Handling Negation in General Deductive Databases: A Program Transformation Method

Weiling Li, Komal Khabya, Ming Fang, and Rajshekhar Sunderraman

Department of Computer Science
Georgia State University
Atlanta, Georgia 30303, U.S.A
email: {wli16,kkhabya1,mfang1}@student.gsu.edu, raj@cs.gsu.edu

Abstract

In this paper, we present a program transformation approach to compute the stable models of general deductive databases. Stable models have important applications in knowledge representation, database integration and repairs as well as in efficiently solving NP-complete problems. The method introduced in the paper first transforms the given deductive database with arbitrary negation into a "semantically" equivalent deductive database with "limited" negation. It then computes the "Fitting" model (a weaker form of the well-founded model) in a bottom-up manner. The Fitting model is a unique 3-valued model and the proposed method uses this model as a starting point to generate-and-test models for stability. Experimental analysis shows a large speed-up compared to generating the stable models from definition.

The transformation algorithm introduces two new predicates, `p_plus` and `p_minus`, for each predicate `p` in the original deductive database. These new predicates collect the positive and negative conclusions for the predicates from the database defined by the Fitting model. The transformed database does not contain the original negations as they are replaced by the plus and minus versions of the predicates. Although the transformation algorithm introduces negations, these are easily handled by traditional bottom up evaluators as these negations are always present in a limited manner.

1 Introduction

Deductive databases were introduced over 30 years ago ([5, 6, 7]) as a powerful extension to the relational data model with recursive views. A deductive database consists of a set of facts that correspond to a relational database and a set of logical rules that define new predicates that correspond to relational views. However, unlike relational databases where the views are restricted to be non-recursive, deductive databases can express recursive views. For example, the following is a deductive database expressing data related to a graph and a recursive view defining the predicate `path`:

```
edge(a,b).
edge(b,c).
edge(c,d).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Deductive rules with arbitrary *negation* in their bodies are much more expressive and can represent many views that are not possible with rules without negation or with restricted negation such as stratified negation, where negation is not allowed within a recursive view. Deductive databases with arbitrary negation in the body of rules have important applications in knowledge representation, data integration and database repairs, and many other emerging areas.

The semantics of deductive databases with negation have been studied in great detail over the past several decades and two popular semantics have emerged. These are the well founded model semantics ([17]) and the stable model semantics ([8]). These semantics are well understood but there has not been sufficient research in devising efficient methods to compute these models. In [2], a bottom-up algebraic approach to compute the Fitting model (a weaker version of the well-founded model) was proposed which utilized a paraconsistent relational model and algebra ([1]). The paraconsistent data model extends the traditional relational model with explicit negation by allowing both positive and negative facts to be stored in paraconsistent relations. The relational algebraic operators were appropriately extended to operate on paraconsistent relations.

In this paper we use some of the ideas from the paraconsistent relational model and algebra to devise a program transformation method which transforms

International Conference on Management of Data
COMAD 2010, Nagpur, India, December 8–10, 2010
©*Computer Society of India, 2010*

a deductive database with arbitrary negation into a Fitting-model equivalent deductive database in which the negation is present in a very controlled manner. In particular, the negations that are introduced in the transformed database always appear in predicates whose argument variables are constrained to take values from the domain of all values. The transformed database has several advantages including the possibility of using traditional bottom up evaluators that have been proven to be efficient in computing the desired models of the database. The bottom-up evaluator is easily able to handle the constrained negations by using the difference operator of the relational algebra.

The unique 3-valued Fitting model generated in a bottom-up manner provides the starting point for a generate-and-test procedure to compute the different stable models of the original deductive database. The "unknowns" are assigned truth values and the resulting complete model is tested for stability. Experimental results indicate a considerable speed-up compared to computing the stable models from definition.

Computing stable models have important applications in speeding up solutions to a class of NP-complete problems. NP-Datalog ([10, 9]) is a simplified version of Datalog with unstratified negation, which allows the user to express NP search and optimization problems in a simple and intuitive way. The solutions to the NP-complete problems are obtained by computing the stable models of NP-Datalog programs. We envision that the methodology proposed in this paper to compute the stable models can provide a better (and faster) alternative to solve the NP-Datalog formulations of the intrinsically difficult problems in NP search and optimization.

The paper is organized as follows: Section 2 presents some background information necessary to follow the rest of the paper, Section 3 introduces the database transformation algorithm along with examples, Section 4 presents the stable model computation approach, and Section 5 discusses the experimental results. Finally, Section 6 presents concluding remarks and possible directions for future work.

2 Background

In this section we give a brief overview of general deductive databases and two popular semantics: the Fitting model and the Stable model. For a detailed exposition the reader is referred to ([4] and [8]).

We assume an underlying language with a finite set of constant, variable, and predicate symbols, but no function symbols. A *term* is either a variable or a constant. An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and the $t_i's$ are terms. A *literal* is either a *positive literal* A or a *negative literal* $\neg A$, where A is an atom. For any literal l we let l' denote its complementary literal, i.e. if l is positive then $l' = \neg l$, otherwise $l = \neg l'$.

Definition 1: A *general deductive database* is a finite set of clauses of the form

$$a \leftarrow l_1, l_2, \ldots, l_m$$

where a is an atom, $m \geq 0$ and each l_i is a literal.

A term, atom, literal, or clause is called *ground* if it contains no variables. A *ground instance* of a term, atom, literal, or clause Q is the term, atom, literal, or clause, respectively, obtained by replacing each variable in Q by a constant. For any general deductive database DB, we let $DB^\star$ denote the set of all ground instances of clauses in DB. Note that since the underlying language has no function symbols, unlike logic programs, $DB^\star$ is always finite.

The *Herbrand Base* of the underlying language is the set of all ground atoms. Any subset of the Herbrand Base is termed a *Herbrand interpretation* (atoms in the interpretation are assumed to be true and those outside the interpretation are assumed to be false). A Herbrand interpretation is a *model* of the database if all the facts and rules evaluate to true in the interpretation. A model is a *minimal model* if none of its proper subsets is a model.

Definition 2: A *partial interpretation* is a pair $I = \langle I^+, I^- \rangle$, where I^+ and I^- are any subsets of the Herbrand Base.

2.1 Fitting Model

The Fitting model of a general deductive database DB is the least fixpoint of the immediate consequence function T^F_{DB} on consistent partial interpretations defined as follows:

Definition 3: Let I be a partial interpretation. Then $T^F_{DB}(I)$ is a partial interpretation, given by

$$\begin{aligned}
T^F_{DB}(I)^+ \;=\; \{a \mid & \text{ for some clause } a \leftarrow l_1, \ldots, l_m \text{ in } DB^\star,\\
& \text{for each } i, 1 \leq i \leq m,\\
& \text{if } l_i \text{ is positive then } l_i \in I^+, \text{ and}\\
& \text{if } l_i \text{ is negative then } l'_i \in I^-\},\\
T^F_{DB}(I)^- \;=\; \{a \mid & \text{ for each clause } a \leftarrow l_1, \ldots, l_m \text{ in } DB^\star,\\
& \text{there is some } i, 1 \leq i \leq m,\\
& \text{such that}\\
& \text{if } l_i \text{ is positive then } l_i \in I^-, \text{ and}\\
& \text{if } l_i \text{ is negative then } l'_i \in I^+\}.
\end{aligned}$$

It is obvious that T^F_{DB} is monotonic and thus possesses a least fixpoint, which is referred to as the Fitting model for DB. In other words, starting from an empty partial interpretation as the initial value of I, if we repeatedly apply the T^F_{DB} operator to the previous value of I to generate the next value of I, we are assured to reach a steady state where no more values are added to the input I in the output.

Let DB be the following general deductive database:

```
r(a,c).
r(b,b).
s(a,a).
p(X) :- r(X,Y), not p(Y).
p(Y) :- s(Y,a).
```

We start with the empty partial interpretation: $\langle\emptyset,\emptyset\rangle$. Then,

$$\begin{aligned}(T^F_{DB}(<\emptyset,\emptyset>)^+ &= \{\ \texttt{r(a,c), r(b,b), s(a,a)}\ \}\\ (T^F_{DB}<\emptyset,\emptyset>)^- &= \{\ \texttt{r(a,a), r(a,b), r(b,a),}\\ &\quad\ \texttt{r(b,c), r(c,a), r(c,b),}\\ &\quad\ \texttt{r(c,c), s(a,b), s(a,c),}\\ &\quad\ \texttt{s(b,a), s(b,b), s(b,c),}\\ &\quad\ \texttt{s(c,a), s(c,b), s(c,c)}\}\end{aligned}$$

$T^F_{DB}(T^F_P(<\emptyset,\emptyset>))^+ = I \cup T^F_{DB}(<\emptyset,\emptyset>)$, where I is the partial interpretation $\langle$ {p(a)}, {p(c)}$\rangle$. At this point we reach a steady state. Note that in the Fitting model the atom p(a) is *true* and the atom p(c) is *false*. No truth value is assigned to the atom p(b).

2.2 Stable Model

The stable model semantics was originally introduced by Gelfond and Lifschitz ([8]) as a two-valued model for general deductive databases. They defined a transformation, called the GL-transform, that converted a (ground) general deductive database into a negation-free (ground) deductive database based on a given Herbrand interpretation. The given interpretation was called stable if it coincided with the minimal model of the transformed database. The GL-transform is discussed now.

For any set S of atoms from the Herbrand base of a general deductive database DB, let DB^S be the program obtained from DB by deleting:

1. each rule with a negative literal **not** B_i in body with $B_i \in S$, and
2. all negative literals from bodies of remaining rules.

If S is a minimal model of DB^S, then S is a stable model of DB.

Consider the program

```
p(1,2).
q(x) :- p(x,y), not q(y).
```

The set of constants is {1,2} and the set of ground atoms is {q(1), q(2), p(1,1), p(1,2), p(2,1), p(2,2)}. Let δ be obtained from the program with the second rule replaced by its ground instances:

```
p(1,2).
q(1) :- p(1,1), not q(1).
q(1) :- p(1,2), not q(2).
q(2) :- p(2,1), not q(1).
q(2) :- p(2,2), not q(2).
```

Let S={q(2)}. Then δ^S is

```
p(1,2).
q(1) :- p(1,1).
q(2) :- p(2,1).
```

The minimal Herbrand model of this program is {p(1,2)}, which is different from S, thus S is not stable. Now let S={p(1,2),q(1)}. In this case, δ^S is

```
p(1,2).
q(1) :- p(1,2).
q(2) :- p(2,2).
```

The minimal Herbrand model of this program is {p(1,2), q(1)}, i.e., S. Hence {p(1,2), q(1)} is stable.

3 Database Transformation

Consider a general deductive database, DB, consisting of an extensional part EDB and an intensional part IDB. Without loss of generality, we will assume that there are no predicates common to EDB and IDB. For each predicate p of DB, we introduce two predicates p_plus and p_minus in the transformed general deductive database $tr(DB)$. p_plus will be used to collect all positive consequences of the database under predicate p. Similarly, p_minus will collect all negative consequences of p.

Example 1: We will illustrate the steps of the transformation algorithm on the following general deductive database (found in [15]):

