Available online at www.sciencedirect.com
SCIENCE DIRECT®
Artificial Intelligence 170 (2006) 739–778
Artificial Intelligence
www.elsevier.com/locate/artint

Solving logic program conflict through strong and weak forgettings [☆]

Yan Zhang [a,*], Norman Y. Foo [b]

[a] *Intelligent Systems Laboratory, School of Computing and Mathematics, University of Western Sydney, Penrith South DC, NSW 1797, Australia*
[b] *School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia*

Received 6 September 2005; received in revised form 23 February 2006; accepted 23 February 2006
Available online 29 March 2006

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, reveal their close connections to the notion of forgetting in classical propositional theories, and provide a precise semantic characterization for them. Based on these notions, we then develop a general framework for conflict solving in logic programs. We investigate various semantic properties and features in relation to strong and weak forgettings and conflict solving in the proposed framework. We argue that many important conflict solving problems can be represented within this framework. In particular, we show that all major logic program update approaches can be transformed into our framework, under which each approach becomes a specific conflict solving case with certain constraints. We also study essential computational properties of strong and weak forgettings and conflict solving in the framework.

Keywords: Conflict solving; Knowledge representation; Answer set semantics; Logic program update; Computational complexity

1. Introduction

1.1. Motivation

One promising approach in the research of reasoning about knowledge dynamics is to represent agents' knowledge bases as logic programs on which necessary updates/revisions are conducted as a way of modeling agents' knowledge evolution. A key issue in this study is to solve various conflicts and inconsistencies in logic programs, e.g. [15].

We observe that some typical conflict solving problems in applications are essential in reasoning about agents' knowledge change, but they may not be properly handled by traditional logic program updates. Let us consider a

[☆] Some results presented in this paper were published in IJCAI-2005 and AAAI-2005 [Y. Zhang, N.Y. Foo, K. Wang, Solving logic program conflict through strong and weak forgettings, in: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05), 2005, pp. 627–632; Y. Zhang, N.Y. Foo, A unified framework for representing logic program updates, in: Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05), 2005, pp. 707–712].

[*] Corresponding author.
E-mail addresses: yan@cit.uws.edu.au (Y. Zhang), norman@cse.unsw.edu.au (N.Y. Foo).

0004-3702/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2006.02.002

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 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. We represent John's knowledge base Π_J:

$$r_1\colon\ \mathit{complete}(\mathit{John}, \mathit{Assignment}) \leftarrow \mathit{help}(\mathit{Sue}, \mathit{John}),$$

$$r_2\colon\ \mathit{help}(\mathit{Sue}, \mathit{John}) \leftarrow \mathit{not}\ \mathit{Busy}(\mathit{Sue}),$$

$$r_3\colon\ \mathit{goodFriend}(\mathit{John}, \mathit{Tom}) \leftarrow,$$

$$r_4\colon\ \mathit{copy}(\mathit{Tom}, \mathit{Assignment}) \leftarrow \mathit{goodFriend}(\mathit{John}, \mathit{Tom}),\ \mathit{complete}(\mathit{John}, \mathit{Assignment}),$$

and Sue's knowledge base Π_S:

$$r_5\colon\ \mathit{hate}(\mathit{Sue}, \mathit{Tom}) \leftarrow,$$

$$r_6\colon\ \leftarrow \mathit{help}(\mathit{Sue}, \mathit{John}),\ \mathit{copy}(\mathit{Tom}, \mathit{Assignment}).$$

In order to take Sue's knowledge base into account, John may update his knowledge base Π_J in terms of Sue's Π_S. In this way, John obtains a solution: $\Pi_J^{\mathit{final}} = \{r_1, r_2, r_3, r_5, r_6\}$ 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 intention to contain the information that Sue hates Tom into his new knowledge base.

As an alternative, John may just weaken his knowledge base by *forgetting* atom $\mathit{copy}(\mathit{Tom}, \mathit{Assignment})$ from Π_J in order to accommodate Sue's constraint on help. Then John will have a new program $\Pi_J^{\mathit{final}'} = \{r_1, r_2, r_3\}$—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 [18] 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. [13,14,19,24]. Then a natural question is: whether can we develop an analogous theory of forgetting in logic programs and apply it as a foundational basis for various conflict solving in logic programs? This paper provides an answer to this question.

1.2. Summary of contributions of this paper

The main contributions of this paper can be summarized as follows.

(1) We define two notions of strong and weak forgettings in logic programs under answer set programming semantics. We reveal their close connections to the notion of forgetting in classical propositional theories, and provide a precise semantic characterization for them.
(2) Based on these notions, we develop a general framework for conflict solving called *logic program contexts*. Under this framework, conflicts can be solved by strongly or/and weakly forgetting certain sets of atoms from corresponding programs. We show that our framework is general enough to represent many important conflict solving problems. In particular, for the first time we demonstrate that all major logic program update approaches can be transformed into our framework.
(3) We investigate essential computational properties in relation to strong and weak forgettings and conflict solving in the proposed framework. Specifically, we show that under the answer set programming with no disjunction in the head, the associated inference problem for strong and weak forgettings is coNP-complete, and the irrelevance problem related to strong and weak forgettings and conflict solving is coDP-complete. We also study other computational problems related to the computation of strong and weak forgetting and conflict solving.

1.3. Structure of the paper

The rest of this paper is organized as follows. We first present preliminary definitions and concepts in Section 2. In Section 3, we give formal definitions of strong and weak forgettings in logic programs, and present their essential properties. Based on notions of strong and weak forgettings, in Section 4 we propose a framework called logic program contexts for general conflict solving in logic programs. In Section 5, we investigate various semantic properties and features in relation to strong and weak forgettings and conflict solving in the proposed framework. In Section 6, we show that our conflict solving framework is general enough to represent all major logic program update approaches. In Section 7, we study essential computational properties of strong and weaking forgettings and conflict solving. Finally, in Section 8 we conclude the paper with some discussions.

2. Preliminaries

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

$$a \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n, \tag{1}$$

where a is either a propositional atom or empty, $b_1, \ldots, b_m$, $c_1, \ldots, c_n$ are propositional atoms, and *not* presents the negation as failure. From (1) we know that a normal logic program does not contain classical negation and has no disjunction in the head. When a is empty, rule (1) is called a *constraint*. Given a rule r of the form (1), we denote $head(r) = \{a\}$, $pos(r) = \{b_1, \ldots, b_m\}$, $neg(r) = \{c_1, \ldots, c_n\}$, and $body(r) = pos(r) \cup neg(r)$. Therefore, rule (1) may simply be represented as the form:

$$head(r) \leftarrow pos(r), not\ neg(r), \tag{2}$$

here we denote $not\ neg(r) = \{not\ c_1, \ldots, not\ c_n\}$. We also use $atom(r)$ to denote the set of all atoms occurring in rule r. For a program Π, we define notions $head(\Pi) = \bigcup_{r\in\Pi} head(r)$, $pos(\Pi) = \bigcup_{r\in\Pi} pos(r)$, $neg(\Pi) = \bigcup_{r\in\Pi} neg(r)$, $body(\Pi) = \bigcup_{r\in\Pi} body(r)$, and $atom(\Pi) = \bigcup_{r\in\Pi} atom(r)$. Given sets of atoms P and Q, we may use notion

$$r'\colon\ head(r) \leftarrow \big(pos(r) - P\big), not\big(neg(r) - Q\big)$$

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 *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 $a \leftarrow b_1, \ldots, b_m$ from Π, if $b_1, \ldots, b_m \in S$, 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 if $c \in S$; and (2) all subformulas 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 [7]. We also call Π^S is the result of Gelfond–Lifschitz transformation on Π with S. It is easy to see that 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.

Two programs 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 [17]. The concept of strong equivalence can be used to simplify a program. For example, if two programs are strongly equivalent, then whenever one program is contained in a particular program, it can be replaced by the other program safely. The following observation gives two instances for this case which will be useful for our later formalization.

Observation 1. *Let Π be a logic program Π. Then Π is strongly equivalent to the empty set iff 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) *$pos(r) \cap neg(r) \neq \emptyset$.*[1]

[1] This result can be viewed as a special case of more general results proved in [9] and [20] respectively.

For convenience in the later reference in this paper, we call the two types of rules mentioned above *valid 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 into the clause: $a \vee \neg b_1 \vee \cdots \vee \neg b_m \vee c_1 \vee \cdots \vee c_m$. Note that this is not a translation in a classical sense since here we replace negation as failure *not* with classical negation $\neg$. For instance, if $\Pi = \{a \leftarrow \mathit{not}\ b, c \leftarrow a\}$, then we have $[\Pi]^C = (a \vee b) \wedge (c \vee \neg a)$. In general, we may write $[\Pi]^C = \{C_1, \ldots, C_n\}$ where each C_i is a conjunct of $[\Pi]^C$. If C is a clause, we call any subformula of C a *subclause* of C.

Now we introduce the notion of forgetting in a classical propositional theory [18,19]. Let T be a propositional theory. We use $T[p/\mathit{true}]$ (or $T[p/\mathit{false}]$, 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 $T[q/\mathit{true}] = \{r \supset s\}$ and $T[q/\mathit{false}] = \{\neg p\}$.[2] 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 $\mathit{Forget}(T, P)$, is defined inductively as follows:

$$\begin{aligned}
&\mathit{Forget}(T, \emptyset) = T,\\
&\mathit{Forget}\big(T, \{p\}\big) = T[p/\mathit{true}] \vee T[p/\mathit{false}],\\
&\mathit{Forget}\big(T, P \cup \{p\}\big) = \mathit{Forget}\big(\mathit{Forget}(T, p), P\big).
\end{aligned}$$

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 $\mathit{Forget}(T, \{q\}) = \{(r \supset s) \vee \neg p\}$.

3. Strong and weak forgettings in logic programs

3.1. Definitions

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. Those atoms having certain connections to forgotten atoms through rules in the program might or might not be affected depending on the situation, while all other atoms should not be affected. We observe that the forgetting definition in propositional theories cannot be directly used for logic programs as logic programs themselves cannot be disjuncted together. Further, different ways of handling negation as failure in forgetting may also lead to different resulting programs.

For example, suppose we have a program Π containing two rules:

$$\begin{aligned}
&a \leftarrow b,\\
&b \leftarrow c.
\end{aligned}$$

Now if we want to forget atom b, we can simply remove the second rule and replace the first rule with $a \leftarrow c$. In this case, forgetting b is just to remove b through the rule replacement. However, things become not so simple if we change the program to:

$$\begin{aligned}
&a \leftarrow \mathit{not}\ b,\\
&b \leftarrow c,
\end{aligned}$$

and we still want to forget atom b. In this case, the method of replacement mentioned above seems not working because replacing the first rule with $a \leftarrow \mathit{not}\ c$ will change the entire semantics of the program. One way we can do is to completely remove the second rule since b is forgotten, and the first rule may be either reduced to $a \leftarrow$ or completely removed depending on whether we assume b true or false. These two examples actually reflect our intuition of defining forgetting notions in logic programs.

To formalize our idea of forgetting in logic programs, we first introduce a program transformation called *reduction*. The intuition behind reduction may be easily illustrated as follows. Given a program $\Pi = \{p \leftarrow q, p' \leftarrow p, \mathit{not}\ q'\}$, performing a reduction on Π with respect to atom p will result in a new program $\Pi' = \{p' \leftarrow q, \mathit{not}\ q'\}$. The formal definition is presented as follows.

[2] For convenience, we may consider a finite set of formulas as a single conjunction of all elements in the set.

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'': $head(r') \leftarrow (pos(r') - \{p\}), pos(r), not\ (neg(r) \cup neg(r'))$; (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 propositional atoms. Then the reduction of Π with respect to P is inductively defined as follows:

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

Note that in our program reduction definition, Step (1) is the same as Sakama and Seki's [23] and Brass and Dix's [4] *unfolding* in logic programs. While unfolding is to eliminate positive middle occurrences of an atom in a logic program, the reduction, on other hand, is further to remove those rules with heads of this atom. Now let us consider a program $\Pi = \{a \leftarrow b, b \leftarrow a, d \leftarrow not\ e\}$. Then

$$Reduct\big(Reduct\big(\Pi, \{a\}\big), \{b\}\big) = \{b \leftarrow b, d \leftarrow not\ e\}, \quad \text{and}$$
$$Reduct\big(Reduct\big(\Pi, \{b\}\big), \{a\}\big) = \{a \leftarrow a, d \leftarrow not\ e\}.$$

A brief glimpse of this example seems to indicate that the program reduction is not well defined since these two programs look different. However, it is easy to see that they are strongly equivalent, and both can be simplified to $\{d \leftarrow not\ e\}$. The following proposition actually shows that our program reduction is well defined under the strong equivalence.

Proposition 1. *Let Π be a logic program and p, q two propositional atoms. Then $Reduct(Reduct(\Pi, \{p\}), \{q\})$ is strongly equivalent to $Reduct(Reduct(\Pi, \{q\}), \{p\})$.*

Proof. To prove this result, we need to consider a general case of iterated reductions which captures all possible features. For this purpose, it is sufficient to deal with a program $\Pi = \Pi_1 \cup \Pi_2$, where all possible reductions related to atoms p and q are only happened within Π_1. That is, we can assume Π_1 consists of six parts: $\Pi_{11} \cup \Pi_{12} \cup \Pi_{13} \cup \Pi_{14} \cup \Pi_{15} \cup \Pi_{16}$:

Π_{11}:

r_1: $p \leftarrow pos(r_1), not\ neg(r_1)$,

$\ldots,$

r_h: $p \leftarrow pos(r_h), not\ neg(r_h)$,

Π_{12}:

r_{h+1}: $p \leftarrow q, pos(r_{h+1}), not\ neg(r_{h+1})$,

$\ldots,$

r_k: $p \leftarrow q, pos(r_k), not\ neg(r_k)$,

Π_{13}:

r_{k+1}: $q \leftarrow pos(r_{k+1}), not\ neg(r_{k+1})$,

$\ldots,$

r_l: $q \leftarrow pos(r_l), not\ neg(r_l)$,

Π_{14}:

r_{l+1}: $q \leftarrow p, pos(r_{l+1}), not\ neg(r_{l+1})$,

$\ldots,$

r_m: $q \leftarrow p, pos(r_m), not\ neg(r_m)$,

Π_{15}:

r_{m+1}: $a_{m+1} \leftarrow p, pos(r_{m+1}), not\ neg(r_{m+1})$,

$\ldots$,

r_n: $a_n \leftarrow p, pos(r_n), not\ neg(r_n)$,

Π_{16}:

r_{n+1}: $b_{n+1} \leftarrow q, pos(r_{n+1}), not\ neg(r_{n+1})$,

$\ldots$,

r_s: $b_s \leftarrow q, pos(r_s), not\ neg(r_s)$,

where $a_i \neq p$, $a_i \neq q$, $b_j \neq p$, and $b_j \neq q$ for all a_i and b_j, and also p, q do not occur in all $pos(r_i)$ ($i = 1, \ldots, s$). We assume that p, q are not in $head(\Pi_2)$ and $pos(\Pi_2)$, i.e. no reduction related to p or q will occur in Π_2.

It is not hard to see that the above Π covers all possible cases of reductions of Π with respect to atoms p and q. In order to avoid a tedious proof, without loss of generality, we may consider a simplified version of program Π as follows. $\Pi = \Pi_1 \cup \Pi_2$, where Π_1 contains the following rules:

r_1: $p \leftarrow pos(r_1), not\ neg(r_1)$,

r_1': $p \leftarrow q, pos(r_1'), not\ neg(r_1')$,

r_2: $q \leftarrow pos(r_2), not\ neg(r_2)$,

r_2': $q \leftarrow p, pos(r_2'), not\ neg(r_2')$,

r_3: $a \leftarrow p, pos(r_3), not\ neg(r_3)$,

r_4: $b \leftarrow q, pos(r_4), not\ neg(r_4)$.

We assume p and q do not occur in $pos(r_1), pos(r_1'), pos(r_2)$ and $pos(r_2')$. Also, all rules in Π_2 do not contain p or q in their heads and positive bodies. We should mention that our following proof can be extended to the general case of Π as constructed earlier.

Firstly, we have $Reduct(\Pi, \{p\}) = \Pi_1' \cup \Pi_2$ consists of the following rules:

r_2: $q \leftarrow pos(r_2), not\ neg(r_2)$,

r_2': $q \leftarrow (pos(r_1) \cup pos(r_2')), not(neg(r_1) \cup neg(r_2'))$,

r_2'': $q \leftarrow q, (pos(r_1') \cup pos(r_2')), not\ (neg(r_1') \cup neg(r_2'))$,

r_3': $a \leftarrow (pos(r_1) \cup pos(r_3)), not(neg(r_1) \cup neg(r_3))$,

r_3'': $a \leftarrow q, (pos(r_1') \cup pos(r_3)), not(neg(r_1') \cup neg(r_3))$,

r_4: $b \leftarrow q, pos(r_4), not\ neg(r_4)$.

From Observation 1, we know that $\{r_2''\}$ is strongly equivalent to the empty set. So we have $\Pi_1' = \{r_2, r_2', r_3', r_3'', r_4\}$. Then by the reduction of $\Pi_1' \cup \Pi_2$ with respect to $\{q\}$, we have the following result: $Reduct(Reduct(\Pi, \{p\}), \{q\}) = \Pi_1'' \cup \Pi_2$, where Π_1'' contains the following rules:

r_3': $a \leftarrow (pos(r1) \cup pos(r_3)), not(neg(r_1) \cup neg(r_3))$,

r^*: $a \leftarrow (pos(r1') \cup pos(r_2) \cup pos(r_3)), not(neg(r_1') \cup neg(r_2) \cup neg(r_3))$,

$r^{*'}$: $a \leftarrow (pos(r1) \cup pos(r1') \cup pos(r_2') \cup pos(r_3)), not\ (neg(r_1) \cup neg(r_1') \cup neg(r_2') \cup neg(r_3))$,

r_4': $b \leftarrow (pos(r_2) \cup pos(r_4)), not(neg(r_2) \cup neg(r_4))$,

r_4'': $b \leftarrow (pos(r_1) \cup pos(r_2') \cup pos(r_4)), not(neg(r_1) \cup neg(r_2) \cup neg(r_4))$.

It is easy to see that programs $\{r_3', r^{*'}\}$ and $\{r_3'\}$ are strongly equivalent because $pos(r_3') \subseteq pos(r^{*'})$ and $neg(r_3') \subseteq neg(r^{*'})$. Therefore, rule $r^{*'}$ can be removed. So finally, we have $\Pi_1'' = \{r_3', r^*, r_4', r_4''\}$.

Now we consider $Reduct(\Pi, \{q\})$. It is easy to see that $Reduct(\Pi, \{q\}) = \Pi^\dagger \cup \Pi_2$, where $\Pi^\dagger$ consists of the following rules:

$$r_1\colon\ p \leftarrow pos(r_1), not\ neg(r_1),$$
$$r_1''\colon\ p \leftarrow \big(pos(r_1') \cup pos(r_2)\big), not\big(neg(r_1') \cup neg(r_2)\big),$$
$$r_1'''\colon\ p \leftarrow p, \big(pos(r_1') \cup pos(r_2')\big), not\big(neg(r_1') \cup neg(r_2')\big),$$
$$r_3\colon\ a \leftarrow p, pos(r_3), not\ neg(r_3),$$
$$r_4'\colon\ b \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),$$
$$r_4''\colon\ b \leftarrow p, \big(pos(r_2') \cup pos(r_4)\big), not\big(neg(r_2') \cup neg(r_4)\big).$$

Also, rule r_1''' can be removed from $\Pi^\dagger$. So we have $\Pi^\dagger = \{r_1, r_1'', r_3, r_4', r_4''\}$. Then $Reduct(Reduct(\Pi, \{q\}), \{p\}) = \Pi^\ddagger \cup \Pi_2$, where $\Pi^\ddagger$ consists of the following rules:

$$r_3'\colon\ a \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big),$$
$$r^*\colon\ \leftarrow \big(pos(r_1') \cup pos(r_2) \cup pos(r_3)\big), not\big(neg(r_1') \cup neg(r_2) \cup neg(r_3)\big),$$
$$r_4'\colon\ b \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),$$
$$r_4''\colon\ b \leftarrow \big(pos(r_1) \cup pos(r_2') \cup pos(r_4)\big), not\big(neg(r_1) \cup neg(r_2') \cup neg(r_4)\big),$$
$$r^\dagger\colon\ b \leftarrow \big(pos(r_1) \cup pos(r_2) \cup pos(r_2') \cup pos(r_4)\big), not\big(neg(r_1) \cup neg(r_2) \cup neg(r_2') \cup neg(r_4)\big).$$

Since $pos(r_4'') \subseteq pos(r_\dagger)$, we know that programs $\{r_4'', r^\dagger\}$ and $\{r^\dagger\}$ are strongly equivalent. So $r^\dagger$ can be removed from $\Pi^\ddagger$. Therefore, $\Pi^\ddagger = \{r_3', r^*, r_4', r_4''\} = \Pi_1''$. This proves our result. $\square$

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

Definition 2 *(Strong forgetting).* Let Π be a logic program, and p a propositional 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) $\Pi' = \Pi' - \{r \mid r \text{ is a valid rule}\}$;
(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 middle occurrences* of p in rules with other rules having p as the head. Step 2 is to remove all valid rules which may be introduced by the reduction of Π with respect to p. From Observation 1, 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 reduction, there does not exist any pair of rules r and r' such that $head(r) = \{p\}$ and $p \in pos(r')$. The intuitive meaning of these two steps 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 $neg(r)$ as 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 *not* p. Therefore we call this kind of forgetting *strong forgetting*.

Definition 2 can be easily extended to the case of strongly forgetting a set of atoms:

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

With a different way of dealing with negation as failure, we have a weak version of forgetting as defined below.

