Summer 7-15-2010

A Novel Stable Model Computation Approach for General Dedcutive Databases

Komal Khabya
Georgia State University

A NOVEL STABLE MODEL COMPUTATION APPROACH FOR GENERAL DEDUCTIVE DATABASES

by

KOMAL KHABYA

Under the Direction of Rajshekhar Sunderraman

ABSTRACT

The aim of this thesis is to develop faster method for stable model computation of non-stratified logic programs and study its efficiency. It focuses mainly on the stable model and weak well founded semantics of logic programs. We propose an approach to compute stable models by where we first transform the logic program using paraconsistent relational model, then we compute the weak-well founded model which is used to generate a set of models consisting of the true and unknown values, which are tested for stability. We perform some experiments to test the efficiency of our approach which incurs overhead to eliminate negative values against a Naïve method of stable model computation.

INDEX WORDS*:* Logic programming, Stable model, Fitting’s model.

A NOVEL STABLE MODEL COMPUTATION APPROACH FOR GENERAL DEDUCTIVE DATABASES

by

KOMAL KHABYA

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2010

A NOVEL STABLE MODEL COMPUTATION APPROACH FOR GENERAL DEDUCTIVE DATABASES

by

KOMAL KHABYA

ACKNOWLEDGEMENTS

I would like to thank my parents and my family for being the constant source of encouragement during my life. Their motivation and support have always been there for me.

I am indebted to my advisor Dr. Rajshekhar Sunderraman for teaching me about the interesting field of deductive databases and logic programming. His motivational words and the guidance throughout have helped me achieve my goals. His ability to work hard and patience inspires me and would keep inspiring other students. He has been a friend, philosopher and guide through all the good and bad days. He has given me confidence and made me grow into a better person. I would also like to thank my committee members Dr. Sushil Prasad and Dr. Yanqing Zhang for giving their valuable time and assessing my work.

I am thankful to God Almighty for giving me the strength and for being kind to me at every step, and enlighten me to take the right course of path.

I am also thankful to my friends here at Georgia State who were there for me to share my problems and help me in every way they can. Last but not the least; I am thankful to my husband for being there for me and supporting me throughout this time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Deductive databases and logic programming have been widely recognized as expressive knowledge representation formalisms. One can draw inferences firstly based on a database and secondly by applying a set of rules to infer more information based on the information in the database. For example it is given that *man (Socrates)* and *mortal(X):- man(X)* then based on the information that Socrates is man and applying the rule we get *mortal (Socrates)*. According to closed world assumption if certain fact is not derivable from the database with any of the inference rules it is assumed to be false, for example if the database has no other rules like *man(Thor)* then infer ¬*man(Thor)*. Negation as failure is not true in classical logic, but it is an assumption made in traditional databases, i.e. if database does not contain information that Socrates is manger of Department of Sales then assumes he is not, but what if the information is not yet available then the appropriate answer would be unknown.

There has been a continuing research on the correct semantics of logic programs. The idea of using first order predicate logic as a programming language was introduced by van Emden and Kowalski in [1]. In this paper they provide semantics for class of logic programs called the Horn programs. A number of extensions were found to be necessary in order to gain expressivity. Initially, the Horn logic programs were extended to include negation in the body of rules. Clark [2] proposed a notion of a completion of a logic program, a notion developed further by Shepherdson [3, 4]. Fitting [5] and Kunen [6] developed this into the 3-valued theory. These are some of the semantics that have emerged as being the most widely accepted by research community which gave more uniform semantics by interpreting the program completion in 3-valued constructive logic. The third truth value is ‘unknown’. These semantics are the weak-well founded [5], well founded model [7] and the stable model semantics [8]. Research has shown that these semantics have higher expressible power than some of the other semantics mentioned above.

This thesis focuses mainly on the stable model and weak well founded semantics of the logic programs. It has been motivated by our efforts to develop faster algorithms to compute stable models. The outline of the thesis is as follows. Chapter 2 is an introduction to deductive databases and logic programming focusing mainly on definite logic programs and its semantics. In Chapter 3 we introduce negation in logic programs, its types and the 3-valued semantics of general logic programs i.e. the weak well founded, well founded and stable model semantics. Chapter 4 goes over the *paraconsistent* data model, a data model based on the open world assumption. We introduce an algorithm for transforming the logic program consisting of harmful negation into harmless negation using paraconsistent data model. In part 2 we introduce our approach for faster stable model computation. In chapter 5, we propose our approach for stable model computation which goes through the assumptions and the actual processes involved. We first transform the original logic program into another logic program using paraconsistent data model. Then we compute the Fitting's model of the program that gives us the true, false and unknown values. Using the true and unknown values we generate possible sets of models that are tested for stability. We also generate stable models using the Naïve approach. In chapter 6, we go over the experiments conducted to compare the results and efficiency of our approach i.e., using Fitting's model with a Naïve method. The time taken to compute the stable models is taken under consideration, and the efficiency of the two methods is compared. The results show that Fitting's approach of stable model computation is much faster than the Naïve approach.

Part I

BACKGROUND

CHAPTER 2
DEDUCTIVE DATABASES AND LOGIC PROGRAMMING

2.1 Introduction

In recent years deductive databases have been an area of intense research which has brought dramatic advances in the field of theory, systems and applications. A salient feature of deductive databases is their capability of supporting a declarative, rule-based style of expressing queries and applications on database. Relational databases, which are lacking in built-in reasoning capabilities, have also demonstrated the desirability of using a declarative logic-based language. Therefore deductive databases provide a declarative, logic based language for expressing queries, reasoning and complex applications on databases [23].

Research on deductive databases has also contributed to areas such as non-monotonic reasoning and knowledge representation by extending the declarative semantics of Horn Clauses (based on the concepts minimal model and least-fixpoint [10, 11]) to non-monotonic constructs such as negation and sets. Concepts, such as stratification [9], well-founded models, and stable models have shed new light on various aspects of non-monotonic reasoning and knowledge representation, and have also provided formal semantics to seemingly unrelated concepts such as non-determinism [12]. Many of these theoretical contributions had a practical impact, current deductive database systems provide efficient support for stratified negation; work is progressing on finding efficient ways to support more powerful semantics (e.g., well-founded models).

A deductive database is commonly viewed as a general logic program. A general logic program is a set of rules that have both negative and positive subgoals. The rules in deductive database consist of EDB (extensional database) rules known as facts that sit above the IDB (intentional database) rules. The IDB rules are evaluated using the EDB in a recursive manner to give the meaning of the program. The example taken from [13], shown next consists of three EDB rules and two IDB rules.

Example 2.1.1

t0(2).

g(2, 3, 4).

g(3 ,4, 5).

g(5, 1, 3).

$t(Z) \leftarrow t0(Z).$

$t(Z) \leftarrow g(X, Y, Z), t(X), \text{not } t(Y).$

Before going into details of the semantics of general logic programs, we go through the background of logic programs without negation and its semantics.

2.2 Logic Programs

Logic programs have emerged as a very expressive tool for knowledge representation. It is programming by description which uses logic to represent knowledge and uses deduction to solve problems by deriving logical consequences. We introduce the basic concepts of logic programming and focus mainly on the declarative semantics of logic programs. These semantics include the model theoretic and fix point semantics. The reader is referred to [11] for a more detailed description of operational semantics of logic programs.

2.2.1 Definite Logic Programs

We now introduce definite logic programs that are logic programs without negation. A definite logic program is a set of Horn Clauses. Before defining Horn clause we look into some basic structures.

A *term* is either a variable or an expression *f(t1, t2,... tn)* where f is the *function symbol* and t_i are *terms*. *Constants* are 0-ary function symbols. An *atom* of the language is of the form *P(t1, t2,…,tn)* or negation of *P(t1, t2,…,tn)* where *P* is the *predicate* symbol with finite arity $n \geq 0$ and *t1,…, tn* are terms. A *literal* is either an atom or its negation denoted by *p(t1, t2…, tn).*

A definite logic program is a set of *rules* of the form

$$A \leftarrow B1, B2\ \ldots, Bn$$

Where *A, B1, B2,…, Bn* are *atoms*. Here A is called the *head* or conclusion of the rule and conjunction of $B1 \wedge B2 \wedge \ldots \wedge Bn$ is called the *body* or premise of the *rule*. We now describe the model theoretic and fixpoint semantics of logic programs.

2.2.2 Model Theoretic Semantics

This is the declarative semantics of the logic program that describes the meaning of a logic program in terms of the set of models of the program viewed as a logical theory. To determine the set of models of a logic program, we can use the work of Herbrand, who showed how to define models from given theories, and showed that any consistent theory always has a model, which is denumerable. This is the theory’s *Herbrand model*. To determine the *Herbrand model*, we first construct the *Herbrand universe* of the logic program. For a logic program *P* the *Herbrand universe* U_p is the set of all possible ground terms constructed recursively using the constants and function symbols occurring in the program *P*. A term, atom, literal, rule or program is *ground* if it is free of variables. A *ground instance* of a rule is obtained by replacing the variables in a program with elements from U_p in every possible way. A *ground program* is the union of the ground instances of the rules in the program. An example of *Herbrand universe* of a logic program is shown below.

Example 2.2.2

Consider a logic program as

natural_number(0).

natural_number (s(X)) ← *natural_number(X).*

Here the set of constants is {*0*} and set of function symbols is {*s*}. Thus the *Herbrand universe* is {*0, s(0), s(s(0)),…….*}. If there are no function symbols we get a finite *Herbrand universe*. So, the *Herbrand universe* is the set of all possible terms that the theory can make assertions about. The *Herbrand base* of *P*, denoted by HB_p, is the set of all possible ground atoms whose predicate symbols occur in *P* and whose arguments are elements of $U_{p.}$ For example for the *Herbrand base* of above program is

{*natural_number(0), natural_number(s(0)), natural_number(s(s(0))), … }*

Herbrand Interpretation I of *P* is any subset of the *Herbrand base* of *P*. It is an assignment of truth or falsity to each element of *Herbrand base.* For the natural number example the entire *Herbrand base* must be assigned true. A Herbrand interpretation simultaneously associates, with every n-ary predicate symbol in *P*, a unique n-ary relation over U_P.

1. A ground atomic formula A is true in a Herbrand interpretation I iff $A \in I$.
2. A ground negative literal $\neg A$ is true in iff $A \notin I$.
3. A ground clause $L_1 \vee L_2 \vee \ldots \vee L_m$ is true in I iff at least one literal L_i, is true in I.
4. In general a clause C is true in I iff every ground instance $C\sigma$ of C is true in I. (Cσ is obtained by replacing every occurrence of a variable in C by a term in U_P. Different occurrences of the same variable are replaced by the same term.)
5. A set of clauses **A** is true in I iff each clause in **A** is true in I.

A literal, clause, or set of clauses is false in *I iff* it is not true. If **A** is true in *I*, then we say that *I* is a Herbrand model of **A**. Let **M(A)** be the set of all Herbrand models of **A**; then ∩**M(A)**, the intersection of all Herbrand models of A, is itself a Herbrand interpretation of **A**. This holds for any set of clauses **A** even if **A** is inconsistent. If **A** is a consistent set of Horn clauses then ∩**M(A)** is itself a Herbrand model of **A**. More generally, Horn clauses have the model intersection property: If **L** is any nonempty set of Herbrand models of **A** then ∩**L** is also a model of **A**, and is the least such model of **A** which are the declarative semantics of logic programs. For details refer to [1].

2.2.3 Fix-Point Semantics

The least model semantics provide logic based declarative definition of the meaning of a program. We need to now consider constructive semantics and effective means to realize the minimal model semantics. A constructive semantics follows from viewing the rules as constructive derivation patterns, whereby, from the tuples that satisfy the patterns specified by the goals in a rule, we construct the corresponding head atoms. For a positive program P, it is customary to consider the mapping T_P called the *Immediate Consequence Operator*, for P, defined as follows:

$$T_P(I) = \{A \mid A :- B_1, B_2, \ldots, B_n \in ground(P) \text{ and } \{B_1, B_2, \ldots, B_n\} \subseteq I\}$$

$T_P(I)$ contains a ground atomic formula $A \in HBp$ iff for some ground instance $C\sigma$ of a clause C $\in P$, $C\sigma = A \leftarrow B_1, B_2, \ldots, B_n \in ground(P)$ and $\{B_1, B_2, \ldots, B_n\} \subseteq I$, $n \geq 0$. Thus T_P is a mapping from Herbrand Interpretations of P to Herbrand Interpretations of P.

The least fixpoint computation amounts to an iterative procedure, where partial results are added to a relation until steady state is reached. The least fixpoint of T_P is the least model of P. This result relies on the fact that T_P is monotonic and hence posses a least fixpoint. T_P is monotonic means for any interpretations $I_1$and I_2 such that $I_1 \subseteq I_2$ then $T_P(I_1) \subseteq T_P(I_2)$. The least fixpoint is given by:

$$\cap\{I: T_P(I) \subseteq I\}$$

For a definite logic program **P** let **M(P)** be its Herbrand Models and let $\cap$**M(P)** be its least model. Let **C(P)** be set all interpretations closed under T_P, i.e., I ϵ **C(P)** iff $T_P(I) \subseteq I$. We need to show that $\cap$**M(P)** = $\cap$**C(P)**. It is easier to show that **M(P)** = **C(P).**

Theorem 1.2.1 *If P is a definite logic program then M(P) = C(P), i.e.* $|=_I P$ *iff* $T(I) \subseteq I$*, for all Herbrand Interpretation I of P.*

Proof. ($|=_I P$ implies $T_P(I) \subseteq I$). Suppose I is a model of P, we want to show that if $A \in T_P(I)$ then $A \in I$. Assume that $A \in T_P(I)$. Then by definition of T_P there is a clause $C \in P$ such that $C\sigma = A \leftarrow B_1, B_2, \ldots, B_n$, where $B_1, B_2, \ldots, B_n \in I$. Since I is a model of P, $C\sigma$ is true in I which means that A is true in I since $\neg B_1, \neg B_2, \ldots, \neg B_n$ are false in I. Therefore $A \in I$.

