Computing Loops with at Most One External Support Rule

Xiaoping Chen and **Jianmin Ji**
University of Science and Technology of China
P. R. China
xpchen@ustc.edu.cn, jizheng@mail.ustc.edu.cn

Fangzhen Lin
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
flin@cs.ust.hk

Abstract

If a loop has no external support rules, then its loop formula is equivalent to a set of unit clauses; and if it has exactly one external support rule, then its loop formula is equivalent to a set of binary clauses. In this paper, we consider how to compute these loops and their loop formulas in a normal logic program, and use them to derive consequences of a logic program. We show that an iterative procedure based on unit propagation, the program completion and the loop formulas of loops with no external support rules can compute the same consequences as the "Expand" operator in smodels, which is known to compute the well-founded model when the given normal logic program has no constraints. We also show that using the loop formulas of loops with at most one external support rule, the same procedure can compute more consequences, and these extra consequences can help ASP solvers such as cmodels to find answer sets of certain logic programs.

Introduction

The notions of loops and loop formulas were first proposed by Lin and Zhao (2004) for propositional normal logic programs. They have been extended to disjunctive logic programs (Lee & Lifschitz 2003), general logic programs (Ferraris, Lee, & Lifschitz 2006), normal logic programs with variables (Chen *et al.* 2006), and propositional circumscription (Lee & Lin 2006). They can be used for computing answer sets of a logic program by using an off-the-shelf SAT solver (Lin & Zhao 2004) or by modifying an existing SAT procedure (Giunchiglia, Lierler, & Maratea 2006). They are also useful for ASP (Answer Set Programming) solvers that use SAT-like strategies such as conflict-clause learning (Gebser *et al.* 2007).

In this paper, we propose to use loops and loop formulas to compute *consequences* of a logic program, which are formulas that are satisfied by every answer set of the logic program. In particular, we focus on consequences that are literals and can be computed efficiently. Our main motivation for this work is to use these consequences to speed-up ASP solvers.

Lin and Zhao (2004) showed that a set of atoms is an answer set of a normal logic program iff it satisfies the completion and the loop formulas of the program. Thus the consequences of a logic program are the logical consequences of its completion and loop formulas. However, deduction in propositional logic is coNP-hard and in general there may be an exponential number of loops (Lifschitz & Razborov 2006). One way to overcome these problems is to use some tractable inference rules and consider only those loop formulas that can be used effectively by these inference rules and at the same time can be computed efficiently.

In this paper, we choose unit propagation as the inference rule. To find loop formulas that are "unit propagation" friendly, let us look at the form of loop formulas.

According to (Lin & Zhao 2004), a loop L is a set of atoms, and its loop formula is a sentence of the form:

$$\bigvee L \supset \bigvee_{r \in R^-(L)} body(r),$$

where $R^-(L)$ is the set of so-called *external support rules* of L, and $body(r)$ is the conjunction of the literals in the body of the rule r. Without going into details about the definition of external support rules and how they are computed, we see that if a loop L has no external support rules, then its loop formula is equivalent to the following set of literals:

$$\{\neg a \mid a \in L\},$$

and if a loop L has exactly one external support rule, say r, then its loop formula is equivalent to the following set of binary clauses:

$$\{\neg a \vee l \mid a \in L,\ l \in body(r)\}.$$

Thus we see that loops that have at most one external support rule are special in that their loop formulas will yield unit or binary clauses that can be used effectively by unit propagation.

More generally, if we assume a set A of literals, then for any loop that has at most one external support rule whose body is not false under A, its loop formula is equivalent to either a set of literals or a set of binary clauses under A.

Since the completion of a logic program can be computed and converted to a set of clauses in linear time (by introducing new variables if necessary), if these loop formulas can also be computed in polynomial time, we then have a polynomial time algorithm for computing some consequences of a logic program. One such algorithm is as follows:

1. compute the program completion and convert it to clauses;
2. compute the loop formulas of the loops with at most one external support rule;
3. apply unit propagation to the set of clauses obtained so far;
4. compute the loop formulas of the loops with at most one external support under the set of literals computed in the previous step, add them to the set of clauses, and go back to the previous step until it does not produce any new consequences.

We shall show that for normal logic programs, the loop formulas of loops with at most one external support rule can indeed be computed in polynomial time. Furthermore, if we consider only loops that do not have any external support rules in the above procedure, then it computes essentially the same set of literals as does the $Expand$ operator in smodels (Simons, Niemelä, & Soininen 2002). In particular, it computes the well-founded model when the given normal logic program has no constraints. In general, this procedure can be more powerful than the $Expand$ operator in smodels, when loops with external support rules are considered. As an example, consider the Hamiltonian Circuit (HC) problem. If the set of vertices of the given graph has two parts, A and B, such that there is exactly one arc from a node in A to a node in B, and one arc from a node in B to a node in A, then any HC of the graph must go through these two arcs. This means that every answer set of the logic program for this HC problem must contain these two arcs, and knowing this in advance should help in computing an answer set of the program. But none of these two "must in" arcs cannot be computed using the $Expand$ operator, but one of them can be computed using our procedure.

Notice that the above procedure works in principle for more general logic programs, such as disjunctive logic programs, for which notions like the program completion, loops, and loop formulas can be defined. Thus it may prove to be a useful way for extending the well-founded semantics to these more expressive logic programs, just as it does for normal logic programs.

This paper is organized as follows. We briefly review the basic notions of logic programming in the next section. We then define loops with at most one external support rule under a given set of literals, and consider how to compute their loop formulas. We then consider how to use these loop formulas to derive consequences of a logic program using unit propagation, and relate it to the $Expand$ operator in smodels. Finally, we discuss some other possible uses of these loop formulas.

Preliminaries

In this paper, we consider only fully grounded finite normal logic programs that may have constraints. That is, a logic program here is a finite set of rules of the form:

$$H \leftarrow p_1, \ldots, p_k, not\ q_1, \ldots, not\ q_m, \qquad (1)$$

where p_i, $1 \leq i \leq k$, and q_j, $1 \leq j \leq m$, are atoms, and H is either empty or an atom. If H is empty, then this rule is also called a *constraint*, and if H is an atom, it is a *proper rule*.

Given a logic program P, we denote by $Atoms(P)$ the set of atoms in it, and $Lit(P)$ the set of literals constructed from $Atoms(P)$:

$$Lit(P) = Atoms(P) \cup \{\neg a \mid a \in Atoms(P)\}.$$

Given a literal l, the *complement* of l, written $\bar{l}$ below, is $\neg a$ if l is a and a if l is $\neg a$, where a is an atom. For a set L of literals, we let $\overline{L} = \{\bar{l} \mid l \in L\}$.

For a rule r of the form (1), we let $head(r)$ be its head H, $body(r)$ the set $\{p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_m\}$ of literals obtained from the body of the rule with "not" replaced by "$\neg$", $body^+(r)$ the set of atoms in its body, $\{p_1, \ldots, p_k\}$, and $body^-(r)$ the set of atoms under not in its body, $\{q_1, \ldots, q_m\}$.

