# Dynamic Updates of Non-Monotonic Knowledge Bases

**J. J. Alferes**
Dept. Matemática
Univ. Évora and
A.I. Centre
Univ. Nova de Lisboa,
2825 Monte da Caparica
Portugal

**J. A. Leite, L. M. Pereira**
A.I. Centre
Dept. Informática
Univ. Nova de Lisboa
2825 Monte da Caparica
Portugal

**H. Przymusinska**
Computer Science
California State
Polytechnic Univ.
Pomona, CA 91768
USA

**T. C. Przymusinski**
Computer Science
Univ. of California
Riverside, CA 92521
USA

**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 rule bodies but also in their heads.We start by introducing the notion of an update $P \oplus U$ of one 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 compositional sequences of logic programs updates $P_1 \oplus P_2 \oplus \ldots$, defining a dynamic program update, and 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.

Specifically, 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 employ the mutual relationships existing between different modules to precisely determine, at any given module composition stage, the declarative as well as the procedural semantics of the combined program resulting from the modules.

## 1 Introduction

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. This is a serious drawback when dealing with *dynamic knowledge bases* in which not only the *extensional* part (the set of facts) changes dynamically but so does the *intensional* part (the set of rules).

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 rule bodies but also in their heads. This is needed, in particular, in order to specify that some atoms should became false, i.e., should be deleted. However, our updates are far more expressive than a mere insertion and deletion of facts. They can be specified by means of arbitrary program rules and thus they themselves are logic programs. Consequently, our approach demonstrates how to update one generalized logic program $P$ (the initial program) by another generalized logic program $U$ (the updating program), obtaining as a result a new, updated logic program $P \oplus U$.

Several authors have addressed the issue of updates of logic programs and deductive databases (see e.g. [12, 13, 14, 1]), most of them following the so called *"interpretation update"* approach, originally proposed in [15, 8]. This approach 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 (models[1]). 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 $DB$, i.e., "the set of models of $DB'$" = "the set of updated models of $DB$". Thus, according to the interpretation update approach, the problem of finding an update of a *deductive* database $DB$ is reduced to the problem of finding individual updates of all of its *relational instantiations* (models) $M$. Unfortunately, such an approach suffers, in general, from several important drawbacks[2]:

- In order to obtain the update $DB'$ of a knowledge base $DB$ one has to first compute all the models $M$ of $DB$ (typically, a daunting task) and then individually compute their (possibly multiple) updates $M_U$ by $U$. An update $M_U$ of a given interpretation $M$ is obtained by changing the status of only those literals in $M$ that are *"forced"* to change by the update $U$, while keeping all the other literals intact by *inertia* (see e.g. [12, 13, 14]).
- The updated knowledge base $DB'$ is not defined directly but, instead, it is indirectly characterized as a knowledge base whose models coincide with the set of all updated models $M_U$ of $DB$. In general, there is therefore no natural way of computing[3] $DB'$ because the only straightforward candidate for $DB'$ is the typically intractably large knowledge base $DB''$ consisting of all clauses that are entailed by all the updated models $M_U$ of $DB$.
- Most importantly, while the *semantics* of the resulting knowledge base $DB'$ indeed represents the *intended* meaning when just the *extensional* part of the knowledge base $DB$ (the set of facts) is being updated, it leads to strongly *counter-intuitive* results when also the *intensional* part of the database (the set of rules) undergoes change, as the following example shows.

**Example 1.1** *Consider the logic program $P$ :*

$$P: \begin{array}{lll} sleep & \leftarrow & not\, tv\_on \\ tv\_on & \leftarrow & \\ watch\_tv & \leftarrow & tv\_on. \end{array} \tag{1}$$

*Clearly $M = \{tv\_on, watch\_tv\}$ is its only stable model. Suppose now that the update $U$ states that there is a power failure, and if there is a power failure then the TV is no longer on, as represented by the logic program $U$:*

$$U: \begin{array}{lll} not\, tv\_on & \leftarrow & power\_failure \\ power\_failure & \leftarrow & \end{array} \tag{2}$$

[1]The notion of a model depends on the type of considered knowledge bases and on their semantics. In this paper we are considering (generalized) logic programs under the stable model semantics.

[2]In [1] the authors addressed the first two of the drawbacks mentioned below. They showed how to directly construct, given a logic program $P$, another logic program $P'$ whose partial stable models are exactly the interpretation updates of the partial stable models of $P$. This eliminates both of these drawbacks (in the case when knowledge bases are logic programs) but it does not eliminate the third, most important drawback.

[3] In fact, in general such a database $DB'$ may not exist at all.

*According to the above mentioned interpretation approach to updating, we would obtain* $M_U = \{power\_failure, watch\_tv\}$ *as the only update of* $M$ *by* $U$*. This is because* $power\_failure$ *needs to be added to the model and its addition forces us to make tv_on false. As a result, even though there is a power failure, we are still watching TV. However, by inspecting the initial program and the updating rules, we are likely to conclude that since "watch_tv" was true only because "tv_on" was true, the removal of "tv_on" should make "watch_tv" false by default. Moreover, one would expect "sleep" to become true as well. Consequently, the intended model of the update of* $P$ *by* $U$ *is the model* $M_U^{'} = \{power\_failure, sleep\}$.

*Suppose now that another update* $U_2$ *follows, described by the logic program:*

$$U_2 : \quad not\, power\_failure \leftarrow \tag{3}$$

*stating that power is back up again. We should now expect the TV to be on again. Since power was restored, i.e. "*$power\_failure$*" is false, the rule "*$not\, tv\_on \leftarrow power\_failure$*" of* $U$ *should have no effect and the truth value of "tv_on" should be obtained by inertia from the rule "*$tv\_on \leftarrow$ *" of the original program* $P$. □

This example illustrates that, when updating knowledge bases, it is not sufficient to just consider the truth values of literals figuring in the heads of its rules because the truth value of their rule bodies may also be affected by the updates of other literals. In other words, it suggests that the *principle of inertia* should be applied not just to the individual literals in an interpretation but rather to the *entire rules of the knowledge base.*

The above example also leads us to another important observation, namely, that the notion of an update $DB'$ of one knowledge base $DB$ by another knowledge base $U$ should not just depend on the *semantics* of the knowledge bases $DB$ and $U$, as it is the case with interpretation updates, but that it should also depend on their *syntax*. This is best illustrated by the following, even simpler, example:

**Example 1.2** *Consider the logic program* $P$ :

$$P : \quad innocent \leftarrow not\, found\_guilty \tag{4}$$

*whose only stable model is* $M = \{innocent\}$*, because* $found\_guilty$ *is false by default. Suppose now that the update* $U$ *states that the person has been found guilty:*

$$U : \quad found\_guilty \leftarrow . \tag{5}$$

*Using the interpretation approach, we would obtain* $M_U = \{innocent, found\_guilty\}$ *as the only update of* $M$ *by* $U$ *thus leading us to the counter-intuitive conclusion that the person is both innocent and guilty. This is because* $found\_guilty$ *must be added to the model* $M$ *and yet its addition does not force us to make innocent false. However, it is intuitively clear that the interpretation* $M_U^{'} = \{found\_guilty\}$*, stating that the person is guilty but no longer presumed innocent, should be the only model of the updated program. Observe, however, that the program* $P$ *is* semantically equivalent *to the following program* $P'$ :

$$P' : \quad innocent \leftarrow \tag{6}$$

*because the programs* $P$ *and* $P'$ *have exactly the same set of stable models, namely the model* $M$*. Nevertheless, while the model* $M_U = \{innocent, found\_guilty\}$ *is not the intended model of the update of* $P$ *by* $U$ *it is in fact the only reasonable model of the update of* $P'$ *by* $U$. □

In this paper we investigate the problem of updating knowledge bases represented by generalized logic programs and we propose a new solution to this problem that attempts to eliminate the drawbacks of the previously proposed approaches. Specifically, given one generalized logic program $P$ (the so called initial program) and another logic program $U$ (the updating program) we define a new generalized logic program $P \oplus U$ called the *update* of $P$ by $U$. The definition of the updated program $P \oplus U$ does not require any computation of the models of either $P$ or $U$ and is in fact obtained by means of a simple, *linear-time* transformation of the programs $P$ and $U$. As a result, the update transformation can be accomplished very efficiently and its *implementation* is quite straightforward[4].