(T_P $(I) \subseteq I$ implies $\models_I P$). Suppose that I is not a model of P. Then for some clause $C \in P$, $C\sigma = A \leftarrow B_1, B_2, \ldots, B_n$ is false in I, i.e., $B_1, B_2, \ldots, B_n \in I$ and $A \notin I$. But by definition of T_P, since $B_1, B_2, \ldots, B_n \in I$, $A \in T_P(I)$. Thus $T_P \nsubseteq I$.

It can also be shown that the least model is the limit of the increasing, possibly infinite sequence of iterations $\emptyset$, $T_P(\emptyset)$, $T_P(T_P(\emptyset)), \ldots$. There is a standard notation used to denote elements of the sequence of interpretations constructed for P. Namely:

$$T_P \uparrow 0 = \emptyset$$

$$T_P \uparrow \mathrm{i}+1 = T_P\ (T_P \uparrow \mathrm{i})$$

$$T_P \uparrow \omega = \bigcup_{i=0}^{\infty} Tp \uparrow i$$

We show the iterations of T_P operator with an example.

Example 2.2.2 Consider the definite logic program

odd (s(0)).

odd (s(s(X))) :- odd(X).

$$\mathrm{T_P} \uparrow 0 = \emptyset$$

$$T_P \uparrow 1 = \{\mathrm{odd(s(0))}\}$$

$$\vdots$$

$$T_P \uparrow \omega = \{\mathrm{odd}\ (\mathrm{s}^{\mathrm{n}}(0) \mid \mathrm{n} \in \{1, 3, 5, \ldots\}\}$$

In conclusion the least fix point approach and least model approach assign the same meaning to a positive logic program.

CHAPTER 3

NEGATION

3.1 Introduction

In this chapter we describe some of the results in extending Horn clause programs to include negation in the body of clauses. Such logic programs are called *general logic programs* or *normal logic programs*. A *normal logic program* is a finite set of *normal clauses.* A normal clause is a rule of the form:

$$A \leftarrow B_1, B_2 \ldots B_n, \sim C_1 \ldots \sim C_m$$

where, A is an atom and $B_1, \ldots B_n$ and $C_1, \ldots C_m$ are literals. When we have a collection of Horn clauses (rules without negation) , then we know there is a unique minimal model of the program that assigns the meaning to the program. However these types of rules are often too limited in covering the expanse of queries that could be answered. But, as soon as we introduce negation in the rules there is no guarantee of a unique minimal and in fact, it is normal to have more than one minimal model. This can be illustrated from an example from [13]. There are two bus lines from, red and blue, which runs between pairs of cities. Predicate *blue(X, Y)* is true if blue line runs a bus from city X to city Y, while *red(X, Y)* has corresponding meaning for red line. The president of red line wants to find out where red has monopoly, i.e. a pair of cities such red runs bus between them, but on blue buses you cannot even travel from X to Y through a sequence of intermediate cities. Suppose the data relation for both blue and red are *blue(1, 2), red(1, 2) and red(2, 3).*

Example 3.1.1

(1) bluePath (X, Y) ← blue (X, Y).

(2) bluePath (X, Y) ← bluePath (X, Z), bluePath (Z, Y).

(3) monopoly (X, Y) ← red (X, Y) , not bluePath (X, Y).

The above program has two minimal models (A):*{bluePath (1, 2), monopoly (2, 3)}* and (B):*{bluePath (1, 2), bluePath (2, 3) , bluePath (1, 3)}*, both having the EDB rules. The first model makes sense the only *bluePath* is one that follows from Rule 1 and *monopoly* fact follows from Rule 3, but the second model makes no sense and the facts *bluePath(2, 3)* and *bluePath(1, 3)* seems to appear from nowhere. However it is also a minimal model, in that

1. When you make any substitutions of constants for X, Y and (if necessary) Z, rules (1) – (3) are true if the true ground atomic formulas are those given in (B), plus the given data.
2. If we delete one or more facts from (B), point (1) no longer holds.

But the question is which one is the intended model of the program, and the first model seems to be the intended one.

3.2 Stratified Logic

The least controversial type of negation is *stratified negation*, where there is no recursion involved in negated subgoals. This idea was arrived at by Van Gelder [14], Apt, Blair and Walker [9], and Naqvi [15]. A normal logic program *P* is stratified if there is an assignment of integers (0, 1, 2…3) to predicates *p* in *P* such that for each clause *r* in *P* the following holds. If *p* is the predicate in head of *r* and *q* is the predicate L_i in body of *r* then *stratum (p)* ≥ *stratum (q)*, if *Li* is positive, and *stratum (p)* > *stratum (q)*, if L_i is negative. Thus, example 3.1.1 is stratified as monopoly depends negatively on bluePath but bluePath does not depend at all on monopoly, i.e. there are no cycles with negation. We can draw a dependency graph for the above logic program to check whether it is stratified or not. The dependency graph which does not involve cycles with not, depicts a stratified logic program. The EDB predicates are drawn lowest while the IDB predicates are drawn higher, and if there is a rule *p :- q* then *p* is drawn above *q*. For the above logic program the dependency graph is as follows:

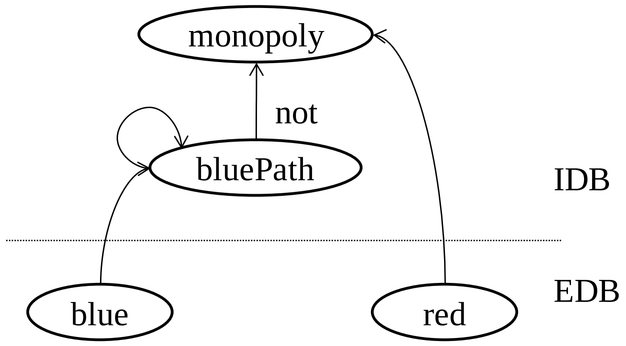


Figure 3.1 Dependency graph for example 3.1.1

So, we can compute bluePath facts completely from Rules (1) and (2) and then use Rule (3) to compute monopoly facts. This process yields the first model *{bluePath(1, 2), monopoly (2, 3)}* for the example above and confirms that this should be the intended model. The result of computing predicates this way is often known as the *perfect model* which is defined by taking least fixed point in order from lower strata to higher strata. An alternative view is *circumscription* [16], of dealing with negation that says the only facts true for predicates are those that can be followed from rules and given data. Then for example 3.1.1 we circumscribe *bluePath* and declaring those facts as true that follow from rules (1) and (2) and given *blue data*, and declaring all other pairs of X, Y for *bluePath* as false. Then these facts are used in rule (3) to assert *monopoly* facts.

The idea of stratification was extended Przymusinska and Przymusinski [17] into *locally stratified* programs. Here the predicate can negatively depend on itself, but when rules are instantiated by constants the program contains no cycles. A program can be *locally stratified* for one set of *EDB* rules and *non- stratified* for another. The semantics of *locally stratified* programs have been treated in [14, 9, 17, 18] where they give a definition of *perfect models* and have shown that locally stratified programs have one. An example of *locally stratified* program is given next, which represents a board game that says a player *wins* the game if board is on position *X* and there is a legal *move* from *X* to *Y* and *Y* is not a winning position.

Example 3.2.1

$$win(X) \leftarrow move(X,Y)\ ,\ not\ win(Y).$$

Here win depends *negatively* on itself, so it is *not stratified*. However if move is *acyclic* i.e., you can move from *X* to *Y* but there is no sequence of moves that takes you from *Y* to *X*. Then if we instantiate the rules in all possible ways, there is no way *win(a)*, for a particular board game *a*, can depend negatively on itself, Thus, *win* rule is *locally stratified* provided *move* is *acyclic*. Next, we present the semantics of non-stratified programs.

3.3 3-Valued Semantics

The landmark paper of Fitting [5] introduced semantics for logic programs with negation that was very different and gave a more uniform semantics, based on the 3-valued logic given by Kleene. The 3rd truth value, connotes *unknown* truth value, thus now an atom can be possibly *true, false, or unknown*. A principal result was that every program has a minimum 3-valued model and that according to Fitting could be taken as the semantics of the program from now on known as Fitting's semantics. Another model based on 3-valued logic, which has competing thrust to provide meaning of non-stratified program is the *well founded model* of van Gelder, ross and Schilpf [7]. We briefly describe here the Fitting's semantics and the well founded semantics.

3.3.1 Fitting's Semantics

Fitting's semantics is based on the notion of partial interpretations. We give a brief description here, the reader is referred to [5] for detailed information.

Definition 1. *A partial interpretation is a pair $\langle I^+, I^- \rangle$, where I^+ and I^- are any subsets of the Herbrand base.*

A partial interpretation is consistent if $I^+ \cap I^- = \emptyset$. For any partial interpretations I and J we let I ∩ J be the partial interpretations $\langle I^+ \cap J^+, I^- \cap J^- \rangle$, and I ∪ J be the partial interpretations $\langle I^+ \cup J^+, I^- \cup J^- \rangle$. We also say that $I \subseteq J$, whenever $I^+ \subseteq J^+$ and $I^- \subseteq J^-$. The Fitting's model for a general logic program P is the least fixed point of the immediate consequence function T_P^F on consistent partial interpretations defined as follows (let P* be the ground version of P):

Definition 2. *Let I be the partial interpretation, then T_P^F (I) is the partial interpretation given by*

$T_P^F(I^+) = \{a \mid$ *for some clause* $a \leftarrow l_1, l_2 \ldots l_m \in P^*$, *for each* $1 \leq i \leq m$

if l_i *is positive* $l_i \in I^+$ *and,*

if l_i *is negative* $l_i' \in I^-\}$

$T_P^F(I^-) = \{a \mid$ *for some clause* $a \leftarrow l_1, l_2 \ldots l_m \in P^*$, *for each* $1 \leq i \leq m$

if l_i *is positive* $l_i \in I^-$ *and,*

if l_i *is negative* $l_i' \in I^+\}$

where l_i' is the complement of literal l_i. It is easily seen that T_P^F is monotonic and its application on consistent partial interpretation results in consistent partial interpretation. It thus poses a least model that is the Fitting model for P. This least fixed point is easily shown to be $T_P^F \uparrow \omega$, where the ordinal powers of T_P^F are defined as follows:

Definition 3. *For any ordinal α,*

$$T_P^F \uparrow \alpha = \begin{cases} \langle \emptyset, \emptyset \rangle & \text{if } \alpha = 0, \\ T_P^F\,(T_P^F \uparrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal,} \\ \langle \cup_{\beta<\alpha}(T_P^F \uparrow \beta)^+, \cup_{\beta<\alpha}(T_P^F \uparrow \beta)^- \rangle & \text{if } \alpha \text{ is limit ordinal} \end{cases}$$

We show an example of Fitting semantics computation on a general deductive database.

Example 3.3.1 Consider a general deductive database P :

r(a, c).

r(b, b).

s(a, a).

p(X) ← r(X, Y) , not p(Y).

p(Y) ← s(Y, a).

Then $T_P^F \uparrow 0 = \langle \emptyset, \emptyset \rangle$. $T_P^F \uparrow 1$ is given by the following partial interpretation:

$$(T_P^F \uparrow 1)^+ = \{r(a, c), r(b, b), s(a, a)\},$$

$$(T_P^F \uparrow 1)^- = \{r(a ,a), r(a, b), r(b, a), r(b, c), r(c, a), r(c, b), r(c, c),$$

s(a, b), s(a, c), s(b, a), s(b, b), s(b, c),

s(c, a), s(c, b), s(c, c)} .

And $T_P^F \uparrow 2 = \mathrm{I} \cup T_P^F \uparrow 1$, where I is the partial interpretation *⟨{p(a)},{p(c)}⟩*. Furthermore, for every ordinal $\alpha > 2$, $T_P^F \uparrow \alpha$ can be seen to be same as $T_P^F \uparrow 2$. So, we can see in Fitting's model that it assigns *p(a)* as *true*, *p(c)* as *false* and no truth value is assigned to *p(b)*. Fitting's semantics has the distinction of being the first semantics to provide unique model for general logic programs. However, they fail to capture *positive recursion*.

Example 3.3.2: Consider the following logic program:

$$a(0) \leftarrow b(0).$$

$$b(0) \leftarrow a(0).$$

The Fitting's model for this program is $\langle \emptyset, \emptyset \rangle$, and it assigns truth value unknown to both *a(0)* and *b(0)*. It is easily seen that there is positive recursion between *a(0)* and *b(0)*. This is captured by the *well founded semantics*.

3.3.2 Well Founded Semantics and Unfounded Sets

The well founded semantics are also the 3-valued semantics given by Van Gelder et al. It assigns some ground atoms truth value as true, some as false and rests are unknown. The unfounded sets form the basis of negative conclusions in well founded semantics. For detailed description the reader is referred to [7].

Definition 4. Let a program P, its Herbrand base H and a partial interpretation I be given. Then A ⊆ H is an unfounded set of P with respect to I if each atom p ∈ A satisfies the following condition: For each instantiated rule r of P whose head is p, (at least) one of the following holds.

1. Some positive subgoal q or negative subgoal not q of body occurs in ¬I i.e., is inconsistent with I.
2. Some positive subgoal of body occurs in A.

