Updating Logic Programs

Yan Zhang[1] and Norman N. Foo[2]

Abstract. It is well known that rule-based knowledge bases are usually realized by logic programs. An update has to be performed to the current knowledge base if some new information occurs. In many applications associated with dynamic domains, however, such new information may occur not only as a set of simple facts but also a set of rules. In this paper, we view knowledge bases as extended logic programs [1] such that defeasible knowledge can be explicitly represented. We then consider the problem of how to update an extended logic program with another extended logic program while the former program represents the current knowledge base and the later one represents the new information that should be added to the current knowledge base. We show how the concept of priority associated with logic programs is employed to achieve our purpose. We also investigate semantical properties of logic program updates in detail.

1 INTRODUCTION

It is a convention to express rule-based knowledge bases as logic programs. Such knowledge bases, in principle, consist of two components: facts and rules. Traditionally, facts express the current state of the knowledge base and are usually changeable, while rules express constraints about the domain and hence are usually unchangeable [2, 3]. However, in many applications associated with dynamic domains, a knowledge base has to be updated not only because some new facts occur, but also because some old rules are no longer suitable to characterize new domain constraints (e.g. the domain is evolved). Therefore, it is important to have a more powerful update mechanism to maintain a knowledge base to a stable status whenever some new information, which may be a set of simple facts as well as a set of rules, is combined into the current knowledge base.

In this paper, we view a knowledge base as an extended logic program [1] such that defeasible knowledge can be explicitly represented. We then consider the problem of how to update an extended logic program with another extended logic program, while the former program represents the current knowledge base and the later one represents the new information (or simply called *update effect*) that should be added in the current knowledge base. In particular, our problem can be described as follows: given two extended logic programs Π_0 and Π_1, what is the resulting extended logic program Π_0' after updating Π_0 with Π_1? To illustrate some key issues associated with this update problem, let us first consider the following example.

Example 1 Suppose an access control policy base for a computer system is represented by an extended logic program Π_0 consisting of following rules:

$$\begin{array}{l} Member(A,G) \leftarrow, \quad Member(B,G) \leftarrow, \\ Access(A,F_2) \leftarrow, \\ Access(x,F_1) \leftarrow Member(x,G), \\ \neg Access(x,F_2) \leftarrow Member(x,G), not\ Access(x,F_2). \end{array}$$

Intuitively, this policy base represents following explicit facts that users A and B are members of group G, and A can access file F_2. Further, it also represents two general rules saying that (1) if any user x is a member of G, then x can access file F_1; and (2) for any user x, if x is a member of group G and there is no explicit representation stating that x can access file F_2, then user x can not access file F_2. If we adopt Gelfond and Lifschitz's *answer set semantics* for extended logic programs [1], it is clear that implicit facts $Access(A,F_1)$, $Access(B,F_1)$ and $\neg Access(B,F_2)$ are actually derivable from Π_0.

Now suppose we want to update this policy base Π_0 with the following extended logic program Π_1 which represents the agent's new access control policy for the system:

$$\begin{array}{l} Member(C,G) \leftarrow, \\ \neg Access(x,F_1) \leftarrow Member(x,G), \\ Access(x,F_2) \leftarrow Member(x,G), not\ \neg Access(x,F_2). \end{array}$$

Our question is: what is the resulting policy base Π_0' after updating Π_0 with Π_1? Since Π_1 represents the agent's new information/knowledge about the domain, it is obvious that Π_1 should be included in Π_0', i.e. $\Pi_1 \subseteq \Pi_0'$.

It is also clear that rule $Access(x,F_1) \leftarrow Member(x,G)$ in Π_0 represents a contradictory meaning compared with the rule $\neg Access(x,F_1) \leftarrow Member(x,G)$ in Π_1, and if we specify $\Pi_0' = \Pi_0 \cup \Pi_1$, Π_0' is inconsistent under the answer set semantics. Hence, $Access(x,F_1) \leftarrow Member(x,G)$ should be excluded from Π_0'.

On the other hand, although rule $\neg Access(x,F_2) \leftarrow Member(x,G), not\ Access(x,F_2)$ in Π_0 does not contradict any rules in Π_1, it indeed *conflicts* with rule $Access(x,F_2) \leftarrow Member(x,G), not\ \neg Access(x,F_2)$ in Π_1 since retaining it in Π_0' will lead Π_0' to have different answer sets showing that $Access(B,F_2)$ (or $Access(C,F_2)$) is true in one case and $Access(B,F_2)$ (or $Access(C,F_2)$) is not true in the other. Again, since rules in Π_1 represent the agent's new information/knowledge and intuitively should have higher priorities than rules in Π_0, in this case we will expect that rule

[1] School of Computing and Information Technology, University of Western Sydney, Nepean, Kingswood, NSW 2747, Australia, E-mail: yan@cit.nepean.uws.edu.au
[2] School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia, E-mail: norman@cse.unsw.edu.au

ECAI'98 *13th European Conference on Artificial Intelligence*
Edited by Henri Prade
Published in 1998 by John Wiley & Sons, Ltd

$\neg Access(x, F_2) \leftarrow Member(x, G)$, *not* $Access(x, F_2)$ in Π_0 is *overridden* by rule $Access(x, F_2) \leftarrow Member(x, G)$, *not* $\neg Access(x, F_2)$ in the resulting program Π'_0.