Due to the fact that we apply the inertia principle not just to atoms but to entire program rules, the semantics of our updated program $P \oplus U$ avoids the drawbacks of interpretation updates and it seems to properly represent the intended semantics. As mentioned above, the updated program $P \oplus U$ does not just depend on the *semantics* of the programs $P$ and $U$, as it was the case with interpretation updates, but it also depends on their *syntax*. In order to make the meaning of the updated program clear and easily verifiable, we provide a *complete characterization* of the semantics of updated programs $P \oplus U$.

Nevertheless, while our notion of program update significantly differs from the notion of interpretation update, it coincides with the latter (as originally introduced in [12] under the name of *revision program* and later reformulated in the language of logic programs in [13, 14]) when the initial program $P$ is purely *extensional*, i.e., when the initial program is just a set of facts. Our definition also allows significant flexibility and can be easily modified to handle updates which incorporate *contradiction removal* or specify different inertia rules. Consequently, our approach can be viewed as introducing a general dynamic logic programming *framework for updating logic programs* which can be suitably modified to make it fit different application domains and requirements.

Finally, we extend the notion of program updates to sequences of programs, defining the so called *dynamic program updates*. The idea of dynamic updates 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 role of the dynamic program update $\bigoplus\{P_s : s \in S\}$ is to use the mutual relationships existing between different states (as specified by the order 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. Whereas traditional logic programming has concerned itself mostly with representing static knowledge, we show how to use logic programs to represent dynamically changing knowledge.

Our results extend and improve upon the approach initially proposed in [10], where the authors first argued that the principle of inertia should be applied to the rules of the initial program rather than to the individual literals in an interpretation. However, the specific update transformation presented in [10] suffered from some drawbacks and was not sufficiently general.

We begin in Section 2 by defining the language of *generalized logic programs,* which allow default negation in rule heads. We describe stable model semantics of such programs as a special case of the approach proposed earlier in [11]. In Section 3 we define the program update $P \oplus U$ of the initial program $P$ by the updating program $U$. In Section 4 we provide a complete characterization of the semantics of program updates $P \oplus U$ and in Section 5 we study their basic properties. In Section 6

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

we introduce the notion of dynamic program updates. We close the paper with concluding remarks and notes on future research.

## 2 Generalized Logic Programs and their Stable Models

In order to represent *negative* information in logic programs and in their updates, we need more general logic programs that allow default negation $not\, A$ not only in premises of their clauses but also in their heads[5]. We call such programs *generalized logic programs.* In this section we introduce generalized logic programs and extend the stable model semantics of normal logic programs [6] to this broader class of programs. In the subsequent paper [5] we extend our results to *3-valued (partial)* models of logic programs, and thus, in particular, to *well-founded semantics.*

The class of generalized logic programs can be viewed as a special case of a yet broader class of programs introduced earlier in [11]. While our definition is different and seems to be simpler than the one used in [11], when restricted to the language that we are considering, the two definitions can be shown to be equivalent. It should be stressed that the class of generalized logic programs differs from the class of programs with the so called *"classical"* negation [7] which allow the use of *strong* rather than *default* negation in their heads.

It will be convenient to *syntactically* represent generalized logic programs as *propositional Horn theories.* In particular, we will represent default negation $not\, A$ 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 $\mathcal{L}$ whose set of propositional variables consists of:

$$\{A : A \in \mathcal{K}\} \cup \{not\, A : A \in \mathcal{K}\}. \tag{7}$$

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 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:

$$L \leftarrow L_1, \ldots, L_n \tag{8}$$

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.

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 an interpretation $M$ we define:

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

By a (2-valued) *model* $M$ of a generalized logic program $P$ we mean a (2-valued) interpretation of $P$ that satisfies all of its clauses. A program is called *consistent* if it has a model. A model $M$ is considered *smaller* than a model $N$ if the set of *objective* atoms of $M$ is properly contained in the set of objective atoms of $N$. A model of $P$ is called *minimal* if there is no smaller model of $P$. A model of $P$ is called *least* if it is the smallest model of $P$. It is well-known that every consistent program $P$ has the least model $M = \{A : A$ is an atom and $P \vdash A\}$.

[5]For further motivation and intuitive reading of logic programs with default negations in the heads see [11].

**Definition 2.1** ***(Stable models of generalized logic programs)*** *We say that a (2-valued) interpretation $M$ of $\mathcal{L}_{\mathcal{K}}$ is a stable model of a generalized logic program $P$ if $M$ is the least model of the Horn theory $P \cup M^-$:*

$$M = Least(P \cup M^-), \tag{9}$$

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

**Example 2.1** *Consider the program:*

$$\begin{array}{rclcrclcrcl} a & \leftarrow & not\, b & & c & \leftarrow & b & & e & \leftarrow & not\, d \\ not\, d & \leftarrow & not\, c, a & & d & \leftarrow & not\, e & & & & \end{array} \tag{10}$$

*and let $\mathcal{K} = \{a, b, c, d, e\}$. 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:*

$$M = Least(P \cup \{not\, b, not\, c, not\, d\}). \tag{11}$$

*On the other hand, the interpretation $N = \{not\, a, not\, e, b, c, d\}$ is not a stable model because:*

$$N \neq Least(P \cup \{not\, e, not\, a\}) = \{d, not\, a, not\, e\}. \ \square \tag{12}$$

Following an established tradition, whenever convenient we will be omitting the default (negative) atoms when describing interpretations and models. Thus the above model $M$ will be simply listed as $M = \{a, e\}$.

Throughout the paper by the *semantics* of a generalized logic program we mean the stable semantics. We also say that two generalized logic programs in a given language $\mathcal{L}$ are *semantically equivalent* if they have the same set of stable models.

Given a generalized logic program $P$ and an interpretation $M$, by the *Gelfond-Lifschitz transform* of $P$ w.r.t. $M$ we mean a generalized logic program $\frac{P}{M}$ obtained from $P$ by (a) removing from $P$ all clauses which contain default premises *not* $A$ such that $A \in M$, and, (b) removing default premises *not* $A$ from all the remaining clauses. Clearly, the above definition extends the notion of the Gelfond-Lifschitz transform [6] from the class of normal programs to the class of generalized logic programs. The following proposition easily follows from the definition of stable models.

**Proposition 2.1** *An interpretation $M$ of $\mathcal{L}_{\mathcal{K}}$ is a stable model of a generalized logic program $P$ if and only if*

$$M^+ = \{A : A \in \mathcal{K} \text{ and } \frac{P}{M} \vdash A\} \tag{13}$$

*and*

$$M^- \supseteq \{not\, A : A \in \mathcal{K} \text{ and } \frac{P}{M} \vdash not A\}. \square \tag{14}$$

Clearly, the second condition in the above proposition is always vacuously satisfied for normal programs. Since the first condition characterizes stable models of normal programs [6], we immediately obtain:

**Proposition 2.2** *The class of stable models of generalized logic programs extends the class of stable models of normal programs. More precisely, an interpretation is a stable model of a normal program in the sense of Definition 2.1 if and only if it is a stable model in the sense of* Gelfond-Lifschitz *[6].* □

# 3 Program Updates

Suppose 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 $\mathcal{K}$:

$$\widehat{\mathcal{K}} = \mathcal{K} \cup \{A^-,\ A_P,\ A_P^-,\ A_U,\ A_U^- : A \in \mathcal{K}\}. \tag{15}$$

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 extension 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 $\mathcal{L}$. 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:**

$$A_P \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \tag{16}$$
$$A_P^- \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \tag{17}$$

*for any clause:*

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

*respectively,*

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

*in the original program $P$. The rewritten clauses are obtained from the original ones by replacing atoms $A$ (respectively, the atoms $not\, A$) occurring in their heads by the atoms $A_P$ (respectively, $A_P^-$) and by replacing negative premises $not\, C$ by $C^-$.*
*The role of the new meta-level atoms $A_P$ and $A_P^-$ is to indicate the fact that these clauses originally came from the program P. Moreover, as we will demonstrate below, the new atoms $A^-$ serve as meta-language representation of the default atoms $not\, A$.*

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

$$A_U \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \tag{18}$$
$$A_U^- \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \tag{19}$$

