Linköping Electronic Articles in
Computer and Information Science
Vol. 2(1997): nr 18

# Dynamic Logic Programming

José Júlio Alferes
João A. Leite
Luís Moniz Pereira
Halina Przymusinska
Teodor C. Przymusinski

Linköping University Electronic Press
Linköping, Sweden

http://www.ep.liu.se/ea/cis/1997/018/

*Published on December 12, 1997 by*
*Linköping University Electronic Press*
*581 83 Linköping, Sweden*

**Linköping Electronic Articles in**
**Computer and Information Science**
*ISSN 1401-9841*
*Series editor: Erik Sandewall*


*Typeset by the authors using LaTeX*
*Formatted using* étendu *style*

**Recommended citation:**
*<Authors>. <Title>. Linköping Electronic Articles in Computer and Information Science, Vol. 2(1997): nr 18.*
http://www.ep.liu.se/ea/cis/1997/018/. *December 12, 1997.*

*This URL will also contain a link to the authors' home pages.*

**Abstract:**

In this paper we investigate updates of knowledge bases represented by logic programs. In order to represent negative information, we use generalized logic programs which allow default negation not only in their bodies but also in their heads.

We start by introducing the notion of an update $P \oplus U$ of a logic program $P$ by another logic program $U$. Subsequently, we provide a precise semantic characterization of $P \oplus U$, and study some basic properties of program updates. In particular, we show that our update programs generalize the notion of interpretation update.

We then extend this notion to sequences of logic programs updates $P_1 \oplus P_2 \oplus \ldots$, defining dynamic program updates, thereby introducing the paradigm of dynamic logic programming. This paradigm significantly facilitates modularization of logic programming, and thus modularization of non-monotonic reasoning as a whole.

Suppose that we are given a set of logic program modules, each describing a different state of our knowledge of the world. Different states may represent different time points or different sets of priorities or perhaps even different viewpoints. Consequently, program modules may contain mutually contradictory as well as overlapping information. The role of the dynamic program update is to use the mutual relationships existing between different states to precisely determine, at any given state, the declarative and the procedural semantics of the combined program, resulting from all these modules.

**Present addresses of the authors:**

**Jose Julio Alferes:** Dep. Matematica, Univ. Évora and A.I. Centre, Univ. Nova de Lisboa, 2825 Monte da Caparica, Portugal. Email: jja@di.fct.unl.pt

**Joao A. Leite:** A.I. Centre, Univ. Nova de Lisboa, 2825 Monte da Caparica, Portugal. Email: jleite@di.fct.unl.pt

**Luis Moniz Pereira:** A.I. Centre, Univ. Nova de Lisboa, 2825 Monte da Caparica, Portugal. Email: lmp@di.fct.unl.pt

**Halina Przymusinska:** Department of Computer Science, California State Polytechnic University, Pomona, CA 91768, USA. Email: halina@cs.ucr.edu

**Teodor C. Przymusinski:** Department of Computer Science, University of California, Riverside, CA 92521, USA. Email: teodor@cs.ucr.edu

**Keywords:**

Logic Programming, Nonmonotonic Logics, Updates.

# 1 Introduction

Most of the work done so far in the field of logic programming has focused on representing static worlds i.e. worlds that do not change. This is a serious drawback when attempting to deal with knowledge bases representing a dynamic world by means of logic programs, in which not only the extentional part of the program (i.e. the facts) but also the intensional part (i.e. the rules) may change.

In this paper we investigate updates of knowledge bases represented by logic programs. In order to represent negative information, we use generalized logic programs which allow default negation not only in their bodies but also in their heads. This is, in particular, needed in order to specify that some atoms should became false, i.e., should be deleted. However, our logic program updates allow much more than mere insertion and deletion of facts.

Several authors have addressed the issue of updates of logic programs [9, 10, 1], most of them following the so called "interpretation update" approach, originally proposed in [11, 5]. According to this approach, a knowledge base $DB'$ is considered the update of a knowledge base $DB$ if the set of its models coincides with the set of individually updated models of $DB$. This approach allows us to reduce the issue of knowledge base updates to the issue of finding updates of individual interpretations (models). In order to determine an update of a given interpretation $M$, we typically change the status of all literals in $M$ that are affected by the updating rules $U$ while keeping all the other literals intact by inertia (see e.g. [9, 10]).

While such an approach may be justified when only the *extensional* part of the knowledge base (i.e., individual facts) is being updated, as the following example shows, it leads to strongly counter-intuitive results when also the intensional part of the database (i.e., program rules) undergoes change.

