Update-programs can update programs

José Júlio Alferes[1] and Luís Moniz Pereira[2]

[1] DM, U. Évora and CITIA, U. Nova de Lisboa 2825 Monte da Caparica, Portugal
[2] DCS and CITIA, U.Nova de Lisboa 2825 Monte da Caparica, Portugal

Abstract. In the recent literature the issue of program change via updating rules (also known as revision rules) has been reduced to the issue of obtaining a new set of models, by means of the update rules, from each of the models of an initial program. Any program whose models are exactly the new set of models will count as an update of the original program. Following the classical approaches to theory updating, it is of course essential to start by specifying precisely how a program's models are to change, before even attempting to specify program change. But to stop there is to go only halfway.
Another limitation of existing approaches to logic program updating concerns their not dealing with 3-valuedness, i.e. with partial models. The limitation is twofold: on the one hand, only programs under 2-valued semantics are approachable; on the other, when there are contradictory update rules, in lieu of leaving undefined the effects of the contradictory rules and keeping those of the others, no update is possible at all.
In this paper, we generalize the notion of justified update to partial (or 3-valued) interpretations and expound a correct transformation on normal programs which, from an initial program, produces another program whose models enact the required change in the initial program's models, as specified by the update rules. Forthwith, we generalize our approach to logic programs as well as update programs extended with explicit negation.

1 Introduction

Logic program evolution by specifying update rules has hardly been studied. As the world changes so must programs that represent knowledge about it. Whereas simple fact by fact updates have long been addressed [9, 10, 6], providing transitional rules to govern the change of one program into another is still a blind spot in the research literature. The overall purpose of this paper is to show how transition rules for updating a logic program can be specified by some other logic program.

Program updating is distinct from program revision, where a program accommodates, perhaps non-monotonically by revising assumptions, additional information about a *static* world state. Work on program revision (or contradiction removal) has received more attention (e.g. in [1, 8, 2, 19, 18]) then transitional updating. The following realistic situation chisels the differences between program update and revision crisply.

Example 1. My secretary has just booked me on a flight from here to London on wednesday but can't remember to which airport, Gatwick or Heathrow. Clearly, this statement can be represented by:

$$booked\_for\_gatwick \vee booked\_for\_heathrow$$

Now someone tells me there never are flights from here to Gatwick on wednesday, i.e. I'm told $\neg booked\_for\_gatwick$. I conclude that I'll be flying to Heathrow, i.e. $booked\_for\_heathrow$. This is knowledge revision. The state of the world hasn't changed with respect to the flight information, but on obtaining more information I have thereby revised accordingly my knowledge about that same state of the world.

Alternatively, I hear on the radio that all flights to Gatwick on wednesday have been canceled, i.e. the world changed such that now $\neg booked\_for\_gatwick$ holds. I'm at a loss regarding whether I still have a flight to London on wednesday. This is knowledge update. The world of flights has changed, and refining my knowledge about its previous state is inadequate: I cannot conclude that I'm booked to fly to Heathrow ($booked\_for\_heathrow$). I have obtained knowledge about the new world state but it doesn't help me to disambiguate the knowledge I had about its previous state. What I can do is pick up the phone and book me a flight to Heathrow on wedsnesday, on any airline. That will change my flight world and at the same time update my knowledge about that change. However, I'm now unsure whether I might not have two flights booked to Heathrow. But if my secretary suddenly remembers he had definitely booked me to Gatwick, then I will no longer believe I have two flights to Heathrow on wednesday.

Notice how updating may produce models that are further refinable by revision. Another example of revision is when assumptions are revised in the light of new information about the same world state. For example, the reader may have presupposed that the secretary in this example is a woman till we used the giveaway pronoun "he".

Knowledge or theory update is usually performed "model by model" [17, 11], where a set of formulae T_U is a *theory update* of T, following an update request U, iff the models of T_U result from updating each of the models of T by U. Thus a theory update is determined by the update of its models.

The same idea can be applied to logic programming: a program P_U is a *program update* of P, following an update request U, iff the models of P_U (according to some logic program semantics $\mathcal{S}$) are the result of updating each of the models of P (given by semantics $\mathcal{S}$) by U. So, to obtain P_U, first compute all models of P according to a given semantics $\mathcal{S}$; to each of these models apply the update request U to obtain a new set of models $\mathcal{M}$; P_U is then any logic program whose models are exactly $\mathcal{M}$. This process can be depicted as in Figure 1.

In the recent literature [12, 13, 3, 16], the issue of program change via updating rules (also known as revision rules) has been reduced, rather simply, to the issue of obtaining a new set of models by means of the update rules, from each of the models of the given program (i.e. only the part of the above picture which is inside the box). Any program whose models are exactly the new set of

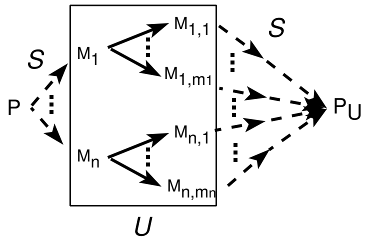

Fig. 1. Model by model theory update

models will count as an update of the original program. However, no procedure is set forth by the cited authors for obtaining one. (Except in the trivial case where the original and final programs are just sets of facts [3, 16].) Following [17, 11], it is of course essential to start by specifying precisely how a program's models are to change, before even attempting to specify program change. But to stop there is to go only halfway.

Another limitation of these first inroads into the problems of logic program change concerns their not dealing with 3-valuedness, i.e. with partial models. The limitation is twofold: on the one hand, only programs under 2-valued semantics are approachable; on the other, when there are contradictory update rules, in lieu of leaving undefined the effects of the contradictory rules and keeping those of the others, no update is possible at all. Furthermore, these authors have not considered models comprising explicit negation.

In the sequel, we begin with a short overview of previous work on interpretation updating in what concerns its basic definitions, and move on to generalize the latter to partial interpretations. Next we expound a correct transformation on normal programs which, from an initial program, produces another program whose models enact the required change in the initial program's models, as specified by the update rules. Forthwith, we generalize our approach to logic programs as well as update programs extended with explicit negation. In the end we draw some conclusions.

2 Overview of Interpretation Updates

For more detailed motivation and background to this section of the present paper the reader is referred to [12, 16]. For the purpose of our generalization we prefer the basic definitions as deployed in [12], which we will adapt: on account of the required distinction between revison and update, we will speak instead of update rule, update program, and justified update, rather than use those authors' vocabulary of revision rule, revision program, and justified revision. Otherwise the definitions are the same.

The language used is similar to that of logic programming: *update programs* are collections of *update rules*, which in turn are built of atoms by means of the special operators: $\leftarrow$, in, out, and ",".

Definition 1 (Update rules for atoms). Let U be a countable set of atoms. An *update in-rule* or, simply, an *in-rule*, is any expression of the form:

$$in(p) \leftarrow in(q_1), \ldots, in(q_m), out(s_1), \ldots, out(s_n) \tag{1}$$

where p, q_i, $1 \leq i \leq m$, and s_j, $1 \leq j \leq n$, are all in U, and $m, n \geq 0$.

