Solving Logic Program Conflict through Strong and Weak Forgettings

Yan Zhang
School of Computing & IT
University of Western Sydney
NSW 1797, Australia
E-mail: yan@cit.uws.edu.au

Norman Foo
School of Comp. Sci. & Eng.
University of New South Wales
NSW 2052, Australia
E-mail: norman@cse.unsw.edu.au

Kewen Wang
School of Computing & IT
Griffith University
QLD 4111, Australia
E-mail: k.wang@cit.gu.edu.au

Abstract

We consider how to forget a set of atoms in a logic program. Intuitively, when a set of atoms is forgotten from a logic program, all atoms in the set should be eliminated from this program in some way, and other atoms related to them in the program might also be affected. We define notions of strong and weak forgettings in logic programs to capture such intuition and reveal their close connections to the notion of forgetting in classical propositional theories. Based on these notions, we then propose a framework for conflict solving in logic programs, which is general enough to represent many important conflict solving problems. We also study some essential semantic and computational properties in relation to strong and weak forgettings and conflict solving in our framework.

1 Introduction

One promising approach in the research of reasoning about knowledge dynamics is to represent agents' knowledge bases as logic programs on which necessary updates are conducted as a way to model agents' knowledge evolution. A key issue in this study is to solve various conflicts and inconsistencies in logic programs, e.g. [Leite, 2003].

While different logic program update approaches have been developed recently, we observe that some typical conflict solving problems in logic programs have yet to be thoroughly investigated in the literature. Let us consider a scenario. John wants Sue to help him to complete his assignment. He knows that Sue will help him if she is not so busy. Tom is a good friend of John and wants John to let him copy John's assignment. Then John also learns that Sue hates Tom, and will not help him if he lets Tom copy his assignment, which will be completed under Sue's help. While John does not care whether Sue hates Tom or not, he has to consider Sue's condition to offer him help. What is John going to do? We formalize this scenario in a logic programming setting. John's knowledge base Π_J:

$r_1 : Complete(John, Assignment) \leftarrow Help(Sue, John)$,
$r_2 : Help(Sue, John) \leftarrow \texttt{not}\, Busy(Sue)$,
$r_3 : Goodfriend(John, Tom) \leftarrow$,
$r_4 : Copy(Tom, Assignment) \leftarrow Goodfriend(John, Tom), Complete(John, Assignment)$,

and Sue's knowledge base Π_S:

r_5: $Hate(Sue, Tom) \leftarrow$,
r_6: $\leftarrow Help(Sue, John), Copy(Tom, Assignment)$.

In order to take Sue's knowledge base into account, suppose John updates his knowledge base Π_J in terms of Sue's Π_S. By applying proper logic program update approach, John may obtain a solution: $\Pi_J^{final} = \{r_1, r_2, r_3, r_5, r_6\}$ or its stable model, from which we know that Sue will help $John$ to complete the assignment and $John$ will not let Tom copy his assignment. Although the conflict between Π_J and Π_S has been solved by updating, the result is somehow not always satisfactory. For instance, while John wants Sue to help him, he may have no interest at all in integrating the information that Sue hates Tom into his new knowledge base.

As an alternative, John may just weaken his knowledge base by *forgetting* atom $Copy(Tom, Assignment)$ from Π_J in order to accommodate Sue's constraint. Then John may have a new program $\Pi_J^{final} = \{r_1, r_2, r_3\}$ - John remains a maximal knowledge subset which is consistent with Sue's condition without being involved in Sue's personal feeling about Tom.

The formal notion of forgetting in propositional theories was initially considered by Lin and Reiter from a cognitive robotics perspective [Lin and Reiter, 1994] and has recently received a great attention in KR community. It has been shown that the theory of forgetting has important applications in solving knowledge base inconsistencies, belief update and merging, abductive reasoning, causal theories of actions, and reasoning about knowledge under various propositional (modal) logic frameworks, e.g. [Lang and Marquis, 2002; Lang *et al.*, 2003; Lin, 2001; Su *et al.*, 2004].

In this paper, we consider how to forget a set of atoms from a propositional normal logic program and how this idea can be used in general conflict solving under the context of logic programs. The rest of this paper is organized as follows. We present preliminary definitions and concepts in section 2. In section 3, we give formal definitions of strong and weak forgettings in logic programs. Based on these notions, in section 4, we propose a framework called logic program contexts for

general conflict solving in logic programs. In section 5, we investigate related semantic and computational properties. In section 6 we conclude the paper with some discussions.

2 Preliminaries

We consider finite propositional normal logic programs in which each rule has the form:

$$a \leftarrow b_1, \cdots, b_m, \texttt{not}\, c_1, \cdots, \texttt{not}\, c_n \quad (1)$$

where a is either a propositional atom or empty, and $b_1, \cdots, b_m$, $c_1, \cdots, c_n$ are propositional atoms. When a is empty, rule (1) is called a *constraint*. Given a rule r of the form (1), we denote $head(r) = \{a\}$, $pos(r) = \{b_1, \cdots, b_m\}$ and $neg(r) = \{c_1, \cdots, c_n\}$, and therefore, rule (1) may be represented as the form:

$$head(r) \leftarrow pos(r), \texttt{not}\, neg(r). \quad (2)$$

We also use $Atom(r)$ to denote the set of all atoms occurring in rule r. For a program Π, we define notions $head(\Pi) = \bigcup_{r\in\Pi} head(r)$, $pos(\Pi) = \bigcup_{r\in\Pi} pos(r)$, $neg(\Pi) = \bigcup_{r\in\Pi} neg(r)$, $Atom(\Pi)$ the set of all propositional atoms occurring in program Π. Given sets of atoms P and Q, we may use notion

$$r' : head(r) \leftarrow (pos(r) - P), \texttt{not}\, (neg(r) - Q)$$

to denote rule r' obtained from r by removing all atoms occurring in P and Q in the positive and negation as failure parts respectively.