*for any clause:*

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

*respectively,*

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

*in the updating program $U$. The rewritten clauses are obtained from the original ones by replacing atoms $A$ (respectively, the atoms $not\, A$) occurring in their heads by the atoms $A_U$ (respectively, $A_U^-$) and by replacing negative premises $not\, C$ by $C^-$.*
*The role of the new meta-level atoms $A_U$ and $A_U^-$ is to indicate the fact that these clauses originally came from the updating program U. Moreover, as we will demonstrate below, the new atoms $A^-$ serve as meta-language representation of the default atoms $not\, A$.*

**(UR) Update rules:**

$$A \leftarrow A_U \tag{20}$$

$$A^- \leftarrow A_U^- \tag{21}$$

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

**(IR) Inheritance rules:**

$$A \leftarrow A_P, not\, A_U^- \tag{22}$$

$$A^- \leftarrow A_P^-, not\, A_U \tag{23}$$

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

**(DR) Default rules:**

$$A^- \leftarrow not\, A_P, not\, A_U \tag{24}$$

$$not\, A \leftarrow A^-, \tag{25}$$

*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 also ensures that $A$ and $A^-$ cannot both be true.* □

**Proposition 3.1** *Any model $N$ of $P \oplus U$ is* coherent*, by which we mean that $A$ is true (respectively, false) in $N$ iff $A^-$ is false (respectively, true) in $N$, for any $A \in \mathcal{K}$. In other words, every model of $P \oplus U$ satisfies the constraint $not\, A \equiv A^-$.*

*Proof.* Clearly, due to the second default rule, $A$ and $A^-$ cannot both be true in $N$. On the other hand, if both $A$ and $A^-$ are false in $N$ then, due to the update rules, both $not\, A_U$ and $not\, A_U^-$ must be true. From the first inheritance axiom we infer that $not\, A_P$ must hold, which, in view of the first default rule, leads to a contradiction. □

**Remark 3.1** *According to the above proposition, the atoms $A^-$ can be simply regarded as* meta-level representation *of the default negation atoms $not\, A$. Similarly, the remaining, newly added atoms, $A_P$, $A_P^-, A_U$ and $A_U^-$ serve as meta-level representation of the atoms (or their default negations) derivable from programs $P$ and $U$, respectively.*
*When we discuss interpretations or models of the extended language $\widehat{\mathcal{L}} = \mathcal{L}_{\widehat{\mathcal{K}}}$ we often restrict our attention to the* "relevant" *atoms, i.e., to the atoms from the base language $\mathcal{L} = \mathcal{L}_{\mathcal{K}}$ and thus we ignore, whenever justified, the auxiliary, meta-level atoms $A^-$, $A_P$, $A_P^-, A_U$ and $A_U^-$.*

**Example 3.1** *Consider the programs $P$ and $U$ from Example 1.1:*

$$\begin{array}{lrcl} P: & sleep & \leftarrow & not\, tv\_on \\ & tv\_on & \leftarrow & \\ & watch\_tv & \leftarrow & tv\_on \\ \\ U: & not\ tv\_on & \leftarrow & power\_failure \\ & power\_failure & \leftarrow & \end{array} \tag{26}$$

*The update of the program $P$ by the program $U$ is the logic program $P \oplus U = (RP) \cup (RU) \cup (UR) \cup (IR) \cup (DR)$, where:*

$$
\begin{array}{lrcl}
RP: & sleep_P & \leftarrow & tv\_on^- \\
 & tv\_on_P & \leftarrow & \\
 & watch\_tv_P & \leftarrow & tv\_on \\
 & & & \\
RU: & tv\_on_U^- & \leftarrow & power\_failure \\
 & power\_failure_U & \leftarrow &
\end{array}
\tag{27}
$$

*It is easy to verify that $M = \{power\_failure, sleep\}$ is the only stable model (restricted to relevant atoms) of $P \oplus U$. Indeed, $power\_failure$ follows from the second clause of (RU) and from the Update Rules (UR). Now from $power\_failure$, the first rule of (RU) and the Update Rules (UR) we deduce $tv\_on^-$ and thus also $not\ tv\_on$. From the first rule of (RP) we infer $sleep_P$ and from the Inheritance Rules (IR) we deduce $sleep$. Finally, $watch\_tv^-$ and $not\ watch\_tv$ follow from the default rules.* □

# 4 Semantic Characterization of Program Updates

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

Let $P$ and $U$ be *fixed* generalized logic programs in the language $\mathcal{L}$. Since the update program $P \oplus U$ is defined in the extended language $\widehat{\mathcal{L}}$, we begin by first showing how interpretations of the language $\mathcal{L}$ can be naturally extended to interpretations of the extended language $\widehat{\mathcal{L}}$.

Since any model $N$ of the update program $P \oplus U$ is coherent (see Proposition 3.1) and since the atoms $A_P$, $A_P^-$, $A_U$ and $A_U^-$ appear only in the heads of the rewritten program rules, if $N$ is a minimal (in particular, stable) model of the update program $P \oplus U$ then $N$ must satisfy, for any $A \in \mathcal{K}$:

$$
\begin{array}{rll}
A^- \in N & \text{iff} & not\, A \in N \\
A_P \in N & \text{iff} & \exists\ A \leftarrow Body \in P \ \text{ and } \ N \models Body \\
A_P^- \in M & \text{iff} & \exists\ not\, A \leftarrow Body \in P \ \text{ and } \ N \models Body \\
A_U \in M & \text{iff} & \exists A \leftarrow Body \in U \text{ and } \ N \models Body \\
A_U^- \in \widehat{N} & \text{iff} & \exists\ not\, A \leftarrow Body \in U \text{ and } \ N \models Body.
\end{array}
\tag{28}
$$

Accordingly, the truth or falsity in $N$ of the atoms $A^-$, $A_P$, $A_P^-$, $A_U$ and $A_U^-$ depends only on the truth or falsity in $N$ of the atoms $A$ from $\mathcal{K}$. This leads us to the following definition:

**Definition 4.1 (Extended Interpretation)** *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:*

$$
\begin{array}{rll}
A^- \in \widehat{M} & \textit{iff} & not\, A \in M \\
A_P \in \widehat{M} & \textit{iff} & \exists\ A \leftarrow Body \in P \ \textit{ and } \ M \models Body \\
A_P^- \in \widehat{M} & \textit{iff} & \exists\ not\, A \leftarrow Body \in P \ \textit{ and } \ M \models Body \\
A_U \in \widehat{M} & \textit{iff} & \exists A \leftarrow Body \in U \textit{ and } \ M \models Body \\
A_U^- \in \widehat{M} & \textit{iff} & \exists\ not\, A \leftarrow Body \in U \textit{ and } \ M \models Body. \square
\end{array}
$$

This definition immediately implies:

**Proposition 4.1** *If $N$ is a minimal model of the update program $P \oplus U$ and $M = N|L$ is its restriction to the language $\mathcal{L}$ then $N = \widehat{M}$.* $\square$

We will also need the following definition:

**Definition 4.2** *For any interpretation $M$ of the language $\mathcal{L}$ define:*

$$\begin{array}{lcl} Defaults[M] & = & \{not\, A : \forall (A \leftarrow Body) \in P \cup U \text{ we have } M \not\models Body\}; \\ Rejected[M] & = & \{A \leftarrow Body \in P : M \models Body,\ \exists\ (not\, A \leftarrow Body') \in U,\ M \models Body'\} \cup \\ & & \cup\ \{not\, A \leftarrow Body \in P : M \models Body,\ \exists\ (A \leftarrow Body') \in U,\ M \models Body'\}; \\ Residue[M] & = & P \cup U - Rejected[M]. \end{array}$$ $\square$

The set $Defaults[M]$ contains default negations $not\, A$ of all *unsupported* atoms $A$ in $M$, i.e., atoms that have the property that the body of every clause from $P \cup U$ with the head $A$ is false in $M$. 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 the interpretation $M$. 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 $U$. Note that all the three sets depend on the interpretation $M$ as well as on the *syntax* of the programs $P$ and $U$.

