A Modified Semantics for LUPS

João Alexandre Leite*

CENTRIA - Centro de Inteligência Artificial
Universidade Nova de Lisboa
2829-516 Caparica
Portugal
jleite@di.fct.unl.pt

Abstract. Following the introduction of *Dynamic Logic Programming* in [1], the language of updates *LUPS* was introduced in [2]. Whereas *Dynamic Logic Programming* provides a meaning to sequences of logic programs, each of them representing a state of the world, *LUPS* allows the specification of such states and state transitions.
In this paper, we take a closer look at the language *LUPS* and identify one problem with its semantics and a possible, important, extension to its set of commands. We then propose an extension to the syntax of *LUPS* as well as a new semantics that solves the identified problem. We illustrate the changes by means of two examples.

1 Introduction and Motivation

In the past few years, research in the area of *Nonmonotonic Reasoning* has devoted some attention to the problem of dynamically adapting a knowledge base (KB) to correctly represent a world that changes, i.e. how to update a KB. For the case where the KB is a theory in classical propositional logic, adequate solutions have been proposed in [8] and [15]. In [12,13], these solutions were adapted to allow for the update of logic programs and deductive databases, following the so called *interpretation update* approach. This approach has been shown inadequate when applied to nonmonotonic theories, often leading to counterintuitive results as pointed out in [9]. Since then, several approaches for updating KBs represented by logic programs have been proposed [1,3,4,9,10,14,16]. Each of these approaches proposes a semantics for what the outcome of the update of a logic program by another such program ought to be or, more generally, what the meaning of a sequence of logic programs should be. For considerations and comparisons wrt. these approaches see [4].

In [2], the authors argue that besides assigning a meaning to a sequence of logic programs, one also needs a language to specify how such sequence of programs is to be constructed i.e., besides declaratively specifying the states of a KB, it is advantageous to also declaratively specify the state transitions. And these state transitions should be allowed to depend on the states themselves. To this purpose, the language of updates *LUPS* [2] was introduced. In the *LUPS*

* Partially supported by PRAXIS XXI scholarship no. BD/13514/97

P. Brazdil and A. Jorge (Eds.): EPIA 2001, LNAI 2258, pp. 261–275, 2001.

framework, the next state of the KB is produced according to a set of commands and the KB at the current state. Such *LUPS* commands allow the specification of statements such as: "*a certain rule should belong to the next state if some condition holds in the current state*", which would be represented by the command "**assert** *Rule* **when** *Condition*". Similar commands exist for the retraction of rules, for the specification of permanent assert commands, i.e. an assert command that is to be executed at every state after being issued, and for the cancellation of such persistent commands. For further motivation for the need of a language as *LUPS*, and some application examples, see [2]. Throughout the remainder of this paper we assume that the reader is familiar with *LUPS* and its *Dynamic Logic Programming* (*DLP*) based semantics. In Appendix we provide an overview of *DLP* and *LUPS* with all the relevant intuitions and definitions.

1.1 Motivation

In *LUPS*, the authors introduced a class of commands whose immediate effect should only hold in the successor state and should not persist by inertia in subsequent states. This kind of non-inertial commands are indicated by the keyword **event** (e.g. "**assert event** *Rule* **when** *Condition*").

According to the intuitive reading above, if we want to update an initial KB, P_1, with two consecutive updates U_2 and U_3, such that U_2 only contains such non-inertial commands, and U_3 is empty, it seems reasonable to expect that after the second update, U_3, what holds true is exactly equal to what held true before the first update, i.e. what holds true at P_1. Unfortunately this is not the case in *LUPS*, as illustrated by the following example:

Example 1. Consider the simple case where $P_1 = \{a \leftarrow\}$, possibly obtained by a past update command such as **assert** $a \leftarrow$, and the following sequence of updates:

$$U_2 = \{\textbf{assert event } a \leftarrow\}$$
$$U_3 = \{\}$$

At state 3, i.e. after the update U_3, according to the semantics of *LUPS* we have $M_3 = \{\}$ as the only stable model, i.e. a is not a consequence of the knowledge base at state 3.

In this example, $M_3 = \{\}$ is the only stable model because the command **assert event** $a \leftarrow$ asserts the rule $a \leftarrow$ at state 2, but then causes the removal of all rules (past and present) of the form $a \leftarrow$, i.e. both the rule specified by U_2 and the rule of P_1 are removed. We argue that $M_3^{'} = \{a\}$ should be the only stable model at state 3, because the command **assert event** $a \leftarrow$ should not affect (remove at state 3) the rule $a \leftarrow$ that was previously asserted at state 1, i.e. the rule in P_1. Let us look at another example:

Example 2. Consider a slight modification in the previous example, such that the initial program is, now, $P_1^* = \{a \leftarrow not\ \bot\}$ (where, as usual, $\bot$ is a reserved

proposition with the property of being false in every stable model i.e, $not\ \bot$ belongs to every stable model). If the same update sequence is performed, after the update U_3 we have $M_3^* = \{a\}$ as the only stable model.

