Useful Transformations in Answer set programming

Mauricio Osorio and **Juan Carlos Nieves**
Universidad de las Américas
CENTIA
Sta Catarina Martir, Cholula
Puebla, 72820 México
josorio@mail.udlap.mx

Chris Giannella
Department of Computer Science
Indiana University
Bloomington, IN 47405 USA
cgiannel@cs.indiana.edu

Abstract

We define a reduction system $\mathcal{CS}_3$ which preserves the stable semantics. This system includes two types of transformation rules. One type (which we call $\mathcal{CS}_2$) preserves the stable semantics regardless of the EDB (extensional database). So, it can be used at compilation time. The other (which we call $\mathcal{CS}_1$) does not preserve the stable semantics across changes to the EDB. Thus, it should be used at run time. Nonetheless $\mathcal{CS}_1$ can reduce the program size considerably and is quadratic time computable. Sometimes $\mathcal{CS}_3$ can transform a cyclic program into an acyclic one. At these times, a satisfiability solver can be used to obtain the stable models.

Introduction

Recent research (Babovich, Erdem, & Lifschitz 2000), has shown that when the stable semantics corresponds to the supported semantics, a satisfiability solver (e.g. SATO (Zhang. March 1993)) can be used to obtain stable models. Let *sys* be any system that is capable of grounding and completing a schematic program and clausifying the completion. This process, as indicated in (Babovich, Erdem, & Lifschitz 2000), can be viewed as "preprocessing" the input program. Interestingly, some examples are presented in (Babovich, Erdem, & Lifschitz 2000) where the running time of SATO is approximately ten times faster than SMODELS [1].

One of the conclusions drawn in (Babovich, Erdem, & Lifschitz 2000) is that satisfiability solvers may serve as useful computational tools in answer set programming. Our paper presents results along the same line. We define a polynomial time reduction system $\mathcal{CS}_3$ that includes two types of transformation rules. One type (which we call $\mathcal{CS}_2$) preserves the stable semantics regardless of the EDB and can be used at compilation time. The other (which we call $\mathcal{CS}_1$) does not preserve the stable semantics across changes to the EDB, so, should be used at run time. We propose to include $\mathcal{CS}_3$ as part of the preprocessing stage of *sys*.

Sometimes $\mathcal{CS}_3$ can transform a cyclic program into an acyclic one. This idea is illustrated with the following example program $Easy$:

```
x ∨ y.
x        ← y.
y        ← x.
p.
a ∨ b    ← c, ¬d.
c        ← a.
h ∨ e    ← ¬a, p.
```

This program has two stable models $\{p, h, x, y\}$ and $\{p, e, x, y\}$. The supported models of the program include the stable models but also others as well (e.g. $\{x, y, p, a, c\}$). Reducing $Easy$ by $\mathcal{CS}_3$, yields $red(Easy)$:

```
x.
y.
p.
h ∨ e.
```

One of our main results is that $\mathcal{CS}_3$ preserves the stable semantics so the set of stable models of $Easy$ is the same as that of $red(Easy)$. Since $Easy$ is acyclic, it has the same supported models.

Our paper is structured as follows. In the next section, we define the basic concepts of disjunctive logic program and the rewriting systems $\mathcal{CS}_1, \mathcal{CS}_2, \mathcal{CS}_3$. In the following section, we describe some examples where the application of $\mathcal{CS}_1$ helps in finding stable models by converting a cyclic program to an acyclic one. In the section after that, we present an algorithm for finding stable models. Finally, in last section, we give conclusions.

Background

A signature $\mathcal{L}$ is a finite set of elements that we call atoms. By $\mathcal{L}_P$ we understand it to mean the signature of P, i.e. the set of atoms that occurs in P. The language of propositional logic has an alphabet consisting of

(i) proposition symbols: $p_0, p_1, ...$

(ii) connectives : $\vee, \wedge, \leftarrow, \neg, \bot, \top$

(iii) auxiliary symbols : (,).

Where $\vee, \wedge, \leftarrow$ are 2-place connectives, $\neg$ is 1-place connectives and $\bot, \top$ are 0-place connectives. The proposition symbols and $\bot$ stand for the indecomposable propositions, which we call *atoms*, or *atomic propositions*. A literal is an atom, a, or the negation of an atom $\neg a$. Given a set of

[1] one of the leading stable model finding systems (Simons 1997)

atoms $\{a_1, ..., a_n\}$, we write $\neg\{a_1, ..., a_n\}$ to denote the set of literals $\{\neg a_1, ..., \neg a_n\}$.