Definition 3 *(Weak forgetting).* Let Π be a logic program, and p a propositional atom. We define a program to be the result of *weakly forgetting* p in Π, denoted as $WForgetLP(\Pi, \{p\})$, if it is obtained from the following transformation:

(1) $\Pi' = Reduct(\Pi, \{p\})$;
(2) $\Pi' = \Pi' - \{r \mid r$ is a valid rule$\}$;
(3) $\Pi' = \Pi' - \{r \mid head(r) = \{p\}\}$;
(4) $\Pi' = \Pi' - \{r \mid p \in pos(r)\}$;
(5) $\Pi' = \Pi' - \Pi^* \cup \Pi^\dagger$, where $\Pi^* = \{r \mid p \in neg(r)\}$ and
$\Pi^\dagger = \{r' \mid r'\colon\ head(r) \leftarrow pos(r), not(neg(r) - \{p\})$ where $r \in \Pi^*\}$;
(6) $WForgetLP(\Pi, \{p\}) = \Pi'$.

$WForgetLP(\Pi, \{p\})$ is defined in the same way as $SForgetLP(\Pi, \{p\})$ except Step 5. Suppose we have a rule like $r\colon\ b \leftarrow pos(r), not\ neg(r)$ 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 $r'\colon\ b \leftarrow pos(r), not(neg(r) - \{p\})$. The notion of weakly forgetting a set of atoms, denoted as $WForgetLP(\Pi, P)$, is defined accordingly:

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

The following proposition ensures that our strong and weak forgettings in logic programs are well defined under strong equivalence.

Proposition 2. *Let Π be a logic program and p, q two propositional atoms. Then*

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

Proof. We only prove Result 1, as Result 2 is proved in a similar way. Similar to the proof of Proposition 1, without loss of generality, we consider a program $\Pi = \Pi_1 \cup \Pi_2$, where Π_1 contains the following rules:

$$r_1\colon\ p \leftarrow pos(r_1), not\ neg(r_1),$$
$$r_1'\colon\ p \leftarrow q, pos(r_1'), not\ neg(r_1'),$$
$$r_2\colon\ q \leftarrow pos(r_2), not\ neg(r_2),$$
$$r_2'\colon\ q \leftarrow p, pos(r_2'), not\ neg(r_2'),$$
$$r_3\colon\ a \leftarrow p, pos(r_3), not\ neg(r_3),$$
$$r_4\colon\ b \leftarrow q, pos(r_4), not\ neg(r_4).$$

We assume p and q do not occur in $pos(r_1), pos(r_1'), pos(r_2)$ and $pos(r_2')$. Also, all rules in Π_2 do not contain p or q in their heads and positive bodies, but may contain $not\ p$ or $not\ q$.

Then we have $Reduct(\Pi, \{p\}) = Reduct(\Pi_1, \{p\}) \cup \Pi_2$, where, according to the proof of Proposition 1, $Reduct(\Pi_1, \{p\})$ consists of the following rules:

$$r_2\colon\ q \leftarrow pos(r_2), not\ neg(r_2),$$
$$r_2'\colon\ q \leftarrow \big(pos(r_1) \cup pos(r_2')\big), not\big(neg(r_1) \cup neg(r_2')\big),$$

$$r_2'': q \leftarrow q, \big(pos(r_1') \cup pos(r_2')\big), not\big(neg(r_1') \cup neg(r_2')\big),$$
$$r_3': a \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big),$$
$$r_3'': a \leftarrow q, \big(pos(r_1') \cup pos(r_3)\big), not\big(neg(r_1') \cup neg(r_3)\big),$$
$$r_4: b \leftarrow q, pos(r_4), not\ neg(r_4).$$

Then after Step 2 (removing valid rules), rule r_2'' is removed. So we can write *SForgetLP*($\Pi, \{p\}) = \Pi_1' \cup \Pi_2'$, where $\Pi_1' = \{r_2, r_2', r_3', r_3'', r_4\}$, and $\Pi_2' \subseteq \Pi_2$ in which all rules containing *not p* are removed. Note that rules in Π_1' may be removed if they contain *not p*, according to Step 5 in the transformation.

Now we consider *SForgetLP*($\Pi_1' \cup \Pi_2', \{q\}$). Since Π_2' does not contain any rule having q in its head or positive body, *Reduct*($\Pi_1' \cup \Pi_2', \{q\}$) = *Reduct*($\Pi_1', \{q\}) \cup \Pi_2'$. By ignoring the details, we will have the final resulting program: *SForgetLP*($\Pi_1' \cup \Pi_2', \{q\}) = \Pi_1'' \cup \Pi_2''$, where Π_1'' consists of the following rules:

$$r_3': a \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big),$$
$$r^*: a \leftarrow \big(pos(r_1') \cup pos(r_2) \cup pos(r_3)\big), not\big(neg(r_1') \cup neg(r_2) \cup neg(r_3)\big),$$
$$r_4': b \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),$$
$$r_4'': b \leftarrow \big(pos(r_1) \cup pos(r_2') \cup pos(r_4)\big), not\big(neg(r_1) \cup neg(r_2) \cup neg(r_4)\big),$$

and $\Pi_2'' \subseteq \Pi_2'$ in which all rules containing *not q* are removed. Again, rules among $\{r_3', r^*, r_4', r_4''\}$ will be removed if they contain *not q*. Let us denote the resulting program after such elimination as $\Pi_1^{*'}$, i.e. $\Pi_1^{*'} \subseteq \Pi_1''$ where each rule in Π_1'' containing *not p* or *not q* is removed from $\Pi_1^{*'}$.

Let us examine the result of *SForgetLP*(*SForgetLP*($\Pi, \{q\}), \{p\}$). Firstly, we have *Reduct*($\Pi, \{q\}$) = *Reduct*(Π_1, $\{q\}) \cup \Pi_2$, where *Reduct*($\Pi_1, \{q\}$) consists of the following rules:

$$r_1: p \leftarrow pos(r_1), not\ neg(r_1),$$
$$r_1'': p \leftarrow \big(pos(r_1') \cup pos(r_2)\big), not\big(neg(r_1') \cup neg(r_2)\big),$$
$$r_1''': p \leftarrow p, \big(pos(r_1') \cup pos(r_2')\big), not\big(neg(r_1') \cup neg(r_2')\big),$$
$$r_3: a \leftarrow p, pos(r_3), not\ neg(r_3),$$
$$r_4': b \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),$$
$$r_4'': b \leftarrow p, \big(pos(r_2') \cup pos(r_4)\big), not\big(neg(r_2') \cup neg(r_4)\big).$$

Again, after Step 2, rule r_1''' is removed. So we can write *SForgetLP*($\Pi, \{q\}) = \Pi_1^* \cup \Pi_2^*$, where $\Pi_1^* = \{r_1, r_1'', r_3, r_4', r_4''\}$, and $\Pi_2^* \subseteq \Pi_2$ in which all rules containing *not q* are removed. Also rules in Π_1^* will be removed if they contain *not q*.

Now we consider *SForgetLP*($\Pi_1^* \cup \Pi_2^*, \{p\}$). Since Π_2^* does not contain any rule having p in its head or positive body, *Reduct*($\Pi_1^* \cup \Pi_2^*, \{p\}$) = *Reduct*($\Pi_1^*, \{p\} \cup \Pi_2^*$). Then we have *Reduct*($\Pi_1^*, \{p\}) = \Pi_1^{*'}$, which has the following rules:

$$r_3': a \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big),$$
$$r^*: \leftarrow \big(pos(r_1') \cup pos(r_2) \cup pos(r_3)\big), not\big(neg(r_1') \cup neg(r_2) \cup neg(r_3)\big),$$
$$r_4': b \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),$$
$$r_4'': b \leftarrow \big(pos(r_1) \cup pos(r_2') \cup pos(r_4)\big), not\big(neg(r_1) \cup neg(r_2') \cup neg(r_4)\big).$$

This program is Π_1'' as we have shown above. Also note that rules in $\{r_3', r^*, r_4', r_4''\}$ will be removed if they contain *not p* according to Step 5. Then clearly, such resulting program is $\Pi_1^{*'}$ as mentioned above.

So after performing Steps 2–5, we finally have *SForgetLP*($\Pi_1^* \cup \Pi_2^*, \{p\}) = \Pi_1^{*'} \cup \Pi_2^{*'}$, where $\Pi_2^{*'} \subseteq \Pi_2^*$, in which all rules containing *not p* are removed. Obviously, $\Pi_2^{*'} = \Pi_2''$. This proves our result. □

Example 2. Let $\Pi_1 = \{a \leftarrow not\ b, b \leftarrow not\ a\}$, and $\Pi_2 = \{a \leftarrow b, not\ c, a \leftarrow not\ e, d \leftarrow a, e, e \leftarrow not\ a\}$. Then

$$\begin{aligned}
&SForgetLP\big(\Pi_1, \{a\}\big) = \emptyset,\\
&SForgetLP\big(\Pi_1, \{a, b\}\big) = \emptyset,\\
&WForgetLP\big(\Pi_1, \{a\}\big) = \{b \leftarrow\},\\
&WForgetLP\big(\Pi_1, \{a, b\}\big) = \emptyset,\\
&SForgetLP\big(\Pi_2, \{a\}\big) = \{d \leftarrow b, e, not\ c\},\\
&WForgetLP\big(\Pi_2, \{a\}\big) = \{d \leftarrow b, e, not\ c, e \leftarrow\}.
\end{aligned}$$

3.2. Relationship to forgetting in propositional theories

As we argued earlier, the notion of forgetting in propositional theories is not applicable to logic programs generally. However, as we will show next, there are close connections between forgetting in propositional theories and strong and weak forgettings in logic programs. Let us first consider the following example.

Example 3. Let $\Pi = \{b \leftarrow a, c, d \leftarrow not\ a, e \leftarrow not\ f\}$. Then we have

$$\begin{aligned}
&SForgetLP\big(\Pi, \{a\}\big) = \{e \leftarrow not\ f\}, \quad \text{and}\\
&WForgetLP\big(\Pi, \{a\}\big) = \{d \leftarrow, e \leftarrow not\ f\}.
\end{aligned}$$

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

$$\begin{aligned}
&\models Forget\big([\Pi]^C, \{a\}\big) \supset \big[SForgetLP\big(\Pi, \{a\}\big)\big]^C, \quad \text{and}\\
&\models \big[WForgetLP\big(\Pi, \{a\}\big)\big]^C \supset Forget\big([\Pi]^C, \{a\}\big).
\end{aligned}$$

The above example motivates us to examine deeper connections between strong and weak forgettings in logic programs and forgetting in propositional theories. To begin with, we introduce a useful notion. 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 $[Reduct(\Pi', P)]^C$ is a single clause and L is a subclause of $[Reduct(\Pi', P)]^C$. Intuitively, the coherence notion tries to specify those clauses that are parts of clauses generated from program Π through reduction.

Consider program $\Pi = \{a \leftarrow b, d \leftarrow a, not\ c, e \leftarrow not\ d\}$. Clause $d \vee \neg b$ 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 3. *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 $\models [\Pi]^C \supset \Phi$.*

Proof. Note that if L is Π-coherent, then we can find a subset Π' of Π and a set of atoms $P \subseteq atom(\Pi)$, such that $Reduct(\Pi', P)$ only contains one rule r and L is a subclause of $[\{r\}]^C$. Recall that the reduction $Reduct(\Pi', P)$ is just to eliminate positive middle occurrences of P in rules of Π' and remove the rules with heads of P if such positive middle occurrences exist in Π'. Then it is easy to observe that $\models [\Pi]^C \supset [Reduct(\Pi', P)]^C$. If $L = [Reduct(\Pi', P)]^C$, then $\models [\Pi]^C \supset L$. If L is a proper subclause of $[Reduct(\Pi', P)]^C$, then $\models L \supset [Reduct(\Pi', P)]^C$. This proves our result. $\square$

Definition 4. 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 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 Π-coherent. φ_2 is a *weakest premiss* of φ *with respect to* Π if φ_2 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 4. (*Example* 3 *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.

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.*

Proof. We only prove the first part of the result, while the second part is proved in a similar way. To simplify our proof, we consider set P to be a singleton, i.e. $P = \{p\}$. The general case can be proved by induction on the size of P. Without loss of generality, we assume program Π is of the following form: $\Pi = \Pi_1 \cup \Pi_2 \cup \Pi_3$, where Π_1 only contains rules which are related to the process of the reduction of Π with respect to p, Π_2 does not contain any rules containing p in heads or positive bodies (i.e. Π_2 is irrelevant to the reduction process) but contains rules having p in their negative bodies, and Π_3 does not contain any rules having p in their heads, positive or negative bodies. Obviously, Π_3 is irrelevant to the process of strongly forgetting p in Π. In particular, we assume Π_1 and Π_2 have the following forms:

Π_1:

r_1: $p \leftarrow pos(r_1), not\ neg(r_1)$,

$\ldots$,

r_k: $p \leftarrow pos(r_k), not\ neg(r_k)$,

r_{k+1}: $q_{k+1} \leftarrow p, pos(r_{k+1}), not\ neg(r_{k+1})$,

$\ldots$,

r_m: $q_m \leftarrow p, pos(r_m), not\ neg(r_m)$,

Π_2:

r_{m+1}: $q_{m+1} \leftarrow pos(r_{m+1}), not\ p, not\ neg(r_{m+1})$,

$\ldots$,

r_n: $q_n \leftarrow pos(r_n), not\ p, not\ neg(r_n)$.

In Π_1, we may assume that p is not in $pos(r_i)$ for $i = 1, \ldots, m$ (otherwise, those rules having p as heads can be omitted from Π_1 according to Observation 1). For Π_2, on the other hand, p is not in $pos(r_j)$ for $j = m+1, \ldots, n$.

Then according to Definition 1, we have $Reduct(\Pi, \{p\}) = \Pi_1' \cup \Pi_2 \cup \Pi_3$, where Π_1' is as follows:

$r_{1,k+1}$: $q_{k+1} \leftarrow pos(r_1), pos(r_{k+1}), not\ neg(r_1), not\ neg(r_{k+1})$,

$\ldots$,

$r_{1,m}$: $q_m \leftarrow pos(r_1), pos(r_m), not\ neg(r_1), not\ neg(r_m)$,

$\ldots$,

$r_{k,k+1}$: $q_{k+1} \leftarrow pos(r_k), pos(r_{k+1}), not\ neg(r_k), not\ neg(r_{k+1})$,

$\ldots$,

$r_{k,m}$: $q_m \leftarrow pos(r_k), pos(r_m), not\ neg(r_k), not\ neg(r_m)$.

Note that p may occur in negative bodies of some rules in Π_1'. However, to simplify our proof, we may consider that no p occurs in negative bodies in all rules of Π_1' because p's occurrences in negative bodies have been presented in the case of Π_2.

Now we consider $SForgetLP(\Pi, \{p\})$. Clearly, $SForgetLP(\Pi, \{p\}) = \Pi'_1 \cup \Pi_3$, where Π_2 is removed from Step 5 in Definition 2. Then we conclude that

$$[SForgetLP(\Pi, \{p\})]^C = [\Pi'_1]^C \wedge [\Pi_3]^C,$$

where $[\Pi'_1]^C$ consists of the following clauses:

$$\Big(q_{k+1} \vee \neg pos(r_1) \vee \neg pos(r_{k+1}) \vee \bigvee neg(r_1) \vee \bigvee neg(r_{k+1})\Big),^3$$
$$\ldots$$
$$\Big(q_m \vee \neg pos(r_k) \vee \neg pos(r_m) \vee \bigvee neg(r_k) \vee \bigvee neg(r_m)\Big).$$

Obviously, each clause of $SForgetLP(\Pi, \{p\})$ is Π-coherent.

Now we consider $Forget([\Pi]^C, \{p\})$. Firstly, it is to observe that

$$Forget([\Pi]^C, \{p\}) = \Phi \wedge [\Pi_3]^C,$$

where Φ is formula $([\Pi_1]^C[p/true] \wedge [\Pi_2]^C[p/true]) \vee ([\Pi_1]^C[p/false] \wedge [\Pi_2]^C[p/false])$. $[\Pi_1]^C[p/true] \wedge [\Pi_2]^C[p/true]$ consists of the following clauses:

$$q_{k+1} \vee \neg pos(r_{k+1}) \vee \bigvee neg(r_{k+1}),$$
$$\ldots,$$
$$q_m \vee \neg pos(r_m) \vee \bigvee neg(r_m),$$

and $[\Pi_1]^C[p/false] \wedge [\Pi_2]^C[p/false]$ contains the following clauses:

$$\neg pos(r_1) \vee \bigvee neg(r_1),$$
$$\ldots,$$
$$\neg pos(r_k) \vee \bigvee neg(r_k),$$
$$q_{m+1} \vee \neg pos(r_{m+1}) \vee \bigvee neg(r_{m+1}),$$
$$\ldots,$$
$$q_n \vee \neg pos(r_n) \vee \bigvee neg(r_n).$$

Then by translating Φ into CNF, say $Con(\Phi)$, it is easy to see that all clauses of $[\Pi'_1]^C$ are contained in $Con(\Phi)$. So $[SForgetLP(\Pi, \{p\})]^C$ is a consequence of $Forget([\Pi]^C, \{p\})$ with respect to Π.

Observing $Con(\Phi)$'s structure, we know that $Con(\Phi)$ also contains the following clauses:

$$q_{k+1} \vee \neg pos(r_{k+1}) \vee \bigvee neg(r_{k+1}) \vee q_{m+1} \vee \neg pos(r_{m+1}) \vee \bigvee neg(r_{m+1}),$$
$$\ldots,$$
$$q_{k+1} \vee \neg pos(r_{k+1}) \vee \bigvee neg(r_{k+1}) \vee q_n \vee \neg pos(r_n) \vee \bigvee neg(r_n),$$
$$\ldots,$$
$$q_m \vee \neg pos(r_m) \vee \bigvee neg(r_m) \vee q_n \vee \neg pos(r_n) \vee \bigvee neg(r_n).$$

According to the structure of Π, none of these clauses is Π-coherent. Therefore, there does not exist another consequence φ' of $Forget([\Pi]^C, \{p\})$ with respect to Π such that $\models \varphi' \supset [SForgetLP(\Pi, \{p\})]^C$. This proves our result. □

Theorem 1 actually states that under a certain set of propositional atoms P, the conjunctive normal form of the strong forgetting of P in program Π is the strongest formula which is implied by the forgetting of P in the corresponding propositional theory, while the conjunctive normal form of the weak forgetting of P in Π is the weakest

[3] Here $\neg pos(r)$ presents the disjunction of all negative atoms whose atoms occur in $pos(r)$ and $\bigvee neg(r)$ presents the disjunction of all atoms in $neg(r)$.

formula that implies it. So semantically, our notions of strong and weak forgettings in logic programs are strongest necessary and weakest sufficient conditions respectively for the forgetting in the corresponding propositional theory.

3.3. A semantic characterization

From previous presentation, we can see that our strong and weak forgettings are defined in a syntactic way. This is one of the major differences comparing with the forgetting notion in propositional theories, where an equivalent model theoretic semantics is provided for the resulting theory after forgetting some atoms [19]. Although we do not have corresponding model theoretic definitions for strong and weak forgettings, the following property precisely characterizes the stable models of strong and weak forgettings.

Firstly, 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 $\Pi = \{a \leftarrow, b \leftarrow not\ a, not\ b\}$, then weakly forgetting a in Π will result in an inconsistent program $\{b \leftarrow not\ b\}$. Similarly, strongly forgetting a from an inconsistent program $\Pi = \{b \leftarrow not\ a,\ c \leftarrow b, not\ c\}$ will get a consistent program $\{c \leftarrow b, not\ c\}$. Theorem 2 explains how this happens.

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 $head(r) = \{p\}$ is in 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': $head(r) \leftarrow pos(r), not(neg(r) - P)$ 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. Then we have the following result.

Theorem 2. *Let Π be a program and P a set of atoms. A set of atoms S 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 $S = S' - P$.*

Proof. From the definition of X, we can see that X contains exactly all those rules of Π that are not affected by $Reduct(\Pi, P)$ but have to be removed from $SForgetLP(\Pi, P)$. So we have $SForgetLP(\Pi, P) = Reduct(\Pi, P) - X = Reduct((\Pi - X), P)$ (we suppose that no valid rule is presented here as it does not influence the result). So it is easy to see that $SForgetLP(\Pi, P)$ has a stable model S iff $\Pi - X$ has a stable model S' where $S = S' - P$. Similarly, we can observe that $WForgetLP(\Pi, P) = Reduct((\Pi - X) \cup Y, P)$. $\square$

It is interesting to note that given program Π and set of atoms P, although computing $SForgetLP(\Pi, P)$ or $WForgetLP(\Pi, P)$ may need exponential time (see Section 7), its stable models can be computed through some program that is obtained from Π in linear time.

4. Logic program contexts—A framework for conflict solving

In this section, we define a general framework called logic program contexts 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 will be shown, notions of strong and weak forgettings that we proposed earlier will provide a basis for solving such conflicts.

Definition 5 *(Logic program context).* A *logic program context* is an n-ary tuple $\Sigma = (\Phi_1, \ldots, \Phi_n)$, 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. A logic program context 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 Definition 5, each component Φ_i in Σ 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. Now the problem of conflict solving under this setting can be stated as follows: given a logic program context $\Sigma = (\Phi_1, \ldots, \Phi_n)$, which may not be consistent or conflict-free, how can we find an alternative logic program context $\Sigma' = (\Phi'_1, \ldots, \Phi'_n)$ such that Σ' is conflict-free *and* is *closest* to the original Σ in some sense.

We first present formal definitions about the solution that solves conflicts in a logic program context.

Definition 6 *(Solution).* Given a logic program context $\Sigma = (\Phi_1, \ldots, \Phi_n)$, 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) For each Φ'_i in Σ', $\Phi'_i = (\Pi'_i, \mathcal{C}_i, \mathcal{F}_i)$, where $\Pi'_i = \mathit{SForgetLP}(\Pi_i, P_i)$ or $\Pi'_i = \mathit{WForgetLP}(\Pi_i, P_i)$ for some $P_i \subseteq \mathcal{F}_i$.

We denote the set of all solutions of solving conflicts in Σ as $\mathit{Solution}(\Sigma)$.