An *update out-rule* or, simply, an *out-rule*, is any expression of the form:

$$out(p) \leftarrow in(q_1), \ldots, in(q_m), out(s_1), \ldots, out(s_n) \tag{2}$$

where p_i, q_i, $1 \leq i \leq m$, and s_j, $1 \leq j \leq n$, are all in U, and $m, n \geq 0$.

Intuitively, update programs can be regarded as operators which, given some initial interpretation I_i, produce its updated version I_u.

Example 2. Let P be the update program:

$$\begin{array}{l} in(a) \leftarrow out(b) \\ in(b) \leftarrow out(a) \end{array}$$

Assume the initial 2-valued interpretation $I_i = \langle \{\}; \{a, b\} \rangle$[3]. There are two viable 2-valued updated interpretations: $I_u = \langle \{a\}; \{b\} \rangle$ and $I_u = \langle \{b\}; \{a\} \rangle$.

Example 3. Let $P = \{out(a) \leftarrow in(a)\}$ and let $I_i = \langle \{a\}; \{\} \rangle$ be the initial interpretation. No interpretation can be regarded as the update of I_i by P. For example, imagine $I_u = \langle \{\}; \{a\} \rangle$ were such an update. Making a false could only be justified on the basis of the single update rule. But the body of that rule requires a to be true *in the final interpretation* for the rule to take effect. Similarly, $I_u = I_i$ cannot be an update as well.

Definition 2 (Necessary change). Let P be an update program with least (2-valued) model $M = \langle T_M; F_M \rangle$. The *necessary change* determined by P is the pair (I_P, O_P), where:

$$\begin{array}{c} I_P = \{a : in(a) \in T_M\} \\ O_P = \{a : out(a) \in T_M\} \end{array}$$

If $I_P \cap O_P = \{\}$ then P is said *coherent*.

Intuitively, the necessary change determined by a program P specifies those atoms that must be added and those atoms that must be deleted, whatever the initial interpretation.

Example 4. Take the update program $P = \{out(b) \leftarrow out(a); \quad in(b); \quad out(a)\}$. The necessary changes are irreconcilable and P is incoherent.

---

[3] In the sequel, an interpretation I will be represented as a pair $I = \langle T; F \rangle$, where T and F are disjoint sets of atoms. Atoms in T are true in I; atoms in F are false in I; all other atoms are undefined in I. An interpretation I is dubbed 2-valued (or total) iff there are no undefined atoms.

Definition 3 (Justified update[4]). Let P be an update program and $I_i = \langle T_i; F_i \rangle$ and $I_u = \langle T_u; F_u \rangle$ two (total) interpretations, whose true and false atoms are made explicit. The *reduct* $P_{I_u|I_i}$ with respect to I_i and I_u is obtained by the following operations:

- Removing from P all rules whose body contains some $in(a)$ and $a \in F_u$;
- Removing from P all rules whose body contains some $out(a)$ and $a \in T_u$;
- Removing from the body of remaining rules of P all $in(a)$ such that $a \in T_i$;
- Removing from the body of remaining rules of P all $out(a)$ such that $a \in F_i$.

Whenever P is coherent, I_u is a P-justified update of I_i if the two update stability conditions hold:

$$T_u = (T_i - O_{P_{I_u|I_i}}) \cup I_{P_{I_u|I_i}}$$
$$F_u = (F_i - I_{P_{I_u|I_i}}) \cup O_{P_{I_u|I_i}}$$

The first two operations delete rules which are useless given $I_u = \langle T_u; F_u \rangle$. Because of stability, the initial interpretation is preserved as much as possible in the final one. The last two rules achieve this because any exceptions to preservation are explicitly dealt with by the union and difference operations in the two stability conditions.

With its insistance on total interpretations, the above definition runs into problems:

Example 5. Let $I_i = \langle \{a, b\}; \{c\} \rangle$. Let $P = \{in(c) \leftarrow; \quad out(b) \leftarrow\}$. Clearly, the meaning of this program is that c must be made true and b must be made false. Hence $I_u = \langle \{a, c\}; \{b\} \rangle$. Now add to P the rule $out(a) \leftarrow in(a)$. There is no longer any justified update, even though we might want to retain the results concerning b and c while remaining undecided about a, i.e. obtain the partial interpretation $\langle \{c\}; \{b\} \rangle$.

Example 6. Consider again the update program P of Example 2 and the same initial interpretation, $I_i = \langle \{\}; \{a, b\} \rangle$. There is no P-justified update of I_i corresponding to the final empty interpretation $I_u = \langle \{\}; \{\} \rangle$, even though it would be desirable to have an update result that remained undecided between the alternatives of making a or making b true.

Furthermore, the insistance on total interpretations allows less freedom in the modelling of knowledge about the world since, often, our knowledge about it is incomplete to start with.

3 Updates of Partial Interpretations

To remedy the above shortcomings we have generalized the definitions of necessary update and justified update to cope with partial interpretations, whilst preserving their results in the case of total ones.

[4] This slightly different formulation is clearly equivalent to the original one.

Example 7. Consider the following logic program P:

$$\begin{aligned} gatwick &\leftarrow not\ heathrow \\ heathrow &\leftarrow not\ gatwick \end{aligned}$$

representing two conflicting defaults about a flight being booked for Gatwick or Heathrow, and consider also the update request $\{out(gatwick)\}$.

The stable models of P are:

$$\langle\{gatwick\};\{heathrow\}\rangle \qquad \text{and} \qquad \langle\{heathrow\};\{gatwick\}\rangle$$

the latter stating that the flight is booked for Heathrow and not for Gatwick, and the former that the flight is booked for Gatwick and not for Heathrow. The P-justified updates of these models by $\{out(gatwick)\}$ are, respectively, $\langle\{\};\{heathrow, gatwick\}\rangle$ and $\langle\{heathrow\};\{gatwick\}\rangle$, the former stating that the flight is booked for neither Heathrow nor Gatwick, and the latter as before.

If the well-founded semantics is used instead, another model of P exists, viz. $\langle\{\};\{\}\rangle$, stating that one remains undecided about the flight being booked either for Gatwick or for Heathrow. Intuitively, one expects that the above update request should not modify this model of P in what regards *heathrow*, but should falsify *gatwick*. However, the definitions presented before cannot come to this conclusion, since they do not apply to partial models.

Example 8. Consider the initial interpretation $\langle\{\};\{heathrow, gatwick\}\rangle$, and the update program:

$$\begin{aligned} in(gatwick) &\leftarrow out(heathrow) \\ in(heathrow) &\leftarrow out(gatwick) \end{aligned}$$

stating that, in the final interpretations, if the flight is not booked for Heathrow then it must be booked for Gatwick, and if the flight is not booked for Gatwick, then it must be booked for Heathrow.

The previous definition yields two updated interpretations:

$$\langle\{gatwick\};\{heathrow\}\rangle \qquad \text{and} \qquad \langle\{heathrow\};\{gatwick\}\rangle$$

whose readings can be found in Example 7. Intuitively, the partial interpretation $\langle\{\};\{\}\rangle$, stating that one is undecided about both the flight being booked for Gatwick or for Heathrow, is also expectable as a result of the update. In fact, before the update one is sure that the flight is booked for neither airport. After the update, because of the conflict between the update rules, one expectable situation is to be undecided whether the flight is booked for Heathrow or for Gatwick.