A general clause, C, is denoted: $a_1 \vee \ldots \vee a_m \leftarrow l_1, \ldots, l_n$,[2] where $m \geq 0$, $n \geq 0$, each a_i is an atom, and each l_i a literal. When $n = 0$ and $m > 0$ the clause is an abbreviation of $a_1 \vee \ldots \vee a_m \leftarrow \top$[3], where $\top$ is $\neg\bot$. When $m = 0$ the clause is an abbreviation of $\bot \leftarrow l_1 \wedge \ldots \wedge l_n$[4]. Clauses of theses forms are called constraints (the rest, non-constraint clauses). Sometimes, we denote a clause C by $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$, where $\mathcal{A}$ contains all the head atoms, $\mathcal{B}^+$ contains all the positive body atoms and $\mathcal{B}^-$ contains all the negative body atoms. We also use $body(C)$ to denote $\mathcal{B}^+ \cup \neg\mathcal{B}^-$. When $\mathcal{A}$ is a singleton set, the clause can be regarded as a normal clause. A definite clause (Lloyd 1987) is a normal clause with $\mathcal{B}^- = \emptyset$.

A *pure* disjunction is a disjunction consisting solely of positive or solely of negative literals. A (general) program, P, is a finite set of clauses. As in normal programs, we use $HEAD(P)$ to denote the set of atoms occurring in the heads of P. Given a signature $\mathcal{L}$, we write $Prog_{\mathcal{L}}$ to denote the set of all programs defined over $\mathcal{L}$. We use $\models$ to denote the consequence relation for classical first-order logic. We will also consider interpretations and models as usual in classical logic.

The following defines a mapping from programs to normal programs. Given a program, P, we define $non-c(P) := \{C \in P : C\ is\ a\ non-constraint\ clause\}$. Given a non-constraint clause $C := \mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$, we write *dis-nor(C)* to denote the set of normal clauses: $\{a \leftarrow \mathcal{B}^+, \neg(\mathcal{B}^- \cup (\mathcal{A} \setminus \{a\})) | a \in \mathcal{A}\}$. We extend this definition to programs as follows. If P is a program, let *dis-nor(P)* denote the normal program:

$$\bigcup_{C \in non-c(P)} dis-nor(C).$$

Given a normal program, P, we write $Definite(P)$ to denote the definite program that is obtained from P by removing every negative literal in P. Given a definite program, P, $MM(P)$ denotes the unique minimal model of P (which always exist (Lloyd 1987)). Unless otherwise stated, we work with disjunctive programs.

The following example illustrates the above definitions. Let P be the program:

$$\begin{array}{ll} \mathtt{p} \vee \mathtt{q} & \leftarrow \neg \mathtt{r}. \\ \mathtt{p} & \leftarrow \mathtt{s}, \neg \mathtt{t}. \end{array}$$

Then $HEAD(P) = \{\mathtt{p}, \mathtt{q}\}$, and $dis-nor(P)$ consists of the clauses:

$$\begin{array}{ll} \mathtt{p} & \leftarrow \neg \mathtt{r}, \neg \mathtt{q}. \\ \mathtt{q} & \leftarrow \neg \mathtt{r}, \neg \mathtt{p}. \\ \mathtt{p} & \leftarrow \mathtt{s}, \neg \mathtt{t}. \end{array}$$

$Definite(dis-nor(P))$ consists of the clauses:

$$\begin{array}{ll} \mathtt{p}. & \\ \mathtt{q}. & \\ \mathtt{p} & \leftarrow \mathtt{s}. \end{array}$$

Finally *MM(Definite(dis-nor(P)))*=$\{\mathtt{p}, \mathtt{q}\}$.

Definition 1 (Supported model,(Brass & Dix 1997)) *A two-valued model I of a (disjunctive) logic program P is supported if and only if for every ground atom a with* $I \models a$ *there is a rule* $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ *in P with* $a \in \mathcal{A}, I \models \mathcal{B}^+, \neg\mathcal{B}^-$, *and* $I \not\models \mathcal{A} \setminus \{a\}$.

The definition of the stable semantics for disjunctive programs is well known and can be found in (Gelfond & Lifschitz 1988).

The following transformations are defined in (Brass & Dix 1997; Brewka & Dix 1996) and generalize the corresponding definitions for normal programs.