Informally the well founded semantics uses condition (1) and (2) to draw negative conclusions. We illustrate unfounded sets through example 3.3.3.

Example 3.3.3

Consider the following ground logic program:

$$p(a) \leftarrow p(c),\ not\ p(b).$$
$$p(b) \leftarrow not\ p(a).$$
$$p(e) \leftarrow not\ p(d).$$
$$p(c) \leftarrow .$$
$$p(d) \leftarrow q(a),\ not\ q(b).$$
$$p(d) \leftarrow q(b),\ not\ q(c).$$
$$q(a) \leftarrow p(d).$$
$$q(b) \leftarrow q(a).$$

The atoms {*p(d), q(a), q(b), q(c)*} form the unfounded set with respect to interpretation ∅. *q(c)* satisfies the first condition and *p(d), q(a)* and *q(b)* satisfy the second condition. It is easily seen that *p(d), q(a)* and *q(b)* depend positively on each other. As a result none of them can be the first to be proven true. Also declaring one of them false does not make any other remaining two true. This is where the set {*p(a), p(b)*} does not form an unfounded set, even though they depend on each other. This is because they depend negatively on each other. As a result making one of them false makes the other true. And if both are declared false at once we have inconsistency. The intuition of preceding example is immediate that union of arbitrary unfounded sets is an unfounded set. This leads naturally to:

Definition 5. *The greatest unfounded set of P with respect to I, $GUS_P(I)$ is the union of all unfounded sets with respect to I***.**

We now define three transformation needed to in turn define the well founded partial model.

Definition 6. *The transformations $T_P(I)$, $U_P(I)$ and $W_P(I)$ are defined as follows*:

- *$T_P(I)$ is the transformation defined by $p \in T_P(I)$ if and only if there is some instantiated rule* ***r*** *of* ***P*** *such that* **r** *has head* ***p****, and each subgoal literal in body of* **r** *occurs in* ***I****.*
- *$U_P(I)$ is the transformation defined by $U_P(I) = \neg G$, where G is $GUS_P(I)$.*
- *Finally $W_P(I) = T_P(I) \cup U_P(I)$.*

Definition 7. *The well founded semantics of a program P is the least fixed point of $W_P(I)$. Every positive literal denotes that its atom is true, every negative literal denotes that its atom is false and missing atoms have undefined truth value.*

3.4 Stable Model Semantics

Another competing thrust that provides meaning to general logic programs is the stable model semantics. They were proposed by Gelfond and Lifschitz [8] at around the same time as well founded model.

In its original form it is a two-valued semantics that is every atom is either true or false. The notable feature of stable model semantics is its simplicity. We first define stable models for logic programs without negation i.e., definite logic programs.

Definition 8. *The least model of a definite logic program is the smallest set of atoms M such that for every rule of the form*

$$A \leftarrow B_1, B_2, \ldots, B_n.$$

If $B_1, B_2, \ldots, B_n \in M$ then $A \in M$.

This definition is same as T_P for definite logic programs as defined by Emden and Kowalski. Thus for general logic program the stable model is a set of atoms. We assume that a set of atoms is available to us and based on certain transformations we decide whether the given set is stable or not.

Definition 8. *Let P be a ground general logic program and let S be a set of atoms. The Gelfond-Lifschitz transformation P^S of P with respect to S is obtained by*:

1. *Deleting every rule with ~L in body with $L \in S$.*
2. *Deleting negative literals from body of the remaining rules.*

P^S is a definite logic program. S is a stable model of P if S is the least model of P^S.

The definition of stable model semantics is simple and elegant but, the stable model semantics are not constructive and thus computationally expensive. As it can be seen a general logic program can have more than one stable model. Consider the following ground program:

Example 3.4.1

$$a \leftarrow not\ b.$$

$$b \leftarrow not\ a.$$

Above program has two stable models {a} and {b}, while the well founded model is ∅. The stable model semantics differ from other semantics discussed so far. The well founded conclusions are only those that are necessarily true. However each stable model corresponds to a possible set of beliefs. Thus, when the program has more than one stable model, it essentially means that there is more than one way in which the meaning of the program can be interpreted.

If there is a unique stable model of a program then it is taken to be the preferred model of the program. Also if there is a two valued well founded model i.e., no ground atom is assigned unknown value then this model is the unique stable model, however the converse is not true as shown in [7]. There are programs with unique stable models that do not coincide with the well founded model.

Example 3.4.2 An example taken from propositional logic [13]

(1) $p \leftarrow not\ q.$

(2) $r \leftarrow p.$

(3) $q \leftarrow not\ p.$

(4) $r \leftarrow not\ r.$

The taking the model as $\{p, r\}$, and using P^S transformation we first remove rules (3) and (4) and now applying (2) of P^S transformation we get the following definite logic program.

$$p \leftarrow .$$

$$r \leftarrow p.$$

The least model for this definite logic program is $\{p, r\}$. Thus this is a stable model and it is unique but well founded model for the above program is $\emptyset$.

There have been number developments relating and modifying stable models and well founded models. For example Sacca and Zaniolo [12] look at intersection of stable models, Baral and Subramaninan [19] consider sets of stable models as meaning of program. We do not get into details of these here. Other developments are Przymusinski [20] gives 3-valued extensions to original two-valued definition of stable models, and shows that they coincide with well founded models.

CHAPTER 4

PARACONSISTENT RELATIONAL DATA MODEL

In this chapter we present a key background material related to our proposed approach. We introduce a model based that is the generalization of the relational data model, the paraconsistent relational model. Here we give a brief overview of this model, for a detailed description the reader is referred to [21].

4.1 Paraconsistent Relations

Paraconsistent relations are the fundamental mathematical structures underlying the model, which essentially contains two kinds of tuples, ones that definitely belong to the relation and others that do not belong to the relation. These structures are strictly more general than the ordinary relations, in that for every ordinary relation there is a paraconsistent relation but not vice-versa. They provide a framework for incomplete or even inconsistent information about the tuples. They naturally model the belief systems rather the knowledge systems, and are thus generalizations of ordinary relations. The operators on ordinary relations can also be generalized for paraconsistent relations.

4.2 Formal Definition of Paraconsistent Relations

Let a *relation scheme* (or just scheme) Σ be a finite set of *attribute* names, where for any attribute name $A \in \Sigma$, *dom*(A) is a non-empty *domain* of values for A. A *tuple* on Σ is any map t: $\Sigma \rightarrow \cup_{A \in \Sigma} dom(A)$, such that $t(A) \in dom(A)$, for each $A \in \Sigma$. Let $\tau(\Sigma)$ denote the set of all tuples on Σ.

Definition 9. *A paraconsistent relation on a scheme Σ is a pair $R = \langle R^+, R^- \rangle$, where R^+ and R^- are any subsets of $\tau(\Sigma)$. We let $\mathcal{P}(\Sigma)$ be the set of all paraconsistent relations on Σ.*

Definition 10. *A paraconsistent relation R on scheme Σ, is consistent if $R^+ \cap R^- = \emptyset$. We let $\mathcal{C}(\Sigma)$ be the set of consistent relations on Σ. Moreover R is called complete relation if $R^+ \cup R^- = \tau(\Sigma)$. If R is consistent and complete i.e. $R^- = \tau(\Sigma) - R^+$, then it is a total relation and we let $\mathcal{T}(\Sigma)$ be the set of all total relations on Σ.*

4.3 Algebraic Operators on Paraconsistent Relations

This section presents the algebraic operators on paraconsistent relations. To reflect the generalization of algebraic operators of ordinary relations, a dot is placed over the ordinary relation operator to obtain corresponding paraconsistent relation operator. For example $\bowtie$, denotes the natural join among ordinary relations, and $\dot{\bowtie}$ denotes natural join among the paraconsistent relations. We first define four set-theoretic algebraic operations on paraconsistent relations.

Definition 11. *Let R and S be two paraconsistent relations on scheme Σ. Then,*

a) the union of R and S, denoted by $R \dot{\cup} S$, is a paraconsistent relation on scheme Σ given by, $(R \dot{\cup} S)^+ = R^+ \cup S^+$, $(R \dot{\cup} S)^- = R^- \cap S^-$;

b) the complement of R, denoted by $\dot{-} R$, is a paraconsistent relation on scheme Σ given by, $(\dot{-} R)^+ = R^-$, $(\dot{-} R)^- = R^+$;

c) the intersection of R and S, denoted by $R \dot{\cap} S$, is a paraconsistent relation on scheme Σ given by, $(R \dot{\cap} S)^+ = R^+ \cap S^+$, $(R \dot{\cap} S)^- = R^- \cup S^-$;

d) *the difference of R and S, denoted by $R \dot{-} S$, is a paraconsistent relation on scheme Σ given by, $(R \dot{-} S)^+ = R^+ \cap S^-$, $(R \dot{-} S)^- = R^- \cup S^+$.*

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 $\{t' \in \tau(\Delta) | t'(A) = t(A)$, for all $A \in \Sigma\}$ of all extensions of t. We extend this notion for any $T \subseteq \tau(\Sigma)$ by defining $T^\Delta = \cup_{t \in T} t^\Delta$. We now define some relation-theoretic operators on paraconsistent relations.

Definition 12. *Let R and S be paraconsistent relations on schemes Σ and Δ, respectively. Then, natural join of R and S, denoted by $R \dot{\bowtie} S$, is a paraconsistent relation on the 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 natural join among relations.*

Definition 13. *Let R be a paraconsistent relation on scheme Σ, and Δ be any scheme. Then, the projection of R onto Δ, denoted by $\dot{\pi}_\Delta(R)$ is a paraconsistent relation on Δ given by, $\dot{\pi}_\Delta(R)^+ = \pi_\Delta((R^+)^{\Sigma \cup \Delta})$, and $\dot{\pi}_\Delta(R)^- = \{ t \in \tau(\Sigma) \mid t^{\Sigma \cup \Delta} \subseteq (R^-)^{\Sigma \cup \Delta} \}$, where π_Δ is the usual projection over Δ on ordinary relations.*

Definition 14. *Let R be a paraconsistent relation on scheme Σ, and let F be any logic formula involving attribute names in Σ, constant symbols (denoting values in the attribute domains), equality symbol =, negation symbol ¬, and connectives ∧ and ∨. Then, the selection of R by F, denoted $\dot{\sigma}_F(R)$, is a paraconsistent relation on scheme Σ, given by $\dot{\sigma}_F(R)^+ = \sigma_F(R^+)$, and $\dot{\sigma}_F(R)^- = R- \cup \sigma_{\neg F}(\tau(\Sigma))$, where σ_F is usual selection of tuples satisfying F.*

Example 4.3.1. Strictly speaking, relation schemes are set of finite attribute names, but in this example they are treated as ordered sequences of attribute names, so tuples can be viewed as the usual list of values. Let {a, b, c} be a common domain for all attribute names, and let R and S be the following paraconsistent relations on schemes ⟨X, Y⟩ and ⟨Y, Z⟩, respectively:

$$R^{+} = \{(b, b), (b, c)\},\ R^{-} = \{(a, a), (a, b), (a, c)\}$$

$$S^{+} = \{(a, c), (c, a)\},\ S^{-} = \{(c, b)\}.$$

Then R $\dot{\bowtie}$ S, is the following paraconsistent relation on scheme ⟨X, Y, Z⟩:

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

$$(R \dot{\bowtie} S)^{-} = \{(a, a, a), (a, a, b), (a, a, c), (a, b, a), (a, b, b), (a, b, c), (a, c, a),$$
$$(a, c, b), (a, c, c), (b, c, b), (c, c, b)\}.$$

Observe how $(R \dot{\bowtie} S)^{-}$ blows up to contain extensions of all tuples in R- and S- . Now $\dot{\pi}_{\langle X, Z\rangle}(R \dot{\bowtie} S)$ becomes the following paraconsistent relation scheme ⟨X, Z⟩:

$$\dot{\pi}_{\langle X, Z\rangle}(R \dot{\bowtie} S)^{+} = \{(b, a)\}, \qquad \dot{\pi}_{\langle X, Z\rangle}(R \dot{\bowtie} S)^{-} = \{(a, a), (a, b), (a, c)\}.$$

The tuples in negative component of the projected paraconsistent relation are such that all their extensions were present in negative component of original paraconsistent relation.

Part II

A NOVEL APPROACH FOR STABLE MODEL COMPUTATION

CHAPTER 5

THE PROPOSED APPROACH

In this chapter we present a novel approach for stable model computation, which is motivated by the idea to develop faster algorithms for computing stable models of a logic program. We give the overview of the model and then present a detailed description of each of the modules involved.

5.1 Assumptions

We assume the following conditions hold for the logic program that is the input to our approach:

1. Let *L* be the given underlying language with a finite set of constants, variables, and predicate symbols, but no function symbols. A *term* is either a variable or a constant. An *atom* is of the form $p(t_1, t_2, \ldots, t_n)$ where p is the *predicate symbol* and t_i are *terms*. A *literal* is either a *positive literal A* or a *negative literal* $\neg A$, where A is an *atom*. Our input logic program would be a finite set of clauses of the form :

   $$a \leftarrow b_1, b_2, \ldots, b_m$$

   where $m \geq 0$ and a and each b_i is an atom.

2. The terms involved in the *IDB* (intentional database) of the logic program can only consist of variables and not constants. *p(X, 2)* where *p* is the predicate symbol, is not allowed as a term in the logic program. Thus there won't be any use of *select* operator in our approach.

