate with each state a set of transition rules to obtain the next state. As a result, an interleaving sequence of states and rules of transition will be obtained. Imperative programming specifies transitions and leaves states implicit. Logic programming, up to now, could not specify state transitions. With the language of dynamic updates LUPS we make both states and their transitions declarative.

Usually updates are viewed as actions or commands that make the knowledge base evolve from one state to another. This is the classical view e.g. in relational databases: the knowledge (data) is expressed declaratively via a set of relations; updates are commands that change the data. In [1], updates were viewed declaratively as a given update store consisting of the sequence of programs. They were more in the spirit of state transition rules, rather than commands. Of course, one could say that the update commands were implicit. For instance, in example 1, the sequence $P \oplus U \oplus U_2$ could be viewed as the result of, starting from P, performing first the update command *assert abortion*, and then the update command *assert not jail* $\leftarrow$ *abortion*. But, if viewed as a language for (implicitly) specifying update commands, dynamic logic programming is quite poor. For instance, it does not provide any mechanism for saying that some rule (or fact) should be asserted only whenever some conditions are satisfied. This is essential in the domain of actions, to specify direct effects of actions. For example, suppose we want to state that *wake_up* should be added to our knowledge base whenever *alarm_rings* is true. As a language for specifying updates, dynamic logic programming does not provide a way of specifying such an update command. Note that the command is distinct from *assert wake_up* $\leftarrow$ *alarm_rings*. With the latter, if the alarm stops ringing (i.e. if *not alarm_rings* is later asserted), *wake_up* becomes false. In the former, we expect *wake_up* to remain true (by

[1] i.e. logic programs which allow default negation not only in rule bodies but also in the heads.

inertia) even after the alarm stops ringing. As a matter of fact, in this case, we don't want to add the rule saying that *wake_up* is true whenever *alarm_rings* is also true. We simply want to add the fact *wake_up* as soon as *alarm_rings* is true. From there on, no connection between *wake_up* and *alarm_rings* should persist.

This simple one-rule example also highlights another limitation of dynamic logic programming as a language for specifying update commands: one must explicitly say to which program in the sequence a rule belongs to. Sometimes, in particular in the domain of actions, there is no way to know a priori to which state (or program) a rule should belong to. Where should we assert the fact *wake_up*? This is not known a priori because we don't know when *alarm_rings*.

In this paper we define (in section 3) a language for specifying logic program updates: LUPS – "Language of dynamic updates". The object language of LUPS is that of generalized logic programs. A sentence U in LUPS is a set of simultaneous update commands (or actions) that, given a pre-existing sequence of logic programs $P_0 \oplus \ldots \oplus P_n$ (i.e. a dynamic logic program), whose semantics corresponds to our knowledge at a given state, produces a sequence with one more program, $P_0 \oplus \ldots \oplus P_n \oplus P_{n+1}$, corresponding to the knowledge that results from the previous sequence after performing all the simultaneous commands. A program in LUPS is a sequence of such sentences.

Given a program in LUPS, its semantics is first defined (in section 4) by means of a dynamic logic program generated by the sequence of commands. In section 5, we also describe a translation of a LUPS program into a generalized logic program, whose stable models exactly correspond to the semantics of the original LUPS program. Finally, in section 6, we briefly discuss some of the similarities and differences between LUPS and "Action Languages".

2 Object language

In order to represent *negative* information in logic programs and their updates, we need more general logic programs, allowing default negation *not* A not only in the premises of clauses but also in their heads[2]. In [1], such programs are dubbed *generalized logic programs*, and their semantics is defined as a generalization of the stable model semantics of normal logic programs [4][3].

For convenience, generalized logic programs are *syntactically* represented as *propositional Horn theories.* In particular, default negation *not* A is represented as a standard propositional variable (atom). Suppose that $\mathcal{K}$ is an arbitrary set of propositional variables whose names do not begin with a " *not*". By the

[2] See [1] for an explanation on why default negation is needed in rule heads, rather than explicit negation. Note that a default negated atom in a rule's head means that the atom should no longer be assumed true, whilst an explicit negated atom would mean that the atom should become false. In an update context this difference is similar to the difference between deleting a fact and asserting its complement.

[3] The class of generalized logic programs can be viewed as a special case of a yet broader class of programs introduced earlier in [5].

propositional language $\mathcal{L}_\mathcal{K}$ generated by the set $\mathcal{K}$ we mean the language whose set of propositional variables consists of $\{A : A \in \mathcal{K}\} \cup \{not\, A : A \in \mathcal{K}\}$. Atoms $A \in \mathcal{K}$, are called *objective atoms* while the atoms $not\, A$ are called *default atoms*. From the definition it follows that the two sets are disjoint. By "literals" we mean objective or default atoms in $\mathcal{L}_\mathcal{K}$.