Now we are able to describe the semantics of the 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 program $P \oplus U$ if and only if $N$ is the extension $N = \widehat{M}$ of an interpretation $M$ of the language $\mathcal{L}$ that satisfies the condition:*

$$M = Least(P \cup U - Rejected[M] \cup Defaults[M]), \tag{29}$$

*or, equivalently:*

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

*Proof.* ( $\Longrightarrow$ ) Suppose that $N$ is a stable model of the update program $P \oplus U$ and let $R = (P \oplus U) \cup N^-$. From Definition 2.1 it follows that:

$$N = \text{Least}(R) = \text{Least}(\ (P \oplus U) \cup N^-). \tag{30}$$

Let $T = Residue[M] \cup Defaults[M]$ and let $H$ =Least$(T)$ be its least model (in the language $\mathcal{L}$). We are supposed to show that the restriction $M = N|\mathcal{L}$ of $N$ to the language $\mathcal{L}$ coincides with $H$. From Proposition 4.1, we infer that the following equivalences hold true for every atom $A \in \mathcal{K}$:

$$\begin{array}{llll} A^- \in N & \text{iff} & not\, A \in M & \text{iff} \quad not\, A \in N \\ A_P \in N & \text{iff} & \exists\ A \leftarrow Body \in P,\ M \models Body & \text{iff} \quad \exists\ A \leftarrow Body \in P, N \models Body \\ A_P^- \in N & \text{iff} & \exists\ not\, A \leftarrow Body \in P,\ M \models Body & \text{iff} \quad \exists\ not\, A \leftarrow Body \in P,\ N \models Body \\ A_U \in N & \text{iff} & \exists A \leftarrow Body \in U,\ M \models Body & \text{iff} \quad \exists A \leftarrow Body \in U,\ N \models Body \\ A_U^- \in N & \text{iff} & \exists\ not\, A \leftarrow Body \in U,\ M \models Body & \text{iff} \quad \exists\ not\, A \leftarrow Body \in U,\ N \models Body. \end{array} \tag{31}$$

Denote by $\mathcal{S}$ the sub-language of $\widehat{\mathcal{L}}$ that includes only propositional symbols $\{A : A \in \mathcal{K}\} \cup \{A^- : A \in \mathcal{K}\}$. By means of several simple reductions we will transform the program $R = (P \oplus U) \cup N^-$ in the language $\widehat{\mathcal{L}}$ into a simpler program $Y$ in the language $\mathcal{S}$ so that:

- The least model $J = \text{Least}(Y)$ of $Y$ is equal to the least model $N = \text{Least}(R)$ of $R$ when restricted to the language $\mathcal{S}$, i.e., $J = N|\mathcal{S}$;
- The program $Y$ in the language $\mathcal{S}$ is syntactically identical to the program $T = Residue[M] \cup Defaults[M]$ in the language $\mathcal{L}$, except that $notA$ is everywhere replaced by $A^-$.

First of all, we observe that from (31) it follows that for any $A \in \mathcal{K}$ neither $A_P$ nor $A_U$ belongs to $N$ if and only if $notA \in Defaults[M]$. Accordingly, the first default rule in $R = (P \oplus U) \cup N^-$, namely, $A^- \leftarrow not\, A_P, not\, A_U$, can be replaced by the rule:

$$A^-, \text{ for all } A \in \mathcal{K} \text{ such that } notA \in Defaults[M]$$

without affecting the least model of $R$. As a result we obtained a transformed program $R'$.

¿From (31) it also follows that the inheritance rules (IR):

$$A \leftarrow A_P, not\, A_U^- \tag{32}$$

$$A^- \leftarrow A_P^-, not\, A_U \tag{33}$$

in $R'$ can be replaced by the simpler rules:

$$A \leftarrow A_P \tag{34}$$

$$A^- \leftarrow A_P^- \tag{35}$$

without affecting the least model of $R'$ restricted to the language $\mathcal{S}$ as long as we remove from $R'$ all the rules:

$$A_P \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \tag{36}$$

$$A_P^- \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \tag{37}$$

respectively, that correspond to program rules:

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

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

from $P$ that were rejected by $U$, i.e., to the rules that belong to $Rejected[M]$. This is due to the fact that both $A_P$ and $A_U^-$ (respectively, both $A_P^-$ and $A_U$) are true in $N$ if and only if there is a program clause:

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

respectively:

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

in $P$ that belongs to $Rejected[M]$. Since the propositional symbols $A_P$ and $A_P^-$ do not appear in bodies of any other clauses from $R'$, removing these rules from $R'$ does not in anyway affect the truth of the propositional symbols $A$ and $A^-$ and thus it does not affect the least model of $R'$ restricted to the language $\mathcal{S}$. As a result we obtain the program $R''$.

We can now remove all the negative facts in $N^-$ and the default rules $not\, A \leftarrow A^-$ from $R''$ because they only involve propositional symbols $not\, A$ which no longer appear in bodies of any other clauses from $R''$and thus do not affect the least model of $R^{''}$ restricted to the language $\mathcal{S}$. As a result we obtain the program $R'''$.

Finally, since we are only interested in the sub-language $\mathcal{S}$, we can now safely remove from $R'''$ all the auxiliary propositional symbols $A_P$ and $A_P^-$, obtaining as a result the final program $Y$ in the language $\mathcal{S}$ that consists of all the clauses:

$$A \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$$
$$A^- \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$$

corresponding to the clauses from $Residue[M] = P \cup U - Rejected[M]$ together with all the atomic facts:

$$A^-, \text{ where } not A \in Defaults[M].$$

Clearly, this program is entirely identical to the program $T = Residue[M] \cup Defaults[M]$, except that $not A$ is everywhere replaced by $A^-$. Consequently, the least model $J$ of $Y$ is identical to the least model $H$ of $T$, except that $not A$ is everywhere replaced by $A^-$. Moreover, due to the way in which it was obtained, the least model $J = \text{Least}(Y)$ of the program $Y$ is equal to the least model $N = \text{Least}(R)$ of $R$ restricted to the language $\mathcal{S}$, i.e., $J = N|\mathcal{S}$. This implies that for any $A \in \mathcal{K}$:

$$A \in N \quad \text{iff} \quad A \in J \quad \text{iff} \quad A \in H$$
$$A^- \in N \quad \text{iff} \quad A^- \in J \quad \text{iff} \quad not A \in H.$$

We conclude that $M = N|\mathcal{L} = H$, because from (31) it follows that $not A \in N$ iff $A^- \in N$. This completes the proof of the implication from left to right.

The converse implication is established in a completely analogous way. □

**Example 4.1** *Consider again the programs $P$ and $U$ from Example 1.1. Let $M = \{power\_failure, sleep\}$. We obtain:*

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

*and thus it is easy to see that*

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

*Consequently, $\widehat{M}$ is a stable model of the update program $P \oplus U$. In fact, it is the only stable model of this program.* □

# 5 Properties of Program Updates

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

$$M = Least(Residue[M] \cup Defaults[M]) \tag{C1}$$

which, according to Theorem 4.1, is equivalent to $\widehat{M}$ being a stable model of $P \oplus U$, clearly implies the condition:

$$M = Least(Residue[M] \cup M^-) \tag{C2}$$

which, according to Proposition 2.1, is equivalent to $M$ being a stable model of $Residue[M] = P \cup U - Rejected[M]$. Consequently, we immediately obtain:

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

In general, the converse of the above implication does not hold. This is because (C1) states that the model $M$ is *determined* not just by the set $M^-$ of all negative atoms *not* $A$ but rather by the generally smaller set $Defaults[M]$ of negations of unsupported atoms.