Finally, all other rules in Π_0 are consistent and conflict-free with each rule of Π_1, and then should be kept in Π'_0 from a general principle of minimal change. Therefore, we may get a resulting policy base Π'_0 after updating Π_0 with Π_1 as follows:

$Member(A, G) \leftarrow$, $\quad Member(B, G) \leftarrow$,
$Member(C, G) \leftarrow$, $\quad Access(A, F_2) \leftarrow$,
$\neg Access(x, F_1) \leftarrow Member(x, G)$,
$Access(x, F_2) \leftarrow Member(x, G)$, *not* $\neg Access(x, F_2)$,

from which implicit facts $\neg Access(A, F_1)$, $\neg Access(B, F_1)$, $\neg Access(C, F_1)$, $Access(B, F_2)$, and $Access(C, F_2)$ are derivable. ■

The main purpose of this paper is to propose a formal method to deal with this kind of updates. As we will see next, to achieve our purpose, the concept of *priority* associated with logic programs plays an important role in our formalization. The paper is organized as follows. Section 2 introduces prioritized logic programs which are essential in our logic program updates. Section 3 then proposes a formal approach for logic program updates, while section 4 investigates relevant semantical properties of logic program updates. Finally, section 5 concludes this paper with some remarks.

2 PRIORITIZED LOGIC PROGRAMS

In this section we introduce prioritized logic programs (PLPs) proposed by Zhang and Foo recently [6]. The language $\mathcal{L}$ of PLPs is a language of extended logic programs [1] with the following augments:
- *Names:* $N, N_1, N_2, \cdots$.
- A strict partial ordering (i.e. antireflexive, antisymmetric and transitive) $<$ on names.
- A naming function $\mathcal{N}$, which maps a rule to a name.

Terms, atoms, literals and rules in PLPs are defined as the same in extended logic programs. For the naming function $\mathcal{N}$, we require that for any rules r and r' in a PLP (see the following definition), $\mathcal{N}(r) = \mathcal{N}(r')$ iff r and r' indicate the same rule.

A *prioritized logic program* (PLP) $\mathcal{P}$ is a triplet $(\Pi, \mathcal{N}, <)$, where Π is an extended logic program, $\mathcal{N}$ is a naming function mapping each rule in Π to a name, and $<$ is a set of relations representing all strict partial orderings on names. The following is an example of prioritized logic program.

$\mathcal{P}_1$:
$N_1 : P \leftarrow$ *not* Q, *not* R,
$N_2 : Q \leftarrow$ *not* P,
$N_3 : R \leftarrow$ *not* P,
$N_1 < N_2, N_2 < N_3$.

We also use notations $\mathcal{P}_1(\Pi)$, $\mathcal{P}_1(\mathcal{N})$, and $\mathcal{P}_1(<)$ to denote the sets of rules, naming function's values and $<$-relations of $\mathcal{P}_1$ respectively. Consider another program as follows:

$\mathcal{P}_2$:
$N_1 : P \leftarrow$ *not* Q, *not* R,
$N_2 : Q \leftarrow$ *not* P,
$N_3 : R \leftarrow$ *not* P,
$N_1 < N_2, N_2 < N_3, N_1 < N_3$.

Obviously, the only difference between $\mathcal{P}_1$ and $\mathcal{P}_2$ is that there is one more relation $N_1 < N_3$ in $\mathcal{P}_2$. As mentioned earlier, $<$ is a strict partial ordering (i.e., antireflexive, antisymmetric and transitive), we would expect that $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent in some sense. To make this precise, $<$-*closure* is introduced as follows.