Definition 1 (Generalized Logic Program). *A generalized logic program P in the language $\mathcal{L}_\mathcal{K}$ is a (possibly infinite) set of propositional rules of the form $L \leftarrow L_1, \ldots, L_n$, where $L, L_1, \ldots, L_n$ are literals.*

If none of the literals appearing in heads of rules of P are default ones, then we say that the logic program P is normal.

By a (2-valued) *interpretation* M of $\mathcal{L}_\mathcal{K}$ we mean any set of atoms from $\mathcal{L}_\mathcal{K}$ satisfying the condition that for any A in $\mathcal{K}$, precisely one of the atoms A or $not\, A$ belongs to M. Given an interpretation M we define:

$$M^+ = \{A \in \mathcal{K} : A \in M\} \quad \text{and} \quad M^- = \{not\, A : not\, A \in M\} = \{\ not\, A : A \notin M\}$$

By a (2-valued) *model* M of a generalized logic program we mean a (2-valued) interpretation that satisfies all of its clauses.

Definition 2 (Stable models of generalized logic programs). *We say that a (2-valued) interpretation M of $\mathcal{L}_\mathcal{K}$ is a stable model of a generalized logic program P if M is the least model of the Horn theory $P \cup M^-$, or, equivalently, if $M = \{L : L$ is a literal and $P \cup M^- \vdash L\}$.*

As proven in [1], the class of stable models of generalized logic programs extends the class of stable models of normal programs [4], in the sense that, for the special case of normal programs, both semantics coincide.

3 Language for updates

In our update framework, knowledge evolves from one knowledge state to another as a result of update commands stated in object language.

Knowledge states KS_i represent dynamically evolving states of our knowledge. They undergo change due to *update actions*. Without loss of generality (as will become clear below) we assume that the initial knowledge state, KS_0, is empty and that in it all predicates are *false* by default. This is the *default knowledge state*. Given the *current knowledge state* KS, its *successor knowledge state* $KS[U]$ is produced as a result of the occurrence of a non-empty set U of simultaneous *updates*. Each of the updates can be viewed as a set of (parallel) *actions* and consecutive knowledge states are obtained as $KS_n = KS_0[U_1][U_2]...[U_n]$ where U_i's represent consecutive sets of updates. We also denote this state by:

$$KS_n = U_1 \otimes U_2 \otimes \ldots \otimes U_n$$

So defined sequences of updates will be called *update programs*. In other words, an update program is a finite sequence $\mathcal{U} = \{U_s : s \in S\}$ of updates indexed by the set $S = \{1, 2, \ldots, n\}$. Each updates is a set of update commands. Update commands (to be defined below) specify *assertions* or *retractions* to the current knowledge state. By the current knowledge state we mean the one resulting from the last update performed.

Knowledge can be queried at any state $q \leq n$, where n is the index of the current knowledge state. A query will be denoted by:

$$holds(B_1, \ldots, B_k, not\, C_1, \ldots, not\, C_m)\ at\ q?$$

and is true iff the conjunction of its literals holds at the state KB_q. If $q = n$, we simply skip the state reference "*at q*".

3.1 Update commands

Update commands cause changes to the current knowledge state leading to a new successor state. The simplest command consists of adding a rule to the current state: $assert\ L \leftarrow L_1, \ldots, L_k$. For example, when a law stating that abortion is punished by jail is approved, the knowledge state might be updated via the command: $assert\ jail \leftarrow abortion$.

In general, the addition of a rule to a knowledge state may depend upon some precondition. To allow for that, an assert command in LUPS has the form:

$$assert\ L \leftarrow L_1, \ldots, L_k\ when\ L_{k+1}, \ldots, L_m \tag{1}$$

The meaning of such assert rule is that if the precondition $L_{k+1}, \ldots, L_m$ is true in the current knowledge state, then the rule $L \leftarrow L_1, \ldots, L_k$ should belong to the successor knowledge state. Normally, the so added rule persists, or is in force, from then on by inertia, until possibly defeated by some future update or until retracted. This is the case for the assert-command above: the rule $jail \leftarrow abortion$ remains in effect by inertia from the sucessor state onwards unless later invalidated.