5.2 Overview of the Steps Involved

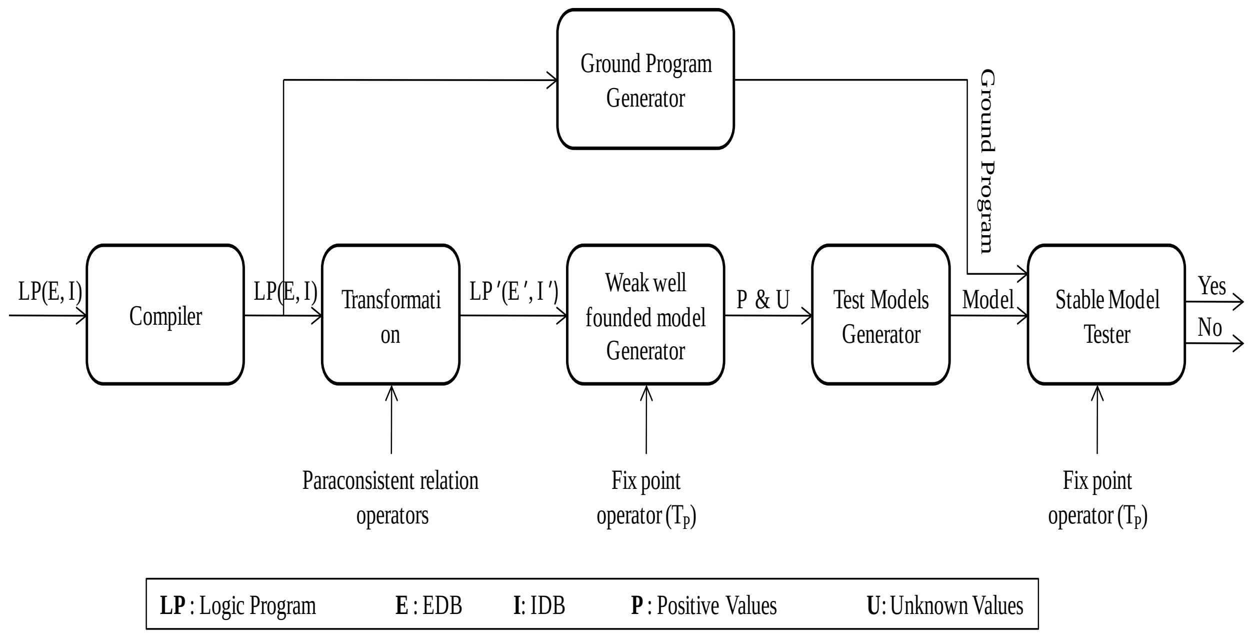


Figure 5.1 Block diagram for the proposed approach

The above figure shows the block diagram of various steps involved in the computation. The process starts with compiling a logic program that performs the syntax and semantic checks and produces a data structure called *rules* consisting of the logic program. These rules are then transformed using the paraconsistent relation operators into another logic program consisting of *transformed rules*. The transformed rules are then used to compute the *weak well founded model* using the fix point operator. After the weak well founded model is computed the *positive* and the *unknown* values are drawn from it and send to generate set of all possible models that are tested for stability. The rules are also sent for ground program generation. The ground program and the models for test are sent one by one to the *stable model tester* which tests each of them for stability and returns a *yes* if model is stable and *no* otherwise. We also note the time taken to complete the process of stable model computation. Next we describe each of the modules in detail.

5.3 Modules

In this section we go over each of the modules in our described above. We start with the compiler and the details about the Datalog language. Next we introduce two algorithms namely, CONVERT and TRANSFORM in the transformation module. Then we go over model generation for stability testing and ground program generation and finally the Stable model tester is described.

5.3.1 Datalog Compiler

Datalog (one without function symbols) with negation, with a well defined declarative semantics based on the work in logic programming has been widely accepted as standard deductive database language [25, 26]. We use Datalog as our language and build a compiler so as to do the syntax and semantic checks and create a data structure, for efficient storage of the logic program. Some definitions related to Datalog.

Definition 15. *Atomic formula*:

a. *$p(x_1, x_2, \ldots, x_n)$ where p is a relation name (predicate name) and $x_1, x_2, \ldots, x_n$ are variables or constants. According to our assumption $x_1, x_2, \ldots, x_n$ can only be variables in EDB.*
b. *x <op> y where x and y are either constants or variables and <op> is one of the following six comparison operators: <, <=, >, >=, =, != . In our language we assume there are no such atomic formulas present.*

Variables that appear only once in the rule can be replaced by anonymous variable (represented by underscore). Every anonymous variable is different from all other variables.

Definition 16. *Datalog rule*:

$$p \text{ :- } q_1, q_2, \ldots, q_n.$$

Where, p is an atomic formula and q_1, q_2, ..., q_n are either atomic formula or negated atomic formula (i.e. atomic formula preceded by not). p is referred to as the head and q_1, q_2, ..., q_n are referred to as subgoals of body.

Definition 17. *Safe Datalog rule*:

A Datalog rule p :- q_1, q_2, ..., $q_{n.}$ is safe

a. *If every variable that occurs in a negated subgoal also appears in a positive subgoal and*
b. *If variable that appears in the head of the rule also appears in the body of the rule.*

The compiler is build using the JFlex and JCup technologies that builds the lexer and parser. A block diagram depicts the process. We input a logic program in the compiler and get a data structure called *rules* as the output if there is no syntax or semantic errors.

The *data structure* is build using two classes namely *predicate* and *rule*. The parameters and its types for the classes are shown below (using the definitions above):

Table 5.1 Predicate

Parameter Name	**Data Type**	**Description**
Name	String	Stores the relation name as p for above atomic formula
Arglist	Vector	Stores the (x_1, x_2…x_n) arguments
isNegative	Boolean	Stores the information whether atomic formula is positive or negative.

Table 5.2 Rule

Parameter Name	Data Type	Description
Head	Predicate	Stores the atomic formula p.
Body	Vector	A vector of predicates that stores (q_1, q_2 …q_n).
isEDB	Boolean	Whether rule is EDB (only has head) or IDB.

Finally, the data structure *rules* is a vector where each element is of type rule.

We also perform some semantic checks, which are as follows:

1. *Arity Check:* If an atomic formula appears more than once in the rules, then for its each instance the argument list should be of the same size, i.e. if p(x_1, x_2…x_n) and q(y_1, y_2, …y_n) are in logic program and if p = q, then n should be equal to m (n = m).

   Example 5.3.1

   (1) p(X, Y) :- r(X, Y, Z), s(X, Z).

   (2) q(Z) :- r(X, Y), s(Z).

   the above program has error as the relation named *r* has argument lists of size 2 in rule 1 and of size 3 in rule 2.

2. *Safety Checks*

   a. Every variable that appears in negated subgoal should appear in the positive subgoal. Suppose there is rule of the form:

      p(X, Y) :- r(X, Y), not s(X, Z)

      then the rule is *not safe* as the variable *Z* only appears in a negative subgoal and not in a positive subgoal.

   b. Every variable that appears in the head of the rule must appear in the body of the rule. Suppose there is rule as follows:

      p(X, Y) :- r(X, Z), s(Z).

is *not safe* as variable *Y* does not appear in the body of the rule.

Once we get an error free logic program we move to the next step that is transformation of the logic program into a new logic program.

5.3.2 Transformation

We now present a transformation of a general deductive database P. In this method the paraconsistent relations are the semantic objects associated with the predicate symbols in P. The method involves two steps. The first step is to convert P into a set of paraconsistent relation definitions for predicate symbols occurring in P taken from [21]. These definitions are of the form

$$p = D_p,$$

where, p is a predicate symbol of P, and D_p is an algebraic expression involving predicate symbols of p and paraconsistent relation operators. The second step is to generate a new logic program from the algebraic expression that can be used to compute the weak well founded model.

Before describing the method to convert the given database P into set of definitions for predicate symbol in P, let us look at an example. Suppose the following are the only clauses with the predicate symbol p in their heads:

$$p(X) \leftarrow r(X, Y), \neg p(Y)$$

$$p(Y) \leftarrow s(Y, Z)$$

From these clauses the algebraic definition constructed for symbol p is the following:

$$p = (\dot{\pi}_{\{X\}}(r(X, Y) \mathbin{\dot{\bowtie}} \dot{-}\, p(Y)))[X] \;\; \dot{\cup}\; (s(Y, Z))\, [Y]$$

Such a conversion exploits the close connection between attribute names in relation schemes and variables in clauses, as pointed out in [25]. The expression thus constructed can be used to arrive at a better approximation of paraconsistent relation *p* from some approximations of *p*, *r* and *s*. We now give the algorithm to convert one clause into an expression.

The algorithm presented here is a modification of the original convert algorithm as our deductive database does not involve any select conditions and the terms of IDB do not contain constant values.

Algorithm 1 CONVERT

Input: A general deductive database clause $l_0 \leftarrow l_1, l_2, \ldots, l_m$.

Let l_0 be of the form $p_0(A_{01}, \ldots, A_{0k0})$, and each l_i, $1 \leq i \leq m$, be either of the form $p_i(A_i, \ldots, A_{iki})$ or of the form $\neg p_i(A_i, \ldots, A_{iki})$. For any i, $1 \leq i \leq m$, let V_i be the set of all variables occurring in l_i.

Output: An algebraic expression involving paraconsistent relations.

Method: The expression is constructed using the following steps:

1. Let $\hat{l}_i$ be the atom $p_i(B_{i1}, \ldots, B_{iki})$ and F_i be the conjunction of $C_{i1} \wedge C_{i2} \ldots \wedge C_{iki}$. If l_i is a positive literal, then let Q_i, be the expression $\dot{\pi}_{\mathrm{Vi}}(\dot{\sigma}_{Fi}(\hat{l}i))$). Otherwise, let Q_i be the expression $\dot{-}\, \dot{\pi}_{\mathrm{Vi}}(\dot{\sigma}_{Fi}(\hat{l}i))$).

   As a syntactic optimization, if all conjuncts of F_i are true (i.e. all argument of l_i are distinct variables), then both $\dot{\sigma}_{Fi}$ and $\dot{\pi}_{\mathrm{Vi}}$ are reduced to identity operations, and hence are dropped from the expression. For example, if $l_i = \neg p(X, Y)$, then $Q_i = \dot{-}p(X, Y)$. As our language does not contain any select conditions we drop both $\dot{\sigma}_{Fi}$ and $\dot{\pi}_{\mathrm{Vi}}$ to identity operation always.

2. Let E be the natural join ($\dot{\bowtie}$) of Q_i*'s* thus obtained, $1 \leq i \leq m$. The output expression is $(\dot{\pi}_V(E))[B_{01}, \ldots, B_{0k0}]$, where V is the set of variables occurring in $\hat{l}_0$.

From the algebraic expressions obtained by algorithm CONVERT for all clauses in general deductive database we construct another logic program using algorithm Transform.

Before going to transform we give an example from [13] of a deductive database and application of convert algorithm on its clauses.

Example 5.3.2

This example represents a circuit consisting of an unusual sort of a logic gate, with one positive input *X*, and one negative input *Y*, the its output is 1 or "true" if and only if *X* is 1 and *Y* is 0("false"). There is an *EDB* predicate *g(X, Y, Z) that* says there is a gate of this type with positive input *X*, negative input *Y*, and output *Z.* We may think of inputs and outputs as being terminal or wire nets. There is also an *EDB* predicate *t0* that is true of those input terminals that are externally set to 1. Input terminals that are set to 0 do not appear in *t0*.

The IDB predicate is *t*. The intended significance of the positive ground atom *t(a)* being in the model is that the circuit value of terminal *a* is 1. If ¬*t(a)* is in the model, then the value of terminal is 0. What if the value that terminal a has ambiguous; either it depends on critical race in the circuit or oscillates in normal circuit operation? Then, we expect *t(a)* to have a third, "unknown" value of three valued logic. The following are the rules defining the operation of the gates:

t0(2).

g(5,1,3).

g(1,2,4).

g(3,4,5).

t(Z) :- t0(Z) .

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

The data in the EDB i.e. $t_0(2)$, *g(5, 1, 3), g(1, 2, 4)* and *g(3, 4, 5)* represents the circuit of Figure 5.5.2 with only second input set to true.

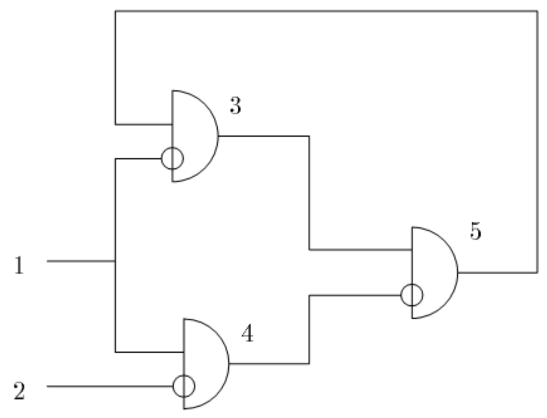


Figure 5.2 Circuit for Example 5.5.2

We now apply Algorithm **CONVERT** on the two *IDB* clauses:

1. *t(Z) :- t0(Z).* The expression of this is *t(Z) :- t0(Z).*
2. *t(Z) :- g(X, Y, Z), t(X), not t(Y).* The positive literals *g(X, Y, Z)* and *t(X)* remain the same and the literal not *t(Y)* becomes $\dot{-}t(Y)$. Then on the application of step 2 we get the following as the algebraic expression:

$$t(Z) :- \dot{\pi}_{[Z]}(E)$$

where, *E* is the natural join ($\dot{\bowtie}$) of *g(X, Y, Z), t(X)* and $\dot{-}t(Y)$.

When we get the algebraic expressions for all the clauses of IDB we move to the next step. Here we introduce the algorithm **TRANSFORM** that takes these algebraic expressions as input and returns a logic program as output. This logic program contains positive and negative parts for all the different predicate symbols including the EDB relations. The transformation converts the potentially harmful negation in the logic program into harmless negation. We also create some new relations like the temporary and domain.