Definition 2 (Basic Transformation Rules) *A transformation rule is a binary relation on* $Prog_{\mathcal{L}}$. *The following transformation rules are called basic. Let a program* $P \in Prog_{\mathcal{L}}$ *be given.*[5]

RED$^+$: *Replace a rule* $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ *by* $\mathcal{A} \leftarrow \mathcal{B}^+, \neg(\mathcal{B}^- \cap HEAD(P))$.

RED$^-$: *Delete a clause* $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ *if there is a clause* $\mathcal{A}' \leftarrow \top$ *such that* $\mathcal{A}' \subseteq \mathcal{B}^-$.

SUB: *Delete a clause* $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ *if there is another clause* $\mathcal{A}_1 \leftarrow \mathcal{B}_1^+, \neg\mathcal{B}_1^-$ *such that* $\mathcal{A}_1 \subseteq \mathcal{A}$, $\mathcal{B}_1^+ \subseteq \mathcal{B}^+$, $\mathcal{B}_1^- \subseteq \mathcal{B}^-$.

TAUT: (Tautology) *Suppose P contains a clause of the form:* $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ *and* $\mathcal{A} \cap \mathcal{B}^+ \neq \emptyset$, *then we delete the given clause.*

Failure (F): *Suppose that P includes an atom* $a \notin HEAD(P)$ *and a clause* $q \leftarrow Body$ *such that a is a positive literal in* $Body$. *Then we erase the given clause.*

Contra (C): *Suppose that P includes a clause where a literal appears both positively and negatively in the body of the given clause. Then, we remove that clause.*

Definition 3 (Dloop(Dp), (Arrazola, Dix, & Osorio 1999)) *For a program* P_1, *let* $unf(P_1) := \mathcal{L} \setminus MM(Definite(dis-nor(P_1)))$. *The transformation* **Dloop(Dp)** *reduces a program* P_1 *to* $P_2 := \{\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^- \in P_1 | \mathcal{B}^+ \cap unf(P_1) = \emptyset\}$. *We assume that the given transformation takes place only if* $P_1 \neq P_2$.

Example 1 *After applying Dp to the program Easy described earlier, we obtain:*

$$\begin{array}{ll} \mathtt{x} \vee \mathtt{y}. & \\ \mathtt{x} & \leftarrow \mathtt{y}. \\ \mathtt{y} & \leftarrow \mathtt{x}. \\ \mathtt{p}. & \\ \mathtt{h} \vee \mathtt{e} & \leftarrow \neg \mathtt{a}, \mathtt{p}. \end{array}$$

Let **Dsuc** be the natural generalization of **suc** (Brass *et al.* 2001) to disjunctive programs, formally:

Definition 4 (Dsuc, (Arrazola, Dix, & Osorio 1999)) *Suppose that P is a program that includes a constant clause a and a clause* $\mathcal{A} \leftarrow Body$ *such that* $a \in Body$. *Then we replace this clause by the clause* $\mathcal{A} \leftarrow Body \setminus \{a\}$.

[2] $l_1, \ldots, l_n$ represents the formula $l_1 \wedge \ldots \wedge l_n$.

[3] or simply $a_1 \vee \ldots \vee a_m$

[4] In fact $\bot$ is used to define $\neg A$ as $A \rightarrow \bot$.

[5] We use $P_1 \rightarrow^T P_2$ to denote that we get P_2 from P_1 using the transformation T.

Definition 5 *Let P be a disjunctive logic program and a be an atom such that $a \in \mathcal{L}_P$. We define $P \sqcup \{\neg a\}$ as follows:*

$$P \sqcup \{\neg a\} := \{C \sqcup \{\neg a\} \mid C \in P\}$$

where $C \sqcup \{\neg a\}$ is defined as follows:

$$C \sqcup \{\neg a\} := \begin{cases} \mathcal{A} \setminus \{a\} \leftarrow \mathcal{B}^+, \neg(\mathcal{B}^- \setminus \{a\}) & \textit{if } a \notin \mathcal{B}^+ \\ \top & \textit{otherwise} \end{cases}$$

Definition 6 (W-N-A)
Let P_1 be a disjunctive logic program and a an atom such that $a \in \mathcal{L}_{P_1}$. If $P_1 \cup \{a\} \vdash_{Dsuc} b$ and $P_1 \cup \{a\} \vdash_{Dsuc} \neg b$, then the transformation W-N-A transforms P_1 to $P_2 := P_1 \sqcup \{\neg a\}$.