However, there are cases where this persistence by inertia should not be assumed. Take, for instance, the *alarm_ring* discussed in the introduction. This fact is a one-time event that should not persist by inertia, i.e. it is not supposed to hold by inertia after the successor state. In general, facts that denote names of events or actions should be *non-inertial*. Both are true in the state they occur, and do not persist by inertia for later states. Accordingly, the rule within the assert command may be preceeded with the keyword *event*, indicating that the added rule is non-inertial. Assert commands are thus of the form (1) or of the form[4]:

$$assert\ event\ L \leftarrow L_1, \ldots, L_k\ when\ L_{k+1}, \ldots, L_m \tag{2}$$

While some update commands, such as *assert republican_congress*, represent newly incoming information, and are thus one-time non-persistent update commands (whose effect, i.e. the truth of *republican_congress*, may nevertheless

[4] In both cases, if the precondition is empty we just skip the whole *when* subclause.

persist by inertia), some other update commands are liable to be *persistent*, i.e., to remain in force until cancelled. For example, an update like:

$$assert\ jail \leftarrow abortion\ when\ rep\_congress, rep\_president$$

or

$$assert\ wake\_up\ when\ alarm\_sounds$$

might be always true, or at least true until cancelled. Enabling the possibility of such updates allows our system to dynamically change without any truly new updates being received. For example, the persistent update command:

$$assert\ set\_hands(T)\ when\ get\_hands(C) \wedge get\_time(T) \wedge (T - C) > \Delta$$

defines a perpetually operating clock whose hands move to the actual time position whenever the difference between the clock time and the actual time is sufficiently large.

In order to specify such persistent updates commands (which we call laws) we introduce the syntax:

$$always\ [event]\ L \leftarrow L_1, \ldots, L_k\ when\ L_{k+1}, \ldots, L_m \tag{3}$$

For cancelling persistent update commands, we use:

$$cancel\ L \leftarrow L_1, \ldots, L_k\ when\ L_{k+1}, \ldots, L_m \tag{4}$$

The first statement means that, in addition to any new set of arriving update commands, we are also supposed to keep executing this persistent update command. The second statement cancels this persistent update, when the conditions for cancelation are met.

The existence of persistent update commands requires a “trivial” update, which does not specify any truly new updates but simply triggers all the already defined persistent updates to fire, thus resulting in a new modified knowledge state. Such “no-operation” update ensures that the system continues to evolve, even when no truly new updates are specified, and may be represented by *assert true*. It stands for the *tick of the clock* that drives the world being modelled.

To deal with the deletion of rules, we introduce the *retraction* command:

$$retract\ [event]\ L \leftarrow L_1, \ldots, L_k\ when\ L_{k+1}, \ldots, L_m \tag{5}$$

meaning that, subject to precondition $L_{k+1}, \ldots, L_m$, the rule $L \leftarrow L_1, \ldots, L_k$ is either retracted from now on, or just retracted temporarily in the next state (non-inertial retract, i.e. an event of retraction, triggered by the *event* keyword).

The cancelling of an update command is not equivalent to retracting a rule. Cancelling an update just means it will no longer be added as a command to updates, it does not cancel the inertial effects of its previous application(s). However, retracting an update causes any of its inertial effects to be cancelled from now on, as well as cancelling a persistent law. Also, note that "*retract event ...*" does not mean the retracting of an event, because events persist only for one state and thus do not require retraction. It represents a temporary removal of a rule from the successor state (a temporary retraction event).

Definition 3 (LUPS). *An update program in LUPS is a finite sequence of updates, where an update is a set of commands of the form (1) to (5).*

Example 2. Consider the scenario: once Republicans take over both Congress and the Presidency they establish a law stating that abortions are punishable by jail; once Democrats take over both Congress and the Presidency they abolish such a law; in the meantime, there are no changes in the law because always either the President or the Congress vetoes such changes; performing an abortion is an event, i.e. a non-inertial update. Consider the following update history: (1) a Democratic Congress and a Republican President (Reagan); (2) Mary performs abortion; (3) Republican Congress is elected (Republican President remains in office: Bush); (4) Kate performs abortion; (5) Clinton is elected President; (6) Ann performs abortion; (7) Gore is elected President and Democratic Congress is in place (year 2000?); (8) Susan performs abortion.

The specification in LUPS would be:

Persistent update commands:

$$\begin{array}{l} always\ jail(X) \leftarrow abn(X)\ when\ repC \wedge repP \\ always\ not\, jail(X) \leftarrow abn(X)\ when\ not\, repC \wedge not\, repP \end{array}$$

Alternatively, instead of the second clause, in this example, we can use a retract statement

$$retract\ jail(X) \leftarrow abn(X)\ when\ not\, repC \wedge not\, repP$$

Note that, in this example, since there is no other rule implying $jail$, retracting the rule is safely equivalent to retracting its conclusion.