Definition 7 *(Ordering on solutions).* Given three logic program contexts Σ, Σ' and Σ'' where $\Sigma', \Sigma'' \in \mathit{Solution}(\Sigma)$. 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 $\Pi'_i = \mathit{SForgetLP}(\Pi_i, P_i)$ or $\Pi'_i = \mathit{WForgetLP}(\Pi_i, P_i)$ for some $P_i \subseteq \mathcal{F}_i$, and $\Pi''_i = \mathit{SForgetLP}(\Pi_i, Q_i)$ or $\Pi''_i = \mathit{WForgetLP}(\Pi_i, Q_i)$ 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 $\Sigma'' \not\preceq_\Sigma \Sigma'$.

Proposition 4. *$\preceq_\Sigma$ is a partial ordering.*

Proof. From the definition of $\preceq_\Sigma$, it is easy to see that $\preceq_\Sigma$ is reflexive and antisymmetric. So we only need to show $\preceq_\Sigma$ is transitive, and this is obvious according to Definition 7. □

Definition 8 *(Preferred solution).* Given two logic program contexts Σ and Σ'. We say that Σ' is a *preferred solution* that solves conflicts in Σ, if $\Sigma' \in \mathit{Solution}(\Sigma)$ and there does not exist another $\Sigma'' \in \mathit{Solution}(\Sigma)$ such that $\Sigma'' \prec_\Sigma \Sigma'$.

It should be noted that in order to achieve a preferred solution, both strong and weak forgettings may have to apply alternatively. Consider the following simple example.

Example 5. Let $\Sigma = (\Phi_1, \Phi_2)$, where

$$\begin{array}{llll}
\Phi_1: & & \Phi_2: & \\
\quad \Pi_1: & a \leftarrow, & \quad \Pi_2: & c \leftarrow, \\
& b \leftarrow a, \mathit{not}\ c, & & d \leftarrow \mathit{not}\ e, \\
& d \leftarrow a, \mathit{not}\ e, & & e \leftarrow, \\
& f \leftarrow d, & & f \leftarrow d, \\
\quad \mathcal{C}_1: & \leftarrow d, \mathit{not}\ f, & \quad \mathcal{C}_2: & \leftarrow b, \mathit{not}\ c, \\
& \leftarrow f, \mathit{not}\ d, & & \\
\quad \mathcal{F}_1: & \{a, b, c\}, & \quad \mathcal{F}_2: & \{a, b, c, d, e, f\}.
\end{array}$$

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

$$\begin{aligned}
&\Phi'_1 = \big(\mathit{SForgetLP}\big(\Pi_1, \{c\}\big), \mathcal{C}_1, \mathcal{F}_1\big), \\
&\Phi'_2 = \big(\mathit{WForgetLP}\big(\Phi_2, \{e\}\big), \mathcal{C}_2, \mathcal{F}_2\big), \quad \text{and} \\
&\Phi''_1 = \big(\mathit{WForgetLP}\big(\Pi_1, \{b, c\}\big), \mathcal{C}_1, \mathcal{F}_1\big), \\
&\Phi''_2 = \big(\mathit{WForgetLP}\big(\Phi_2, \{e\}\big), \mathcal{C}_2, \mathcal{F}_2\big).
\end{aligned}$$

It can be verified that both Σ_1 and Σ_2 are solutions that solve the conflict in Σ, but only Σ_1 is a preferred solution.

From the above example, we can see that to solve conflicts in a logic program context, the agent may apply strong forgettings, weak forgettings, or both to obtain a (preferred) solution. In this sense, the agent has a freedom to choose the ways of conflict solving if no specific constraint is taken into account. It is noted that sometimes solving conflict through strong forgetting will loose more atoms than weak forgetting, or *vice versa*. Therefore, in order to minimally forget atoms from a logic program, the agent can apply strong and weak forgettings alternatively in different components. However, in practice, it may be more desirable for an agent to use a unified approach in conflict solving. Our approach provided here can certainly accommodate this requirement by simply re-defining the solution of a logic program context by applying strong or weak forgetting only.

Example 6. We consider a conflict solving scenario. A couple John and Mary are discussing their family investment plan. They consider to invest four types of different shares *shareA, shareB, shareC* and *shareD*, where *shareA* and *shareB* are of high risk but also have high returns, and *shareC* and *shareD* are property investment shares and hence are of lower risk and may be suitable for a long term investment. John is very interested in *shareA* and wants to buy it definitely. He also tends to invest *shareB* if they invest neither *shareC* nor *shareD*. However, if they do not invest *shareB*, John may consider to invest *shareC* or *shareD* if the house price will keep growing, which John is actually not sure yet. But John does not consider to invest both of them. On the other hand, Mary is more conservative. She prefers to invest both *shareC* and *shareD* because she believes that the house price will continue growing as she is confident that the government has no plan to increase the Reserve Bank interest. Mary definitely does not consider to invest both *shareA* and *shareB*. At most, she may consider to buy some *shareB* if they invest neither *shareA* nor *shareC*. But Mary insists that they should invest at least one of *shareC* and *shareD* in any case. Now how can John and Mary negotiate to achieve a common agreement?

We first represent John and Mary's investment preferences as the following programs respectively:

$$
\begin{aligned}
&\Pi_J:\\
&\quad r_1:\ \mathit{shareA} \leftarrow,\\
&\quad r_2:\ \mathit{shareB} \leftarrow \mathit{not\ shareC}, \mathit{not\ shareD},\\
&\quad r_3:\ \mathit{shareC} \leftarrow \mathit{houseIncrease}, \mathit{not\ shareB}, \mathit{not\ shareD},\\
&\quad r_4:\ \mathit{shareD} \leftarrow \mathit{houseIncrease}, \mathit{not\ shareB}, \mathit{not\ shareC},\\
&\Pi_M:\\
&\quad r_5:\ \mathit{shareC} \leftarrow \mathit{houseIncrease},\\
&\quad r_6:\ \mathit{shareD} \leftarrow \mathit{houseIncrease},\\
&\quad r_7:\ \mathit{shareB} \leftarrow \mathit{not\ shareA}, \mathit{not\ shareC},\\
&\quad r_8:\ \mathit{houseIncrease} \leftarrow \mathit{not\ interestUp}.
\end{aligned}
$$

To negotiate with each other, John and Mary set up their conditions respectively that they do not want to compromise:

$$
\begin{aligned}
&\mathcal{C}_J:\\
&\qquad \leftarrow \mathit{not\ shareA},\\
&\qquad \leftarrow \mathit{shareC}, \mathit{shareD}, \quad \text{and}\\
&\mathcal{C}_M:\\
&\qquad \leftarrow \mathit{shareA}, \mathit{shareB},\\
&\qquad \leftarrow \mathit{not\ shareC}, \mathit{not\ shareD}.
\end{aligned}
$$

John and Mary then specify a logic program context to solve the conflict about their family investment plan: $\Sigma_{JM} = ((\Pi_J, \mathcal{C}_J, F_J), (\Pi_M, \mathcal{C}_M, F_M))$, where $F_J = \{\mathit{shareB}, \mathit{shareC}, \mathit{shareD}\}$ (note that *shareA* is not a forgettable atom for John as he definitely wants to buy it) and $F_M = \{\mathit{shareA}, \mathit{shareB}, \mathit{shareC}, \mathit{shareD}\}$.

Unfortunately, it is easy to check that Σ_{JM} has no (preferred) solution. That means, it is impossible for John and Mary to solve their conflict by just weakening their own belief sets. So John and Mary realize that they have to make further compromise that both of them should not only weaken their own belief sets, but also take the other's beliefs into account. However, their strategy is to take the other's beliefs as little as possible. To this end, John and Mary specify a new logic program context as follows: $\Sigma_{JM}^{New} = ((\Pi_J \cup \Delta_M, C_J, F'_J), (\Pi_M \cup \Delta_J, C_M, F'_M))$, where

Δ_M:

r'_5: $shareC \leftarrow houseIncrease, not\ l_{r'_5}$,

r'_{51}: $l_{r'_5} \leftarrow not\ h_{r'_5}$,

r'_6: $shareD \leftarrow houseIncrease, not\ l_{r'_6}$,

r'_{61}: $l_{r'_6} \leftarrow not\ h_{r'_6}$,

r'_7: $shareB \leftarrow not\ shareA, not\ shareC, not\ l_{r'_7}$,

r'_{71}: $l_{r'_7} \leftarrow not\ h_{r'_7}$,

r'_8: $houseIncrease \leftarrow not\ interestUp, \quad not\ l_{r'_8}$,

r'_{81}: $l_{r'_8} \leftarrow not\ h_{r'_8}$,

Δ_J:

r'_1: $shareA \leftarrow not\ l_{r'_1}$,

r'_{11}: $l_{r'_1} \leftarrow not\ h_{r'_1}$,

r'_2: $shareB \leftarrow not\ shareC, not\ shareD, not\ l_{r'_2}$,

r'_{21}: $l_{r'_2} \leftarrow not\ h_{r'_2}$,

r'_3: $shareC \leftarrow houseIncrease, not\ shareB, not\ shareD, not\ l_{r'_3}$,

r'_{31}: $l_{r'_3} \leftarrow not\ h_{r'_3}$,

r'_4: $shareD \leftarrow houseIncrease, not\ shareB, not\ shareC, not\ h_{r'_4}$,

r'_{41}: $l_{r'_4} \leftarrow not\ h_{r'_4}$,

and $F'_J = F_J \cup \{h_{r'_i}, l_{r'_i} \mid i = 5, \ldots, 8\}$ and $F'_M = F_M \cup \{h_{r'_i}, l_{r'_i} \mid i = 1, \ldots, 4\}$ ($h_{r'_i}, l_{r'_i}$ ($i = 1, \ldots, 8$) are newly introduced atoms).

Let us take a closer look at Δ_M. During the conflict solving, if none of $h_{r'_i}, l_{r'_i}$ ($i = 5, \ldots, 8$) has been strongly or weakly forgotten, then all rules r'_i ($i = 5, \ldots, 8$) in Δ_M equipped with the corresponding rules from Π_M will be defeated. In this case, John does not need to take any of Mary's beliefs into his consideration. On the other hand, if for some j ($5 \leqslant j \leqslant 8$) $h_{r'_j}$ is strongly forgotten (or $l_{r'_j}$ is weakly forgotten), then rules r'_j in Δ_M will be initiated and hence will affect John's decision for conflict solving. As only a minimal number of $h_{r'_i}$ (or $l_{r'_i}$) ($i = 5, \ldots, 8$) will be strongly forgotten (or weakly forgotten, resp.) in the conflict solving, John just takes a minimal number of Mary's rules for his consideration. The same explanation applies for Δ_J.

Σ_{JM}^{New} has a unique preferred solution $((\Pi'_J, C_J, F'_J), (\Pi'_M, C_M, F'_M))$, where

$$\Pi'_J = WForgetLP\big(\Pi_J \cup \Delta_M, \{shareB, l_{r'_8}\}\big), \quad \text{and}$$
$$\Pi'_M = WForgetLP\big(\Pi_M \cup \Delta_J, \{shareC, l_{r'_1}\}\big).$$

Π'_J has two stable models which include {*shareA*, *shareC*} and {*shareA*, *shareD*} respectively, and Π'_M has one stable model including *shareA* and *shareD*. Therefore, John has two options: either to invest *shareA* and *shareC*, or to invest *shareA* and *shareD*, while Mary will only consider to invest *shareA* and *shareD*. Finally, John and Mary can reach an agreement to invest *shareA* and *shareD*.

Example 6 presents an application of our approach to solve complex logic program conflicts involving negotiation and belief merging that most of current methods have difficulties to deal with.

5. Semantic properties

In this section, we study important semantic properties in relation to strong and weak forgettings and logic program contexts.

5.1. Irrelevance

Irrelevance is an important issue related to forgetting [18]. Basically, if we are able to answer an query q against a logic program Π, i.e. $\Pi \models q$, then we are interested in knowing whether we still can answer this query in the resulting program after strongly or weakly forgetting a set of atoms from Π, because this will enable us to significantly simplify the inference problem in the resulting logic program. We first give a formal definition of irrelevance in relation to strong and weak forgetting.

Definition 9 *(Irrelevance).* Let Π be a logic program and P a set of atoms. We say that atom a is *irrelevant* to the strong forgetting (or weak forgetting) of P from Π, or simply say that a is *s-irrelevant* (or *w-irrelevant*, resp.) to P in Π, if $\Pi \models a$ iff $\mathit{SForgetLP}(\Pi, P) \models a$ (or $\mathit{WForgetLP}(\Pi, P) \models a$ resp.). We say that a is *irrelevant* to P in Π if a is either s-irrelevant or w-irrelevant to P in Π.

Trivially, if Π is inconsistent, then a is s-irrelevant (w-irrelevant) to any P in Π iff $\mathit{SForgetLP}(\Pi, P) \models a$ (or $\mathit{WForgetLP}(\Pi, P) \models a$, resp.). Also if for some $P \subseteq \mathit{atom}(\Pi)$, $\mathit{SForgetLP}(\Pi, P)$ ($\mathit{WForgetLP}(\Pi, P)$) is inconsistent, then a is s-irrelevant (or w-irrelevant, resp.) to P in Π iff $\Pi \models a$. To provide a general characterization result for irrelevance, we need a notion of support.

Definition 10. Let Π be a program and a an atom. We define a's *support* with respect to Π to be a set of atoms $\mathit{Support}(a)$ specified as follows:

$$S_0 = \{p \mid p \in \mathit{body}(r) \text{ where } r \in \Pi \text{ and } \mathit{head}(r) = \{a\}\};$$
$$S_{i+1} = S_i \cup \{p \mid p \in \mathit{body}(r) \text{ where } r \in \Pi \text{ and } \mathit{head}(r) \subseteq S_i\};$$
$$\mathit{Support}(a) = \bigcup_{i=0}^{\infty} S_i.$$

An atom $p \in \mathit{Support}(a)$ is called a *positive* (or *negative*) *support* of a if $p \in \mathit{pos}(r)$ (or $\in \mathit{neg}(r)$, resp.) for some rule r occurring in defining $\mathit{Support}(a)$.[4]

Basically, $\mathit{Support}(a)$ contains all atoms that occur in those rules related to a's derivation in program Π. Therefore, changing or removing any rules which contain atoms in $\mathit{Support}(a)$ may affect atom a. It turns out that the notion of support plays an important role in deciding the irrelevance.

Theorem 3. *Let Π be a logic program, P a set of atoms and a an atom. Suppose Π, $\mathit{SForgetLP}(\Pi, P)$ and $\mathit{WForgetLP}(\Pi, P)$ are consistent. Then the following results hold.*

(1) *If $a \notin \mathit{head}(\Pi)$, then a is irrelevant to P in Π;*
(2) *If $a \in P$, then a is irrelevant to P in Π iff $\Pi \not\models a$;*
(3) *If $a \notin P$ and $P \cap \mathit{Support}(a) = \emptyset$, then a is irrelevant to P in Π.*

Proof. Proofs for Results 1 and 2 are trivial. Here we only prove Result 3. To prove this result, we need a result about program splitting from [26]. Before we present this program splitting result, we introduce a notion. Given a program Π and a set of atoms S, we use $e(\Pi, S)$ to denote the program obtained from Π by deleting: (1) each rule in Π having a form *not a* in its body with $a \in S$; and (2) all atoms a in the bodies of the remaining rules with $a \in S$. Intuitively,

[4] Note that an atom in $\mathit{Support}(a)$ could be both positive and negative supports of a.

$e(\Pi, S)$ can be viewed as a simplified form of Π given those atoms in S to be true. Then we can re-state Theorem 5 in [26] under the normal logic program setting:

> A set of atoms S is a stable model of program Π if and only if $\Pi = \Pi_1 \cup \Pi_2$ such that $body(\Pi_1) \cap head(\Pi_2) = \emptyset$, and $S = S_1 \cup S_2$, where S_1 is a stable model of Π_1 and S_2 is a stable model of program $e(\Pi_2, S_1)$.

From the definition of *Support*(a), we can see that Π can be expressed as $\Pi = \Pi_1 \cup \Pi_2$, where Π_1 is the subset of Π containing all rules mentioned in *Support*(a). So we have $\Pi_1 \cap \Pi_2 = \emptyset$. Also, it is observed that $body(\Pi_1) \cap head(\Pi_2) = \emptyset$. Because if this is not true, then there must be some rule $r \in \Pi_2$ such that $body(r) \cap body(\Pi_1) \neq \emptyset$. According to Π_1's construction, this leads to $r \in \Pi_1$ as well. That is, $\Pi_1 \cap \Pi_2 \neq \emptyset$. This is a contradiction. Since $P \cap Support(a) = \emptyset$, it is clear that all rules containing some atoms in P are in Π_2. We may use $\Pi(P)$ to denote this set of rules of Π.

From $body(\Pi_1) \cap head(\Pi_2) = \emptyset$, we know that each stable model S of Π can be expressed as $S = S_1 \cup S_2$, where S_2 is a stable model of program $e(\Pi_2, S_1)$. Also, since rule $r^a \in \Pi_1$, this implies that $\Pi \models a$ iff $\Pi_1 \models a$.

Now from the definitions of strong and weak forgettings and condition $\Pi(P) \subseteq \Pi_2$, we know that both strong and weak forgettings only influence rules in Π_2. So we have

$$SForgetLP(\Pi, P) = \Pi_1 \cup \Pi^{\dagger}, \quad \text{and}$$
$$WForgetLP(\Pi, P) = \Pi_1 \cup \Pi^{\ddagger},$$

where $head(\Pi^{\dagger}) \subseteq head(\Pi_2)$ and $head(\Pi^{\ddagger}) \subseteq head(\Pi_2)$. This follows:

$$\Pi_1 \cap \Pi^{\dagger} = \emptyset, body(\Pi_1) \cap head(\Pi^{\dagger}) = \emptyset, \quad \text{and}$$
$$\Pi_1 \cap \Pi^{\ddagger} = \emptyset, body(\Pi_1) \cap head(\Pi^{\ddagger}) = \emptyset.$$

By the result stated above, we have that each stable model S^s of $SForgetLP(\Pi, P)$ can be expressed as $S^s = S_1 \cup S^{\dagger}$, and each stable model S^w of $WForgetLP(\Pi, P)$ can be expressed as $S^w = S_1 \cup S^{\ddagger}$, where S_1 is a stable model of Π_1, $S^{\dagger}$ and $S^{\ddagger}$ are stable models of $\Pi^{\dagger}$ and $\Pi^{\ddagger}$ respectively.

Finally, from the observation that $\Pi \models a$ iff $\Pi_1 \models a$, we have ($\Pi \models a$ iff $SForgetLP(\Pi, P) \models a$) and ($\Pi \models a$ iff $WForgetLP(\Pi, P) \models a$). This proves our result. □

Theorem 3 provides common conditions under which atom a is both s-irrelevant and w-irrelevant to P in Π. However, we should note that in general, an atom's s-irrelevance does not imply its w-irrelevance, and *vice versa*. Usually we need to deal with these two types of irrelevances separately. The following theorem illustrates different sufficient conditions to ensure these irrelevances respectively.

Theorem 4. *Let Π be a logic program, P a set of atoms and a an atom where $a \notin P$. Suppose that Π, SForgetLP(Π, P) and WForgetLP(Π, P) are consistent. Then the following results hold*:

(1) *If for each $p \in P \cap Support(a)$, p is a negative support of a and $\Pi \not\models p$, then a is w-irrelevant to P in Π*;
(2) *If for each $p \in P \cap Support(a)$, p is a negative support of a and $\Pi \models p$, then a is s-irrelevant to P in Π.*

Proof. We only prove Result 1, while Result 2 can be proved in a similar way. From the proof of Theorem 3, given *Support*(a), program Π can be expressed as $\Pi = \Pi_1 \cup \Pi_2$, where $\Pi_1 \cap \Pi_2 = \emptyset$, Π_1 contains all rules used in computing *Support*(a), and $\Pi \models a$ iff $\Pi_1 \models a$. Now let us consider *WForgetLP*(Π, P). We will show that *WForgetLP*(Π, P) can be also expressed as $WForgetLP(\Pi, P) = \Pi'_1 \cup \Pi'_2$, such that $body(\Pi') \cap head(\Pi'_2) = \emptyset$, and $\Pi'_1 \models a$ iff $\Pi_1 \models a$.

To simplify our presentation, we may assume $P = \{p\}$ where the proof for the general case can be easily extended from this special case. Without loss of generality, we can consider that $\Pi = \Pi_1 \cup \Pi_2$, where Π_1 includes the following rules in relation to P (note that Π_1 may also contain other rules):

$$r_1\colon\ head(r_1) \leftarrow pos(r_1), not\ p, not\ neg(r_1),$$
$$r_2\colon\ p \leftarrow pos(r_2), not\ neg(r_2),$$
$$r_3\colon\ head(r_3) \leftarrow p, pos(r_3), not\ neg(r_3),$$

and Π_2 includes the following rules related to P (again, Π_2 may contain other rules):

$$r_4\colon\ head(r_4) \leftarrow p, pos(r_4), not\ neg(r_4),$$
$$r_5\colon\ head(r_5) \leftarrow pos(r_5), not\ p, not\ neg(r_5),$$

we should indicate that Π_2 does not contain a rule with head of p, because this rule will be contained in Π_1 as a rule used for computing $Support(a)$.