```
%% Extensional Database
t0(1).
g(1,2,3).
g(2,5,4).
g(2,4,5).
g(5,3,6).
%% Intensional Database
t(Z) :- t0(Z). %% rule 1
t(Z) :- g(X,Y,Z), t(X). %% rule 2
t(Z) :- g(X,Y,Z), not t(Y). %% rule 3
```

This database instance with rules is inspired from an electronic circuit shown in Figure 1. In the circuit, each logic gate consists of one positive input and one negative input. The EDB predicate g(X,Y,Z) states the input-output relationship for a logic gate; here X is the positive input, Y is the negative input, and Z is the output. There is a second EDB predicate t_0 that is true of those input terminals that are set externally to 1. Input terminals that are set to 0 do not appear in t_0. The three rules define a predicate t(Z) which gives the value at a particular terminal. If a gate has either positive input X or negative input Y, then its output is 1 or "true" if and only if either X is 1 ("true") or Y is 0("false").

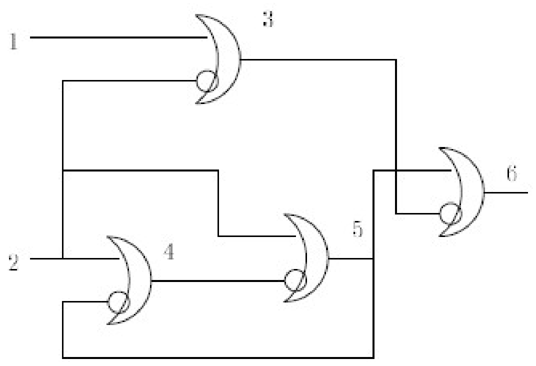

Figure 1: Circuit for Example 1 (source: [15])

3.1 Transformation Algorithm

The transformation of DB is done in four steps:

Step 1: Domain Predicate:

Introduce a unique unary predicate `dom`. For each constant symbol, `a`, present in DB, output the fact:

```
    dom(a).
```

For the example database, the following facts are produced in the output of Step 1:

```
    dom(1).
    dom(2).
    dom(3).
    dom(4).
    dom(5).
    dom(6).
```

Step 2: Extensional Database:

For each fact `p(a1,...,an)` in EDB, output the fact:

```
  p_plus(a1,...,an).
```

For each predicate p with arity k in EDB, output the rule:

```
  p_minus(X1,...,Xk) :-
    dom(X1),...,dom(Xk),
    not p_plus(X1,...,Xk).
```

For the example database, the following facts and rules are produced in the output of Step 2:

```
    t0_plus(1).
    t0_minus(X) :- dom(X), not t0_plus(X).

    g_plus(1,2,3).
    g_plus(2,5,4).
    g_plus(2,4,5).
    g_plus(5,3,6).
    g_minus(X,Y,Z) :-
      dom(X), dom(Y), dom(Z),
      not g_plus(X,Y,Z).
```

Step 3: Intensional Database:

Consider a rule in IDB of the form:

```
    p(W1,...,Wl) :-
      q1(X1), ..., qn(Xn),
      not r1(Y1), ..., not rm(Ym).
```

where `p, q1, ..., qn, r1, ..., rm` are positive predicates, `W1,..., Wl` are distinct variables, and `X1, ..., Xn, Y1, ..., Ym` are vectors of variables/constants. For each such rule, perform Steps 3a and 3b.

Step 3a. Output "plus" rule:

Output the following rule for `p_plus`:

```
    p_plus(W1,...,Wl) :-
      q1_plus(X1), ..., qn_plus(Xn),
      r1_minus(Y1), ..., rm_minus(Ym).
```

For the example database, the following plus rules will be produced:

```
    t_plus(Z) :- t0_plus(Z).
    t_plus(Z) :- g_plus(X,_,Z), t_plus(X).
    t_plus(Z) :- g_plus(_,Y,Z), t_minus(Y).
```

Step 3b. Output temporary "minus" rules:

Let `V1, ..., Vk` be the distinct variables present in the body of the rule. We shall assume that the distinct variables present in the head of the rule are also present in the body of the rule (an assumption usually made in general deductive databases to ensure safety of rules). Let `W1, ..., Wl` be the distinct variables in the head of the rule.

Step 3b-1:

For each positive subgoal in rule, `qi(Xi)`, output:

```
      temp_p_j(V1,...,Vk) :-
        dom(U1),..., dom(Ua), qi_minus(Xi).
```

where `j` is the unique rule number and {`U1,..., Ua`} is the set of variables present in {`V1,...,Vk`} but not in `Xi`.

Step 3b-2:

For each negative subgoal in rule, `not ri(Yi)`, output:

```
    temp_p_j(V1,...,Vk) :-
      dom(U1),..., dom(Ua), ri_plus(Yi).
```

where `j` is the unique rule number and {`U1,..., Ua`} is the set of variables present in {`V1,...,Vk`} but not in `Yi`.

Step 3b-3:

Output the following two rules:

```
temp_p_j_2(W1,...,Wl) :-
  dom(V1), ..., dom(Vk),
  not temp_p_j(V1,...,Vk).
p_minus_j(W1, ...,Wl) :-
  dom(W1), ..., dom(Wl),
  not temp_p_j_2(W1,...,Wl).
```

For the example database, the following temporary minus rules are generated:

```
%% rule 1
temp_t_1(Z) :- t0_minus(Z).
temp_t_1_2(Z) :-
  dom(Z), not temp_t_1(Z).
t_minus_1(Z) :-
  dom(Z), not temp_t_1_2(Z).