These rules state that we are always supposed to update the current state with the rule $jail(X) \leftarrow abn(X)$ provided $repC$ and $repP$ hold true and that we are supposed to assert the opposite (or just retract this rule) provided $not\, repC$ and $not\, repP$ hold true. Such rules should be added to U_1.

Sequence of non-persistent update commands:

$$\begin{array}{ll} U_1 : assert\ repP & U_5 : assert\ not\, repP \\ \quad\;\; assert\ not\, repC & U_6 : assert\ event\ abn(ann) \\ U_2 : assert\ event\ abn(mary) & U_7 : assert\ not\, repC \\ U_3 : assert\ repC & U_8 : assert\ event\ abn(susan) \\ U_4 : assert\ event\ abn(kate) & \end{array}$$

Of course, in the meantime we could have a lot of trivial update events representing ticks of the clock, or any other irrelevant updates.

Example 3. Consider the spring-loaded suitcase with two latches situation [8], where the suitcase opens whenever both latches are up, and there is an action of toggling each latch. This situation can be represented in LUPS by adding to U_1 the persistent update commands:

$$\begin{array}{l} always\ open \leftarrow up(l1), up(l2) \\ always\ up(L)\ when\ not\, up(L), toggle(L) \\ always\ not\, up(L)\ when\ up(L), toggle(L) \end{array}$$

A concrete situation, where initially $l1$ is down, $l2$ is up and the suitcase is closed, plus the action of toggling $l1$, would be represented by the sequence of updates:

$$\begin{array}{l} U_1 = \{assert\ not\, up(l1), assert\ up(l2), assert\ not\, open\} \\ U_2 = \{assert\ event\ toggle(l1)\} \end{array}$$

Note how an initial state can easily be characterized by asserting appropriate rules in U_1. To model a situation where both $l1$ and $l2$ are toggled simultaneously, simply add *assert event toggle*$(l2)$ to U_2.

4 Semantics of LUPS

In this section we provide update programs with a meaning, by translating them into dynamic logic programs. The semantics of an update program is then determined by the semantics of the so obtained dynamic program. We recall that a dynamic program is a sequence $P_0 \oplus \ldots \oplus P_n$ (also denoted $\bigoplus \mathcal{P}$, where $\mathcal{P}$ is a set of generalized logic programs indexed by $1, \ldots, n$ and $P_0 = \{\}$). The notion of *"dynamic program update at a given state s"*, represented by $\bigoplus_s \mathcal{P}$, precisely characterizes a generalized logic program whose stable models correspond to the meaning of the dynamic program when queried at state s. If some literal or conjunction of literals ϕ holds in all stable models of $\bigoplus_s \mathcal{P}$, we write $\bigoplus_s \mathcal{P} \models_{sm} \phi$. Lack of space prevents us from giving here the complete definition of dynamic programs semantics. The reader is refered to [1] where such programs are defined, and their precise semantic characterization is provided.

The translation of an update program into a dynamic program is made by induction, starting from the empty program P_0, and for each update U_i, given the already built dynamic program $P_0 \oplus \ldots \oplus P_{i-1}$, determining the resulting program $P_0 \oplus \ldots \oplus P_{i-1} \oplus P_i$. To cope with persistent update commands we will further consider, associated with every dynamic program in the inductive construction, a set containing all currently active persistent commands, i.e. all those that were not cancelled until that point in the construction, from the time they were introduced. To be able to retract rules, we need to uniquely identify each such rule. This is achieved by augmenting the language of the resulting dynamic program with a new propositional variable "$rule(L \leftarrow L_1, \ldots, L_n)$" for every rule $L \leftarrow L_1, \ldots, L_n$ appearing in the original LUPS program[5].

[5] Note that, by definition, all such rules are ground and thus the new variable uniquely identifies the rule, where rule/1 is a reserved predicate.

Definition 4 (Translation into dynamic programs). *Let $\mathcal{U} = U_1 \otimes \ldots \otimes U_n$ be an update program. The corresponding dynamic program $\Upsilon(\mathcal{U}) = \mathcal{P} = P_0 \oplus \ldots \oplus P_n$ is obtained by the following inductive construction, using at each step i an auxiliary set of persistent commands PC_i:*

Base step: $P_0 = \{\}$ *with* $PC_0 = \{\}$.