This example shows that update programs should desirably be able to undefine atoms, i.e. it might then happen that a P-justified update of a model I has more undefined atoms than I: an update may cause additional undefinedness. To cater for this possible behaviour, the definition of necessary change must be modified. Before, necessary change was simply determined by a set of atoms that must become true (I_P), and a set of atoms that must become false (O_P). In the generalization below, necessary change is also defined by a set of atoms, NI_P, that cannot remain true (even if they do not become false), and a set of atoms, NO_P, that cannot remain false (even if they do not become true).

Definition 4 (3-valued necessary change). Let P be an update program with least 3-valued model $M = \langle T_M; F_M \rangle$.
The *necessary change* determined by P is the tuple (I_P, O_P, NI_P, NO_P), where

$$\begin{aligned} I_P &= \{a : in(a) \in T_M\} \\ O_P &= \{a : out(a) \in T_M\} \\ NI_P &= \{a : out(a) \notin F_M\} \\ NO_P &= \{a : in(a) \notin F_M\} \end{aligned}$$

Atoms in I_P (resp. O_P) are those that must become true (resp. false). Atoms in NI_P (resp. NO_P) are those that cannot remain true (resp. false). If $I \cap O = \{\}$ then P is said *coherent*.

In the sequel, we omit the index P whenever the update program is clear from the context. Note that, by definition of 3-valued interpretation, $I = \langle T; F \rangle$, and T and F are disjoint, and so $I_P \subseteq NO_P$ and $O_P \subseteq NI_P$; i.e., as expected, atoms that must become true cannot remain false, and atoms that must become false cannot remain true.

Intuitively, $out(a) \notin F_M$ means that $out(a)$ is either true or undefined in M, but definitely not false. Thus, a should not remain true. If, additionally, $out(a) \in T_M$ then the a should also become false. Similar arguments justify the definition of NO_P.

3-valued justified updates rely on this new definition of necessary change, which falls back on the previous one for 2-valued interpretations. Only the third and fourth operations below are new, forged to deal with the effect on update rules of undefined literals in I_u.

Definition 5 (3-valued justified update). Let P be an update program and $I_i = \langle T_i; F_i \rangle$ and $I_u = \langle T_u; F_u \rangle$ two partial (or 3-valued) interpretations. The *reduct* $P_{I_u|I_i}$ with respect to I_i and I_u is obtained by the following operations, where **u** is a reserved atom undefined in every interpretation:

- Removing from P all rules whose body contains some $in(a)$ and $a \in F_u$;
- Removing from P all rules whose body contains some $out(a)$ and $a \in T_u$;
- Substituting in the body of remaining rules of P the reserved atom **u** for every $in(a)$, such that a is undefined in I_u[5] and $a \in T_i$;
- Substituting in the body of remaining rules of P the reserved atom **u** for every $out(a)$ such that a is undefined in I_u and $a \in F_i$;
- Removing from the body of remaining rules of P all $in(a)$ such that $a \in T_i$;
- Removing from the body of remaining rules of P all $out(a)$ such that $a \in F_i$.

[5] I.e. $a \notin T_u$ and $a \notin F_u$.

Let (I, O, NI, NO) be the 3-valued necessary change determined by $P_{I_u|I_i}$. Whenever P is coherent, I_u is a P-justified update of I_i if the two stability conditions hold:

$$T_u = (T_i - NI) \cup I$$
$$F_u = (F_i - NO) \cup O.$$

The two new operations are justified as follows. In a rule with $in(a)$ in its body, where a is undefined in I_u and $a \notin T_i$, $in(a)$ cannot be true, and also it cannot be substituted by undefined for, otherwise, the irrelevant rule $in(a) \leftarrow in(a)$, which must be possible to add to any update program inconsequently, would become $in(a) \leftarrow \mathbf{u}$, with the paradoxical effect of including a in NO for subtraction from F_i. More generally, in such a case $in(a)$ cannot become undefined in the reduct of P by virtue of undefining it in the body of a rule. Not so if $a \in T_i$. In that case, $in(a)$ in a rule body can and should be replaced by $\mathbf{u}$ to test stability. Finally, because of the inertial character of the stability conditions, any literal undefined in I_i will remain so in I_u, unless explicitly added as true or as false by the reduct of P. So no inertia operations are called for regarding undefined literals in I_i.

A similar, symmetric, reasoning applies in the case of occurences of $out(a)$ in the rule bodies.

Example 9. Consider the update program $P = \{out(gatwick)\}$ and an initial interpretation where both *gatwick* and *heathrow* are undefined, i.e. $\langle\{\};\{\}\rangle$.

Trivially, the reduct of P is equal to P, independently of the final interpretation: in the reduct all modifications to the original program are conditional on literals in the body of update rules, and P has none. The least model of P is:

$$\langle\{out(gatwick)\};\{in(gatwick), in(heathrow), out(heathrow)\}\rangle$$

Thus, the necessary change is $I = \{\}$, $O = \{gatwick\}$, $NI = \{gatwick\}$, $NO = \{\}$. Accordingly, the only P-justified update of $\langle\{\};\{\}\rangle$ is $\langle\{\};\{gatwick\}\rangle$.

Example 10. Consider the update program P of Example 8, and the initial model

$$I_i = \langle\{\};\{heathrow, gatwick\}\rangle.$$

It is easy to check that the two P-justified updates

$$\langle\{gatwick\};\{heathrow\}\rangle \quad \text{and} \quad \langle\{heathrow\};\{gatwick\}\rangle,$$

obtained as per Definition 3, are also 3-valued justified updates. Moreover, $I_u = \langle\{\};\{\}\rangle$ is also a 3-valued justified update.

In fact, the reduct with respect to this final model is:

$$in(gatwick) \leftarrow \mathbf{u}$$
$$in(heathrow) \leftarrow \mathbf{u}$$

Note that *heathrow* is undefined in I_u and false in I_i. So $out(heathrow)$ was replaced by $\mathbf{u}$. Similarly for $out(gatwick)$.

The reduct's least model is $\langle\{\};\{out(gatwick), out(heathrow)\}\rangle$. Thus the necessary change is $I = O = NI = \{\}$, $NO = \{gatwick, heathrow\}$. Given that $\{\} = (\{\} - NI) \cup I$, and $\{\} = (\{\} - NO) \cup O$, $\langle\{\};\{\}\rangle$ is indeed a 3-valued justified update.

Example 11. Consider again the same update program, but now the initial model

$$I_i = \langle\{gatwick, heathrow\};\{\}\rangle.$$

In this case the only justified update is I_i itself. Note that $I_u = \langle\{\};\{\}\rangle$ is not a justified update.

In this case the reduct of P with respect to I_i and I_u is:

$$\begin{array}{l} in(gatwick) \leftarrow out(heathrow) \\ in(heathrow) \leftarrow out(gatwick) \end{array}$$

where $out(heathrow)$ was not replaced by $\mathbf{u}$ because, even though $heathrow$ is undefined in I_u, it is not false in I_i.