Updates are somewhat syntactical in nature (cf. [1]), but in this example, this syntactical difference in behaviour should not exist. We argue that both examples should have the same outcome, i.e. they should both have $M = \{a\}$ as the only stable model at state 3. To avoid such problem, we argue that an event command should be exerted on the single asserted (or retracted) instance of a rule and not on all other syntactically equal ones.

The main contribution of this paper is a proposal for a change in the semantics of *LUPS* to correct this undesirable behaviour.

The second contribution of this paper resides in the extension of the *LUPS* syntax with the introduction of a persistent retract command. In *LUPS*, besides the assertion and retraction commands, denoted by the keywords **assert** and **retract** respectively, that may only contribute to the state transition for which they were specified, there is a command, denoted by the keyword **always**, which can be seen as a permanent assert command, i.e. until it is cancelled, it will cause the assertion of a specific rule every time the specified condition holds. We argue that such persistent command should also exist for the retraction of rules. To illustrate our claim, let us consider the following example from [2]:

Example 3. 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. The specification in *LUPS*, as presented in [2], comprises the following persistent update commands[1]:

$$\textbf{always}\ jail(X) \leftarrow abortion(X)\ \textbf{when}\ repC, repP \tag{1}$$

$$\textbf{always}\ not\, jail(X) \leftarrow abortion(X)\ \textbf{when}\ not\, repC, not\, repP \tag{2}$$

The authors further state that in this example, alternatively, instead of the second command they could have used a retract command:

$$\textbf{retract}\ jail(X) \leftarrow abortion(X)\ \textbf{when}\ not\, repC, not\, repP \tag{3}$$

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

We argue that neither of the proposed solutions, to represent the abolition of the abortion law when the Democrats take over (commands (2) and (3)), properly represents the intuition stated in the scenario.

[1] 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.

The argument against the use of command (2) is implicitly stated by the authors of [2] when they say that "*since there is no other rule implying jail, retracting the rule is safely equivalent to retracting its conclusion*". In fact, if there were other rules implying jail, which is fair to expect in a realistic scenario, such as for example a rule stating that assassinations are punishable by jail, then some undesirable effects would occur: -suppose such rule was represented by an update command, at the initial state, of the form:

$$\textbf{assert}\ jail(X) \leftarrow assassination(X) \tag{4}$$

If this command had been previously issued, then, after the Democrats take over both Congress and the Presidency, someone ($mary$) that both assassins someone and performs an abortion would *not* be punished by jail. This is so because the rule asserted by command (2) would reject, according to the DLP semantics, the previously asserted rule specified by command (4). The resulting knowledge base would have $not\, jail(mary)$ as a consequence.

The argument against the use of command (3) resides in the fact that, to effectively represent the intended meaning, this command would have to belong to every update i.e. it would have to be explicitly added to the specification of *every* state transition. Not only this does not add to the simplicity of the language but, also, the fact that the representation of the effect of Republicans taking over requires a single update command, and that the representation of the effect of Democrats taking over requires several update commands, one at each update, is somehow unintuitive. The introduction of the **always** command in [2] was justified with the need to avoid such consecutive repetitions of the **assert** command. We believe that such persistent command should also exist for the retraction of rules. A command that, until cancelled, retracts a specific rule from the KB every time the specified condition holds. In this paper we introduce such command.

Throughout, to differentiate the original $LUPS$ language and the extended and modified version here proposed, we refer to the latter as $LUPS^*$.

The remainder is structured as follows: First, in Sect.2, we introduce the syntax of the extended language $LUPS^*$. In Sect.3 we propose a semantics for this extended language that corrects the above mentioned problem. In Sect.4 we illustrate with two examples. In Sect. 5 we compare both semantics by means of a desirable property. We then conclude in Sect.6.

2 $LUPS^*$ - Syntax

In this Section we formalize the language of $LUPS^*$. We will keep all the commands of $LUPS$, with a slight modification in the syntax of two of such commands, which will be explained below, and introduce the above mentioned persistent retract command and its corresponding cancellation command.

The language of $LUPS^*$ will contain the basic non-persistent commands for the assertion and retraction of rules, denoted by the keywords **assert** and

retract. They are of the form:

$$\textbf{assert } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{5}$$
$$\textbf{retract } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{6}$$

Both these commands can be made persistent, i.e. until cancelled they are added to all subsequent updates. We will identify such persistent commands by prefixing the non-persistent commands with the keyword **always**. They are thus of the form:

$$\textbf{always assert } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{7}$$
$$\textbf{always retract } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{8}$$

Note here the first modification in the syntax of a *LUPS* command. In *LUPS*, command (7) does not have the keyword **assert**, being identified by the keyword **always** alone. Since we are now introducing the persistent retraction command, this new notation becomes more symmetrical and therefore more intuitive.