Definition 1 *Given a program $\mathcal{P} = (\Pi, \mathcal{N}, <)$. $\mathcal{P}(<^+)$ is the* $<$-closure *of $\mathcal{P}$ iff $\mathcal{P}(<^+)$ is the smallest set containing $\mathcal{P}(<)$ and closed under transitivity.*

We say that two prioritized logic programs $\mathcal{P}_1$ and $\mathcal{P}_2$ are *equivalent* if $\mathcal{P}_1(\Pi) = \mathcal{P}_2(\Pi)$, $\mathcal{P}_1(\mathcal{N}) = \mathcal{P}_2(\mathcal{N})$, and $\mathcal{P}_1(<^+) = \mathcal{P}_2(<'^+)$. A prioritized logic program is defined as an extended logic program by associating with a partial ordering $<$ to it. Intuitively such ordering represents a preference of applying rules during the evaluation of a query of the program. In particular, if in a program $\mathcal{P}$, relation $\mathcal{N}(r) < \mathcal{N}(r')$ holds, rule r would be preferred to apply over rule r' during the evaluation of $\mathcal{P}$ (i.e. rule r is more preferred than rule r'). Consider the following classical example represented in our formalism:

$\mathcal{P}_3$:
$N_1 : Fly(x) \leftarrow Bird(x)$, *not* $\neg Fly(x)$,
$N_2 : \neg Fly(x) \leftarrow Penguin(x)$, *not* $Fly(x)$,
$N_3 : Bird(Tweety) \leftarrow$,
$N_4 : Penguin(Tweety) \leftarrow$,
$N_2 < N_1$.

Obviously, rules N_1 and N_2 conflict with each other as their heads are complementary literals, and applying N_1 will defeat N_2 and *vice versa*. However, as $N_2 < N_1$, we would expect that rule N_2 is preferred to apply first and then defeat rule N_1 after applying N_2 so that the desired solution $\neg Fly(Tweety)$ could be derived.

The semantics of PLPs is defined in terms of the answer set semantics of extended logic programs. In program $\mathcal{P}_3$, we have seen that rules N_1 and N_2 conflict with each other. Since $N_2 < N_1$, we try to solve the conflict by applying N_2 first and defeating N_1.

Definition 2 *Let Π be an extended logic program and r a rule with the form $L_0 \leftarrow L_1, \cdots, L_m$, not $L_{m+1}, \cdots$, not L_n (r does not necessarily belong to Π). Rule r is* defeated *by Π iff for any answer set $Ans(\Pi)$ of Π, there exists some $L_i \in Ans(\Pi)$, where $m + 1 \leq i \leq n$.*

The idea of evaluating a PLP is described as follows. Let $\mathcal{P} = (\Pi, \mathcal{N}, <)$. If there are two rules r and r' in $\mathcal{P}(\Pi)$ and $\mathcal{N}(r) < \mathcal{N}(r')$, r' will be ignored in the evaluation of $\mathcal{P}$, *only if* keeping r in $\mathcal{P}(\Pi)$ and deleting r' from $\mathcal{P}(\Pi)$ will result in a defeat of r', i.e. r' is defeated by $\mathcal{P}(\Pi) - \{r'\}$. By eliminating all such potential rules from $\mathcal{P}(\Pi)$, $\mathcal{P}$ is eventually reduced to an extended logic program in which the partial ordering $<$ has been removed. Our evaluation for $\mathcal{P}$ is then based on this extended logic program.

Let us consider program $\mathcal{P}_3$ once again. Since $N_2 < N_1$ and N_1 is defeated by $\mathcal{P}_3 - \{N_1\}$ (i.e. the unique answer set of $\mathcal{P}_3 - \{N_1\}$ is $\{Bird(Tweety), Penguin(Tweety), \neg Fly(Tweety)\}$), rule N_1 should be ignored during the evaluation of $\mathcal{P}_3$.

Definition 3 *Let $\mathcal{P} = (\Pi, \mathcal{N}, <)$ be a prioritized extended logic program. $\mathcal{P}^<$ is a* reduct *of $\mathcal{P}$ with respect to $<$ iff there exists a sequence of sets Π_i $(i = 0, 1, \cdots)$ such that:*

(i) $\Pi_0 = \Pi$;
(ii) $\Pi_i = \Pi_{i-1} - \{r_1, \cdots, r_k \mid$ *(a) there exists $r \in \Pi_{i-1}$ such that $\mathcal{N}(r) < \mathcal{N}(r_i) \in \mathcal{P}(<^+)$ $(i = 1, \cdots, k)$ and $r_1, \cdots, r_k$ are defeated by $\Pi_{i-1} - \{r_1, \cdots, r_k\}$, and (b) there does not exist a rule $r' \in \Pi_{i-1}$ such that $N(r_j) < N(r')$ for some j $(j = 1, \cdots, k)$ and r' is defeated by $\Pi_{i-1} - \{r'\}\}$;*
(iii) $\mathcal{P}^< = \bigcap_{i=0}^{\infty} \Pi_i$.

Clearly $\mathcal{P}^<$ is an extended logic program obtained from Π by eliminating some rules from Π. In particular, if $\mathcal{N}(r) < \mathcal{N}(r')$ and $\Pi - \{r'\}$ defeats r', rule r' is eliminated from Π if no *less preferred rule* can be eliminated (i.e. conditions (a) and (b)). This procedure is continued until a fixed point is reached. Note that due to the transitivity of $<$, we need to consider each $\mathcal{N}(r) < \mathcal{N}(r')$ in the $<$-closure of $\mathcal{P}$. It should be also noted that the reduct of a PLP may not be unique generally.