Clearly, by weakly forgetting $\{p\}$ in Π, only rules r_1–r_5 will be affected, and other rules do remain unchanged. Therefore, we have $WForgetLP(\Pi, \{p\}) = \Pi_1' \cup \Pi_2'$, where the only difference between Π_1 and Π_1' are following rules in Π_1':

$$r_1'\colon\ head(r_1) \leftarrow pos(r_1), not\ neg(r_1),$$
$$r_3'\colon\ head(r_3) \leftarrow \big(pos(r_2) \cup pos(r_3)\big), not\big(neg(r_2) \cup neg(r_3)\big),$$

and the only difference between Π_2 and Π_2' are the following rules in Π_2':

$$r_4'\colon\ head(r_4) \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),$$
$$r_5'\colon\ head(r_5) \leftarrow pos(r_5), not\ neg(r_5).$$

This concludes that $body(\Pi_1') \cap head(\Pi_2') = \emptyset$. Now we show that $\Pi_1' \models a$ iff $\Pi_1 \models a$. Observing that in Π_1', weakly forgetting p actually does not affect the derivation of $head(r_3)$, while $head(r_1)$'s derivation might be affected since *not p* has been removed from r_1'. However, note that $\Pi \not\models a$, in the original rule r_1 in Π_1, formula *not p* does not play any role. So removing *not p* has no any effect on a's derivation. This follows that $\Pi_1' \models a$ iff $\Pi_1 \models a$. So a is w-irrelevant to $\{p\}$ in Π. $\square$

Example 7. Consider the following program Π:

$$a \leftarrow not\ b,$$
$$c \leftarrow d,$$
$$e \leftarrow c,$$
$$b \leftarrow not\ c.$$

It is easy to see that a is w-irrelevant to $\{c\}$ in Π. This is because $\Pi \not\models a$ and $WForgetLP(\Pi, \{c\}) = \{a \leftarrow not\ b, e \leftarrow d, b \leftarrow\} \not\models a$. Indeed, since $Support(a) = \{b, c\}$ where c is a negative support and $\Pi \not\models c$, the condition of Result 1 of Theorem 4 holds. We can also verify that a is not s-irrelevant to $\{c\}$ in Π.

Now suppose we add an extra rule into Π: $\Pi' = \Pi \cup \{d \leftarrow\}$. Here we still have $Support(a) = \{b, c\}$ where c is a negative support. However, since $\Pi' \models c$, according to Result 2 in Theorem 4, a is s-irrelevant to $\{c\}$ in Π'. It is also observed that a is *not* w-irrelevant to $\{c\}$ in Π'.

We can 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 11 *(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 $\Sigma' \models_i a$.

Given a logic program context Σ and an atom a, we would like to know whether there is a preferred solution Σ' of Σ such that a is $(\Sigma, \Sigma')^i$-irrelevant. To answer this question, we need to consider the *preservation of irrelevance* along the preferred ordering $\preceq_\Sigma$ on solutions of Σ. That is, if $\Sigma', \Sigma'' \in Solution(\Sigma)$, $\Sigma' \preceq_\Sigma \Sigma''$ and a is $(\Sigma, \Sigma'')^i$-irrelevant, then under what conditions a is also $(\Sigma, \Sigma')^i$-irrelevant. If for each of those more preferred solutions, a's irrelevance is preserved, then eventually, we can obtain a's irrelevance with respect to Σ and its preferred solution.

We formalize this idea as follows. Let Σ, Σ', Σ'' be three logic program contexts and $\Sigma', \Sigma'' \in \mathit{Solution}(\Sigma)$. We say that Σ' and Σ'' are *forgetting-congruent* on their ith components with respect to Σ, denoted as $\Sigma' \sim^i_\Sigma \Sigma''$, if for each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma$,

$$\Phi'_i = \big(\mathit{SForgetLP}(\Pi_i, P'), \mathcal{C}_i, \mathcal{F}_i\big) \in \Sigma',$$
$$\Phi''_i = \big(\mathit{SForgetLP}(\Pi_i, P''), \mathcal{C}_i, \mathcal{F}_i\big) \in \Sigma'',$$

or

$$\Phi'_i = \big(\mathit{WForgetLP}(\Pi_i, P'), \mathcal{C}_i, \mathcal{F}_i\big) \in \Sigma',$$
$$\Phi''_i = \big(\mathit{WForgetLP}(\Pi_i, P''), \mathcal{C}_i, \mathcal{F}_i\big) \in \Sigma'',$$

where $P', P'' \subseteq \mathcal{F}_i$. In other words, if two solutions of Σ are forgetting-congruent on their ith components, it means that both of their ith components are obtained by performing either strong forgettings or weaking forgettings on some sets of atoms from Σ's ith component. We say that two solutions Σ' and Σ'' of Σ are *forgetting-congruent*, denoted as $\Sigma' \sim_\Sigma \Sigma''$, if $\Sigma' \sim^i_\Sigma \Sigma''$ for each i. The following theorem shows that forgetting-congruence is a sufficient condition for preserving irrelevance in terms of the preferred ordering on solutions.

Theorem 5. *Let Σ, Σ', Σ'' be three logic program contexts and $\Sigma', \Sigma'' \in \mathit{Solution}(\Sigma)$, a an atom. Suppose $\Sigma' \preceq_\Sigma \Sigma''$ and a is $(\Sigma, \Sigma'')^i$-irrelevant. Then a is $(\Sigma, \Sigma')^i$-irrelevant if $\Sigma' \sim^i_\Sigma \Sigma''$.*

Proof. To prove this theorem, we need to show that for $\Sigma', \Sigma'' \in \mathit{Solution}(\Sigma)$, if $\Phi'_i = (\Pi'_i = \mathit{SForgetLP}(\Pi_i, P'), \mathcal{C}_i, \mathcal{F}_i)$ and $\Phi''_i = (\Pi''_i = \mathit{SForgetLP}(\Pi_i, P''), \mathcal{C}_i, \mathcal{F}_i)$, or $\Phi'_i = (\Pi'_i = \mathit{WForgetLP}(\Pi_i, P'), \mathcal{C}_i, \mathcal{F}_i)$ and $\Phi''_i = (\Pi''_i = \mathit{WForgetLP}(\Pi_i, P''), \mathcal{C}_i, \mathcal{F}_i)$, where Π_i is in some $\Phi_i \in \Sigma$, $\Phi'_i \in \Sigma'$, $\Phi''_i \in \Sigma''$, $P' \subseteq P'' \subseteq \mathcal{F}_i$, and $\Pi_i \models a$ iff $\Pi''_i \models a$, then $\Pi_i \models a$ iff $\Pi'_i \models a$. Recall that we do not consider invalid strong and weak forgettings, so here we assume that all Π_i, Π'_i and Π''_i are consistent programs.

In order to avoid unnecessary tediousness in our proof, we consider a simplified case in our proof where $P' = \{p\}$ and $P'' = \{p, q\}$. Note that the proof for the general case of $P' \subseteq P''$ can be obtained in a similar way of this proof. Under the assumption of $P' = \{p\}$ and $P'' = \{p, q\}$, program Π_i may be simplified as a form of $\Pi_i = \Pi_{i1} \cup \Pi_{i2} \cup \Pi_{i3}$, where Π_{i1} contains the following rules:

$$r_1\colon\ p \leftarrow \mathit{pos}(r_1), \mathit{not}\ \mathit{neg}(r_1),$$
$$r'_1\colon\ p \leftarrow q, \mathit{pos}(r'_1), \mathit{not}\ \mathit{neg}(r'_1),$$
$$r_2\colon\ q \leftarrow \mathit{pos}(r_2), \mathit{not}\ \mathit{neg}(r_2),$$
$$r'_2\colon\ q \leftarrow p, \mathit{pos}(r'_2), \mathit{not}\ \mathit{neg}(r'_2),$$
$$r_3\colon\ \mathit{head}(r_3) \leftarrow p, \mathit{pos}(r_3), \mathit{not}\ \mathit{neg}(r_3),$$
$$r_4\colon\ \mathit{head}(r_4) \leftarrow q, \mathit{pos}(r_4), \mathit{not}\ \mathit{neg}(r_4).$$

We assume p and q do not occur in anywhere else in Π_{i1}. Π_{i2} contains the rules not having p and q in their heads and positive bodies, but only having p and q in their negative bodies:

$$r_5\colon\ \mathit{head}(r_5) \leftarrow \mathit{pos}(r_5), \mathit{not}\ p, \mathit{not}\ q, \ldots,$$
$$r_6\colon\ \mathit{head}(r_6) \leftarrow \mathit{pos}(r_6), \mathit{not}\ p, \ldots,$$
$$r_7\colon\ \mathit{head}(r_7) \leftarrow \mathit{pos}(r_7), \mathit{not}\ q, \ldots.$$

Finally, Π_{i3} consists of rules not containing p and q in anywhere.

Case 1. Suppose $\Pi'_i = \mathit{SForgetLP}(\Pi_i, \{p\})$ and $\Pi''_i = \mathit{SForgetLP}(\Pi_i, \{p, q\})$. In this case, Π'_i and Π''_i are as follows:

Π'_i:

$$r_2\colon\ q \leftarrow \mathit{pos}(r_2), \mathit{not}\ \mathit{neg}(r_2),$$
$$r'_{21}\colon\ q \leftarrow \big(\mathit{pos}(r_1) \cup \mathit{pos}(r'_2)\big), \mathit{not}\big(\mathit{neg}(r_1) \cup \mathit{neg}(r'_2)\big),$$

r_{31}: $head(r_3) \leftarrow (pos(r_1) \cup pos(r_3)), not(neg(r_1) \cup neg(r_3))$,

r'_{31}: $head(r_3) \leftarrow q, (pos(r'_1) \cup pos(r_3)), not(neg(r'_1) \cup neg(r_3))$,

r_4: $head(r_4) \leftarrow q, pos(r_4), not\ neg(r_4)$,

r_7: $head(r_7) \leftarrow pos(r_7), not\ q, \ldots$,

Π_{i3},

Π''_i:

r_{31}: $head(r_3) \leftarrow (pos(r_1) \cup pos(r_3)), not(neg(r_1) \cup neg(r_3))$,

r_{32}: $head(r_3) \leftarrow (pos(r_2) \cup pos(r'_1) \cup pos(r_3)), not(neg(r_2) \cup neg(r'_1) \cup neg(r_3))$,

r'_{32}: $head(r_3) \leftarrow (pos(r_1) \cup pos(r'_2) \cup pos(r'_1) \cup pos(r_3)), not(neg(r_1) \cup neg(r'_2) \cup neg(r'_1) \cup neg(r_3))$,

r_{33}: $head(r_4) \leftarrow (pos(r_2) \cup pos(r_4)), not(neg(r_2) \cup neg(r_4))$,

r'_{33}: $head(r_4) \leftarrow (pos(r_1) \cup pos(r'_2) \cup pos(r_4)), not(neg(r_1) \cup neg(r'_2) \cup neg(r_4))$,

Π_{i3}.

Now we assume that for some atom a, $\Pi_i \models a$ iff $\Pi''_i \models a$. From the proof of Theorem 3, we know that $\Pi_i \models a$ iff $\Pi^*_i \models a$, where $\Pi^*_i = \{r \mid r \in \Pi_i$ and occurs in the definition of $Support(a)\}$. Let $\Pi'^*_i = \{r \mid r \in \Pi'_i$ and occurs in the definition of $Support(a)\}$ and $\Pi''^*_i = \{r \mid r \in \Pi''_i$ and occurs in the definition of $Support(a)\}$. From $\Pi_i \models a$ iff $\Pi''_i \models a$, we have $\Pi^*_i \models a$ iff $\Pi''^*_i \models a$. Then we will show that $\Pi'^*_i \models a$ iff $\Pi'^*_i \models a$, this will follow $\Pi_i \models a$ iff $\Pi''_i \models a$.

Comparing structures of programs Π_i and Π''_i, it is clear that rules r_5, r_6 and r_7 do not play any role in deriving a even if they are in Π^*_i because these rules are removed from Π''_i. Consequently, rule r_7 does not play any role in deriving a in Π'_i even if it is in Π'^*_i. On the other hand, for all rules in Π'^*_i, they are either in Π''^*_i or have been replaced in Π''^*_i by the corresponding rules after reduction on $\{q\}$. Then we have the fact that $\Pi''^*_i \models b$ iff $\Pi'^*_i \models b$ for all atoms which are not q. Now consider that $a = q$. since $\Pi''_i \not\models q$, and q is $(\Sigma, \Sigma'')^i$-irrelevant, we have $\Pi_i \not\models q$. Then we can conclude that $\Pi''^*_i \not\models q$ as well because if this is not the case, we will have $\Pi_i \models q$ (observing that rules r_1, r_2 and r'_2 used to derive q can be replaced by r_2 and r'_{21} in Π''_i), which contradicts with $\Pi''_i \not\models q$. So the result holds.

Case 2. Suppose $\Pi'_i = WForgetLP(\Pi_i, \{p\})$ and $\Pi''_i = WForgetLP(\Pi_i, \{p, q\})$. In this case, we have:

Π'_i:

r_2: $q \leftarrow pos(r_2), not\ neg(r_2)$,

r'_{21}: $q \leftarrow (pos(r_1) \cup pos(r'_2)), not(neg(r_1) \cup neg(r'_2))$,

r_{31}: $head(r_3) \leftarrow (pos(r_1) \cup pos(r_3)), not(neg(r_1) \cup neg(r_3))$,

r'_{31}: $head(r_3) \leftarrow q, (pos(r'_1) \cup pos(r_3)), not(neg(r'_1) \cup neg(r_3))$,

r_4: $head(r_4) \leftarrow q, pos(r_4), not\ neg(r_4)$,

r'_5: $head(r_5) \leftarrow pos(r_5), not\ q, \ldots$,

r'_6: $head(r_5) \leftarrow pos(r_5), \ldots$,

r_7: $head(r_7) \leftarrow pos(r_7), not\ q, \ldots$,

Π_{i3},

Π''_i:

r_{31}: $head(r_3) \leftarrow (pos(r_1) \cup pos(r_3)), not(neg(r_1) \cup neg(r_3))$,

r_{32}: $head(r_3) \leftarrow (pos(r_2) \cup pos(r'_1) \cup pos(r_3)), not(neg(r_2) \cup neg(r'_1) \cup neg(r_3))$,

r'_{32}: $head(r_3) \leftarrow (pos(r_1) \cup pos(r'_2) \cup pos(r'_1) \cup pos(r_3)), not(neg(r_1) \cup neg(r'_2) \cup neg(r'_1) \cup neg(r_3))$,

r_{33}: $head(r_4) \leftarrow (pos(r_2) \cup pos(r_4)), not(neg(r_2) \cup neg(r_4))$,

r'_{33}: $head(r_4) \leftarrow (pos(r_1) \cup pos(r'_2) \cup pos(r_4)), not(neg(r_1) \cup neg(r'_2) \cup neg(r_4))$,

r''_5: $head(r_5) \leftarrow pos(r_5), \ldots,$

r''_6: $head(r_6) \leftarrow pos(r_6), \ldots,$

r''_7: $head(r_7) \leftarrow pos(r_7), \ldots,$

Π_{i3}.

In a similar way as described above, we can show that $\Pi''_i \models a$ iff $\Pi'_i \models a$. □

Corollary 1. *Let $\Sigma', \Sigma'' \in Solution(\Sigma)$, where Σ'' is a preferred solution of Σ, and a an atom. Then a is $(\Sigma, \Sigma'')^i$-irrelevant if a is $(\Sigma, \Sigma')^i$-irrelevant and $\Sigma' \sim^i_\Sigma \Sigma''$.*

Example 8. Let us consider a logic program context $\Sigma = (\Phi_1, \Phi_2, \Phi_3)$, where

Φ_1:	Φ_2:	Φ_3:
Π_1: $a \leftarrow not\ b$,	Π_2: $d \leftarrow$,	Π_3: $b \leftarrow not\ a$,
$c \leftarrow a$,	$b \leftarrow not\ c$,	$c \leftarrow not\ a$,
$d \leftarrow not\ e$,		$d \leftarrow not\ c$,
$\mathcal{C}_1$: $\emptyset$,	$\mathcal{C}_2$: $\leftarrow not\ d$,	$\mathcal{C}_3$: $\leftarrow c, d$,
	$e \leftarrow c$,	
$\mathcal{F}_1$: $\{a, b, c, d, e\}$,	$\mathcal{F}_2$: $\{b, c, d\}$,	$\mathcal{F}_3$: $\{a, b, c, d\}$.

It is easy to see that conflicts occur in Σ. That is, $\Pi_1 \cup \mathcal{C}_2$, $\Pi_1 \cup \mathcal{C}_3$, and $\Pi_3 \cup \mathcal{C}_2$ are inconsistent. By performing strong and weak forgettings, we obtain a solution of Σ: $\Sigma' = (\Phi'_1, \Phi_2, \Phi'_3)$, where $\Phi'_1 = (\Pi'_1, \mathcal{C}_1, \mathcal{F}_1)$, $\Phi'_3 = (\Pi'_3, \mathcal{C}_3, \mathcal{F}_3)$, $\Pi'_1 = WForgetLP(\Pi_1, \{a, c, e\}) = \{d \leftarrow\}$ and $\Pi'_3 = SForgetLP(\Pi_3, \{a\}) = \{d \leftarrow \texttt{not}\, c\}$. We can verify that atom a is $(\Sigma, \Sigma')^i$-irrelevant for all $i = 1, 2, 3$.

On the other hand, by weakly forgetting only $\{c, e\}$ in Π_1, we further obtain a more preferred solution of Σ: $\Sigma'' = (\Phi''_1, \Phi_2, \Phi'_3)$, where $\Phi''_1 = (\Pi''_1, \mathcal{C}_1, \mathcal{F}_1)$, and $\Pi''_1 = WForgetLP(\Pi_1, \{c, e\}) = \{a \leftarrow not\ b, d \leftarrow\}$. In fact, Σ'' is also a preferred solution of Σ. Since $\Sigma' \sim_\Sigma \Sigma''$, according to Corollary 1, we know that a is also $(\Sigma, \Sigma'')^i$-irrelevant $(i = 1, 2, 3)$.

5.2. Characterizing solutions for conflict solving

In this subsection, we focus our study on the semantic characterization on conflict solving solutions, because such characterizations are useful to optimize the procedure of conflict solving in logic program contexts. To begin with, we give a general result for the existence of preferred solutions for arbitrary logic program context.

Theorem 6. *Let Σ be a logic program context. Σ has a preferred solution iff $Solution(\Sigma) \neq \emptyset$.*

Proof. Obviously, if Σ has a preferred solution, then $Solution(\Sigma) \neq \emptyset$. Now we assume that $Solution(\Sigma) \neq \emptyset$. In this case, we only need to show that for each $\Sigma' \in Solution(\Sigma)$, a new solution Σ'' can always be generated from Σ' such that $\Sigma'' \preceq_\Sigma \Sigma'$. If no such solution can be generated from Σ', then Σ' itself is a preferred solution. We present the following algorithm for this purpose.

```
Algorithm: Solution-Generation
Input: $\Sigma = (\Phi_1, \ldots, \Phi_n)$ and $\Sigma' = (\Phi'_1, \ldots, \Phi'_n)$, where
    $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i)$ and $\Phi'_i = (\Pi'_i, \mathcal{C}_i, \mathcal{F}_i)$;
Output: $\Sigma'' = (\Phi''_1, \ldots, \Phi''_n)$;
for $i = 1$ to $n$
    let $\Phi'_i = (\Pi'_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma'$ and
    $\Pi'_i = SForgetLP(\Pi, P)$ or $\Pi'_i = WForgetLP(\Pi, P)$ $(P \subseteq \mathcal{F}_i)$;
    while $Q \subset P$
        testing the consistency of $SForgetLP(\Pi, Q) \cup \mathcal{C}_j$
        for all $j = 1, \ldots, n$;
        if consistency holds, then $\Pi''_i = SForgetLP(\Pi, Q)$;
        if consistency does not hold, then
                testing the consistency of $WForgetLP(\Pi, Q) \cup \mathcal{C}_j$
                for all $j = 1, \ldots, n$;
                if consistency holds, then
                        $\Pi''_i = WForgetLP(\Pi, Q)$, otherwise $\Pi''_i = \Pi'_i$;
return $\Sigma'' = ((\Pi''_1, \mathcal{C}_1, \mathcal{F}_1), \ldots, (\Pi''_n, \mathcal{C}_n, \mathcal{F}_n))$.
```

It is easy to see that algorithm **Solution-Generation** terminates as the procedures of computing *SForgetLP*(Π_i, Q) and *WForgetLP*(Π_i, Q), and consistency testing for a program can always finish in finite steps respectively. Furthermore, the output Σ'' is either the same as Σ' or $\Sigma'' \preceq_\Sigma \Sigma'$. This proves our result. □

The proof of Theorem 5 actually provides a method to generate a preferred solution for a logic program context. That is, once we have an initial solution for a logic program context, we can always generate a more preferred solution from the current one. We continue the process until a preferred solution is finally achieved. However, not every logic program has a solution. For instance, a logic program context $\Sigma = (\Phi_1, \Phi_2) = (\emptyset, \{\leftarrow a, not\ b\}, \emptyset)$, $(\{a \leftarrow not\ b\}, \emptyset, \emptyset))$ has no solution.

Proposition 5. *Let $\Sigma = (\Phi_1, \ldots, \Phi_n)$ be a logic program context. If for each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i)$, Π_i does not contain a constraint rule (a rule with empty head), $\mathcal{C}_i$ is consistent, and for each $r \in \Pi_i$, $atom(r) \cap \mathcal{F}_i \neq \emptyset$, then Solution$(\Sigma) \neq \emptyset$.*

Proof. We show that $\Sigma' = (\Phi'_1, \ldots, \Phi'_n)$, where $\Phi'_i = (\emptyset, \mathcal{C}_i, \mathcal{F}_i)$ $(1 \leqslant i \leqslant n)$ 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 $j = 1, \ldots, n$ are consistent. So $((\emptyset, \mathcal{C}_1, \mathcal{F}_1), \ldots, (\emptyset, \mathcal{C}_n, \mathcal{F}_n))$ is a solution of Σ. □

We should indicate that many conflict solving scenarios can be represented in the type of logic program context in Proposition 5. For example, the negotiation scenario discussed in Example 6 and most logic program update approaches (see Section 6) can be specified under logic program contexts with this form. Therefore, solving conflicts for this particular type of logic program context has a special interest in various applications. This motivates us to study more detailed properties related to the solution of this type of logic program contexts.