**Example 1.1** *Given the logic program*

$$P_1: \quad sleep \leftarrow not\ tv\_on \qquad tv\_on \leftarrow$$
$$watch\_tv \leftarrow tv\_on$$

*whose only stable model is* $M_1 = \{tv\_on, watch\_tv\}$. *Consider the update stating that there is a power failure and if there is a power failure then the tv is no longer on, represented by the logic program:*

$$P_2: \quad not\ tv\_on \leftarrow power\_failure \qquad power\_failure \leftarrow$$

*According to the above mentioned approach to updating, we would obtain* $M_2 = \{power\_failure, watch\_tv\}$ *as the only update of* $M_1$ *by* $P_2$. *As a results, even though there is a power failure, we are still watching tv. Indeed, all we need to do in order to satisfy update* $M_1$ *is to add "*$power\_failure$*" and delete "*$tv\_on$*". However, by inspecting the initial program and the updating rules, we are likely to argue that*

*since "watch_tv" was true because "tv_on" was also true, the removal of "tv_on" should make "watch_tv" false by default. Moreover, one would expect "sleep" to become true. Presumably, the intended model of the update of $P_1$ by $P_2$ is the model $M_2' = \{power\_failure, sleep\}$.*

*Suppose now that an another update $P_2$, described by the logic program:*

$$P_3: \quad not\ power\_failure \leftarrow$$

*follows, stating that power is back up again. We should now expect the tv to be on again. Since the power was restored, i.e. "$power\_failure$" is false, the rule "$not\ tv\_on \leftarrow power\_failure$" of $P_2$ should have no effect and the truth value of "$tv\_on$" should be obtained from the rule "$tv\_on \leftarrow$ " of $P_1$.* □

This example illustrates that, when updating programs, it is insufficient to consider which is, by inertia, the truth value of some literal figuring in the head of a rule. Indeed, the truth value of the rule body may be affected by the updating of other literals.

This approach was first adopted in [7], where the authors advocate that the principle of inertia should be applied to the rules of the initial program rather than to the individual literals in a model. A transformation which, given an initial program and an update program, produces an updated program was presented. Although an important result, this transformation falls short in what the exact embedding of model updates is concerned, as shown in the following example:

**Example 1.2** *Consider the initial program $P = \{tv\_on \leftarrow\}$ whose only stable model is $M_1 = \{tv\_on\}$, and the update program $U = \{not\ tv\_on \leftarrow not\ tv\_on\}$. According to [7] $I_1 = \{tv\_on\}$ and $I_2 = \{\}$ are the models of the update of $P$ by $U$. Since $P$ is a program consisting of facts, the result of updating $P$ by $U$ should be the same as updating $M_1$ by $U$, in which case only $I_1 = \{tv\_on\}$ would be accepted according to [9]. Note that $I_2$ violates the desirable principle of minimal change.* □

In this paper we define program updates in a way that properly extends model updates introduced in [9] (or their reformulation in the language of logic programs by [10]), i.e. if the initial program is just a set of facts then program updates and model updates coincide.

We also extend the notion of updates to sequences of programs, defining dynamic program updates. Suppose that we are given a set of program modules each describing a different state of our knowledge of the world. Different states may represent different time periods or different sets of priorities or perhaps even different viewpoints. The role of dynamic program update is to use the mutual relationships existing between different states (and specified in the form of an ordering relation) to precisely determine, at any given state, the *declarative* as well as the *procedural* semantics of the combined program,

composed of all these modules. This allows us to specify the meaning of a world representation incrementally subject to successive program updates occurring in different states of the world.

In section 2 we define the language of generalized logic programs (with default negation in their heads) and describe its stable model semantics as a special case of the approach proposed in [8]. When restricted to the language we are considering, our definition, while different, is equivalent to the one proposed in [8].

In section 3 we define program update $P \oplus U$, characterize its semantics in section 4, and elaborate on some of its properties in section 5. In section 6, the dynamic program update is set forth, a notion that supports the important paradigm of *dynamic logic programming*. Whereas traditional logic programming has concerned itself mostly with static knowledge, we show here how to successively update such knowledge by a rather general form of logic program. In short, we show how to update one program with another and how this process can be iterated.

## 2 Generalized Logic Programs and their Stable Models

In order to allow for programs and updating programs with *negative* information, we will use logic programs which allow default negation $not\, A$ not only in premises of their clauses but also in their heads. We call such programs *generalized logic programs*. This class differs from the class of programs with the so called "classical" or strong negation [4]. This section describes the notation used and extends the stable model semantics [3] of normal programs to generalized logic programs[1].

In order to syntactically represent generalized logic programs as *propositional theories*, we use propositional languages which include propositional variables (atoms) of the form $not\ A$. Suppose that $\mathcal{K}$ is an arbitrary set of propositional symbols 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 $\mathcal{L}$ whose set of propositional variables consists of:

$$\{A : A \in \mathcal{K}\} \cup \{not\ A : A \in \mathcal{K}\}.$$

Atoms $A$, $A \in \mathcal{K}$, are called *objective atoms* while the atoms $not\ A$, $A \in \mathcal{K}$, are called *default atoms*. From the definition it follows that the two sets are disjoint.