To define the *answer sets* of a logic program with constraints, we first define the *stable models* of a logic program that does not have any constraints (Gelfond & Lifschitz 1988). Given a logic program P without constraints, and a set S of atoms, the Gelfond-Lifschitz transformation of P on S, written P_S, is obtained from P by deleting:

1. each rule that has a negative literal $not\ q$ in its body with $q \in S$, and
2. all negative literals in the bodies of the remaining rules.

Clearly for any S, P_S is a set of rules without any negative literals, so that P_S has a unique minimal model, denoted by $\Gamma_P(S)$. Now a set S of atoms is a stable model of P iff $S = \Gamma_P(S)$.

In general, given a logic program P that may have constraints, a set S of atoms is an answer set of P iff it is a stable model of the program obtained by deleting all the constraints in P, and it satisfies all the constraints in P, i.e. for any constraint of the form (1) such that H is empty, either $p_i \notin S$ for some $1 \leq i \leq k$ or $q_j \in S$ for some $1 \leq j \leq m$.

Notice that an answer set is a set of atoms, and can be considered as a logical model: the model assigns true to an atom iff it is in the answer set. In the following, when we say that an answer set satisfies a logical formula, we refer to the answer set as a logical model. Thus a sentence is a consequence of a logic program if it is true in every answer set of the logic program.

The well-founded semantics (Van Gelder, Ross, & Schlipf 1991) for programs without constraints can be defined using the fixed points of the operator Γ_P^2 (Van Gelder 1989), defined as follows: $\Gamma_P^2(S) = \Gamma_P(\Gamma_P(S))$. The operator Γ_P is anti-monotonic, so Γ_P^2 is monotonic. Thus it has a least and a greatest fixed point which we denote by $\text{lfp}(\Gamma_P^2)$ and $\text{gfp}(\Gamma_P^2)$, respectively. Informally, the atoms in the least fixed point are those that must be true, while those not in the greatest fixed point must be false. Formally, if P is a logic program without constraints, then its *well-founded model* is the set of literals

$$\text{lfp}(\Gamma_P^2) \cup \overline{Atoms(P) \setminus \text{gfp}(\Gamma_P^2)}.$$

As we mentioned, if a literal is in the well-founded model, then it is satisfied by every answer set. Of course, a logic program may not have any answer sets.

It is not very clear how the well-founded semantics can be extended to logic programs with constraints. One possibility is to take it to be the output of the $Expand$ operator of smodels as the operator outputs the well-founded model if the given program has no constraints, see (Baral 2003).

Another way of providing a semantics to logic programs is to transform them to theories in classical logic. This starts with Clark's (1978) completion semantics which says that an atom is true if and only if there is a rule with this atom as its head such that the body of this rule is true. Formally, given a program P without constraints, the completion of an atom p is the disjunction of the bodies of all rules in P that have p as their head. If there is no such rule, then the completion of p is $\neg p$. The Clark's completion of P is then the set of completions of all atoms in P. If P has constraints, then the completion of P is the union of Clark's completion and the set of sentences corresponding to the constraints in P: if $\leftarrow p_1, ..., p_n, not\ q_1, ..., not\ q_m$ is a constraint, then its corresponding sentence is $\neg(p_1 \wedge \cdots \wedge p_n \wedge \neg q_1 \wedge \cdots \wedge \neg q_m)$.

As we mentioned, we will convert the completion into a set of clauses. Since we are going to use inference rules that may not be logically complete, it matters how we convert it. In the following, let $comp(P)$ be the set of following clauses:

1. for each $a \in Atoms(P)$, if there is no rule in P with a as its head, then add $\neg a$;
2. if r is a proper rule, then add $head(r) \vee \bigvee \overline{body(r)}$;
3. if r is a constraint, then add the clause $\bigvee \overline{body(r)}$;
4. if a is an atom and $r_1, ..., r_n$, $n > 0$, are all the rules in P with a as their head, then introduce n new variables $v_1, ..., v_n$, and add the following clauses:

$$\neg a \vee v_1 \vee \cdots \vee v_n,$$
$$v_i \vee \bigvee \overline{body(r_i)}, \text{ for each } 1 \leq i \leq n,$$
$$\neg v_i \vee l, \text{ for each } l \in body(r_i) \text{ and } 1 \leq i \leq n.$$

Clearly, the size of $comp(P)$ is linear in the size of P.

It is known that if A is an answer set of P, then A is also a propositional model of the completion of P, but the converse may not be true in general as the completion semantics does not handle cycles. For instance, if the only rule about p in a program is $p \leftarrow p$, then the completion of p is $p \equiv p$, which is a tautology. But according to the answer set semantics p is false as one should not prove p by assuming p. To address this problem with the completion semantics, Lin and Zhao (2004) proposed adding so-called loop formulas to the completion of a program and show that the resulting propositional theory coincides with the answer set semantics.

To define loop formulas, we have to define loops, which are defined in terms of positive dependency graphs. Given a logic program P, the *positive dependency graph* of P, written G_P, is the directed graph whose vertices are atoms in P, and there is an arc from p to q if there is a rule $r \in P$ such that $p = head(r)$ and $q \in body^+(r)$. A set L of atoms is said to be a loop of P if for any p and q in L, there is a non-empty path from p to q in G_P such that all the vertices in the path are in L, i.e. the L-induced subgraph of G_P is strongly connected.

Given a loop, Lin and Zhao defined a formula which says that an atom in the loop cannot be proved by the atoms in the loop only. Thus atoms in the loop can be proved only by using some atoms and rules that are "outside" of the loop. Formally, a rule r is called an *external support* of a loop L if $head(r) \in L$ and $L \cap body^+(r) = \emptyset$. In the following, let $R^-(L)$ be the set of external support rules of L. Then the *loop formula* of L under P, written $LF(L, P)$, is the following implication:

$$\bigvee_{p \in L} p \supset \bigvee_{r \in R^-(L)} body(r), \tag{2}$$

where we have abused the notation and use $body(r)$ also for the conjunction of the literals in it.

Lin and Zhao showed that a set of atoms is an answer set of a logic program iff it is a propositional model of the completion and the loop formulas of all loops of the program. Thus the logical consequences of the program completion and loop formulas are identical to the consequences of the logic program. In particular, all literals in the well-founded model of a program are logical consequences of the program completion and loop formulas. However, checking logical entailment is coNP-complete, and in general, a program may have an exponential number of loops (Lifschitz & Razborov 2006). So to use program completion and loop formulas to derive consequences of a logic program efficiently, we have to use an efficient rule of inference such as unit propagation.