We first introduce some useful concepts. A logic program Π's *dependency graph* [1], denoted as $G(\Pi)$, is a directed graph $(atom(\Pi), E)$, where $atom(\Pi)$ is the set of vertices, and E is the set of edges. An edge $(a, b) \in E$ iff there is a rule $r \in \Pi$ such that $a \in pos(r) \cup neg(r)$ and $\{b\} = head(r)$. Edge (a, b) is labelled "positive" if $a \in pos(r)$ and "negative" if $a \in neg(r)$. Then a logic program is called *call-consistent* [12] if it does not contain a constraint (i.e. a rule with empty head) and its dependency graph has no simple cycles with odd number of negative edges.[5]

Lemma 1. *Let Π_1 and Π_2 be two logic programs and Π_1 be consistent. Then program $\Pi_1 \cup \Pi_2$ is consistent if $body(\Pi_1) \cap head(\Pi_2) = \emptyset$ and Π_2 is call-consistent.*

Proof. Similar to the proof of Theorem 3, To prove this lemma, we need a result about program splitting from [26]. To remain a completeness of the proof, we present this result again. Before we present this program splitting result, we introduce a notion. Given a program Π and a set of atoms S, we use $e(\Pi, S)$ to denote the program obtained from Π by deleting: (1) each rule in Π having a form *not a* in its body with $a \in S$; and (2) all atoms a in the bodies of the remaining rules with $a \in S$. Intuitively, $e(\Pi, S)$ can be viewed as a simplicity of Π giving those atoms in S to be true. Then we can re-state Theorem 5 in [26] under the normal logic program setting:

[5] A simple cycle is the one that does not contain any other cycles.

A set of atoms S is a stable model of program Π if and only if $\Pi = \Pi_1 \cup \Pi_2$ such that $body(\Pi_1) \cap head(\Pi_2) = \emptyset$, and $S = S_1 \cup S_2$, where S_1 is a stable model of Π_1 and S_2 is a stable model of program $e(\Pi_2, S_1)$.

From this result, we can see that under the condition that Π_1 is consistent, $\Pi_1 \cup \Pi_2$ is consistent if $body(\Pi_1) \cap head(\Pi_2) = \emptyset$, and for each stable model S_1 of Π_1, $e(\Pi_2, S_1)$ is also consistent.

Since a call-consistent program is also consistent [25], to prove our result, we will prove that if Π_2 is call-consistent, then $e(\Pi_2, S_1)$ is also call-consistent for any set of atoms S_1. From the definition of call-consistency, it is clear that if Π_2 is call-consistent, its dependency graph does not contain a simple cycle with odd number of negative edges. Observing that for any set of atoms S_1, program $e(\Pi_2, S_1)$'s dependency graph $G(e(\Pi_2, S_1))$ can be obtained from $G(\Pi_2)$ by removing more edges and nodes from $G(\Pi_2)$. That is, $G(e(\Pi_2, S_1))$ is a subgraph of $G(\Pi_2)$. This concludes that $G(e(\Pi_2, S_1))$ does not contain a simple cycle with odd number of negative edges. So $e(\Pi_2, S_1)$ is also call-consistent. □

We need to mention that in Lemma 1, the call-consistency condition for program Π_2 is important. It is easy to see that Π_2's consistency does not imply the consistency of $\Pi_1 \cup \Pi_2$ even if the other conditions of Lemma 1 remain the same. For example, consider two programs $\Pi_1 = \{b \leftarrow\}$ and $\Pi_2 = \{a \leftarrow b, not\ a\}$. Both Π_1 and Π_2 are consistent and $body(\Pi_1) \cap head(\Pi_2) = \emptyset$. But $\Pi_1 \cup \Pi_2$ has no stable model. The following theorem states that the procedure of generating a more preferred solution may be simplified under certain conditions.

Theorem 7. *Let $\Sigma = ((\Pi_1, \mathcal{C}_1, \mathcal{F}_1), \ldots, (\Pi_n, \mathcal{C}_n, \mathcal{F}_n))$ be a logic program context satisfying the conditions stated in Proposition 5. Suppose $\Sigma' = ((\Pi'_1, \mathcal{C}_1, \mathcal{F}_1), \ldots, (\Pi'_n, \mathcal{C}_n, \mathcal{F}_n))$ is a solution of Σ, where each Π'_i is of the form SForgetLP(Π_i, P_i) or WForgetLP(Π_i, P_i) $(P_i \subseteq \mathcal{F}_i)$.*[6] *Then a logic program context $\Sigma'' = ((\Pi''_1, \mathcal{C}_1, \mathcal{F}_1), \ldots, (\Pi''_n, \mathcal{C}_n, \mathcal{F}_n))$ is a solution of Σ and $\Sigma'' \preceq_\Sigma \Sigma'$, if for each i either $\Pi''_i = \Pi'_i$, or Π''_i is of the form $\Pi''_i =$ SForgetLP(Π_i, Q_i) or $\Pi''_i =$ WForgetLP(Π_i, Q_i) for some $Q_i \subseteq P_i$ such that $body(\bigcup_{i=1}^n \mathcal{C}_i) \cap head(\Pi''_i) = \emptyset$ and Π''_i is call-consistent.*

Proof. From Lemma 1, it follows that if for each i, $body(\bigcup_{i=1}^n \mathcal{C}_i) \cap head(\Pi''_i) = \emptyset$ and Π''_i is call-consistent, then all programs $\Pi''_i \cup \mathcal{C}_1, \ldots, \Pi''_i \cup \mathcal{C}_n$ are consistent. So Σ'' is a solution of Σ. On the other hand, since for each i, $Q_i \subseteq P_i$, this concludes that $\Sigma'' \preceq_\Sigma \Sigma'$. □

In Theorem 7, the condition that $body(\bigcup_{i=1}^n \mathcal{C}_i) \cap head(\Pi'_i) = \emptyset$ and Π'_i is call-consistent ensures that Σ' is a solution of Σ, while the minimal subset P_i of $atom(\Pi_i)$ implies that Σ' is a preferred solution. The following Example 9 illustrates how a preferred solution can be obtained under the condition of Theorem 7.

Example 9. Consider a logic program context $\Sigma = (\Phi_1, \Phi_2, \Phi_3)$, where

Φ_1:	Φ_2:	Φ_3:
Π_1: $a \leftarrow not\ b$,	Π_2: $d \leftarrow$,	Π_3: $a \leftarrow not\ b$,
$c \leftarrow a, not\ d$,	$f \leftarrow not\ b$,	$c \leftarrow not\ b$,
	$e \leftarrow not\ d$,	
$\mathcal{C}_1$: $e \leftarrow d$,	$\mathcal{C}_2$: $\leftarrow a, c$,	$\mathcal{C}_3$: $f \leftarrow d$,
$\mathcal{F}_1 = \{a, b, c, d\}$,	$\mathcal{F}_2 = \{b, d, e, f\}$,	$\mathcal{F}_3 = \{a, b, c\}$.

Clearly, Σ is not conflict free since $\Pi_1 \cup \mathcal{C}_2$, $\Pi_2 \cup \mathcal{C}_1$, $\Pi_2 \cup \mathcal{C}_3$ and $\Pi_3 \cup \mathcal{C}_2$ are not consistent. We can verify that a logic program context $\Sigma_1 = (\Phi'_1, \Phi'_2, \Phi'_3)$ is a solution of Σ, where

$$\Phi'_1 = (\mathit{SForgetLP}(\Pi_1, \{c\}), \mathcal{C}_1, \mathcal{F}_1),$$
$$\Phi'_2 = (\mathit{SForgetLP}(\Pi_2, \{d, e, f\}), \mathcal{C}_2, \mathcal{F}_2),$$
$$\Phi'_3 = (\mathit{WForgetLP}(\Pi_3, \{a\}), \mathcal{C}_3, \mathcal{F}_3).$$

[6] Note that from Proposition 5, a solution of Σ always exists. In the initial case, Π'_i could be $\emptyset$.

Now we consider a program $WForgetLP(\Pi_2, \{d\})$:

$$f \leftarrow not\ b,$$

$$e \leftarrow .$$

Since $\{e, f\} \cap body(\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3) = \emptyset$ and $WForgetLP(\Pi_2, \{d\})$ is call-consistent, according to Theorem 7, we know that $\Sigma_1'' = (\Phi_1', \Phi_2'', \Phi_3')$, where $\Phi_2' = (WForgetLP(\Pi_2, \{d\}), \mathcal{C}_2, \mathcal{F}_2)$ is also a solution of Σ and $\Sigma'' \preceq_\Sigma \Sigma'$. In fact Σ_1'' is a preferred solution of Σ.

6. Representing logic program updates

Logic program updates have been considerably studied in recent years. While similarities and differences among these different approaches have been addressed by many researchers, it is believed that comparing different types of update approaches at some formal level is generally difficult (discussions on this topic are referred to [5,6,15,28]). In this section, we show that four major logic program update approaches can be transformed into the framework of logic program contexts, in which all these update approaches become special cases of conflict solving problems with different types of constraints.

6.1. Representing causal rejection based approach

Eiter et al.'s update approach is based on a principle called *causal rejection* where a sequence of logic program updates is allowed [5]. Let $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$, where $\Pi_1, \ldots, \Pi_1$ are extended logic programs, be an (extended logic program) *update sequence* and $\mathcal{A}$ a set of atoms. We say that $\mathbf{P}$ is *over* $\mathcal{A}$ iff $\mathcal{A}$ represents the set of all atoms occurring in the rules in $\Pi_1, \ldots, \Pi_n$. We use $Lit_{\mathcal{A}}$ to denote the set of all literals whose corresponding atoms are in $\mathcal{A}$. We assume a set $\mathcal{A}^*$ of atoms extending $\mathcal{A}$ by new and pairwise distinct atoms $rej(r)$ and a_i, for each rule r occurring in $\Pi_1, \ldots, \Pi_n$ and each atom $a \in \mathcal{A}$. Then Eiter et al.'s update process is defined by the following two definitions (here we only consider ground extended logic programs in our investigation).

Definition 12. [5] Given an update sequence $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$ over a set of atoms $\mathcal{A}$, the *update program* $\mathbf{P}_\triangleleft = \Pi_1 \triangleleft \cdots \triangleleft \Pi_n$ over $\mathcal{A}^*$ consisting of the following items:

(1) all constraints in $\Pi_1, \ldots, \Pi_n$ (recall that a constraint is a rule with an empty head);
(2) for each r in Π_i $(1 \leqslant i \leqslant n)$:

$$l_i \leftarrow body(r), not\ rej(r) \quad \text{if } head(r) = \{l\};$$

(3) for each $r \in \Pi_{i-1}$ $(2 \leqslant i \leqslant n)$:

$$rej(r) \leftarrow body(r), \neg l_i \quad \text{if } head(r) = \{l\};$$

(4) for each literal l occurring in $\Pi_1 \cup \cdots \cup \Pi_n$:

$$l_{i-1} \leftarrow l_i\ (1 < i \leqslant n), \quad l \leftarrow l_1.$$

A set $S \subseteq Lit_{\mathcal{A}}$ is an *update answer set* of $\mathbf{P}$ iff $S = S' \cap Lit_{\mathcal{A}}$ for some answer set S' of $\mathbf{P}_\triangleleft$.

As an example, consider an update sequence $\mathbf{P} = (\Pi_1, \Pi_2, \Pi_3)$, where Π_1, Π_2 and Π_3 consist of the following rules respectively [5],

Π_1:

r_1: $sleep \leftarrow not\ tv\_on$,

r_2: $night \leftarrow$,

r_3: $tv\_on \leftarrow$,

r_4: $watch\_tv \leftarrow tv\_on$;

Π_2:

r_5: $\neg tv\_on \leftarrow power\_failure$,

r_6: $power\_failure \leftarrow$,

Π_3:

r_7: $\neg power\_failure \leftarrow$.

According to Definition 12, it is easy to see that $\mathbf{P} = (\Pi_1, \Pi_2, \Pi_3)$ has a unique update answer set $S = \{\neg power\_failure, tv\_on, watch\_tv, night\}$, which is consistent with our intuition.

In order to transform this update approach into our framework of logic program context, we first re-formulate this approach in a normal logic program setting. In particular, given an update sequence $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$ over $\mathcal{A}$, we extend the set $\mathcal{A}$ to $\overline{\mathcal{A}}$ by adding atom $\bar{a}$ to $\mathcal{A}$ for each $a \in \mathcal{A}$. Then by replacing each negative atom $\neg a$ occurring in Π_i with $\bar{a}$, and adding constraint $\leftarrow a, \bar{a}$ for each $a \in \mathcal{A}$, we obtain a translated (normal logic program) update sequence $\overline{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$ over $\overline{\mathcal{A}}$.

We also extend set $\overline{\mathcal{A}}$ to $\overline{\mathcal{A}}^*$ by including new atoms $rej(r)$, a_i and $\bar{a}_i$ for each rule r in $\overline{\Pi}_1, \ldots, \overline{\Pi}_n$ and each pair of atoms $a, \bar{a} \in \overline{\mathcal{A}}$. Then following Definition 12, we can obtain the corresponding update program $\overline{\mathbf{P}}_\triangleleft$ which is also a normal logic program. We also call a stable model of $\overline{\mathbf{P}}_\triangleleft$ *update stable model* of $\overline{\mathbf{P}}$.

Proposition 6. *Let $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$ be an update sequence, $\mathbf{P}_\triangleleft$ the update program of $\mathbf{P}$, and $\overline{\mathbf{P}}$ and $\overline{\mathbf{P}}_\triangleleft$ the corresponding translations of $\mathbf{P}$ and $\mathbf{P}_\triangleleft$ respectively as described above. $S \subseteq Lit_{\mathcal{A}}$ is an update answer set of $\mathbf{P}$ iff there is an update stable model $\overline{S}$ of $\overline{\mathbf{P}}$ such that $S = (\overline{S} \cap \mathcal{A}) \cup \{\neg a \mid \bar{a} \in \overline{S}\}$.*[7]

Having Proposition 6, we only need to consider a transformation from a normal logic program update sequence $\overline{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$, where $\overline{\mathbf{P}}$ is translated from an extended logic program update sequence $\mathbf{P}$ as described above, to a conflict solving problem under the framework of logic program contexts.

Definition 13. Let $\overline{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$ $(n > 1)$ be a normal logic program update sequence over $\overline{\mathcal{A}}$. We specify a sequence of logic program contexts $\Omega_{CR} = (\Sigma_1, \ldots, \Sigma_{n-1})$[8] over the set of atoms $\mathcal{B} = \overline{\mathcal{A}}^* \cup \{l_{a_i}, l_{\bar{a}_i} \mid a_i, \bar{a}_i \in \overline{\mathcal{A}}^*, i = 1, \ldots, n\}$ where l_{a_i} and $l_{\bar{a}_i}$ are newly introduced atoms:

(1) $\Sigma_1 = ((\overline{\Pi}^*_1, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$, where
  (a) $\overline{\Pi}^*_1$ consists of the following rules:
    (i) all constraints in $\overline{\Pi}_1, \ldots, \overline{\Pi}_n$;
    (ii) for each $r \in \overline{\Pi}_i$: $a \leftarrow body(r)$ or $\bar{a} \leftarrow body(r)$ $(i = 1, \ldots, n)$, $a_i \leftarrow body(r), not\ l_{\bar{a}_i}$, or $\bar{a}_i \leftarrow body(r), not\ l_{a_i}$ respectively,
    (iii) for each $a, \bar{a}$ in $\overline{\mathcal{A}}$,
      $a_{i-1} \leftarrow a_i, \bar{a}_{i-1} \leftarrow \bar{a}_i \quad (i = 1, \ldots, n)$,
      $a \leftarrow a_1, \bar{a} \leftarrow \bar{a}_1$.
  (b) $\mathcal{F}_1 = \{l_{a_{n-1}}, l_{\bar{a}_{n-1}} \mid \forall a \in \mathcal{A}\}$,
  (c) $\mathcal{C}_1 = \{\leftarrow a_{n-1}, \bar{a}_n, \leftarrow \bar{a}_{n-1}, a_n \mid \forall a \in \mathcal{A}\}$;
(2) $\Sigma_i = ((\overline{\Pi}^*_i, \emptyset, \mathcal{F}_i), (\emptyset, \mathcal{C}_i, \emptyset))$ $(i = 1, \ldots, n)$, where
  (a) $\overline{\Pi}^*_i = \overline{\Pi}^\dagger_{i-1}$, and $\overline{\Pi}^\dagger_{i-1}$ is in a preferred solution of Σ_{i-1}:
    $\Sigma'_{i-1} = ((\overline{\Pi}^\dagger_{i-1}, \emptyset, \mathcal{F}_{i-1}), (\emptyset, \mathcal{C}_{i-1}, \emptyset))$,
  (b) $\mathcal{F}_i = \{l_{a_{n-i}}, l_{\bar{a}_{n-i}} \mid \forall a \in \mathcal{A}\}$,
  (c) $\mathcal{C}_i = \{\leftarrow a_{n-i}, \bar{a}_{n-i+1}, \leftarrow \bar{a}_{n-i}, a_{n-i+1} \mid \forall a \in \mathcal{A}\}$.

[7] Note that S is reduced to $Lit_{\mathcal{A}}$ if both a and $\neg a$ are in S for some $a \in \mathcal{A}$.

[8] Note that when $n = 1$ our transformation becomes trivial since we can simply specify Ω_{CR} to consist of a single logic program context $\Sigma = ((\overline{\Pi}_1, \emptyset, \emptyset), (\emptyset, \emptyset, \emptyset))$. In this case Σ has a (preferred) solution iff $\overline{\Pi}_1$ is consistent. So in the rest of the paper we will only consider the case $n > 1$.

A subset $S \subseteq \mathcal{B}$ is called a *model* of Ω_{CR} if S is a stable model of $\overline{\Pi}^{\dagger}_{n-1}$, where $\overline{\Pi}^{\dagger}_{n-1}$ is in a preferred solution of Σ_{n-1}: $\Sigma'_{n-1} = ((\overline{\Pi}^{\dagger}_{n-1}, \emptyset, \mathcal{F}_{n-1}), (\emptyset, \mathcal{C}_{n-1}, \emptyset))$.

Let us take a closer look at Definition 13. Given an update sequence $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$, Definition 13 specifies a sequence of logic program contexts $\Omega_{CR} = (\Sigma_1, \ldots, \Sigma_{n-1})$, where each Σ_i solves certain conflicts embedded in $\bar{\mathbf{P}}$. Σ_1 represents the first level of conflict solving, where $\overline{\Pi}^*_1$ is similar to $\bar{\mathbf{P}}_{\lhd}$ except that the possible conflict between a_{n-1} and $\bar{a}_n$ (or $\bar{a}_{n-1}$ and a_n) has been reformulated as a constraint $\leftarrow a_{n-1}, \bar{a}_n$ (or $\leftarrow \bar{a}_{n-1}, a_n$ resp.) in $\mathcal{C}_1$. Note that in rules specified in (ii) of Definition 13: $a_i \leftarrow body(r), not\ l_{\bar{a}_i}$, $\bar{a}_i \leftarrow body(r), not\ l_{a_i}$, formulas $not\ l_{\bar{a}_{n-1}}$ and $not\ l_{a_{n-1}}$ (here $i = n-1$) are introduced to solve the conflict between a_{n-1} and $\bar{a}_n$ (or $\bar{a}_{n-1}$ and a_n resp.).

Observe that Σ_1 *only* solves conflicts between atoms at level $n-1$. For example, if both a_{n-1} and $\bar{a}_n$ can be derived from $\overline{\Pi}^*_1$, then rule $a_{n-1} \leftarrow body(r), not\ l_{\bar{a}_{n-1}}$ will be eliminated from Π_1 by strongly forgetting atom $l_{\bar{a}_{n-1}}$ under the constraint $\leftarrow a_{n-1}, \bar{a}_n$ in $\mathcal{C}_1$.

In the sequence $\Omega_{CR} = (\Sigma_1, \ldots, \Sigma_{n-1})$, conflicts are solved in a *downwards* manner with respect to the update sequence $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$, where each Σ_i ($i > 1$) is specified for the purpose of solving conflicts between atoms a_{n-i} and $\bar{a}_{n-i+1}$ (or $\bar{a}_{n-i}$ and a_{n-i+1}).

Example 10. Consider the TV example mentioned earlier, where $\mathbf{P} = (\Pi_1, \Pi_2, \Pi_3)$ is an update sequence. It is easy to translate $\mathbf{P}$ to the corresponding normal logic program update sequence $\bar{\mathbf{P}} = (\overline{\Pi}_1, \overline{\Pi}_2, \overline{\Pi}_3)$, where $\neg tv\_on$ and $\neg power\_failure$ are replaced by atoms $\overline{tv\_on}$ and $\overline{power\_failure}$ respectively. According to Definition 13, we then specify a sequence of logic program contexts $\Omega_{CR} = (\Sigma_1, \Sigma_2)$ to solve the conflict occurring in $\bar{\mathbf{P}}$. $\Sigma_1 = ((\overline{\Pi}^*_1, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$, where $\overline{\Pi}^*_1$ consists of the following rules[9]:

$$
\begin{aligned}
& sleep_1 \leftarrow not\ tv\_on, not\ l_{\overline{sleep_1}}, \\
& night_1 \leftarrow not\ l_{\overline{night_1}}, \\
& tv\_on_1 \leftarrow not\ l_{\overline{tv\_on_1}}, \\
& watch\_tv_1 \leftarrow tv\_on, not\ l_{\overline{watch\_tv_1}}, \\
& \overline{tv\_on}_2 \leftarrow power\_failure, not\ l_{tv\_on_2}, \\
& power\_failure_2 \leftarrow not\ l_{\overline{power\_failure_2}}, \\
& \overline{power\_failure}_3 \leftarrow not\ l_{power\_failure_3}, \\
& night \leftarrow night_1, \\
& tv\_on \leftarrow tv\_on_1, \\
& watch\_tv \leftarrow watch\_tv_1, \\
& \overline{tv\_on}_1 \leftarrow \overline{tv\_on}_2, \\
& \overline{tv\_on} \leftarrow \overline{tv\_on}_1, \\
& \overline{power\_failure}_2 \leftarrow \overline{power\_failure}_3, \\
& \overline{power\_failure}_1 \leftarrow \overline{power\_failure}_2, \\
& \overline{power\_failure} \leftarrow \overline{power\_failure}_1, \\
& power\_failure_1 \leftarrow power\_failure_2, \\
& power\_failure \leftarrow power\_failure_1, \\
& \mathcal{F}_1 = \{l_{power\_failure_2}, l_{\overline{power\_failure_2}}\}, \quad \text{and} \\
& \mathcal{C}_1 = \{\leftarrow power\_failure_2, \overline{power\_failure}_3\}.
\end{aligned}
$$

[9] To avoid unnecessarily tedious details, here we omit some irrelevant rules and atoms from $\overline{\Pi}^*_1$, $\mathcal{F}_1$ and $\mathcal{C}_1$. The same for Σ_2.