Suppose that $\mathcal{K}$ is an arbitrary set of propositional symbols and let $\mathcal{L} = \mathcal{L}_\mathcal{K}$ be the language generated by $\mathcal{K}$. By a *(2-valued) interpretation* $M$ of $\mathcal{L}_\mathcal{K}$ we mean any set of atoms from $\mathcal{L}_\mathcal{K}$ that satisfies the condition that for any $A$ in $\mathcal{K}$, precisely one of the atoms $A$ or $not\ A$ belongs to $M$. Given $M$ we define:

[1] In a forthcoming paper we extend our results to 3-valued (partial) models of logic programs, and, in particular, to well-founded models.

$$M^+ = \{A \in \mathcal{K} : A \in M\}$$

$$M^- = \{not\ \ A : not\ \ A \in M\} = \{\ not\, A : A \notin M\}.$$

By a *generalized logic program* $P$ in the language $\mathcal{L}_\mathcal{K}$ we mean a finite or infinite set of propositional Horn clauses of the form:

$$L \leftarrow L_1, \ldots, L_n$$

where $L$ and $L_i$ are atoms from $\mathcal{L}_\mathcal{K}$. If all the atoms $L$ appearing in heads of clauses of $P$ are objective atoms, then we say that the logic program $P$ is *normal*. Consequently, from a syntactic standpoint a logic program is simply viewed as a propositional Horn theory. However, its semantics significantly differs from the semantics of classical propositional theories and is determined by the class of stable models defined below.

**Definition 2.1 (Stable models of generalized logic programs)**

*We say that 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 $P \cup M^-$:*

$$M = Least(P \cup M^-),$$

*or, equivalently, $M = \{L : L$ is an atom and $P \cup M^- \vdash L\}$ .* □

**Example 2.1** Consider the program:

$$\begin{array}{lll} a \leftarrow not\ \ b & c \leftarrow b & e \leftarrow not\ \ d \\ not\ \ d \leftarrow not\ \ c, a & d \leftarrow not\ \ e & \end{array}$$

and assume $\mathcal{K} = \{a, b, c, d, e\}$. This program has precisely one stable model $M = \{a, e, not\ \ b, not\ \ c, not\ \ d\}$. To see that $M$ is stable we simply observe that:

$$M = Least(P \cup \{not\ \ b, not\ \ c, not\, d\}).$$

The interpretation $N = \{not\ \ a, not\, e, b, c, d\}$ is not a stable model because:

$$N \neq Least(P \cup \{not\ \ e, not\ \ a\}).$$ □

The following Proposition easily follows from the definition of stable models.

**Proposition 2.1** *The class of stable models of generalized logic programs extends the class of stable models of normal programs [3].* □

# 3 Program Updates

Suppose again that $\mathcal{K}$ is an arbitrary set of propositional variables, and $P$ and $U$ are two generalized logic programs in the language $\mathcal{L} = \mathcal{L}_{\mathcal{K}}$. By $\widehat{\mathcal{K}}$ we denote the following superset of $\mathcal{K}$:

$$\widehat{\mathcal{K}} = \mathcal{K} \cup \{A^- : A \in \mathcal{K}\} \cup \{A_P,\ A_P^- : A \in \mathcal{K}\} \cup \{A_U,\ A_U^- : A \in \mathcal{K}\}.$$

This definition assumes that the original set $\mathcal{K}$ of propositional variables does not contain any of the newly added symbols of the form $A^-$, $A_P$, $A_P^-$, $A_U$, $A_U^-$ so that they are all disjoint sets of symbols. If $\mathcal{K}$ contains any such symbols then they have to be renamed before the expansion of $\mathcal{K}$ takes place. We denote by $\widehat{\mathcal{L}} = \mathcal{L}_{\widehat{\mathcal{K}}}$ the extension of the language $\mathcal{L} = \mathcal{L}_{\mathcal{K}}$ generated by $\widehat{\mathcal{K}}$.

**Definition 3.1 ( Program Updates)** *Let $P$ and $U$ be generalized programs in the language $\mathcal{L}P$. We call $P$ the original program and $U$ the updating program. By the update of $P$ by $U$ we mean the generalized logic program $P \oplus U$, which consists of the following clauses in the extended language $\widehat{\mathcal{L}}$:*

**(RP) Rewritten original program clauses:**

$$A_P \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \quad (1)$$
$$A_P^- \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \quad (2)$$

*for any clause*

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

*respectively, for any clause*

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

*in the original program $P$.*

**(RU) Rewritten updating program clauses:**

$$A_U \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \quad (3)$$
$$A_U^- \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \quad (4)$$

*for any clause*

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

*respectively, for any clause*

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

*in the updating program $U$.*

**(UR) Update rules:**

$$A \leftarrow A_U; \qquad A^- \leftarrow A_U^- \tag{5}$$

*for all objective atoms $A \in \mathcal{K}$. The update rules state that an atom $A$ in $P \oplus U$ must be true (respectively, false) if it is true (respectively, false) in the updating program $U$.*