All the previous commands are inertial. To obtain the corresponding non-inertial commands one simply adds the keyword **event** to obtain the following commands:

$$\textbf{assert event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{9}$$
$$\textbf{retract event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{10}$$
$$\textbf{always assert event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{11}$$
$$\textbf{always retract event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{12}$$

Just as in *LUPS* there is a cancellation command for the persistent command **always**, denoted by the keyword **cancel**, so there will be one in *LUPS** but now denoted by the keyword **cancel assert**, to simplify the introduction of a cancellation command for the persistent retraction command, which will be denoted by the keyword **cancel retract**. These two commands are:

$$\textbf{cancel assert } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{13}$$
$$\textbf{cancel retract } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{14}$$

The syntax of the update commands of *LUPS** is, formally:

Definition 1 (*LUPS** - Update Commands). *A* LUPS* update command, *or* command *for short, is a propositional expression of any of the forms*[2]

$$[\textbf{always }]\,\textbf{assert}\,[\,\textbf{event}]\ \ R \textbf{ when } \phi \tag{15}$$
$$[\textbf{always }]\,\textbf{retract}\,[\,\textbf{event}]\ \ R \textbf{ when } \phi \tag{16}$$
$$\textbf{cancel assert } R \textbf{ when } \phi \tag{17}$$
$$\textbf{cancel retract } R \textbf{ when } \phi \tag{18}$$

[2] Where [**a**] will be used for notational convenience denoting either the presence or absence of **a**.

where R is a rule of the form $L \leftarrow L_1, \ldots, L_k$ and ϕ is a (possibly empty) conjunction of literals, $L_{k+1}, \ldots, L_m$, where L and each L_i are literals. If ϕ is empty, we simply omit the **when** *keyword. We refer to those commands* without *(resp.* with*) the keyword* **event** *as* inertial *(resp.* non-inertial*) commands. We refer to those commands* with *(resp.* without*) the keyword* **always** *as* persistent *(resp.* non-persistent*) commands.*

The following table summarizes the correspondence between the syntax of $LUPS^*$ and that of $LUPS$:

$LUPS^*$		$LUPS$
assert [**event**]	$\leftrightarrow$	**assert** [**event**]
retract [**event**]	$\leftrightarrow$	**retract** [**event**]
always assert [**event**]	$\leftrightarrow$	**always** [**event**]
always retract [**event**]		non existing
cancel assert	$\leftrightarrow$	**cancel**
cancel retract		non existing

An update program in $LUPS^*$ is defined as follows:

Definition 2 ($LUPS^*$ - Update Program). *An update program in* LUPS* *is a finite sequence of updates, where an update is a set of commands of the form (15) to (18).*

3 $LUPS^*$ - Semantics

The semantics of $LUPS^*$ is defined by incrementally translating update programs into sequences of generalized logic programs and by considering the semantics of the DLP formed by them.

Let $\mathcal{U} = U_1 \otimes ... \otimes U_n$ be a $LUPS^*$ update program. At every state t we determine the corresponding DLP, $\Upsilon_t(\mathcal{U}) = \mathcal{P}_t$.

The translation of a $LUPS^*$ program into a dynamic program is similar to the one presented for $LUPS$. It is made by induction, starting from the empty program P_0 and, for each update U_t, given the already built dynamic program $P_0 \oplus \cdots \oplus P_{t-1}$, determining the resulting program $P_0 \oplus \cdots \oplus P_{t-1} \oplus P_t$. To cope with persistent update commands, as for the $LUPS$ semantics, we also consider a set containing all currently active persistent commands, although its definition must be extended to deal with the newly introduced persistent retraction commands. As in $LUPS$, the retraction of rules imposes its unique identification. Therefore, the language of the resulting dynamic logic program must be extended with a new propositional variable "$N(R)$" for every rule R appearing in the original $LUPS$ program. To properly handle non-inertial commands, we also need to uniquely associate those rules appearing in non-inertial commands with the states they belong to. To this end, the language of the resulting dynamic logic program must also be extended with a new propositional variable "$Ev(R, S)$" for every rule R appearing in a non-inertial command in the original $LUPS$ program, and every state S.

We now present the translation for $LUPS^*$:

Definition 3 (Translation into dynamic logic programs). *Let* $\mathcal{U} = U_1 \otimes \cdots \otimes U_n$ *be a* LUPS* *update program. The corresponding dynamic logic program* $\Upsilon^*(\mathcal{U}) = \mathcal{P} = \mathcal{P}_n = P_0 \oplus \cdots \oplus P_n$ *is obtained by the following inductive construction, using at each step* t *an auxiliary set of persistent commands* PC_t*:*