The stable model of a program Π is defined as follows. Firstly, we consider Π to be a program in which each rule does not contain negation as failure sign not. A finite set S of propositional atoms is called a *stable model* of Π if S is the smallest set such that for each rule $a \leftarrow b_1, \cdots, b_m$ from Π, if $b_1, \cdots, b_m \in S$, then $a \in S$. Now let Π be an arbitrary normal logic program. For any set S of atoms, program Π^S is obtained from Π by deleting (1) each rule from Π that contains not c in the body and $c \in S$; and (2) all forms of not c in the bodies of the remaining rules. Then S is a stable model of Π if and only if S is a stable model of Π^S. A program may have one, more than one, or no stable models at all. A program is called *consistent* if it has a stable model. We say that an atom a *is entailed* from program Π, denoted as $\Pi \models a$ if a is in every stable model of Π.

Given two programs Π_1 and Π_2. Π_1 and Π_2 are *equivalent* if Π_1 and Π_2 have the same stable models. Π_1 and Π_2 are called *strongly equivalent* if for every program Π, $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ are equivalent [Lifschitz *et al.*, 2001].

Observation: Let Π be a logic program. If each rule r in Π is of one of the following two forms: (1) $head(r) \neq \emptyset$ and $head(r) \subseteq pos(r)$, or (2) $pos(r) \cap neg(r) \neq \emptyset$, then Π is strongly equivalent to the empty set.

For a later reference, we call the two types of rules mentioned above *redundant rules*.

Let Π be a logic program. We use $[\Pi]^C$ to denote the conjunctive normal form obtained from Π by translating each rule of the form (1) in Π in to the clause: $a \vee \neg b_1 \vee \cdots \vee \neg b_m \vee c_1 \vee \cdots \vee c_n$. For instance, if $\Pi = \{a \leftarrow \texttt{not}\, b, c \leftarrow a\}$, then we have $[\Pi]^C = (a \vee b) \wedge (c \vee \neg a)$. In general, we may write $[\Pi]^C = \{C_1, \cdots, C_n\}$ where each C_i is a conjunct of $[\Pi]^C$. If C_i is a clause, we call any sub formula of C_i a *subclause* of C_i.

Now we introduce the notion of forgetting in a classical propositional theory [Lin and Reiter, 1994; Lin, 2001]. Let T be propositional theory. We use $T(p/true)$ (or $T(p/false)$, resp.) to denote the theory obtained from T by substituting all occurrences of propositional atom p with *true* (or *false*, resp.). For instance, if $T = \{p \supset q, (q \wedge r) \supset s\}$, then $T(q/true) = \{r \supset s\}$ and $T(q/false) = \{\neg p\}$. Then we can define the notion of forgetting in terms of a propositional theory. For a given propositional theory T and a set of propositional atoms P, the result of *forgetting* P in T, denoted as $Forget(T, P)$, is defined inductively as follows:

$Forget(T, \emptyset) = T$,
$Forget(T, \{p\}) = T(p/true) \vee T(p/false)$,
$Forget(T, P \cup \{p\}) = Forget(Forget(T, \{p\}), P)$.

It is easy to see that the ordering in which atoms in P are considered does not affect the final result of forgetting P from T. Consider $T = \{p \supset q, (q \wedge r) \supset s\}$ again. From the above definition, we have $Forget(T, \{q\}) = \{(r \supset s) \vee \neg p\}$.

3 Strong and Weak Forgettings in Logic Programs

Let us consider how to forget a set of atoms from a logic program. Intuitively, we would expect that after forgetting a set of atoms, all occurrences of these atoms in the underlying program should be eliminated in some way, and moreover, other atoms having connections to them through rules in the program might also be affected. We observe that the notion of forgetting in propositional theories is not applicable to logic programs since there is no disjunctive operation for logic programs. Further, different ways of handling negation as failure in forgetting may also lead to different resulting programs. To formalize our idea of forgetting in logic programs, we first introduce a program transformation called *reduction*.

Definition 1 *(**Program reduction**) Let Π be a program and p an atom. We define the* reduction of Π with respect to p, *denoted as $Reduct(\Pi, \{p\})$, to be a program obtained from Π by (1) for each rule r with $head(r) = p$ and each rule r' with $p \in pos(r')$, replacing r' with a new rule $r'' : head(r') \leftarrow (pos(r) \cup pos(r') - \{p\}), \texttt{not}(neg(r) \cup neg(r'))$; (2) if there is such rule r' in Π and has been replaced by r'' in (1), then removing rule r from the remaining program. Let P be a set of atoms. Then the* reduction of Π with respect to P *is inductively defined as follows:*

$Reduct(\Pi, \emptyset) = \Pi$,
$Reduct(\Pi, P \cup \{p\}) = Reduct(Reduct(\Pi, \{p\}), P)$.