**(IR) Inheritance rules:**

$$A \leftarrow A_P, not\ A_U^- \qquad A^- \leftarrow A_P^-, not\ A_U \tag{6}$$

*for all objective atoms $A \in \mathcal{K}$. The inheritance rules say that an atom $A$ (respectively, $A^-$) in $P \oplus U$ can be inherited (by inertia) from the original program $P$ provided it is not forced to be false by the updating program $U$.*

**(DR) Default rules:**

$$A^- \leftarrow not\ A_P, not\ A_U; \qquad not\ A \leftarrow A^- \tag{7}$$

*for all objective atoms $A \in \mathcal{K}$. The first default rule states that an atom $A$ in $P \oplus U$ is false if it is neither true in the original program $P$ nor in the updating program $U$. The second says that if an atom is false then it can be assumed to be false by default. It ensures that $A$ and $A^-$ cannot both be true.* □

It is easy to show that any model $N$ of $P \oplus U$ is *coherent*, i.e., $A$ is true (respectively, false) in $N$ iff $A^-$ is false (respectively, true) in $N$, for any $A \in \mathcal{K}$. In other words, every stable model of $P \oplus U$ satisfies the constraint $not\ A \equiv A^-$. So $A^-$ can simply be regarded as a (internal) representation for the default negation of $A$.

**Example 3.1** Consider the programs $P_1$ and $P_2$ from example 1.1:

$$P_1: \begin{array}{l} sleep \leftarrow not\ tv\_on \\ watch\_tv \leftarrow tv\_on \\ tv\_on \leftarrow \end{array} \qquad P_2: \begin{array}{l} not\ tv\_on \leftarrow power\_failure \\ power\_failure \leftarrow \end{array}$$

The update of the program $P_1$ by the program $P_2$ is the logic program $P_1 \oplus P_2 = (RP_1) \cup (RP_2) \cup (UR) \cup (IR) \cup (DR)$ where

$$RP_1: \begin{array}{l} sleep_P \leftarrow tv\_on^- \\ watch\_tv_P \leftarrow tv\_on \\ tv\_on_P \leftarrow \end{array} \qquad RP_2: \begin{array}{l} tv\_on_U^- \leftarrow power\_failure \\ power\_failure_U \leftarrow \end{array}$$

whose only stable model (modulo irrelevant literals) is:

$$M = \{power\_failure, sleep\}$$

□

# 4 Semantic Characterization of Program Updates

In this section we provide a complete semantic characterization of update programs $P \oplus U$ by describing stable models of the update program $P \oplus U$. This characterization precisely shows how the semantics of an update program $P \oplus U$ depends on the syntax and semantics of the programs $P$ and $U$.

We begin by defining a natural extension of an arbitrary interpretation $M$ of the language $\mathcal{L}$ to an interpretation $\widehat{M}$ of the extended language $\widehat{\mathcal{L}}$.

**Definition 4.1 (Extended Interpretations)** *For any interpretation $M$ of $\mathcal{L}$ we denote by $\widehat{M}$ its extension to an interpretation of the extended language $\widehat{\mathcal{L}}$ defined, for any atom $A \in \mathcal{K}$, by the following rules:*

$$A^- \in \widehat{M} \text{ iff } not\ A \in M$$

$$A_P \in \widehat{M} \text{ iff there exists } A \leftarrow Body \text{ in } P \text{ and } M \models Body$$

$$A_P^- \in \widehat{M} \text{ iff there exists } not\ A \leftarrow Body \text{ in } P \text{ and } M \models Body$$

$$A_U \in \widehat{M} \text{ iff there exists } A \leftarrow Body \text{ in } P \text{ and } M \models Body$$

$$A_U^- \in \widehat{M} \text{ iff there exists } not\ A \leftarrow Body \text{ in } P \text{ and } M \models Body. \ \square$$

**Definition 4.2** *For any interpretation $M$ of $\mathcal{L}$ which is a model of $U$ define:*

$$\begin{aligned}
Defaults[M] = \ & \{not\ A : M \models \neg Body, \\
& \quad \text{for every clause } A \leftarrow Body \in P \cup U\}; \\
Rejected[M] = \ & \{A \leftarrow Body \in P : \exists\ not\ A \leftarrow Body' \in U \text{ and} \\
& \quad M \models Body'\} \cup \\
& \{not\ A \leftarrow Body \in P : \exists\ A \leftarrow Body' \in U \text{ and} \\
& \quad M \models Body'\}; \\
Residue[M] = \ & P \cup U - Rejected[M]. \qquad \square
\end{aligned}$$

The set $Defaults[M]$ contains default negations $not\ A$ of all *unsupported* atoms $A$. Consequently, negation $not\, A$ of these unsupported atoms $A$ can be assumed by default. The set $Rejected[M] \subseteq P$ represents the set of clauses of the original program $P$ that are *rejected* (or contradicted) by the update program $U$ and its model M. The residue $Residue[M]$ consists of all clauses in the union $P \cup U$ of programs $P$ and $U$ that were *not* rejected by the update program $U$.

Now we are able to give a description of the semantics of an update program $P \oplus U$ by providing a complete characterization of its stable models.