Given a set Γ of clauses, we let $UP(\Gamma)$ be the set of literals that can be derived from Γ by unit propagation. Formally, it can be defined as follows:

Function $UP(\Gamma)$

if $(\emptyset \in \Gamma)$ **then return** Lit;
$A := unit\_clause(\Gamma)$;
if A is inconsistent **then return** Lit;
if $A \neq \emptyset$ **then return** $A \cup UP(assign(A, \Gamma))$
else return $\emptyset$;

where $unit\_clause(\Gamma)$ returns the union of all unit clauses in Γ, and $assign(A, \Gamma)$ is

$$\{c \mid \text{ for some } c' \in \Gamma, c' \cap A = \emptyset, \text{ and } c = c' \setminus \overline{A}\}.$$

Loops with at Most One External Support

Consider first loops without any external support rules. If a loop L has no external support rules, i.e. $R^-(L) = \emptyset$, then its loop formula (2) is equivalent to $\bigwedge_{p \in L} \neg p$. More generally, if we already know that X is a set of literals that are true in all answer sets, then if X entails $\neg body(r)$, written $X \models \neg body(r)$, for every rule $r \in R^-(L)$, then under X, the loop formula of L is equivalent to $\bigwedge_{p \in L} \neg p$, where $X \models \neg body(r)$ means that the complement of one of the literals in $body(r)$ is in X.

Thus we extend the notion of external support rules, and have it conditioned on a given set of literals.

Let P be a logic program, and X a set of literals. We say that a rule $r \in R^-(L)$ is an *external support of L under X* if $X \not\models \neg body(r)$. In the following, we denote by $R^-(L, X)$

the set of external support rules of L under X. Now given a logic program P and a set X of literals, let

$$loop_0(P,X) = \{\neg a \mid a \in L \text{ for a loop } L \text{ of } P \text{ such that } R^-(L,X) = \emptyset\}.$$

Then $loop_0(P,X)$ is equivalent to the set of loop formulas of the loops that do not have any external support rules under X. In particular, the set of loop formulas of the loops that do not have any external support rules is equivalent to $loop_0(P,\emptyset)$.

Similarly, we can consider the set of loop formulas of the loops that have exactly one external support rule under a set X of literals:

$$loop_1(P,X) = \{\neg a \vee l \mid a \in L,\ l \in body(r),\ \text{for some loop } L \text{ and rule } r \text{ such that } R^-(L,X) = \{r\}\}$$

In particular, $loop_1(P,\emptyset)$ is equivalent to the set of loop formulas of the loops that have exactly one external support rule in P.

We now consider how to compute $loop_0(P,X)$ and $loop_1(P,X)$. We start with $loop_0(P,X)$. Let $ML_0(P,X)$ be the set of maximal loops that do not have any external support rules under X: a loop is in $ML_0(P,X)$ if it is a loop of P such that $R^-(L,X) = \emptyset$, and there does not exist any other such loop L' such that $L \subset L'$. Clearly,

$$loop_0(P,X) = \bigcup_{L \in ML_0(P,X)} \overline{L}.$$

Proposition 1 *Let P be a logic program and X a set of literals. If L_1 and L_2 are two loops of P that do not have any external support rules under X, and $L_1 \cap L_2 \neq \emptyset$, then $L_1 \cup L_2$ is also a loop of P that does not have any external support rules under X.*

Proof: Suppose otherwise, and let $r \in R^-(L_1 \cup L_2, X)$. Then $head(r) \in L_1 \cup L_2$, $body^+(r) \cap (L_1 \cup L_2) = \emptyset$, and $X \not\models \neg body(r)$. Thus either $head(r) \in L_1$ or $head(r) \in L_2$. So either $r \in R^-(L_1, X)$ or $r \in R^-(L_2, X)$, a contradiction with the assumption that L_1 and L_2 do not have any external support rules under X. ■

Thus for any P, and $X \subseteq Lit(P)$, loops in $ML_0(P,X)$ are pair-wise disjoint. This means that there can only be at most $|Atoms(P)|$ loops in $ML_0(P,X)$.

To compute $ML_0(P,X)$, consider G_P, the positive dependency graph of P. If L is a loop of P, then there must be a strongly connected component C of G_P such that $L \subseteq C$. For L to be in $ML_0(P,X)$, there are two cases: either $L = C$ and $R^-(C,X) = \emptyset$ or $L \subset C$, $R^-(C,X) \neq \emptyset$ and $R^-(L,X) = \emptyset$. If it is the latter, then for any $r \in R^-(C,X)$, it must be that $head(r) \notin L$, for otherwise, r must be in $R^-(L,X)$, a contradiction with $R^-(L,X) = \emptyset$. Thus if $R^-(C,X) \neq \emptyset$, then any subset of C that is in $ML_0(P,X)$ must also be a subset of $S = C \setminus \{head(r) \mid r \in R^-(C,X)\}$. Thus instead of G_P, we can recursively search the S induced subgraph of G_P. This motivates the following procedure for computing $ML_0(P,X)$.

$ML_0(P,X) := ML_0(P,X,Atoms(P))$;

Function $ML_0(P,X,S)$

$ML := \emptyset$; $G :=$ the S induced subgraph of G_P;

For each strongly connected component L of G:

if $R^-(L,X) = \emptyset$ **then** add L to ML **else** append $ML_0(P,X,L \setminus \{head(r) \mid r \in R^-(L,X)\})$ to ML.

return ML.

From our discussions above, the following result is immediate.

Theorem 1 *For any normal logic program P, any $X \subseteq Lit(P)$, and any $C \subseteq Atoms(P)$, the above function $ML_0(P,X,C)$ returns the following set of loops:*

$\{L \mid L \subseteq C$ is a loop of P such that $R^-(L,X) = \emptyset$, and there does not exist any other such loop L' such that $L \subset L'\}$

in $O(n^2)$, where n is the size of P as a set.

We now consider the problem of computing $loop_1(P,X)$. For any rule r, let $ML_1(P,X,r)$ be the set of maximal loops of P that have r as their only external support rule under X: L is in $ML_1(P,X,r)$ if it is a loop of P such that $R^-(L,X) = \{r\}$ and there is no other such loop L' such that $L \subset L'$. Notice that this definition is meaningful only if r is a proper rule of P. If r is a constraint, then it can never be an external support rule of any loop, thus $ML_1(P,X,r) = \emptyset$. Now let

$$ML_1(P,X) = \bigcup_{r \in P} ML_1(P,X,r).$$

It is easy to see that loops in $ML_1(P,X,r)$ are pair-wise disjoint. Thus the size of $ML_1(P,X,r)$ is bounded by $|Atoms(P)|$, and $ML_1(P,X)$ by $m|Atoms(P)|$, where m is the number of proper rules in P.

It is easy to see that $loop_1(P,X)$ is the following set:

$$\bigcup_{L \in ML_1(P,X)} \{\neg a \vee l \mid a \in L, l \in body(r), R^-(L,X) = \{r\}\}.$$

Thus to compute $loop_1(P,X)$, we need to compute only $ML_1(P,X)$, and for the latter, we only need to compute $ML_1(P,X,r)$ for all proper rules $r \in P$ such that $X \not\models \neg body(r)$. At first glance, this problem can be trivially reduced to that of computing $ML_0(P \setminus \{r\}, X)$, the set of maximal loops that do not have any external support rules under X in the program obtained from P by deleting r from it. However, while it is true that if $L \in ML_0(P \setminus \{r\}, X)$ and r is the only external support rule of L under X in P, then $L \in ML_1(P,X,r)$, the converse is not true in general.