Note that in our program reduction definition, step (1) is the same as logic program *unfolding* [Brass and Dix, 1999]. While unfolding is to eliminate positive body occurrences of an atom in a logic program, the reduction, on other hand, is further to remove those rules with heads of this atom.

Example 1 Let $\Pi_1 = \{a \leftarrow \texttt{not}\, b, a \leftarrow d, c \leftarrow a, \texttt{not}\, e\}$, $\Pi_2 = \{a \leftarrow c, \texttt{not}\, b, c \leftarrow \texttt{not}\, d\}$, and $\Pi_3 = \{a \leftarrow b, b \leftarrow \texttt{not}\, d, c \leftarrow a, \texttt{not}\, e\}$. Then $Reduct(\Pi_1, \{a\}) = \{c \leftarrow \texttt{not}\, b, \texttt{not}\, e, c \leftarrow d, \texttt{not}\, e\}$, $Reduct(\Pi_2, \{a\}) = \Pi_2$, and $Reduct(\Pi_3, \{a, b\}) = \{c \leftarrow \texttt{not}\, d, \texttt{not}\, e\}$. □

Definition 2 *(**Strong forgetting**) Let Π be a logic program, and p an atom. We define a program to be the result of* strongly forgetting *p in Π, denoted as $SForgetLP(\Pi, \{p\})$, if it is obtained from the following transformation:*

1. $\Pi' = Reduct(\Pi, \{p\})$;
2. $\Pi' = \Pi' - \{r \mid r \text{ is a redundant rule}\}$;
3. $\Pi' = \Pi' - \{r \mid head(r) = p\}$;
4. $\Pi' = \Pi' - \{r \mid p \in pos(r)\}$;
5. $\Pi' = \Pi' - \{r \mid p \in neg(r)\}$;
6. $SForgetLP(\Pi, \{p\}) = \Pi'$.

Let us take a closer look at Definition 2. Step 1 is just to perform reduction on Π with respect to atom p. This is to replace those *positive body occurrences* of p in rules with other rules having p as the head. Step 2 is to remove all redundant rules which may be introduced by the reduction of Π with respect to p. From Proposition **??**, we know that this does not change anything in the program. Steps 3 and 4 are to remove those rules which have p as the head or in the positive body. Note that after Steps 1 and 2, there does not exist any pair of rules r and r' such that $head(r) = \{p\}$ and $p \in pos(r')$. Then the intuitive meaning of Steps 3 and 4 is that after forgetting p, any atom's information in rules having p as their heads or positive bodies will be lost because they are all relevant to p, i.e. these atoms either serve as a support for p or p is in part of the supports for these atoms. On the other hand, Step 5 states that any rule containing p in its negation as failure part will be also removed. The consideration for this step is as follows. If we think $neg(r)$ is a part of support of $head(r)$, then when $p \in neg(r)$ is forgotten, $head(r)$'s entire support is lost as well. Clearly, such treatment of negation as failure in forgetting is quite strong in the sense that more atoms may be lost together with $\texttt{not}\, p$. Therefore we call this kind of forgetting *strong forgetting*.

With a different way of dealing with negation as failure, we have a weak version of forgetting. We define a program to be the result of *weakly forgetting* p in Π, denoted as $WForgetLP(\Pi, \{p\})$, exactly in the same way as in Definition 2 except that Step 5 is replaced by the following step:

$\Pi' = (\Pi' - \Pi^*) \cup \Pi^\dagger$, where
$\Pi^* = \{r \mid p \in neg(r)\}$ and $\Pi^\dagger = \{r' \mid r' : head(r) \leftarrow pos(r), \texttt{not}(neg(r) - \{p\})$ where $r \in \Pi^*\}$.

Suppose we have a rule like $r : head(r) \leftarrow pos(r), \texttt{not}\, neg(r)$ where $p \in neg(r)$. Instead of viewing $neg(r)$ as part of the support of $head(r)$, we may treat it as a default evidence of $head(r)$, i.e. under the condition of $pos(r)$, if all atoms in $neg(r)$ are not presented, then $head(r)$ can be derived. Therefore, forgetting p will result in the absence of p in any case. So r may be replaced by $r' : head(r) \leftarrow pos(r), \texttt{not}\, (neg(r) - \{p\})$.

Strong and weak forgettings can be easily extended to the case of a set of atoms:

$SForgetLP(\Pi, \emptyset) = \Pi$,
$SForgetLP(\Pi, P \cup \{p\}) = SForgetLP(SForgetLP(\Pi, \{p\}), P)$,

and $WForgetLP(\Pi, P)$ is defined accordingly. The following proposition ensures that our strong and weak forgettings in logic programs are well defined under strong equivalence.