**Example 5.1** *Let $P$ contain only the fact $A \leftarrow$ , let $U$ contain only the clause $not\, A \leftarrow not\, A$ and let $M = \{not\, A\}$. Since $Residue[M] = U$ clearly $M$ is a stable model of $Residue[M]$ and thus satisfies the condition $(C2)$. However, since $Defaults[M] = \emptyset$ the interpretation $M$ does not satisfy $(C1)$ and thus $\widehat{M}$ is not a stable model of the updated program $P \oplus U$. In fact, $M = \{\, A\}$ is the only stable model of $P \oplus U$.* □

However, if $Rejected[M] = \emptyset$ then the two conditions (C1) and (C2) coincide because then $M$ is a model of $Residue(M) = P \cup U$ and thus $Defaults[M] = M^-$. In particular, $Rejected[M] = \emptyset$ if $M$ is a stable model of $P \cup U$ which yields:

**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 5.2** *Consider again the programs $P$ and $U$ from Example 1.1. It is easy to see that $P \cup U$ is contradictory.* □

Similarly, if either $P$ or $U$ is empty then, for any interpretation $M$, $Rejected[M] = \emptyset$, and, therefore we conclude:

**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 $P \cup U$.* □

If both $P$ and $U$ are normal programs, or, alternatively, if both $P$ and $U$ have only clauses with default atoms *not* $A$ in their heads, then also $Rejected[M] = \emptyset$ and therefore we obtain:

**Proposition 5.4** *If both $P$ and $U$ are normal programs (or if both have only clauses with default 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

In this section we show that *interpretation updates*, originally introduced under the name *"revision programs"* by Marek and Truszczynski [12], and subsequently given a somewhat simpler characterization by Przymusinski and Turner [13, 14], constitute a special case of program updates. Here, we identify the *"revision rules"*:

$$\begin{array}{rcl} in(A) & \leftarrow & in(B), out(C) \\ out(A) & \leftarrow & in(B), out(C) \end{array} \tag{38}$$

used in [12], with the following generalized logic program clauses:

$$\begin{array}{rcl} A & \leftarrow & B, not\, C \\ not\, A & \leftarrow & B, not\, C. \end{array} \tag{39}$$

**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 $\mathcal{L}$ defined by*

$$P_I = \{A \ \leftarrow : A \in I\} \ \cup \ \{not\, A \ \leftarrow : not\, A \in I\}.$$

*Then $\widehat{M}$ is a stable model of the program update $P_I \oplus U$ of the program $P_I$ by the program $U$ iff $M$ is an interpretation update of $I$ by $U$ (in the sense of [12]).*

*Proof.* Przymusinski and Turner [13, 14] showed that an interpretation $M$ of $\mathcal{L}$ is an interpretation update (in the sense of [12]) of $I$ by a program $U$ iff $M$ is a stable model of the following program $P(I, U)$:

**Encoded interpretation I:**

$$A_I \leftarrow$$

for every $A$ such that $A$ is in I, and ,

$$A_I^- \leftarrow$$

for every $A$ such that $not\, A$ is in I.

**Rewritten clauses from $U$:**

$$A \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \tag{40}$$

$$A^- \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \tag{41}$$

for any clause:

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

respectively,

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

in the updating program $U$.

**Inheritance rules:**

$$A \leftarrow A_I, not\, A^- \tag{42}$$

$$A^- \leftarrow A_I^-, not\, A \tag{43}$$

for all objective atoms $A \in \mathcal{K}$.

**Default rule:**

$$not\, A \leftarrow A^-,$$

for all objective atoms $A \in \mathcal{K}$.

It is easy to see that the above program $P(I, U)$ is semantically equivalent to the program update $P_I \oplus U$ of the program $P_I$ by the updating program $U$. □

This theorem shows that when the initial program $P$ is purely *extensional*, i.e., contains only positive or negative *facts*, then the interpretation update of $P$ by $U$ is semantically equivalent to the updated program $P \oplus U$. As shown by the Examples 1.1 and 1.2, when $P$ contains deductive rules then the two notions become significantly different.

**Remark 5.1** *It is easy to see that, equivalently, we could include only positive facts in the definition of the program $P_I$:*

$$P_I = \{A \ \leftarrow : A \in I\}$$

*thus resulting in a normal program $P_I$.* □

### 5.2 Adding Strong Negation

We now show that it is easy to add *strong negation* $-A$ ([7, 2, 3]) to generalized logic programs. This demonstrates that the class of generalized logic programs is at least as expressive as 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 be both true we also assume, for all $A \in \mathcal{K}$, the following strong negation axioms, which themselves are generalized logic program clauses:*

$$\begin{array}{rcl} not\, A & \leftarrow & -A \\ not\, -A & \leftarrow & A. \end{array} \qquad \text{((SN))}$$

**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 (see the next section).* □

# 6 Dynamic Program Updates

In this section we introduce the notion of *dynamic program update* $\bigoplus\{ P_s : s \in S\}$ over an ordered set $\mathcal{P} = \{ P_s : s \in S\}$ of logic programs which provides an important generalization of the notion of single program updates $P \oplus U$ introduced in Section 3.

The idea of dynamic updates, inspired by [9], is 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 role 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 finite or infinite set $S = \{1, 2, \ldots, n, \ldots\}$ of natural numbers. 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 could 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 for every state $s$, the initial state $s_0$ is the $n$-th immediate predecessor of $s$, 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:

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

As before, this definition assumes that the original set $\mathcal{K}$ of propositional variables does not contain any of the newly added symbols of the form $A^-, A_s,\ A_s^-, A_{P_s},\ A_{P_s}^-$ so that they are all disjoint sets of symbols. If the original language $\mathcal{K}$ contains any such symbols then they have to be *renamed* before the extension of $\mathcal{K}$ takes place. 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 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 $\overline{\mathcal{L}}$:*

**(RP) Rewritten program clauses:**

$$A_{P_s} \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \tag{44}$$

$$A_{P_s}^- \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \tag{45}$$

*for any clause:*

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

*respectively, for any clause:*

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

*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 $A_{P_s}$ (respectively, $A_{P_s}^-$) and by replacing negative premises $not\, C$ by $C^-$.*

**(UR) Update rules:**

$$\begin{array}{rcl} A_s & \leftarrow & A_{P_s} \\ A_s^- & \leftarrow & A_{P_s}^- \end{array} \tag{46}$$

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

**(IR) Inheritance rules:**

$$A_s \leftarrow A_{s-1}, not\, A_{P_s}^- \tag{47}$$

$$A_s^- \leftarrow A_{s-1}^-, not A_{P_s} \tag{48}$$

*for all objective atoms $A \in \mathcal{K}$ and for 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 the updating program $P_s$.*

**(DR) Default rules:**

$$A_0^-, \tag{49}$$

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

Observe that the dynamic program update $\biguplus\mathcal{P}$ 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 $\biguplus\mathcal{P}$ does not contain the 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. The notion of the dynamic program update $\bigoplus_s\mathcal{P}$ at a given state $s \in S$ changes that.

**Definition 6.2** ***(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 the following:*

**($\mathbf{CS}_s$) Current State Rules:**

$$A \leftarrow A_s \tag{50}$$

$$A^- \leftarrow A_s^- \tag{51}$$

$$not\, A \leftarrow A_s^- \tag{52}$$

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

*In particular, if the set $S$ has the largest element* max *then we simply write $\bigoplus\mathcal{P}$ instead of $\bigoplus_{max}\mathcal{P}$.*

Observe that although for any particular state $s$ the program $\biguplus\mathcal{P}$ is not required to be coherent, the program update $\bigoplus_s\mathcal{P}$ at the state $s$ must be coherent (see Proposition 3.1).

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 $P_1$ and $P_2$ be arbitrary generalized logic programs and let $S = \{1,2\}$. The dynamic program update $\bigoplus\{P_1, P_2\} = \bigoplus_2\{P_1, P_2\}$, at the state $s = 2$, is semantically equivalent to the program update $P_1 \oplus P_2$ defined in Section 3.*

*Proof.* The dynamic program update $\bigoplus\{P_1, P_2\}$ contains the following rules.