The least model of the reduct is:

$$\langle\{\};\{in(gatwick), in(heathrow), out(gatwick), out(heathrow)\}\rangle$$

and so $I = O = NI = NO = \{\}$. $\langle\{\};\{\}\rangle$ is not in fact a (3-valued) justified update because: $\{\} \neq (\{gatwick, heathrow\} - \{\}) \cup \{\}$.

Example 12. Consider again Example 5. Now the partial interpretation $\langle\{c\};\{b\}\rangle$ is a (3-valued) justified update of augmented P.

Example 13. Consider once again Example 6. The empty interpretation $I_u = \langle\{\};\{\}\rangle$ is indeed a (3-valued) P-justified update of $I_i = \langle\{\};\{a, b\}\rangle$ as desired. The other two updates are preserved.

Theorem 6 (Generalization of updates). *The 3-valued versions of necessary change and justified update both reduce to their 2-valued versions whenever the initial and final interpretations are total.*

Proof. If the final and initial interpretations are total then the new rules introduced in the definition of reduct are useless, and thus the definition of reduct for three-valued justified update exactly coincides with the one for a two-valued justified update. Moreover, no symbol $\mathbf{u}$ occurs in the reduct. Thus the least model of the reduct is total, and so $NI = O$ and $NO = I$. Clearly, under these conditions the stability equalities also coincide.

An update is to be performed only if necessary. In other words, if an update program does not contradict the initial model, the only justified update should equal the initial model. This is indeed the case for 3-valued justified updates[6].

[6] In [12], its authors prove that this is also the case for their (2-valued) justified updates.

Definition 7. An interpretation $M = \langle T; F \rangle$ satisfies $in(a)$ (resp. $out(a)$) iff $a \in T$ (resp. $a \in F$). It satisfies the body of a rule iff it satisfies each literal of the body, and it satisfies a rule iff whenever it satisfies the body it also satisfies the head of the rule. An interpretation M satisfies an update program P iff it satisfies all its rules.

Theorem 8. *Let M be an interpretation satisfying an update program P. Then the only 3-valued justified update of M by P is M.*

Proof. If an interpretation $M = \langle T_i; F_i \rangle$ satisfies an update program P then:

1. *It is a justified update.* I_P adds nothing to T_i, and O_P adds nothing to F_i, because any conclusion of a rule with satisfied body in the reduct of P is necessarily satisfied as well. Furthermore, NI_P can subtract nothing from T_i: first, because for any true conclusion $out(a)$, since O_P adds nothing to F_i, a cannot belong to T_i; second, because any undefined literal in M cannot introduce **u** in the rules of the reduct of P, since that requires the literal being in F_i. Similarly, NO can subtract nothing from F_i. Consequently $F_u = F_i$ and $T_u = T_i$.
2. *There is no other justified update $R = \langle T_u; F_u \rangle$.* Indeed, first note that $T_u \subseteq T_i$ and $F_u \subseteq F_i$, since I_P can add nothing to T_i and O_P can add nothing to F_i, because of the form of the justified update operations and because I_i satisfies P. Second, neither NI nor NO can subtract from I_i and F_i. Consider NI. Again, there can be no new out conclusion from the reduct of P. Also there cannot be any undefined $out(a)$ arising from the reduct of P. Such an undefined $out(a)$ would require some $in(b)$ (resp. $out(b)$) to be undefined in the body of some rule and that $b \in T_i$ (resp. $b \in F_i$). But for $in(b)$ (resp. $out(b)$) to be undefined, since $b \in T_i$ (resp. $b \in F_i$) there must be some rule with conclusion $out(b)$ (resp. $in(b)$) undefined so that b, which does not belong to R, is subtracted from T_i (resp. F_i). Applying this reasoning systematically, eventually either $out(b)$ (resp. $in(b)$) depends on $in(b)$ or on $out(b)$. If it depends on $in(b)$ then, because $b \in T_i$ and M satisfies P, $b \in F_i$ (contradiction). If it depends on $out(b)$ then $b_i \in F_i$ too (contradiction).
   The same reasoning applies to NO, showing that neither NI nor NO subtract from T_i and F_i, respectively, and consequently $R = M$.

Example 14. Consider the update program P of Example 8, and the initial model

$$I = \langle \{heathrow, gatwick\}; \{\} \rangle.$$

Since neither $out(gatwick)$ nor $out(heathrow)$ are satisfied by I, both rules of P are trivially satisfied by I. As shown in Example 11, I is in fact the only justified update of itself.

An obvious consequence of the above theorem is that tautological update rules (i.e. update rules satisfied by every interpretation, e.g. $in(L) \leftarrow in(L)$) do not modify the initial interpretation.

4 Normal Program Updates

We've seen in the introduction that, till now, program updating has been only implicitly achieved, by recourse to the updating of each of a program's models to obtain a new set of updated models, via the update rules. Following [11], any program whose models are exactly the updated models counts as a program update. But how to obtain such a program?

Our stance, while consistent with the view depicted in Figure 1, is quite different. We aim at producing a program transformation path to updating that directly obtains, from the original program, an updated program with the required models, which is similar to the first. The update program's models will be exactly those derivable, one by one, from the original program's models through the update rules. Thus we are able to sidetrack the model generation path. Furthermore, due to the similarity between both programs, any revisions targeted for the old program can be correctly performed on the new one. This new approach can be depicted as in Figure 2.

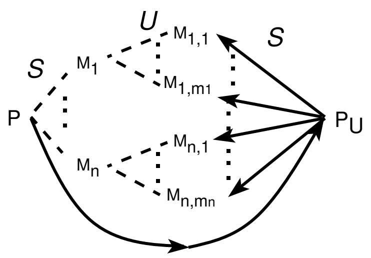

Fig. 2. Program update via program transformation

We shall see that any update program can be transformed into an extended logic program which, by the way, becomes part of the new updated program. This transformation is similar in character to the one in [16], which serves a different purpose though[7]. The normal program which is subjected to the update program, has to be transformed too. The final updated program is the union of the two transformations.

We will resort to a 3-valued well-founded semantics of extended logic programs, namely $WFSX$ [14], that introduces explicit negation in Well-Founded Semantics [7] by means of its *coherence principle*, stating that explicit negation $\neg L$ entails default negation *not* L, for every objective literal L[8]. In the case of stratified (and locally stratified) programs $WFSX$ coincides with Answer-Sets

[7] In the next section, we shall have to consider a more complex transformation, in the case where the program to be updated is also an extended logic program.

[8] An objective literal is, as usual, an atom A or its explicit negation $\neg A$.

Semantics. However, its 3-valued character is essential for dealing with partial interpretations, and its skeptical character for allowing (otherwise impossible) skeptical updates (viz. Example 13).

Definition 9 (Translation of update programs into extended LPs). Given an update program UP, its translation into the update logic program ULP is defined as follows.

- Each update in-rule of the form (1) translates into:

$$p \leftarrow q_1, \ldots, q_m, \neg s_1, \ldots, \neg s_n$$

- Each update out-rule of the form (2) translates into:

$$\neg p \leftarrow q_1, \ldots, q_m, \neg s_1, \ldots, \neg s_n$$

The rationale for this translation can best be understood in conjunction with the next definition, for they go together. Suffice it to say that we can simply equate explicit negation $\neg$ with *out*, since the programs to be updated are normal ones, and thus devoid of explicit negation (so no confusion can arise).