Proposition 1 *Let Π be a logic program and p, q two propositional atoms. Then*

1. *$SForgetLP(SForgetLP(\Pi, \{p\}), \{q\})$ is strongly equivalent to $SForgetLP(SForgetLP(\Pi, \{q\}), \{p\})$;*
2. *$WForgetLP(WForgetLP(\Pi, \{p\}), \{q\})$ is strongly equivalent to $WForgetLP(WForgetLP(\Pi, \{q\}), \{p\})$.*

Example 2 Let $\Pi = \{b \leftarrow a, c,\ d \leftarrow \texttt{not}\, a,\ e \leftarrow \texttt{not}\, f\}$. Then we have $SForgetLP(\Pi, \{a\}) = \{e \leftarrow \texttt{not}\, f\}$, and $WForgetLP(\Pi, \{a\}) = \{d \leftarrow, e \leftarrow \texttt{not}\, f\}$. Now we consider $Forget([\Pi]^C, \{a\})$, which is logically equivalent to formula $(b \vee \neg c \vee d) \wedge (f \vee e)$. Then it is clear that $\models Forget([\Pi]^C, \{a\}) \supset [SForgetLP(\Pi, \{a\})]^C$, and $\models [WForgetLP(\Pi, \{a\})]^C \supset Forget([\Pi]^C, \{a\})$. $\square$

The above example motivates us to examine the deeper relationships between strong and weak forgettings in logic programs and forgetting in propositional theories. Let Π be a program and L a clause, i.e. $L = l_1 \vee \cdots \vee l_k$ where each l_i is a propositional literal. We say that L is Π-*coherent* if there exists a subset Π' of Π and a set of atoms $P \subseteq Atom(\Pi)$ (P could be empty) such that L is a subclause of $[Reduct(\Pi', P)]^C$ (i.e. $[Reduct(\Pi', P)]^C$ is a single clause). The intuition behind this notion is to specify those clauses that are parts of clauses generated from program Π through reduction. Consider program $\Pi = \{a \leftarrow b, d \leftarrow a, \texttt{not}\, c, e \leftarrow \texttt{not}\, d\}$. Clause $d \vee b$ is Π-coherent, where clause $\neg d \vee e$ is not. Obviously, for each rule $r \in \Pi$, $[\{r\}]^C$ is Π-coherent. The following proposition provides a semantic account for Π-coherent clauses.

Proposition 2 *Let Π be a program and L a Π-coherent clause. Then either $\models [\Pi]^C \supset L$ or $\models L \supset \Phi$ for some clause Φ where $\models [\Pi]^C \supset \Phi$.*

Definition 3 *Let Π be a logic program, φ, φ_1 and φ_2 three propositional formulas where φ_1 and φ_2 are in conjunctive normal forms.*

1. *φ_1 is called a* consequence *of φ* with respect to Π *if $\models \varphi \supset \varphi_1$ and each conjunct of φ_1 is Π-coherent. φ_1 is a* strongest consequence *of φ* with respect to Π *if φ_1 is a consequence of φ with respect to Π and there does not exist another consequence φ_1' of φ ($\varphi_1' \not\equiv \varphi_1$) with respect to Π such that $\models \varphi_1' \supset \varphi_1$.*
2. *φ_2 is called a* premiss *of φ* with respect to Π *if $\models \varphi_2 \supset \varphi$ and each conjunct of φ_2 is Π-coherent. φ_2 is a* weakest premiss *of φ* with respect to Π *if φ_2 is a premiss of φ with respect to Π and there does not exist another premiss φ_2' of φ ($\varphi_2' \not\equiv \varphi_2$) with respect to Π such that $\models \varphi_2 \supset \varphi_2'$.*

Example 3 (Example 2 continued) It is easy to verify that $[SForgetLP(\Pi, \{a\})]^C$ is a strongest consequence of $Forget([\Pi]^C, \{a\})$ and $[WForgetLP(\Pi, \{a\})]^C$ is a weakest premiss of $Forget([\Pi]^C, \{a\})$. In fact, the following theorem confirms that this is always true. $\square$

Theorem 1 *Let Π be a logic program and P a set of atoms. Then $[SForgetLP(\Pi, P)]^C$ is a strongest consequence of $Forget([\Pi]^C, P)$ with respect to Π and $[WForgetLP(\Pi, P)]^C$ is a weakest premiss of $Forget([\Pi]^C, P)$ with respect to Π.*

Theorem 1 actually provides a precise semantic characterization for strong and weak forgettings in logic programs in terms of the forgetting notion in the corresponding propositional theory.

4 Solving Conflicts in Logic Program Contexts

In this section, we define a general framework called logic program context to represent a knowledge system which consists of multiple agents' knowledge bases. We consider the issue of conflicts occurring in the reasoning within the underlying logic program context. As we will show, the notions of strong and weak forgettings provide an effective way to solve such conflicts.

Definition 4 *(**Logic program context**) A* logic program context *is a n-ary tuple $\Sigma = (\Phi_1, \cdots, \Phi_n)$, where each Φ_i is a triplet $(\Pi_i, \mathcal{C}_i, \mathcal{F}_i)$ - Π_i and $\mathcal{C}_i$ are two logic programs, and $\mathcal{F}_i \subseteq Atom(\Pi_i)$ is a set of atoms. We also call each Φ_i the ith* component *of Σ. Σ is* consistent *if for each i, $\Pi_i \cup \mathcal{C}_i$ is consistent. Σ is* conflict-free *if for any i and j, $\Pi_i \cup \mathcal{C}_j$ is consistent.*

In the above definition, for a given logic program context Σ, each component Φ_i represents agent i's local situation, where Π_i is agent i's knowledge base, $\mathcal{C}_i$ is a set of constraints that agent i should comply and will not change in any case, and $\mathcal{F}_i$ is a set of atoms that agent i may forget if necessary. To simplify our following discussion, we assume that for each component Φ_i, the corresponding agent's knowledge base Π_i does not contain constraints (i.e. rules with empty heads). Alternatively such constraints will be contained in the constraint set $\mathcal{C}_i$ though $\mathcal{C}_i$ may also contain rules with nonempty heads.