Algorithms 2 TRANSFORM

Input**:** EDB clauses and Algebraic expressions involving paraconsistent relations for IDB clauses.

Output: A general logic program consisting of clauses of the form $l_0 \leftarrow l_1, l_2, \ldots, l_m$.
Let l_0 be of the form $p_0(A_{01}, \ldots, A_{0k0})$, and each l_i, $0 \leq i \leq m$, be either of the form $p_i(A_i, \ldots, A_{iki})$ or of the form $\neg p_i(A_i, \ldots, A_{iki})$.

Method: The logic program is constructed is using the following steps.

1. Transform the EDB clauses

   a. Let $a_1, \ldots, a_n$ be the constants present in EDB. Then, for each constant value a_i create the following predicates with *dom* (domain) as the predicate symbol as follows

$$dom\ (a_1).$$
$$\vdots$$
$$dom\ (a_n).$$

b. Let $l_1,\ldots,l_n$ be the *EDB* predicates, where l_i is $p_i(B_1,\ldots,B_m)$, $B_1,..,B_m$ are *constants,* and $1 \leq i \leq n$. Complete each *EDB* predicate p_i as follows. Firstly, rename the existing predicates and add it to the new logic program:

$$p_1\_plus(B_1, \ldots, B_m).$$

$$\vdots$$

$$p_n\_plus(B_1, \ldots, B_m).$$

For each unique predicate name p_i in *EDB* add a rule as follows:

$$p_i\_minus(V_1,\ldots,V_n) \text{ :- } dom(V_1), dom(V_2),\ldots, dom(V_n), not\ p_i\_plus(V_1,\ldots, V_n).$$

where, $V_1,\ldots,V_n$ are variables.

For example there are two *EDB* predicates *p(1, 2)* and *p(2, 3)* then we add the following predicates, *p_plus(1, 2), p_plus(2, 3)* and a rule written below to the new logic program.

$$p\_minus(X, Y) \text{ :- } dom(X), dom(Y), not\ p\_plus(X, Y).$$

2. Renaming: If n IDB expressions have head with same predicate name *p,* and n > 1, then, rename the clauses as *p1, p2, ..., pn* in the algebraic expression. Let the argument list be $(V_1, \ldots, V_m)$ where $V_1,\ldots, V_m$ are variables. Add the following positive predicates to the logic program:

$$p\_plus(V_1,\ldots,V_m) \text{ :- } p1\_plus(V_1,\ldots,V_m).$$

$$\vdots$$

$$p\_plus(V_1,\ldots,V_m) \text{ :- } pn\_plus(V_1,\ldots,V_m).$$

Add the following rule for the negative predicate.

$$p\_minus(V_1,\ldots,V_m) \text{ :- } p1\_minus(V_1,\ldots,V_m),\ \ldots.,\ pn\_minus(V_1,\ldots,V_m).$$

And, add the following rule for unknown values.

$$p\_unknown(V_1,\ldots,V_m) \text{ :- } dom(V_1),\ \ldots,\ dom(V_m), \text{ not } p\_plus(V_1,\ldots,V_m) \text{ , not } p\_minus(V_1,\ldots,V_m).$$

3. Construct paraconsistent trees for each IDB expression.
   a. Let the IDB clause be $l_0 \leftarrow l_1, \ldots, l_n, p_1,\ldots,p_m$ where, $l_1, \ldots, l_n$ are positive literals and $p_1, \ldots, p_m$ are negative literals. For this clause let the following be the algebraic expression:

$$l_0 \text{ :- } (\dot{\pi}_V(E))[B_{01}, \ldots, B_{0k0}]$$

where, V is the set of variables occurring in l_0 and E is the natural join of $l_1, \ldots, l_n$, $p_1,\ldots,p_m$.

Let $l_0 = p(B_1, \ldots, B_n)$, $l_i = a_i(C_1,\ldots, C_m)$, where $1 \leq I \leq m$ and $p_j = \neg\, b_j(D_1, \ldots, D_k)$ where $0 \leq j \leq k$, and $C_1,\ldots, C_m$ and $D_1,\ldots, D_k$ are variables. So, the paraconsistent tree of the above expression would be depicted as follows:

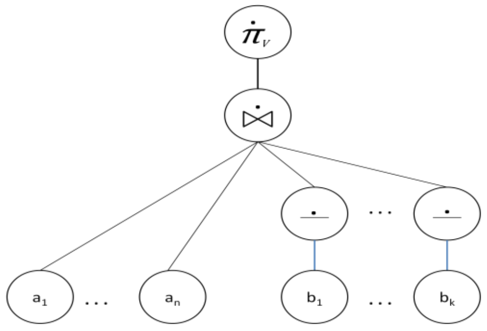


Figure 5.3 Paraconsistent expression tree

b. Naming the tree

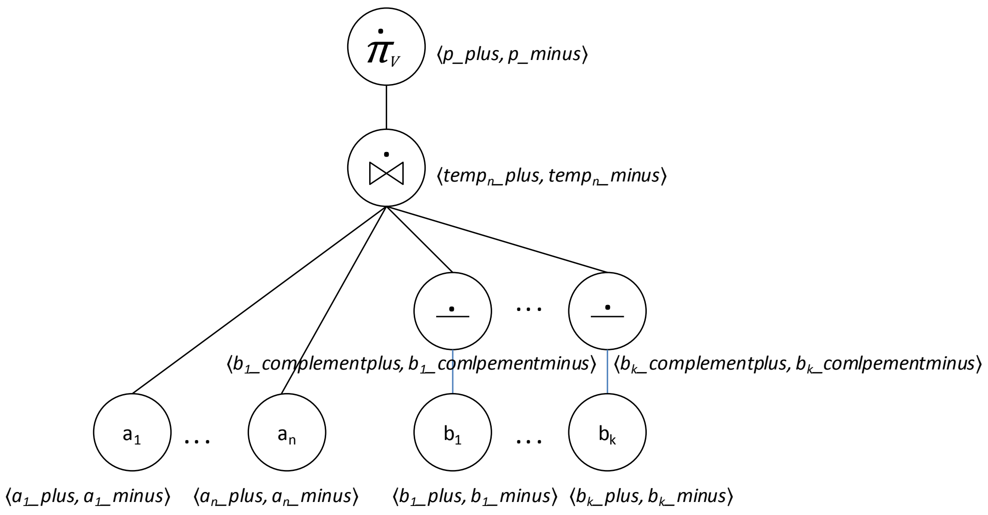
i. Name the child node a_i for $1 \leq i \leq$ n with its pair ⟨a_i*_plus*, a_i*_minus*⟩.

ii. Name the *complement* ($\dot{-}$) nodes with child node as b_j as a pair ⟨b_k*_complementplus*, b_k*_complementminus*⟩, where $0 \leq j \leq$ k.

iii. If join ($\dot{\bowtie}$) is an internal node name it $temp_n$ where n is the nth rule in IDB.

iv. Name the root node with the head predicate p as a pair ⟨*p_plus, p_minus*⟩.

The named tree of figure 5.2.3 is as follows:



⟨p_plus, p_minus⟩
⟨temp_n_plus, temp_n_minus⟩
⟨b1_complementplus, b1_comlpementminus⟩
⟨bk_complementplus, bk_comlpementminus⟩
a1
an
b1
bk
⟨a1_plus, a1_minus⟩
⟨an_plus, an_minus⟩
⟨b1_plus, b1_minus⟩
⟨bk_plus, bk_minus⟩

Figure 5.4 Named paraconsistent tree

4. Create rules for paraconsistent trees of all IDB expressions using the steps below. Start writing rules bottom-up for all internal nodes.

a. If node type is complement ($\dot{-}$) and the child node is predicate $c(B_1, B_2, \ldots, B_n)$.

$$c\_complementplus(B_1, B_2, \ldots, B_n) \text{ :- } c\_minus(B_1, B_2, \ldots, B_n).$$

$$c\_complementminus(B_1, B_2, \ldots, B_n) \text{ :- } c\_plus(B_1, B_2, \ldots, B_n).$$

b. If node type is join ($\dot{\bowtie}$) and it is the root node named ⟨*p_plus, p_minus*⟩, with the child nodes named as ⟨c_1*_plus*, c_1*_minus*⟩,…,⟨c_n*_plus*, c_n*_minus*⟩ then add the following rules:

$p\_plus(B_1,\ldots B_z)$:- $c_1\_plus(V_1, \ldots, V_m), \ldots, c_n\_plus(T_1,\ldots T_k)$.

Also, add *n* rules where, *n* is the number of child nodes, and $1 \leq i \leq$ n as follows:

$p\_minus(B_1,\ldots B_z)$:- $c_i\_minus(U_1,\ldots,U_m)$.

if $m < n$ that is if $B_1,\ldots B_j = U_1, \ldots, U_m$ and $i \leq j \leq z$, then extend the rule by adding dom predicates for $B_j,\ldots B_z$ to the above rule:

$p\_minus(B_1,\ldots B_n)$:- $c_i\_minus(U_1,\ldots,U_m), dom(B_j), \ldots., dom(B_z)$.

c. If node type is join ($\dot{\bowtie}$) and it is an internal node named $temp_n$ with child nodes named as ⟨c_1*_plus*, c_1*_minus*⟩,…,⟨c_n*_plus*, c_n*_minus*⟩, then let the argument list of *temp* be *V*, where *V* is the set of all the variables occurring the child nodes of $temp_n$. And we add the rule as follows:

$temp_n\_plus(V)$:- $c_1\_plus(V_1, \ldots, V_m), \ldots, c_n\_plus(T_1,\ldots T_k)$.

Also, add *n* rules where, *n* is the number of child nodes, and $1 \leq i \leq$ n as follows:

$temp_n\_minus(V)$:- $c_i\_minus(U_1,\ldots,U_m), dom(B_1), \ldots., dom(B_z)$.

where $B_1,\ldots, B_z$ are the set of variables not present in $U_1,\ldots, U_m$.

d. If node type is projection ($\dot{\pi}_V$) named *p*, and *V* is the set projected variables, and the child node is named $temp_n$. In $temp_n$, variables that do not appear in *V* are anonymous and can be denoted by underscore.

$p\_plus(V)$:- $tempn(V, \_,\ldots)$.

(represent the projected variables in temp and rest of them with underscore)

We add three more rule for the negative predicate as follows:

$temp_{n1}(A_1, \dots, A_n)$:- $dom(A_1), \dots, dom(A_n)$.

$temp_{n2}(V)$:- $tempn1(A_1, \dots, A_n)$, not $temp_n\_minus(A_1, \dots, A_n)$.

$p\_minus(V)$:- $dom(V_1), .., dom(V_n)$, not $temp_{n2}(V)$.

Where, $A_1, \dots, A_n$ is the set of variables in child node $temp_n$ and $V_1, \dots, V_n$ are the set of variables in V.

e. If the expression tree is a single child tree and it does not involve even projection then for such tree we write the rules as follows. If root node is named ⟨*p_plus, p_minus*⟩ and the child node is ⟨*c_plus, c_minus*⟩ then the rules are:

$p\_plus(B_1, \dots, B_n)$:- $c\_plus(B_1, \dots, B_n)$.

$p\_minus(B_1, \dots, B_n)$:- $c\_minus(B_1, \dots, B_n)$.

5. For each unique IDB predicate *p*, except for those created in Step 2, add the following rule for unknown values:

$p\_unknown(B_1..B_n)$:- $dom(B_1), \dots, dom(B_n)$, not $p\_plus(B_1, \dots, B_n)$, not $p\_minus(B_1, \dots, B_n)$.

The TRANSFORM algorithm removes the 'harmful negation' or we can say the unsafe negation from the original program because with each negative predicate it introduces the dom predicates, which are joined with the negative predicate, that limits the domain to all constant value present in the Herbrand base. This could otherwise cause safety issue as to negation of a relation can be infinite. Thus, TRANSFORM algorithm eliminates the arbitrary negation from general deductive databases and at the same retains the meaning of the deductive databases with respect to Fitting's model.

Next we present an example of application of transform algorithm on algebraic expressions of example 5.3.2.

Example 5.3.3

Consider the expressions shown below:

t0(2).

g(1, 2, 4).

g(3, 4, 5).

g(5, 1, 3).

(1) t(Z) :- t0(Z).

(2) t(Z) :- $\dot{\pi}_{\{Z\}}(E)$

where, *E* is the natural join ($\dot{\bowtie}$) of *g(X, Y, Z)*, *t(X)* and $\dot{-}$*t(Y).*

1. (a) The constant in the EDB of the program are {1, 2, 3, 4, 5}, so we add the following five EDB rules:

   dom(1), dom(2), dom(3), dom(4) and dom(5).

   (b) Now we complete the EDB predicates t_0 and *g* by adding the following rules.

   t_0*plus(2), g_plus(1, 2, 4), g_plus(3, 4, 5), g_plus(5, 1, 3).*

   t_0*_minus(Z) :- dom(Z) , not* t_0*_plus(Z).*

   g_minus(X, Y, Z) :- dom(X), dom(Y), dom(Z), not g_plus(X, Y, Z).