Definition 10 (Update transformation of a normal program). Given an update program UP, consider its corresponding update logic program ULP. For any normal logic program P, its updated program U with respect to ULP (or to UP) is obtained through the operations:

- The rules of ULP belong to U;
- The rules of P belong to U, subject to the changes below;
- For each atom A figuring in a head of a rule of ULP:
  Replace in every rule of U originated in P all occurences of A by A', where A' is a new atom;
  Include in U the rules $A \leftarrow A', not\ \neg A$ and $\neg A \leftarrow not\ A', not\ A$.

The purpose of the first operation is to ensure change according to the update program. The second operation guarantees that, for inertia, rules in P remain unchanged unless they can be affected by some update rule. The third operation changes all atoms in rules originating in P which are possibly affected by some update rule, by renaming such atoms. The new name stands for the atom as defined by the P program. The fourth operation introduces inertia rules, stating that any possibly affected atom contributes to the definition of its new version, unless actually affected through being overriden by the contrary conclusion of an update rule; the $not\ \neg A$ and $not\ A$ conditions cater for this test.

Example 15. Consider again the update program of Example 8, and the initial empty program with respect to the vocabulary $\{g, h\}$, where g stands for *gatwick* and h for heathrow. The updated logic program is U:

$$\begin{array}{lll} g \leftarrow \neg h & h \leftarrow h', not\ \neg h & g \leftarrow g', not\ \neg g \\ h \leftarrow \neg g & \neg h \leftarrow not\ h', not\ h & \neg g \leftarrow not\ g', not\ g \end{array}$$

The WFSX model of the initial (empty) program is $\langle\{\};\{g,h\}\rangle$. The WFSX models of U are $\langle\{\};\{h',g'\}\rangle$, $\langle\{g,\neg h\};\{\neg g,h,g',h'\}\rangle$, and $\langle\{\neg g,h\};\{g,\neg h,g',h'\}\rangle$. Modulo the primed and explicitly negated atoms we have:

$$M_1 = \langle\{\};\{\}\rangle \qquad M_2 = \langle\{g\};\{h\}\rangle \qquad M_3 = \langle\{h\};\{g\}\rangle$$

These models exactly correspond to the justified updates obtained in Example 10.

Example 16. Consider once more the update program of Example 8, and the initial program $P = \{g \leftarrow; \quad h \leftarrow\}$. In this case, the updated logic program is again U, of the previous example, plus the facts $\{g' \leftarrow; \quad h' \leftarrow\}$, and its only WFSX model is:

$$\langle\{g,h,g',h'\};\{\neg g,\neg h\}\rangle$$

Modulo the primed and explicitly negated atoms, this model exactly corresponds to the result of Example 11.

Example 17. Let $UP = \{in(p) \leftarrow in(q)\}$. Then $ULP = \{p \leftarrow q\}$. Take the normal program P:

$$\begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \end{array}$$

The updated logic program U of P with respect to ULP (or to UP) is U:

$$\begin{array}{lrr} p \leftarrow q \qquad & p' \leftarrow not\ q \qquad & p \leftarrow p', not\ \neg p \\ & q \leftarrow not\ p' \qquad & \neg p \leftarrow not\ p', not\ p \end{array}$$

The $WFSX$ models of P are:

$$M_0 = \langle\{\};\{\}\rangle \quad M_1 = \langle\{p\};\{q\}\rangle, \quad M_2 = \langle\{q\};\{p\}\rangle$$

The $WFSX$ models of U are:

$$M_3' = \langle\{\};\{\}\rangle \quad M_4' = \langle\{p,q\};\{p',\neg p\}\rangle \quad M_5' = \langle\{p,p'\};\{q,\neg p\}\rangle$$

Modulo the primed and explicitly negated atoms we have:

$$M_3 = \langle\{\};\{\}\rangle \quad M_4 = \langle\{p,q\};\{\}\rangle \quad M_5 = \langle\{p\};\{q\}\rangle$$

which happen to be the $UP-$justified updates of the models of P.

Example 18. Let UP be:

$$\begin{array}{l} out(q) \leftarrow out(p) \\ out(p) \leftarrow \end{array}$$

Then ULP is:

$$\begin{array}{l} \neg q \leftarrow \neg p \\ \neg p \leftarrow \end{array}$$

Take again the normal program P of Example 17. The updated logic program U of P with respect to ULP (or UP) is U:

$$
\begin{array}{lll}
\neg q \leftarrow \neg p & p' \leftarrow not\ q' & p \leftarrow p', not\ \neg p \\
\neg p \leftarrow & q' \leftarrow not\ p' & \neg p \leftarrow not\ p', not\ p \\
 & & q \leftarrow q', not\ \neg q \\
 & & \neg q \leftarrow not\ q', not\ q
\end{array}
$$

The $WFSX$ models of U are:

$$\langle\{\neg p, \neg q\}; \{p, q\}\rangle\ \langle\{\neg p, \neg q, p'\}; \{p, q, q'\}\rangle\ \langle\{\neg p, \neg q, q'\}; \{p, q, p'\}\rangle$$

Modulo the primed and explicitly negated atoms all these models coincide with $\langle\{\}; \{p, q\}\rangle$, corresponding to the single UP-justified update of the models of P.

Theorem 11 (Correctness of the update transformation). *Let P be a normal logic program and UP a coherent update program. Modulo any primed and explicitly negated elements, the $WFSX$ models of the updated program U of P with respect to UP, are exactly the (3-valued) UP-justified updates of the $WFSX$ models of P.*

Proof. In this proof with assume that every atom A of the original program has been replaced by a new atom A', and that "inertia" rules have been added for all atom. While the transformation only does so for atoms that actually occur in the head of some update rule, this is clearly a simplification, and adding them for all atoms does not change the results.

Let I be a $WFSX$ model of ULP. Then I_i represents the restriction of I to primed atoms, and I_u the remainder of I.

Given the properties of $WFSX$ (namely those of relevance and modularity), it is easy to prove that:

- If I is a $WFSX$ model of ULP then I_i is a $WFSX$ model of P.
- If I' is a $WFSX$ model of P then there exists at least one $WFSX$ model I of ULP such that $I' = I_i$.

Given this, it suffices to prove that:

1. If I is a $WFSX$ model of ULP then I_u is a justified update of I_i.
2. If I_i is a $WFSX$ model of P and I_u a justified update of I_i then $I = I_i \cup I_u$ is a $WFSX$ model of ULP.

(1) To prove this point, we begin by removing all primed atoms of ULP, by taking I_i into account, thus constructing a program $\mathcal{T}(ULP)$, as follows:

- remove all rules whose head is primed;
- for all primed atoms A' true in I_i, add to ULP a rule $A \leftarrow not\ \neg A$;
- for all primed atoms A' false in I_i, add to ULP a rule $\neg A \leftarrow not\ A$;
- for all primed atoms A' undefined in I_i, add to ULP the rules: $A \leftarrow \mathbf{u}, not\ \neg A$ and $\neg A \leftarrow \mathbf{u}, not\ A$.