Rewritten program rules:

$$A_{P_s} \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$$

$$A_{P_s}^- \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$$

Update and inheritance rules:

$$\begin{array}{lcl} A_1 & \leftarrow & A_{P_1} \\ A_1^- & \leftarrow & A_{P_1}^- \end{array} \qquad \begin{array}{lcl} A_2 & \leftarrow & A_{P_2} \\ A_2^- & \leftarrow & A_{P_2}^- \end{array}$$

$$\begin{array}{lcl} A_1 & \leftarrow & A_0, not\, A_{P_1}^- \\ A_1^- & \leftarrow & A_0^-, not A_{P_1} \end{array} \qquad \begin{array}{lcl} A_2 & \leftarrow & A_1, not\, A_{P_2}^- \\ A_2^- & \leftarrow & A_1^-, not A_{P_2} \end{array}$$

Default rules:

$$A_0^-,$$

Current state rules:

$$A \leftarrow A_2$$

$$A^- \leftarrow A_2^-$$

$$not\, A \leftarrow A_2^-$$

The rewritten program rules are the same as the corresponding rules in $P_1 \oplus P_2$. By eliminating $A_0$'s and $A_2$'s, the remaining rules reduce to:

$$\begin{array}{rcl} A_1 & \leftarrow & A_{P_1} \\ A_1^- & \leftarrow & A_{P_1}^- \end{array} \qquad \begin{array}{rcl} A & \leftarrow & A_{P_2} \\ A^- & \leftarrow & A_{P_2}^- \end{array}$$

$$\begin{array}{rcl} not\, A & \leftarrow & A^- \\ A_1^- & \leftarrow & not A_{P_1} \end{array} \qquad \begin{array}{rcl} A & \leftarrow & A_1, not\, A_{P_2}^- \\ A^- & \leftarrow & A_1^-, not A_{P_2} \end{array}$$

By further eliminating $A_1$'s, the above rules reduce to:

$$\begin{array}{rcl} A & \leftarrow & A_{P_2} \\ A^- & \leftarrow & A_{P_2}^- \end{array}$$

$$\begin{array}{rcl} not\, A & \leftarrow & A^- \\ A^- & \leftarrow & not A_{P_1},\ not A_{P_2} \end{array} \qquad \begin{array}{rcl} A & \leftarrow & A_{P_1}, not\, A_{P_2}^- \\ A^- & \leftarrow & A_{P_1}^-, not A_{P_2} \end{array}$$

and thus coincide with the remaining rules in $P_1 \oplus P_2$, which completes the proof.□

## 6.1 Examples

**Example 6.1** *Let $\mathcal{P} = \{P_1, P_2, P_3\}$, where $P_1$, $P_2$ and $P_3$ are as follows:*

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

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

$$\begin{array}{ll} RP_1 : & sleep_{P_1} \leftarrow tv\_on^- \\ & watch\_tv_{P_1} \leftarrow tv\_on \\ & tv\_on_{P_1} \leftarrow \\ \\ RP_2 : & tv\_on_{P_2}^- \leftarrow power\_failure \\ & power\_failure_{P_2} \leftarrow \\ \\ RP_3 : & power\_failure_{P_3}^- \leftarrow \end{array} \tag{53}$$

*and the dynamic program update at the state $s$ is $\bigoplus_s \mathcal{P} = \biguplus \mathcal{P} \cup (CS_s)$. Consequently, as intended, $\bigoplus_1 \mathcal{P}$ has a single stable model $M_1 = \{tv\_on, watch\_tv\}$; $\bigoplus_2 \mathcal{P}$ has a single stable model $M_2 = \{sleep, power\_failure\}$ and $\bigoplus \mathcal{P} = \bigoplus_3 \mathcal{P}$ has a single stable model $M_3 = \{tv\_on, watch\_tv\}$ (all models modulo irrelevant literals). Moreover. $\bigoplus_2 \mathcal{P}$ is semantically equivalent to $P_1 \oplus P_2$.* □

As mentioned in the introduction, in dynamic logic programming, logic program modules describe states of our knowledge of the world, where different states may represent different time points or different sets of priorities or even different viewpoints. It is not our purpose in this paper to discuss in detail how to apply dynamic logic programming to any of these application domains[6]. However, since all of the examples presented so far relate different program modules with changing time, below we illustrate how to use dynamic logic programming to represent the well known problem in the domain of taxonomies by using priorities among rules.

**Example 6.2** *Consider the well-known problem of flying birds. In this example we have several rules with different priorities. First, the animals-do-not-fly rule, which has the lowest priority; then the birds-fly rule with a higher priority; the penguins-do-not-fly rule with an even higher priority; and, finally, with the highest priority, all the rules describing the actual taxonomy (penguins are birds, birds are animals, etc.). This can be coded quite naturally in dynamic logic programming:*

$$
\begin{array}{ll}
P_1: & not\,fly(X) \leftarrow animal(X) \\
P_2: & fly(X) \leftarrow bird(X) \\
P_3: & not\,fly(X) \leftarrow penguin(X) \\
P_4: & animal(X) \leftarrow bird(X) \\
 & bird(X) \leftarrow penguin(X) \\
 & animal(pluto) \\
 & bird(duffy) \\
 & penguin(tweety)
\end{array}
$$

The reader can easily check that, as intended, the dynamic logic program at state 4, $\bigoplus_4 \{P_1, P_2, P_3, P_4\}$, has a single stable model where $fly(duffy)$ is true, and both $fly(pluto)$ and $fly(tweety)$ are false. The reader can also use the implementation of dynamic updates (available at `http://www-ssdi.di.fct.unl.pt/~jja/updates/`) to verify this claim. □

Sometimes it is useful to have some kind of a *background knowledge*, i.e., knowledge that is true in every program module or state. This is true, for example, in the case of the *strong negation axioms* (SN) discussed in the previous section 5.2, because these axioms must be true in every program module. This is true as well in the case of the taxonomy rules discussed in the previous example as well as in the general case of laws in the domain of actions and effects of action. These laws must be valid in every state and at any time (for example, the law saying that if there is no power then the TV must be off).

Rules describing background knowledge, i.e., background rules, are easily representable in dynamic logic programming: if a rule is valid in every program state, simply add that rule to every program state. However, this is not a very practical, and, especially, not a very efficient way of representing background rules. Fortunately, in dynamic program updates at a given state $s$, adding a rule to every state is equivalent to adding that rule only in the state $s$:

**Proposition 6.1** *Let $\bigoplus_s \mathcal{P}$ be a dynamic program update at state $s$, and let $r$ be a rule such that $\forall P_i \in \mathcal{P}, r \in P_i$. Let $\mathcal{P}'$ be the set of logic programs obtained from $\mathcal{P}$ such that $P_s \in \mathcal{P}'$ and*

$$\forall i \neq s, P'_i = P_i - \{r\} \in \mathcal{P}' \textit{ iff } P_i \in \mathcal{P}$$

*Let $SM\left(\bigoplus_s \mathcal{P}\right)|\mathcal{K}$ (respectively $SM\left(\bigoplus_s \mathcal{P}'\right)|\mathcal{K}$) denote the set of all stable models of $\bigoplus_s \mathcal{P}$ (respectively. $\bigoplus_s \mathcal{P}'$) restricted to the language $\mathcal{K}$. Then:*

$$SM\left(\bigoplus_s \mathcal{P}\right)|\mathcal{K} = SM\left(\bigoplus_s \mathcal{P}'\right)|\mathcal{K}$$

*Proof.* We begin by proving the following lemma, which shows that if a rule belongs to two consecutive states, then it may be removed from the one with a smaller index, without affecting the stable models (restricted to $\mathcal{K}$):

[6] In fact, this is the subject of our ongoing research. In particular, the application of dynamic logic programming to the domain of actions is the subject of our ongoing research (see also [5, 4]).

**Lemma 6.1** *Let $\mathcal{P}_1 = \{P_i^1 : i \in S\}$ be such that there exists a rule r:*

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

*where L is a literal, and $r \in P_j^1$ and $r \in P_{j+1}^1$, for some $j < s$. Let $\mathcal{P}_2 = \{P_i^2 : i \in S\}$ be such that*

$$\forall i \neq j \;\; P_i^2 = P_i^1 \quad \text{and} \quad P_j^2 = P_j^1 - \{r\}$$

*Then $SM\left(\bigoplus_s \mathcal{P}_1\right)|\mathcal{K} = SM\left(\bigoplus_s \mathcal{P}_2\right)|\mathcal{K}$.*

*Proof.* The proof is made by separately proving that:

1. for every stable model $M_1$ of $\bigoplus_s \mathcal{P}_1$, there exists a stable model $M_2$ of $\bigoplus_s \mathcal{P}_2$ such that $M_2|\mathcal{K} = M_1|\mathcal{K}$;
2. and, for every stable model $M_2$ of $\bigoplus_s \mathcal{P}_2$, there exists a stable model $M_1$ of $\bigoplus_s \mathcal{P}_1$ such that $M_1|\mathcal{K} = M_2|\mathcal{K}$.