Example 2 Using Definition 1 and 3, it is not difficult to conclude that $\mathcal{P}_1$ and $\mathcal{P}_3$ have unique reducts as follows respectively:

$$\mathcal{P}_1^< = \{P \leftarrow not\ Q,\ not\ R\},$$
$$\mathcal{P}_3^< = \{\neg Fly(x) \leftarrow Penguin(x),\ not\ Fly(x),\ Bird(Tweety) \leftarrow, Penguin(Tweety) \leftarrow\}.$$

■

Now it is quite straightforward to define the answer set for a prioritized extended logic program.

Definition 4 *Let $\mathcal{P} = (\Pi, \mathcal{N}, <)$ be a PLP and Lit the set of all ground literals in the language of $\mathcal{P}$. For any subset S of Lit, S is an* answer set *of $\mathcal{P}$, denoted as $Ans^P(\mathcal{P})$, iff $S = Ans(\mathcal{P}^<)$, where $Ans(\mathcal{P}^<)$ is an answer set of extended logic program $\mathcal{P}^<$. A ground literal L is* derivable *from a PLP $\mathcal{P}$, denoted as $\mathcal{P} \vdash L$, iff L belongs to every answer set of $\mathcal{P}$.*

Example 3 Immediately from Definition 4 and Example 1, we have the following solutions:

$$Ans^P(\mathcal{P}_1) = \{P\},$$
$$Ans^P(\mathcal{P}_3) = \{Bird(Tweety),\ Penguin(Tweety),\ \neg Fly(Tweety)\},$$

which, respectively, are also consistent with our intuitions. ■

3 UPDATING LOGIC PROGRAMS

3.1 The Approach

Now we are ready to propose a formal approach for logic program updates. We first introduce several useful notions. An extended logic program is *well defined* if it has an consistent answer set and each of its answer sets is also consistent. A rule with the form $L_0 \leftarrow L_1, \cdots, L_m,\ not\ L_{m+1}, \cdots, L_n$ is *satisfied* in a set of ground literals S if and only if the fact that $L_1, \cdots, L_m$ are in S and $L_{m+1}, \cdots, L_n$ are not in S implies the fact that L_0 in S. Given a consistent set $\mathcal{B}$ of ground literals, $calB$ is *coherent* with respect to an extended logic program Π is Π is well defined and there exists an answer set $Ans(\Pi)$ of Π such that $\mathcal{B} \subseteq Ans(\Pi)$.

¿From Example 1 discussed previously, we can see that there are two essential steps to perform an update on Π_0 with Π_1: (1) eliminating contradictory rules from Π_0 with respect to Π_1; and (2) solving conflicts between the remaining rules in Π_0 and Π_1. Our ideas to deal with these two issues are described as follows.

Eliminating contradictory rules

To eliminate contradictory rules from Π_0 with respect to Π_1, it cannot be simply to extract a maximal subset Π of Π_0 by requiring $\Pi \cup \Pi_1$ to be well defined. For instance, suppose $\Pi_0 = \{P \leftarrow, R \leftarrow Q\}$ and $\Pi_1 = \{Q \leftarrow P, \neg R \leftarrow Q\}$. Clearly, both $\Pi = \{P \leftarrow\}$ and $\Pi' = \{R \leftarrow P\}$ are maximal subsets of Π_0 such that $\Pi \cup \Pi_1$ and $\Pi' \cup \Pi_1$ are well defined. But intuitively, rule $R \leftarrow Q$ represents a contradictory semantics compared with rule $\neg R \leftarrow Q$ in Π_1, and hence we would like to delete $R \leftarrow Q$ instead of deleting $P \leftarrow$ from Π_0.

To achieve this purpose, we first *update* each answer set $Ans(\Pi_0)$ of Π_0 with Π_1. The result of this update is a set of ground literals, denoted as $\mathcal{B}_{(\Pi_0,\Pi_1)}$, which has minimal difference from $Ans(\Pi_0)$ and satisfies each rule in Π_1. If $\mathcal{B}_{(\Pi_0,\Pi_1)}$ is consistent, we then *extract* a maximal subset $\Pi_{(\Pi_0,\Pi_1)}$ of Π_0 such that $\mathcal{B}_{(\Pi_0,\Pi_1)}$ is coherent with $\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1$ (i.e. $\mathcal{B}_{(\Pi_0,\Pi_1)}$ is a subset of an answer set of $\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1$). Doing this, $\Pi_{(\Pi_0,\Pi_1)}$ is guaranteed to maximally retain rules of Π_0 which are not contradictory to rules of Π_1. The program $\Pi_{(\Pi_0,\Pi_1)}$ is called a *transformed program* from Π_0 with respect to Π_1. If $\mathcal{B}_{(\Pi_0,\Pi_1)}$ is *not* consistent, on the other hand, $\Pi_{(\Pi_0,\Pi_1)}$ is simply specified to be any maximal subset of Π_0 such that $\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1$ is well defined. The following example illustrates how this idea works.