It is easy to see that Σ_1 is not conflict free since $\overline{\Pi}_1^* \cup \mathcal{C}_1$ is not consistent (i.e it has no stable model). To specify Σ_2, we first need to obtain a preferred solution of Σ_1. In fact Σ_1 has a unique preferred solution $\Sigma_1' = ((\overline{\Pi}_1^\dagger, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$, where

$$\overline{\Pi}_1^\dagger = \mathit{SForgetLP}\big(\overline{\Pi}_1^*, \{l_{\overline{power\_failure_2}}\}\big) = \overline{\Pi}_1^* - \big\{power\_failure_2 \leftarrow not\ l_{\overline{power\_failure_2}}\big\}.$$

Now we specify $\Sigma_2 = ((\overline{\Pi}_1^\dagger, \emptyset, \mathcal{F}_2), (\emptyset, \mathcal{C}_2, \emptyset))$, where $\mathcal{F}_2 = \{l_{tv\_on_1}, l_{\overline{tv\_on}_1}\}$ and $\mathcal{C}_2 = \{\leftarrow tv\_on_1, \overline{tv\_on}_2\}$. Note that Σ_2 is already conflict free. So by ignoring those atoms with subscripts, Ω_{CR} has a unique model $\{\overline{power\_failure}, tv\_on, watch\_tv, night\}$, which is the same as the update stable model of update sequence $\bar{\mathbf{P}}$.

Theorem 8. *Let $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$ $(n > 1)$ be a normal logic program update sequence over the set of atoms $\overline{\mathcal{A}}$. A subset $\bar{S}$ of $\overline{\mathcal{A}}$ is an update stable model of $\bar{\mathbf{P}}$ iff there is a sequence of logic program contexts $\Omega_{CR} = (\Sigma_1, \ldots, \Sigma_{n-1})$ constructed from $\bar{\mathbf{P}}$ as specified in Definition* 13 *such that Ω_{CR} has a model S satisfying $\bar{S} = S \cap \overline{\mathcal{A}}$.*

Proof. We prove this result by induction on the length n of normal logic program update sequence $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$.

Case 1. We first consider the case $n = 2$, i.e. $\bar{\mathbf{P}} = (\overline{\Pi}_1, \overline{\Pi}_2)$. In this case, $\Omega_{CR} = (\Sigma_1)$, where $\Sigma_1 = ((\overline{\Pi}_1^*, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$ is formed as follows:

(a) $\overline{\Pi}_1^*$ consists of the following rules:
  (i) all constraints in $\overline{\Pi}_1$ and $\overline{\Pi}_2$;
  (ii) for each $r \in \overline{\Pi}_i$: $a \leftarrow body(r)$ or $\bar{a} \leftarrow body(r)$ $(i = 1, 2)$,
  $a_i \leftarrow body(r), not\ l_{\bar{a}_i}$, or $\bar{a}_i \leftarrow body(r), not\ l_{a_i}$ respectively,
  (iii) for each $a, \bar{a}$ in $\overline{\mathcal{A}}, \ldots, \overline{\Pi}_n$,
  $a_1 \leftarrow a_2, \bar{a}_1 \leftarrow \bar{a}_2$,
  $a \leftarrow a_1, \bar{a} \leftarrow \bar{a}_1$.
(b) $\mathcal{F}_1 = \{l_{a_1}, l_{\bar{a}_1} \mid \forall a \in \mathcal{A}\}$,
(c) $\mathcal{C}_1 = \{\leftarrow a_1, \bar{a}_2, \leftarrow \bar{a}_1, a_2 \mid \forall a \in \mathcal{A}\}$.

Note that in above (ii), for rule $r \in \overline{\Pi}_2$, $a_2 \leftarrow body(r), not\ l_{\bar{a}_2}$, or $\bar{a}_2 \leftarrow body(r), not\ l_{a_2}$ can be simplified as $a_2 \leftarrow body(r)$, or $\bar{a}_2 \leftarrow body(r)$ respectively since atom $l_{\bar{a}_2}$ or l_{a_2} is not forgettable.

Now we consider the update program $\bar{\mathbf{P}}_\triangleleft$ built upon $\bar{\mathbf{P}}$ (see Definition 12), which consists of the following rules:

(1) all constraints in $\overline{\Pi}_1$ and $\overline{\Pi}_2$;
(2) $a_1 \leftarrow body(r), not\ rej(r)$ or $\bar{a}_1 \leftarrow body(r), not\ rej(r)$ for $r \in \overline{\Pi}_1$, and $a_2 \leftarrow body(r)$ or $\bar{a}_2 \leftarrow body(r)$ for $r \in \overline{\Pi}_2$;
(3) $rej(r) \leftarrow body(r), \bar{a}_2$ if $head(r) = \{a_1\}$ or $rej(r) \leftarrow body(r), a_2$ if $head(r) = \{\bar{a}_1\}$ for $r \in \overline{\Pi}_1$;
(4) for all $a \in \overline{\mathcal{A}}$, $a_1 \leftarrow a_2, \bar{a}_1 \leftarrow \bar{a}_2, a \leftarrow a_1, \bar{a} \leftarrow \bar{a}_1$.

Now suppose $\bar{S}$ is an update stable model of $\bar{\mathbf{P}}$. Then we can extend $\bar{S}$ to $\bar{S}^*$ over set $\overline{\mathcal{A}}^*$ so that $\bar{S}^*$ is a stable model of program $\bar{\mathbf{P}}_\triangleleft$, which contains atoms $rej(r)$ for some $r \in \overline{\Pi}_1$. Note that those rules in item (2) above with $rej(r) \in \bar{S}^*$ actually play no roles and hence viewed as been removed from $\bar{\mathbf{P}}$. Then we specify a set $P \subseteq \mathcal{F}_1$ which includes those l_{a_1} or $l_{\bar{a}_1}$ whose corresponding rules $r \in \overline{\Pi}_1$ in (ii) are removed from $\bar{\mathbf{P}}$ as indicated above. Then it can be verified that S where $\bar{S} = S \cap \overline{\mathcal{A}}$ must be a stable model of program $\mathit{SForgetLP}(\overline{\Pi}_1^*, P)$, and P is a minimal such set to make $\mathit{SForgetLP}(\overline{\Pi}_1^*, P)$ consistent. That is, S is a model of Ω_{CR}.

On the other hand, consider a stable model S of $\mathit{SForgetLP}(\overline{\Pi}_1^*, P)$, where $\mathit{SForgetLP}(\overline{\Pi}_1^*, P)$ is in a preferred solution of Σ_1. Let $\bar{S} = S \cap \overline{\mathcal{A}}$. Similarly, for each l_{a_1} or $l_{\bar{a}_1}$ in P, we extend $\bar{S}$ to $\bar{S}^*$ to contain atoms $rej(r)$ in $\bar{S}^*$. Note that for each $rej(r)$, such $r \in \overline{\Pi}_1$ corresponds to $a_1 \leftarrow body(r), not\ l_{\bar{a}_1}$ or $\overline{a_1} \leftarrow body(r), not\ l_{a_1}$ in (ii) specified above. Now we do a Gelfond–Lifschitz transformation on program $\overline{\mathcal{P}}_\triangleleft$ in terms of set $\bar{S}^*$: $\overline{\mathcal{P}}_\triangleleft^{\bar{S}^*}$. By avoiding tedious checkings, we can show that $\bar{S}^*$ is a stable model of $\overline{\mathcal{P}}_\triangleleft^{\bar{S}^*}$.

Case 2. Suppose for all $n < k$, $\bar{S}$ is an update stable model of $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$ iff there is a $\Omega_{CR} = (\Sigma_1, \ldots, \Sigma_{n-1})$ such that Ω_{CR} has a model S satisfying $\bar{S} = S \cap \overline{\mathcal{A}}$. Now we consider the case of $n = k$.

($\Rightarrow$) Let $\bar{S}$ be an update stable model of $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_k)$. We will show that we can generate a sequence of logic program contexts Ω_{CR} with length of $k - 1$ such that Ω_{CR} has a model S satisfying $\bar{S} = S \cap \overline{\mathcal{A}}$.

We first specify a new normal logic program update sequence with length of $k-1$: $\bar{\mathbf{P}}' = (\overline{\Pi}_1, \ldots, \overline{\Pi}'_{k-1})$, where $\overline{\Pi}'_{k-1} = \overline{\Pi}^*_{k-1} \cup \overline{\Pi}_k$, and $\overline{\Pi}^*_{k-1} = \overline{\Pi}_{k-1} - \{r \mid rej(r) \in \bar{S}^*\}$.[10] Then from Definition 12, we can see that $\bar{S}$ is also an update stable model of $\bar{\mathbf{P}}'$. Now suppose $\Omega'_{CR} = (\Sigma_1, \ldots, \Sigma_{k-2})$ is a sequence of logic program contexts constructed from $\bar{\mathbf{P}}'$ according to Definition 13. From the induction assumption, we know that Ω'_{CR} has a model S satisfying $\bar{S} = S \cap \overline{\mathcal{A}}$.

Now we show that $\Omega'_{CR} = (\Sigma_1, \ldots, \Sigma_{k-2})$ actually can be extended to another $\Omega_{CR} = (\Sigma'_1, \Sigma_1, \ldots, \Sigma_{k-2})$ with a length of $k-1$, which eventually is constructed from $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_k)$.

Observe $\overline{\Pi}'_{k-1}$ in $\bar{\mathbf{P}}'$, we can see that those a_{k-1} or $\bar{a}_{k-1}$ cannot be derived if $\bar{a}_k$ or a_k is already presented in $\bar{S}$. That is, no conflict between a_{k-1} and $\bar{a}_k$ (or $\bar{a}_{k-1}$ and a_k) exists in $\overline{\Pi}'_{k-1}$. So the first logic program context Σ_1 in Ω'_{CR} is specified as $\Sigma_1 = ((\overline{\Pi}^*_1, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$:

(1) $\overline{\Pi}^*_1$ consists of the following rules:
   (a) all constraints in $\overline{\Pi}_1, \ldots, \overline{\Pi}'_{k-1}$;
   (b) for each r: $a \leftarrow body(r)$ or $\bar{a} \leftarrow body(r)$ in $\overline{\Pi}_i$ $(i = 1, \ldots, k-2)$ or in $\overline{\Pi}'_{k-1}$: $a_i \leftarrow body(r), not\ l_{\bar{a}_i}$, or $\bar{a}_i \leftarrow body(r), not\ l_{a_i}$ respectively,
   (c) for each $a, \bar{a}$ in $\overline{\mathcal{A}}$, $a_{i-1} \leftarrow a_i$, $\bar{a}_{i-1} \leftarrow \bar{a}_i$ $(i = 1, \ldots, n)$,
   $a \leftarrow a_1$, $\bar{a} \leftarrow \bar{a}_1$.
(2) $\mathcal{F}_1 = \{l_{a_{k-2}}, l_{\bar{a}_{k-2}} \mid \forall a \in \mathcal{A}\}$,
(3) $\mathcal{C}'_1 = \{\leftarrow a_{k-2}, \bar{a}_{k-1}, \leftarrow \bar{a}_{k-2}, a_{k-1} \mid \forall a \in \mathcal{A}\}$.

Thus, we can view Σ_1 in Ω'_{CR} represents a preferred solution of logic program context $\Sigma'_1 = ((\overline{\Pi}^{*'}_1, \emptyset, \mathcal{F}'_1), (\emptyset, \mathcal{C}'_1, \emptyset))$,[11] where

(1) $\overline{\Pi}^{*'}_1$ consists of the following rules:
   (a) all constraints in $\overline{\Pi}_1, \ldots, \overline{\Pi}_k$;
   (b) for each $r \in \overline{\Pi}_i$: $a \leftarrow body(r)$ or $\bar{a} \leftarrow body(r)$ $(i = 1, \ldots, k)$, $a_i \leftarrow body(r), not\ l_{\bar{a}_i}$, or $\bar{a}_i \leftarrow body(r), not\ l_{a_i}$ respectively,
   (c) for each $a, \bar{a}$ in $\overline{\mathcal{A}}$,
   $a_{i-1} \leftarrow a_i$, $\bar{a}_{i-1} \leftarrow \bar{a}_i$ $(i = 1, \ldots, n)$,
   $a \leftarrow a_1$, $\bar{a} \leftarrow \bar{a}_1$.
(2) $\mathcal{F}'_1 = \{l_{a_{k-1}}, l_{\bar{a}_{k-1}} \mid \forall a \in \mathcal{A}\}$,
(3) $\mathcal{C}'_1 = \{\leftarrow a_{k-1}, \bar{a}_k, \leftarrow \bar{a}_{k-1}, a_k \mid \forall a \in \mathcal{A}\}$.

Now we form a new $\Omega_{CR} = (\Sigma'_1, \Sigma_1, \ldots, \Sigma_{k-2})$. Obviously S is model of Ω_{CR} iff S is a model of Ω'_{CR}. On the other hand, According to Definition 13, it turns out that Ω_{CR} can be viewed as such a sequence of logic program contexts formed from $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_k)$.

($\Leftarrow$) Given $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_k)$ and $\Omega_{CR} = (\Sigma_1, \ldots, \Sigma_{k-1})$ which is specified as in Definition 13. Suppose S is a model of Ω_{CR}. We show that $S \cap \overline{\mathcal{A}}$ is an update stable model of $\bar{\mathbf{P}}$. Now we consider a subsequence of $\Omega'_{CR} = (\Sigma_2, \ldots, \Sigma_{k-1})$, where $\Sigma_2 = ((\overline{\Pi}^*_2, \emptyset, \mathcal{F}_2), (\emptyset, \mathcal{C}_2, \emptyset))$, which is a preferred solution of Σ_1 in Ω_{CR}. So we can represent $\overline{\Pi}_2 = SForgetLP(\overline{\Pi}^*_1, P)$, where $P \subseteq \mathcal{F}_1 = \{l_{a_{k-1}}, l_{\bar{a}_{k-1}} \mid \forall a \in \mathcal{A}\}$, and $\overline{\Pi}^*_1$ is in Σ_1. Now we define a program based on $\bar{\mathbf{P}}_{\triangleleft}$:

$$\bar{\mathbf{P}}'_{\triangleleft} = \bar{\mathbf{P}}_{\triangleleft} - \big(\{r\colon a_{k-1} \leftarrow body(r), not\ rej(r) \mid l_{a_{k-1}} \in P\} \cup \{r\colon a_{k-1} \leftarrow body(r), not\ rej(r) \mid l_{\bar{a}_{k-1}} \in P\}\big).$$

Equivalently, we can view $\bar{\mathbf{P}}'_{\triangleleft}$ as the update program of a new sequence $\bar{\mathbf{P}}' = (\overline{\Pi}_1, \ldots, \overline{\Pi}^*_{k-1})$ where $\overline{\Pi}^*_{k-1} = \overline{\Pi}'_{k-1} \cup \overline{\Pi}_k$, and $\overline{\Pi}'_{k-1} = \overline{\Pi}_{k-1} - \{r \mid$ those corresponding rules removed in $\bar{\mathbf{P}}'_{\triangleleft}\}$. Also, it is easy to verify that Ω'_{CR} can be

[10] Here we denote $\bar{S}^*$ to be the extension of $\bar{S}$ containing atoms from $\overline{\mathcal{A}}^*$.

[11] Note the difference between $\overline{\Pi}^*_1$ and $\overline{\Pi}^{*'}_1$.

generated from $\overline{\Pi}'_{k-1}$ following Definition 13. According to the induction assumption, we know that $S \cap \overline{\mathcal{A}}$ is an update stable model of $\bar{\mathbf{P}}'$.

On the other hand, since $\bar{S} = S \cap \overline{\mathcal{A}}$ is an update stable model of $\bar{\mathbf{P}}'$, we can extend $\bar{S}$ to $\bar{A}^*$ containing those atoms in $\overline{\mathcal{A}}^*$. Therefore, for each rule r: $a_{k-1} \leftarrow body(r), not\ rej(r)$ or r: $a_{k-1} \leftarrow body(r), not\ rej(r)$ removed from $\bar{\mathbf{P}}_{\triangleleft}$ (see the definition for $\bar{\mathbf{P}}'_{\triangleleft}$ above), atom $rej(r)$ should be in $\bar{A}^*$. Otherwise, this will violate the induction assumption. This follows that $\bar{S}$ must be an update model for $\bar{\mathbf{P}}$ too. This completes our proof. □

6.2. *Representing dynamic logic program approach*

Logic program update based on dynamic logic programs (DLP) (or simply called DLP update approach) was proposed by Alferes, Leite, Pereira, et al. [2], and then extended for various purposes [15]. DLP deals with *generalized logic programs* in which negation as failure *not* is allowed to occur in the head of a rule while classical negation $\neg$ is excluded from the entire program. Let $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$ be a sequence of generalized logic programs over set of atoms $\mathcal{A}$, we extend $\mathcal{A}$ to $\mathcal{A}_D$ by adding pairwise distinct atoms $\bar{a}, a_i, \bar{a}_i, a_{P_i}, \bar{a}_{P_i}$, for each $a \in \mathcal{A}$.

Definition 14. [15] Given a update sequence $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$ over $\mathcal{A}$, where each Π_i is a generalized logic program, the corresponding *dynamic update program* $\mathbf{P}_\oplus = \Pi_1 \oplus \cdots \oplus \Pi_n$ over $\mathcal{A}_D$ is a generalized logic program consisting of the following rules:

(1) for each $r \in \Pi_i$: $head(r) \leftarrow pos(r), not\ neg(r)$,
  $a_{P_i} \leftarrow pos(r), not\ neg(r)$ if $head(r) = \{a\}$ or
  $\bar{a}_{P_i} \leftarrow pos(r), not\ neg(r)$, if $head(r) = \{not\ a\}$;
(2) for each a occurring $\mathbf{P}$ and each $i = 1, \ldots, n$,
  $a_i \leftarrow a_{P_i}$ and $\bar{a} \leftarrow \bar{a}_{P_i}$;
(3) for each a occurring $\mathbf{P}$ and each $i = 1, \ldots, n$,
  $a_i \leftarrow a_{i-1}, not\ \bar{a}_{P_i}$,
  $\bar{a}_i \leftarrow \bar{a}_{i-1}, not\ a_{P_i}$;
(4) for each a occurring $\mathbf{P}$, $\bar{a}_0 \leftarrow$, $a \leftarrow a_n$, $\bar{a} \leftarrow \bar{a}_n$, $not\ a \leftarrow \bar{a}_n$.

The semantics of DLP is defined in terms of the dynamic stable model semantics [15]. However, it is easy to characterize this through the original stable model semantics.

Proposition 7. *Given a dynamic update program* $\mathbf{P}_\oplus = \Pi_1 \oplus \cdots \oplus \Pi_n$, *we define* $\mathbf{P}^*_\oplus = \mathbf{P}_\oplus - \{not\ a \leftarrow \bar{a}_n \mid a \in \mathcal{A}\}$.[12] *Then S is a dynamic stable model of* $\mathbf{P}_\oplus$ *iff* $S = S' \cup \{not\ a \mid \bar{a}_n \in S'\}$, *where S' is a stable model of* $\mathbf{P}^*_\oplus$.

Now we can represent a transformation from $\mathbf{P}^*_\oplus$ to a sequence of logic program contexts which captures the dynamic logic programming update approach.

Definition 15. Given a dynamic update program $\mathbf{P}_\oplus = \Pi_1 \oplus \cdots \oplus \Pi_n$ over $\mathcal{A}_D$ (see Definition 14), and let $\mathbf{P}^*_\oplus = \mathbf{P}_\oplus - \{not\ a \leftarrow a_n^- \mid a \in \mathcal{A}\}$. We specify a sequence of logic program contexts $\Omega_{DLP} = (\Sigma_1, \ldots, \Sigma_n)$ over the set of atoms $\mathcal{A}^*_D = \mathcal{A}_D \cup \{h_{a_i}, h_{\bar{a}_i}, l_{a_i}, l_{\bar{a}_i} \mid a_i, \bar{a}_i \in \mathcal{A}_D, i = 0, \ldots, n\}$ where $h_{a_i}, h_{\bar{a}_i}, l_{a_i}, l_{\bar{a}_i}$ are newly introduced atoms:

(1) $\Sigma_1 = ((\Pi^*_1, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$, where
  (a) Π^*_1 consists of the following rules:
    (i) all rules in $\mathbf{P}^*_\oplus$ except the following rules ($i = 1, \ldots, n$):
      $a_i \leftarrow a_{i-1}, not\ \bar{a}_{p_i}$, and
      $\bar{a}_i \leftarrow \bar{a}_{i-1}, not\ a_{p_i}$,
    (ii) for each pair of rules in $\mathbf{P}^*_\oplus$ ($i = 1, \ldots, n$):
      $a_i \leftarrow a_{i-1}, not\ \bar{a}_{p_i}$, and

[12] Clearly, $\mathbf{P}^*_\oplus$ is a normal logic program.