Each one of these is proven by constructing, for the various possible $M_1$ (respectively. $M_2$, for the second item) the corresponding $M_1$ (respectively. $M_2$).

First note that, by definition of *dynamic program update at a given state*, $\bigoplus_s \mathcal{P}_1 = \bigoplus_s \mathcal{P}_2 \cup \{r_j\}$, where $r_j$ is[7]:

$$L_{P_j} \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$$

1. Let $M_1$ be a stable model of $\bigoplus_s \mathcal{P}_1$.

   If $\{B_1, \ldots, B_m, C_1^-, \ldots, C_n^-\} \not\subseteq M_1$ then it is clear that removing from a program a (definite) rule whose body is false, does not affect the stable model. In fact:

   $$Least\left(\frac{\bigoplus_s \mathcal{P}_1}{M_1}\right) = Least\left(\frac{\bigoplus_s \mathcal{P}_1 - \{r_j\}}{M_1}\right).$$

   So, given that $r_j$ does not contain any default literals, $M_1$ itself is also a stable model of $\bigoplus_s \mathcal{P}_2$.

   If $\{B_1, \ldots, B_m, C_1^-, \ldots, C_n^-\} \subseteq M_1$ then $\{L_{P_j}, L_{P_{j+1}}\} \subseteq M_1$ and also, given the update rules (UR), $\{L_j, L_{j+1}\} \subseteq M_1$ . Moreover, assume that $M_1$ itself is not a stable model of $\bigoplus_s \mathcal{P}_2$ (otherwise the lemma is obviously satisfied). In this case, a stable model of $\bigoplus_s \mathcal{P}_2$ cannot contain $L_{P_j}$ (note that the only difference between $\bigoplus_s \mathcal{P}_1$ and $\bigoplus_s \mathcal{P}_2$ is that the latter does not have a rule whose head is $L_{P_j}$).

   The only rules, in either $\bigoplus_s \mathcal{P}_1$ or $\bigoplus_s \mathcal{P}_2$, with $L_{P_j}$ in the body are[8]:

   $$(r_1)\; L_j^- \leftarrow L_{j-1}^-, not\, L_{P_j} \qquad \text{and} \qquad (r_2)\; L_j \leftarrow L_{P_j}$$

   and the only rules with either $L_j^-$ or $L_j$ in the body are:

   $$(r_3)\; L_{j+1} \leftarrow L_j, not\, L_{P_{j+1}}^- \qquad \text{and} \qquad (r_4)\; L_{j+1}^- \leftarrow L_j^-, not\, L_{P_{j+1}}$$

   Moreover, both $\bigoplus_s \mathcal{P}_1$ and $\bigoplus_s \mathcal{P}_2$ contain the rules:

   $$(r_5)\; L_{P_{j+1}} \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^- \qquad \text{and} \qquad (r_6)\; L_{j+1} \leftarrow L_{P_{j+1}}$$

[7] Where $L_{P_j}$ is $A_{P_j}$ if $L$ is an atom $A$, or $A_{P_j}^-$ if $L$ is a default literal *not* $A$.

[8] In the following rules $L^-$ should be interpreted as the complement of $L$ wrt to $-$. I.e. if $L$ is of the form $A^-$ then $L^-$ is of the form $A$.

- If $L_{j-1}^- \notin M_1$ then $M_2 = M_1 - \{L_j, L_{P_j}\}$ is a stable model of $\bigoplus_s \mathcal{P}_2$. In fact, note that the differences between $\frac{\bigoplus_s \mathcal{P}_1}{M_1}$ and $\frac{\bigoplus_s \mathcal{P}_2}{M_2}$ are that $L_j^- \leftarrow L_{j-1}^-$ belongs only to the latter, and $r_j$ only to the former. Then, $L_{j-1}^- \notin Least(\frac{\bigoplus_s \mathcal{P}_2}{M_2})$, and since both $(r_5)$ and $(r_6)$ belong to $\bigoplus_s \mathcal{P}_2$, $\{L_{P_{j+1}}, L_{j+1}\} \subseteq Least(\frac{\bigoplus_s \mathcal{P}_2}{M_2})$. Moreover, since $r_j$ does not belong to $\bigoplus_s \mathcal{P}_2$, it is clear that neither $L_j$ nor $L_{P_j}$ belong to the least model. So $M_2$ is a stable model of $\bigoplus_s \mathcal{P}_2$.
- If $L_{j-1}^- \in M_1$ then $M_2 = M_1 \cup \{L_j^-\} - \{L_j, L_{P_j}\}$ is a stable model of $\bigoplus_s \mathcal{P}_2$. This proof is similar to the one in the previous point. Simply note that, given that $L_{j+1} \in M_2$, there are no rules in $\frac{\bigoplus_s \mathcal{P}_2}{M_2}$ with $L_j^-$ in the body. In fact, the only rule in $\bigoplus_s \mathcal{P}_2$ with $L_j^-$ in the body is an inheritance rule which also has *not* $L_{j+1}$, and thus the rule is removed in the modulo operation.

In both cases $M_2|\mathcal{K} = M_1|\mathcal{K}$.

2. The proof of this point is similar to the one above, and is omitted for brevity. □

Let $\mathcal{P}^n = \{P_i^n : i \in S\}$ be such that: if $i < n$ then $P_i^n = P_i - \{r\}$; otherwise $P_i^n = P_i$. We prove by induction on $n$ that:

$$\forall n \leq s : SM\left(\bigoplus_s \mathcal{P}\right)|\mathcal{K} = SM\left(\bigoplus_s \mathcal{P}^n\right)|\mathcal{K}$$

**Base:** If $n = 0$ then $\mathcal{P} = \mathcal{P}^n$, and the stable models are trivially the same.

**Step:** By induction hypothesis, $SM\left(\bigoplus_s \mathcal{P}\right)|\mathcal{K} = SM\left(\bigoplus_s \mathcal{P}^{n-1}\right)|\mathcal{K}$. By lemma 6.1, and since $n \leq s$[9], $SM\left(\bigoplus_s \mathcal{P}^{n-1}\right)|\mathcal{K} = SM\left(\bigoplus_s \mathcal{P}^n\right)|\mathcal{K}$. So, $SM\left(\bigoplus_s \mathcal{P}\right)|\mathcal{K} = SM\left(\bigoplus_s \mathcal{P}^n\right)|\mathcal{K}$.

Since $\mathcal{P}^s = \mathcal{P}'$ (by construction of $\mathcal{P}^n$), it follows that $SM\left(\bigoplus_s \mathcal{P}\right)|\mathcal{K} = SM\left(\bigoplus_s \mathcal{P}'\right)|\mathcal{K}$. □

Consequently, such background rules need not necessarily be added to every program state. Instead, they can simply be added at the final state $s$. Such background rules are therefore similar to the axioms $CS(s)$, which are added only when the state $s$ is fixed. In particular, considering the background rules in every program state is equivalent to considering them *as part of* the axioms $CS(s)$. A more detailed discussion of the formalization and usage of background knowledge appears in our follow-up paper [4].

## 6.2 Dealing with Contradiction

One of the important and also very difficult issues involving dynamic updates is the issue of *consistency* of the updated program $P \oplus U$. As stated in Section 2, we consider a program to be *consistent* if it has at least one stable model and thus a well-defined *stable semantics*. There are two basic reasons why the updated program may not be consistent:

1. The updated generalized logic program $P \oplus U$ may contain *explicitly contradictory* information, such as $A$ and *not* $A$, and thus not have any stable models. There are basically three cases to consider:

   (a) The contradictory information may be inherited from the original program $P$, which was already inconsistent. In this case one of the possible approaches is to prevent the contradictory information from being inherited by inertia by *limiting the inheritance by inertia*. This approach is discussed in more detail in the next section. Changing the rules of inertia so that they better suit our needs is also discussed in [4].

[9] Otherwise the lemma would not be applicable.

(b) The contradictory information may be the result of the fact that the updating program $U$ is itself contradictory. This is more difficult to eliminate because the rules of the updating program $U$ must be, by definition, true in the updated program $P \oplus U$. One approach is to always require the updating program $U$ to be consistent. If such a requirement is impossible to satisfy, we could accept contradiction in the current update but prevent it from proliferating further to the subsequent updates by using the approach discussed in (a).