%% rule 2
temp_t_2(X,Y,Z) :- g_minus(X,Y,Z).
temp_t_2(X,Y,Z) :-
  dom(Y), dom(Z), t_minus(X).
temp_t_2_2(Z) :-
  dom(X), dom(Y), dom(Z),
  not temp_t_2(X,Y,Z).
t_minus_2(Z) :-
  dom(Z), not temp_t_2_2(Z).

%% rule 3
temp_t_3(X,Y,Z) :- g_minus(X,Y,Z).
temp_t_3(X,Y,Z) :-
  dom(X), dom(Z), t_plus(Y).
temp_t_3_2(Z) :-
  dom(X), dom(Y), dom(Z),
  not temp_t_3(X,Y,Z).
t_minus_3(Z) :-
  dom(Z), not temp_t_3_2(Z).
```

Step 4. Output "minus" rules:

For each IDB predicate p defined in rules numbered i1,...,in, output the following rule:

```
p_minus(W1,...,Wl) :-
  dom(W1),...,dom(Wl),
  p_minus_i1(W1,...,Wl),
  ...,
  p_minus_in(W1,...,Wl).
```

For the example database, the following minus rules are generated:

```
t_minus(Z) :-
  dom(Z),
  t_minus_1(Z),
  t_minus_2(Z),
  t_minus_3(Z).
```

This ends the description of the transformation algorithm.

A bottom-up evaluation of the output program for the example database results in the following values for t_plus and t_minus:

```
t_plus = {<1>,<3>}
t_minus = {<2>}
```

This is verified by computing the "minimal model" of the output program using bottom-up computation until a steady state is reached. It can also be easily verified that the model produced by our transformed program coincides with the Fitting model for the input program.

3.2 Discussion

The rationale behind the steps of the algorithm is discussed now. The main idea behind the transformation is motivated by the paraconsistent relational model and algebra that was used to compute the Fitting model of general deductive databases in [2]. The approach in this program transformation method is to explicitly use predicates for both positive as well as negative consequences of the database.

Step 1 of the transformation algorithm introduces dom, a unary predicate that collects all constants (elements of the Herbrand Universe).

Step 2 of the algorithm is also straightforward; it explicitly states that the facts specified in the *EDB* are positive facts and those that are missing are negative facts.

Step 3a of the algorithm is also reasonably straightforward. The "plus" component of the *IDB* predicate defined in the head of the rule is obtained by replacing positive body predicates by the corresponding "plus" predicates and the negative body predicates by the corresponding "minus" predicates. The reason is that for the positive body predicate to be true, a tuple (corresponding to the arguments of the predicate) must be present in the "plus" predicate and for the negative body predicate to be true, a tuple (corresponding to the arguments of the predicate) must be present in the "minus" predicate.

To understand Step 3b of the algorithm, we briefly present the paraconsistent model ([1]) and the relevant algebraic operators. A paraconsistent relation R is defined as a pair $\langle R^+, R^- \rangle$, where R^+ and R^- are sets of tuples in the relational schema, where tuples in R^+ denote positive facts and tuples in R^- denote negative facts. Some of the relevant algebraic operations are shown below (note the dot on top of the paraconsistent relational operator - to distinguish it from the ordinary operator):

Definition 4: Let R and S be paraconsistent relations on scheme Σ. Then,

(a) the *union* of R and S, denoted $R \mathbin{\dot\cup} S$, is a paraconsistent relation on scheme Σ, given by

$$(R \mathbin{\dot\cup} S)^+ = R^+ \cup S^+, \qquad (R \mathbin{\dot\cup} S)^- = R^- \cap S^-$$

(b) the *intersection* of R and S, denoted $R \dot{\cap} S$, is a paraconsistent relation on scheme Σ, given by

$$(R \dot{\cup} S)^+ = R^+ \cap S^+, \qquad (R \dot{\cup} S)^- = R^- \cup S^-$$

(c) the *complement* of R, denoted $\dot{-}\, R$, is a paraconsistent relation on scheme Σ, given by

$$(\dot{-}\, R)^+ = R^-, \qquad (\dot{-}\, R)^- = R^+$$

If Σ and Δ are relation schemes such that $\Sigma \subseteq \Delta$, then for any tuple $t \in \tau(\Sigma)$, we let t^Δ denote the set $\{t' \in \tau(\Delta) \mid t'(A) = t(A), \text{for all } A \in \Sigma\}$ of all extensions of t. We extend this notion for any $T \subseteq \tau(\Sigma)$ by defining $T^\Delta = \cup_{t \in T}\, t^\Delta$.

Definition 5: Let R and S be partial relations on schemes Σ and Δ, respectively. Then, the *natural join* (or just *join*) of R and S, denoted $R \dot{\bowtie} S$, is a partial relation on scheme $\Sigma \cup \Delta$, given by

$$(R \dot{\bowtie} S)^+ = R^+ \bowtie S^+,$$

$$(R \dot{\bowtie} S)^- = (R^-)^{\Sigma \cup \Delta} \cup (S^-)^{\Sigma \cup \Delta},$$

where $\bowtie$ is the usual natural join among ordinary relations.

It is instructive to observe that $(R \dot{\bowtie} S)^-$ contains all extensions of tuples in R^- and S^-, because at least one of R and S is believed false for these extended tuples.

Definition 6: Let R be a paraconsistent relation on scheme Σ, and $\Delta \subset \Sigma$ be any scheme. Then, the *projection* of R onto Δ, denoted $\dot{\pi}_\Delta(R)$, is a paraconsistent relation on Δ, given by

$$\dot{\pi}_\Delta(R)^+ = \pi_\Delta(R^+),$$

$$\dot{\pi}_\Delta(R)^- = \{t \in \tau(\Delta) \mid t^\Sigma \subseteq R^-\}$$

where π_Δ is the usual projection over Δ of ordinary relations.

Now, we return to the discussion on Step 3b of the algorithm.

The first point to note is that the negative component of the "join" of paraconsistent relations corresponds to a union of extension of tuples from each of its operands. This is the reason behind producing a separate rule (using temp predicates) for each of the positive (Step 3b-1) and negative (Step 3b-2) predicates in the body of the rules, thereby implementing the "union". It should also be noted that tuples are extended to the full schema of the body by adding `dom(X)` in the body of the temp rules for each `X` that is not present in the predicate.

Step 3b-3 of the algorithm can be justified by looking at the "projection" operator on paraconsistent relations. Usually, in the bottom up methods to compute immediate consequences of the deductive database, the projection operator is used to remove the unnecessary variables from the body relation. In the definition of the negative component of the projection operator, we notice a forall-quantifier being used (implicitly in the $\subseteq$ expression). In essence, the definition states that if a particular value (for the projected variables) is associated with "all" values from the domain for the remaining variables of the body, the particular value is kept in the negative part of the projection. To implement the forall-quantifier using deductive rules, we employ a two-rule strategy with limited negations in each as seen in Step 3b-3. This is the only step in which negations are introduced in the transformed database, an important point to note. Otherwise the rest of the transformed program is negation-free. Furthermore, the negation introduced is a controlled one, i.e. it is present next to a universe of values obtained by the cartesian product of the domains. This type of negation is easily handled by bottom up evaluators.

Step 4 of the algorithm relies on the "union" operation of the paraconsistent algebra. To obtain the negative component of the output of the union of two paraconsistent relations, the intersection of the negative components is taken. The output rule in Step 4 indicates this intersection.

Here is another complete example of the transformation algorithm:

Example 2: Consider the following general deductive database:

```
r(1,2).
r(2,3).
r(3,4).
p(X,Y) :- r(X,Y), not q(Y). %% rule 1
q(X) :- r(Y,X), not p(X,Y). %% rule 2
```

The transformed database is:

```
%% Output of Step 1:
dom(1).
dom(2).
dom(3).
dom(4).

%% Output of Step 2:
r_plus(1,2).
r_plus(2,3).
r_plus(3,4).
r_minus(X1,X2) :-
  dom(X1), dom(X2), not r_plus(X1,X2).

%% Output of Step 3:
p_plus(X,Y) :- r_plus(X,Y), q_minus(Y).
q_plus(X) :- r_plus(Y,X), p_minus(X,Y).

temp_p_1(X,Y) :- r_minus(X,Y).
temp_p_1(X,Y) :- dom(X), q_plus(Y).
temp_p_1_2(X,Y) :-
  dom(X), dom(Y), not temp_p_1(X,Y).
p_minus_1(X,Y) :-
  dom(X), dom(Y), not temp_p_1_2(X,Y).
```

```
temp_q_2(X,Y) :- r_minus(Y,X).
temp_q_2(X,Y) :- p_plus(X,Y).
temp_q_2_2(X) :-
  dom(X), dom(Y), not temp_q_2(X,Y).
q_minus_2(X) :-
  dom(X), not temp_q_2_2(X).

%% Output of Step 4
p_minus(X,Y) :-
  dom(X), dom(Y), p_minus_1(X,Y).
q_minus(X) :-
  dom(X), q_minus_2(X).
```

A bottom-up evaluation of the output program results in the following values for p_plus, p_minus, q_plus, and q_minus:

```
p_plus = {}
p_minus =
{<1,1><1,2>,<1,3>,<1,4>,
 <1,1><1,2>,<1,3>,<1,4>,
 <1,1><1,2>,<1,3>,<1,4>,
 <1,1><1,2>,<1,3>,<1,4>}
q_plus = {<2>,<3>,<4>}
q_minus = {<1>}
```

which coincides with the Fitting model for the input program.

The following theorem establishes the correctness of the algorithm:

Theorem 1: Let DB be a general deductive database and let $tr(DB)$ be the output of the transformation algorithm. Then,

1. $tr(DB)$ has a unique and complete well-founded model.
2. $p(a1, ..., an)$ belongs to the positive component of the Fitting model of DB if and only if $p\_plus(a1, ..., an)$ belongs to the well-founded model of $tr(DB)$.
3. $p(a1, ..., an)$ belongs to the negative component of the Fitting model of DB if and only if $p\_minus(a1, ..., an)$ belongs to the well-founded model of $tr(DB)$.

All the variables in the negated predicates of the transformed database are always constrained to be elements of dom. This allows bottom-up computation of the head predicate to proceed by employing the "limited-complement" operator of the relational algebra. The net result is that the bottom-up computation reaches a steady state in a complete model, which coincides with the well-founded model of the transformed database. The transformed database can be shown to be modularly stratified ([13]), which allows us to employ more efficient methods to compute the well-founded model of the transformed database.

4 Stable Model Computation

Recently, the stable models of general deductive databases have been shown to be useful in speeding up solutions to many NP-complete problems in graph theory ([9, 10]). Therefore, computing the stable models in an efficient manner is of importance. We propose a methodology that uses the database transformation described in the previous section to compute the stable models of a general deductive database. This approach will be shown to be substantially faster than the naive approach to computing the stable models that uses the basic definition of the Gelfond-Lifschitz transform.

4.1 Naive Approach

Here, we summarize the naive approach for computing the stable models shown in Figure 2. This method uses the Gelfond-Lifschitz([8]) transformation DB^S of DB with respect to S, where DB is the original deductive database and S is a candidate Herbrand model. The deductive database is first compiled using a Datalog compiler (coded using Java JCup/JFlex) and a data structure is produced that contains the essence of the facts and rules. The main loop is a generate-and-test loop in which a candidate Herbrand model (S) is generated using the data structure as input. Next, the data structure and the candidate Herbrand model are used as inputs to produce a ground instance of the deductive database. Useless rules are eliminated in the process to minimize the size of the ground instance. The candidate model is then tested for stability. To perform the stability test, the minimal model of the resulting ground deductive database is computed in a bottom-up manner and tested to see if it coincides with the candidate Herbrand model.

4.2 Database Transformation Approach

We propose to use the transformation algorithm and the subsequent bottom up evaluation of the Fitting model as the preprocessing steps to computing the stable models. Figure 3 explains the modified approach. The "Generate Candidate Model" module is now preceded by the "Transformation" and the "Fitting Model". The extra time spent in the preprocessing step that computes the Fitting model is offset by the big reduction in the number of candidate models generated.

4.2.1 Transformation Module

This module uses the database transformation algorithm mentioned in Section 3.1. We introduce unknown values via rules of the form:

```
p_unknown(X1,...,Xk) :-
  dom(X1),..., dom(Xk),
  not p_plus(X1,...,Xk),
  not p_minus(X1,...,Xk).
```

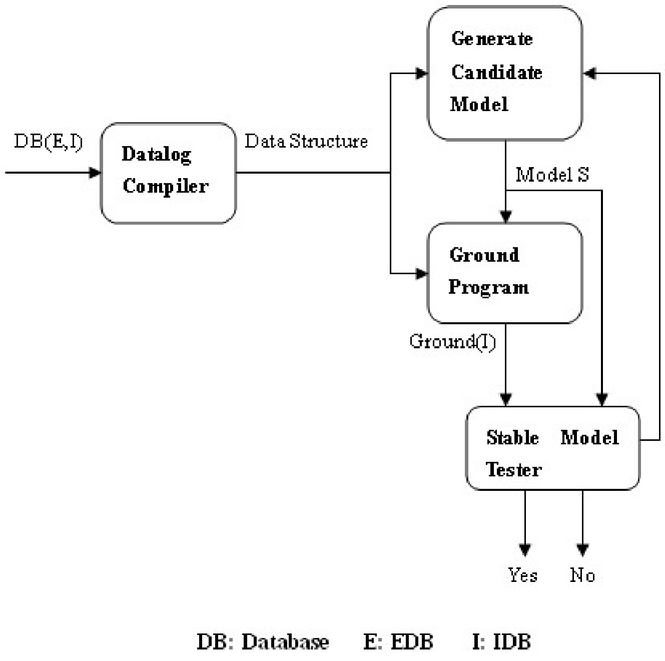

Figure 2: Naive approach of computing the stable models

for each IDB predicate.

For the example 1, the following unknown rules are generated:

```
t_unknown(Z) :-
  dom(Z),
  not t_plus(Z),
  not t_minus(Z).
```

The output program for the example 1 database results in the following values for `t_unknown`:

```
t_unknown = {<4>,<5>,<6>}
```

It coincides with the meaning of the circuit in Figure 1.

4.2.2 Fitting Model Module

After we have generated the deductive database by the transformation algorithm we use the Fixpoint semantics and apply the Fixpoint operator T_{DB} on the transformed deductive database. The application of T^F_{DB} gives us the Fitting model for the deductive database.

4.2.3 Generate Candidate Models Module

This module is essentially the same as before, except that it generates the candidate models by systematically "completing" the potentially incomplete Fitting model. The models for stability testing are generated from the unknown and positive values of the Fitting model. Because unknowns can be positive or negative,

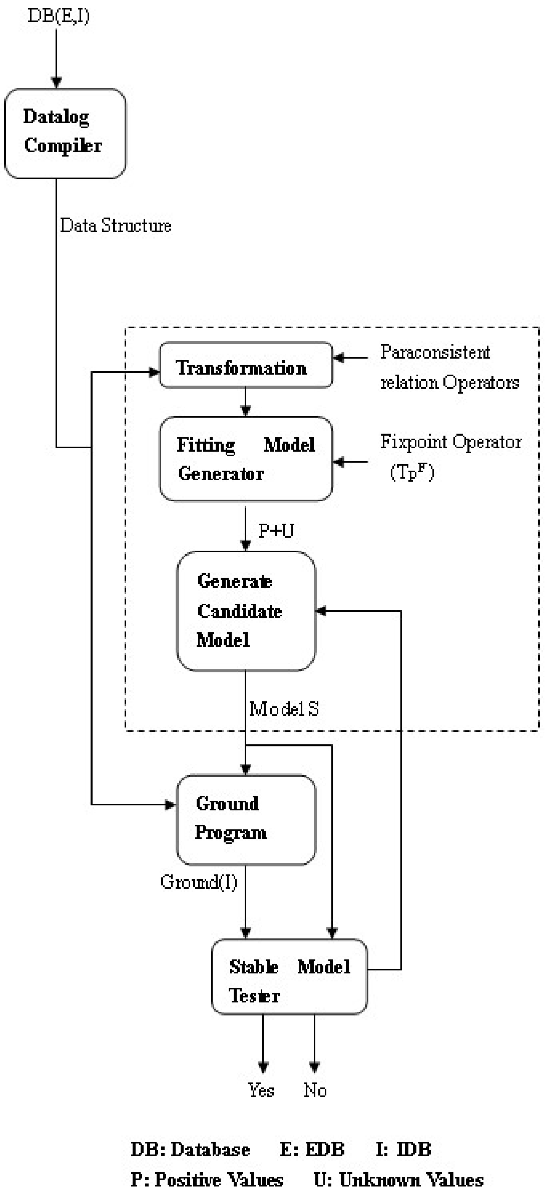

Figure 3: Our approach with the preprocessing steps

these unknowns can be systematically placed in the "plus" part of the model and the resulting complete model can be tested for stability.

5 Experiments

We present the experiments performed to test the efficiency of our architecture. We perform three experiments to compare the time taken to compute the stable models using our proposed architecture and a naive method of stable model computation. We use the IDB from Example 1 we discussed above with various EDBs as our logic program.

