LUPS – a language for updating logic programs [⋆]

José Júlio Alferes [a] Luís Moniz Pereira [a] Halina Przymusinska [b,c] and Teodor C. Przymusinski [c]

[a] *Centro de Inteligência Artificial, FCT/UNL, P-2825-114 Caparica, Portugal*

[b] *Computer Science, California State Polytechnic Univ. Pomona, CA 91768, USA*

[c] *Computer Science, Univ. of California Riverside, CA 92521, USA*

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 *dynamic logic programming*. There, they studied and defined the declarative and operational semantics of sequences of logic programs (or dynamic logic programs). Each program in the sequence contains knowledge about some given state, where different states may, for example, represent different time periods or different sets of priorities.

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, given the sequence, determining what should follow, it does not provide a good practical language for the specification of the sequence of updates which may be conditional on the intervening states.

Here we define the language LUPS – "Language for dynamic updates" – designed for specifying changes to logic programs. Given an initial knowledge base (as 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 by the semantics of the dynamic logic program generated by those actions. Additionally, we provide a translation of the sequence of update statements sets into a single logic program written in a meta-language, in such a way that the stable models of the resulting program correspond to the previously defined declarative semantics. Finally, we exhibit the usage of LUPS in several application domains.

[⋆] Extended version of the paper presented at LPNMR'99.

1 Introduction

Several authors [21,22,3] 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 [24,15], 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], the approach of [24,15], while adequate for the purpose of updating theories in classical propositional logic (for which it was targeted), when applied to non-monotonic theories 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 [3] 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:*

$$free \leftarrow not\, jail$$
$$jail \leftarrow abortion$$

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 so $free$ 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, as originally pointed out in [18]. 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

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

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 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 program more, $P_0 \oplus \ldots \oplus P_n \oplus P_{n+1}$, that corresponds to the knowledge resulting from the previous sequence after performing all the new simultaneous commands. A program in LUPS is a sequence of such sentences.

Given a LUPS program, 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 furthermore describe a translation of any LUPS program into a single generalized logic program, whose stable models exactly correspond to the semantics of the original LUPS program.

In section 6, we argue that the new language LUPS represents a natural, powerful and expressive tool for representing dynamically changing knowledge. We do so by demonstrating the applicability of LUPS to several broad knowledge representation domains. Finally, in section 7, we make some concluding remarks and discuss future work.

2 Object language

In order to represent *negative* information in logic programs and their updates, we require more general logic programs, allowing for default negation *not* A not only in the premises of rules but also in their heads. In updates a *not* A head means atom A is deleted if the body holds (cf. [2]). Deleting A means that A is no longer true, not necessarily that it is false. When some form of closed world assumption (CWA) is adopted as well, then this deletion causes A to be false. In the updates setting, as we will make clear in Section 4, the CWA must be explicitly encoded from the start, by making all *not* A false in the initial program being updated. That is, the two concepts, deletion and CWA, are orthogonal and must be separately incorporated. Thus, in general, using logic programs extended with explicit negation [10] wouldn't be adequate, because explicitly negated heads express the negated is false, not just deleted.

In the stable models [19,14] and well-founded semantics [6] of single generalized programs, the CWA is adopted *ab initio*, and default negation in the heads is conflated with non-provability because there is no updating and thus no deletion. Note however that, unlike with single generalized programs (cf. [14]), in updates the head *not*s cannot be moved freely into the body, to obtain simple denials: there is inescapable pragmatic information in specifying exactly which *not* literal figures in the head, namely the one being deleted when the body holds true. It is not indifferent that any other (positive) body literal in the denial would be moved to the head. Example 9 shows just that.

In this section we recall the semantics of *single generalized logic programs*, as defined in [1,2]. The class of generalized logic programs can be viewed as a special case of yet broader classes of programs, introduced earlier in [14] and in [19]. As shown in [2], their semantics coincides with the stable models semantics [9] for the special case of normal programs. Moreover, the semantics also coincides with the one in [19] (and, consequently, with the one in [14]) when the latter is restricted to the language of generalized programs.

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 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 2 (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:

$$\begin{aligned} M^+ &= \{A \in \mathcal{K} : A \in M\} \qquad\qquad \text{and} \\ M^- &= \{not\, A : not\, A \in M\} = \{\, not\, A : A \notin M\} \end{aligned}$$

By a (2-valued) *model* M of a generalized logic program we mean a (2-valued) interpretation that satisfies all of its clauses. As usual, a clause is satisfied in an interpretation if whenever its body belongs to the interpretation its head does too.

Definition 3 (Stable models of generalized logic programs) *An 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:*

$$M = \{L : L \text{ is a literal and } P \cup M^- \vdash L\}$$

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:

$$\textbf{holds } B_1, \ldots, B_k, not\, C_1, \ldots, not\, C_m \textbf{ at } q?$$

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:

$$\textbf{assert } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \qquad (1)$$

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 successor 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 preceded with the keyword **event**, indicating that the added rule is non-inertial. Assert commands are thus of the form (1) or of the form[2]:

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

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 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:

$$\textbf{assert } jail \leftarrow abortion \textbf{ when } rep\_congress, rep\_president$$

or

$$\textbf{assert } wake\_up \textbf{ when } alarm\_sounds$$

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:

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

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:

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

and:

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

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

For cancelling persistent update commands, we use:

$$\textbf{cancel } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{5}$$

The first two statements mean that, in addition to any new set of arriving update commands, the persistent update command keep executing with them too. In the first case without, and in the second one with the **event** keyword. The third statement cancels execution of this persistent update, once the conditions for cancellation 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 modeled.

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

$$\textbf{retract } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{6}$$

and:

$$\textbf{retract event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{7}$$

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 (in (6)), or just retracted temporarily in the next state (in (7)). The latter represents a 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 4 (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 (6).*

Example 5 *Consider the following 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, a Democrat, is elected President; (6) Ann performs abortion; (7) Gore, a Democrat, is elected President and Democratic Congress is in place (year 2000?); (8) Susan performs abortion.

The specification in LUPS would be[3] *:*

Persistent update commands:

$$\textbf{\textit{always}}\ jail(X) \leftarrow abt(X)\ \textbf{\textit{when}}\ repC \wedge repP$$

$$\textbf{\textit{always}}\ not\, jail(X) \leftarrow abt(X)\ \textbf{\textit{when}}\ not\, repC \wedge not\, repP$$

Alternatively, instead of the second clause, in this example, we could have used a retract statement

$$\textbf{\textit{retract}}\ jail(X) \leftarrow abt(X)\ \textbf{\textit{when}}\ not\, repC \wedge not\, repP$$

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

The above rules state that we are always supposed to update the current state with the rule $jail(X) \leftarrow abt(X)$ *provided* $repC$ *and* $repP$ *hold true and that we are supposed to assert the opposite provided* $not\, repC$ *and* $not\, repP$ *hold true. Such persistent update commands should be added to* U_1.