2. Next we have two IDB rules with same head *t* so we rename them to t1 and t2

t1(Z) :- t0(Z).

t2(Z) :- $\dot{\pi}_{[Z]}$*(E).*

where, E is (g(X, Y, Z) $\dot{\bowtie}$ *t(X)* $\dot{\bowtie}$ *(*$\dot{-}$*t(Y)))*

and add the following rules:

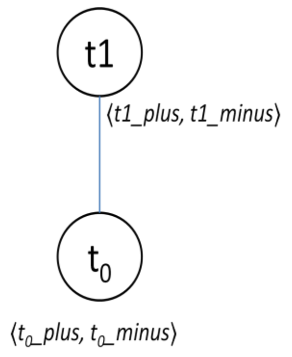
t_plus(Z) :- t1_plus(Z).

t_plus(Z) :- t2_plus(Z).

t_minus(Z) :- t1_minus(Z), t2_minus(Z).

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

3. Now we create the trees for the IDB expressions as follows:



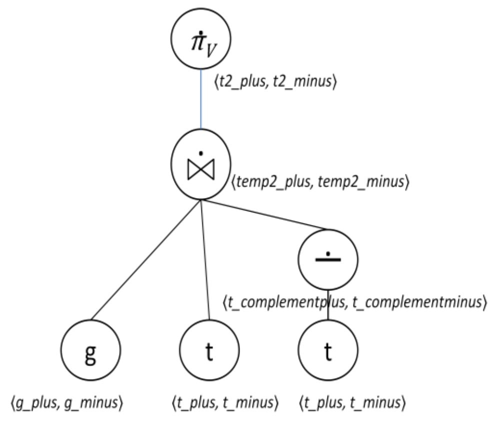


Figure 5.5 Tree for Expression 1 Figure 5.6 Tree for Expression 2

4. Now we write rules from the above expression trees:

Expression 1*: t1(Z) :- t0(Z).*

t1_plus(Z) :- t0_plus(Z).

t1_minus(Z) :- t0_minus(Z).

Expression 2: *t2(Z) :-* $\dot{\pi}_{[Z]}$*(E).*

We start bottom-up in the tree with first internal node that is ($\dot{-}$), adding the following rules:

a. t_complementplus(Z) :- t(Z).

t_complementminus(Z) :- t(Z).

b. Next, we add rules for join ($\dot{\bowtie}$) node as follows:

temp2_plus(X, Y, Z) :- g_plus(X, Y, Z), t_plus(X), t_complementplus(Y).

temp2_minus(X, Y, Z) :- g_minus(X, Y, Z).

temp2_minus(X, Y, Z) :- t_minus(X), dom(Y), dom(Z).

temp2_minus(X, Y, Z) :- t_complementminus(Y), dom(X), dom(Z).

c. Now we write rules for the projection node ($\dot{\pi}_V$) named *t2*, where V = {Z}.

t2_plus(Z) :- temp2_plus(_, _, Z).

temp21(X, Y, Z) :- dom(X), dom(Y), dom(Z).

temp22(Z) :- temp21(X, Y, Z), not temp2_minus(X, Y, Z).

t2_minus(Z) :- dom(Z), not temp22(Z).

5. Now we add the rules for unknown predicates. But, here they have already been added in step 2. So, by keeping the unknown rules in the end we get the following transformed program:

dom(2).

dom(5).

dom(1).

dom(3).

dom(4).

t0_plus(2).

g_plus(5, 1, 3).

g_plus(1, 2, 4).

```
g_plus(3, 4, 5).
t0_minus(Z):-dom(Z), not t0_plus(Z).
g_minus(X, Y, Z):-dom(X), dom(Y), dom(Z), not g_plus(X, Y, Z).
t1_minus(Z):-t0_minus(Z).
t1_plus(Z):-t0_plus(Z).
t_plus(Z):-t1_plus(Z).
t_complementplus(Y):-t_minus(Y).
t_complementminus(Y):-t_plus(Y).
temp2_minus(X, Y, Z):-g_minus(X, Y, Z).
temp2_minus(X, Y, Z):-t_minus(X), dom(Y), dom(Z).
temp2_minus(X, Y, Z):-t_complementminus(Y), dom(X),dom(Z).
temp2_plus(X, Y, Z):-g_plus(X, Y, Z), t_plus(X), t_complementplus(Y).
t2_plus(Z):-temp2_plus(_,_,Z).
t_plus(Z):-t2_plus(Z).
temp21(X, Y, Z):-dom(X), dom(Y), dom(Z).
temp22(Z):-temp21(X, Y, Z), not temp2_minus(X, Y, Z).
t2_minus(Z):-dom(Z), not temp22(Z).
t_minus(Z) :- t1_minus(Z), t2_minus(Z).
t_unknown(Z) :- dom(Z), not t_plus(Z), not t_minus(Z).
```

5.3.3 Fitting's Model Generation

After we have generated the logic program we use the fix point semantics and apply the fix point operator T_P on the logic program. The application of T_P gives us the meaning of the program and in this case it is the weak well founded or the Fitting's model for the program. We use this model as preprocessing mechanism for stable model computation. We know, the least fixpoint computation amounts to an iterative procedure, where partial results are added to a relation until steady state is reached. In order to compute the partial relations we make use *DBEngine*, a database system where we store the information present in our logic program and compute the partial relations to generate the least fix point of the logic program. The *DBEngine* consists of three classes namely, *Driver*, *Relation and Tuple* and it provides the usual database operations like join, minus, project, select and others.

Next we show the results of applying the T_P operator on our logic program. The values which we are interested in are for the *t* relation as it is the only predicate present in *IDB*. We start with the EDB predicates *t0* and *g*. *t0_plus* contains values {*2*} and *t0_minus* has four values {*1, 3, 4, 5*}. Similarly, in the first pass we get *g_plus* with three tuples *{{5, 1, 3}, {1, 2, 4}, {3, 4, 5}}* and *g_minus* has 122 tuples as it has all the combinations of these values i.e. 5^3 except the three tuples in *g_plus*. These values remain same for all the passes for EDB predicates *g* and *t0*. Next the values that get populated are for the IDB and we show the values of relation *t*.

Table 5.3 Fix point computation for predicate *t*

Iteration	**t1_plus**	**t1_minus**	**t2_plus**	**t2_minus**	**t_plus**	**t_minus**
1	{2}	{1, 3, 4, 5}	∅	{1, 2, 4}	{2}	{1, 4}
2	{2}	{1, 3, 4, 5}	∅	{1, 2, 4}	{2}	{1,4}

We get the steady state in pass 2. Since we take the paraconsistent union of t1 and t2 we get the following:

$$(t1 \dot{\cup} t2)^{+} = t1^{+} \cup t2^{+} , (t1 \dot{\cup} t2)^{-} = t1^{-} \cap t2^{-};$$

Where *(t1 $\dot{\cup}$ t2)*$^+$ is *t_plus* and *(t1$\dot{\cup}$ t2)*$^-$ is *t_minus*. Thus, the partial relation for *t* would be:

$$\langle t^+, t^- \rangle = \langle \{2\}, \{1, 4\} \rangle$$

Also, from the rule:

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

We get the unknown values as *t(3)* and *t(5).*

Thus we get the Fitting's model for our original program of example 5.3.2 as:

Positive values: {t0(2), g(5, 1, 3), g(3, 4, 5), g(1, 2, 4), t(2)}

Negative values: {t(1), t(4)}

Unknown values: {t(3), t(5)}

After the computation of the weak well founded model we move to the next step and that is stable model generation.

5.3.4 Models and Ground Program Generation

After the generation of the Fitting's model, the next step is come up with models that can be tested for stability. The models for test are generated from the unknown and positive values of the Fitting's model. The unknown values may be positive or negative, thus we take them into consideration for stability testing. Let the *P* be the set of positive values as $\{P_1, P_2, \ldots, P_n)$ and *U* be the set of unknown values $\{U_1, U_2, \ldots, U_m)$. Then the number of possible stable models that can be generated using the *P* and *U* values are $2^{(n+m)}$, where each model has the EDB rules of the original logic program which are always true. The number models that can be generated from example 5.2.2 with one positive and two unknown predicates is $2^{(1+2)}$ i.e. 8, where, EDB is {*t0(2), g(1, 2, 4), g(3, 4, 5), g(5, 1, 3)*}, positives are {*t(2)*} and unknowns are {*t(3), t(5)*}. Then, following are the models that would be tested for stability.

1. *∅ + EDB*
2. *{t(2)} + EDB*
3. *{t(3)} + EDB*
4. *{t(5)} + EDB*
5. *{t(2), t(3)} + EDB*
6. *{t(2), t(5)} + EDB*
7. *{t(3), t(5)} + EDB*
8. *{t(2), t(3), t(5)} +EDB*

After the models are generated we need to test each one for stability. The stability testing is done using the Gelfond-Lifschitz transformation P^S of *P* with respect to *S*, where *P* is the original logic program and *S* would be the model for testing. To do the stability testing we need to generate the ground program for our original logic program. The ground program is generated by replacing all the variables in the program with constant values. Let *c* be the number of constants in the program $A(B_{1,}\ldots,\ B_n)$ be a predicate in the IDB rule with largest argument list, then the number of ways it can be instantiated by constants is n^c. And using this instantiation we instantiate the complete rule. Thus for the rule:

t(Z) :- t0(Z)

with 5 constants and one variable, the possible instantiations are 5^1 i.e. 5. But, only one of them is useful i.e.

t(2) :- t0(2)

Similarly, the rule:

t(Z) :- g(X, Y, Z), t(X), not t(Y)

with 5 constants and largest argument list 3(of predicate g (X, Y, Z)) can be instantiated in 5^3 i.e. 125 ways, out of which only 3 are useful, they are as follows:

t(4) :- g(1, 2, 4), t(1), not t(2)

t(5) :- g(3, 4, 5), t(3), not t(4)

t(3) :- g(5, 1, 3), t(5), not t(1)

Thus the ground program would contain 130 rules + EDB, but the useful rules are 4 of those and EDB. Next we show the stability testing of one of the models from the above shown 8 models. For this process, we built the stable model tester that takes the input as the ground program and one model at a time and tests it for stability. It gives the output as yes or no and adds the model to a vector if it is stable.

5.3.5 Stable Model Tester

This module is designed to test the stability of a model given a ground logic program and the model for test. We use the rules stated by the *Definition 8*. We take the ground logic program *P* and the model for test *S* and get the transformation P^S. Following is step by step application of definition 8 for S as {*t(2) + EDB*}.

The ground program for example 5.5.2 with useful IDB rules is as follows:

t(2) :- t0(2)

t(4) :- g(1, 2, 4), t(1), not t(2)

t(5) :- g(3, 4, 5), t(3), not t(4)

t(3) :- g(5, 1, 3), t(5), not t(1)

After the application of first step 1, i.e. deleting every rule with ~L in body with L ∈ S we get the following:

t(2) :- t0(2)

t(5) :- g(3, 4, 5), t(3), not t(4)

t(3) :- g(5, 1, 3), t(5), not t(1)

After the application of step 2, i.e. deleting negative literals from remaining rules we get the following transformation P^S:

t(2) :- t0(2)

t(5) :- g(3, 4, 5), t(3).

t(3) :- g(5, 1, 3), t(5).

Now we again apply the T^P operator on this logic program. *t(2)* becomes true because of *t0(2)* but *t(5)* and *t(3)* are dependent on each other and none of them is true so they become false. Thus, the least fix point of this program as {*t(2) + EDB*} which is the model for test i.e. *S*. Since the least fix point of P^S and *S* are the same we can conclude *S* is a stable model for *P* which our original logic program. Similarly, we test the rest of the 7 models for stability and none of them is stable. Thus, our logic program has one stable model. But suppose if we add the following

EDB predicates {*g(2, 5, 6), g(2, 6, 3)*} to the original program and then apply the same procedure we get 2 stable models where {t(3), t(5), t(6)} are unknown.

5.4 Implementation of the Modules

This section briefly describes how each of the modules have actually been implemented with the java code.

5.4.1 Compiler

Input: A logic program

Output: Vector rules where each element is of type RULE.

The Datalog compiler is built using the JFlex and JCup technologies. The JFlex creates a lexical analyzer which creates tokens that are forwarded to the parser. The parsers tests each token for syntax and requests more tokens from the lexer, and finally create our data structure rules.

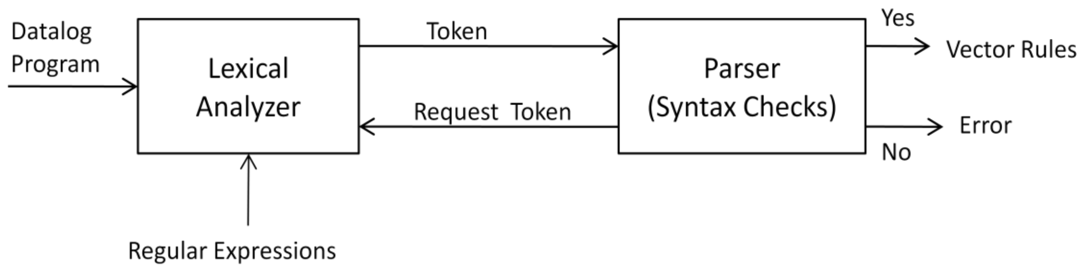


Figure 5.7 Block diagram for compiling process

The grammar for our parser is as follows:

```
ddb           ::= rules DOLLAR
idb_rules     ::= idb_rule |  idb_rule idb_rules
idb_rule      ::= predicate IMPLIES idb_body PERIOD | predicate PERIOD
```

```
idb_body     ::= literal |  literal COMMA idb_body
literal      ::= NOTOP predicate | predicate
predicate    ::= NAME LPAREN arg_list RPAREN
arg_list     ::= arg | arg COMMA arg_list
arg          ::= NAME | NUMBER | VARIABLE
constant     ::= NUMBER | STRING
```

We write java statements for each grammar rule in the Cup file, for details refer to appendix section 9.2. One example of the extracting data from rules is as follows which gives us a rule from the program:

```
idb_rule::= predicate: head IMPLIES idb_body:body PERIOD
            {:
                Rule R = new Rule (head, body, false);
                RESULT = R;
            :}
```

Similarly we write java code for each of the grammar statements and if there are no errors in the syntax of the program the data structure rules which is of type Vector is created. In Rules each element is of type Rule (the class introduced earlier). This data structure provides a way to store and retrieve data from our program. After this step we perform the semantic checks mentioned earlier,

1. Arity Check: From rules we collect all the predicates with same name and check its argument list if they are not equal, an error is reported else we move to safety checks. This process is repeated for all unique predicates.
2. Safety Checks:
    a. We collect the variables in head of the rule and collect variables from the body of the rules and compare both the list to check whether all the variables

in head list are present in the body list. This process in repeated for all the rules. For actual code refer section 9.6.

b. Similarly, for each rule we collect the variables in negative predicates from the rule and collect variables from positive predicates from the body of the rule. Then the two lists are compared to check whether each variable in negative list is present in positive list.