Example 4 Consider extended logic programs $\Pi_0 = \{P \leftarrow, R \leftarrow Q, S \leftarrow not\ T\}$ and $\Pi_1 = \{Q \leftarrow P, \neg R \leftarrow Q, T \leftarrow not\ S\}$. Clearly, Π_0 has a unique answer set $Ans(\Pi_0) = \{P, S\}$. Updating $Ans(\Pi_0)$ with Π_1, intuitively, we would expect to have the result $\mathcal{B}_{(\Pi_0,\Pi_1)} = \{P, Q, \neg R, S\}$[3], which has a minimal difference from Π_0 and satisfies each rule in Π_1. Then it is easy to see that $\Pi_{(\Pi_0,\Pi_1)} = \{P \leftarrow, S \leftarrow not\ T\}$ is the unique maximal subset of Π_0 such that $\mathcal{B}_{(\Pi_0,\Pi_1)}$ is coherent with $\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1$. Therefore, $\Pi_{(\Pi_0,\Pi_1)}$ is the transformed program from Π_0 with respect to Π_1. ■

Solving conflicts

After transforming Π_0 to $\Pi_{(\Pi_0,\Pi_1)}$, we need to solve possible conflicts between rules in $\Pi_{(\Pi_0,\Pi_1)}$ and Π_1. To do so, we specify a prioritized logic program $\mathcal{P}_{(\Pi_0,\Pi_1)} = (\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1, \mathcal{N}, <)$, where for each rule r in Π_1 with name N and rule in $\Pi_{(\Pi_0,\Pi_1)}$ with name N', we specify $N < N'$. The intuitive idea behind this is that rules in Π_1 should be *more* preferred than rules in $\Pi_{(\Pi_0,\Pi_1)}$ as Π_1 expresses the agent's latest knowledge. Whenever there is a conflict between rules r and r' where $r \in \Pi_1$ and $r' \in \Pi_{(\Pi_0,\Pi_1)}$ respectively, r will override r'. Then, we finally specify the possible resulting program Π_0' after updating Π_0 with Π_1 to be a reduct of $\mathcal{P}_{(\Pi_0,\Pi_1)} = (\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1, \mathcal{N}, <)$, i.e. $\mathcal{P}^<_{(\Pi_0,\Pi_1)}$ (see Definition 3).

[3] We will formally describe this update procedure next.

Example 5 (Example 4 continued). Consider the same Π_0 and Π_1 as in Example 4. ¿From the previous step of eliminating contradictory rules in Π_0, Π_0 is transformed to $\Pi_{(\Pi_0,\Pi_1)} = \{P \leftarrow,\ S \leftarrow not\ T\}$. Now we specify $\mathcal{P}_{(\Pi_0,\Pi_1)} = (\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1, \mathcal{N}, <)$ as follows:

$N_1 : P \leftarrow,$
$N_2 : S \leftarrow not\ T,$
$N_3 : Q \leftarrow P,$
$N_4 : \neg R \leftarrow Q,$
$N_5 : T \leftarrow not\ S,$
$N_3 < N_1, N_3 < N_2, N_4 < N_1, N_4 < N_2,\ N_5 < N_1,$
$N_5 < N_2.$

¿From Definition 3, it is not difficult to see that $\Pi_{(\Pi_0,\Pi_1)}$ has a unique reduct $\{P \leftarrow, Q \leftarrow P, \neg R \leftarrow Q, T \leftarrow not\ S\}$, which, as we expect, is the resulting program Π_0' after updating Π_0 with Π_1. ■

3.2 Formal Descriptions

We first consider how to update an answer set of Π_0 with Π_1. Let $\mathcal{L}$ be the language of Π_0 and Π_1. We specify $\mathcal{L}_{new}^P$ to be a language of PLPs based on $\mathcal{L}$ with one more augment: For each predicate symbol P in $\mathcal{L}$, there is a corresponding predicate symbol $New\text{-}P$ in $\mathcal{L}_{new}^P$ with the same arity of P. To simplifying our presentation, in $\mathcal{L}_{new}^P$ we use notation $New\text{-}L$ to denote the corresponding literal L in $\mathcal{L}$. For instance, if a literal L in $\mathcal{L}$ is $\neg P(x)$, then notation $New\text{-}L$ simply means $\neg New\text{-}P(x)$. We use Lit_{new} to denote the set of all ground literals of $\mathcal{L}_{new}^P$. Clearly, $Lit_{new} = Lit \cup \{New\text{-}L \mid L \in Lit\}$.