Sequence of non-persistent update commands:

[3] Where the rules with variables simply stand, as usual, for all the ground rules that result from replacing the variables by all the ground terms in the language.

U_1 : ***assert*** $repP$

assert $not\, repC$

U_2 : ***assert event*** $abt(mary)$

U_3 : ***assert*** $repC$

U_4 : ***assert event*** $abt(kate)$

U_5 : ***assert*** $not\, repP$

U_6 : ***assert event*** $abt(ann)$

U_7 : ***assert*** $not\, repC$

U_8 : ***assert event*** $abt(susan)$

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

Intuitively, the results of this LUPS program should be the following:

- *Initially, there is no rule about going to jail or not whenever an abortion is performed. The rules asserted in U_1 do not change this.*
- *When, in U_2, Mary opts for abortion, since there is no rule concerning it, $jail(mary)$ does not become true, and so should be false by default.*
- *With the rule asserted in U_3, both repC and repP become true (repP is true because it was true before, and remains so by inertia). Thus, by the first persistent command, the rule $jail(X) \leftarrow abt(X)$ must be asserted.*
- *When Kate undergoes an abortion (in U_4) the above rule is in force, and so $jail(kate)$ becomes true.*
- *With the fact asserted in U_5, none of the **when**-conditions of the two persistent commands hold. So, none of the two rules, $jail(X) \leftarrow abt(X)$ and $not\, jail(X) \leftarrow abt(X)$, are to be asserted here. However, note that the rule asserted in U_3 remains true by inertia.*
- *Now Ann chooses abort and, since the rule asserted in U_3 holds by inertia, $jail(ann)$ becomes true.*
- *When $not\, repC$ is asserted, in U_7, the **when**-conditions of the second persistent command become true and, consequently, $not\, jail(X) \leftarrow abt(X)$ is asserted.*
- *When Susan subjects herself to an abortion, the rule asserted in U_7 is in force by inertia. Moreover this rule, being the more recent, is used to "reject" the rule introduced in U_3. Accordingly, $not\, jail(susan)$ is true in this state.*

We come back to this example after the definition of the declarative semantics for LUPS, and then show that these intuitive results are indeed obtained.

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.

For clarity, we start by briefly describing the language and semantics of dynamic logic programs of [1]. 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 $P_0 = \{\}$). Intuitively such a sequence may be viewed as the result of, starting with program P_1, updating it with program P_2, ..., and updating it with program P_n. In such a view, dynamic logic programs are to be used in knowledge bases that evolve[4] . New rules (coming from new, or newly acquired, knowledge) can be added at the end of the sequence, bothering not whether they conflict with previous knowledge. The role of dynamic programming is to ensure that these newly added rules are in force, and that previous rules are still valid (by inertia) as far as possible, i.e. that they are kept for as long as they do not conflict with more recent ones.

The semantics of dynamic logic programs is defined according to the rationale above. Given a model M of the last program P_n, start by removing all the rules from previous programs whose head is the complement of some later rule with true body in M (i.e. by removing all rules which conflict with later ones). All others persist through, by inertia. Then, as for the stable models of a single generalized program, add facts $not\, A$ for all atoms A which have no rule at all with true body in M, and compute the least model. If M is a fixpoint of this construction, M is a stable model of the sequence up to P_n.

Definition 6 (Rejected rules) *Let $\bigoplus\{P_i : i \in S\}$ be a dynamic logic program, let $s \in S$, and let M be a model of P_s. Then:*

$$Reject_s(M) = \{L_0 \leftarrow Body \in P_i \mid \exists\; not\, L_0 \leftarrow Body' \in P_j,\, i < j \leq s \,\wedge\, M \models Body'\}$$

where $not\, L_0$ denotes the complement of the literal L_0 (i.e. denotes $not\, A$ if L_0 is an atom A, and denotes A if L_0 is a default literal $not\, A$) , and both $Body$ and $Body'$ are conjunctions of literals.

Note that, according to this definition, even rules with false body might be rejected. In fact, the condition for rejection does not impose $Body$ to be true in M. However, as we remark below, the rejection of rules with false body does not influence the resulting semantics. So, to simplify the definition, we

[4] Instead of viewing programs in the sequence as different stages of knowledge in the linear evolution of the knowledge base, these can also be viewed as different time points in possible future evolutions of the knowledge, or even as knowledge of ever more specific objects organized in a hierarchy (see [5] for more on this view). Since our goal here is simply to recap dynamic programming for the purpose of better understanding LUPS, we do not develop these other views herein.

do not impose $M \models Body$.

Definition 7 (Default rules) *Let M be a model of a generalized logic program P. Then:*

$$Default(P,M) = \{not\, A \mid \nexists\ A \leftarrow L1, \ldots, L_n \in P : M \models L1, \ldots, L_n\}$$

To allow for querying a dynamic program at any state s, the definition of stable model is parameterized by the state:

Definition 8 (Stable Models of a DLP at state s) *Let $\bigoplus \mathcal{P} = \bigoplus \{P_i : i \in S\}$ be a dynamic logic program, let $s \in S$, and let $\mathcal{U} = \bigcup_{i \leq s} P_i$. A model M of P_s is a stable model of $\bigoplus \mathcal{P}$ at state s iff:*

$$M = least([\mathcal{U} - Reject_s(M)] \cup Default(\mathcal{U}, M))$$

If some literal or conjunction of literals ϕ holds in all stable models of $\bigoplus \mathcal{P}$ at state s, we write $\bigoplus_s \mathcal{P} \models_{sm} \phi$.

Mark here that, as noted after Definition 6, the rejection of rules with false body in M do not affect the resulting semantics. In fact, adding or removing such rules does not affect the result of *least*.

Example 9 *Consider the DLP $P_1 \oplus P_2$, where P_1 and P_2 are:*

$$\begin{array}{llll} P_1: & c \leftarrow & \qquad\qquad P_2: & not\, a \leftarrow c \\ & a \leftarrow not\, b & & \end{array}$$

The only stable model at P_2 is $M = \{c, not\, a, not\, b\}$. In fact, $Default(P_1 \cup P_2, M) = \{not\, b\}$, $Reject_2(M) = \{a \leftarrow not\, b\}$, and:

$$M = \{c, not\, a, not\, b\} = least((P_1 \cup P_2 - \{a \leftarrow not\, b\}) \cup \{not\, b\})$$

Note here that, as mentioned in Section 2, in DLPs the head not 's cannot be moved freely into the body, to obtain denials. The rule in P_2 includes the pragmatic information that $not\, a$ is to be deleted if c is true, information that would be lost with the denial. Intuitively that rule makes a different statement from that of the rule $not\, c \leftarrow a$, which however yields the same denial. And this difference is reflected by the defintion of stable models for DLPs. In fact, if the rule in P_2 is replaced by this other one, the only stable model at P_2 would be $\{not\, c, a, not\, b\}$ instead.