Now the problem of conflict solving under this setting can be stated as follows: given a logic program context $\Sigma = (\Phi_1, \cdots, \Phi_n)$, which may not be consistent or conflict-free, how can we find an alternative logic program context $\Sigma' = (\Phi'_1, \cdots, \Phi'_n)$ such that Σ' is conflict-free *and* is *closest* to the original Σ in some sense?

Definition 5 *(**Solution**) Let $\Sigma = (\Phi_1, \cdots, \Phi_n)$ be a logic program context, where each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i)$. We call a logic program context Σ' a* solution *that solves conflicts in Σ, if Σ' satisfies the following conditions:*

1. *Σ' is conflict-free;*
2. *$\Sigma' = (\Phi'_1, \cdots, \Phi'_n)$, where $\Phi'_i = (\Pi'_i, \mathcal{C}_i, \mathcal{F}_i)$, and $\Pi'_i = SForgetLP(\Pi_i, P_i)$ or $\Pi'_i = WForgetLP(\Pi_i, P_i)$ for some $P_i \subseteq \mathcal{F}_i$.*

We denote the set of all solutions of Σ as $Solution(\Sigma)$.

Definition 6 *(**Ordering on solutions**) Let Σ, Σ' and Σ'' be three logic program contexts, where $\Sigma', \Sigma'' \in Solution(\Sigma)$. We say that Σ' is* closer *or* as close to Σ as Σ''*, denoted as $\Sigma' \preceq_\Sigma \Sigma''$, if for each i, $\Phi'_i = (\Pi'_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma'$ and $\Phi''_i = (\Pi''_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma''$, where $\Pi'_i = SForgetLP(\Pi_i, P_i)$ or $\Pi'_i = WForgetLP(\Pi_i, P_i)$ for some $P_i \subseteq \mathcal{F}_i$, and $\Pi''_i = SForgetLP(\Pi_i, Q_i)$ or $\Pi''_i = WForgetLP(\Pi_i, Q_i)$ for some $Q_i \subseteq \mathcal{F}_i$ respectively, we have $P_i \subseteq Q_i \subseteq \mathcal{F}_i$. We denote $\Sigma' \prec_\Sigma \Sigma''$ if $\Sigma' \preceq_\Sigma \Sigma''$ and $\Sigma'' \not\preceq_\Sigma \Sigma'$.*

Definition 7 *(**Preferred solution**) Let Σ and Σ' be two logic program contexts. We say that Σ' is a* preferred solution *of Σ, if $\Sigma' \in Solution(\Sigma)$ and there does not exist another $\Sigma'' \in Solution(\Sigma)$ such that $\Sigma'' \prec_\Sigma \Sigma'$.*

Example 4 Let $\Sigma = (\Phi_1, \Phi_2)$, where

Φ_1:
Π_1: $a \leftarrow$,
$b \leftarrow a, \mathtt{not} c$,
$d \leftarrow a, \mathtt{not} e$,
$f \leftarrow d$,
$\mathcal{C}_1$: $\leftarrow d, \mathtt{not} f$,
$\leftarrow \mathtt{not} d, \mathtt{not} f$,
$\mathcal{F}_1$: $\{a, b, c\}$,

Φ_2:
Π_2: $c \leftarrow$,
$d \leftarrow \mathtt{not} e$,
$e \leftarrow c$,
$f \leftarrow d$,
$\mathcal{C}_2$: $\leftarrow b, \mathtt{not} c$,
$b \leftarrow c$,
$\mathcal{F}_2$: $\{a, b, c, d, e, f\}$.

It is easy to see that Σ is consistent but not conflict-free because neither $\Pi_1 \cup \mathcal{C}_2$ nor $\Pi_2 \cup \mathcal{C}_1$ is consistent. Now consider two logic program contexts $\Sigma_1 = (\Phi'_1, \Phi'_2)$ and $\Sigma_2 = (\Phi''_1, \Phi''_2)$, where

$\Phi'_1 = (SForgetLP(\Pi_1, \{c\}), \mathcal{C}_1, \mathcal{F}_1)$,
$\Phi'_2 = (WForgetLP(\Phi_2, \{e\}, \mathcal{C}_2, \mathcal{F}_2)$, and
$\Phi''_1 = (WForgetLP(\Pi_1, \{a, c\}), \mathcal{C}_1, \mathcal{F}_1)$,
$\Phi''_2 = (WForgetLP(\Phi_2, \{e\}, \mathcal{C}_2, \mathcal{F}_2)$.

It can be verified that both Σ_1 and Σ_2 are solutions of Σ, but only Σ_1 is a preferred solution. □

5 Semantic and Computational Properties

In this section, we study some important semantic and computational properties in relation to strong and weak forgettings and conflict solving.

5.1 Semantic Characterizations

We observe that the consistency of program Π does not necessarily imply a consistent $SForgetLP(\Pi, P)$ or $WForgetLP(\Pi, P)$ for some set of atoms P, and *vice versa*. For example, consider program $\Pi = \{a \leftarrow, b \leftarrow \mathtt{not} a, \mathtt{not} b\}$, then weakly forgetting a in Π will result in an inconsistent program $\{b \leftarrow \mathtt{not} b\}$. Similarly, strongly forgetting a from an inconsistent program $\Pi = \{b \leftarrow \mathtt{not} a, c \leftarrow b, \mathtt{not} c\}$ will get a consistent program $\{c \leftarrow b, \mathtt{not} c\}$.