By $P_1 \vdash_{Dsuc} a$ we mean that $a \in P_2$ where P_1 relates to P_2 in the reflexive and transitive closure of the transformation $Dsuc$ over $Prog_{\mathcal{L}}$.

Example 2 *Let P be the following program:*

$$\begin{array}{ll} \mathtt{n} \vee \mathtt{a} & \leftarrow \neg \mathtt{m}. \\ \mathtt{m} & \leftarrow \neg \mathtt{n}, \neg \mathtt{a}. \\ \mathtt{b} & \leftarrow \mathtt{a}. \\ & \leftarrow \mathtt{a}, \mathtt{b}. \end{array}$$

Applying the transformation rule W-N-A, we get the following program:

$$\begin{array}{ll} \mathtt{n} & \leftarrow \neg \mathtt{m}. \\ \mathtt{m} & \leftarrow \neg \mathtt{n}. \end{array}$$

Definition 7 (W-EQ)
Let P_1 be a disjunctive logic program and a,b be two atoms such that $a, b \in \mathcal{L}_{P_1}$. If $P_1 \cup \{a\} \vdash_{Dsuc} b$ and $P_1 \cup \{b\} \vdash_{Dsuc} a$ then we replace every atom b in P_1 by the atom a and add the clause $b \leftarrow a$.

Example 3 *Considering the program of the example 1 and applying the transformation rule W-EQ we get the following program:*

$$\begin{array}{ll} \mathtt{x} \vee \mathtt{x}. & \\ \mathtt{x} & \leftarrow \mathtt{x}. \\ \mathtt{x} & \leftarrow \mathtt{x}. \\ \mathtt{y} & \leftarrow \mathtt{x}. \\ \mathtt{p}. & \\ \mathtt{h} \vee \mathtt{e} & \leftarrow \neg \mathtt{a}, \mathtt{p}. \end{array}$$

The clause `x` *can be substituted for* `x ∨ x.`.

Definition 8 ($\mathcal{CS}_1$,$\mathcal{CS}_2$,$\mathcal{CS}_3$)
Let $\mathcal{CS}_1$ be the rewriting system based on the transformations {SUB, RED$^+$, RED$^-$, Dp, Dsuc, Failure}. Let $\mathcal{CS}_2$ be {Contra, Taut, W-N-A, W-EQ }. Let $\mathcal{CS}_3$ be $\mathcal{CS}_1 \cup \mathcal{CS}_2$.

We do not include the well known GPPE transformation (defined in (Brass & Dix 1997)) in $\mathcal{CS}_1$ because GPPE can cause the program to grow exponentially (Brass *et al.* 2001). The following results suggest that it makes sense to reduce a program by $\mathcal{CS}_1$, because this reduction can be computed efficiently.

Example 4 *Considering the program Easy from the introduction. Applying the rewriting $\mathcal{CS}_3$ system until we can not apply more any transfirmation we get the following program:*

$$\begin{array}{l} \mathtt{x}. \\ \mathtt{y}. \\ \mathtt{p}. \\ \mathtt{h} \vee \mathtt{e}. \end{array}$$

This program is equal to the program red(Easy) from the introduction.

Lemma 1 ($\mathcal{CS}_1$ is quadratic time computable)
Let P be a program and P_1 a reduced form of P under $\mathcal{CS}_1$ (i.e. P_1 is obtained from P by a sequence of reductions from P and P_1 cannot be reduced any further by $\mathcal{CS}_1$). Then P_1 is quadratic time computable with respect to the size of P.

Proof. Dp is the most expensive reduction. Clearly $Definite(dis-nor(P))$ is obtained in linear time. Computing the minimal model of a Definite program is linear time computable and so Dp is linear time computable. Every reduction step decreases the size of the program. So, the entire process is quadratic time computable. ■

Lemma 2 ($\mathcal{CS}_2$ is cubic time computable)
Let P be a program and P_1 a reduced form of P under $\mathcal{CS}_2$. Then P_1 is cubic time computable with respect to the size of P.