**Theorem 4.1 (Characterization of stable models of update programs)** *An interpretation $N$ of the language $\widehat{\mathcal{L}} = \mathcal{L}_{\widehat{\mathcal{K}}}$ is a stable model of*

*the update $P \oplus U$ if and only if $N$ is an extension $N = \widehat{M}$ of a model $M$ of $U$ that satisfies the condition:*

$$M = Least(P \ \cup \ U - Rejected[M] \ \cup \ Defaults[M]),$$

*or, equivalently, $M = Least(Residue[M] \ \cup \ Defaults[M])$.* □

**Example 4.1** Consider again the programs $P_1$ and $P_2$ from example 1.1:

$$P_1: \begin{array}{l} sleep \leftarrow not\ tv\_on \\ watch\_tv \leftarrow tv\_on \\ tv\_on \leftarrow \end{array} \qquad P_2: \begin{array}{l} not\ tv\_on \leftarrow power\_failure \\ power\_failure \leftarrow \end{array}$$

Let $M = \{power\_failure, sleep\}$. We then have that:

$$Defaults[M] = \{not\ \ watch\_tv, \} \qquad Rejected[M] = \{tv\_on \leftarrow\}$$

$$Residue[M] \ = \left\{ \begin{array}{l} sleep \leftarrow not\ tv\_on \\ watch\_tv \leftarrow tv\_on \\ not\ tv\_on \leftarrow power\_failure \\ power\_failure \leftarrow \end{array} \right\}$$

it is easy to see that $M = Least(Residue[M] \ \cup \ Defaults[M])$ . □

# 5 Properties of Program Updates

In this section we study basic properties of program updates. Since $Defaults[M] \subseteq M^-$, we conclude that the condition

$$M = Least(Residue[M] \cup Defaults[M])$$

clearly implies $M = Least(Residue[M] \cup M^-)$ and thus we immediately obtain:

**Proposition 5.1** *If $N$ is a stable model of $P \oplus U$ then its restriction $M = N|\mathcal{L}$ is a stable model of $Residue[M]$.* □

However, the condition $M = Least(Residue[M] \cup Defaults[M])$ says much more than just that $M$ is a stable model of $Residue[M]$. It says that $M$ is completely determined by the set of atoms $Defaults[M]$, i.e., by the set of negations of unsupported atoms that can be assumed to be false by default. Also, if $M$ is a stable model of $P \cup U$ then clearly $Rejected[M] = \emptyset$ and $Defaults[M] = M^-$ which implies:

**Proposition 5.2** *If $M$ is a stable model of the union $P \cup U$ of programs $P$ and $U$ then its extension $N = \widehat{M}$ is a stable model of the update program $P \oplus U$. Thus, the semantics of the update program $P \oplus U$ is always weaker than or equal to the semantics of the union $P \cup U$ of programs $P$ and $U$ .* □

In general, the converse of the above result does not hold. In particular, the union $P \cup U$ may be a contradictory program with no stable models.

**Example 5.1** Consider again the programs $P_1$ and $P_2$ from example 1.1. It is trivial to see that $P \cup U$ is contradictory. □

If either $P$ or $U$ is empty and $M$ is a stable model of $P \cup U$ then $Rejected[M] = \emptyset$ and therefore $M$ is also a stable model of $P \oplus U$.

**Proposition 5.3** *If either $P$ or $U$ is empty then $M$ is a stable model of $P \cup U$ iff $N = \widehat{M}$ is a stable model of $P \oplus U$. Thus, in this case, the semantics of the update program $P \oplus U$ coincides with the semantics of the union $P \cup U$.* □

**Proposition 5.4** *If both $P$ and $U$ are normal programs (or if both have only clauses with negated atoms not $A$ in their heads) then $M$ is a stable model of $P \cup U$ iff $N = \widehat{M}$ is a stable model of $P \oplus U$. Thus, in this case the semantics of the update program $P \oplus U$ also coincides with the semantics of the union $P \cup U$ of programs $P$ and $U$.* □

## 5.1 Program Updates Generalize Interpretation Updates

*Interpretation updates*, originally introduced by Marek and Truszczynski [9], under the name "program revisions", and subsequently given a simpler characterization by Przymusinski and Turner [10], constitute a special case of program updates. We identify the *"revision rules"*:

$$in(A) \leftarrow in(B), out(C) \quad \text{(respectively, } out(A) \leftarrow in(B), out(C)\text{)},$$

introduced in [9], with the standard generalized logic program clauses:

$$A \leftarrow B, not\ C \quad \text{(respectively, } not\ A \leftarrow B, not\ C\text{)}.$$

**Theorem 5.1 (Program updates generalize interpretation updates)** *Let $I$ be any interpretation and $U$ any updating program in the language $\mathcal{L}$. Denote by $P_I$ the generalized logic program in $\mathcal{L}$: $P_I = \{A \leftarrow : A \in I\} \cup \{not\ A \leftarrow : not\ A \in I\}$.*