To understand why this may happen, we first introduce some notions. Given program Π and a set of atoms P, we specify two programs X and Y. Program X is a subset of Π containing three types of rules in Π: (1) for each $p \in P$, if $p \notin head(\Pi)$, then rule $r \in \Pi$ with $p \in pos(r)$ is in X; (2) for each $p \in P$, if $p \notin pos(\Pi)$, then rule $r \in \Pi$ with $head(r) = \{p\} \subseteq X$; and (3) rule $r \in \Pi$ with $neg(r) \cap P \neq \emptyset$ but not of the types (1) and (2) is also in X. Clearly, X contains those rules of Π satisfying $Atom(r) \cap P \neq \emptyset$ but will not be affected by $Reduct(\Pi, P)$. On the other hand, program Y is obtained as follows: for each rule r in X of the type (3), a replacement of r of the form: $r' : head(r) \leftarrow pos(r), \mathtt{not}(neg(r) - P)$ is in Y. It should be noted that both X and Y can be obtained in linear time in terms of the sizes of Π and P.

Theorem 2 *Let Π be a program and P a set of atoms. A subset S of atoms occurring in $SForgetLP(\Pi, P)$ (or in $WForgetLP(\Pi, P)$) is a stable model of $SForgetLP(\Pi, P)$*

(or $WForgetLP(\Pi, P)$ resp.) iff program $\Pi - X$ (or $(\Pi - X) \cup Y$ resp.) has a stable model S' such that $S = S' - P$.

Theorem 2 presents an interesting result: given program Π and set of atoms P, although computing $SForgetLP(\Pi, P)$ or $WForgetLP(\Pi, P)$ may need exponential time (see Section 5.3), its stable models, however, can be computed through some program that is obtained from Π in linear time.

Now we consider the existence of (preferred) solutions for logic program contexts in conflict solving. It is easy to see that not every logic program context has a preferred solution. For instance, $\Sigma = (\{a \leftarrow \texttt{not}\, a\}, \emptyset, \emptyset))$ has no solution, and hence has no preferred solution neither. The following result shows that the existence of a Σ's solution always implies the existence of Σ's preferred solution, and *vice versa*.

Theorem 3 *Let Σ be a logic program context. Σ has a preferred solution iff $Solution(\Sigma) \neq \emptyset$.*

Although deciding whether a Σ has a (preferred) solution is NP-hard (see Theorem 6 in section 5.3), we can identify a fairly general class of logic program contexts whose solutions always exist (we showed in our full paper that all major logic program update approaches can be transformed into the following form of conflict solving context)..

Proposition 3 *Let $\Sigma = (\Phi_1, \cdots, \Phi_n)$ be a logic program context. If for each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i)$, $\mathcal{C}_i$ is consistent, and for each $r \in \Pi_i$, $\mathcal{F}_i \cap Atom(r) \neq \emptyset$, then $Solution(\Sigma) \neq \emptyset$.*

Proof: We show that $\Sigma' = (\Phi'_1, \cdots, \Phi'_n)$, where $\Phi'_i = (\emptyset, \mathcal{C}_i, \mathcal{F}_i)$ $(1 \leq i \leq n)$ is a solution of Σ. Since for each i, $\mathcal{F}_i \cap Atom(r) \neq \emptyset$ for each $r \in \Pi_i$, we have $\Pi'_i = SForgetLP(\Pi_i, \mathcal{F}_i) = \emptyset$ (note that this is because we already assumed that Π_i does not contain any rules with empty heads. Instead, this type of rule is contained in $\mathcal{C}_i$). This follows that $\Pi'_i \cup \mathcal{C}_j = \mathcal{C}_j$ for all $j = 1, \cdots, n$ are consistent. So $((\emptyset, \mathcal{C}_1, \mathcal{F}_1), \cdots, (\emptyset, \mathcal{C}_n, \mathcal{F}_n))$ is a solution of Σ. □

5.2 Representing Logic program Updates

One major advantage of the proposed framework of logic program contexts is that it can represent new conflict solving scenarios for which the traditional logic program update approaches may not handle properly, like the one discussed in section 1 (or Example 4). In fact, our framework can also represent previous logic program update approaches. To illustrate this, we take Sakama and Inoue's update approach [Sakama and Inoue, 1999] as an example (note that we need to restrict their approach to a normal logic program setting).

Definition 8 *[Sakama and Inoue, 1999] Let Π_1 and Π_2 be two consistent logic programs. Program Π' is a SI-*result *of a theory update of Π_1 by Π_2 if (1) Π' is consistent, (2) $\Pi_2 \subseteq \Pi' \subseteq \Pi_1 \cup \Pi_2$, and (3) there is no other consistent program Π'' such that $\Pi' \subset \Pi'' \subseteq \Pi_1 \cup \Pi_2$.*

Now we transform Sakama and Inoue's theory update into a logic program context. First, for each rule $r \in \Pi_1$, we introduce a new atom l^r which does not occur in $Atom(\Pi_1 \cup \Pi_2)$. Then we define a program Π'_1: for each $r \in \Pi_1$, rule $r' : head(r) \leftarrow pos(r), \texttt{not}(neg(r) \cup \{l^r\})$ is in Π'_1. That is, for each $r \in \Pi_1$, we simply extend its negative body with a unique atom l^r. This will make each r' in Π'_1 be removable by strongly forgetting atom l^r without influencing other rules. Finally, we specify $\Sigma_{SI} = (\Phi_1, \Phi_2)$, where $\Phi_1 = (\Pi'_1, \emptyset, \{l^r \mid r \in \Pi_1\})$ and $\Phi_2 = (\emptyset, \Pi_2, \emptyset)$.