Proof. Clearly *W-EQ* is the most expensive reduction. For each pair of atoms we must check whether $P_1 \cup \{a\} \vdash_{Dsuc} b$ and $P_1 \cup \{b\} \vdash_{Dsuc} a$. This check can be carried out in linear time. The desired result follows. Note that, in the algorithm that apply the trasformation rule *W-EQ* is not necesary to add the clause $b \leftarrow a$. Then *W-EQ* keeps the size of the program. ■

Lemma 3 (STABLE is closed under $\mathcal{CS}_3$)
Let P_1 and P_2 two programs related by any transformation in $\mathcal{CS}_3$. Then P_1 and P_2 have the same stable models.

Proof. By definition 8, $\mathcal{CS}_3 = \mathcal{CS}_2 \cup \mathcal{CS}_1$ and it is well known that $\mathcal{CS}_1 \setminus \{Dp\}$ is closed under stable models (see (Brewka, Dix, & Konolige 1997)). Then we only have to prove that $\mathcal{CS}_2$ and *Dp* are closed under stable models. But it is also well known that $\mathcal{CS}_2 \setminus$ { *W-N-A, W-EQ*} is closed under stable models (see (Brewka, Dix, & Konolige 1997)). Then it suffices to prove that *W-N-A, W-EQ* and *Dp* are closed under stable models.

W-N-A: Let P be a disjunctive program, and *a* a atom such that the assumptions of *W-N-A* are satisfied. Then $P \equiv_I P \cup \{\neg a\}$ and also $P \cup \{\neg a\} \equiv_I (P \sqcup \{\neg a\}) \cup \{\neg a\} \equiv_{stable} P \sqcup \{\neg a\}$. $P_1 \equiv_I P_2$ denotes that P_1 is equivalent to P_2 in intuitionistic logic and $P_1 \equiv_{stable} P_2$ denotes that *stable-models(P_1) = stable-models(P_2)*

W-EQ: If $P_1 \rightarrow^{W-EQ} P_2$. Then we have to prove that $P_1 \equiv_{stable} P_2$. By using the substitution theorem (see 5.2.5 from (van Dalen 1980)) one can prove that $P_1 \equiv_i P_2 \cup \{a \leftarrow b\}$ (equivalent under intuitionistic logic). Then $P_1 \equiv_{stable} P_2 \cup \{a \leftarrow b\}$ by (van Dalen 1980). Then by *GPPE* and *TAUT* $P_2 \cup \{a \leftarrow b\} \equiv_{stable} P_2$, finally by transitive property $P_1 \equiv_{stable} P_2$.

Dp: Let A be $unf(P_1)$. It is easy to show that for every stable model M of P_1, $M \cap A = \emptyset$. Thus $P_1 \cup \neg A \equiv_{stable} P_1$. It is easy to prove that $P_2 \cup \neg A \equiv_I P_1 \cup \neg A$ and by

(Pearce 1999) it follows that $P_2 \cup \neg A \equiv_{stable} P_1 \cup \neg A$. Hence, $P_2 \cup \neg A \equiv_{stable} P_1$. Since the atoms in A do not occur in the head of P_2, then $P_2 \cup \neg A \equiv_{stable} P_2$. Finally, $P_1 \equiv_{stable} P_2$.

■

Some applications of $\mathcal{CS}_1$ in answer set programming

The following example illustrates how $\mathcal{CS}_1$ can be used in answer set programming. Consider the program HC (a slight variant of a program in (Babovich, Erdem, & Lifschitz 2000)).

```
in(U,V) ∨ out(U,V)    ← edge(U,V).
                      ← edge(U,W), in(U,V),
                      in(U,W), V != W.
                      ← edge(V,W), in(U,W),
                      in(V,W), U != V.
reachable(V)          ← vertex(V), in(0,V),
reachable(V)          ← edge(U,V),
                      reachable(U), in(U,V).
                      ← vertex(U),
                      ¬reachable(U).
```

This program calculates the Hamiltonian cycles of a directed graph, where the graph is defined by the facts `vertex` and `edge`; 0 is assumed to be one of the vertices. The authors of (Babovich, Erdem, & Lifschitz 2000) showed that HC with the extension database E_1:={ `vertex(0), vertex(1), edge(0,0), edge(1,1)` } does not have any stable models, but has supported models (Babovich, Erdem, & Lifschitz 2000). However, instantiating[6] and reducing $HC \cup E_1$ using $\mathcal{CS}_1$ we obtain the acyclic program HC_1.

```
                      ←
                      ← ¬reachable(0).
reachable(0)          ← in(0,0).
in(1,1) ∨ out(1,1).
in(0,0) ∨ out(0,0).
```

Its stable and supported semantics correspond. Since HC_1 has no supported models, then it has no stable models.