*Then $\widehat{J}$ is a stable model of the program update $P_I \oplus U$ of the program $P_I$ by the program $U$ iff $J$ is an interpretation update of $I$ by $U$ (in the sense of [9]).* □

**Remark 5.1** It is easy to see that, optionally, we could include only positive rules $A \leftarrow$ in the program $P_I$ thus making it a normal program. □

**Example 5.2** Let $P_I = \{tv\_on \leftarrow\ \}$ corresponding to the interpretation $I = \{tv\_on\}$, and the update program

$$U = \{not\ \ tv\_on \leftarrow not tv\_on\}$$

(as in ex. 1.2). The only stable model of $P_I \oplus U$ is $I = \{tv\_on\}$ which corresponds to the only interpretation update of $I$ by $U$. □

### 5.2 Adding Strong Negation

We now show how to represent *strong negation* $-A$ [2] in generalized logic programs. This shows that the class of generalized logic programs is actually *broader* than the class of logic programs with strong negation. It also allows us to update logic programs with strong negation and to use strong negation in updating programs.

**Definition 5.1 (Adding strong negation)** *Let $\mathcal{K}$ be an arbitrary set of propositional variables. In order to add strong negation to the language $\mathcal{L} = \mathcal{L}_\mathcal{K}$ we just augment the set $\mathcal{K}$ with new propositional symbols $\{-A : A \in \mathcal{K}\}$, obtaining the new set $\mathcal{K}^*$, and consider the extended language $\mathcal{L}^* = \mathcal{L}_{\mathcal{K}^*}$. In order to ensure that $A$ and $-A$ cannot both be true we also assume, for all $A \in \mathcal{K}$, the following strong negation rules, which are themselves generalized logic program clauses:*

$$(SN) \qquad not\ \ A \leftarrow\ -A. \quad \square$$

**Remark 5.2** In order to prevent the strong negation rules (SN) from being inadvertently overruled by the updating program $U$, one may want to make them always part of the most current updating program. □

## 6 Dynamic Program Updates

In this section we itroduce the notion of *dynamic program update*

$$\bigoplus \{\ P_s : s \in S\}$$

of an ordered set $\mathcal{P} = \{\ P_s : s \in S\}$ of programs which provides an important generalization of the notion of program updates.

The idea of dynamic updates, inspired by [6], is very simple and quite fundamental. Suppose that we are given a set of program modules $P_s$, indexed by different states of the world $s$. Each program $P_s$ contains some knowledge that is supposed to be true at the state $s$. Different states may represent different time periods or different sets of priorities or perhaps even different viewpoints. Consequently, the individual program modules may contain mutually contradictory as well as overlapping information. The rôle of the dynamic program update $\bigoplus \{P_s : s \in S\}$ is to use the mutual relationships existing between different states (and specified in the form of the ordering relation) to precisely determine, at any given state $s$, the *declarative* as

well as the *procedural* semantics of the combined program, composed of all modules.

Consequently, the notion of a dynamic program update supports the important paradigm of *dynamic logic programming*. Given individual and largely *independent* program modules $P_s$ describing our knowledge at different states of the world (for example, the knowledge acquired at different times), the dynamic program update $\bigoplus \{P_s \ : \ s \ \in S \ \}$ specifies the exact meaning of the union of these programs. Dynamic programming significantly facilitates modularization of logic programming and, thus, modularization of non-monotonic reasoning as a whole.

Suppose that $\mathcal{P} = \{P_s : s \in S\}$ is a finite or infinite sequence of generalized logic programs in the language $\mathcal{L} = \mathcal{L}_\mathcal{K}$, indexed by the set $S = \{1, \ldots, n, \ldots\}$ . We will call elements $s$ of the set $S \cup \{0\}$ *states* and we will refer to 0 as the *initial state*.

**Remark 6.1** Instead of a linear sequence of states $S \cup \{0\}$ one can as well consider any finite or infinite ordered set with the smallest element $s_0$ and with the property that every state $s$ other than $s_0$ has an immediate predecessor $s - 1$ and that $s_0 = s - n$, for some finite $n$. In particular, one may use a finite or infinite tree with the root $s_0$ and the property that every node (state) has only a finite number of ancestors. □

By $\overline{\mathcal{K}}$ we denote the following superset of the set $\mathcal{K}$ of propositional variables:

$$\begin{aligned}\overline{\mathcal{K}} \ = \ & \mathcal{K} \cup \{A^- : A \in \mathcal{K}\} \cup \ \{A_s, \ A_s^- : A \in \mathcal{K}, \ s \in S \cup \{0\}\} \cup \\ & \cup \{A_{P_s}, \ A_{P_s}^- : A \in \mathcal{K}, \ s \in S\}.\end{aligned}$$

We denote by $\overline{\mathcal{L}} = \mathcal{L}_{\overline{\mathcal{K}}}$ the extension of the language $\mathcal{L} = \mathcal{L}_\mathcal{K}$ generated by $\overline{\mathcal{K}}$.