Inductive step: Let $\mathcal{P}_i = P_0 \oplus \ldots \oplus P_i$ with the set of persistent commands PC_i be the translation of $\mathcal{U}_i = U_1 \otimes \ldots \otimes U_i$. The translation of $\mathcal{U}_{i+1} = U_1 \otimes \ldots \otimes U_{i+1}$ is $\mathcal{P}_{i+1} = P_0 \oplus \ldots \oplus P_{i+1}$ with the set of persistent commands PC_{i+1}, where:

$$
\begin{aligned}
&PC_{i+1} = PC_i \cup \\
&\cup\{assert\ R\ when\ C : always\ R\ when\ C \in U_{i+1}\} \\
&\cup\{assert\ event\ R\ when\ C : always\ event\ R\ when\ C \in U_{i+1}\} \\
&-\{assert\ [event]\ R\ when\ C : cancel\ R\ when\ D \in U_{i+1} \wedge \textstyle\bigoplus_i \mathcal{P}_i \models_{sm} D\} \\
&-\{assert\ [event]\ R\ when\ C : retract\ R\ when\ D \in U_{i+1} \wedge \textstyle\bigoplus_i \mathcal{P}_i \models_{sm} D\}
\end{aligned}
$$

$$NU_{i+1} = U_{i+1} \cup PC_{i+1}$$

$$
\begin{aligned}
&P_{i+1} = \{R,\ rule(R) : assert\ [event]\ R\ when\ C \in NU_{i+1} \wedge \textstyle\bigoplus_i \mathcal{P}_i \models_{sm} C\} \\
&\cup\ \{not\, rule(R) : retract\ [event]\ R\ when\ C \in NU_{i+1} \wedge \textstyle\bigoplus_i \mathcal{P}_i \models_{sm} C\} \\
&\cup\ \{not\, rule(R) : assert\ event\ R\ when\ C \in NU_i \wedge \textstyle\bigoplus_{i-1} \mathcal{P}_{i-1} \models_{sm} C\} \\
&\cup\ \{rule(R) : retract\ event\ R\ when\ C \in NU_i \wedge \textstyle\bigoplus_{i-1} \mathcal{P}_{i-1} \models_{sm} C, rule(R)\}
\end{aligned}
$$

where R denotes a generalized logic program rule, and C and D a conjunction of literals. $assert\ [event]\ R\ when\ C$ and $retract\ [event]\ R\ when\ C$ are used for notation convenience, and stand for either the assert or the assert-event command (resp. retract and retract-event). So, for example in the first line of the definition of P_{i+1}, R and $rule(R)$ must be added either if there exists a command $assert\ R\ when\ C$ or a command $assert\ event\ R\ when\ C$ obeying the conditions there.

In the inductive step, if $i = 0$ the last two lines are ommitted. In that case NU_i does not exist.

Definition 5 (LUPS semantics). *Let $\mathcal{U}$ be an update program. A query $holds(L_1, \ldots, L_n)$ at q is true in $\mathcal{U}$ iff $\bigoplus_q \Upsilon(\mathcal{U}) \models_{sm} L_1, \ldots, L_n$.*

From the results on dynamic programs in [1], it is clear that LUPS generalizes the language of updates of "revision programs" defined in [9]:

Theorem 1 (LUPS generalizes revision programs). *Let I be an interpretation and R a revision program. Let $\mathcal{U} = U_1 \otimes U_2$ be the update program where:*

$$
\begin{aligned}
&U_1 = \{assert\ A : A \in I\} \\
&U_2 = \{assert\ A \leftarrow B_1, \ldots, not\, B_n : in(A) \leftarrow in(B_1), \ldots, out(B_n) \in R\} \\
&\quad \cup\ \{assert\ not\, A \leftarrow B_1, \ldots, not\, B_n : out(A) \leftarrow in(B_1), \ldots, out(B_n) \in R\}
\end{aligned}
$$

Then, M is a stable model of $\Upsilon(\mathcal{U})$ iff M is an interpretation update of I by R in the sense of [9].

5 Translation into generalized logic programs

The previous section established semantics for LUPS. However, its definition is based on a translation into dynamic logic programs, and is not purely syntatic. Indeed, to obtain the translated dynamic program, one needs to compute, at each step of the inductive process, the consequences of the previous one.

In this section we present a translation of update programs and queries into normal logic programs written in a meta-language. The translation is purely syntactic, and is correct in the sense that a query holds in an update program iff the translation of the query belongs to all stable models of the translation of the update program. This translation also directly provides a mechanism for implementing update programs: with a pre-processor performing the translations, query answering is reduced to that of normal logic programs. Such a pre-processor and a meta-interpreter for answering queries have been implemented[6].

The translation presented here assumes the existence of a sequence of consecutive updates. Nevertheless, it is easy to see that the translation is modular (i.e. adding an extra update does not modify what has been already translated). Thus, in practice, the various updates can be iteratively translated, one at a time.