If any of the above checks fail the program throws and error and it is stopped. Else we move to the next step that is transformation.

5.4.2 Generating Transformed Program

Input: Data structure rules

Output: Data structure transformed rules (a Fitting's model equivalent).

The transformation is carried out using the same procedure as explained in the TRANSFORM algorithm, although we do not create the equation using paraconsistent trees and expressions.

We start with modifying the rule vector if more than one rule has same head *p* we change the head names in Vector itself to p1, …, pn and add the new rules for its union and intersection in a new vector same type as rules named as paraconsistent rules. After this step we begin the EDB transformation and IDB transformations:

1. EDB Transformation:
    a. Extract all the constants from the original program and add dom rules for each constant in the new vector paraconsistent rules
    b. Then for each unique EDB predicate we complete the rules with its plus and minus counterparts. For example the code to implement is shown where P is new

predicate with name p_plus and AL is its argument list. In Rule we add this predicate P as its head in rule R1, and finally add it to new vector EDB rules.

```
Predicate P = new Predicate(pname, AL,false);
Rule R1 = new Rule(P,null,true);
EDBRules.add(R1);
```

c. When all the rules have been created they are added to the paraconsistent rules vector.

2. IDB transformation:

   a. We start by checking whether the rule has projection or not. If yes then it is treated a bit differently than the one without projection.

   b. For each rule we start by checking whether it is positive or negative, if it is negative then add the rules mentioned for the $(\dot{-})$ complement node, because it will always be the bottom most node.

   c. Then we add rules for normal join which is union of all negative goals and intersection of positive goals. We check whether any of the negative goals has lesser variables than the head then we add *dom* predicates to complete the node.

   d. Finally we handle projection by writing rules for temp node, and it is numbered based on its position in the rules vector so that if there is more than one temp node the program we can distinguish them. There is a separate function in the code that takes input as rule that has the temp node and returns a vector for the rule created.

   e. After this all the rules that are created are added to the paraconsistent rules vector which is returned to the main program.

For example to complete the complement predicates in the following rule

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

Steps to implement transformation for complement:

- Start by checking the rule for a negative subgoal.
- Let P be the negative predicate not t(Y) of the rule so we extract in P.
- Create the body predicates as new Predicate *b1 = new Predicate ("t_plus", {Y}, false).*
- Add it to a vector that is bodyVector1 = {b1}.
- Create head of the rule as Predicate *h1 = new Predicate ("t_complementminus", {Y}, false}*.
- Finally create the new Rule R1 with head h1 body bodyVector1 and isEDB false.

  Rule R1 = new Rule (h1, bodyVector1, false).
- Similarly we complete the negative part and its rule to the paraconsistent rules vector.

For details about this module refer section 9.7 in the appendix.

5.4.3 Fitting's Model Generation

Input: Transformed Rules

Output: A Hash map consisting of mappings for each predicate name to its respective table which is a vector of all tuples of that predicate. (Fitting's Model)

The fitting's model is generated when we apply fix-point operator on paraconsistent rules or our transformed program. For generating the model we use the DBEngine a database system that allows us to perform the basic operation like join, union, projection etc. For this we first start by populating the data in .dat files for each predicate a corresponding .dat file is created same as a table of data. We already have the EDB facts from the program so we use them to create our files. Next we start evaluating each rule in IDB from the transformed program one by one using the data in EDB and applying the operations on the body of the rules. We start by joining the positive goals then apply the minus operation with negative goals. And then finally apply projection.

Eg:

t2_plus(Z) :- g_plus(X, Y, Z), t_plus(X), t_complementplus(Y).

We perform the following operations where R1,…Rn are relations of type relation class from DBEngine. Let *R1 = g_plus, R2 = t_plus, R3 = t_complementplus.*

R4 = R1 join R2

R5 = R4 join R3.

R6 = R5.projection (V).

where, V is a vector, which contains elements from the head of rule. Thus now we can fill the table of t2_plus with relation R6. We keep a track of files that have been created using a HashMap so that in the files that have already been created we can add data, and if they are not created we create them. Also we can check from the hash map whether or not a new value is added to any of tables or not. Once we see that no new values are added to the Hash map we are sure that program has reached steady state and thus we can compute the unknowns too. For details refer to section 9.8 of appendix.

5.4.4 Models Generation

Input: Positive and unknown values extracted from the Hash map.

Output: A set of models consisting of all possible combination of these values which are each an element of a vector.

When we compute the weak well founded model we get the positive, negative and unknown values of our program which are present in respective plus, minus and unknown .dat files. Now we use these values to generate the model for test. In hash map we store map a predicate name with its values and so we get the IDB predicates from the original program and extract their arguments from the hash map to construct predicates for test. For example the hash map has a mapping for predicate t_plus with values {2} and t_unkown is mapped to {3, 5} then we construct three predicates {t(2), t(3), t(5)} and generate all possible sets of these predicates which is done in recursive manner using the functions *GenerateAllCombinations* and *DoCombine* in section 9.11. So this set will generate 2^3 models that would be tested for stability. To test the stability we also generate the Ground Program using the code in section 9.10. Ground program is generated by instantiating the rules of the original program in all possible ways from the constants. So the ground program generation is an iterative procedure where we generate the ground rules for each rule.

5.4.5 Stable Model Tester

Input: Models for test and the ground program where each model is an element of Vector.

Output: Stable models.

In this module all the models that are generated in previous step are tested for stability. We create a separate class called the stable model tester which taken in input the models for test and the ground program. Then for each model we perform the following the steps:

1. Remove the rules from the ground program vector in which negation of one the atoms sent for test is present.
2. Remove negated subgoals (predicates) from rest of the rules.
3. Finally we get a program we apply fix point operation.

For the fix point operation we again make use of the DBEngine where we store the data in original rules and create tables whenever needed for the new IDB tables formed. Here, also a hash map is created which stores the mapping of predicate name and its values. So, we again check whether mapping for IDB predicates is same as those in model sent testing. If it is same then we output the stable model.

So we can compute the stable models of a logic program using the steps described above. Now, we need to test the efficiency of our proposed approach. This is done on the next chapter.

CHAPTER 6

EXPERIMENTS

6.1 Introduction

In this chapter we present the experiments performed to test the efficiency of our approach. We perform two experiments and compute the stable models using our proposed approach and a naïve method of stable model computation. In both the experiments, the aim is to compare the time taken to compute stable models of a logic program using our approach and the Naïve approach. We use the IDB from example 5.5.2 as our logic program, and note the time taken to compute the stable models with various EDBs. The experiments are performed on Windows 7 professional operating system with 3 GB RAM and a 32-bit operating system.

We also analyze the results obtained from the experiments, which shows that our prediction of the proposed approach performs considerably better than the naïve approach in case of larger databases is correct.

6.2 Design of Experiments

We have designed two experiments for testing the efficiency of our approach:

1. Given the IDB rules we keep the number of constants to be used the program fixed to 10 and vary the number of EDB rules or facts, in increments of 5, starting from 5 and going up to 40. The argument list for the facts would be randomly generated from the given set of constants. We note the time taken to compute the stable model from our approach and the Naïve approach for each set of facts.
2. Given the IDB rules we keep the number of EDB rules or facts fixed to 30 and vary the number of constants present in the program in increments of 2, starting from 5 and going

up to 15. The data for facts would be generated randomly i.e. the argument list for the facts would be formed randomly from constant values. We note the time taken to compute the stable models from our approach and the Naïve approach for each set of constants.

Using these two designs we can see how the time varies in both the cases i.e. varying the number of facts and varying the number of constants. Also, we work on large data in real life scenarios, so, it is worthwhile to check the improvement in efficiency of stable model computation with Fitting's model used as a pre processing mechanism over a Naïve approach, which is computationally considered to be quite expensive. Next we present the procedure followed for stable model computation using our approach and the Naïve approach respectively.

Steps to perform the experiments with our approach are as follows:

1. Generate random data for facts for the IDB rules of example 5.5.2 consisting of EDB predicates *t0* and *g*, to create a logic program *P*.
2. Compile the program *P* to get the data structure *rules*.
3. Transform the original logic program *P* into a new logic program P′ which is used for computing the Fitting's model.
4. Compute the Fitting's model.
5. Using the positive and unknown values from the Fitting's model generate all possible models that are tested for stability.
6. Generate the ground program from the original logic program P.
7. Test each of the model generated in step 5 for stability and output the model if it is stable.
8. Note the time taken to perform Step 1 to Step 7.

Steps to perform the experiments with the Naïve approach.

1. Generate random data for facts for the IDB rules of example 5.5.2 consisting of EDB predicates *t0* and *g*, to create a logic program *P*.
2. Compile the program *P* and perform the semantic checks to get the data structure *rules*.
3. Using the constant values and IDB predicates generate all possible model that could be tested for stability. For example in case of the example 5.2.2 there are 5 constant values and one IDB predicate *t*, the possible values for *t* are {*1, 2, 3, 4, 5*} and the number of possible models is 2^5 i.e. 32.
4. Generate the ground program from the original logic program P.
5. Test each of the model generated in step 3 for stability and output the model if it is stable.
6. Note the time taken to perform Step 1 to Step 5.

As it can be seen from above the number of steps involved in the Naïve approach is lesser than the steps involved in our approach, but the number of models that are being tested in case of Naïve approach is 32 and in our approach they are reduced to 8 for example 5.2.2. The overhead of transformation and computing the Fitting's model is added in our approach. Thus, we perform the experiments and see whether the overhead of reducing the possible models of test is worth the work involved.

Comparing our approach with Naïve approach we can see there is an overhead of three steps transformation, weak well founded model generation and extraction of positive and unknown values to eliminate negative values. So, when models that are generated for testing only consists of positive and unknown values. On the other hand the number of steps performed in naïve approach is less than our approach. The figure shown below makes it more clear.

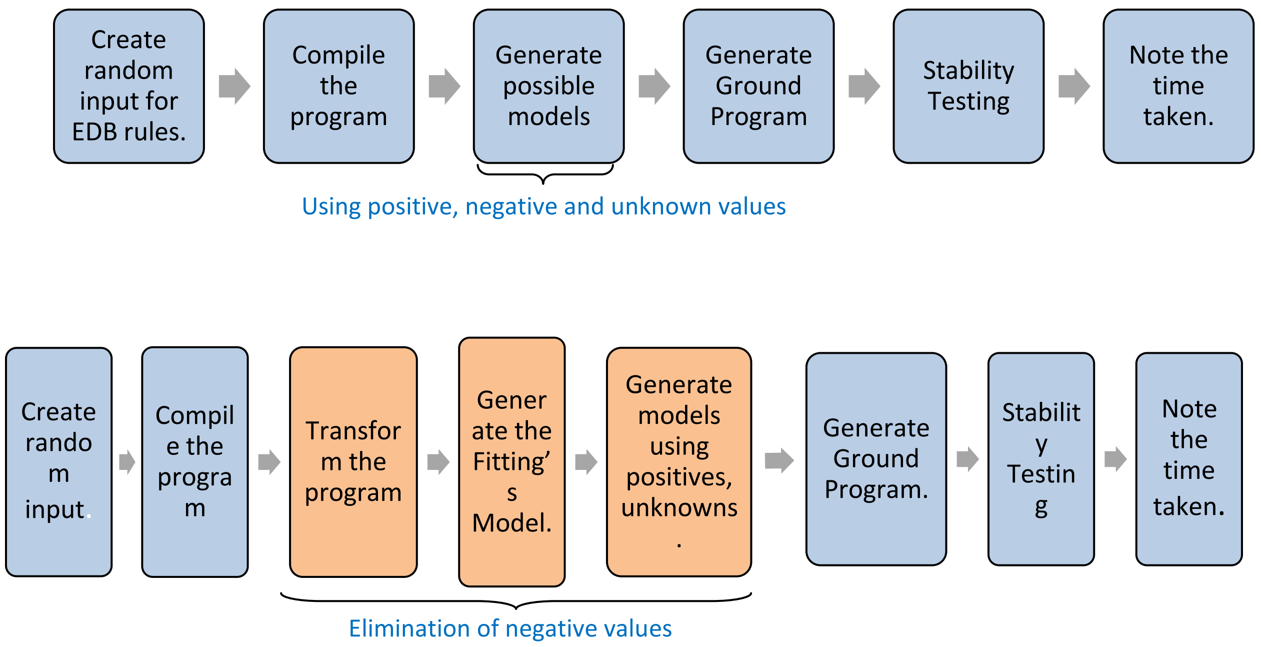


Figure 6.1 Comparison of our approach with naïve approach in terms of steps involved

To test whether the elimination of negation reduces the time to compute stable models and it is worth the work involved the experiments are performed. The time taken to do the transformation and generate the fitting's model according to our prediction should much less than testing the models with all positive, negative and unknown values.