The rule *Dp* was not required in the reduction of HC (e.g. the system $\mathcal{CS}_1 \setminus \{Dp\}$ applied to HC also yields HC_1). The following example illustrates a situation where *Dp* is required. Let E_2 be the extensional database { `v(0), v(1), v(2), v(3), edge(0,1), edge(2,3), edge(3,2)` }. By instantiating and reducing the program $HC \cup E_2$ with the transformation rules $\mathcal{CS}_1 \setminus \{Dp\}$ we get the program HC_2:

[6] We are using lparse for this purpose.

```
                      ← ¬reachable(3).
                      ← ¬reachable(2).
                      ← ¬reachable(1).
                      ← ¬reachable(0).
reachable(1)          ← in(0,1),
reachable(2)          ← reachable(3), in(3,2).
reachable(3)          ← reachable(2), in(2,3).
reachable(1)          ← reachable(0), in(0,1).
in(3,2) ∨ out(3,2).
in(2,3) ∨ out(2,3).
in(0,1) ∨ out(0,1).
```

Observe that HC_2 has no stable models but has supported models. Moreover, HC_2 has clauses with positive cycles.

Instantiating and reducing $HC \cup E_2$ with $\mathcal{CS}_1$ we get the program HC_3:

```
                      ←
                      ← ¬reachable(1).
in(3,2) ∨ out(3,2).
in(2,3) ∨ out(2,3).
in(0,1) ∨ out(0,1).
```

In this case, *Dp* eliminates the clauses causing cycles. So *Dp* removed undesirable supported models.

These examples demonstrate how the use of *Dp* can produce acyclic programs, and so helps in eliminating undesirable supported models.

Another interesting example is the *shortest path* problem:

```
const n = 30
num(0..c).
s_le(X1,Y1,C)      ← edge(X1,Y1,C).
s_le(X1,Y1,C)      ← node(X1),node(Y1),node(Z1),
                     num(C),num(C1),num(C2),
                     edge(X1,Z1,C1),
                     short(Z1,Y1,C2), C=C1+C2.
short(X,X,0)       ← node(X).
short(X,Y1,C)      ← node(X),node(Y1),
                     num(C), X != Y1,
                     s_le(X,Y1,C),
                     not s_l(X,Y1,C).
s_l(X1,Y1,S)       ← node(X1),node(Y1),
                     num(S),num(C1),
                     s_le(X1,Y1,C1), C1<S.
path(X,Y)∨         complement(X,Y) ← edge(X,Y,C).
                   ← node(X),ini(A),path(X,A).
                   ← node(X),fin(D),path(D,X).
                   ← node(X),node(Y),node(Y1),
                     path(X,Y),path(X,Y1),
                     neq(Y,Y1).
                   ← node(Y),node(X),node(X1),
                     path(X,Y),path(X1,Y),
                     neq(X,X1).
r(X)               ← ini(X).
r(X)               ← num(C),node(X),node(Y),
                     r(Y),path(Y,X).
k(X)               ← node(X),node(Y),path(X,Y).
k(Y)               ← node(X),node(Y),path(X,Y).
                   ← node(D),k(D),notr(D).
                   ← fin(D),notr(D).
cost(X,Y,C)        ← node(X),node(Y),num(C),
                     path(X,Y), edge(X,Y,C).
cost(X,Y,C)        ← node(X),node(Y),node(Z),
                     num(C),num(C1),
                     num(C2),path(X,Z),
                     edge(X,Z,C1),
                     cost(Z,Y,C2),
                     C = C1 + C2.
                   ← num(C),num(C1),ini(A),
                     fin(D), cost(A,D,C),
                     short(A,D,C1), C > C1.
```

Considering the EDB :=

$$\{\mathtt{edge}(1,2,1), \mathtt{edge}(1,3,2), \mathtt{edge}(2,3,1), \mathtt{edge}(3,1,1)\}$$

the size of the instantiated program is 5110 atoms, while the size of the reduced program (after applying $\mathcal{CS}_1$) is 812 atoms. Moreover, the reduced program is acyclic.