The translation uses a meta-language generated by the language of the update programs. For each objective atom A in the language of the update program, and each special propositional symbol $rule_{L \leftarrow Body}$ or $cancel_{L \leftarrow Body}$ (where these symbols are added to the language for each rule $L \leftarrow Body$ in the update program), the meta-language includes the following symbols: $A(s,t)$, $A^u(s,t)$, $\overline{A(s,t)}$, and $\overline{A^u(s,t)}$, where s and t range over the indexes of the update program. Intuitively, these new symbols mean, respectively: A is true at state s considering all states until t; A is true due to the update program at state s, considering all states until t; A is false at state s considering all states until t; *not* A is true due to the update program at state s, considering all states until t.

Intuitively, the first index argument added to atoms stands from the update state where the atom has been introduced. So, according to the transformation below, in non-persistent asserts the first argument of atoms in the head of rules is instanciated with the index of the update state where the rule was asserted. In persistent asserts, the argument ranges over the indexes where the rule should be asserted (i.e. all those greater than the state where the corresponding *always* command is).

The second index argument stands for the query state. Accordingly, when translating (non-event) asserts, the second argument of atoms in the head of rules ranges over all states greater than that where the rule was asserted. For event asserts, the second argument is instanciated with the index of the update state where the event was asserted. This is so in order to guarantee that the

[6] The system, running under XSB-Prolog, is available at:
`http://centria.di.fct.unl.pt/~jja/lups.p`
Rather than stable models, the well-founded semantics is used.

event is only true when queried in that state (it does not remain, by inertia, to subsequent query states).

Inertia rules are added to allow for the usage of rules asserted in states before the query one. Such rules say that one way to prove L at state s with query state t, is by proving L at state $s-1$ with the same query state (unless its complement is proven at state s, thus *blocking* the inertia of L).

Literals in the body of asserted rules are translated such that both arguments are instanciated with the query state. This guarantees that body literals are always evaluated in the query state. Literals in the when clause have both arguments instanciated with the state prior to that when the rule was asserted. This guarantees that those literals are always evaluated considering that state as the query state.

Definition 6 (Translation of update programs). *By the translation of an update program $\mathcal{U} = U_1 \otimes \ldots \otimes U_n$ in the language $\mathcal{L}$, $Tr(\mathcal{U})$, we mean the normal logic program consisting of the following rules, in the meta-language above:*

Default knowledge state rules. *For all objective atoms $A \in \mathcal{L}$, and $t \geq 0$:*

$$\overline{A(0,t)}$$

These rules state that in the initial state all objective atoms are false.

Update rules. *For all objective atoms $A \in \mathcal{L}$, and $s, t \geq 0$:*

$$A(s,t) \leftarrow A^u(s,t) \qquad\qquad \overline{A(s,t)} \leftarrow \overline{A^u(s,t)}$$

These update rules state that A is true (resp. false) at state s if A (resp. $not\, A$) is true due to the update program at state s.

Inertia rules. *For all objective atoms $A \in \mathcal{L}$, and $s, t > 0$:*

$$A(s,t) \leftarrow A(s-1,t), not\, \overline{A^u(s,t)} \qquad\qquad \overline{A(s,t)} \leftarrow \overline{A(s-1,t)}, not\, A^u(s,t)$$

Inertia rules say that A is true (resp. false) if it is true (resp. false) in the previous state and its complement is not true due to the update at s.

Translation of asserts. *For all update commands*

$$assert\ L \leftarrow B_1, \ldots, not\, B_k\ when\ C_1, \ldots, not\, C_m \in U_s$$

for any $1 \leq s \leq n$ and $t > s$:

$$rule^u_{L \leftarrow B_1, \ldots, not\, B_k}(s+1,t) \leftarrow C_1(s,s), \ldots, \overline{C_m(s,s)}$$

$$TL \leftarrow B_1(t,t), \ldots \overline{B_k(t,t)}, rule_{L \leftarrow B_1, \ldots, not\, B_k}(t,t), C_1(s,s), \ldots, \overline{C_m(s,s)}$$

where $TL = A^u(s+1,t)$ if L is an objective atom A, and $TL = \overline{A^u(s+1,t)}$ if L is a default atom $not\, A$. The rule $L \leftarrow B_1, \ldots, not\, B_k$ is added at state $s+1$ provided condition $C_1, \ldots, not\, C_m$ holds at state s (considering only states till s). It will remain true by inertia for all $t \geq s+1$ unless the literal $rule_{L \leftarrow B_1, \ldots, not\, B_k}(t,t)$ becomes false. Moreover, beginning at state $s+1$, $rule_{L \leftarrow B_1, \ldots, not\, B_k}(t,t)$ is true (and so remains by inertia).