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

Inductive step: Let $\Upsilon^*_{t-1}(\mathcal{U}) = \mathcal{P}_{t-1} = P_0 \oplus \cdots \oplus P_{t-1}$ with the set of persistent commands PC_{t-1} be the translation of $\mathcal{U}_{t-1} = U_1 \otimes \cdots \otimes U_{t-1}$. The translation of $\mathcal{U}_t = U_1 \otimes \cdots \otimes U_t$ is $\Upsilon^*_t(\mathcal{U}) = \mathcal{P}_t = P_0 \oplus \cdots \oplus P_{t-1} \oplus P_t$ with the set of persistent commands PC_t, where:

$$
\begin{aligned}
PC_t = PC_{t-1} \cup \{&\textbf{assert } R \textbf{ when } \phi : \textbf{always assert } R \textbf{ when } \phi \in U_t\} \cup \\
\cup \{&\textbf{retract } R \textbf{ when } \phi : \textbf{always retract } R \textbf{ when } \phi \in U_t\} \cup \\
\cup \{&\textbf{assert event } R \textbf{ when } \phi : \textbf{always assert event } R \textbf{ when } \phi \in U_t\} \cup \\
\cup \{&\textbf{retract event } R \textbf{ when } \phi : \textbf{always retract event } R \textbf{ when } \phi \in U_t\} \cup \\
- \{&\textbf{assert } [\textbf{event}] \ R \textbf{ when } \phi : \textbf{cancel assert } R \textbf{ when } \psi \in U_t \wedge \\
&\qquad \wedge \bigoplus \mathcal{P}_{t-1} \vDash \psi\} - \\
- \{&\textbf{assert } [\textbf{event}] \ R \textbf{ when } \phi : \textbf{always retract } [\textbf{event}] \ R \textbf{ when } \psi \in U_t \wedge \\
&\qquad \wedge \bigoplus \mathcal{P}_{t-1} \vDash \psi\} - \\
- \{&\textbf{retract } [\textbf{event}] \ R \textbf{ when } \phi : \textbf{cancel retract } R \textbf{ when } \psi \in U_t \wedge \\
&\qquad \wedge \bigoplus \mathcal{P}_{t-1} \vDash \psi\} - \\
- \{&\textbf{retract } [\textbf{event}] \ R \textbf{ when } \phi : \textbf{always assert } [\textbf{event}] \ R \textbf{ when } \psi \in U_t \wedge \\
&\qquad \wedge \bigoplus \mathcal{P}_{t-1} \vDash \psi\}
\end{aligned}
$$

$$NU_t = U_t \cup PC_t$$

$$
\begin{aligned}
P_t = \{&not\, N(R) \leftarrow : \textbf{retract } R \textbf{ when } \phi \in NU_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \phi\} \cup \\
\cup \{&N(R) \leftarrow ; H(R) \leftarrow B(R), N(R) : \textbf{assert } R \textbf{ when } \phi \in NU_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \phi\} \cup \\
\cup \{&H(R) \leftarrow B(R), Ev(R,t) : \textbf{assert event } R \textbf{ when } \phi \in NU_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \phi\} \cup \\
\cup \{&not\, N(R) \leftarrow Ev(R,t) : \textbf{retract event } R \textbf{ when } \phi \in NU_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \phi\} \cup \\
\cup \{&not\, Ev(R,t-1) \leftarrow ; Ev(R,t) \leftarrow\}
\end{aligned}
$$

Note that the body of every rule R must be extended either with the predicate $N(R)$ or with the predicate $Ev(R,t)$. The differences between this transformation and the original $LUPS$ transformation are the following: - the modification in the definition of the set of active persistent commands, PC_t, to deal with the newly introduced persistent retraction command and the corresponding cancellation command; - the modification in the definition of the generalized logic program at state t, P_t, to properly deal with non-inertial commands. This is achieved by treating inertial and non-inertial rules in a separate manner: the former are dealt with as in LUPS while the latter are extended with the predicate $Ev(R,t)$ which is only made true for the duration of one state. This is achieved simply by including the rules $not\, Ev(R,t-1) \leftarrow$ and $Ev(R,t) \leftarrow$ in the generalized logic program of every state.

The semantics of $LUPS^*$ is, as expected:

Definition 4 (*LUPS** Semantics). *Let* $\mathcal{U} = U_1 \otimes ... \otimes U_n$ *be a* LUPS* *program. A query* **holds** $L_1, \ldots, L_k$ **at** t*? is true in* $\mathcal{U}$ *iff* $\bigoplus \Upsilon_t^*(\mathcal{U}) \models L_1, \ldots, L_k$*, or, equivalently, iff* $\bigoplus \mathcal{P}_t \models L_1, \ldots, L_k$*. By* $SM(\mathcal{U})$ *we mean the set of all stable models of* $\Upsilon^*(\mathcal{U})$ *at state* n*, modulo the newly introduced literals* $N(R)$ *and* $Ev(R, S)$*.*

4 Illustrative Examples

Follows an example to illustrate the different behaviour between *LUPS* and *LUPS**:

Example 4. Consider a public building with several floors. Initially, the security policy of the building states that any person (P) is allowed into any of its floors (F) only if they have a special permit for that floor. This can be represented by the update U_1:

$$\begin{aligned} U_1 : & \textbf{assert}\,(allowed(P,F) \leftarrow permission(P,F)) \\ & \textbf{assert}\,(not\, allowed(P,F) \leftarrow not\, permission(P,F)) \end{aligned}$$