$\bar{a}_i \leftarrow \bar{a}_{i-1}, not\ a_{p_i}$,
replace them with the following rules in Π_1^*:
$a_i \leftarrow a_{i-1}, not\ l_{a_i}, \bar{a}_i \leftarrow \bar{a}_{i-1}, not\ l_{\bar{a}_i}$,
$l_{a_i} \leftarrow not\ h_{a_i}, l_{\bar{a}_i} \leftarrow not\ h_{\bar{a}_i}$,
$h_{a_i} \leftarrow \bar{a}_{P_i}, h_{\bar{a}_i} \leftarrow a_{P_i}$,

(b) $\mathcal{F}_1 = \{h_{a_1}, h_{\bar{a}_1} \mid \forall a \in \mathcal{A}\}$,
(c) $\mathcal{C}_1 = \{\leftarrow a_1, \bar{a}_{P_1}, \leftarrow \bar{a}_1, a_{P_1} \mid \forall a \in \mathcal{A}\}$;

(2) $\Sigma_i = ((\Pi_i^*, \emptyset, \mathcal{F}_i), (\emptyset, \mathcal{C}_i, \emptyset))$, where
(a) $\Pi_i^* = \Pi_{i-1}^\dagger$, and $\Pi_{i-1}^\dagger$ is in a preferred solution of Σ_{i-1}:
$\Sigma_{i-1}' = ((\Pi_{i-1}^\dagger, \emptyset, \mathcal{F}_{i-1}), (\emptyset, \mathcal{C}_{i-1}, \emptyset))$,
(b) $\mathcal{F}_i = \{h_{a_i}, h_{\bar{a}_i} \mid \forall a \in \mathcal{A}\}$,
(c) $\mathcal{C}_i = \{\leftarrow a_i, \bar{a}_{P_i}, \leftarrow \bar{a}_i, a_{P_i} \mid \forall a \in \mathcal{A}\}$.

A subset $S \subseteq \mathcal{A}_D^*$ is called a *model* of Ω_{DLP} if S is a stable model of $\Pi_n^\dagger$, where $\Pi_n^\dagger$ is in a preferred solution of Σ_n: $\Sigma_n' = ((\Pi_n^\dagger, \emptyset, \mathcal{F}_n), (\emptyset, \mathcal{C}_n, \emptyset))$.

In Definition 15, the sequence of logic program contexts $\Omega_{DLP} = (\Sigma_1, \ldots, \Sigma_n)$ represents a way of solving conflicts between atoms in an *upwards* manner. Starting from $i = 1$, for each i Σ_i solves conflicts between atoms a_i and $\bar{a}_{P_i}$ (or $\bar{a}_i$ and a_{P_i} resp.) through *weakly* forgetting h_{a_i} or $h_{\bar{a}_i}$. For instance, if both a_{i-1} and $\bar{a}_{P_i}$ are derived from Π_i^*, then both a_i and $\bar{a}_i$ can be derived from Π_i^* as well. Therefore a conflict would occur. Σ_i solves such conflict by weakly forgetting h_{a_i}. In particular, after weakly forgetting h_{a_i}, rule $h_{a_i} \leftarrow \bar{a}_{P_i}$ in Π_i^* will be removed, atom l_{a_i} is then derived from $l_{a_i} \leftarrow$ (note that formula *not* h_{a_i} is deleted from rule $l_{a_i} \leftarrow not\ h_{a_i}$). Consequently rule $a_i \leftarrow a_{i-1}, not\ l_{a_i}$ is defeated so that atom a_i cannot be derived from a_{i-1} via the corresponding inertia rule. This process continuous until all conflicts among atoms from level 1 to level n are solved.

Theorem 9. *Let $\mathbf{P}_\oplus^*$ be specified as above over set of atoms $\mathcal{A}_D$. A subset $S^* \subseteq \mathcal{A}_D$ is a stable model of $\mathbf{P}_\oplus^*$ iff there is a sequence of logic program contexts $\Omega_{DLP} = (\Sigma_1, \ldots, \Sigma_n)$ constructed from $\mathbf{P}_\oplus^*$ as specified in Definition* 15 *such that Ω_{DLP} has a model S satisfying $S^* = S \cap \mathcal{A}_D$.*

Since the proof for this theorem is tedious but similar to the proof of Theorem 8, we skip it here.

6.3. Representing syntax based approach

Sakama and Inoue's update approach is viewed as a typical syntax based logic program update approach [22], which solves conflicts between two programs on a basis of syntactic coherence.

To simplify our discussion, we restrict Sakama and Inoue's approach from an extended logic program setting to a normal logic program setting. Note that this restriction does not affect the result presented in this subsection. In fact, we may use the method described in last subsection to translate an extended logic program update into a normal logic program update by introducing new atoms in the underlying language.

Definition 16. [22] 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': $head(r) \leftarrow pos(r), not\ (neg(r) \cup \{l^r\})$ 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 Π^{-notP} 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 $\Pi = \{a \leftarrow b, not\ c, not\ d\}$, then $\Pi^{-not\{c\}} = \{a \leftarrow b, not\ d\}$. Now we have the following characterization result.

Theorem 10. *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 $\Pi' = \Pi^{-not\{l^r \mid r \in \Pi_1\}} \cup \Pi_2$, where $\Sigma' = ((\Pi, \emptyset, \{l^r \mid r \in \Pi_1\}), (\emptyset, \Pi_2, \emptyset))$ is a preferred solution of Σ_{SI}.*

Proof. From the specifications of Σ_{SI} and Σ', we know that $\Pi = SForgetLP(\Pi', P)$, where P is a minimal subset of $\{l^r \mid r \in \Pi_1\}$ such that $\Pi \cup \Pi_2$ is consistent. Note that each rule $r \in \Pi$ is of the form: $head(r) \leftarrow pos(r), not(neg(r) \cup \{l^r\})$, which can actually be simplied as $head(r) \leftarrow pos(r), not\ neg(r)$ since atom l^r does not play any role in the program evaluation. That is, $\Pi' \cup \Pi_2$ is equivalent to $\Pi^{-not\{l^r \mid r \in \Pi_1\}} \cup \Pi_2$, which is a *SI*-result of the update of Π_1 with Π_2. □

6.4. Representing integrated update approach

Different from both model based and syntax based approaches, Zhang and Foo's update approach integrated both desirable semantic and syntactic features of (extended) logic program updates [27]. Their approach also solves default conflicts caused by negation as failure in logic programs by using a prioritized logic programming language. Consequently, Zhang and Foo's update approach can generate an explicit resulting program for a logic program update and also avoid some undesirable solutions embedded in Sakama–Inoue's approach [28].

Since we do not consider default conflict solving in this paper, we will only focus on the transformation from first part of Zhang–Foo's update approach, that is, the conflict (contradiction) elimination, into a logic program context.

Let Π_1 and Π_2 be two extended logic programs. Updating Π_1 with Π_2 consists of two stages. Step (1): Simple fact update—updating an answer set S of Π_1 by program Π_2. The result of this update is a collection of sets of literals, denoted as $Update(S, \Pi_2)$. Step (2): Select a $S' \in Update(S, \Pi_2)$, and extract a maximal subset Π^* of Π_1 such that program $\Pi^* \cup \{l \leftarrow \mid l \in S'\}$ (or simply represented as $\Pi^* \cup S'$) is consistent. Then $\Pi^* \cup \Pi_2$ is called a *resulting program* of updating Π_1 with Π_2.

Note that in Step (1), the simple fact update is achieved through a prioritized logic programming [27]. Recently, Zhang proved an equivalence relationship between the simple fact update and Sakama and Inoue's program update [28]:

$$Update(S, \Pi_2) = \bigcup \mathcal{S}\big(SI\text{-}Update\big(\Pi(S), \Pi_2\big)\big),$$

where $\Pi(S) = \{l \leftarrow \mid l \in S\}$, and $\bigcup \mathcal{S}(SI\text{-}Update(\Pi(S), \Pi_2))$ is the class of all answer sets of resulting programs after updating $\Pi(S)$ by Π_2 using Sakama–Inoue's approach.

Example 11. Consider two extended logic programs Π_1 and Π_2 as follows:

Π_1:	Π_2:
$a \leftarrow,$	$b \leftarrow a,$
$c \leftarrow b,$	$\neg c \leftarrow b.$
$d \leftarrow not\ e.$	

Π_1 has a unique answer set $\{a, d\}$. Then Step (1) Zhang–Foo's simple fact update of $\{a, d\}$ by Π_2, $Update(\{a, d\}, \Pi_2)$, which is equivalently to update $\{a \leftarrow, d \leftarrow\}$ with Π_2 using Sakama–Inoue's approach, will contain a single set $\{a, b, \neg c, d\}$. Applying Step (2), we obtain the final update result $\{a \leftarrow, d \leftarrow not\ e\} \cup \Pi_2$.

As we have already provided a transformation from Sakama–Inoue's approach to a logic program context, to show that Zhang–Foo's update approach can also be represented within our framework, it is sufficient to only transform Step (2) above into a conflict solving problem under certain logic program context.

As before, given two extended logic programs Π_1 and Π_2 over the set of atoms $\mathcal{A}$, we extend $\mathcal{A}$ to $\overline{\mathcal{A}}$ with new atom $\bar{a}$ for each $a \in \mathcal{A}$. Then by replacing each $\neg a$ in S' and Π_2 with $\bar{a}$, we obtain the corresponding normal logic programs $\overline{\Pi}_1$ and $\overline{\Pi}_2$ respectively. Suppose $Update(\overline{S}, \overline{\Pi}_2)$ is the result of the simple fact update, where $\overline{S}$ is a stable model of $\overline{\Pi}_1$.

Definition 17. Let $\overline{\Pi}_1$, $\overline{\Pi}_2$, and $Update(\overline{S}, \overline{\Pi}_2)$ be defined as above, and $\overline{S'} \in Update(\overline{S}, \overline{\Pi}_2)$. We specify a logic program context $\Sigma_{ZF} = ((\overline{\Pi}'_1, \emptyset, \mathcal{F}), (\emptyset, \mathcal{C}, \emptyset))$ over the set of atoms $\overline{A} \cup \{l_r \mid r \in \overline{\Pi}_1\}$ where l_r are newly introduced atoms:

(1) $\overline{\Pi'}_1$ consists of rules: (a) for each rule r: $head(r) \leftarrow pos(r), not\ neg(r)$ in $\overline{\Pi}_1$, $head(r) \leftarrow body(r), not\ l_r$ is in $\overline{\Pi'}_1$, and (b) $\overline{S'} \subseteq \overline{\Pi'}_1$,
(2) $\mathcal{F} = \{l_r \mid r \in \overline{\Pi}_1\}$,
(3) $\mathcal{C} = \{\leftarrow a, \bar{a} \mid a, \bar{a} \in \overline{A}\}$.

The following theorem shows that Step (2) in Zhang–Foo's approach can be precisely characterized by a logic program context specified in Definition 17.

Theorem 11. *Let $\overline{\Pi}_1$, $\overline{\Pi}_2$, Σ_{ZF}, and $Update(\overline{S}, \overline{\Pi}_2)$ be defined as above, and $\overline{S'} \in Update(\overline{S}, \overline{\Pi}_2)$. $\overline{\Pi}^*$ is a maximal subset of $\overline{\Pi}_1$ such that $\overline{\Pi'} = \overline{\Pi}^* \cup \overline{S'}$ is consistent iff $\overline{\Pi''}$ is in a preferred solution of Σ_{ZF}: $\Sigma'_{ZF} = ((\overline{\Pi''}, \emptyset, \mathcal{F}), (\emptyset, \mathcal{C}, \emptyset))$, where $\overline{\Pi''} = \{r\colon head(r) \leftarrow pos(r), not\ neg(r), not\ l^r \mid r \in \overline{\Pi}^*\}$.*

The proof of Theorem 11 is similar to that of Theorem 10.

6.5. Further discussions: Updates, constraints, and expressiveness

From previous descriptions, we observe that the key step to transform an update approach into a sequence of logic program contexts (or one logic program context like the case of SI approach) is to construct the underlying constraints for conflict solving. In both Eiter et al.'s causal rejection and DLP approaches, constraints are specified based on atoms, e.g. $\leftarrow a_{n-i}, \bar{a}_{n-i+1}$ in Ω_{CR}, and $\leftarrow a_i, a_{P_i^-}$ in Ω_{DLP}.

For SI approach, on the other hand, the underlying constraints are specified as the entire update program. For instance, consider the update of Π_1 by Π_2 using SI approach, the corresponding logic program context for this update is of the form $\Sigma = ((\Pi, \emptyset, \mathcal{F}), (\emptyset, \Pi_2, \emptyset))$, in which program Π_2 serves as constraints for conflict solving.

Finally, since Zhang and Foo's integrated update approach combined both model and syntax based approaches, the transformation of this approach into logic program context framework consists of two steps: an equivalent SI transformation with program based constraints, followed by another transformation with atoms based constraints (see Definition 17).

From the above observation, we can see that the main difference between model based and syntax based update approaches is to solve conflicts under different types of constraints, namely atoms based and program based constraints respectively.

While we have shown that our conflict solving approach provides a unified framework to represent different kinds of logic program updates, we should indicate that our approach does not give specific computational advantages over these logic program update approaches. As we will see in Section 7, conflict solving under our framework is generally intractable. From previous definitions, we also observe that transforming model based logic program updates into a sequence of logic program contexts may need exponential time because it involves the computation of solutions of logic program contexts, although transforming syntax based logic program updates can always be done in polynomial time.

Nevertheless, the most significant feature of using our logic program contexts to represent logic program updates is to provide an expressive framework that unifies many different logic program update approaches. Under the unified framework, it becomes possible to analyze and compare syntactic and semantic properties of these different approaches.

7. Computational issues

In this section, we study related computational issues. In particular, we consider two major computational problems concerning (1) irrelevance in reasoning with respect to strong and weak forgettings and conflict solving, and (2) general decision problems for conflict solving under the framework of logic program contexts.

We first introduce basic notions from complexity theory and refer to [21] for further details. Two important complexity classes are P and NP. The class P includes all languages recognizable by a polynomial-time deterministic Turing machine. The class NP, on the other hand, consists of those languages recognizable by a polynomial-time nondeterministic Turing machine. The class of coNP is the complements of class NP. 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. The class $\Sigma_2^P = NP^{NP}$ includes all languages recognizable in polynomial time by a nondeterministic Turing machine with an NP oracle, where the class Π_2^P is the complement of Σ_2^P, i.e. $\Pi_2^P = co\Sigma_2^P$. It is well known that $P \subseteq NP \subseteq DP \subseteq \Sigma_2^P$, and these inclusions are generally believed to be proper.

7.1. Complexity results on irrelevance

By definitions, we can see that the main computation of strong and weak forgettings relies on the procedure of reduction that further inherits the computation of the conventional program unfolding. Hence, it is easy to observe that in the worst case, the size of the resulting program after strong (or weak) forgetting could be exponentially larger than the original program. This means that in general computing strong and weak forgettings in logic programs is hard. However, the following result shows that this actually does not increase the complexity of the associated inference problem.

Theorem 12. *Let Π be a logic program, P a set of atoms, and a an atom. Deciding whether $SForgetLP(\Pi, P) \models a$ (or $WForgetLP(\Pi, P) \models a$) is coNP-complete.*

Proof. The hardness is obvious when $P = \emptyset$. To prove the membership, we first specify two transformations on Π with respect to P. The program $STrans(\Pi, P)$ is obtained from Π by removing some rules in Π: (1) for each $p \in P$, if $p \notin head(\Pi)$, then removing rules r in Π with $p \in pos(r)$; (2) if $p \notin pos(\Pi)$, then removing rules r in Π with $head(r) = p$; and (3) removing rules r in Π with $p \in neg(r)$. The program $WTrans(\Pi, P)$, on the other hand, is obtained from Π in the same way as program $STrans(\Pi, P)$ except (3): for rules r in Π having $p \in neg(r)$, change it to be of the form: r': $head(r) \leftarrow pos(r), not(neg(r) - \{p\})$. Now we prove the following two results:

Result 1. *$SForgetLP(\Pi, P)$ is consistent if and only if program $STrans(\Pi, P)$ is consistent, and each of $SForgetLP(\Pi, P)$'s stable models S' can be expressed as $S' = S - P$, where S is a stable model of $STrans(\Pi, P)$.*

Result 2. *$WForgetLP(\Pi, P)$ is consistent if and only if program $WTrans(\Pi, P)$ is consistent, and each of $WForgetLP(\Pi, P)$'s stable models S' can be expressed as $S' = S - P$, where S is a stable model of $WTrans(\Pi, P)$.*

Here we give the proof of Result 1, while Result 2 can be proved in a similar way. Firstly, we assume that $SForgetLP(\Pi, P)$ is consistent and S' is a stable model of $SForgetLP(\Pi, P)$. Then we show that $STrans(\Pi, P)$ must have a stable model S such that $S' = S - P$. Observing the construction of the structure of $STrans(\Pi, P)$, we can see that for each $p \in P$ occurring in $STrans(\Pi, P)$, there are two rules r_1 and r_2 in $STrans(\Pi, P)$ of the forms:

$$r_1\colon\ p \leftarrow pos(r_1), not\ neg(r_1),$$

$$r_2\colon\ head(r_2) \leftarrow p, pos(r_2), not\ neg(r_2),$$

and furthermore, we also have $P \cap neg(STrans(\Pi, P)) = \emptyset$. Now we present an algorithm to construct a set S of atoms as follows:

```
Algorithm: **Generating** $S$
**Input**: $STrans(\Pi, P)$ and $S'$ where $S'$ is a stable model of $SForgetLP(\Pi, P)$;
**Output**: a set $S$ of atoms;
**let** $S = S'$;
selecting a rule $r$ from $STrans(\Pi, P)$ of the form:
    $r$: $p \leftarrow pos(r), not\ neg(r)$, where $p \in P$ and $pos(r) \cap P = \emptyset$;
**if** no such rule exists in $Strans(\Pi, P)$, **then return** $S$;
**else**
    **if** each $a \in pos(r)$ is in $S'$ and each $b \in neg(r)$ is not in $S'$,
        **then** $S = S \cup \{p\}$;
**repeat** the following two steps until $S$ no longer changes
    selecting a rule $r'$ from $STrans(\Pi, P)$ of the form:
        $r'$: $p \leftarrow pos(r'), not\ neg(r')$ where $p \in P$;
    **if** each $a \in pos(r')$ is in $S$ and each $b \in neg(r')$ is not in $S$,
        **then** $S = S \cup \{p\}$;
**return** $S$.
```

We need to show that S generated from the above algorithm is a stable model of *STrans*(Π, P). We perform Gelfond–Lifschitz transformation on *STrans*(Π, P) with S, and obtain program *STrans*$(\Pi, P)^S$. First, we prove that for each rule r: $head(r) \leftarrow pos(r)$ in *STrans*$(\Pi, P)^S$, if $pos(r) \subseteq S$, then $head(r) \in S$.

Case 1. If $pos(r) \subseteq S'$, then $head(r) \subseteq S$ according to the algorithm.

Case 2. Suppose r is of the form: r: $head(r) \leftarrow p, pos(r)$, where $p \in P$, $\{p\} \cup pos(r) \subseteq S$ and $pos(r) \subseteq S'$. In this case, we show $head(r) \in S$. This is true if $head(r) \in P$ according to the above algorithm. Now suppose $head(r) \not\subseteq P$. Consider r's original form in *STrans*(Π, P): r': $head(r) \leftarrow p', pos(r), not\ neg(r')$ (i.e. the part $not\ neg(r')$ is removed in *STrans*$(\Pi, P)^S$). Recall the structure of *STrans*(Π, P), in which there exists a rule r'': $p \leftarrow pos(r''), not\ neg(r'')$. By performing proper reduction, eventually we can replace r'' with a new rule: r^*: $p \leftarrow pos(r^*), not\ neg(r^*)$ such that $P \cap pos(r^*) = \emptyset$ (note that if we can not reach this form of rule r^*, for instance, $P \cap pos(r^*) \neq \emptyset$, we will have $p \notin S$ according to the above algorithm). As $p \in S$, we must have $pos(r^*) \subseteq S$, and hence $pos(r^*) \subseteq S'$. On the other hand, it is not hard to observe that a rule of the form is in *SForgetLP*$(\Pi, P)^{S'}$: $head(r) \leftarrow pos(r), pos(r^*)$. Since we already know that $pos(r) \cup pos(r^*) \subseteq S'$ and S' is a stable model of *SForgetLP*(Π, P), it follows that $head(r) \in S'$ and hence $head(r) \in S$ as $S' \subseteq S$.

On the other hand, it is also easy to show that S' generated from the above algorithm is the smallest set to have the above property for program *STrans*(Π, P). This proves that S is a stable model of *STrans*(Π, P).

Now we assume that *STrans*(Π, P) is consistent and S is a stable model of *STrans*(Π, P). In this case, we simply prove that $S' = S - P$ is a stable model of *SForgetLP*(Π, P). We omit the proof as it is easy to verify.

Having these results, the membership is proved as follows. For the case of strong forgetting, we consider the complement of the problem. Clearly, it is easy to see that the *STrans*(Π, P) can be obtained from Π in polynomial time. Guessing a S stable model of *STrans*(Π, P), verifying it, and checking whether $a \notin S - P$ can be done in polynomial time. So the complement of the problem is in NP. Consequently, the problem is in coNP. Proof for the case of weak forgetting is the same. □

From the above result, we can show the complexity of irrelevance in relation to strong and weak forgettings.

Theorem 13. *Let Π be a logic program, P a set of atoms and a an atom. Deciding whether a is irrelevant to P in Π is coDP-complete.*

Proof. To prove this theorem, we need to show deciding whether $\Pi \models a$ iff *SForgetLP*$(\Pi, P) \models a$ (s-irrelevant) is coDP-complete, and deciding whether $\Pi \models a$ iff *WForgetLP*$(\Pi, P) \models a$ (w-irrelevant) is coDP-complete. Here we only give the proof of the first statement, and the second can be proved in a similar way.

Membership. To decide whether $\Pi \models a$ iff *SForgetLP*$(\Pi, P) \models a$, we need to show $\Pi \models a$ and *SForgetLP*$(\Pi, P) \models a$, or $\Pi \not\models a$ and *SForgetLP*$(\Pi, P) \not\models a$. Clearly, given Π, P and a, deciding whether $\Pi \models a$ and *SForgetLP*$(\Pi, P) \models a$ is in coNP, and deciding whether $\Pi \not\models a$ and *SForgetLP*$(\Pi, P) \not\models a$ is in NP (see Theorem 12). So the problem is in coDP.