Translation of retracts. *For all update commands*

$$retract\ L \leftarrow B_1, \ldots, not\, B_k\ when\ C_1, \ldots, not\, C_m \in U_s$$

for any $1 \leq s \leq n$ *and* $t > s$*:*

$$\overline{rule^u_{L \leftarrow B_1, \ldots, not\, B_k}(s+1,t)} \leftarrow C_1(s,s), \ldots, \overline{C_m(s,s)}$$

$$cancel^u_{L \leftarrow B_1, \ldots, not\, B_k}(s+1,t) \leftarrow C_1(s,s), \ldots, \overline{C_m(s,s)}$$

The rule $L \leftarrow B_1, \ldots, not\, B_k$ *is retracted at state* $s+1$ *provided that condition* $C_1, \ldots, not\, C_m$ *holds at state* s*. Retractions also cancel persistent update rules.*

Translation of persistent asserts. *For all update commands*

$$always\ L \leftarrow B_1, \ldots, not\, B_k\ when\ C_1, \ldots, not\, C_m \in U_s$$

for any $1 \leq s \leq n$ *and* $t, q+1 > s$*:*

$$\overline{cancel^u_{L \leftarrow B_1, \ldots, not\, B_k}(s+1,t)} \leftarrow$$

$$rule^u_{L \leftarrow B_1, \ldots, not\, B_k}(q+1,t) \quad \leftarrow C_1(q,q), \ldots, \overline{C_m(q,q)}, \overline{cancel_{L \leftarrow B_1, \ldots, not\, B_k}(q+1,q+1)}$$

$$TL \leftarrow B_1(t,t), \ldots \overline{B_k(t,t)}, rule_{L \leftarrow B_1, \ldots, not\, B_k}(t,t), \overline{cancel_{L \leftarrow B_1, \ldots, not\, B_k}(q+1,q+1)}, C_1(q,q), \ldots, \overline{C_m(q,q)}$$

where $TL = A^u(q+1,t)$ *if* L *is an objective atom* A*, and* $TL = \overline{A^u(q+1,t)}$ *if* L *is a default atom* $not\, A$*. The rule* $L \leftarrow B_1, \ldots, not\, B_k$ *is added to any state greater than* s*, provided condition* $C_1, \ldots, not\, C_m$ *holds at that state* s*, and will remain true by inertia for all* $t > q$*, unless retracted or cancelled.*

Translation of cancellation rules. *For all update commands*

$$cancel\ L \leftarrow B_1, \ldots, not\, B_k\ when\ C_1, \ldots, not\, C_m \in U_s$$

for any $1 \leq s \leq n$ *and* $t > s$*:*

$$cancel^u_{L \leftarrow B_1, \ldots, not\, B_k}(s+1,t) \leftarrow C_1(s,s), \ldots, \overline{C_m(s,s)}$$

The persistent update of rule $L \leftarrow B_1, \ldots, not\, B_k$ *is cancelled at state* $s+1$ *provided condition* $C_1, \ldots, not\, C_m$ *holds at state* s*.*

Translation of assert events. *For all update commands*

$$assert\ event\ L \leftarrow B_1, \ldots, not\, B_k\ when\ C_1, \ldots, not\, C_m \in U_s$$

for any $1 \leq s \leq n$*:*

$$TL \leftarrow B_1(s+1,s+1), \ldots \overline{B_k(s+1,s+1)}, C_1(s,s), \ldots, \overline{C_m(s,s)}$$

where $TL = A^u(s+1,s+1)$ *if* L *is objective atom* A*, and* $TL = \overline{A^u(s+1,s+1)}$ *if* L *is default atom* $not\, A$*. The rule* $L \leftarrow B_1, \ldots, not\, B_k$ *is added at state* $s+1$*, but does not remain true through inertia.*

Translation of retract events. *For all update commands*

$$retract\ event\ L \leftarrow B_1, \ldots, not\, B_k\ when\ C_1, \ldots, not\, C_m \in U_s$$

for any $1 \leq s \leq n$*:*

$$\overline{rule^u_{L \leftarrow B_1, \ldots, not\, B_k}(s+1, s+1)} \leftarrow C_1(s,s), \ldots, \overline{C_m(s,s)}$$

$$cancel^u_{L \leftarrow B_1, \ldots, not\, B_k}(s+1, s+1) \leftarrow C_1(s,s), \ldots, \overline{C_m(s,s)}$$

The rule $L \leftarrow B_1, \ldots, not\, B_k$ *is retracted at state* $s+1$ *under the conditions. The retraction does not remain true through inertia.*