Now we present some experimental results using normal programs. In order to use SATO, it was necessary to get the clausal form of the program after finding the Clark's completion.[7] For this, we used Wilson's method, which has linear time complexity (Wilson 1990). We considered the well known *queens-n* problem of placing *n* queens on a chessboard so that none are attacked. The following program $Queens$ models the problem.

```
const n=15.
pc(1..n).
d(I,J)             ← pc(I),pc(J),¬otro(I,J).
otro(I,J)          ← pc(I),pc(J),pc(J1),
                     ¬ig(J,J1),d(I,J1).
ig(X,X)            ← pc(X).
                   ← pc(I),pc(J),pc(I1),neq(I,I1),
                     d(I,J),d(I1,J).
                   ← pc(I),pc(J),pc(I1),pc(J1),
                     d(I,J),d(I1,J1),diag(I,J,I1,J1).
diag(I,J,I1,J1)    ← pc(I),pc(J),pc(I1),pc(J1),
                     pc(K),I1 = I + K,J1 = J + K.
diag(I,J,I1,J1)    ← pc(I),pc(J),pc(I1),pc(J1),
                     pc(K),I1 = I + K,J1 = J − K.
```

This program is acyclic, so, SATO can be used (after completing the program). With n = 15 (i.e. 15 queens) the run time of SATO was of 3.54 seconds and the run time of SMODELS was of 11.80 seconds. With n = 17 the run time of SATO was of 7.60 seconds and the run time of SMODELS was of 97.80 seconds.[8].

An Algorithm for finding stable models

As previously pointed out, SATO can sometimes be used to find stable models in a much faster way than SMODELS. Therefore it makes sense to consider an approach that first attempts to convert a cyclic program into an acyclic one. Moreover, it is also helpful to reduce the cyclic program as much as possible. We are therefore interested in transformations which preserve the set of supported models. The transformation *By-Cases* is useful in this respect.

Definition 9 (By-Cases (B-C),(Nieves & Cervantes 2000)) *Let P be a normal logic program. P_2 result from P if the following condition holds. Suppose b is an atom. Let $P_3 := \{a \leftarrow \mathcal{B}^+, \neg(\mathcal{B}^- \setminus \{b\}) \mid a \leftarrow \mathcal{B}^+, \neg\mathcal{B}^- \in P\}$ and $P_4 := \{a \leftarrow \mathcal{B}^+ \setminus \{b\}, \neg\mathcal{B}^- \mid a \leftarrow \mathcal{B}^+, \neg\mathcal{B}^- \in P\}$. Let P_3' and P_4' programs resulting from P_3 and P_4 respectively by applying $Dsuc^*$ and let $H := \{p | \ p \in P_3' \cap P_4'\}$ Then the transformation $By-Cases$ derives $P \cup \{a\}$ where $a \in H$ and $a \neq b$. In order to emphasis the role of a, b then we write $By-Cases^a_b$.* [9]

Lemma 4
The transformation rule By-Cases is closed under supported models.

Proof.
Straightforward. ■

The transformation rule By-Cases is not closed under Stable Models Semantics. Let P be the following program:

```
a ← b.    a ← ¬b.    b ← a.
a ← ¬c.   c ← ¬d.    d ← ¬c.
```

P has only one stable model ($\{\mathtt{d}, \mathtt{a}, \mathtt{b}\}$). Apply By-cases, we get P':

```
a.
a ← b.    a ← ¬b.    b ← a.
a ← ¬c.   c ← ¬d.    d ← ¬c.
```

P' has two stable models ($\{\mathtt{d}, \mathtt{a}, \mathtt{b}\}, \{\mathtt{c}, \mathtt{a}, \mathtt{b}\}$).

[7] Clark's completion is a characterization of supported models.

[8] All tests were conducted on a Sun sparc station 5.

[9] T^* denotes the reflexive and transitive closure of the relation T.

We propose the following algorithm for computing stable models. We first "compile" the program by applying transformations that preserve the semantics regardless of the extensional database (the input in ASP). In our case, we use $\mathcal{CS}_2$ (this transformation may be applied over a (not yet) grounded program). Let $P_{compiled} := res_{\mathcal{CS}_2}(P)$, where $P_{compiled}$ is the input program to the function Stable(P). We also obtain the dependency graph of the program. At run time, we instantiate the program and proceed as follows:

Function Stable(P)
  $P_2 := res_{\mathcal{CS}_1}(P)$.
  If(HEDLP(P_2))
  {
    P_{D-N} := *dis-nor(P_2).*
    If(ACYCLIC(P_{D-N}))
      return(cmodels(P_{D-N})).
    Else
      return(SMODELS(P_{D-N})).
  }
  Else
    return(Disjunctive-Stable(P_2)).