Later on, the administration decided to open up a public relations office, to be situated in the ground floor. They had to update the security policy which, from then on, would allow any person in the ground floor provided they have some form of identification. Let us suppose this happened in the second day (state), and is represented by the update U_2:

$$U_2 : \textbf{assert}\,(allowed(P, ground) \leftarrow id(P))$$

Further down the line (at the third day), the administration decided to, once every now and then, declare an open day when everyone, for the duration of one day, was allowed to visit the entire building, provided they have some form of Id. This is represented by the update U_3:

$$U_3 : \textbf{always assert event}\ \ (allowed(P,F) \leftarrow id(P))\ \ \textbf{when}\ open\_day$$

Suppose that both Mary and John have Ids and Mary has a permission for the second floor, represented by the facts $permission(mary, second)$, $id(john)$ and $id(mary)$, in the KB. At day five there is an open day represented by the update at state 4:

$$U_4 : \textbf{assert event}\ \ (open\_day \leftarrow)$$

According to the *LUPS** semantics, at state 4 Mary is allowed in the ground and second floors, and John is only allowed in the ground floor; at state 5 both Mary and John are allowed in all floors of the building; at state 6 and thereafter everything is back as it was at state 4, with Mary being allowed in the ground and second floors, and John only being allowed in the ground floor, as expected. If this problem was to be tackled with the semantics of *LUPS*, everything would be the same until (and including) state 5 but, at state 6, John and Mary (and

anybody else) would no longer be allowed in the ground floor, which seems to be rather unintuitive. This is so because not only the rule asserted by the **always assert event** command in U_3 is removed, but so is the rule asserted by U_2, simply because it has the same syntax. □

We now show how Example 3 would be encoded in $LUPS^*$:

Example 5. Consider the scenario of Example 3. The specification in $LUPS^*$ would comprise the following persistent update commands:

$$\textbf{always assert } jail(X) \leftarrow abortion(X) \textbf{ when } repC, repP$$
$$\textbf{always retract } jail(X) \leftarrow abortion(X) \textbf{ when } not\, repC, not\, repP$$

If, as before, at the initial state we also have the following update command:

$$\textbf{assert } jail(X) \leftarrow assassination(X)$$

Then, after the Democrats take over both Congress and the Presidency, if Mary both assassins someone and performs an abortion, she will now be punished by jail, as intended. □

5 Comparison

In this section we compare $LUPS$ and $LUPS^*$ by means of a property that partially characterizes the desired behaviour of the semantics wrt. non-inertial commands.

Before we express this property, we start with the definition of *update equivalence* for update programs, according to which two update programs are *update equivalent* iff their semantics coincides after an arbitrary number of updates with arbitrary update programs. Formally:

Definition 5 (Update Equivalence). *Let $\mathcal{U}_1$ and $\mathcal{U}_2$ be two update programs. We say that $\mathcal{U}_1$ and $\mathcal{U}_2$ are* update equivalent*, denoted by $\mathcal{U}_1 \overset{\otimes}{\equiv} \mathcal{U}_2$, iff for every update program $\mathcal{Q}$, $\mathcal{U}_1 \otimes \mathcal{Q}$ is semantically equivalent to $\mathcal{U}_2 \otimes \mathcal{Q}$, i.e.:*

$$SM\,(\mathcal{U}_1 \otimes \mathcal{Q}) = SM\,(\mathcal{U}_2 \otimes \mathcal{Q})$$

where if $\mathcal{R} = R_{r_1} \otimes ... \otimes R_{r_n}$ and $\mathcal{S} = S_{s_1} \otimes ... \otimes S_{s_m}$ are two update programs, by $\mathcal{R} \otimes \mathcal{S}$ we mean the update program $R_{r_1} \otimes ... \otimes R_{r_n} \otimes S_{s_1} \otimes ... \otimes S_{s_m}$.

With this property, we can now come back to our main claim. In particular, we claim that if we have a sequence of updates such that only non-inertial commands are executed, the knowledge state before the execution of such sequence of commands and the knowledge state after the execution of all such commands should be *update equivalent*. This means that the long term effect (more than one state) of non-inertial commands should only reside in their interaction with inertial commands, be them persistent or not, i.e., rules asserted or retracted

by non-inertial commands should only change the semantics at the next state to allow and/or prevent the execution of other commands at that state, which may be inertial and therefore change the semantics at subsequent states. In particular, if there is a sequence of updates without any inertial command to be executed, followed by a state transition specified by an empty update, then, after its execution the knowledge state should be *equivalent* to the initial one. In other words, we want the effect of non-inertial commands to be, in fact, non inertial. The referred empty update is necessary because the immediate effect of every non-inertial command holds for one state, i.e. we need an extra state transition for such effect to be removed.

This is formally defined as follows:

Definition 6 (Stability wrt non-inertial commands). *An update language is* stable wrt. non-inertial commands *iff for every update programs $\mathcal{U}_1$ and $\mathcal{U}_2$ such that $\mathcal{U}_1$ consists of updates with non-persistent commands only, and $\mathcal{U}_2$ consists of updates with non-persistent, non-inertial commands only,*

$$\mathcal{U}_1 \otimes U_\emptyset \overset{\otimes}{\equiv} \mathcal{U}_1 \otimes \mathcal{U}_2 \otimes U_\emptyset$$

where $U_\emptyset$ denotes an empty update.