$\mathcal{T}(ULP)$ corresponds to ULP after a partial evaluation with respect to I_i. Given that partial evaluation does not change the $WFSX$ semantics of programs, and that I is a $WFSX$ model of ULP, clearly I_u is a $WFSX$ model of ULP_u.

The proof proceeds by showing that[9], one by one, the modifications on $\mathcal{T}(ULP)$ corresponding to the modifications made to construct the reduct $P_{I_u|I_i}$ do not change the semantics of $\mathcal{T}(ULP)$. In the end, we conclude that I_u is a $WFSX$ model of the translation of the reduct $P_{I_u|I_i}$, as per Definition 9, plus the rules added when constructing $\mathcal{T}(ULP)$. Let R denote the resulting program.

It remains to be proven that such an I_u satisfies the two stability conditions on the reduct $P_{I_u|I_i}$. Below we prove that I_u satisfies the first stability condition. Proving that it also satisfies the second stability condition is quite similar, and is omitted for brevity.

By definition, $A \in I_{P_{I_u|I_i}}$ iff $in(A) \in least(P_{I_u|I_i})$. Since the translation of $P_{I_u|I_i}$ does not introduce any default negated literals, $in(A) \in least(P_{I_u|I_i})$ iff A belongs to the least model of the translation of the reduct. Given that I_u is a $WFSX$ model of R, and R contains the translation of the reduct, clearly $A \in T_u$.

Again by definition of justified update, $A \in T_i$ and $A \notin NI_{P_{I_u|I_i}}$ iff $A \in T_i$ and $not\ out(A) \in least(P_{I_u|I_i})$. Again, since the translation of $P_{I_u|I_i}$ does not introduce any default negated literals, $not\ out(A) \in least(P_{I_u|I_i})$ iff $not\ \neg A$ belongs to the least model of the translation of the reduct. If $A \in T_i$, then $A \leftarrow not\ \neg A \in \mathcal{T}(ULP)$, and $\mathcal{T}(ULP)$ has no rules for $\neg A$. Thus, given that I_u is a $WFSX$ model of R, $A \in T_u$.

Consequently, if $A \in (T_i - NI) \cup I$ then $A \in T_u$. If $A \in T_u$ then either there is a rule in the translation of the reduct proving A, in which case $A \in I$, or $not\ \neg A \in I_u$ and $A \in T_i$ (otherwise no enertia rule capable of proving A would exist in $\mathcal{T}(ULP)$). Thus $A \in T_i - NI$. This proves that I_u satisfies the first stability condition.

(2) The proof of this point is similar to the reverse of the previous point's proof. First note that if I_u is a justified update of I_i by the update program UP, it is also one by the reduct of the update program. The proof proceeds by showing that I_u is a $WFSX$ model of R, the translation of the reduct plus the rules in $\mathcal{T}(ULP)$. This is done similarly to the last part of the previous point's proof. The proof proceeds by showing that, one by one, the modifications on R corresponding to the reverse of the modifications made to construct the reduct $P_{I_u|I_i}$ do not change the semantics of R. After all the modifications, the resulting program clearly corresponds to ULP. Since the semantics is not changed, I_u is also a $WFSX$ model of ULP.

5 Updating models and programs with explicit negation

Since the updated programs of normal programs are, in general, extended programs, to update them in turn we need to address the issue of updating extended

[9] This part of the proof is quite long and is not presented here.

programs. This is one motivation for defining updates for models and programs with explicit negation. Another motivation is to be able to update knowledge bases represented by extended logic programs. In fact, much work has been done in representing knowledge with extended logic programs (see e.g. [4, 15]). For this knowledge to be updated one needs a definition of updates of models and of programs with explicit negation. To do so, we first generalize the language of update rules:

Definition 12 (Update rules for objective literals). Let U be a countable set of objective literals. Update in-rules or, simply, in-rules, and update out-rules or, simply, out-rules, are as per Definition 1, but with respect to this new set U.

Intuitively, an objective literal is quite similar to an atom. For example, $in(\neg a)$, simply means that $\neg a$ must become true, and this can be achieved as for atoms. The same applies to $out(\neg a)$, which means that $\neg a$ must become false. Thus, it is not surprising that the definitions of necessary change and justified update for models with explicit negation are quite similar to those for models without explicit negation.

Despite the similarities, there is a subtle but important additional condition that justified updates of models with explicit negation must obey.

Example 19. Consider the initial model (with explicit negation):

$$\langle \{gatwick, \neg heathrow\}; \{\neg gatwick, heathrow\} \rangle$$

meaning that one knows that a flight is booked for Gatwick, and no flight is booked for Heathrow.

Consider now the update program $\{in(\neg gatwick)\}$. This update means that $\neg gatwick$ must become true, i.e. $gatwick$ must become (explicitly) false. Performing this update on the initial model requires more than simply adding $\neg gatwick$: it also requires the removal of $gatwick$ from the positive part of the model. In fact, if the world changed so that $gatwick$ became explicitly false, certainly the truth of $gatwick$ will have to be removed. Intuitively, the result of this update on the initial model is $\langle \{\neg gatwick, \neg heathrow\}; \{gatwick, heathrow\} \rangle$.

If later we are told $in(heathrow)$, the result should then be:

$$\langle \{\neg gatwick, heathrow\}; \{gatwick, \neg heathrow\} \rangle.$$

What this example shows is that $in(\neg a)$ additionally requires $out(a)$, and $in(a)$ additionally requires $out(\neg a)$. This can be seen as a consistency requirement on updated models. If $\neg a$ (resp. a) must become true, then a (resp. $\neg a$) must be guaranteed absent from the truth part of the model. Instead of changing the definitions of necessary change and justified update to cope with these impositions, for simplicity we add instead, to every update program, the two update rules:

$$out(A) \leftarrow in(\neg A) \qquad \text{and} \qquad out(\neg A) \leftarrow in(A)$$

for every atom A in the language, and then determine the justified updates as before:

Definition 13 (Extended Justified Update). Let P be an update program with explicit negation, and $I_i = \langle T_i; F_i \rangle$ and $I_u = \langle T_u; F_u \rangle$ two partial interpretations with explicit negation[10].

Let $P' = P \cup \{out(L) \leftarrow in(\neg L) : L \in U\}$, where U is the set of all objective literals. I_u is a P-justified update of I_i iff it is a P'-justified update of I_i according to Definition 5, where all explicitly negated atoms are simply viewed as new atoms.

Mark that, when the initial program does include explicit negation, the update transformation for normal programs of Definition 10 does not yield an updated extended program whose models are the updates of the initial program's models. This is so because the transformation does not distinguish between making $\neg a$ true ($in(\neg a)$) and making a false ($out(a)$): both are transformed into $\neg a$. However, the definition of justified update of models with explicit negation views the two updates differently:

Example 20. Consider the initial model $\langle \{a\}; \{\neg a, b, \neg b\} \rangle$, and the update $out(a)$. It is easy to check that the only justified update is $\langle \{\}; \{a, \neg a, b, \neg b\} \rangle$, resulting from removing the truth of a.