Example 1 Consider the following logic program P:

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

It is easy to see that $ML_1(P,\emptyset,b \leftarrow c) = \{\{a,b\}\}$, but $ML_0(P \setminus \{b \leftarrow c\}, \emptyset) = \{\{a,b,c\}\}$.

We do not yet know any efficient way of computing $ML_1(P, X, r)$, but for the purpose of computing $loop_1(P, X) \cup loop_0(P, X)$, $ML_0(P \setminus \{r\}, X)$ is enough.

Proposition 2 *For any normal logic program P and a set X of literals, $loop_0(P, X)$ implies that $loop_1(P, X)$ is equivalent to the following theory*

$$\bigcup_{X \not\models \neg body(r), L \in ML_0(P \setminus \{r\}, X)} \{\neg a \vee l \mid a \in L, l \in body(r)\}. \quad (3)$$

Proof: Suppose $loop_0(P, X)$ and $loop_1(P, X)$ are true. We show that for any proper rule r of P such that $X \not\models \neg body(r)$, and any $L \in ML_0(P \setminus \{r\}, X)$,

$$\{\neg a \vee l \mid a \in L, l \in body(r)\} \quad (4)$$

is true. Firstly, L is also a loop of P, and that either $R^-(L, X) = \emptyset$ or $R^-(L, X) = \{r\}$. For the first case, $\overline{L} \subseteq loop_0(P, X)$, thus (4) is true. For the second case, (4) is contained in $loop_1(P, X)$, thus true.

Now suppose $loop_0(P, X)$ and (3) are true. We show that $loop_1(P, X)$ is true, i.e. for any proper rule $r \in P$ such that $X \not\models \neg body(r)$ and any $L \in ML_1(P, X, r)$, (4) holds. Firstly, L is a loop of $P \setminus \{r\}$ that has no external support rules under X. Thus there exists $L' \in ML_0(P \setminus \{r\}, X)$ such that $L \subseteq L'$. Now L' is also a loop of P, and w.r.t. P, either $R^-(L', X) = \emptyset$ or $R^-(L', X) = \{r\}$. In the first case, $\overline{L} \subseteq \overline{L'} \subseteq loop_0(P, X)$, thus (4) holds. In the second case, L' must be in $ML_1(P, X, r)$ as well, thus $L = L'$ and (4) holds. ■

So to summarize, to compute $loop_0(P, X) \cup loop_1(P, X)$, we first compute $ML_0(P, X)$, and then for each proper rule $r \in P$ such that $X \not\models \neg body(r)$, we compute $ML_0(P \setminus \{r\}, X)$. The worst case complexity of this procedure is $O(n^3)$, where n is the size of P. There are a lot of redundancies in this procedure as described here as the computations of $ML_0(P, X)$ and $ML_0(P \setminus \{r\}, X)$ overlap a lot. These redundancies can and should be eliminated in the actual implementation.

Computing Consequences of a Logic Program

By Lin and Zhao's theorem on loop formulas, logical consequences of $comp(P) \cup loop_0(P, X) \cup loop_1(P, X)$ are also consequences of P. Since $comp(P)$, $loop_0(P, X)$, and $loop_1(P, X)$ can all be computed in polynomial time, with a polynomial time inference rule, we thus get a polynomial time algorithm for computing some consequences of a logic program. In this paper, we consider using $UP(C)$, the unit propagation.

Consider first loops without any external support rules. With $comp(P)$, $loop_0(P, X)$, and $UP(C)$, we can do the following:

$Y := comp(P) \cup loop_0(P, \emptyset)$; $X := \emptyset$;
while $X \neq UP(Y)$ **do**
  $X := UP(Y)$; $Y := Y \cup loop_0(P, X)$;
return $X \cap Lit(P)$.

Formally, the above procedure computes the least fixed point of the following operator:

$$f(X) = UP(comp(P) \cup X \cup loop_0(P, X)) \cap Lit(P).$$

As it turns out, this least fixed point is essentially the well-founded model when the given logic normal logic program has no constraints. This means that, surprisingly perhaps, the well-founded models amount to repeatedly applying unit propagation to the program completion and loop formulas of loops that do not have any "applicable" external support rules. In general, for normal logic programs that may have constraints, the least fixed point of the above operator is essentially $Expand(P, \emptyset)$ in smodels (Simons, Niemelä, & Soininen 2002; Baral 2003). More generally, $Expand(P, A)$ corresponds to the least fixed point of the following operator:

$$U_A^P(X) = UP(comp(P) \cup A \cup X \cup loop_0(P, X)) \cap Lit(P). \quad (5)$$

Now if we add in $loop_1(P, X)$, a more powerful operator can be defined:

$$T_A^P(X) = Lit(P) \cap UP(comp(P) \cup A \cup X \cup loop_0(P, X) \cup loop_1(P, X)).$$

In the following, we denote by $T(P, A)$ the least fixed point of the operator T_A^P. Clearly, $U_A^P(X) \subseteq T_A^P(X)$, and the least fixed point of U_A^P, denoted by $U(P, A)$, is contained in $T(P, A)$, the least fixed point of T_A^P. The following example shows that the containments can be proper.

Example 2 Consider the following logic program P:

$$\begin{array}{rcl} x & \leftarrow & not\ e. \\ e & \leftarrow & not\ x. \\ n & \leftarrow & x. \\ n & \leftarrow & m. \\ m & \leftarrow & n. \\ & \leftarrow & not\ n. \end{array}$$

Clearly, $x \in T(P, \emptyset)$ but $x \notin U(P, \emptyset)$.

The following proposition says that if an answer set of P satisfies A, then it also satisfies $T(P, A)$. In particular, every literal in $T(P, \emptyset)$ is a consequence of P.

Proposition 3 *Let P be a normal logic program and A a set of literals in P. If S is an answer set of P that satisfies A, then S also satisfies $T(P, A)$.*

Notice that $T(P, A)$, the least fixed point of the operator T_A^P, can be computed by an iterative procedure like the one described in Introduction.

Expand in smodels

We mentioned that the least fixed point of our operator U_A^P defined by (5) coincides with the output of the $Exapnd$ operator used in smodels. We now make this precise. Our following presentation of the $Expand$ operator in smodels follows that in (Baral 2003).