For convenience, we also use $\Pi^{-\texttt{not}P}$ to denote a program obtained from Π by removing all occurrences of atoms in P from the negative bodies of all rules in Π. For instance, if $\Pi = \{a \leftarrow b, \texttt{not}c, \texttt{not}d\}$, then $\Pi^{-\texttt{not}\{c\}} = \{a \leftarrow b, \texttt{not}d\}$. Now we have the following characterization result.

Theorem 4 *Let Π_1 and Π_2 be two consistent programs, and Σ_{SI} as specified above. Π' is a SI-result of updating Π_1 by Π_2 iff $\Pi' = \Pi^{-\texttt{not}\{l^r \mid r \in \Pi_1\}} \cup \Pi_2$, where $\Sigma' = ((\Pi, \emptyset, \{l^r \mid r \in \Pi_1\}), (\emptyset, \Pi_2, \emptyset))$ is a preferred solution of Σ_{SI}.*

In our full paper we have showed that other logic program update approaches dealing with sequence of programs including Eiter et al's causal rejection and Dynamic Logic Programming [Eiter *et al.*, 2002; Leite, 2003] can also be embedded into our framework. In this sense, the logic program context provides a unified framework for logic program updates.

5.3 Complexity Results

We assume that readers are familiar with the complexity classes of P, NP, coNP, $\Sigma_2^{\texttt{P}}$ and $\Pi_2^{\texttt{P}} = \texttt{co}\Sigma_2^{\texttt{P}}$. The class of DP contains all languages L such that $L = L_1 \cap L_2$ where L_1 is in NP and L_2 is in coNP. The class coDP is the complement of class DP (readers refer to [Papadimitriou, 1994] for further details).

We observe that the main computation of strong and weak forgettings relies on the procedure of reduction that further inherits the computation of program unfolding. Hence, in general computing strong and weak forgetting may need exponential steps of rule substitutions in terms of the sizes of the input program and the set of forgotten atoms. However, the following result shows that the inference problem associated to strong and weak forgettings still remains in coNP.

Proposition 4 *Let Π be a logic program, P a set of atoms, and a an atom. Then deciding whether $SForgetLP(\Pi, P) \models a$ (or $WForgetLP(\Pi, P) \models a$) is* coNP*-complete.*

Proof: (*Sketch*) The hardness is followed by setting $P = \emptyset$, and the membership can be proved by using Theorem 2. □

Now we consider the complexity of *irrelevance* in relation to strong and weak forgettings and conflict solving. From a semantic viewpoint, the irrelevance tells us whether a strong/weak forgetting or conflict solving procedure will affect some particular atoms occurring in the underlying programs. Hence, studying its associated computational properties is important.

Definition 9 ***(Irrelevance)*** *Let Π be a logic program, P a set of atoms, and a an atom. We say that a is* irrelevant *to P in Π, if either $\Pi \models a$ iff $SForgetLP(\Pi, P) \models a$, or $\Pi \models a$ iff $WForgetLP(\Pi, P) \models a$.*

We generalize the notion of irrelevance to the logic program context. Formally, let Σ be a logic program context and a an atom, we say that a is *derivable* from Σ's ith component, denoted as $\Sigma \models_i a$, if $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma$ and $\Pi_i \models a$.

Definition 10 *(**Irrelevance wrt logic program contexts**) Let Σ and Σ' be two logic program contexts where $\Sigma' \in Solution(\Sigma)$, and a an atom. We say that a is* irrelevant *with respect to Σ and Σ' on their ith components, or simply say that a is $(\Sigma, \Sigma')^i$-irrelevant, if $\Sigma \models_i a$ iff $\Sigma' \models_i a$.*

Theorem 5 *Let Π be a logic program, P a set of atoms, a an atom, and Σ and Σ' two logic program contexts where $\Sigma' \in Solution(\Sigma)$. The the following results hold:*

1. *Deciding whether a is irrelevant to P in Π is* coDP*-complete;*
2. *Deciding whether a is $(\Sigma, \Sigma')^i$-irrelevant is* coDP*-complete.*