Definition 5 *Let $Ans(\Pi_0)$ be an answer set of Π_0. The* specification of updating $Ans(\Pi_0)$ with Π_1 *is defined as a PLP of $\mathcal{L}_{new}$, denoted as $\mathcal{P}^* = (\Pi^*, \mathcal{N}, <)$, as follows:*

1. *Π^* consists of following rules:*
   Initial fact rules*: for each L in $Ans(\Pi_0)$, there is a rule $L \leftarrow$;*
   Inertia rules*: for each predicate symbol P in $\mathcal{L}$, there are two rules:*
   $New\text{-}P(x) \leftarrow P(x),\ not\ \neg New\text{-}P(x)$[4], and
   $\neg New\text{-}P(x) \leftarrow \neg P(x), not\ New\text{-}P(x)$,
   Update rules*: for each rule $L_0 \leftarrow L_1, \cdots, L_m, not\ L_{m+1}, \cdots, not\ L_n$ in Π_1, there is a rule $New\text{-}L_0 \leftarrow New\text{-}L_1, \cdots, New\text{-}L_m, not\ New\text{-}L_{m+1}, \cdots, not\ New\text{-}L_n$;*
2. *Naming function $\mathcal{N}$ assigns a unique name N for each rule r in Π^*;*
3. *For any inertia rule with name N and update rule with name N', there is a partial ordering between N and N': $N < N'$.*

Definition 6 *Let Π_0, Π_1, $Ans(\Pi_0), \mathcal{L}$ and $\mathcal{L}_{new}^P$ be specified as above, and $\mathcal{P}^*$ the specification of updating $Ans(\Pi_0)$ with Π_1 as defined in Definition 5. A set of ground literals of $\mathcal{L}$, $\mathcal{B}_{(\Pi_0,\Pi_1)}$, is called a* possible result *with respect to the update specification $\mathcal{P}^*$, iff for some answer set $Ans^P(\mathcal{P}^*)$ of $\mathcal{P}^*$, $\mathcal{B}_{(\Pi_0,\Pi_1)} = \{L \mid New\text{-}L \in Ans^P(\mathcal{P}^*)\}$.*

The update specification $\mathcal{P}^*$ defined in Definition 5 provides a formal method to derive the possible result of updating $Ans(\Pi_0)$ with Π_1. In Particular, inertia rules in Π^* guarantee a minimal change during the update, i.e. any initial fact in $Ans(\Pi_0)$ that is not explicitly changed will persist by default. Update rules, on the other hand, specify effects of the update. Since both inertia and update rules may be defeasible[5], possible conflicts may occur between them. Furthermore, from the minimal change principle, initial facts in $Ans(\Pi_0)$ are always preferred to persist during the update whenever there is no explicit violation of update rules. So we specify that inertia rules are more preferred than update rules. Finally, the possible result of this update is derived from answer sets of $\mathcal{P}^*$ as presented in Definition 6.

It is clear that given an answer set $Ans(\Pi_0)$ of Π_0, updating $Ans(\Pi_0)$ with Π_1 may have more than one possible results. Let $\mathcal{B}$ and $\mathcal{B}'$ be two sets of ground literals. We define $Diff(\mathcal{B}, \mathcal{B}') = \{|L| \mid L \in (\mathcal{B} - \mathcal{B}') \cup (\mathcal{B}' - \mathcal{B})\}$, where $|L|$ indicates the corresponding ground atom of ground literal L. Then the following theorem guarantees that for each $Ans(\Pi_0)$ and the corresponding $\mathcal{B}_{(\Pi_0,\Pi_1)}$, changes between $Ans(\Pi_0)$ and $\mathcal{B}_{(\Pi_0,\Pi_1)}$ are minimal with respect to set inclusion.

Theorem 1 *Let Π_0, Π_1, $Ans(\Pi_0)$ and $\mathcal{B}_{(\Pi_0,\Pi_1)}$ be specified as above. Then there does not exist another $\mathcal{B}'$ such that every rule of Π_1 is satisfied in $\mathcal{B}'$ and $Diff(Ans(\Pi_0), \mathcal{B}') \subset Diff(Ans(\Pi_0), \mathcal{B}_{(\Pi_0,\Pi_1)})$.*

Following previous descriptions, the next two definitions then specify the transformed program from Π_0, which eliminates all contradictory rules from Π_0 with respect to Π_1, and the final resulting program after updating Π_0 with Π_1 respectively.