The reader can check that if the rule in P_2 is replaced by $u \leftarrow a, c, not\, u$ (which, under the stable models semantics, is equivalent to the denial) the results are *also different from the ones above: with this rule instead, there is no stable model at P_2.*

In [2] a transformational semantics for dynamic programs is presented. According to this equivalent definition, a sequence of programs is translated into a single generalized program (with one new argument added to all predicates) whose stable models are in one-to-one correspondence with the stable models of the dynamic program. This transformational semantics, here presented in Appendix A, is the basis of an existing implementation of dynamic logic programming.

The translation of an update program into a dynamic program is obtainable 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, up to 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].

Definition 10 (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:*

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

[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.

$$-\{\textbf{\textit{assert [event]}}\ R\ \textbf{\textit{when}}\ C : \textbf{\textit{retract}}\ R\ \textbf{\textit{when}}\ D \in U_{i+1} \land \bigoplus_i \mathcal{P}_i \models_{sm} D\}$$

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

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

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 notational 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 if there exists either a command* ***assert*** *R* ***when*** *C or a command* ***assert event*** *R* ***when*** *C obeying the conditions C there.*

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

Definition 11 (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 $\bigoplus_q \Upsilon(\mathcal{U}) \models_{sm} L_1, \ldots, L_n$.*

Example 12 *Recall the LUPS program of Example 5, which consisted in the following two persistent update commands (added to U_1):*

$$\textbf{\textit{always}}\ jail(X) \leftarrow abt(X)\ \textbf{\textit{when}}\ repC \land repP$$

$$\textbf{\textit{always}}\ not\,jail(X) \leftarrow abt(X)\ \textbf{\textit{when}}\ not\,repC \land not\,repP$$

plus the sequence of non-persistent commands:

U_1 : ***assert*** $repP$
assert $not\,repC$
U_2 : ***assert event*** $abt(mary)$
U_3 : ***assert*** $repC$
U_4 : ***assert event*** $abt(kate)$
U_5 : ***assert*** $not\,repP$
U_6 : ***assert event*** $abt(ann)$
U_7 : ***assert*** $not\,repC$
U_8 : ***assert event*** $abt(susan)$

It is easy to check that

$$\Upsilon(U_1) = \{\} \oplus \{repP \leftarrow rule_1;\ not\,repC \leftarrow rule_2;\ rule_1 \leftarrow;\ rule_2 \leftarrow\}$$ [6]

Thus, according to the DLP semantics, except for $repP$, everything else is false by default at U_1.

In U_2, $\Upsilon(U_1 \otimes U_2) = \Upsilon(U_1) \oplus \{abt(mary) \leftarrow rule_3;\ rule_3 \leftarrow\}$. Thus, at U_2 $repP$ and $abt(mary)$ are true and everything else false by default.

At state U_3, $repC$ is added, and the rule added via the ***assert event*** *of U_2 must be retracted. Accordingly:*

$$\Upsilon(U_1 \otimes U_2 \otimes U_3) = \Upsilon(U_2) \oplus \{repC \leftarrow rule_4;\ rule_4 \leftarrow;\ not\, rule_3 \leftarrow\}$$

and $repP$ and $repC$ are true at U_3.

Now, since both $repP$ and $repC$ are true at U_3, then by the first persistent command, the rule $jail(X) \leftarrow abt(X)$ must be added, and so:

$$\begin{aligned}\Upsilon(U_1 \otimes \ldots \otimes U_4) = \Upsilon(U_3) \oplus \{\ & jail(X) \leftarrow abt(X), rule_5(X);\ rule_5 \leftarrow;\\ & abt(kate) \leftarrow rule_6;\ rule_6 \leftarrow\}\end{aligned}$$

Thus, at state U_3, $jail(kate)$ is true.

The addition of $not\, repC$ and of $abt(ann)$ in states U_5 and U_6, respectively, yields:

$$\begin{aligned}\Upsilon(U_1 \otimes \ldots \otimes U_6) = \Upsilon(U_4) & \oplus \{not\, repP \leftarrow rule_7;\ rule_7 \leftarrow;\ not\, rule_6 \leftarrow\}\\ & \oplus \{abt(ann) \leftarrow rule_8;\ rule_8 \leftarrow\}\end{aligned}$$

According to the semantics of this DLP program, $jail(ann)$ is true at state U_6.

After the addition of $not\, repC$, in U_7, both $repC$ and $repP$ are false, and so the rule $not\, jail(X) \leftarrow abt(X)$ is added. Thus:

$$\begin{aligned}\Upsilon(U_1 \otimes \ldots \otimes U_8) = \Upsilon(U_6) & \oplus \{\ not\, repC \leftarrow rule_9;\ rule_9 \leftarrow;\ not\, rule_8\}\\ & \oplus \{\ not\, jail(X) \leftarrow abt(X), rule_{10};\ rule_{10} \leftarrow;\\ & \qquad abt(susan) \leftarrow rule_{11};\ rule_{11} \leftarrow\}\end{aligned}$$

The reader can check that the semantics of this DLP program entails that $not\, jail(susan)$ is true at state U_8. In particular note how, in the only stable model at state U_8, the rule $jail(X) \leftarrow abt(X)$, added in U_4, is rejected.

[6] To simplify notation, instead of using the whole rule as a quoted atom as argument of the predicate *rule*, we index the rule names in this example with unique numbers.

From the results on dynamic programs in [1], it is clear that LUPS generalizes the language of updates of "revision programs" defined in [21]:

Proposition 13 (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:*

$$U_1 = \{\boldsymbol{assert}\ A : A \in I\}$$

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

Then, M is a stable model of $\Upsilon(\mathcal{U})$ iff M is an interpretation update of I by R in the sense of [21].

PROOF. This proposition follows easily from theorem 5.1 of [1] (whose proof may be found in [2]). The afore mentioned theorem states that the stable models of the dynamic logic program $P_1 \oplus P_2$ exactly correspond to the interpretation updates of I by R, where P_1 is just the set of facts in I, and P_2 includes the rules:

$$\begin{array}{lll} A \leftarrow B_1, \ldots, not\, B_n & \text{for every} & in(A) \leftarrow in(B_1), \ldots, out(B_n) \in R \\ not\, A \leftarrow B_1, \ldots, not\, B_n & \text{for every} & out(A) \leftarrow in(B_1), \ldots, out(B_n) \in R \end{array}$$

5 Translation into generalized logic programs

The previous section established the semantics for LUPS. However, its definition is based on a translation into dynamic logic programs, and is not purely syntactic. 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 holds in 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 over normal logic programs[7].

[7] See Section 7 for more information about such an implementation.

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.

Note that the translation presented below is not necessary for understanding the example applications shown in the next section. Thus, a reader less interested in the implementation of LUPS, and more interested in its applications, can skip this section without loss of continuity.