**Definition 6.1 (Dynamic Program Updates)** *By the dynamic program update over the sequence of updating programs $\mathcal{P} = \{P_s : s \in S\}$ we mean the logic program $\bigoplus \mathcal{P}$, which consists of the following clauses in the extended language $\overline{\mathcal{L}}$:*

**(RP) Rewritten program clauses:**

$$A_{P_s} \ \leftarrow \ B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \qquad (8)$$

$$A_{P_s}^- \ \leftarrow \ B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \qquad (9)$$

*for any clause $A \leftarrow B_1, \ldots, B_m, not\, C_1, \ldots, not\ C_n$ (respectively, for any clause not $A \leftarrow B_1, \ldots, B_m$, not $C_1, \ldots, not\ C_n$) in the program $P_s$, where $s \in S$.*

**(UR) Update rules:**

$$A_s \leftarrow A_{P_s} \qquad A_s^- \leftarrow A_{P_s}^- \qquad (10)$$

*for all objective atoms $A \in \mathcal{K}$ and for all $s \in S$.*

**(IR) Inheritance rules:**

$$A_s \leftarrow A_{s-1}, not\ \ A^-_{P_s} \qquad A^-_s \leftarrow A^-_{s-1}, not\ \ A_{P_s} \tag{11}$$

*for all objective atoms $A \in \mathcal{K}$ and for all $s \in S$.*

**(DR) Default rules (describing the initial state 0):**

$$A^-_0, \tag{12}$$

*for all objective atoms $A \in \mathcal{K}$. Default rules describe the initial state 0 by making all objective atoms initially false.* □

**Remark 6.2** Observe that the dynamic program update $\bigoplus \mathcal{P}$ is a normal logic program, i.e., it does not contain default negation in heads of its clauses. Moreover, only the inheritance rules contain default negation in their bodies. Also note that the program $\bigoplus \mathcal{P}$ does not contain atoms $A$ or $A^-$, where $A \in \mathcal{K}$, in heads of its clauses. These atoms appear only in the bodies of rewritten program clauses. □

**Definition 6.2 (Iterated 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 $\bigoplus \mathcal{P}$ augmented with the following:*

**(CS) Current state rules:**

$$A \leftarrow A_s \qquad A^- \leftarrow A^-_s \qquad not\, A \leftarrow A^- \tag{13}$$

*for all objective atoms $A \in \mathcal{K}$. Current state rules specify the current state in which the updated program is being considered.* □

The notion of a dynamic program update generalizes the previously introduced notion of an update $P \oplus U$ of two programs $P$ and $U$.

**Theorem 6.1** *Let $P$ and $U$ be arbitrary generalized logic programs, let $S = \{1, 2\}$ and let $P_1 = P$ and $P_2 = U$. The dynamic program update $\bigoplus_2 \{P_1, P_2\}$ at state $s = 2$ is semantically equivalent to the program update $P \oplus U$ .* □

**Example 6.1** Let $\mathcal{P} = \{P_1, P_2, P_3\}$ be the ordered set of programs of ex. 1.1:

$$\begin{array}{lll} P_1: & sleep \leftarrow not\ tv\_on & P_2: \quad not\ tv\_on \leftarrow power\_failure \\ & watch\_tv \leftarrow tv\_on & \qquad\qquad power\_failure \leftarrow \\ & tv\_on \leftarrow & P_3: \quad not\ power\_failure \leftarrow \end{array}$$

the dynamic program update over $\mathcal{P}$ is the logic program $\bigoplus \mathcal{P} = (RP_1) \cup (RP_2) \cup (RP_3) \cup (UR) \cup (IR) \cup (DR)$ where

$$\begin{array}{lll} RP_1: & sleep_{P_1} \leftarrow tv\_on^- & RP_2: \quad tv\_on^-_{P_2} \leftarrow power\_failure \\ & watch\_tv_{P_1} \leftarrow tv\_on & \qquad\qquad power\_failure_{P_2} \leftarrow \\ & tv\_on_{P_1} \leftarrow & RP_3: \quad power\_failure^-_{P_3} \leftarrow \end{array}$$

and the dynamic program update at the state $s$ is $\bigoplus_s \mathcal{P} = \bigoplus \mathcal{P} \cup (CS)$ so that $\bigoplus_1 \mathcal{P}$ has the single stable model $M_1 = \{tv\_on, watch\_tv\}$; $\bigoplus_2 \mathcal{P}$ has the single stable model $M_2 = \{sleep, power\_failure\}$ and $\bigoplus_3 \mathcal{P}$ has the single stable model $M_3 = \{tv\_on, watch\_tv\}$ (all models modulo irrelevant literals). Notice that $\bigoplus_2 \mathcal{P} = P_1 \oplus P_2$ . □

# 7 Conclusions and Future Work

We've defined a program transformation that takes two logic programs $P$ and $U$, and produces the logic program resulting of updating program $P$ by $U$. We've characterized the semantics of program updates and presented some properties. This alone is a generalization of the results of [9]. In fact, we've shown that, for the special case where the initial program is just a set of facts, our program updates coincide with the justified revisions of [9]. When the initial program also contains rules, our program updates precisely characterize which of those rules should remain valid by inertia, and which should be rejected. We've also shown how strong or "classical" negation can be easily incorporated in generalized logic programs, and thus encompass their updating too.