Given a logic program P and a set A of literals, the goal of $Expand(P, A)$ is to extend A as much as possible and as efficiently as possible so that all answer sets of P that agree with A also agree with $Expand(P, A)$. It is defined in terms of two functions named $Atleast(P, A)$ and $Atmost(P, A)$. They form the lower and upper bound of what can be derived from the program P based on A in the sense that those in $Atleast(P, A)$ must be in and those not in $Atmost(P, A)$ must be out. Formally, $Expand(P, A)$ is defined to be the least fixed point of the following operator:

$$E_A^P(X) = Atleast(P, A \cup X) \cup \overline{Atoms(P) \setminus Atmost(P, A \cup X)}. \quad (6)$$

The function $Atleast(P, A)$ is defined as the least fixed point of the operator F_A^P defined as follows:

$F_1(P, X) = \{a \in Atoms(P) \mid$ there is a rule r in P such that $a = head(r)$ and $X \models body(r)\}$,

$F_2(P, X) = \{\neg a \mid a \in Atoms(P)$ and for all $r \in P$, if $a = head(r)$, then $X \models \neg body(r)\}$,

$F_3(P, X) = \{x \mid$ there exists an atom $a \in X$ such that there is only one rule r in P such that $a = head(r)$, $x \in body(r)$, and $X \not\models \neg body(r)\}$,

$F_4(P, X) = \{\bar{x} \mid$ there exists $\neg a \in X$ such that there is a rule r in P such that $a = head(r)$ and $X \cup \{x\} \models body(r)\}$,

$$F_5(P, X) = \begin{cases} Lit(P) & \text{if } X \text{ is inconsistent,} \\ \emptyset & \text{otherwise,} \end{cases}$$

$F_A^P(X) = A \cup X \cup F_1(P, X) \cup F_2(P, X) \cup F_3(P, X) \cup F_4(P, X) \cup F_5(P, X)$.

The function $Atmost(P, A)$ is defined as the least fixed point of the following operator G_A^P:

$G_A^P(X) = \{a \mid$ there is a rule $r \in P$ such that $head(r) = a$, $X \setminus A^- \models body^+(r)$, and $body^-(r) \cap A^+ = \emptyset\} \setminus A^-$,

where $A^+ = \{a \mid a$ is an atom and $a \in A\}$, and $A^- = \{a \mid a$ is an atom and $\neg a \in A\}$.

In the following, a program P is said to be *simplified* if for any $r \in P$, $head(r) \notin body^+(r) \cup body^-(r)$. Notice that any logic program is strongly equivalent to a simplified logic program: if $head(r) \in body^+(r)$, then $\{r\}$ is strongly equivalent to the empty set, thus can be safely deleted from any logic program, and if $head(r) \in body^-(r)$, then $\{r\}$ is strongly equivalent to $\{\leftarrow body(r)\}$ (cf. (Lin & Chen 2007)).

The following theorem relates $U(P, A)$ and $Expand(P, A)$. The proof is given in Appendix.

Theorem 2 *For any normal logic program P, and any $A \subseteq Lit(P)$. $Expand(P, A) \subseteq U(P, A)$. If P is a simplified, then $Expand(P, A) = U(P, A)$.*

As we mentioned, if P has no constraint, then $Expand(P, \emptyset)$ is the same as the well-founded model of P (Baral 2003). Thus for simplified logic programs, the well-founded model can be computed by a bottom-up procedure using unit propagation on sets of clauses from the program completion and the loop formulas of the loops that do not have any external support rules.

The following example shows that if P has a rule such as $p \leftarrow not\ p$, U may be stronger than $Expand$.

Example 3 Consider the following program P:

$$\begin{aligned} p &\leftarrow not\ q. \\ q &\leftarrow not\ p. \\ f &\leftarrow not\ p. \\ f &\leftarrow not\ f. \end{aligned}$$

$U(P, \emptyset) = \{f, q, \neg p\}$, but $Expand(P, \emptyset) = \emptyset$.

Some Experiments

We have implemented a program that for any given normal logic program P, it first computes $T(P, \emptyset)$, and then adds the following set of constraints:

$$\{\leftarrow \bar{l} \mid l \in T(P, \emptyset)\}$$

to P. Our implementation is available at the following URL:

http://www.cs.ust.hk/cloop/

By Proposition 3, adding the above constraints to P does not change the answer sets, and intuitively, should help ASP solvers in computing the answer sets. To verify this, we tried our program on a number of benchmarks.

First, for the normal logic programs at the First Answer Set Programming System Competition[1], $T(P, \emptyset)$ does not return anything beyond the well-founded model of P. Thus for these programs, our program does not add anything new.

Next we tried Niemelä's (1999) encoding of the HC problem, and consider graphs with the special structure as mentioned in Introduction. Specifically, we create some copies of a complete graph, and then randomly add some arcs to connect these copies into a strongly connected graph such that any HC for this graph must go through these special arcs.

Table 1 contains the running times for these programs.[2] In this table, MxN.K stands for a graph with M copies of the complete graph with N nodes: $C_1, ..., C_M$, and with exactly one arc from C_i to C_{i+1} and exactly one arc from C_{i+1} to C_i, for each $1 \leq i \leq M$ (C_{M+1} is defined to be C_1). The extension K stands for a specific way of adding these arcs. The numbers under "cmodels with T" refers to the run times (in seconds) of cmodels (version 3.74 (Giunchiglia, Lierler, & Maratea 2006)) when the results from $T(P, \emptyset)$ are added to the original program as constraints, and those under "T" are the run times of our program for computing $T(P, \emptyset)$. As can be seen, information from $T(P, \emptyset)$ makes cmodels run much faster when looking for an answer set. In addition to cmodels, we also tried smodels and DLV (Leone *et al.* 2006), which unfortunately could not terminate even for the smallest graphs considered here. We also tried clasp (Gebser *et al.* 2007), which is very fast on these programs. On

[1] http://asparagus.cs.uni-potsdam.de/contest/.

[2] Our experiments were done on a Pentium 4 3.2Ghz CPU and 1GB RAM. The reported times are in CPU seconds as reported by Linux "/usr/bin/time" command.

average it returned a solution in a few seconds. Adding literals from $T(P, \emptyset)$ makes it run a little faster, but the speedup is not worth the overhead of computing $T(P, \emptyset)$.

Problem	cmodels	cmodels with T	T
20x12.1	67.62	8.20	2.18
20x12.2	5.02	5.45	1.98
20x12.3	17.33	5.69	2.00
20x12.4	29.91	7.07	1.82
20x12.5	5.56	6.90	1.78
20x12.6	8.76	6.45	2.08
20x12.7	76.11	4.86	2.13
20x12.8	281.31	4.93	1.84
20x12.9	233.56	5.08	2.10
20x12.10	57.98	7.51	2.11
20x20.1	$>$2h	87.01	8.24
20x20.2	114.39	129.31	8.05
20x20.3	$>$2h	59.12	6.49
20x20.4	97.24	63.06	8.22
20x20.5	123.17	86.93	8.30
20x20.6	$>$2h	105.15	8.36
20x20.7	1175.61	79.48	8.23
20x20.8	171.87	76.52	8.25
20x20.9	109.07	63.41	7.95
20x20.10	$>$2h	81.42	8.26

Table 1: Run-time Data.

Conclusion

We have looked at the loops that have at most one external support rule in this paper. These loops are special in that their loop formulas are equivalent to sets of unit or binary clauses. We have considered how they, together with the program completion, can be used to deduce useful consequences of a logic program under unit propagation and in the process relate them to the well-founded semantics and the $Expand$ operator in smodels.