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[8]), 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 instantiated 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 instantiated with the index of the update state where the event was asserted. This is so in order to guarantee that the 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 to prove L at state $s-1$ with the same query state (unless its

[8] Recall that, according to Definition 2, programs are possibly infinite sets of propositional rules. Thus, the special propositional symbols are added for every such propositional rule.

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 instantiated with the query state. This guarantees that body literals are always evaluated in the query state. Literals in the **when** clause have both arguments instantiated 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 14 (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 $s, t \geq 0$:*

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

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 $s, t > 0$:*

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

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*

$$\textbf{\textit{assert}}\ L \leftarrow B_1, \ldots, not\, B_k\ \textbf{\textit{when}}\ C_1, \ldots, not\, C_m \in U_s$$

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*

$$\textbf{\textit{retract}}\ L \leftarrow B_1, \ldots, not\, B_k\ \textbf{\textit{when}}\ C_1, \ldots, not\, C_m \in U_s$$

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*

$$\textbf{\textit{always}}\ L \leftarrow B_1, \ldots, not\, B_k\ \textbf{\textit{when}}\ C_1, \ldots, not\, C_m \in U_s$$

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*

$$\textbf{\textit{cancel}}\ L \leftarrow B_1,\ldots,not\, B_k\ \textbf{\textit{when}}\ C_1,\ldots,not\, C_m \in U_s$$

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*

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

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*

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

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 named conditions. The retraction does not remain true through inertia.*

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

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

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

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

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

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 that of conditions in update commands:

Definition 15 (Translation of queries)
Let $Q = \textbf{\textit{holds}}\ B_1, \ldots, B_k, not\, C_1, \ldots, not\, C_m\ \textbf{at}\ q$ *be a query to an update program* $\mathcal{U}$ *in the language* $\mathcal{L}$*. The translation of* Q*,* $Tr(Q)$*, is the conjunction of literals in the meta-language:*

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

Theorem 16 (Correctness of the translation) *Let* $\mathcal{U}$ *be an update program. A query* Q *is true in* $\mathcal{U}$ *iff* $Tr(\mathcal{U}) \models_{sm} Tr(Q)$*.*

PROOF. See Appendix A.

6 Application domains

In this section we discuss and illustrate with examples the applicability of the language LUPS to several broad knowledge representation domains. The selected domains include: *active databases, theory of actions, legal reasoning, and software specification.* They are not intended by any means to constitute an exhaustive list of potential application domains but just to serve as *sample representatives.* For each of the selected application domains we present an illustrative example, each of which has been run and tested under our implementation of LUPS. Additional examples of application of LUPS can be found in [4].

6.1 Active Knowledge Bases

Persistent update laws allow us to handle not only knowledge states that dynamically change due to newly received updates, but they enable us also to model the much more involved case of *self-updating* or *active knowledge bases,* which undergo change even though no truly new updates occurred. For example, the watch's hands move whether or not new updates are received. Since the high-level language of updates LUPS defined above has a built-in capability to define persistent updates, it permits us to model active knowledge bases. The following example illustrates the language's ability to handle persistent updates.

Example 17 (Timers) *Timers are started by the action $on\_for(N)$ which means the timer will be on for the next N states. This notion can be captured by three persistent rules added to U_1:*

$$\textbf{\textit{always event}}\ on\_for(M)\ \textbf{\textit{when}}\ on\_for(N), N > 0, M = N - 1$$

*With this rule the timer is on for M states (or clock ticks) if it was on for $N = M + 1$ states in the previous state. Timers are on if they are $on\_for(N)$, where $N > 0$: **always** $on \leftarrow on\_for(N), N > 0$, and they are turned off when $N = 0$: **always** not $on \leftarrow on\_for(0)$. Consider the update command asserted at some state U_n: U_n : **assert event** $on\_for(2)$. As intended, on holds in the state $n + 1, n + 2$ but not on holds in the state $n + 3$.*

The problem of building, querying and modifying *active databases* is studied by many researchers in the database community, and is considered to be an important research topic. We believe that our approach is likely to have a major impact on the ongoing research in this area by helping to precisely define both the declarative and the procedural meaning of the notion of active database.

6.2 Reasoning about Actions

An exceptionally successful effort has been made lately in the area of *reasoning about actions*. Beginning with the seminal paper by Gelfond and Lifschitz [11], introducing a declarative language for talking about effects of actions (action language $\mathcal{A}$), through the more recent paper of Giunchiglia and Lifschitz [13] setting forth an enhanced version of the language (the so called language $\mathcal{C}$), a number of very interesting results have been obtained by several researchers significantly moving forward our understanding of actions, causality and effects of actions (see the survey paper [12] for more details on action languages). These recent advances also significantly influenced our own work on dynamic knowledge updates.

The theory of actions is very closely related to knowledge updates. An action taking place at a specific moment of time may cause an effect in the form of a change of the status of some fluent. For example, an action of stepping on a sharp nail may result in severe pain. The occurrence of pain can therefore be viewed as a simple (atomic) knowledge update triggered by a given action. Similarly, a set of parallel actions can be viewed as triggering (causing) parallel atomic updates. The following *suitcase* example illustrates how LUPS can be used to handle parallel updates.

Example 18 (Suitcase) *There is a suitcase with two latches which opens whenever both latches are up, and there is an action of toggling applicable to each latch [20]. This situation is represented by the three persistent rules:*

$$\begin{aligned}
&\boldsymbol{always}\ open \leftarrow up(l1), up(l2)\\
&\boldsymbol{always}\ up(L)\ \boldsymbol{when}\ not\ up(L), toggle(L)\\
&\boldsymbol{always}\ not\ up(L)\ \boldsymbol{when}\ up(L), toggle(L)
\end{aligned}$$

In the initial situation $l1$ *is down,* $l2$ *is up, and the suitcase is closed:*

$$U_1 = \{\boldsymbol{assert}\ not\ up(l1), \boldsymbol{assert}\ up(l2), \boldsymbol{assert}\ not\ open\}$$

Suppose there are now two simultaneous toggling actions:

$$U_2 = \{\boldsymbol{assert}\ \boldsymbol{event}\ toggle(l1), \boldsymbol{assert}\ \boldsymbol{event}\ toggle(l2)\}$$

and afterwards another $l2$ toggling action: $U_3 = \{$***assert event*** $toggle(l2)\}$*. In the knowledge state* 2 *we will have* $up(l1), not\ up(l2)$ *and the suitcase is not open. Only after* U_3 *will latch* $l2$ *be up and the suitcase open.*

However, there are also major differences between dynamic updates of knowledge and theories of actions. While in our approach we want to be able to update one knowledge base by an arbitrary set of rules that constitutes the updating knowledge base, action languages deal only with updates of propositional knowledge states. In other words, action languages are limited to purely atomic assertions and retractions, and thus deal exclusively with purely extensional (or relational) knowledge bases. Our approach further allows us to model self-updating or active knowledge bases that are capable of undergoing change without any outside triggers at all. As a result, from a purely syntactic point of view, LUPS is strictly more expressive than action languages $\mathcal{A}$ or $\mathcal{C}$.