Translation of persistent assert events. *For all update commands*

$$always\ event\ L \leftarrow B_1, \ldots, not\, B_k\ when\ C_1, \ldots, not\, C_m \in U_s$$

for any $1 \leq s \leq n$ *and* $t, q+1 > s$*:*

$$\overline{cancel^u_{L \leftarrow B_1, \ldots, not\, B_k}(s+1, t)} \leftarrow$$

$$TL \leftarrow B_1(q+1, q+1), \ldots \overline{B_k(q+1, q+1)}, \overline{cancel_{L \leftarrow B_1, \ldots, not\, B_k}(q+1, q+1)}, C_1(q,q), \ldots, \overline{C_m(q,q)}$$

where $TL = A^u(q+1, q+1)$ *if* L *is objective atom* A*, and* $TL = \overline{A^u(q+1, q+1)}$ *if* L *is default atom* $not\, A$*.*

The translation of update programs queries is similar to the translation of conditions in update commands:

Definition 7 (Translation of queries).
Let $Q = holds(B_1, \ldots, B_k, not\, C_1, \ldots, not\, C_m)$ *at* q *be a query to an update program* $\mathcal{U}$ *with language* $\mathcal{L}$*. The translation of* Q*,* $Tr(Q)$*, is:*

$$B_1(q,q), \ldots, B_k(q,q), \overline{C_1(q,q)}, \ldots, \overline{C_m(q,q)}$$

Theorem 2 (Correctness of the translation). *Let* $\mathcal{U}$ *be an update program. A query* Q *is true in* $\mathcal{U}$ *iff* $Tr(\mathcal{U}) \models_{sm} Tr(Q)$*.*

6 Comparisons

The language defined in this paper shares some similarities (and common motivations) with "Action Languages"[7]. Indeed, both LUPS and "Action Languages" specify how knowledge changes as an effect of actions or commands. Thus, a comparison with such languages would be in order. However, lack of space prevents us from carrying out such comparisons here. Some work in this area, as well as some more examples on the usage of LUPS, can be found in [3].

[7] For an overview of action languages see [6].

Let it be said that the most notable difference between LUPS and "Action Languages" is that while the latter deal only with updates of propositional knowledge states, LUPS updates knowledge states that consist of *knowledge rules*, i.e. the outcome of a LUPS update is not a simple set of propositional literals but rather a set of rules. Thus, inertia applies to knowledge rules, not just to propositional fluents. In addition, our approach makes it easier to specify so-called "static laws", and to deal with indirect effects of actions. Moreover, with LUPS, "static laws" may not necessarily be forever static: the laws that allow for indirect effects can themselves be subject to change. Another issue easily dealt with in LUPS is that of simultaneous actions. However, unlike most "Action Language", LUPs does not cater for nondeterministic actions. Indeed, in LUPS, updates are just linear sequences of sets of commands.

Acknowledgements

This work was partially supported by PRAXIS XXI project MENTAL, by FCT project ACROPOLE, by the National Science Foundation grant # IRI931-3061, and by a NATO grant for L. M. Pereira to visit Riverside.

References

1. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic logic programming. In A. Cohn and L. Schubert, editors, *KR'98*. Morgan Kaufmann, 1998.
2. J. J. Alferes and L. M. Pereira. Update-programs can update programs. In J. Dix, L. M. Pereira, and T. Przymusinski, editors, *NMELP'96*. Springer, 1996.
3. J. J. Alferes, L. M. Pereira, T. Przymusinski, H. Przymusinska, and P. Quaresma. Preliminary exploration on actions as updates. In *Joint Conference on Declarative Programming, AGP'99*, 1999. To appear.
4. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *ICLP'88*. MIT Press, 1988.
5. M. Gelfond and V. Lifschitz. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *KR'92*. Morgan-Kaufmann, 1992.
6. M. Gelfond and V. Lifschitz. Action languages. *Linkoping Electronic Articles in Computer and Information Science*, 3(16), 1998.
7. H. Katsuno and A. Mendelzon. On the difference between updating a knowledge base and revising it. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR'91*. Morgan Kaufmann, 1991.
8. F. Lin. Embracing causality in specifying the indirect effects of actions. In *IJCAI'95*, pages 1985–1991. Morgan Kaufmann, 1995.
9. V. Marek and M. Truszczynski. Revision specifications by means of programs. In C. MacNish, D. Pearce, and L. M. Pereira, editors, *JELIA '94*. Springer, 1994.
10. T. Przymusinski and H. Turner. Update by means of inference rules. In V. Marek, A. Nerode, and M. Truszczynski, editors, *LPNMR'95*. Springer, 1995.
11. M. Winslett. Reasoning about action using a possible models approach. In V. Marek, A. Nerode, and M. Truszczynski, editors, *AAAI'88*, 1988.