If the update $in(\neg a)$ is considered instead, then the only justified update is $\langle \{\neg a\}; \{a, b, \neg b\} \rangle$.

In the transformation for normal programs, explicit negation $\neg A$ is used to falsify A. Since in $WFSX$ $\neg A$ entails $not\ A$, and the original programs have no objective literals of the form $\neg A$, the only outcome of having $\neg A$ is, in fact, that of imposing $not\ A$, as desired. However, if the program being updated differentiates $\neg A$ from $not\ A$, the former can no longer be used as a means to impose the latter.

Intuitively, an update out-rule $out(A) \leftarrow Body$ means that whenever $Body$ is true in the updated interpretation $not\ A$ must be true. If extended programs allowed default negated literals in the head of rules, this could be solved by transforming $out(A)$ in the head of an update rule by $not\ A$. The intended meaning of a rule $not\ L \leftarrow Body$ would then be "If the body is true then $not\ L$ must be true".

In [5] is shown that $WFSX$ is expressive enough to capture the semantics of extended logic programs with default literals in the heads of rules. This is accomplished via a program transformation, such that the partial stable models of the transformed extended program exactly correspond to those models of the program with default literals at the head, given the above intended reading for such rules. The transformed extended program P^{not} of a program P with default literals in the head is simply constructed as follows:

- For each objective literal A (resp. $\neg A$) in the language of P, program P^{not} contains the rule $A \leftarrow A^p$ (resp. $\neg A \leftarrow A^n$), where A^p (resp. A^n) is a new atom;

[10] Recall that in such interpretations, if $\neg L \in T$ then $L \in F$, for every objective literal L. See [14] for a formal definition.

- For each rule of the form $A \leftarrow Body$ (resp. $\neg A \leftarrow Body$) in P, program P^{not} contains the rule $A^p \leftarrow Body$ (resp. $A^n \leftarrow Body$);
- For each rule of the form $not\ A \leftarrow Body$ (resp. $not\ \neg A \leftarrow Body$) in P, program P^{not} contains the rule $\neg A^p \leftarrow Body$ (resp. $\neg A^n \leftarrow Body$).

The whole idea of the transformation is to allow $not\ A$ and $not\ \neg A$ conclusions, once transmuted into $\neg A^p$ and $\neg A^n$, to falsify the single rules for A and for $\neg A$.

The translation of update programs with explicit negation into extended logic programs mirrors this transformation, where atoms $out(A)$ (resp. $out(\neg A)$) are first translated into $not\ A$ (resp. $not\ \neg A$), both in the bodies and heads of rules.

Definition 14 (Translation of extended update programs into LPs). Given an update program with explicit negation UP, its translation into the update logic program ULP is defined as follows:

1. Each in-rule $in(L_0) \leftarrow in(L_1), \ldots, in(L_m), out(L_{m+1}), \ldots, out(L_n)$ where $n \geq 0$, and the L_i are objective literals, translates into:
$$L_0^* \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n$$
where $L_0^* = A^p$ if $L_0 = A$, and $L_0^* = A^n$ if $L_0 = \neg A$;
2. Each out-rule $out(L_0) \leftarrow in(L_1), \ldots, in(L_m), out(L_{m+1}), \ldots, out(L_n)$ where $n \geq 0$, and the L_i are objective literals, translates into:
$$\neg L_0^* \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n$$
where $L_0^* = A^p$ if $L_0 = A$, and $L_0^* = A^n$ if $L_0 = \neg A$;
3. For every objective literal L such that $in(L)$ belongs to the head of some in-rule of UP, ULP contains $\neg L^* \leftarrow L$ where $L^* = A^n$ if $L = A$, and $L^* = A^p$ if $L = \neg A$;
4. For every atom A occuring in the head of some update rule of UP, ULP contains $A \leftarrow A^p$ and $\neg A \leftarrow A^n$.

Items (1) and (2) directly reflect the transformation P^{not} on the rules that result from translating $in(L)$ by L, and $out(L)$ by $not\ L$. Item (3) reflects the transformation on the update rules $out(L) \leftarrow in(\neg L)$ for each objective literal L (implicitly added to all update programs). Note that if there are no rules in the update program for $in(\neg L)$ then there is no need to add the corresponding rule, as it would be useless. Item (4) simply adds the rules required by P^{not} for establishing a link between objective literals L and the corresponding L^p and L^n ones. Again, if the atom A never occurs in the heads of update rules, then there is no need for adding such rules for it.

Definition 15 (Update transformation of an extended program). Given an update program with explicit negation UP, consider its corresponding ULP. For any extended program P, its updated program U with respect to ULP (or to UP) is obtained by the operations:

– The rules of ULP belong to U;
– The rules of P belong to U, subject to the changes below;
– For each atom A occuring in the head of some update rule of UP:

Replace in every rule of U originated in P all occurences of A by A', where A' is a new atom;

Include in U the rules:

$$\begin{array}{rlrl} A^p & \leftarrow A', not\ \neg A^p & \qquad A^n & \leftarrow \neg A', not\ \neg A^n \\ \neg A^p & \leftarrow not\ A', not\ A^p & \qquad \neg A^n & \leftarrow not\ \neg A', not\ A^n \end{array}$$

As before, atoms A of the initial program that may change their truth value are replaced by new atoms A'. The added rules implement inertia for those atoms. The first rule on the left states that "if A was true in the initial program, and it is not the case that $out(A)$, then A should be in" (note that, in heads, $out(A)$ is transformed into $\neg A^p$). The second rule on the left states that "if A was false in the initial program, and it is not the case that $in(A)$, then A should be out". The rules on the right express correspondingly the same, but for explicitly negated atoms.

Example 21. [11] Consider a university that periodically updates its evaluation of faculty members based on their research and teaching record. Faculty members who are known to be strong researchers and good teachers receive positive evaluation. Those who are not known to be strong researchers are not positively evaluated, and those that are poor teachers receive a negative evaluation. This leads us to the following update program, with the obvious abbreviations, and where non-ground rules stand for the set of their ground instances:

$$\begin{array}{rl} in(g\_eval(X)) & \leftarrow in(g\_res(X)), in(g\_teach(X)) \\ out(g\_eval(X)) & \leftarrow out(g\_res(X)) \\ in(\neg g\_eval(X)) & \leftarrow in(\neg g\_teach(X)) \end{array}$$

At this university, it is common knowledge that anyone who receives a teaching award is considered a good teacher, and anyone who has published many papers is considered a good researcher. It is part of the records that Scott is a good teacher, Jack has received a teaching award, Lisa and Scott have already published many papers but not Jack, and that, in the previous evaluation period, Lisa and Jack obtained good evaluations. This leads to the knowledge representation program P:

$$\begin{array}{rlrl} g\_teach(X) & \leftarrow award(X) & \qquad award(jack) & \leftarrow \\ g\_res(X) & \leftarrow many(X) & \qquad \neg many(jack) & \leftarrow \\ & & \qquad g\_eval(jack) & \leftarrow \\ g\_teach(scott) & \leftarrow & \qquad many(lisa) & \leftarrow \\ many(scott) & \leftarrow & \qquad g\_eval(lisa) & \leftarrow \end{array}$$