At the semantic level, however, the situation is not so simple. The main motivation behind the introduction of the language $\mathcal{C}$ was to be able to express the notion of causality. This is a very different motivation from the motivation that we used when defining the semantics of updated knowledge bases. Here the main principle was to inherit as much information as possible from the previous knowledge state while changing only those rules that truly have to be affected by the given update(s). As a result, one can easily see that, even in simple cases, the semantics of knowledge updates and that of action languages often differ.

In spite of these differences, the strong similarities between the two approaches clearly justify a serious effort to investigate the exact nature of the close relationship between the two research areas and between the respective families of languages, their syntax and semantics. Hopefully, we will be able to bridge the gap between these two intriguing and closely related research areas.

6.3 Legal Reasoning

Robert Kowalski and his collaborators did a truly outstanding research work on using logic programming as a *language for legal reasoning* (see e.g. [16]). However logic programming itself lacks any mechanism for expressing dynamic changes in the law due to revisions of the law or due to new legislation. Dynamic knowledge representation allows us to handle such changes in a very natural way by augmenting the knowledge base only with the newly added or revised data, and automatically obtaining the updated information as a result. We illustrate this capability of LUPS on the following simple example.

Example 19 (Conscientious objector) *Consider a situation where someone is conscripted if he is draftable and healthy. Moreover a person is draftable* *when he attains a specific age. However, after some time, the law changes and a person is no longer conscripted if he is indeed a conscientious objector:*

$$
\begin{aligned}
U_1 &: \textbf{\textit{always}}\ draftable(X)\ \textbf{\textit{when}}\ of\_age(X)\\
&\quad \textbf{\textit{assert}}\ conscripted(X) \leftarrow draftable(X), healthy(X)\\
U_2 &: \textbf{\textit{assert}}\ healthy(a). \quad \textbf{\textit{assert}}\ healthy(b). \quad \textbf{\textit{assert}}\ of\_age(b).\\
&\quad \textbf{\textit{assert}}\ consc\_objector(a).\ \textbf{\textit{assert}}\ consc\_objector(b)\\
U_3 &: \textbf{\textit{assert}}\ of\_age(a)\\
U_4 &: \textbf{\textit{assert}}\ not\ conscripted(X) \leftarrow consc\_objector(X)
\end{aligned}
$$

In state 3*, b is subject to conscription but after the last assertion his situation changes. On the other hand, a is never conscripted.*

In addition to providing automatic updating, LUPS allows us to keep the entire history of past changes and to query the knowledge base at any given time in the past. The ability to keep track of all past changes in the law is a feature of crucial importance in this domain. We expect, therefore, that by using LUPS as a language for representation and reasoning about legal knowledge we may be able to significantly improve upon the work supported on standard logic programming.

6.4 Software Specifications

One of the most important problems in software engineering is that of choosing a suitable software specification language. The following are among the key desired properties of such a language:

(1) Possibility of a concise representation of statements of natural language, commonly used in informal descriptions of various domains.
(2) Availability of query answering systems which allow rapid prototyping.
(3) Existence of a well developed and mathematically precise semantics of the language.
(4) Ability to express conditions that change dynamically.
(5) Ability to handle inconsistencies stemming from specification revisions.

It has been argued in several papers (see e.g. [17,7]) that the language of logic programming is a good potential candidate for the language of software specifications. While logic programming clearly possesses the first three properties, it lacks simple and natural ways of expressing conditions that change dynamically and the ability to handle inconsistencies stemming from specification revisions. The last problem is called *elaboration tolerance* and requires that small modifications of informal specifications result in localized and simple modifications of their formal counterparts. Dynamic knowledge representation based on generalized logic programs extends logic programming exactly with these two missing dynamic update features. Moreover, small informal specification revisions require equally small modifications of the formal specification, while all the remaining information is preserved by inertia. The following banking example illustrates the above claims.

Example 20 (Banking transactions) *Consider a software specification for performing banking transactions. Account balances are modeled by the predicate* $balance(AccountNo, Balance)$. *Predicates* $deposit(AccountNo, Amount)$ *and* $withdrawal(AccountNo, Amount)$ *are used to represent, by means of events, the actions of depositing and withdrawing money into and out of an account, respectively. A withdrawal can only be accomplished if the account has a sufficient balance. This simplified description can easily be modeled in LUPS by* U_1*:*

always $balance(Ac, OB + Up)$ ***when*** $updateBal(Ac, Up), balance(Ac, OB)$
always $not\ balance(Ac, OB)$ ***when*** $updateBal(Ac, NB), balance(Ac, OB)$
assert $updateBal(Ac, -X) \leftarrow withdrawal(Ac, X), balance(Ac, O), O > X$
assert $updateBal(Ac, X) \leftarrow deposit(Ac, X)$

The first two rules state how to update the balance of an account, given any updateBal occurrence: when some updateBal occurs for account Ac, the balance of Ac must be changed by Up. This is accomplished by simultaneosly adding the fact for the new balance (in the first command) and deleting the fact with the old balance of Ac (in the second command). With the last two rules, deposits and withdrawals are carried out, by causing updateBal.

An initial situation can be imposed via assert commands. Deposits and withdrawals can be stipulated by asserting events of $deposit/2$ *and* $withdrawal/2$*. E.g.:*

$$U_2 : \{\textbf{\textit{assert}}\ balance(1, 0), \textbf{\textit{assert}}\ balance(2, 50)\}$$

$$U_3 : \{\textbf{\textit{assert event}}\ deposit(1, 40), \textbf{\textit{assert event}}\ withdrawal(2, 10)\}$$

causes the balance of both accounts 1 and 2 to be 40, after state 3.

Now consider the following sequence of informal specification revisions. Deposits under 50 *are no longer allowed; VIP accounts may have a negative balance up to the limit specified for the account; account #1 is a VIP account with the overdraft limit of* 200*; deposits under* 50 *are allowed for accounts with negative balances. These can in turn be modeled by the sequence:*

U_4 : ***assert*** *not updateBal*$(Ac, X) \leftarrow$ *deposit*$(Ac, X), X < 50$

U_5 : ***assert*** *updateBal*$(Ac, -X) \leftarrow$ *vip*(Ac, L), *withdrawal*(Ac, X),
balance$(Ac, B), B + L \geq X$

U_6 : ***assert*** *vip*$(1, 200)$

U_7 : ***assert*** *updateBal*$(Ac, X) \leftarrow$ *deposit*(Ac, X), *balance*$(Ac, B), B < 0$

This shows dynamic knowledge representation constitutes a powerful tool for software specifications that will prove helpful in the difficult task of building reliable and provably correct software.