(c) Both the original program and the updating program $U$ may be perfectly consistent and yet the resulting updated program may contain contradictory information. This is, for example, the case when the program containing two rules: $A \longleftarrow B$ and *not* $A$ is updated with the single fact: $B$. In this case, as in the case (a), one of the possible approaches is to prevent the contradictory information from being part of the updated program by *limiting the inheritance by inertia.* This approach is also discussed in more detail in the next section. Another possibility is to establish some *priorities* between different rules in order to prevent contradiction from arising in the first place.

2. Explicit contradiction, like the one discussed in (1), can only arise when the updated program contains some rules with default negation in there heads. Thus, it cannot arise when the updated program is normal. However, as is well-known, even normal logic programs may be *implicitly inconsistent* simply because they don't have any stable models. One possible way of dealing with this problem is to replace the stable semantics by the *3-valued stable*, or, equivalently, *well-founded semantics.* Every normal logic program is known to be consistent w.r.t. the well-founded semantics, i.e., it has a well-defined well-founded semantics. In our paper [5] we show how to extend the approach presented in this paper to the 3-valued stable semantics.

There are many other possible approaches to contradiction removal in program updates and they are part of our ongoing research in this area. However, a detailed discussion of this subject goes beyond the scope of the current paper.

### 6.3 Limiting the Inheritance by Inertia

Inheritance rules (IR) describe *the rules of inertia*, i.e., the rules guiding the inheritance of knowledge from one state $s$ to the next state $s'$. Specifically, they prevent the inheritance of knowledge that is explicitly contradicted in the new state $s'$. However, inheritance can be limited even further, by means of a simple modification of the inheritance rules:

**Modified Inheritance Rules (IR'):**

$$A_s \leftarrow A_{s-1}, not\, reject(A_{s-1}); \tag{54}$$

$$A_s^- \leftarrow A_{s-1}^-, not\, reject(A_{s-1}^-) \tag{55}$$

$$reject(A_{s-1}) \leftarrow A_{P_s}^-; \tag{56}$$

$$reject(A_{s-1}^-) \leftarrow A_{P_s} \tag{57}$$

obtained by adding new predicates $reject(A_s)$ and $reject(A_s^-)$ allowing us to specify additional restrictions on inheritance.

One important example of such additional constraints imposed on the inertia rules involves removing from the current state $s'$ of any inconsistency that occurred in the previous state $s$. Such inconsistency could have already existed in the previous state $s$ or could have been caused by the new information added at the current state $s'$. In order to eliminate such contradictory information, it suffices to add to the definition of *reject* the following two rules:

$$reject(A_{s-1}) \leftarrow A^{-}_{s-1} \tag{58}$$

$$reject(A^{-}_{s-1}) \leftarrow A_{s-1} \tag{59}$$

Similarly, the removal of contradictions brought about by the strong negation axioms of 5.1 can be achieved by adding the rules:

$$reject(A_{s-1}) \leftarrow -A_{s-1} \tag{60}$$

$$reject(-A_{s-1}) \leftarrow A_{s-1} \tag{61}$$

Other conditions and applications can be coded in this way. In particular, suitable rules can be used to enact preferences, to ensure compliance with integrity constraints or to ensure non-inertiality of fluents. Also, more complex contradiction removal criteria can be similarly coded[10].

# 7 Conclusions and Future Work

We defined a program transformation that takes two generalized logic programs $P$ and $U$, and, produces the updated logic program $P \oplus U$ resulting from the update of program $P$ by $U$. We provided a complete characterization of the semantics of program updates $P \oplus U$ and we established their basic properties. Our approach generalizes the so called *revision programs* introduced in [12]. Namely, in the special case when the initial program is just a set of facts, our program update coincides with the justified revision of [12]. In the general case, when the initial program also contains rules, our program updates characterize precisely which of these rules remain valid by inertia, and which are rejected. We also showed how strong (or *"classical")* negation can be easily incorporated into the framework of program updates.

With the introduction of dynamic program updates, we have 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 undergoes a sequence of modifications. This opens up the possibility of incremental design and evolution of logic programs, leading to the paradigm of *dynamic logic programming.* We believe that dynamic programming significantly facilitates *modularization* of logic programming and, thus, modularization of non-monotonic reasoning as a whole.

A specific application of dynamic logic programming that we intend to explore, is the evolution and maintenance of software specifications. By using logic programming as a specification language, dynamic programming provides the means of representing the evolution of software specifications.

However, ordered sets of program modules need not necessarily be seen as just a temporal evolution 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 *"union"* of the modules, subject to the given priorities. We intend to further study the relationship between dynamic logic programming and other preference-based approaches to knowledge representation.

Although not explored in 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 $s$.

[10] In all such cases, the semantic characterization of program updates would have to be adjusted accordingly to account for the change in their definition. However, pursuance of this topic is outside of the scope of the present paper.

We have already generalized our approach and results to the *3-valued* case, which enables us to update programs under the well-founded semantics [5]. We have also already developed a working implementation for the 3-valued case with top-down querying (available at `http://www-ssdi.di.fct.unl.pt/~jja/updates/`).

Our approach to program updates has grown out of our research on representing non-monotonic knowledge by means of logic programs. We envisage enriching it in the near future with other dynamic programming features, such as abduction and contradiction removal. Among other applications that we intend to study are productions systems modeling, reasoning about concurrent actions and active and temporal databases (some preliminary results are already published in [5]).

**Acknowledgments**

The extended abstract of this paper appeared in the Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Morgan Kaufmann, 1998, pp. 98-111.

The authors are grateful to the anonymous referees for their helpful comments.

This work was partially supported by PRAXIS XXI project MENTAL, by JNICT project ACROPOLE, by the National Science Foundation grant # IRI931-3061, and a NATO scholarship while L. M. Pereira was on sabbatical leave at the Department of Computer Science, University of California, Riverside. The work of J. A. Leite was supported by PRAXIS Scholarship no. BD/13514/97.

# 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 Workshop 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] Jose Alferes, Luís M. Pereira, and Teodor C. Przymusinski. *"Classical" negation in non-monotonic reasoning and logic programming. Journal of Automated Reasoning*, 20:107–142, 1998.

[4] Jose J. Alferes, Luis M. Pereira, Halina Przymusinska, and Teodor C. Przymusinski. *LUPS - a language for updating logic programs*, Proceedings of the Fifth International Conference on Logic Programming and Non-Monotonic Reasoning, December 2-4, 1999, to appear.

[5] Jose J. Alferes, Luis M. Pereira, Halina Przymusinska, and Teodor C. Przymusinski. *Preliminary exploration of actions as updates,* Proceedings of the Joint Conference on Declarative Programming (APPIA-GULP-PRODE'99), L'Aquila, Italy, September 6-9, 1999, to appear.

[6] M. Gelfond and V. Lifschitz. *The stable model semantics for logic programming.* In R. Kowalski and K. A. Bowen, editors. Fifth International Logic Programming Conference, pages 1070-1080. MIT Press, 1988.

[7] M. Gelfond and V. Lifschitz. *Logic Programs with classical negation.* In Warren and Szeredi, editors, 7th International Logic Programming Conference, pages 579-597. MIT Press, 1990.

[8] 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: Proceedings of the Second International Conference (KR91), pages 230-237, Morgan Kaufmann 1991.

[9] João A. Leite. *Logic Program Updates.* M.Sc. Dissertation, Universidade Nova de Lisboa, 1997.

[10] J. A. Leite and L. M. Pereira. *Generalizing updates: from models to programs.* In: "Logic Programming and Knowledge Representation", LNAI 1471, Berlin, 1998. Springer Verlag. Proceedings of the Third International Workshop, LPKR'97, Port Jefferson, NY, October 1997, Jurgen Dix, Luis M. Pereira, and Teodor C. Przymusinski, editors.

[11] V. Lifschitz and T. Woo. *Answer sets in general non-monotonic reasoning (preliminary report).* In B. Nebel, C. Rich and W. Swartout, editors, Principles of Knowledge Representation and Reasoning, Proceedings of the Third International Conference (KR92), pages 603-614. Morgan-Kaufmann, 1992

[12] 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.

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

[14] Teodor C. Przymusinski and Hudson Turner. Update by means of inference rules. *Journal of Logic Programming*, 30(2):125–143, 1997.

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