With dynamic program updates, we've extended program updates to ordered sets of logic programs (or modules). When this order is interpreted as a time order, dynamic program updates describe the evolution of a logic program which is subject to a sequence of modifications. This opens up the possibility for incremental design and evolution of logic programs (which we've called dynamic logic programming). In general, we believe that dynamic programming significantly facilitates modularization of logic programming and, thus, as non-monotonic reasoning as a whole.

An application of dynamic logic programming that we intend to explore, is that of evolution and maintenance of software specifications. In fact, by using a logic programming as a specification language, dynamic programming provides the means to describe the evolution of software specifications.

Ordered sets of program modules need not necessarily be seen as the evolution in time of a logic program. Different modules can also represent different sets of priorities, or viewpoints of different agents. In the case of priorities, a dynamic program update specifies the exact meaning of the merge of the modules, according to the priorities. In this respect we intend to compare dynamic logic programming to other approaches to preferences in knowledge representation.

Although not explored here, a dynamic program update can be queried not only about the current state but also about other states. If modules are seen as viewpoints of different agents, the truth of some $A_s$ in $\bigoplus \mathcal{P}$ can be read as: $A$ is true according to agent $s$ in a situation where the knowledge of the $\bigoplus \mathcal{P}$ is "visible" to agent $s$.

Our results and approach are being generalized to the 3-valued

case, so as to update programs under the well-founded semantics. We have at present a working implementation for the 3-valued case with top-down querying[2].

Finally, our approach to programming updating has grown out of our research in non-monotonic reasoning in logic programming. We envisage enriching it with other dynamic programming features in the near future, such as abduction and contradiction removal. Other applications we intend to study include modelling productions systems, reasoning about actions, temporal databases, among others.

## References

[1] J. J. Alferes, L. M. Pereira. *Update-programs can update programs.* In J. Dix, L. M. Pereira and T. Przymusinski, editors, Selected papers from the ICLP'96 ws NMELP'96, vol. 1216 of LNAI, pages 110-131. Springer-Verlag, 1997.

[2] J. J. Alferes, L. M. Pereira and T. Przymusinski. *Strong and Explicit Negation in Non-Monotonic Reasoning and Logic Programming.* In J. J. Alferes, L. M. Pereira and E. Orlowska, editors, JELIA '96, volume 1126 of LNAI, pages 143-163. Springer-Verlag, 1996.

[3] M. Gelfond and V. Lifschitz. *The stable model semantics for logic programming.* In R. Kowalski and K. A. Bowen. editors. 5th Int. Conf. on LP, pages 1070-1080. MIT Press, 1988.

[4] M. Gelfond and V. Lifschitz. *Logic Programs with classical negation.* In Warren and Szeredi, editors, 7th Int. Conf. on LP, pages 579-597. MIT Press, 1990.

[5] H. Katsuno and A. Mendelzon. *On the difference between updating a knowledge base and revising it.* In James Allen, Richard Fikes and Erik Sandewall, editors, Principles of Knowledge Representation and Reasoning: Proc. of the Second Int'l Conf. (KR91), pages 230-237, Morgan Kaufmann 1991.

[6] João A. Leite. *Logic Program Updates.* MSc dissertation, Universidade Nova de Lisboa, 1997.

[7] J. A. Leite and L. M. Pereira. *Generalizing updates: from models to programs.* In LPKR'97: ILPS'97 workshop on Logic Programming and Knowledge Representation, Port Jefferson, NY, USA, October 13-16, 1997.

[8] V. Lifschitz and T. Woo. *Answer sets in general nonmonotonic reasoning (preliminary report).* In B. Nebel, C. Rich and W. Swartout, editors, Principles of Knowledge Representation and

[2]The implementation is available from `http://www-ssdi.di.fct.unl.pt/~jja/updates/`.

Reasoning, Proc. of the Third Int'l Conf (KR92), pages 603-614. Morgan-Kaufmann, 1992

[9] V.Marek and M. Truszczynski. *Revision specifications by means of programs.* In C. MacNish, D. Pearce and L. M. Pereira, editors, JELIA '94, volume 838 of LNAI, pages 122-136. Springer-Verlag, 1994.

[10] T. Przymusinski and H. Turner. *Update by means of inference rules.* In V. Marek, A. Nerode, and M. Truszczynski, editors, LP-NMR'95, volume 928 of LNAI, pages 156-174. Springer-Verlag, 1995.

[11] M. Winslett. *Reasoning about action using a possible models approach.* In Proceeding of AAAI'88, pages 89-93. 1988.