7 Conclusions and future work

We have presented LUPS, a language for specifying dynamic updates in non-monotonic knowledge bases. Knowledge bases are represented by generalized logic programs allowing default negation in rule heads. We provided a declarative semantics for the language, by translating LUPS programs into sequences of logic programs, whose semantics is determined by dynamic logic programming [1]. We have also presented a purely syntactic translation of LUPS programs into logic programs written in a meta-language, and proven its correctness with respect to the declarative semantics. The translation immediately leads to a mechanism for implementing LUPS: with a pre-processor performing the translation, query answering is reduced to that over normal programs. Such a pre-processor, that translates LUPS programs into logic programs that can be run in the DLV-system [8] (which computes stable models of normal programs) has been implemented by us and is available at:
`http://centria.di.fct.unl.pt/~jja/updates/`
Another implementation, also to be found there, includes a pre-processor and a meta-interpreter for query answering under the well-founded semantics, and runs under XSB-Prolog. Though not complete according to the semantics defined in this paper, this other implementation coincides with it on the broad class of stratified programs. This follows from the well known result that the well-founded and the stable sematics coincide on that class. And, for such programs, using XSB-Prolog may have some advantage over using DLV, including better efficiency in query answering, and its less restrictive usage of variables and functors in programs. Further discussion of these implementations is, however, beyond the scope of this paper.

Finally we've discussed and illustrated by examples the applicability of LUPS to several broad knowledge representation domains (viz. active databases, theory of actions, legal reasoning and software specifications). The examples in this paper are simply meant to show that the LUPS language can be a useful tool for representing knowledge in those domains. However they do not provide a deep study of the applicability of the LUPS language. Such a deeper study is subject of ongoing and future work, namely on:

- bridging the gap between knowledge updates and the very active research area of *reasoning about actions*, by analyzing the exact nature of the relationship existing between these two closely related areas and by comparing the syntax and semantics of the languages used in both domains of research.
- applying LUPS as a language for representation and reasoning about legal knowledge and for representation and reasoning about multi-agent communication.
- applying knowledge update methodology to the domain of software engineering by using knowledge updating as a tool for software specification and verification of program correctness.
- using the language of knowledge updates for research on the principles, modeling and design of active knowledge bases.

We also intend to explore the possibility of applying dynamic knowledge updating for the sake of equipping knowledge bases with true object-oriented capabilities. Dynamic knowledge updating seems to provide exactly the tools needed in order to allow instances of class objects to fully inherit knowledge contained in their super-classes. Namely, we can view the subclass instance as an update of its super-class in which new rules represent the updating knowledge, while the super-class represents the original knowledge.

In addition, we plan to further investigate the use of LUPS as a natural framework for reasoning about and updating *visual databases*, with the purpose of ensuring efficient querying, storage and retrieval of images based on the description of their visual contents. As pointed out in [23], image databases must be equipped with *"an ability to reason about objects [...] and an ability to maintain an updated view of the world based on the actions that either the system or the user has performed."* Moreover, the need to handle motion requires the ability to deal with spatio-temporal reasoning, where, at any given time moment, information about a given image frame depends on its own content as well as on the content of its temporal neighbors. LUPS seems to be ideally suited for this purpose. In fact we have already tested the prototype implementation of LUPS on some introductory samples of visual databases, with quite encouraging results.

Dynamic Logic Programming, and LUPS, provide a way of avoiding contradictory information. This is done by limiting the inheritance of rules by inertia: when updating a knowledge base KB with another knowledge base KB', if the latter concludes some L, then limit the inertia of rules from the former that lead to the conclusion $not\, L$. Clearly not all forms of contradiction are avoided in this way. For example, if KB' is itself contradictory, the inconsistency of the resulting update is not avoided. We intend to devise effective methods of handling such knowledge inconsistencies by developing new contradiction removal and diagnosis techniques.

Another possible source of inconsistency not dealt with by LUPS, stems from its use of a two-valued semantics in the object language. In fact, an update may be inconsistent and yet have no clear contradiction (such as A and $not\, A$), the inconsistency coming from the inability of the two-valued semantics to handling a specific knowledge representation phenomenon (e.g. resulting from odd "loops" over negation). To overcome this source of inconsistencies, we plan to extend our approach to the three-valued well-founded object language semantics. This would further allow us to model updates with undefined outcome.

Last, but not least, we intend to further pursue the study of *foundations and basic principles* of dynamic knowledge representation and the adequacy of LUPS as a high-level for that purpose.

Acknowledgements

This work was partially supported by PRAXIS XXI project MENTAL, and by a NATO grant for L. M. Pereira to visit Riverside. Special thanks are due to Paulo Quaresma, for his collaboration on the examples of application.

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, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 45(1-3):43–70, 2000.

[3] 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.

[4] 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, *Joint Conference on Declarative Programming, AGP'99*, 1999.

[5] F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *ICLP'99*. MIT Press, 1999.

[6] C. V. Damásio and L. M. Pereira. Default negated conclusions: why not ? In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *ELP'96*, volume 1050 of *LNAI*, pages 103–117. Springer, 1996.

[7] A. Ed-Dbali and P. Deransart. Software formal specification by logic programming: The example of standard PROLOG. Technical report, INRIA, Paris, 1993.

[8] W. Faber and G. Pfeifer. DLV homepage, 1996. Available at `http://www.dbai.tuwien.ac.at/proj/dlv/`.

[9] 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.

[10] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *ICLP'90*. MIT Press, 1990.

[11] M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[12] M. Gelfond and V. Lifschitz. Action languages. *Linkoping Electronic Articles in Computer and Information Science*, 3(16), 1998.

[13] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI'98*, pages 623–630, 1998.

[14] K. Inoue and C. Sakama. Negation as failure in the head. *Journal of Logic Programming*, 35:39–78, 1998.

[15] 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.

[16] R. Kowalski. Legislation as logic programs. In *Logic Programming in Action*, pages 203–230. Springer-Verlag, 1992.

[17] K. K. Lau and M. Ornaghi. The relationship between logic programs and specifications. *Journal of Logic Programming*, 30(3):239–257, 1997.

[18] J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. In J. Dix, L. M. Pereira, and T. Przymusinski, editors, *Logic Programming and Knowledge Representation*, volume 1471 of *LNAI*. Springer, 1997.

[19] V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *KR'92*. Morgan-Kaufmann, 1992.

[20] F. Lin. Embracing causality in specifying the indirect effects of actions. In *IJCAI'95*, pages 1985–1991. Morgan Kaufmann, 1995.

[21] 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.

[22] T. Przymusinski and H. Turner. Update by means of inference rules. In V. Marek, A. Nerode, and M. Truszczynski, editors, *LPNMR'95*. Springer, 1995.

[23] Report. Workshop on high performance computing and communications for grand challenge applications: Computer vision, speech and natural language processing and artificial intelligence. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):138–154, 1993.