We think that this work opens up a line of research that deserves further study:

- We have used unit propagation as the inference rule. One could use others as long as they are "efficient" enough. For example, in addition to unit propagation, one can consider adding the following rule: infer l from $l \vee a$ and $l \vee \neg a$, or even the full resolution on binary clauses.
- We have used these loop formulas for deriving consequences of a logic program. They can of course be used in ASP solvers such as ASSAT and cmodels directly. Whether this has any benefit requires further study.
- As we mentioned, what we have done here for normal logic programs can be extended to more expressive logic programs such as disjunctive logic programs. Among other things, this may provide a new perspective on how to extend the well-founded semantics to these more expressive logic programs. For instance, there have been several competing proposals for extending the well-founded semantics to disjunctive logic programs (Lobo, Minker, & Rajasekar 1992; Przymusinski 1995; Leone, Rullo, & Scarcello 1997; Brass & Dix 1999; Wang & Zhou 2005). It would be interesting to find out which one, if any, of these corresponds to the least fixed point of our operator U_A^P when applied to disjunctive logic programs.

Acknowledgments

This work has been supported in part by the Natural Science Foundations of China under grants 60496322, 60745002, 60573009, and 60703095, and by the Hong Kong RGC CERG 616806.

References

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

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

Chen, Y.; Lin, F.; Wang, Y.; and Zhang, M. 2006. First-order loop formulas for normal logic programs. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR2006)*, 298–307.

Clark, K. L. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logic and Databases*. New York: Plenum Press. 293–322.

Ferraris, P.; Lee, J.; and Lifschitz, V. 2006. A generalization of the lin-zhao theorem. *Annals of Mathematics and Artificial Intelligence* 47(1-2):79–101.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. Conflict-driven answer set solving. In *Proceddings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, 386–392.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming (ICLP-88)*, 1070–1080.

Giunchiglia, E.; Lierler, Y.; and Maratea, M. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning* 36(4):345–377.

Lee, J., and Lifschitz, V. 2003. Loop formulas for disjunctive logic programs. In *Proceedings of the Nineteenth International Conference on Logic Programming (ICLP-03)*, 451–465.

Lee, J., and Lin, F. 2006. Loop formulas for circumscription. *Artificial Intelligence* 170(2):160–185.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* 7(3):499–562.

Leone, N.; Rullo, P.; and Scarcello, F. 1997. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation* 135(2):69–112.

Lifschitz, V., and Razborov, A. 2006. Why are there so many loop formulas? *ACM Transactions on Computational Logic* 7(2):261–268.

Lin, F., and Chen, Y. 2007. Discovering Classes of Strongly Equivalent Logic Programs. *Journal of Artificial Intelligence Research* 28:431–451.

Lin, F., and Zhao, Y. 2004. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence* 157(1-2):115–137.

Lobo, J.; Minker, J.; and Rajasekar, A. 1992. *Foundations of Disjunctive Logic Programming*. MIT Press.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3):241–273.

Przymusinski, T. 1995. Static semantics of logic programs. *Annals of Mathematics and Artificial Intelligence* 14:323–357.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181–234.

Van Gelder, A.; Ross, K.; and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM* 38(3):620–650.

Van Gelder, A. 1989. The alternating fixpoint of logic programs with negation. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS-89)*, 1–10.

Wang, K., and Zhou, L. 2005. Comparisons and computation of well-founded semantics for disjunctive logic programs. *ACM Transactions on Computational Logic* 6(2):295–327.

Proof of Theorem 2

We prove Theorem 2 by showing that for any logic program P and any set A of literals, $Expand(P, A) \subseteq U(P, A)$ (Lemma 2 below) and, for any simplified logic program P, $U(P, A) \subseteq Expand(P, A)$ (Lemma 3 below).

Before proving these two lemmas, we first prove that $\overline{Atoms(P) \setminus Atmost(P, A)}$ is equivalent to $M(P, A)$, the least fixed point of the operator M_P^A defined as follows:

$loop_0^A(P, X) = \{\neg a \mid$ there is a loop L of P such that $a \in L$ and $R^-(L, A \cup X) = \emptyset\}$,

$F_2^A(P, X) = \{\neg a \mid a \in Atoms(P)$ and for all $r \in P$, if $a = head(r)$ then $A \cup X \models \neg body(r)\}$,

$M_P^A(X) = \{\neg a \mid a \in A^-\} \cup X \cup loop_0^A(P, X) \cup F_2^A(P, X)$.

Lemma 1 *For any normal logic program P, and any $A \subseteq Lit(P)$. $M(P, A) = \overline{Atoms(P) \setminus Atmost(P, A)}$.*

Proof: Let $AM(P, A) = \overline{Atoms(P) \setminus Atmost(P, A)}$ and $X \subseteq AM(P, A)$, clearly $\overline{X} \cap Atmost(P, A) = \emptyset$. First we prove that $M(P, A) \subseteq AM(P, A)$. As $M(P, A)$ is the least fixed point of our operator M_P^A, we only need to prove that $M_P^A(X) \subseteq AM(P, A)$.

Let L be a loop of P and $R^-(L, A \cup X) = \emptyset$, clearly $L \cap Atmost(P, A \cup X) = \emptyset$. For any set of atoms Y such that $Y \subseteq Atmost(P, A)$ and $Y \subseteq Atmost(P, A \cup X)$, as $\overline{X} \cap Atmost(P, A) = \emptyset$, $Y \setminus A^- = Y \setminus (A^- \cup \overline{X})$, so $G_A^P(Y) = G_{A \cup X}^P(Y)$, hence $Atmost(P, A \cup X) = Atmost(P, A)$. Now we have $L \cap Atmost(P, A) = \emptyset$, then $\overline{loop_0^A(P, X)} \cap Atmost(P, A) = \emptyset$ and $loop_0^A(P, X) \subseteq AM(P, A)$.

Let a be an atom such that $a \in \overline{F_2^A(P, X)}$, then for all $r \in P$, if $a = head(r)$ then $A \cup X \models \neg body(r)$. Clearly, $a \notin Atmost(P, A \cup X)$ and $a \notin Atmost(P, A)$, hence, $F_2^A(P, X) \subseteq AM(P, A)$.

So $M_P^A(X) \subseteq AM(P, A)$ and $M(P, A) \subseteq AM(P, A)$. Now we prove that $AM(P, A) \subseteq M(P, A)$. Let $S = AM(P, A) \setminus M(P, A)$, we want to prove $S = \emptyset$. The sketch of the proof is: if $S \neq \emptyset$, then there exists a loop $L \subseteq \overline{S}$ and $R^-(L, M(P, A)) = \emptyset$, clearly, $\overline{L} \subseteq loop_0^A(P, M(P, A))$ and $\overline{L} \subseteq M(P, A)$, so $S \cap M(P, A) \neq \emptyset$, which conflicts to the definition of S. Now We give the detail of the proof.