Definition 7 *Given a $\mathcal{B}_{(\Pi_0,\Pi_1)}$ as specified in Definition 6. An extended logic program $\Pi_{(\Pi_0,\Pi_1)}$ is called a* transformed program *from Π_0 with respec to Π_1, iff (1) if $\mathcal{B}_{(\Pi_0,\Pi_1)}$ is consistent, then $\Pi_{(\Pi_0,\Pi_1)}$ is a maximal subset of Π_0 such that $\mathcal{B}_{(\Pi_0,\Pi_1)}$ is coherent with program $\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1$; (2) if $\mathcal{B}_{(\Pi_0,\Pi_1)}$ is inconsistent, then $\Pi_{(\Pi_0,\Pi_1)}$ is any maximal subset of Π_0 such that $\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1$ is well defined.*

Definition 8 *Let $\Pi_{(\Pi_0,\Pi_1)}$ be defined as in Definition 7. We specify a PLP $\mathcal{P}_{(\Pi_0,\Pi_1)} = (\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1, \mathcal{N}, <)$, where naming function $\mathcal{N}$ assigns a unique name for each rule in $\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1$, and for each rule r in Π_1 with name N and each rule r' in $\Pi_{(\Pi_0,\Pi_1)}$ with name N', there is a preference relation $N < N'$. Then an extended logic program Π_0' is called a* possible resulting program *after updating Π_0 with Π_1 iff Π_0' is a reduct of $\mathcal{P}_{(\Pi_0,\Pi_1)}$.*

Example 6 Given two extended logic programs $\Pi_0 = \{P \leftarrow, R \leftarrow not\ Q, Q \leftarrow not\ R\}$ and $\Pi_1 = \{\neg P \leftarrow\}$. Consider an update of Π_0 with Π_1. Firstly, to find out the contradictory rule in Π_0 with respect to Π_1, we update every answer set of Π_0 with Π_1. Clearly, Π_0 has two answer sets $\{P, R\}$ and $\{P, Q\}$. Updating these two answer sets with Π_1, we get $\{\neg P, R\}$ and $\{\neg P, Q\}$ respectively. ¿From Definition 7, it is not difficult to conclude that program $\Pi_{(\Pi_0,\Pi_1)} = \{R \leftarrow not\ Q, Q \leftarrow not\ R\}$ is the unique transformed program from Π_0 with respect to Π_1 (i.e. both $\{P, R\}$ and $\{P, Q\}$ are coherent with $\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1$). Secondly, to solve possible conflicts between rules in $\Pi_{(\Pi_0,\Pi_1)}$ and Π_1, we specify a PLP $\mathcal{P}_{(\Pi_0,\Pi_1)} = (\Pi_{(\Pi_0,\Pi_1)} \cup \Pi_1, \mathcal{N}, <)$ as follows:

[4] x might be a tuple of variables.

[5] Weak negation *not* may be included in these rules.

$N_1 : R \leftarrow not\ Q,$
$N_2 : Q \leftarrow not\ R,$
$N_3 : \neg P \leftarrow,$
$N_3 < N_1, N_3 < N_2.$

Finally, it is concluded from Definition 3 that $\mathcal{P}_{(\Pi_0,\Pi_1)}$ has two reducts $\{\neg P \leftarrow, R \leftarrow not\ Q\}$ and $\{\neg P \leftarrow, Q \leftarrow not\ R\}$, which, as specified in Definition 8, are the two possible resulting programs after updating Π_0 with Π_1. ■

4 SEMANTICAL CHARACTERIZATIONS

In this section, we investigate semantical properties of logic program updates. As it has been mentioned earlier, when we consider an update of Π_0 with Π_1, we assume that both Π_0 and Π_1 are well defined extended logic programs. So the first property we need to guarantee is that the possible resulting program Π'_0 should be also well defined after updating Π_0 with Π_1.

Theorem 2 *Let Π_0 and Π_1 be extended logic programs, and Π'_0 be a possible resulting program after updating Π_0 with Π_1 as specified in Definition 8. Then Π'_0 is well defined.*

Another desirable property for update is so-called *effect preservation*. That means, during an update of Π_0 with Π_1, if a ground literal L is derivable from the update effect Π_1, then L would be also expected to be derivable from the resulting program Π'_0. However, due to the defeasibility of Π_1, this is *not* always true. For instance, let $\Pi_0 = \{P \leftarrow\}$ and $\Pi_1 = \{Q \leftarrow not\ P\}$. Clearly, $\Pi_1 \vdash Q$. But after updating Π_0 with Π_1, the resulting program is $\Pi'_0 = \{P \leftarrow, Q \leftarrow not\ P\}$, from which $\Pi'_0 \not\vdash Q$.

Nevertheless, under some conditions, the effect preservation property may still hold. For a given extended logic program Π, we specify $\mathcal{B}_\Pi = \{L \mid \Pi \vdash L\}$. We also denote the set of all possible resulting programs after updating Π_0 with Π_1 as $Update(\Pi_0, \Pi_1)$. Then we have the following result.