[11] This example is inspired on a similar one due to Teodor Przymusinski.

whose only $WFSX$ model $M = \langle T; F \rangle$ contains the following evaluation record:

$$\begin{array}{r} \{g\_eval(jack), g\_eval(lisa)\} \subset T \\ \{g\_eval(scott), \neg g\_eval(jack), \neg g\_eval(lisa)\} \subseteq F \end{array}$$

The update logic program U obtained by the transformation above is as follows:

$$\begin{array}{rcl} g\_eval^p(X) & \leftarrow & g\_res(X), g\_teach(X) \\ \neg g\_eval^p(X)) & \leftarrow & not\ g\_res(X) \\ g\_eval^n(X)) & \leftarrow & \neg g\_teach(X) \end{array}$$

$$\begin{array}{rclcrl} g\_teach(X) & \leftarrow & award(X) & \quad & award(jack) & \leftarrow \\ g\_res(X) & \leftarrow & many(X) & & \neg many(jack) & \leftarrow \\ & & & & g\_eval'(jack) & \leftarrow \\ g\_teach(scott) & \leftarrow & & & many(lisa) & \leftarrow \\ many(scott) & \leftarrow & & & g\_eval'(lisa)) & \leftarrow \end{array}$$

$$g\_eval(X) \leftarrow g\_eval^p(X) \qquad \neg g\_eval(X) \leftarrow g\_eval^n(X)$$

$$\begin{array}{rcl} g\_eval^p(X) & \leftarrow & g\_eval'(X), not \neg g\_eval^p(X) \\ \neg g\_eval^p(X) & \leftarrow & not\ g\_eval'(X), not\ \neg g\_eval^p(X) \\ g\_eval^n(X) & \leftarrow & \neg g\_eval'(X), not\ \neg g\_eval^n(X) \\ \neg g\_eval^n(X) & \leftarrow & not\ \neg g\_eval'(X), not\ \neg g\_eval^n(X) \end{array}$$

The single justified update of the only $WFSX$ model of P is the single $WFSX$ model of $U = \langle T_u; F_u \rangle$, modulo primed, p-ed and n-ed literals, containing the evaluation record:

$$\begin{array}{r} \{g\_eval(scott), g\_eval(lisa)\} \subset T_u \\ \{g\_eval(jack), \neg g\_eval(jack), \neg g\_eval(lisa)\} \subseteq F_u \end{array}$$

Scott now has a good evaluation because he has many papers (which makes him a good researcher) and he was already know to be a good teacher. Jack has no longer a good evaluation because it is known that he does not have many papers (and so he cannot be considered a good researcher). Though Lisa is not explicitly known to be a good teacher, she keeps her good evaluation record because it is neither the case that she cannot be considered a good researcher nor is it the case that she does not have many papers.

Theorem 16 (Correctness of the transformation). *Let P be an extended logic program and UP a coherent update program. Modulo any primed and A^p and A^n elements, the $WFSX$ models of the updated program U of P with respect to UP are exactly the UP-justified updates of the $WFSX$ models of P.*

The structure of this theorem's proof is similar to that of Theorem 11. The details of the proof are slightly more complex, mainly because the translation T of update rules can now contain default negated literals in the the body of rules of T. To solve this additional complexity the proof appeals to most of the arguments used in [5] to prove that, with the construction above, $WFSX$ is expressive enough to capture the semantics of extended logic programs with default literals in the heads.

6 Conclusion

In this paper we have motivated, demonstrated, and illustrated how to make more general use of update programs. Firstly, their application was extended to encompases partial interpretations. Secondly, we showed how to apply update programs directly to normal and extended logic programs, rather than to their models. For doing so, we generate an updated program whose models are exactly those that would be obtained by individually updating each model of the original program. The updated programs result from transformations, proven correct, which apply jointly to update programs and the given logic programs they operate on. Whereas uptading models may be an exponential process, due to a possible exponential number of models, our updating transformation of programs is clearly a polynomial process. The additional complexity in obtaining the models of the updated program is only exacted on demand. Furthermore successive update transformation may take place before any models are desired.

Acknowlegments

This work was partially supported by JNICT-Portugal project ACROPOLE no. 2519/TIT/95. Thanks are due to Teodor Przymusinski and Mirek Truszczyński for their comments.

References

1. J. J. Alferes, C. V. Damásio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 14:93–147, 1995.
2. J. J. Alferes and L. M. Pereira. Contradiction: when avoidance equal removal. In R. Dyckhoff, editor, *4th ELP*, volume 798 of *LNAI*. Springer–Verlag, 1994.
3. C. Baral. Rule-based updates on simple knowledge bases. In *AAAI'94*, pages 136–141, 1994.
4. C. Baral and M. Gelfond. Logic programming and knowledge representation. *J. Logic Programming*, 19/20:73–148, 1994.
5. C. V. Damásio and L. M. Pereira. Default negated conclusions: why not? In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *ELP'96*. Springer-Verlag, 1996.
6. H. Decker. Drawing updates from derivations. In *Int. Conf on Database Theory*, volume 460 of *LNCS*, 1990.
7. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

8. L. Giordano and A. Martelli. Generalized stable models, truth maintenance and conflit resolution. In D. Warren and P. Szeredi, editors, *7th ICLP*, pages 427–441. MIT Press, 1990.
9. A. Guessoum and J. W. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, 1990.
10. A. Guessoum and J. W. Lloyd. Updating knowledge bases II. *New Generation Computing*, 10(1):73–100, 1991.
11. H. Katsuno and A. Mendelzon. On the difference between updating a knowledge base and revising it. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR'91*, pages 387–394. Morgan-Kaufmann, 1991.
12. V. Marek and M. Truszczyński. 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. V. Marek and M. Truszczyński. Revision programming, database updates and integrity constraints. In *ICDT'95*, pages 368–382. Springer-Verlag, 1995.
14. L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *European Conf. on AI*, pages 102–106. John Wiley & Sons, 1992.
15. L. M. Pereira, J. N. Aparício, and J. J. Alferes. Non–monotonic reasoning with logic programming. *Journal of Logic Programming*, 17(2, 3 & 4):227–263, 1993.
16. T. Przymusinski and H. Turner. Update by means of inference rules. In V. Marek, A. Nerode, and M. Truszczyński, editors, *LPNMR'95*, volume 928 of *LNAI*, pages 156–174. Springer-Verlag, 1995.
17. M. Winslett. Reasoning about action using a possible models approach. In *AAAI'88*, pages 89–93, 1988.
18. C. Witteveen and W. Hoek. Revision by communication. In V. Marek, A. Nerode, and M. Truszczyński, editors, *LPNMR'95*, volume 928 of *LNAI*, pages 189–202. Springer-Verlag, 1995.
19. C. Witteveen, W. Hoek, and H. Nivelle. Revision of non-monotonic theories: some postulates and an application to logic programming. In C. MacNish, D. Pearce, and L. M. Pereira, editors, *JELIA'94*, volume 838 of *LNAI*, pages 137–151. Springer-Verlag, 1994.