[24] M. Winslett. Reasoning about action using a possible models approach. In V. Marek, A. Nerode, and M. Truszczynski, editors, *AAAI'88*, 1988.

A The proof of Theorem 16

Due to the size of the proof, partially resulting from the size of the translation itself, instead of a complete proof, here we only present its sketch. Moreover, we only consider LUPS programs with assert, persistent assert and assert event commands. The extension of the proof for the case of persistent assert event commands is easy to obtain from the corresponding proofs for the assert event commands and persistent assert commands. The translation of retract and cancel commands is adapted from the well known technique of rule naming.

For the proof of this theorem, we first need to recall the transformational semantics for dynamic programs presented in [2].

Definition 21 (Dynamic Program Update) *By the dynamic program update over the sequence of updating programs $\mathcal{P} = \{P_s : s \in S\}$ we mean the logic program $\biguplus\mathcal{P}$, which consists of the following clauses in the extended language $\overline{\mathcal{L}}$, obtained from augmenting the language $\mathcal{L}$ of $\mathcal{P}$, with the symbols $\overline{A}$, $A(s)$, $\overline{A(s)}$, $A^u(s)$, and $\overline{A^u(s)}$ for any atom A of $\mathcal{L}$ and any $s \in S$[9]:*

(RP) Rewritten program clauses:

$$A^u(s) \leftarrow B_1, \ldots, B_m, \overline{C_1}, \ldots, \overline{C_n} \tag{A.1}$$
$$\overline{A^u(s)} \leftarrow B_1, \ldots, B_m, \overline{C_1}, \ldots, \overline{C_n} \tag{A.2}$$

(A.1) for any clause:

$$A \ \leftarrow B_1, \ \ldots, \ B_m, \ \ not\, C_1, \ \ldots, \ not\, C_n$$

(A.2) for any clause:

$$not\, A \ \leftarrow B_1, \ \ldots, \ B_m, \ not\, C_1, \ \ldots, \ not\, C_n$$

[9] In [2] the symbols A^-, A_s, A_s^-, A_{P_s}, and $A_{P_s}^-$ where used, respectively, instead of $\overline{A}$, $A(s)$, $\overline{A(s)}$, $A^u(s)$, and $\overline{A^u(s)}$

in the program P_s, where $s \in S$. The rewritten clauses are simply obtained from the original ones by replacing atoms A (respectively, the atoms not A) occurring in their heads by the atoms $A^u(s)$ (respectively, $\overline{A^u(s)}$) and by replacing negative premises not C by atoms $\overline{C}$.

(UR) Update rules:

$$\begin{aligned} A(s) &\leftarrow A^u(s) \\ \overline{A(s)} &\leftarrow \overline{A^u(s)} \end{aligned} \tag{A.3}$$

for all objective atoms A and all $s \in S$. The update rules state that an atom A must be true (respectively, false) in state $s \in S$ if it is true (respectively, false) in the updating program P_s.

(IR) Inheritance rules:

$$A(s) \leftarrow A(s-1), not\, \overline{A^u(s)} \tag{A.4}$$

$$\overline{A(s)} \leftarrow \overline{A(s-1)}, not\, A^u(s) \tag{A.5}$$

for all objective atoms A and all $s \in S$. The inheritance rules say that an atom A is true (respectively, false) in the state $s \in S$ if it is true (respectively, false) in the previous state $s-1$ and it is not rejected, *i.e., forced to be false (respectively, true), by clauses in the updating program P_s.*

(DR) Default rules:

$$\overline{A(0)} \tag{A.6}$$

for all objective atoms A. Default rules describe the initial state 0 *by making all objective atoms initially false.*

Definition 22 (Dynamic Program Update at a Given State) *Given a fixed state $s \in S$, by the dynamic program update at the state s, denoted by $\bigoplus_s \mathcal{P}$, we mean the dynamic program update $\biguplus \mathcal{P}$ augmented with:*

(CS$_s$) Current State Rules:

$$A \leftarrow A(s) \tag{A.7}$$

$$\overline{A} \leftarrow \overline{A(s)} \tag{A.8}$$

$$not\, A \leftarrow \overline{A(s)} \tag{A.9}$$

for all objective atoms A. Current state rules specify the current state s in which the updated program is being evaluated and determine the values of the atoms A, $\overline{A}$ and not A.

Suppose now instead that we have augmented the language with the predicate symbols with one more argument t (for every $t \in S$), and use the same argument in all the literals of rules from A.3 to A.6. For example, rule A.4 would then be:

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

Note that with this new translation, the modified rules A.3 to A.6, are exactly the same as update, inertia and default knowledge state rules from definition 14.

Moreover, let rules A.1 and A.2 now be replaced, respectively, by:

$$\begin{aligned} A^u(s,t) &\leftarrow B_1(t,t), \ldots, B_m(t,t), \overline{C_1(t,t)}, \ldots, \overline{C_n(t,t)} \\ \overline{A^u(s,t)} &\leftarrow B_1(t,t), \ldots, B_m(t,t), \overline{C_1(t,t)}, \ldots, \overline{C_n(t,t)} \end{aligned}$$

and the current state rules at state s now be:

$$\begin{aligned} A &\leftarrow A(s,s) \\ not\, A &\leftarrow \overline{A(s,s)} \end{aligned}$$

Compared with the previous one, this translation only adds an extra argument that is used only for t equal to the current state s. Clearly, the stable models restricted to atoms A in the original language $\mathcal{L}$ are exactly the same in both translations.

Moreover, in the latter translation A (resp. $not\, A$) is true in a stable model of the dynamic program at state s iff $A(s,s)$ (resp. $\overline{A(s,s)}$) is true in that stable model. Thus, for answering queries in a dynamic program at some state s, the two remaining current state rules are no longer needed provided that the query is posed after adding to it the two extra arguments, both instantiated with state s. In the sequel, this latter translation of a dynamic program $\bigoplus \mathcal{P}$ is represented by $T(\bigoplus \mathcal{P})$.

Lemma 23 *Let $\bigoplus \mathcal{P}$ be a dynamic program. Then $\bigoplus_{\mathcal{P}} \models_{sm} B_1, \ldots, not\, C_n$ iff $T(\bigoplus \mathcal{P}) \models_{sm} B_1(s,s), \ldots, \overline{C_n(s,s)}$.*

At this point, note that if a LUPS program is comprised only of commands of the form **assert** $L \leftarrow L_1, \ldots, L_k$ then the main Theorem is already proven. In fact, in these simple LUPS programs, $\Upsilon(\mathcal{U})$ is obtained by adding $L \leftarrow L_1, \ldots, L_k$ to program P_i iff **assert** $L \leftarrow L_1, \ldots, L_k \in U_i$. Moreover, the translation of a LUPS programs with just these commands, exactly coincides with the translation above of the dynamic program, and the translation of a query coincides with that of the lemma. So the lemma guarantees the correctness of the Theorem (for this simple case).