The restriction imposed on the updates of $\mathcal{U}_1$ to only contain non-persistent commands, i.e. without the keyword **always**, is justified by the fact that if such persistent commands were present, they could be executed at subsequent states, together with the updates of $\mathcal{U}_2$, thus invalidating our goal to have only non-inertial commands being executed at the state transitions corresponding to the updates of $\mathcal{U}_2$. Recall that an inertial command is valid at all subsequent states, until cancelled. Note that to express our intuition it suffices to guarantee that at the state transitions corresponding to the updates of $\mathcal{U}_2$ only non-inertial commands are executed. In fact, this would also be achieved by allowing $\mathcal{U}_1$ to contain persistent non-inertial commands so as long as we ensure that at the state transition corresponding to $U_\emptyset$ no commands are executed, but we will keep to this simpler formulation.

Proposition 1. LUPS *is not stable wrt. non-inertial commands.*

Example 1 above shows that *LUPS* is not stable wrt. non-inertial commands. It is worth noting that the recently proposed language of updates *EPI* [5], being based on the semantics of *LUPS*, is also not stable wrt. non-inertial commands. The same example also applies to *EPI*.

Theorem 1. LUPS* *is stable wrt. non-inertial commands.*

6 Conclusion

In this paper we have drawn the attention of the reader to an intuitively incorrect behaviour of the semantics of the language of updates *LUPS*, when dealing with

non-inertial commands. Hoping to have convinced the reader that such behaviour is in fact incorrect, we have then proposed a modification to the semantics of *LUPS* to correct such problem. Both semantics, the original and the modified, were compared by means of a desirable property that only holds in the latter.

We have also suggested the need for a new persistent retraction command, symmetrical to the persistent assertion command, and extended the syntax and adapted the semantics accordingly.

Each contribution was illustrated by means of an example.

References

1. 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. Abstract titled *Dynamic Logic Programming* appeared in Procs. of KR-98.
2. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS : A language for updating logic programs. In *Procs. of LPNMR-99*, LNAI-1730. Springer, 1999.
3. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In *Procs. of ICLP-99*. MIT Press, 1999.
4. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Considerations on updates of logic programs. In *Procs. of JELIA-00*, LNAI-1919. Springer, 2000.
5. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Specifying update policies for nonmonotonic knowledge bases. In *Procs. of DGNMR-01*, 2001. To appear in Procs. of IJCAI-01.
6. M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In *Procs. of ICLP-88*. MIT Press, 1988.
7. K. Inoue and C. Sakama. Negation as failure in the head. *Journal of Logic Programming*, 35:39–78, 1998.
8. H. Katsuno and A. Mendelzon. On the difference between updating a knowledge base and revising it. In *Procs. of KR-91*. Morgan Kaufmann, 1991.
9. J. A. Leite and L. M. Pereira. Generalizing updates: From models to programs. In *Procs of. LPKR-97*, LNAI-1471. Springer, 1997.
10. J. A. Leite and L. M. Pereira. Iterated logic program updates. In *Procs. of JICSLP-98*. MIT Press, 1998.
11. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In *Procs. of KR-92*. Morgan-Kaufmann, 1992.
12. V. W. Marek and M. Truszczyński. Revision specifications by means of programs. In *Procs. of JELIA-94*, LNAI-838. Springer, 1994.
13. T. C. Przymusinski and H. Turner. Update by means of inference rules. *Journal of Logic Programming*, 30(2):125–143, 1997.
14. C. Sakama and K. Inoue. Updating extended logic programs through abduction. In *Procs. of LPNMR-99*. Springer, 1999.
15. M. Winslett. Reasoning about action using a possible models approach. In *Procs. of NCAI-88*. AAAI Press, 1988.
16. Y. Zhang and N. Foo. Updating logic programs. In *Procs. of ECAI'98*. Morgan Kaufmann, 1998.

A Background

In this Appendix we provide some background on Generalized Logic Programs, Dynamic Logic Programming and *LUPS*.

Generalized Logic Programs Here we recapitulate the syntax and stable semantics of generalized logic programs[3] [1].

By a *generalized logic program* P in a language $\mathcal{L}$ we mean a finite or infinite set of propositional clauses of the form $L_0 \leftarrow L_1, \ldots, L_n$ where each L_i is a literal (i.e. an atom A or the default negation of an atom $not\, A$). If r is a clause (or rule), by $H(r)$ we mean L_0, and by $B(r)$ we mean $L_1, \ldots, L_n$. If $H(r) = A$ (resp. $H(r) = not\, A$) then $not\, H(r) = not\, A$ (resp. $not\, H(r) = A$). By a (2-valued) *interpretation* M of $\mathcal{L}$ we mean any set of literals from $\mathcal{L}$ that satisfies the condition that for any A, *precisely one* of the literals A or $not\, A$ belongs to M. Given an interpretation M we define $M^+ = \{A : A \text{ is an atom}, A \in M\}$ and $M^- = \{not\, A : A \text{ is an atom}, \ not\, A \in M\}$. Wherever convenient we omit the default (negative) atoms when describing interpretations and models. Also, rules with variables stand for the set of their ground instances. We say that a (2-valued) interpretation M of $\mathcal{L}$ is a stable model of a generalized logic program P if $\rho(M) = least\,(\rho(P) \cup \rho(M^-))$, where $\rho(.)$ univocally renames every default literal $not\, A$ in a program or model into new atoms, say not_A. In the remaining, we refer to a GLP simply as a logic program (or LP).