Hardness. We consider a pair (Φ_1, Φ_2) of CNFs and from which we polynomially construct a program Π, a set of atoms P and an atom a, and 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*$(\Pi, P) \not\models a$.

Let $\Phi_1 = \{C_1, \ldots, C_m\}$ and $\Phi_2 = \{C'_1, \ldots, C'_n\}$, where each C_i and C'_j ($1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant n$) are sets of propositional literals respectively. We also assume that Φ_1 and Φ_2 do not share any propositional atoms. Now we construct a program Π based on propositional atoms $atom(\Phi_1) \cup atom(\Phi_2) \cup \hat{X} \cup \hat{Y} \cup \{l_1, \ldots, l_n, p, a, sat^{\Phi_1}, unsat^{\Phi_1}, unsat^{\Phi_2}\}$, where any two sets of atoms are disjoint and $|\hat{X}| = |atom(\Phi_1)|$ and $|\hat{Y}| = |atom(\Phi_2)|$. Program Π consists of four groups of rules:

Π_1:

for each $x \in atom(\Phi_1)$, we have:

$x \leftarrow not\ \hat{x}$,

$\hat{x} \leftarrow not\ x$,

for each $y \in atom(\Phi_2)$, we have:

$y \leftarrow not\ \hat{y}$,

$\hat{y} \leftarrow not\ y$,

Π_2:

$unsat^{\Phi_1} \leftarrow \overline{C_1}$,

$\dots$,

$unsat^{\Phi_1} \leftarrow \overline{C_m}$,

$unsat^{\Phi_2} \leftarrow \overline{C'_1}$,

$\dots$,

$unsat^{\Phi_2} \leftarrow \overline{C'_n}$,

where for each clause C_i (or C'_j), if $b \in C_i$ (or C'_j resp.), then $not\ b \in \overline{C_i}$ (or $\overline{C'_j}$ resp.), and if $\neg b \in C_i$ (or C'_j resp.) then $b \in \overline{C_i}$ (or $\overline{C'_j}$ resp.),

Π_3:

$l_1 \leftarrow unsat^{\Phi_2}, not\ l_2, \dots, not\ l_n$,

$\dots$,

$l_n \leftarrow unsat^{\Phi_2}, not\ l_1, \dots, not\ l_{n-1}$,

$pos(\overline{C'_j}) \leftarrow l_j\ (1 \leqslant j \leqslant n)$,

where $pos(\overline{C'_j}) \leftarrow l_j$ represents a group of rules: for all atoms $b \in \overline{C'_j}$, we have $b \leftarrow l_j$ (note that if $not\ b \in \overline{C'_j}$, no rule is needed),

Π_4:

$sat^{\Phi_1} \leftarrow not\ unsat^{\Phi_1}$,

$a \leftarrow sat^{\Phi_1}$,

$unsat^{\Phi_2} \leftarrow not\ a$,

$p \leftarrow .$

Let us look at the intuition behind this program. Clearly, Π_1 generates all truth assignments for Φ_1 and Φ_2 (recall that $atom(\Phi_1) \cap atom(\Phi_2) = \emptyset$). This ensures that there is a correspondence between stable models of Π and truth assignments of Φ_1 and Φ_2. Π_2 indicates that if Φ_1 (or Φ_2) is unsatisfiable, then atom $unsat^{\Phi_1}$ (or $unsat^{\Phi_2}$ resp.) will be derived. Rules in Π_3 are used to force Φ_2 to be unsatisfiable. That is, if atom $unsat^{\Phi_2}$ is derived from through rule $unsat^{\Phi_2} \leftarrow \texttt{not}\ a$ in Π_4, then the corresponding truth assignment of Φ_2 in each stable model of Π must make some $\overline{C'_j}$ to be true.

Now we prove that Φ_1 is satisfiable or Φ_2 is unsatisfiable if and only if $\Pi \models a$ and $SForgetLP(\Pi, \{p\}) \models a$; or $\Pi \not\models a$ and $SForgetLP(\Pi, \{p\}) \not\models a$. We observe that $SForgetLP(\Pi, \{p\}) = \Pi - \{p \leftarrow\}$, which implies that if $\Pi \models a$ then $SForgetLP(\Pi, \{p\}) \models a$ and if $\Pi \not\models a$ then $SForgetLP(\Pi, \{p\}) \not\models a$.

Suppose that Φ_1 is satisfiable or Φ_2 is unsatisfiable. We consider the following cases. (1) If Φ_1 is satisfiable, then it is easy to see that none of rules in Π_2 with head $unsat^{\Phi_1}$ is applicable and hence atoms sat^{Φ_1} and a can be derived from Π. In this case, no matter if Φ_2 is satisfiable or unsatisfiable, we always have $\Pi \models a$ and $SForgetLP(\Pi, \{p\}) \models a$.

(2) If Φ_2 is unsatisfiable. In this case one of rules in Π_2 having $unsat^{\Phi_2}$ as heads is applicable and hence atom $unsat^{\Phi_2}$ is derivable from Π. In this case, if Φ_1 is satisfiable, then a is derived from Π. Otherwise, a is not derivable from Π. The same for $SForgetLP(\Pi, \{p\})$. So we have the statement: if Φ_1 is satisfiable or Φ_2 is unsatisfiable, then $\Pi \models a$ and $SForgetLP(\Pi, \{p\}) \models a$, or $\Pi \not\models a$ and $SForgetLP(\Pi, \{p\}) \not\models a$.

Suppose $\Pi \models a$ and $\mathit{SForgetLP}(\Pi, \{p\}) \models a$; or $\Pi \not\models a$ and $\mathit{SForgetLP}(\Pi, \{p\}) \not\models a$. (1) If $\Pi \models a$ and hence $\mathit{SForgetLP}(\Pi, \{p\}) \models a$. From the construction of Π, we know that the only way to derive a from Π is that rule $a \leftarrow \mathit{sat}^{\Phi_1}$ in Π_4 is applicable. This implies that none of rules in Π_2 having unsat^{Φ_1} as heads is applicable. Consequently, one of truth assignments generated from Π_1 for Φ_1 must satisfy Φ_1. So Φ_1 is satisfiable.

(2) If $\Pi \not\models a$ and hence $\mathit{SForgetLP}(\Pi, \{p\}) \not\models a$. In this case, sat unsat^{Φ_2} can be derived from rule $\mathit{unsat}^{\Phi_2} \leftarrow \mathit{not}\, a$. Then from rule in Π_3, we know that in each stable model of Π, the corresponding truth assignment of Φ_2 must not satisfy Φ_2. Since all truth assignments of Φ_2 have been represented in Π's stable models, this concludes that Φ_2 is unsatisfiable. This proves our result. □

The following complexity result of irrelevance with respect to logic program contexts is inherited from Theorem 13.

Theorem 14. *Let Σ and Σ' be two logic program contexts where $\Sigma' \in \mathit{Solution}(\Sigma)$, and a an atom. Deciding whether a is $(\Sigma, \Sigma')^i$-irrelevant is coDP-complete.*

7.2. Complexity results on conflict solving

Proposition 8. *Let Σ be a logic program context. Deciding whether Σ has a preferred solution is NP-hard.*

Proof. We consider a special form of logic program context $\Sigma = ((\Pi_1, \emptyset, \emptyset), \ldots, (\Pi_n, \emptyset, \emptyset))$. Clearly, Σ has a solution iff each Π_i has a stable model, and we know checking whether a program has stable is NP-hard. On the other hand, from Theorem 6, we know that Σ has a preferred solution iff $\mathit{Solution}(\Sigma) \neq \emptyset$. Then the result directly follows. □

We observe that computing a solution for a logic program context consists of two major stages: (1) computing strong and weak forgettings, and (2) consistency testing for all $\Pi_i \cup \mathcal{C}_j$ in the resulting logic program context (see Definition 6). While many existing results may be used for efficient consistency testing of a logic program (e.g. see Section 5.2 and Chapter 3 in [3]), it is important to investigate possible optimizations for computing strong and weak forgettings in logic programs.

For this purpose, we first introduce a useful notion. Let Π be a logic program, a an atom in $\mathit{atom}(\Pi)$, and $G(\Pi)$ the dependency graph of Π. In $G(\Pi)$, we call a positive path[13] without cycles starting from a the *inference chain* starting from a. We define the *inference depth* of a, denoted as $i\text{-}\mathit{depth}(a)$, to be the length of the longest inference chain starting from a in $G(\Pi)$. Intuitively, $i\text{-}\mathit{depth}(a)$ represents the maximal number of rules that may be used to derive any other atoms starting from a in program Π. We denote the *inference depth* of Π as

$$i\text{-}\mathit{depth}(\Pi) = \mathit{Max}\big(i\text{-}\mathit{depth}(a)\colon\ a \in \mathit{atom}(\Pi)\big).$$

It turns out that the inference depth plays a key role in characterizing the computation of strong and weak forgettings in logic programs.

Theorem 15. *Let Π be a logic program. If Π has a bounded inference depth, i.e. $i\text{-}\mathit{depth}(\Pi) \leqslant c$ for some constant c, then for any set of atoms $P \subseteq \mathit{atom}(\Pi)$, $\mathit{SForgetLP}(\Pi, P)$ and $\mathit{WForgetLP}(\Pi, P)$ can be computed in polynomial time.*

Proof. To prove this theorem, we only need to show that under the condition of bounded inference depth, $\mathit{Reduct}(\Pi, P)$ is polynomially achievable for any $P \subseteq \mathit{atom}(\Pi)$. Without loss of generality, for $P = \{p_1, \ldots, p_k\}$, we may assume that Π consists of three components:

Π_1:

$r_{11}\colon\ p_1 \leftarrow \mathit{pos}(r_{11}), \mathit{not}\ \mathit{neg}(r_{11}),$

$\ldots,$

$r_{1l_1}\colon\ p_1 \leftarrow \mathit{pos}(r_{1l_1}), \mathit{not}\ \mathit{neg}(r_{1l_1}),$

[13] That is, a path does not contain any negative edges.

r_{21}: $p_2 \leftarrow pos(r_{21}), not\ neg(r_{21})$,

$\ldots$,

r_{2l_2}: $p_2 \leftarrow pos(r_{2l_2}), not\ neg(r_{2l_2})$,

$\ldots$,

r_{k1}: $p_k \leftarrow pos(r_{k1}), not\ neg(r_{k1})$,

$\ldots$,

r_{kl_k}: $p_k \leftarrow pos(r_{kl_k}), not\ neg(r_{kl_k})$,

Π_2[14]:

r_1: $head(r_1) \leftarrow p_1, pos(r_1), not\ neg(r_1)$,

r_2: $head(r_2) \leftarrow p_2, pos(r_2), not\ neg(r_2)$,

$\ldots$,

r_k: $head(r_k) \leftarrow p_k, pos(r_k), not\ neg(r_k)$,

Π_3,

where the reduction only occurs among rules in $\Pi_1 \cup \Pi_2$, and Π_3 contains all rules irrelevant to the reduction process. Now we show that if $i\text{-}depth(\Pi) \leqslant c$ for some constant c, the size of $Reduct(\Pi, P)$ will be at most polynomial times of the size of Π. Indeed, since $i\text{-}depth(\Pi) \leqslant c$, it follows that for each $p_i \in P$, $i\text{-}depth(p_i) \leqslant c$ in program Π_1. This implies that during the reduction, for each p_i's occurrence in other rule's positive body, at most only $h_1 \times \cdots \times h_{c+1}$, where $\{h_1, \ldots, h_{c+1}\} \subseteq \{l_1, \ldots, l_k\}$, new rules will be introduced due to the inference chain in Π_1 starting from a. This number of rules is bounded by $|\Pi|^{c+1}$. If p_i occurs in all other rules' positive bodies in Π_1, the total number of new rules possibly introduced through reduction via p_i is bounded by $|P| \times |\Pi|^{c+1}$. Therefore, the number of all new rules introduced through the entire reduction via P is bounded by $\mathcal{O}(|P|^2 \times |\Pi|^{c+1})$. In other words, to perform $Reduct(\Pi, P)$, the number of all operations on rule substitutions and replacements is bounded by $\mathcal{O}(|P|^2 \times |\Pi|^{c+1})$. □

Theorem 16. *Let $\Sigma = (\Phi_1, \ldots, \Phi_n)$ and $\Sigma' = (\Phi'_1, \ldots, \Phi'_n)$ be two logic program contexts, where for each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma$ $(1 \leqslant i \leqslant n)$, $\Phi'_i \in \Sigma'$ 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 $P_i \subseteq \mathcal{F}_i$. Then the following results hold*:

(1) *Deciding whether Σ' is a solution of Σ is NP-complete*;
(2) *Deciding whether Σ' is a preferred solution of Σ is in Π_2^P provided that strong and weak forgettings in Σ can be computed in polynomial time*;[15]
(3) *For a given atom a, deciding whether for all $\Sigma'' \in Solution(\Sigma)$, $\Sigma'' \models_i a$ is in Π_2^P provided that strong and weak forgettings in Σ can be computed in polynomial time.*

Proof. Result 1 is easy to prove. To check if Σ' is a solution of Σ, we only need to check whether $\Pi'_i \cup C_j$ is consistent for all i and j, and altogether we need to do n^2 such consistency checkings. On the other hand, we know that checking the consistency of $\Pi'_i \cup C_j$ is in NP. So the problem is in NP. For the hardness, just consider a special case where $n = 1$, then Σ' is a solution of Σ iff $\Pi'_1 \cup C_1$ is consistent, and this is NP-hard.

To prove Result 2, we consider the complement of the problem. If Σ' is not a preferred solution of Σ, then there must exist Σ'' such that $\Sigma'' \in Solution(\Sigma)$ and $\Sigma'' \prec_\Sigma \Sigma'$. This equals to that there are $P''_1, \ldots, P''_n$ where $P''_i \subseteq P_i$ and for some k we have $P''_k \subset P_k$ such that (1) $\Sigma'' = ((\Pi''_n, \mathcal{C}_1, \mathcal{F}_1), \ldots, (\Pi''_n, \mathcal{C}_n, \mathcal{F}_n))$, and each Π''_i is of the form $SForgetLP(\Pi_i, P''_i)$ or $WForgetLP(\Pi_i, P''_i)$; and (2) $\Sigma'' \in Solution(\Sigma)$. Clearly, guessing such $P''_1, \ldots, P''_n$ and computing each $SForgetLP(\Pi_i, P''_i)$ and $WForgetLP(\Pi_i, P''_i)$ can be done in polynomial time. Then we can

[14] In Π_2, there may be more than one rules having p_i in their positive bodies. But this simplified case does not affect our proof.

[15] Computing strong and weak forgettings in Σ, we mean that for each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma$ and $P \subseteq \mathcal{F}_i$, we compute $SForgetLP(\Pi_i, P)$ and $WForgetLP(\Pi_i, P)$.

construct a Σ'' in polynomial time, where Σ'' is of the form $\Sigma'' = ((\Pi_1'', \mathcal{C}_i, \mathcal{F}_1), \ldots, (\Pi_n'', \mathcal{C}_n, \mathcal{F}_n))$, in which for each i, Π_i'' can be either *SForgetLP*(Π_i, P_i'') or *WForgetLP*(Π_i, P_i''). Then checking whether Σ'' is a solution of Σ can be achieved with number of n^2 calls for an NP oracle. So the problem is in Σ_2^P. Consequently, the original problem is in Π_2^P.

We prove Result 3 as follows. We guess a set of atoms S_i, and n sets of atoms $P_1, \ldots, P_n$ such that $P_i \subseteq \mathcal{F}_i$ for each $1 \leqslant i \leqslant n$. Then similarly to the proof of Result 2, we can construct a logic program context Σ in polynomial time. Checking whether $\Sigma' \in \mathit{Solution}(\Sigma)$ can be achieved with one call to an NP oracle. Then checking whether S_i is a stable model of a particular Π_i', where $\Phi_i' \in \Sigma'$ and $\Phi_i' = (\Pi', \mathcal{C}_i, \mathcal{F}_i)$, and $a \notin S_i$ can be done in polynomial time as well. So the complement of the problem is in Σ_2^P, and thus the original problem is in Π_2^P. □

8. 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 framework of logic program contexts. We then studied the irrelevance property related to strong and weak forgettings and conflict solving and provided various solution characterizations for logic program contexts. We showed that our approach presented in this paper is quite general and unified all major logic program update approaches. We also analyzed the computational complexity of strong and weak forgettings in logic programs and conflict solving in logic programs contexts.

We noted that there were other methods for solving the inconsistency of logic programs in the literature, especially the work involving abductive reasoning in logic programs. For instance, Inoue's method of deletion and addition of names of rules [8], where certain atoms can be blocked from derivation by removing/adding some rules in the program. In this case, these atoms are still presented in the program. As we have shown in Section 6.3, by introducing new atom such as l^r in the language, our approach can simply model this method to solve program inconsistency. The main difference between our approach and others is that we presented a very general framework based on strong and weak forgettings, and this framework can handle many different types of conflict solving scenarios including logic program updates, negotiation and belief merging, that seem to be difficult for any other single method in the literature (e.g. see Example 6 in Section 4).

Our work presented in this paper can be further extended. One interesting issue is to integrate dynamic preference orderings on forgettable atoms into the current framework of logic program contexts, so that the extended framework can represent domain-dependent conflict solving cases. This is particularly important when we use this approach to represent complex belief merging (e.g. [10,11,16]) and negotiations under the setting of logic programming, in which each agent usually has different preferences on the atoms that she may forget for a final agreement.

Acknowledgements

The authors thank Fangzhen Lin for useful discussions on this topic and valuable comments on an early draft of this paper. The research of the first author was supported in part by Australian Research Council under grant DP0666540. The authors also thank the anonymous reviewers for their insightful comments and criticisms which were helpful in revising and improving the paper.

References

[1] K.R. Apt, H.A. Blair, A. Walker, Towards a theory of declarative knowledge, in: J. Minker (Ed.), Foundations of Deductive Database and Logic Programming, Morgan Kaufmann, 1988, pp. 293–322.
[2] J.J. Alferes, J.A. Leite, L.M. Pereira, et al., Dynamic logic programming, in: Proceedings of KR-98, 1998, pp. 98–111.
[3] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University, 2002.
[4] S. Brass, J. Dix, A general framework for semantics of disjunctive logic programs based on partial evaluation, Journal of Logic Programming 38 (3) (1998) 167–213.
[5] T. Eiter, M. Fink, G. Sabbatini, H. Tompits, On properties of update sequences based on causal rejection, Theory and Practice of Logic Programming 2 (2002) 711–767.
[6] T. Eiter, M. Fink, G. Sabbatini, H. Tompits, Reasoning about evolving nonmonotonic knowledge base, ACM Transaction on Computational Logic 6 (2005) 389–440.
[7] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proceedings of the International Conference on Logic Programming, The MIT Press, 1988, pp. 1070–1080.

[8] K. Inoue, A simple characterization of extended abduction, in: Proceedings of the First International Conference on Computational Logic (CL-2000), 2000, pp. 718–732.
[9] K. Inoue, C. Sakama, Update of equivalence of logic programs, in: Proceedings of JELIA 2004, 2004.
[10] S. Konieczny, R. Pino Pérez, On the logic of merging, in: Proceedings of the 6th International Conference on Knowledge Representation and Reasoning (KR-98), 1998, pp. 488–498.
[11] S. Konieczny, R. Pino Pérez, Propositional belief base merging or how to merge beliefs/goal coming from several sources and some links with social choice theory, European Journal of Operational Research 160 (3) (2005) 785–802.
[12] K. Kunen, Signed data dependencies in logic programs, Journal of Logic Programming 7 (3) (1989) 231–245.
[13] J. Lang, P. Marquis, Resolving inconsistencies by variable forgetting, in: Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR-2002), Morgan Kaufmann Publishers, 2002, pp. 239–250.
[14] J. Lang, P. Liberatore, P. Marquis, Propositional independence—Formula-variable independence and forgetting, Journal of Artificial Intelligence Research 18 (2003) 391–443.
[15] J.A. Leite, Evolving Knowledge Bases: Specification and Semantics, IOS Press, 2003.
[16] P. Liberatore, M. Schaerf, A system for the integration of knowledge bases, in: Proceedings of the 7th International Conference on Knowledge Representation and Reasoning (KR-2000), Morgan Kaufmann Publishers, 2000, pp. 145–152.
[17] V. Lifschitz, D. Pearce, A. Valverde, Strongly equivalent logic programs, ACM Transactions on Computational Logic 2 (4) (2001) 426–541.
[18] F. Lin, R. Reiter, Forget it!, in: Working Notes of AAAI Fall Symposium on Relevance, 1994, pp. 154–159.
[19] F. Lin, On the strongest necessary and weakest sufficient conditions, Artificial Intelligence 128 (2001) 143–159.
[20] F. Lin, Y. Chen, Discovering classes of strongly equivalent logic programs, in: Proceedings of IJCAI-2005, 2005.
[21] C.H. Papadimitriou, Computational Complexity, Addison Wesley, 1995.
[22] C. Sakama, K. Inoue, Updating extended logic programs through abduction, in: Proceedings of LPNMR'99, 1999, pp. 2–17.
[23] C. Sakama, H. Seki, Partial deduction in disjunctive logic programming, Journal of Logic Programming 32 (3) (1997) 229–245.
[24] K. Su, G. Lv, Y. Zhang, Reasoning about knowledge by variable forgetting, in: Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR-2004), Morgan Kaufmann Publishers, 2004, pp. 576–586.
[25] J.-H. You, L. Yuan, A three-valued semantics for deductive databases and logic programs, Journal of Computer and System Sciences 49 (2) (1994) 334–361.
[26] Y. Zhang, Two results for prioritized logic programming, Theory and Practice of Logic Programming 3 (2) (2003) 223–242.
[27] Y. Zhang, N. Foo, Updating logic programs, in: Proceedings of ECAI-1998, 1998, pp. 403–407.
[28] Y. Zhang, Logic program based updates, ACM Transaction on Computational Logic, submitted for publication http://www.acm.org/pubs/tocl/accepted.html, 2006.