HEDLP(P_2) is a function that determines whether the program P_2 is head-cycle free (Ben-Eliyahu & Dechter 1992). If so, Stable-models(P_2) = Stable-models(dis-nor(P_2)). The function *ACYCLIC(P_{D-N})* determines whether the normal program P_{D_N} is acyclic (Ben-Eliyahu & Dechter 1992) [10]. The function *SMODELS* computes a stable model of a normal program or returns *false* if none exist (Simons 1997). The function *cmodels* is given below. The function *Disjunctive-Stable* returns the set of stable models of a disjunctive program. We can use the system **dlv** for this purpose.

Function cmodels(P)
  $P_1 := res_{\mathcal{CS}_1} \cup \{B - C\}(P)$.
  P_2 := *Claus-Comp(P_1).*
  return(SATO(P_2)).

The function Claus-Comp produces the clausal form after completing the program. For this, Wilson's method ((Wilson 1990)) can be used. The function SATO returns a model for P_2 if one exists, otherwise returns *false*. It is based on the well known Davis Putnam procedure.

Conclusion

We defined a reduction system $\mathcal{CS}_3$ that includes several transformation rules that are correct with respect to the stable semantics. We illustrated how sometimes $\mathcal{CS}_3$ can transform a cyclic program into an acyclic one. Our results emphasize that satisfiability solvers may serve as useful computational tools in answer set programming.

Acknowledgments

We would like to thank Michael Gelfond for very helpful discussions in a preliminary version of our paper.

[10] These programs are also called tight in (Fages 1993).

References

Arrazola, J.; Dix, J.; and Osorio, M. 1999. Confluent term rewriting systems for non-monotonic reasoning. *Computacion y Sistemas* II(2-3):299–324.

Babovich, Y.; Erdem, E.; and Lifschitz, V. 2000. Fages' theorem and answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*. ?–?

Ben-Eliyahu, R., and Dechter, R. 1992. Propositional Semantics for Disjunctive Logic Programs. In Apt, K. R., ed., *LOGIC PROGRAMMING: Proceedings of the 1992 Joint International Conference and Symposium*, 813–827. Cambridge, Mass.: MIT Press.

Brass, S., and Dix, J. 1997. Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. *Journal of Logic Programming* 32(3):207–228. (Extended abstract appeared in: Characterizations of the Stable Semantics by Partial Evaluation *LPNMR, Proceedings of the Third International Conference, Kentucky*, pages 85–98, 1995. LNCS 928, Springer.).

Brass, S.; Dix, J.; Freitag, B.; and Zukowski, U. 2001. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming* to appear.

Brewka, G., and Dix, J. 1996. Knowledge representation with logic programs. Technical report, Tutorial Notes of the 12th European Conference on Artificial Intelligence (ECAI '96). Also appeared as Technical Report 15/96, Dept. of CS of the University of Koblenz-Landau. Will appear as Chapter 6 in *Handbook of Philosophical Logic*, 2nd edition (1998), Volume 6, Methodologies.

Brewka, G.; Dix, J.; and Konolige, K. 1997. *Nonmonotonic Reasoning: An Overview*. CSLI Lecture Notes 73. Stanford, CA: CSLI Publications.

Fages, F. 1993. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science* 2.

Gelfond, M., and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In Kowalski, R., and Bowen, K., eds., *5th Conference on Logic Programming*, 1070–1080. MIT Press.

Lloyd, J. W. 1987. *Foundations of Logic Programming*. Berlin: Springer. 2nd edition.

Nieves, J. C., and Cervantes, G. 2000. Is the class of well-behaved semantics so small? In *Proceedings of 12th European Summer School in Logic, Language and Information*, 189–198.

Pearce, D. 1999. Stable inference as intuitionistic validity. *Logic Programming* 38:79–91.

Simons, P. 1997. Towards constraint satisfaction through logic programs and the stable model semantics. Technical Report 47, Helsinki University of Technology, Digital Systems Laboratory.

van Dalen, D. 1980. *Logic and Structure*. Berlin: Springer, second edition.

Wilson, J. 1990. Compact normal forms in propositional logic and integer programming formulations. *Journal of Computers and Operations Research* 90:309–314.

Zhang., H. March 1993. Sato: A decision procedure for propositional logic. *Association for Automated Reasoning Newsletter* 22:1–3.