Proof: (*Sketch*) We describe the main idea of proving the hardness part of Result 1. a is irrelevant to P in Π if (1) $\Pi \models a$ iff $SForgetLP(\Pi, P) \models a$, or (2) $\Pi \models a$ iff $WForgetLP(\Pi, P) \models a$. Here we consider case (1) and proof for case (2) is the same. Let (Φ_1, Φ_2) be a pair of CNFs, where $\Phi_1 = \{C_1, \cdots, C_m\}$ and $\Phi_2 = \{C'_1, \cdots, C'_n\}$, and each C_i and C'_j ($1 \leq i \leq n$, $1 \leq j \leq n$) are sets of propositional literals respectively. We also assume $Atom(\Phi_1) \cap Atom(\Phi_2) = \emptyset$. We know that deciding whether Φ_1 is satisfiable or Φ_2 is unsatisfiable is coDP-complete [Papadimitriou, 1994]. We construct a program Π polynomially based on set $Atom(\Phi_1) \cup Atom(\Phi_2) \cup \hat{X} \cup \hat{Y} \cup \{l_1, \cdots, l_n, p, a, sat^{\Phi_1}, unsat^{\Phi_1}, unsat^{\Phi_2}\}$, where any two sets of atoms are disjoint and $|\hat{X}| = |Atom(\Phi_1)|$ and $|\hat{Y}| = |Atom(\Phi_2)|$. Π consists of four groups of rules where atom p only occurs in Π_4:
Π_1: rules to generate all truth assignments of Φ_1 and Φ_2;
Π_2: rules to derive $unsat^{\Phi_1}$ and $unsat^{\Phi_2}$ if Φ_1 and Φ_2 are unsatisfiable respectively;
Π_3: rules to force a truth assignment of Φ_2 making $\neg\Phi_2$ true if $unsat^{\Phi_2}$ is derivable from Π;
Π_4 contains 4 rules: $sat^{\Phi_1} \leftarrow \texttt{not}\, unsat^{\Phi_1}$, $a \leftarrow sat^{\Phi_1}$, $unsat^{\Phi_2} \leftarrow \texttt{not}\, a$, $p \leftarrow$.
Then we can prove that Φ_1 is satisfiable or Φ_2 is unsatisfiable iff ($\Pi \models a$ and $SForgetLP(\Pi, \{p\}) \models a$) or ($\Pi \not\models a$ and $SForgetLP(\Pi, \{p\}) \not\models a$). □

Finally, the following theorem summarizes major complexity results of conflict solving in logic program contexts.

Theorem 6 *Let $\Sigma = (\Phi_1, \cdots, \Phi_n)$ and $\Sigma' = (\Phi'_1, \cdots, \Phi'_n)$ be two logic program contexts, where for each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma$ ($1 \leq i \leq n$), $\Phi'_i \in \Sigma'$ is of the form $\Phi'_i = (\Pi'_i, \mathcal{C}_i, \mathcal{F}_i)$, where $\Pi'_i = SForgetLP(\Pi_i, P_i)$ or $\Pi'_i = WForgetLP(\Pi_i, P_i)$ for some $P_i \subseteq \mathcal{F}_i$.*

1. *Deciding whether Σ has a preferred solution is* NP*-hard;*
2. *Deciding whether Σ' is a solution of Σ is* NP*-complete;*
3. *Deciding whether Σ' is a preferred solution of Σ is in Π_2^{P}, if strong and weak forgettings in Σ can be computed in polynomial time* [1]*;*
4. *For a given atom a, deciding whether for each $\Sigma'' \in Solution(\Sigma)$, $\Sigma'' \models_i a$ is in Π_2^{P}, if strong and weak forgettings in Σ can be computed in polynomial time.*

[1] In our full paper, we have classified certain classes of logic program contexts, where for each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma$ and each $P_i \subseteq \mathcal{F}_i$, $SForgetLP(\Pi_i, P_i)$ and $WForgetLP(\Pi_i, P_i)$ can always be computed in polynomial time.

6 Conclusions

In this paper, we defined notions of strong and weak forgettings in logic programs, which may be viewed as an analogy of forgetting in propositional theories. Based on these notions, we developed a general framework of logic program contexts for conflict solving and studied the related semantic and computational properties.

Our work presented in this paper can be extended in several directions. One interesting topic is to associate dynamic preferences to sets of forgettable atoms and components in logic program contexts, so that the extended framework is more flexible to handle task-dependent conflict solving.

References

[Brass and Dix, 1999] S. Brass and J. Dix. Semantics of (disjunctive) logic programs based on partial evaluation. *Journal of Logic programming*, 40(1):1–46, 1999.

[Eiter *et al.*, 2002] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of update sequences based on causal rejection. *Theory and Practice of Logic programming*, 2:711–767, 2002.

[Lang and Marquis, 2002] J. Lang and P. Marquis. Resolving inconsistencies by variable forgetting. In *Proceedings of KR2002*, pages 239–250, 2002.

[Lang *et al.*, 2003] J. Lang, P. Liberatore, and P. Marquis. Propositional independence - formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.

[Leite, 2003] J.A. Leite. *Evolving Knowledge Bases: Specification and Semantics*. IOS Press, 2003.

[Lifschitz *et al.*, 2001] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):426–541, 2001.

[Lin and Reiter, 1994] F. Lin and R. Reiter. Forget it! In *Working Notes of AAAI Fall Symposium on Relevance*, pages 154–159, 1994.

[Lin, 2001] F. Lin. On the strongest necessary and weakest sufficient conditions. *Artificial Intelligence*, 128:143–159, 2001.

[Papadimitriou, 1994] C.H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[Sakama and Inoue, 1999] C. Sakama and K. Inoue. Updating extended logic programs through abduction. In *Proceedings of LPNMR'99*, pages 2–17, 1999.

[Su *et al.*, 2004] K. Su, G. Lv, and Y. Zhang. Reasoning about knowledge by variable forgetting. In *Proceedings of KR2004*, pages 576–586, 2004.