Suppose now that we are given a dynamic logic program $\mathcal{P} = P_1 \oplus \ldots \oplus P_i \oplus \ldots \oplus P_n$, resulting from the translation of a LUPS program $U_1 \otimes \ldots \otimes U_i \otimes \ldots \otimes U_n$, and that to U_{i+1} the command **assert** $A \leftarrow L_1, \ldots, L_k$ **when** $L_{k+1}, \ldots, L_m$ is added[10]. The translation of this command (simplified for the case where no retract or cancel commands exist) is the rule:

$$r = A^u(i+1,t) \leftarrow B_1(t,t), \ldots, \overline{B_k(t,t)}, C_1(i,i), \ldots, \overline{C_m(i,i)}$$

According to lemma 23, and since the indexes in the two extra arguments never increase, $T(\mathcal{P}) \cup \{r\} \models_{sm} C_1(i,i), \ldots, \overline{C_m(i,i)}$ iff $\mathcal{P}_i \models_{sm} C_1, \ldots, \overline{C_n}$ (where $\mathcal{P}_i$ denotes the sequence $\mathcal{P}$ up to program P_i). Accordingly, if $\mathcal{P}_i \models_{sm} C_1, \ldots, \overline{C_n}$, then replacing the rule above by $r_t = A^u(i+1,t) \leftarrow B_1(t,t), \ldots, \overline{B_k(t,t)}$ doesn't change the stable models, and clearly $T(\mathcal{P}) \cup \{r_t\}$ is the translation of the dynamic program obtained from $\mathcal{P}$ by adding $A \leftarrow L_1, \ldots, L_k$ to P_{i+1}. I.e., if $\mathcal{P}_i \models_{sm} C_1, \ldots, \overline{C_n}$ then adding r produces the same effects as adding $A \leftarrow L_1, \ldots, L_k$ to P_{i+1}. If $\mathcal{P}_i \not\models_{sm} C_1, \ldots, \overline{C_n}$ then the rule r can be deleted, and so adding **assert** $A \leftarrow L_1, \ldots, L_k$ **when** $L_{k+1}, \ldots, L_m$ to U_{i+1} has no effect on $\mathcal{P}$. This is exactly the semantics of the assert command in Def. 10.

Assume now that, with the dynamic program $\mathcal{P}$ as above resulting from the translation of the same LUPS program, one adds to U_{i+1} the command **always** $A \leftarrow L_1, \ldots, L_k$ **when** $L_{k+1}, \ldots, L_m$. The simplified translation of this command results in the rules:

$$r(q+1) = A^u(q+1,t) \leftarrow B_1(t,t), \ldots, \overline{B_k(t,t)}, C_1(q,q), \ldots, \overline{C_m(q,q)}$$

for all $q+1 > i$. Clearly, each of these $r(q+1)$ rules is the same as the rule that would be introduced in the translation if a command **assert** $A \leftarrow L_1, \ldots, L_k$ **when** $L_{k+1}, \ldots, L_m$ is added to U_{q+1}. So, according to this translation, the addition of **always** $A \leftarrow L_1, \ldots, L_k$ **when** $L_{k+1}, \ldots, L_m$ to U_{i+1} is equivalent to the addition of **assert** $A \leftarrow L_1, \ldots, L_k$ **when** $L_{k+1}, \ldots, L_m$ to every U_{q+1} for $q \geq i$, and this is the meaning of the persistent assert command in definition 10.

Suppose now that we are given a dynamic logic program $\mathcal{P}_{i+1} = P_1 \oplus \ldots \oplus P_{i+1}$, resulting from the translation of a LUPS program $U_1 \otimes \ldots \otimes U_{i+1}$, and that U_{i+1} includes the command **assert event** $A \leftarrow L_1, \ldots, L_k$ **when** $L_{k+1}, \ldots, L_m$. The

[10] The case where the literal in the head is of the form *not* A is in all respects similar to this one, and omitted for brevity.

simplified translation of this command is the rule r:

$$A^u(i+1,i+1) \leftarrow B_1(i+1,i+1),\ldots,\overline{B_k(i+1,i+1)},C_1(i,i),\ldots,\overline{C_m(i,i)}$$

Consider also the rule r' resulting from the translation of the command resulting from the assert event above when dropping the keyword **event**:

$$A^u(i+1,t) \leftarrow B_1(t,t),\ldots,\overline{B_k(t,t)},C_1(i,i),\ldots,\overline{C_m(i,i)}$$

Note that the truth of any literal $L(q,q)$, with $q < i+1$, in the stable models of $T(\mathcal{P}_{i+1})\cup\{r\}$ does not depend on the addition of r, i.e. $T(\mathcal{P}_{i+1}) \models_{sm} L(q,q)$ iff $T(\mathcal{P}_{i+1})\cup\{r\} \models_{sm} L(q,q)$. The same happens when adding rule r' instead. Moreover, note that for $q = i+1$ adding rule r as the same effect has adding r', i.e. $T(\mathcal{P}_{i+1})\cup\{r\} \models_{sm} L(i+1,i+1)$ iff $T(\mathcal{P}_{i+1})\cup\{r'\} \models_{sm} L(i+1,i+1)$. Thus, for any q up to $i+1$ adding r or r' has exactly the same effect, meaning that, up to $i+1$, the semantics of adding the assert event command given by the translation, is the same as that of adding the assert command. This agrees with definition 10. To prove the correctness of assert event commands we have also to prove that, if more commands $U_{i+2}\otimes\ldots\otimes U_n$ are given, the truth of literals $L(q,q)$ for some $q > i+1$ is not affected by that rule (in definition 10 this is guaranteed by adding to P_{i+2} a literal falsifying the body of the rule). And this is indeed the case. For that, note that in the inertial rules the second argument is kept fixed. Thus, since $q \neq i+1$, rule r is never activated via inertia rules. The only possibility of rule r being activated in $Tr(\mathcal{U})$ is via some literal $C(i+1,i+1)$ resulting from the translation of a literal in a **when**-clause of a command in U_{i+1}. But this is exactly the case where its truth is verified in P_{i+1}, and where rule r must be taken into account. The complete proof of this point is slightly more complex, to take into consideration to non-relevance property of stable models. The long and short of is is that because of the stratification on state indexes, non-relevance between states is not introduced. The technicalities of this are omitted here for brevity.

The complete proof continues by showing the correctness of persistent assert event commands. This is done by reducing the translation of such commands to the translation of several assert event commands, in the same way as the proof of persistent assert commands is done by reducing their translation into several assert commands. The results are then generalized to the case where retract and cancel commands exist, and the correctness of these commands is shown. Note apropos that the translation of retract and cancel commands relies on the known technique of naming rules. The proof is thus based on the correctness of naming with respect to the translation into dynamic programs of definition 10.