Dynamic Logic Programming Next we recall the semantics of *dynamic logic programming* [1]. A dynamic logic program $\mathcal{P} = \{P_s : s \in S\} = P_0 \oplus ... \oplus P_n \oplus ...$, is a finite or infinite sequence of LPs, indexed by the finite or infinite set $S = \{1, 2, \ \ldots, \ n, \ \ldots\}$. Such sequence may be viewed as the outcome of updating P_0 with P_1, ..., updating it with P_n,... As we will see, in *LUPS* each P_i is determined by the i^{th} state transition. The role of dynamic logic programming is to ensure that these newly added rules are in force, and that previous rules are still valid (by inertia) for as long as they do not conflict with more recent ones. The notion of dynamic logic program at state s, denoted by $\bigoplus_s \mathcal{P} = P_0 \oplus ... \oplus P_s$, characterizes the meaning of the dynamic logic program when queried at state s, by means of its stable models, defined as follows:

Definition 7 (Stable Models of DLP). *Let* $\mathcal{P} = \{P_s : s \in S\}$ *be a* dynamic logic program*, let* $s \in S$*. An interpretation* M_s *is a* stable model of $\mathcal{P}$ at state s *iff*

$$M_s = least\,([\mathcal{P}_s - Reject(s, M_s)] \cup Default\,(\mathcal{P}_s, M_s))$$

[3] The class of GLPs (i.e. logic programs that allow default negation in the premisses and heads of rules) can be viewed as a special case of yet broader classes of programs, introduced earlier in [7] and in [11], and, for the special case of normal programs, their semantics coincides with the stable models semantics [6].

where

$\mathcal{P}_s = \bigcup_{i \leq s} P_i$
$Reject(s, M_s) = \{r \in P_i : \exists r' \in P_j, i < j \leq s, H(r) = not\, H(r') \wedge M_s \vDash B(r')\}$
$Default\,(\mathcal{P}_s, M_s) = \{not\, A \mid \nexists r \in \mathcal{P}_s : (H(r) = A) \wedge M_s \vDash B(r)\}$

If some literal or conjunction of literals ϕ holds in all stable models of $\mathcal{P}$ at state s, we write $\bigoplus_s \mathcal{P} \vDash \phi$. If s is the largest element of S we simply write $\bigoplus \mathcal{P} \vDash \phi$.

LUPS Here we recall the language of updates *LUPS* closely following its original formulation in [2]. 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 $P_0 \oplus \cdots \oplus P_n$ (i.e. a dynamic logic program), whose semantics corresponds to our knowledge at a given state, produces a sequence with one more program, $P_0 \oplus \cdots \oplus P_n \oplus P_{n+1}$, 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 defined by means of a dynamic logic program generated by the sequence of commands.

In this update framework, knowledge evolves from one knowledge state to another as a result of update commands stated in the object language. Without loss of generality it is assumed that the initial knowledge state is empty. Given the *current knowledge state*, its *successor knowledge state* is produced as a result of the occurrence of a set U of simultaneous *updates*. The knowledge state obtained by performing the sequence of updates $U_1, U_2, \ldots, U_n$ is denoted by $U_1 \otimes U_2 \otimes \cdots \otimes U_n$. 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 update is a set of update commands. Update commands (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 $t \leq n$, where n is the index of the current knowledge state. A query will be denoted by:

$$\textbf{holds}\ L_1, \ldots, L_k\ \textbf{at}\ t?$$

and is true iff the conjunction of its literals holds at the state obtained after the t^{th} update. If $t = n$, the state reference "**at** t" is skipped.

Update commands specify assertions or retractions to the current knowledge state. In *LUPS* a simple assertion is represented by the command:

$$\textbf{assert}\ L \leftarrow L_1, \ldots, L_k\ \textbf{when}\ L_{k+1}, \ldots, L_m \tag{19}$$

Its meaning is that if $L_{k+1}, \ldots, L_m$ is true in the current state, then the rule $L \leftarrow L_1, \ldots, L_k$ is added to its successor state, and persists by inertia, until possibly retracted or overridden by some future update command.

In order to represent rules and facts that do not persist by inertia, i.e. that are one-state events, LUPS includes the modified form of assertion:

$$\textbf{assert event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{20}$$

The retraction of rules is performed with the two update commands:

$$\textbf{retract } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{21}$$

$$\textbf{retract event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{22}$$

Its meaning is that, subject to precondition $L_{k+1}, \ldots, L_m$ (verified at the current state) rule $L \leftarrow L_1, \ldots, L_k$ is either retracted from its successor state onwards, or just temporarily retracted in the successor state (if governed by **event**).