```
%%generate EDB facts of t
%%generate EDB facts of g

t(Z) :- t0(Z).
t(Z) :- g(X,Y,Z), t(X).
t(Z) :- g(X,Y,Z), not t(Y).
```

Note that the facts in the EDB would be generated randomly from constant values in the experiments. In the experiments we keep vary the following parameters:

1. number of constants (#constants).
2. size of EDB (#facts = the number of t0_facts (#t0_facts) + the number of g facts (#g_facts))
3. the percentage of minus values (minus%) in the total number of t values

We use tables as well as graphs to show the results.

5.1 Experiment 1

Given the IDB rules we keep #t0_facts fixed to 2 and #g_facts fixed to 10, and vary the number of constants present in the program in increments of 1, starting from 4 and going up to 9.

Table 1: Results from Experiment 1 (time: ms)

#constants	Our Approach	Naive Approach
4	10735	4234
5	26125	13625
6	26249	41719
7	88688	132562
8	60952	374219
9	203421	1052047

The results of experiment 1, seen in Table 1 and Figure 4, show that our approach performs better than the Naive approach in case of larger number of constants. The number of stable models tested using n input values is 2^n, where each model contains the EDB facts of the original deductive database, which are always true. In the Naive approach $2^{number\_of\_constants}$ models are tested, while computing the Fitting model in

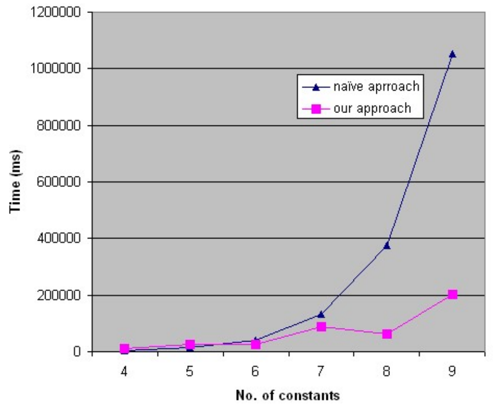

Figure 4: Naive approach vs. our approach with variable number of constants

our approach drastically reduces the possible models for testing as we only consider the set of positive and unknown values. We can see a exponential growth for the Naive approach in Figure 3. Also we can see that in case of smaller data i.e. with 4 constants Naive approach performs better than our approach because of the overhead of using Fitting model as a preprocessing mechanism. But, for larger data i.e. even with 6 constants our approach performs much better. In the results of experiment 1 we found that the time taken for #constants = 7 is larger than that for #constants = 8, because the number of minus input values we delete during preprocessing steps also affects the running time. We will discuss more about this in experiment 3.

5.2 Experiment 2

Given the IDB rules we keep #constants fixed to 7 and vary #facts, in increments of 2, starting from 10 and going up to 20.

Table 2: Results from Experiment 2 (time: ms)

#facts	Our Approach	Naive Approach
10	53859	130125
12	36672	129282
14	59720	133562
16	54891	130406
18	94266	132390
20	99032	143843

The results of experiment 2, seen in Table 2 and Figure 5, show that as we increase the number of facts in our logic program the time taken to compute the stable models increases a little bit for the Naive approach. The percentage of minus values that will be discussed in experiment 3 affects the time taken more than the number of facts.

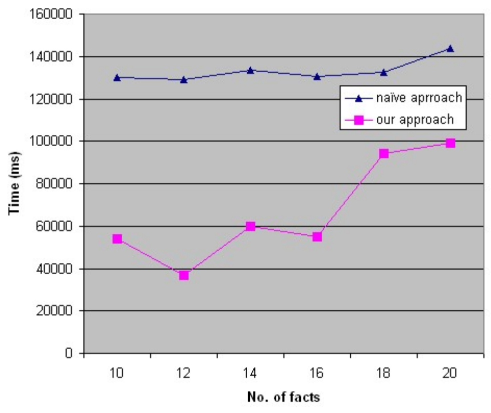

Figure 5: Naive approach vs. our approach with variable number of facts

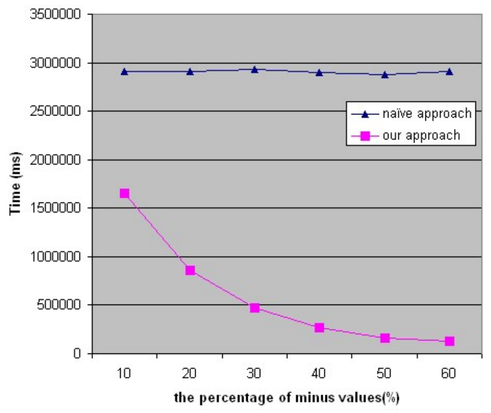

Figure 6: Naive approach vs. our approach with different percentages of minus values

5.3 Experiment 3

Given the IDB rules we keep #constants fixed to 10, #t0_facts fixed to 1, and #g_facts fixed to 15, then we check how the percent of minus values affects the running time.

Table 3: Results from Experiment 3 (time: ms)

minus%	Our Approach	Naive Approach
10	1651750	2905656
20	856078	2906328
30	474890	2932235
40	272187	2898391
50	156671	2878047
60	133188	2905781

As mentioned above, the time taken is related with the number of stable models tested, which is 2^n with n input values. The results of experiment 3 seen in Table 3 and Figure 6 show that in case of a larger percentage of minus value, our approach becomes much better than the naive approach.

From these three experiments discussed above, we can conclude that the time taken for preprocessing steps in our approach has a significant impact on the overall time to compute the stable models. In addition, when we can delete more minus values in the preprocessing steps using the Fitting model, the performance of our approach becomes even more significant.

6 Conclusion

In this paper, we have introduced a database transformation algorithm to eliminate arbitrary negations in general deductive databases and at the same time retain the meaning of the deductive database with respect to the Fitting model. The transformation enables us to use traditional bottom up evaluators for computing the meaning of general deductive databases and allows for query processing in the presence of arbitrary negations in rules. We have also shown that the database transformation method can be effectively used in computing the stable models of general deductive databases.

The stable model computation defined in Section 4 generates Herbrand instantiation ground program using the Gelfond-Lifschitz transformation in the middle of the process (shown in Figure 2). However, this is a very costly operation since many irrelevant rule instances are produced. Instead we would like to construct a gound instantiation containing only relevent rule instances called intelligent grounding ([14]). Ground instances of the example in Section 2.2 can be modified as follows:

```
p(1,2).
q(1) :- p(1,2), not q(2).
```

Assume that the number of the set of constants is n, and the number of constants is $2n$. Since the second rule has two different variables, its Herbrand instantiation contains $(2n)^2$ ground instances of the second rule. While using intelligent grounding, it has only n ground instances of the second rule. Thus, intelligent grounding can be one of efficient strategies in stable model computation in the future work.

In future work, we propose to extend the algorithm to work with well-founded models instead of the weaker Fitting models and compare efficiency with the improved alternating-fixpoint method ([16]). Also, the approach will be extended to disjunctive deductive databases ([11, 12]).

References

[1] R. Bagai and R. Sunderraman. A paraconsistent relational data model. *International Journal of Computer Mathematics*, 55(3), 1995.

[2] Rajiv Bagai and Rajshekhar Sunderraman. Bottom-up computation of the fitting model for

general deductive databases. *J. Int. Information Systems*, 6(1):59–75, 1996.

[3] The DLV Project: A Disjunctive Datalog System and more. http://www.dbai.tuwien.ac.at/research/project/dlv/ Most recently accessed: 14 March, 2010.

[4] M. Fitting. A Kripke-Kleene semantics for logic programs. *J. of Logic Programming*, 4:295–312, 1985.

[5] H. Gallaire and J. Minker. *Logic and Data Bases.* Plenum Press, New York, 1978.

[6] H. Gallaire, J. Minker, and J.M. Nicolas. *Advances in Data Base Theory*, Plenum Press, New York, 1981.

[7] H. Gallaire, J. Minker, and J.M. Nicolas. Logic and databases : A deductive approach. *ACM Computing Surveys*, 16(2):151–184, June 1984.

[8] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming, Seattle, Washington*, pages 1070–1080. IEEE, 1988.

[9] Cristian Molinaro Sergio Greco and Irina Trubitsyna. Implementation and experimentation of the logic language np datalog. In *Proceedings of the 2006 International Conference on Database and Expert Systems Applications (DEXA)*, pages 622–633, 2006.

[10] Irina Trubitsyna Sergio Greco, Cristian Molinaro and Ester Zumpano. NP Satalog: A logic language for expressing NP search and optimization problems. *Theory and Practice of Logic Programming (TPLP)*, 10(2):125–166, 2010.

[11] J. Minker and A. Rajasekar. A fixpoint semantics for disjunctive logic programs. *Journal of Logic Programming*, 9:45–74, 1990.

[12] Teodor C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424, 1991.

[13] Kenneth A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *Proceedings of the ninth ACM SIGACT-SIGART symposium on Pronciples of Database Systems (PODS)*, pages 161-171, 1990.

[14] Stefan B., Jurgen D., Burkhard F., and Ulrich Z. Transformation-Based Bottom-Up Computation of the Well-Founded Model. In *Non-Monotonic Extensions of Logic Programming*, 2001.

[15] J. Ullman. Assigning an appropriate meaning to database logic with negation. In *Computers as Our Better Partners*, p 216–225. World Scientific Press, 1994.

[16] Ulrich Z., Burkhard F., Stefan B. Improving the Alternating Fixpoint: The Transformation Approach. In *4th International Conference on Logic Programming And Nonmonotonic Reasoning*, 1997.

[17] Allen van Gelder, Kenneth A. Ross, and John S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the Seventh Annual ACM Symposium on Principles of database systems*, pages 221–230. ACM, 1988.