For any atom a, if $\neg a \in S$, then there exists a rule r such that $head(r) = a$ and $Atmost(P, A) \setminus A^- \not\models body^+(r)$, which is equivalent to $((Atoms(P) \setminus Atmost(P, A)) \cup A^-) \cap body^+(r) \neq \emptyset$. As $Atoms(P) \setminus Atmost(P, A) = \overline{S} \cup \overline{M(P, A)}$, there exists a rule r such that $head(r) = a$ and $\overline{S} \cap body^+(r) \neq \emptyset$.

Let $G_P^{\overline{S}}$ be the $\overline{S}$ induced subgraph of G, $L = \{a \mid$ for each $b \in \overline{S}$, if there is a path from a to b in $G_P^{\overline{S}}$, then there is a path from b to a in $G_P^{\overline{S}}\}$. Now we prove that, $L \neq \emptyset$ and $R^-(L, M(P, A)) = \emptyset$.

For any atom $a \in \overline{S}$, let $H^-(a) = \{b \mid b \in \overline{S}$, there is a path from a to b and there is not any path from b to a in $G_P^{\overline{S}}\}$ and $H^+(a) = \{b \mid b \in \overline{S}$, there is a path from b to a in $G_P^{\overline{S}}\}$. If $L = \emptyset$, then for all atoms $a \in \overline{S}$, $H^-(a) \neq \emptyset$. If an atom

$b \in H^-(a)$, then $a \in H^+(b)$ and $H^-(b) \subseteq H^-(a) \setminus \{a\}$. If another atom $c \in H^-(b)$ then $H^-(c) \subseteq H^-(a) \setminus \{a, b\}$. So we can form an infinite list of atoms $\{a_1, a_2, \ldots\}$, where $a_{i+1} \in H^-(a_i)$ and $a_{i+1} \neq a_j$ $(1 \leq j \leq i, 1 \leq i \leq \infty)$. But $\overline{S}$ is finite, so $L \neq \emptyset$.

Clearly, L is a loop of P, $L \subseteq \overline{S}$, $L \neq \emptyset$, and for each external support rule r of L, $body^+(r) \cap \overline{S} = \emptyset$. Furthermore, all rules with heads belong to $\overline{S}$ are false under $M(P, A) \cup S$, so $R^-(L, M(P, A)) = \emptyset$. Then $\overline{L} \subseteq loop_0^A(P, M(P, A))$, $\overline{L} \subseteq M(P, A)$ and $\overline{L} \subseteq S$, which conflicts to the definition of S.

So $S = \emptyset$ and $M(P, A) = \overline{Atoms(P) \setminus Atmost(P, A)}$. ■

Lemma 2 *For any normal logic program P, and any $A \subseteq Lit(P)$. $Expand(P, A) \subseteq U(P, A)$.*

Proof: $Expand(P, A)$ is the least fixed point of our operator E_A^P defined by (6). For any set of literals X, such that $X \subseteq U(P, A)$, we want to prove $E_A^P(X) \subseteq U(P, A)$. First we prove that $Atleast(P, A \cup X) \subseteq U(P, A)$.

$Atleast(P, A \cup X)$ is the least fixed point of operator $F_{A \cup X}^P$. Let $Y \subseteq U(P, A)$, we consider $F_i(P, Y)$ $(1 \leq i \leq 5)$ respectively.

Let x be a literal, if $x \in F_1(P, Y)$, then there is a rule $x \leftarrow l_1, \ldots, l_n.$ and $\{l_1, \ldots, l_n\} \subseteq Y$. The corresponding clause $x \vee \bar{l}_1 \vee \cdots \vee \bar{l}_n$ is in $comp(P)$, then $x \in UP(comp(P) \cup Y)$. As $Y \subseteq U(P, A)$ and $U(P, A)$ is the least fixed point of our operator U_A^P defined by (5), $x \in UP(comp(P) \cup U(P, A))$, $x \in U(P, A)$. So $F_1(P, Y) \subseteq U(P, A)$.

If $\bar{x} \in F_2(P, Y)$, then for every rule r, if $head(r) = x$ then $body(r)$ is false under Y. The corresponding clause $\bar{x} \vee v_1 \vee v_2 \vee \cdots \vee v_n$ is in $comp(P)$ and $\neg v_1, \neg v_2, \ldots, \neg v_n \in UP(comp(P) \cup Y)$, where v_i is the new variable stands for the body of the rule. Clearly $\bar{x} \in UP(comp(P) \cup Y)$. In particular, if $n = 0$, there is a clause $\bar{x}$ in $comp(P)$, then $\bar{x} \in UP(comp(P) \cup Y)$. Similar to the prove for $F_1(P, Y)$, $F_2(P, Y) \subseteq U(P, A)$.

If $x \in F_3(P, Y)$, then there exists a clause $\neg a \vee v_1 \vee v_2 \vee \cdots \vee v_n$ in $comp(P)$, $a \in Y$ and for any $j \neq i$, $\neg v_j \in Y$, then $v_i \in UP(comp(P) \cup Y)$. As the clause $\neg v_i \vee x$ is in $comp(P)$, $x \in UP(comp(P) \cup Y)$, hence, $F_3(P, Y) \subseteq U(P, A)$.

If $\bar{x} \in F_4(P, Y)$, then there is a literal $\neg a \in Y$, a rule $a \leftarrow l_1, \ldots, l_n.$, and $\{l_1, \ldots, l_n\}$ is true under $Y \cup \{x\}$. As Y is consistent, then $\{l_1, \ldots, l_n\}$ is not true under Y, so x is equivalent to a literal l_i $(1 \leq i \leq n)$ and for any $j \neq i$, $1 \leq j \leq n$, $l_j \in Y$. The corresponding clause is $a \vee \bar{l}_1 \vee \cdots \vee \bar{l}_n$, then $\bar{x} \in UP(comp(P) \cup Y)$, so $F_4(P, Y) \subseteq U(P, A)$.

If Y is inconsistent, then $F_5(P, Y) = Lit(P)$. $U(P, A)$ is also inconsistent, then $U(P, A) = Lit(P)$, $F_5(P, Y) \subseteq U(P, A)$.

As for any $Y \subseteq U(P, A)$, $F_{A \cup X}^P(Y) \subseteq U(P, A)$, we have proved that $Atleast(P, A \cup X) \subseteq U(P, A)$.

Now we prove that $\overline{Atoms(P) \setminus Atmost(P, A)} \subseteq U(P, A)$.

From Lemma 1, $M(P, A \cup X) = \overline{Atoms(P) \setminus Atmost(P, A \cup X)}$. Clearly for any $Y \subseteq U(P, A)$, $loop_0^A(P, Y) \subseteq U(P, A)$ and $F_2^A(P, Y) \subseteq U(P, A)$, so $M_P^A(P, Y) \subseteq U(P, A)$, $M(P, A \cup X) \subseteq U(P, A)$.