Theorem 3 *Let Π_0, Π_1 be extended logic program and $\Pi'_0 \in Update(\Pi_0, \Pi_1)$, where each rule in Π_0 has the form $L_0 \leftarrow not\ L_1, \cdots, not\ L_n$; or each rule in Π_1 has the form $L_0 \leftarrow L_1, \cdots, L_n$. Then $\mathcal{B}_{\Pi_1} \subseteq \mathcal{B}_{\Pi'_0}$.*

Generally, our logic program update is *nonmonotonic* in the sense that for given programs Π_0, Π_1, Π_2 and Π_3 satisfying $\mathcal{B}_{\Pi_0} \subseteq \mathcal{B}_{\Pi_2}$ and $\mathcal{B}_{\Pi_1} \subseteq \mathcal{B}_{\Pi_3}$, it is not necessarily to have the fact that for any program $\Pi'_0 \in Update(\Pi_0, \Pi_1)$, there exists some $\Pi'_2 \in Update(\Pi_2, \Pi_3)$ such that $\mathcal{B}_{\Pi'_0} \subseteq \mathcal{B}_{\Pi'_2}$. B But our logic program update satisfies a restricted monotonicity as the following theorem shows.

Theorem 4 *Let Π_0, Π_1, Π_2 be extended logic programs satisfying the following conditions: (1) $\mathcal{B}_{\Pi_0} \subseteq \mathcal{B}_{\Pi_2}$, (2) each rule in Π_0 and Π_2 has the form $L_0 \leftarrow L_1, \cdots, L_n$, and (3) each rule in Π_1 has the form $L_0 \leftarrow not\ L_1, \cdots, not\ L_n$. Then for any program $\Pi'_0 \in Update(\Pi_0, \Pi_1)$, there exists some $\Pi'_2 \in Update(\Pi_2, \Pi_1)$ such that $\mathcal{B}_{\Pi'_0} \subseteq \mathcal{B}_{\Pi'_2}$.*

Finally, we can prove that during an update of Π_0 with Π_1, the change from Π_0 to the resulting program Π'_0 is minimal under the condition of eliminating contradictory rules and solving conflicts.

Theorem 5 *Let Π_0, Π_1 be extended logic programs, and $\Pi'_0 \in Update(\Pi_0, \Pi_1)$. Then there does not exist another extended logic program Π, where $\Pi_1 \subseteq \Pi \subseteq \Pi_0 \cup \Pi_1$, such that $\mathcal{B}_{\Pi'_0} \subseteq \mathcal{B}_\Pi$ and $\Pi'_0 \subset \Pi$.*

5 CONCLUDING REMARKS

In this paper, we considered a new update problem - logic program update, which is important in applications associated with dynamic domains (e.g. domains that may be evolving). Clearly, our logic program update is different from those update formulations developed within logic program frameworks previously (e.g. [3, 2]), in which an update was simply considered as an insertion/deletion of an atom to/from a logic program.

The logic program update is also different from the rule-based update first proposed by Marek and Truszczyński and generalized by the authors recently [5]. Basically, the rule-based update addresses the problem of how to update a knowledge base (i.e. a set of ground literals) in terms of a logic program such that the resulting knowledge base satisfies each rule of the logic program. However, we observed that the authors' generalized rule-based update [5] is actually a special case of our current logic program update. For instance, given a knowledge base KB which is a finite set of ground literals, updating KB with an extended logic program Π_1 can be viewed as an update on Π_0 with Π_1, where $\Pi_0 = \{L \leftarrow \mid L \in KB\}$.

Finally, we can see that the computation of logic program update is based on computations of the answer set and reduct of corresponding prioritized logic programs. This issue has been investigated in detail in authors' another paper [7].

REFERENCES

[1] M. Gelfond and V. Lifschitz, Classical negation in logic programs and disjunctive databases. *New Generation Computing*, **9** (1991) 365-386.

[2] A. Guessoum and J.W. Lloyd, Updating knowledge bases II. *New Generation Computing*, **10** (1991) 73-100.

[3] A.C. Kakas and P. Mancarella, Database updates through abduction. In *Proceedings of the 16th VLDB Conference*, pp 650-661, 1990.

[4] L.Y. Yuan and J.H. You, Coherence approach to logic program revision. *IEEE Transaction on Knowledge and Data Engineering* (to appear).

[5] Y. Zhang and N.Y. Foo, Towards generalized rule-based updates. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pp 82-88. Morgan Kaufmann Publishers Inc., 1997.

[6] Y. Zhang and N.Y. Foo, Answer sets for prioritized logic programs. In *Proceedings of the 1997 International Logic Programming Symposium (ILPS'97)*, pp 69-83. MIT Press, 1997.

[7] Y. Zhang and N.Y. Foo, Prioritized logic programming: Semantics, computations and applications. Manuscript, August 1998.