Normally assertions represent newly incoming information. Although its effects remain by inertia (until contravened or retracted), the assert command itself does not persist. However, some update commands may desirably persist in the successive consecutive updates. This is especially the case of laws which, subject to some preconditions, are always valid, or of rules describing the effects of an action. In the former case, the update command must be added to all sets of updates, to guarantee that the rule remains indeed valid. In the latter case, the specification of the effects must be added to all sets of updates, to guarantee that, when the action takes place, its effects are enforced.

To specify such persistent update commands, *LUPS* introduces the following commands:

$$\textbf{always } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{23}$$

$$\textbf{always event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{24}$$

$$\textbf{cancel } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{25}$$

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.

Definition 8 (LUPS). *An update program in LUPS is a finite sequence of updates, where an update is a set of commands of the form (19) to (25).*

The semantics of *LUPS* is defined by incrementally translating update programs into sequences of generalized logic programs and by considering the semantics of the *DLP* formed by them.

Let $\mathcal{U} = U_1 \otimes ... \otimes U_n$ be a *LUPS* programs. At every state t the corresponding *DLP*, $\Upsilon_t(\mathcal{U}) = \mathcal{P}_t$, is determined.

The translation of a *LUPS* program into a dynamic program is made by induction, starting from the empty program P_0, and for each update U_t, given the already built dynamic program $P_0 \oplus \cdots \oplus P_{t-1}$, determining the resulting program $P_0 \oplus \cdots \oplus P_{t-1} \oplus P_t$. To cope with persistent update commands, associated

with every dynamic program in the inductive construction, a set containing all currently active persistent commands is considered, i.e. all those commands that were not cancelled until that point in the construction, from the time they were introduced. To be able to retract rules, a unique identification of each such rule is needed. This is achieved by augmenting the language of the resulting dynamic program with a new propositional variable "$N(R)$" for every rule R appearing in the original *LUPS* program.

Definition 9 (Translation into dynamic logic programs). *Let $\mathcal{U} = U_1 \otimes \cdots \otimes U_n$ be an update program. The corresponding dynamic logic program $\Upsilon(\mathcal{U}) = \mathcal{P} = P_0 \oplus \cdots \oplus P_n$ is obtained by the following inductive construction, using at each step t an auxiliary set of persistent commands PC_t:*

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

Inductive step: Let $\Upsilon_{t-1}(\mathcal{U}) = \mathcal{P}_{t-1} = P_0 \oplus \cdots \oplus P_{t-1}$ with the set of persistent commands PC_{t-1} be the translation of $\mathcal{U}_{t-1} = U_1 \otimes \cdots \otimes U_{t-1}$. The translation of $\mathcal{U}_t = U_1 \otimes \cdots \otimes U_t$ is $\Upsilon_t(\mathcal{U}) = \mathcal{P}_t = P_0 \oplus \cdots \oplus P_{t-1} \oplus P_t$ with the set of persistent commands PC_t, where:

$$
\begin{aligned}
&PC_t = PC_{t-1} \cup \{\textbf{assert } R \textbf{ when } \phi : \textbf{always } R \textbf{ when } \phi \in U_t\} \cup \\
&\cup \{\textbf{assert event } R \textbf{ when } \phi : \textbf{always event } R \textbf{ when } \phi \in U_t\} \cup \\
&- \{\textbf{assert } [\textbf{event}]\ R \textbf{ when } \phi : \textbf{cancel } R \textbf{ when } \psi \in U_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \psi\} - \\
&- \{\textbf{assert } [\textbf{event}]\ R \textbf{ when } \phi : \textbf{retract } R \textbf{ when } \psi \in U_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \psi\} -
\end{aligned}
$$

$$NU_t = U_t \cup PC_t$$

$$
\begin{aligned}
P_t = \{&N(R) \leftarrow; H(R) \leftarrow B(R), N(R) : \textbf{assert } [\textbf{event}]\ R \textbf{ when } \phi \in NU_t \wedge \\
&\wedge \bigoplus \mathcal{P}_{t-1} \vDash \phi\} \cup \\
&\cup \{not\ N(R) \leftarrow : \textbf{retract } [\textbf{event}]\ R \textbf{ when } \phi \in NU_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \phi\} \cup \\
&\cup \{not\ N(R) \leftarrow : \textbf{assert event } R \textbf{ when } \phi \in NU_{t-1} \wedge \bigoplus \mathcal{P}_{t-2} \vDash \phi\} \cup \\
&\cup \{N(R) \leftarrow : \textbf{retract event } R \textbf{ when } \phi \in NU_{t-1} \wedge \bigoplus \mathcal{P}_{t-2} \vDash \phi, N(R)\}
\end{aligned}
$$

Definition 10 (LUPS Semantics). *Let $\mathcal{U}$ be an update program. A query*

$$\textbf{holds } L_1, \ldots, L_n \textbf{ at } t$$

is true in $\mathcal{U}$ iff $\bigoplus_t \Upsilon(\mathcal{U}) \vDash L_1, \ldots, L_n$.