So $E_A^P(X) \subseteq U(P, A)$ and $Expand(P, A) \subseteq U(P, A)$. ■

Lemma 3 *For any simplified logic program P, and any $A \subseteq Lit(P)$. $U(P, A) \subseteq Expand(P, A)$.*

Proof: $U(P, A)$ is the least fixed point of our operator U_A^P defined by (5). For any set of literals X such that $X \subseteq Expand(P, A)$, we want to prove $U_A^P(X) \subseteq Expand(P, A)$. X is consistent, if not, $Expand(P, A) = Lit(P)$ and $U_A^P(X) \subseteq Expand(P, A)$. First we prove that $UP(comp(P) \cup X) \cap Lit(P) \subseteq Expand(P, A)$.

From the definition of $comp(P)$ in Preliminaries, some new variables are introduced and there are four kinds of clauses in $comp(P)$. We consider them one by one. First we give some notions that will be used in the proof.

A set of literals $X \subseteq Lit(comp(P))$ is called a *proper submodel* of a set of clauses $comp(P)$, if for any new variable v_i which stands for the body $\{l_1, \ldots, l_m\}$, $v_i \in X$ implies $\{l_1, \ldots, l_m\} \subseteq X$ and $\neg v_i \in X$ implies there exists some j, $1 \leq j \leq m$ such that $\bar{l}_j \in X$. For any set of literals $Y \subseteq Lit(comp(P))$, $Sub(Y) = \{v_i \mid v_i \in Y, v_i$ stands for the body $\{l_1, \ldots, l_m\}$, and $\{l_1, \ldots, l_m\} \not\subseteq Y\}$.

For any set of literals $X \subseteq Lit(comp(P))$, such that X is a proper submodel of $comp(P)$ and $(X \cap Lit(P)) \subseteq Expand(P, A)$.

For type 1, the clause is $\neg a$ and $\neg a \in F_2(X \cap Lit(P))$.

For type 2, the clause is $\bar{l}_1 \vee \cdots \vee \bar{l}_n \vee l$. Assume that it can be reduced to a unit clause under X. As the rule is a simplified rule, the head of the rule does not belong to the atoms appeared in the body, then there are only two cases. If the remaining literal is l, then $\{l_1, \ldots, l_n\} \subseteq X$, so $l \in F_1(P, X \cap Lit(P))$, $l \in Expand(P, A)$. If the remaining literal is $\bar{l}_i$, then $\bar{l} \in X$, and for any $j \neq i, 1 \leq j \leq n, l_j \in X$, so $\bar{l}_i \in F_4(P, X \cap Lit(P))$, $\bar{l}_i \in Expand(P, A)$. So if the unit clause c is reduced from one of this kind of clauses, then $c \in Expand(P, A)$.

For type 3, the clauses are $\neg a \vee v_1 \vee \cdots \vee v_m$, $v_i \vee \bar{l}_{1_i} \vee \cdots \vee \bar{l}_{n_i}$, $\neg v_i \vee l_{1_i}, \ldots \neg v_i \vee l_{n_i}$ $(1 \leq i \leq m)$.

For the clause $\neg a \vee v_1 \vee \cdots \vee v_m$, assume that it can be reduced to a unit clause under X. There are only two cases. If the remaining literal is $\neg a$, then $\{\neg v_1, \ldots, \neg v_m\} \subseteq X$. As X is a proper submodel of $comp(P)$, for any rule $r \in P$ if $head(r) = a$ then $body(r)$ is false under $X \cap Lit(P)$, so $\neg a \in F_2(P, X \cap Lit(P))$, $\neg a \in Expand(P, A)$. If the remaining literal is v_i, $v_i \notin Lit(P)$, so we do not need to consider this case.

For the clause $v_i \vee \bar{l}_1 \vee \cdots \vee \bar{l}_n$, assume that it can be reduced to a unit clause under X. There are only two cases. If the remaining literal is $\bar{l}_j$, then $\neg v_i \in X$ and for any $k \neq j$, $1 \leq k \leq n$, $l_k \in X$. As X is a proper submodel of $comp(P)$, then $\bar{l}_j \in X$, so $\bar{l}_j \in Expand(P, A)$. If the remaining literal is v_i, $v_i \notin Lit(P)$.

For the clause $\neg v_i \vee l_j$, assume that it can be reduced to a unit clause under X. There are only two cases. If the remaining literal is l_j, then $v_i \in X$. As X is a proper submodel of $comp(P)$, $l_j \in X$. So $l_j \in Expand(P, A)$. If the remaining literal is $\neg v_i$, $v_i \notin Lit(P)$.

So if the unit clause c is reduced from the clause of type 3, then $\{c\} \cap Lit(P) \subseteq Expand(P, A)$.

For type 4, the clause is $\bar{l}_1 \vee \cdots \vee \bar{l}_n$. Assume that it can be reduced to a unit clause under X. If the remaining literal is $\bar{l}_i$, then for any $j \neq i, 1 \leq j \leq n, l_j \in X$, so $\bar{l}_i \in F_4(P, X \cap Lit(P))$. So if the unit clause c is reduced from one of this kind of clauses, then $c \in Expand(P, A)$.

$unit\_clause(assign(X, comp(P)))$ returns the union of unit clauses reduced from $comp(P)$ under X, we denote $UPO(comp(P), X)$ for short. So for any $X \subseteq Lit(comp(P))$, X is a proper submodel of $comp(P)$ and $X \cap Lit(P) \subseteq Expand(P, A)$, we have $UPO(comp(P), X) \cap Lit(P) \subseteq Expand(P, A)$.

Let $Y = UPO(comp(P), X)$, if Y is not a proper submodel of $comp(P)$, then there exists some $v_i \in Y$ which stands for the body $\{l_1, \ldots, l_n\}$ and for some $1 \leq j \leq n$, $l_j \notin Y$. v_i can only come from the clause $\neg a \vee v_1 \vee \cdots \vee v_m$, so $a \in Y$ and for all $k \neq i$, $1 \leq k \leq m$, $\neg v_k \in Y$, then $\{l_1, \ldots, l_n\} \subseteq F_3(P, Y \cap Lit(P))$, $UPO(comp(P), Sub(Y)) \cap Lit(P) \subseteq Expand(P, A)$. Let $Z = Y \setminus Sub(Y)$, it is clear that Z is a proper submodel of $comp(P)$, then $UPO(comp(P), Z) \cap Lit(P) \subseteq Expand(P, A)$. So $UPO(comp(P), Y) \cap Lit(P) = (UPO(comp(P), Z) \cup UPO(comp(P), Sub(Y))) \cap Lit(P) \subseteq Expand(P, A)$.

So $UP(comp(P) \cup X) \cap Lit(P) \subseteq Expand(P, A)$.

Now we prove that $loop_0(P, X) \subseteq Expand(P, A)$.

From Lemma 1, for any $Y \subseteq Expand(P, A)$, $loop_0^A(P, Y) \subseteq Expand(P, A)$, then $loop_0(P, X) \subseteq Expand(P, A)$.

So $U_A^P(X) \subseteq Expand(P, A)$ and $U(P, A) \subseteq Expand(P, A)$. ■