6.3.Results

After performing the experiments we obtain the data which keeps track of number of constants, facts, stable models and the time taken. Table 6.1 shows the results for experiment 1 where we vary the number of constants and number of facts is fixed and table 6.2 shows the result for experiment 2, below in tabular form and in graphical form:

Table 6.1 Results from Experiment 1

No. of Facts	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**
Our Approach (time in seconds)	4.98	44.82	49.76	82.51	53.04	130.81	91.4	143.97
Naïve approach (time in seconds)	2.72	182.59	70.52	192.55	202.95	210.02	221.3	224.48

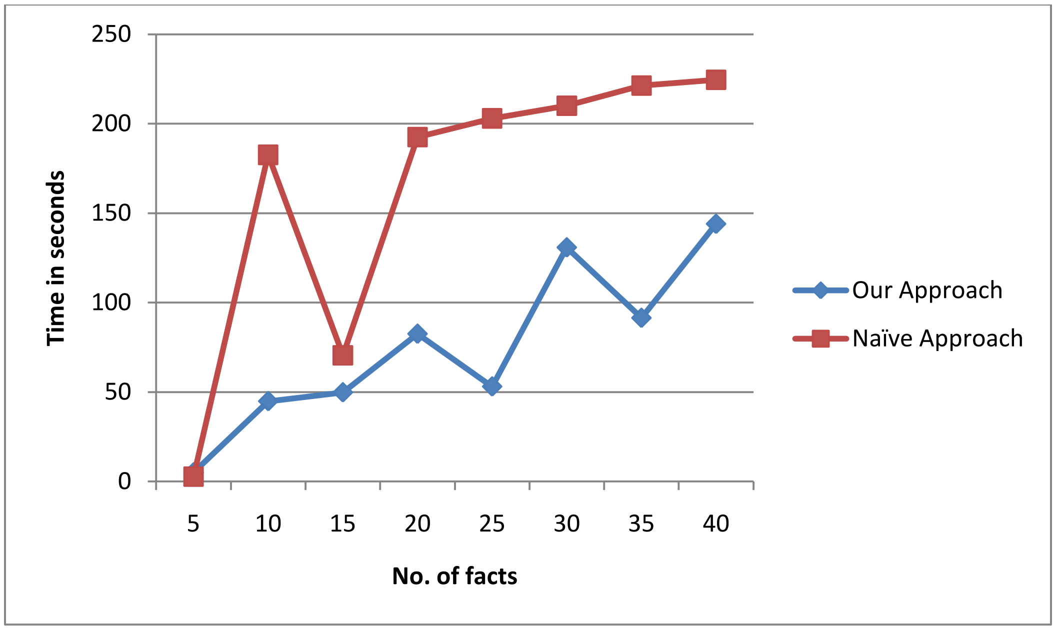


Figure 6.2 Naïve approach vs. our approach with variable number of facts

The data above shows that as we increase the number of facts in our logic program the time taken to compute the stable models increases for both Naïve approach and our approach, but the time taken by our approach is considerably lesser. Also we can see that in case of smaller data i.e. with 5 facts Naïve approach performs better than our approach and the overhead of using Fitting's model a preprocessing mechanism is more for smaller data. But, for bit larger data i.e. even with 10 facts our approach performs much better. Next are the results from experiment 2 with fixed number of facts and variable number of constants.

Table 6.2 Results from Experiment 2

No. of constants	**5**	**7**	**9**	**11**	**13**	**15**
Our Approach (time in seconds)	7.035	12.034	33.255	149.507	62.339	1051.978
Naïve Approach (time in seconds)	4.189	11.398	83.037	557.124	3568.867	21166.96

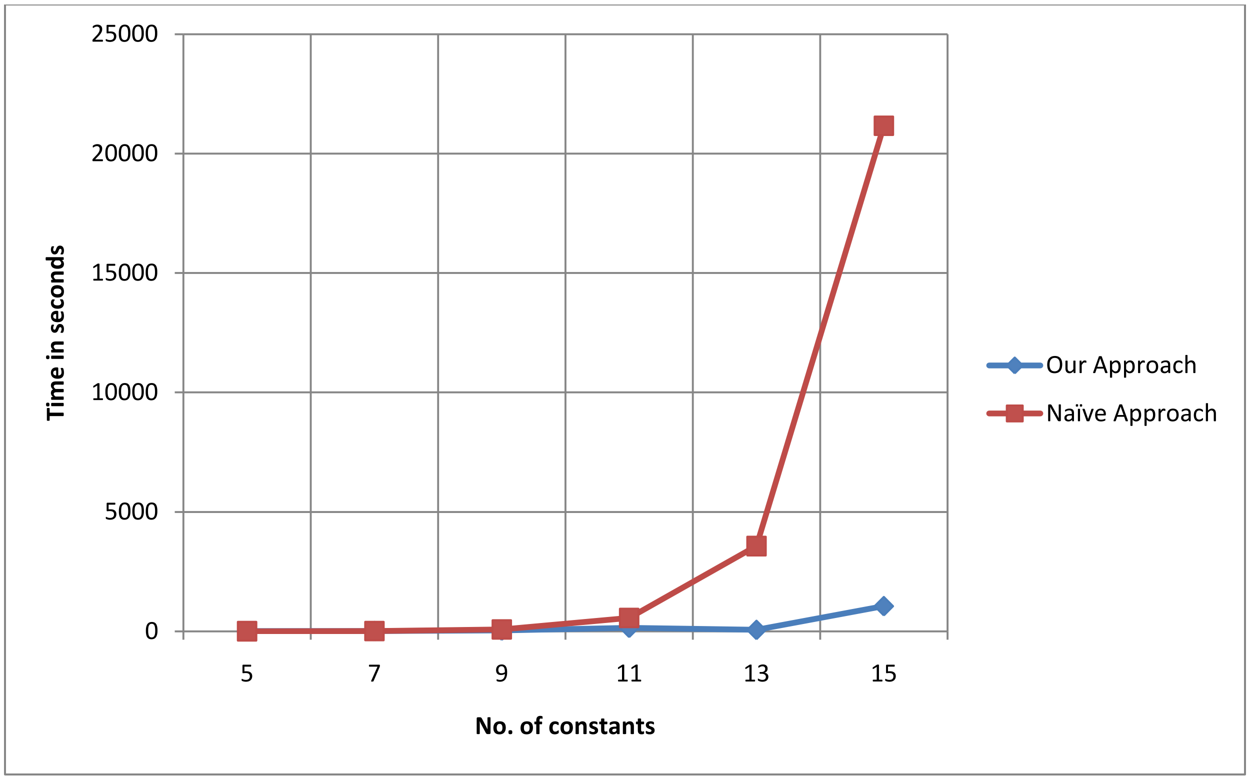


Figure 6.3 Naive approach vs. Our approach with variable number of constants

The results from experiment 2 shows that our approach performs way better than the Naïve approach in case of larger number of constants, because as the number of constants increase the possible number of models for stability testing for Naïve methods is $2^{\text{number of constants}}$, which is a considerable increase. But, in case of our approach the Fitting's model reduces the possible models for test as we consider only the positive and unknown values, which as we can see tend to remain small and so is the time to compute the stable models.

6.4 Analysis of Results

In case of fixed number of constants the we can see the time taken increases gradually for both approaches although our approach performs much better than the naïve approach by eliminating the negative values. Although it can be observed that there is not a significant amount of time change in case of variable number of facts. In the second experiment where we have fixed number of facts and variable number of facts the efficiency of our approach is much higher than the naïve approach. As it can be seen only with 15 constants the time taken by our approach is approximately 17 minutes and in case of naïve approach it is about 5 hours, which is significant. For each values that is found negative in case of given example the time taken reduces to half in comparison to naïve approach, so if one value is declared negative out 10 the possible models for test for our approach is 2^9 and for naïve approach it is 2^{10}, if we get one more as value as negative the time taken for approach is 2^8 while for naïve approach it remains 2^{10}. So the time taken is proportional to models being tested for stability and if they are reduced time also reduces proportionally. In case of more IDB predicates with greater number of variables the difference is much more significant. Number of possible models for *p* positive and *u* unknowns.

- Let $a_1, \ldots, a_n$ be the IDB predicates each having $b_1, \ldots, b_n$ variables.
- Let $m = p + u$. (m = sum of number of positive and unknown predicates)
- Number of ways each predicate can be instantiated is $c^{b1}, \ldots, c^{bn}$, where c is number of constants the program.
- Let $K = c^{b1} + \ldots + c^{bn}$.(total of all the ways).
- Now it is clear m << K
- Number of combinations of instantiated predicates $a_1, \ldots, a_n$ is 2^m i.e. models for test.
- Number of models for test for naïve approach would be 2^K.

CHAPTER 7

CONCLUSION

In this thesis we introduced a novel approach for stable model computation, which is a computationally expensive process otherwise. The process involved made use of Fitting’s model as a pre processing mechanism. We also introduce an algorithm named TRANSFORM that eliminates arbitrary negation in general deductive databases and enables us to use traditional bottom-up evaluators for computing the meaning of the general deductive databases. The experiments performed shows that our approach for stable model computation performs much better than a naïve approach which is time consuming, and in case of larger databases the difference in performance is even greater. Our approach reduce the model for test by considering only the positive and unknown values, while the naïve approach tests all the positive, negative and unknown values, thereby increasing the time taken. This time is much more than the overhead involved in computing the Fitting’s model by applying the transformation and then computing the stable models.

Stable models are one of the widely accepted semantics of general deductive databases. Recently, stable models of general deductive databases have been shown to be useful in speeding up the solutions to many NP-complete problems in graph theory [27, 28]. Thus, an algorithm to speed up stable model computation can be helpful.

In future, the algorithm can be extended to work with well-founded models instead of Fitting’s model. Also, the approach can be extended to disjunctive databases. Experimental studies can be performed to see if generating stable models by pre processing with our approach can improve over other algorithms to compute stable models.

CHAPTER 8

REFERENCES

[1] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. J. ACM, 23(4):733{742, 1976.

[2] K. L. Clark. Negation as failure. In M. L. Ginsberg, editor, Readings in Non-monotonic Reasoning, pages 311{325. Kaufmann, Los Altos, CA, 1987.

[3]J. C. Shepherdson. Negation as failure, II *Journal of Logic Programming*, 2(3) 185-202,1985.

[4] J. C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Workshop on Foundation of Deductive Databases and Logic Programming*, Washington, DC, August 1986.

[5] M. Fitting. A Kripke-KIeene semantics for Logic programs *Journal of Logic Programming*, 2(4).295-312, 1985.

[6] K Kunen. Negation in logic programming. Journal of Logic Programming, 4(4):289-398, 1987.

[7] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM,* 38(3):621-650, 1991.

[8] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. *In Proceedings of the 5th ICSLP*, pages 1070-1080, Seattle, WA, August 1988.

[9] Apt, K.R., BLAIR H.A., AND WALKER, A. 1988. Towards a theory of declarative knowledge. *In Foundations of deductive databases and logic programming*, J. Minker, Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, 89–148.

[10] S. Naqvi and S. Tsur. "*A Logical Language for Data and Knowledge Bases*," W. H. Freeman Publ., 1989.

[11] Lloyd, J. W., *Foundations of Logic Programming, Springer Verlag*, (2nd Edition}, 1987.

[12] S*accá* D., Zaniolo, C., "*Stable Models and Non-Determinism in Logic Programs with Negation*," Proc. 9th ACM SIGMOD-SIGACT Symposium On Principles of Database Systems, 1990.

[13] J. Ullman. Assigning an appropriate meaning to database logic with negation. Technical Report 1994-15, Stanford Infolab, 1994. A corrected version of a paper that appeared in "Computers as Our Better Partners" (H. Yamada, Y. Kambayashi, and S. Ohta, eds.)pp. 216-225, World Scientific, Singapore, 1994.

[14] Van Gelder A., "Negation as failure using tight derivations for general logic programs," *Proc. Symp. On Logic Programming*, IEEE, pp. 127-139.

[15] Naqvi S., Negation as failure for first-order queries," Proc. *Fifth ACM Symposium on Principles of Databases Systems*, pp. 114-122, 1986.

[16] Mc.Carthy, J., "Circumscription- a form of non-monotonic reasoning", *Artificial Intelligence* 13: 1-2, pp. 27-39,1980.

[17] Przymusinska, H. and T. Przymusinski. "Semantic issues in deductive databases and logic programs," *Sourcebook on Formal Approaches in Artificial Intelligence* (A. Banerji, ed.), 1989.

[18] V. Lifschitz, On declarative semantics of logic programs with negation. In J. Minker, editor, Foundations of Deductive Databases and Logic Programming, pages 177-192. Morgan Kaufmann, Los Altos, Ca, 1988.

[19] Baral, C. and V. S. Subramanian. "Dualities between alternative semantics for logic programming and non-monotonic reasoning," *Proc. First International Workshop on Logic Programming and Non-Monotonic Reasoning*, MIT Press, 1992.

[20] Przymusinski, T. "Well-founded semantics coincides with three-valued stable sematics," *Fundamneta Informaticae* 13, pp. 445-463.

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

[22] Rajiv Bagai and Rajshekhar Sunderraman. Bottom-up computation of the Fitting model for general deductive databases. Journal of Intelligent Information Systems, 6(1):59-75,January 1996.

[23] Jeffrey D. Ullman , Carlo Zaniolo, Deductive databases: achievements and future directions, ACM SIGMOD Record, v.19 n.4, p.75-82, Dec. 1990 .

[24] Mengchi Liu, Deductive database languages: problems and solutions, ACM Computing Surveys (CSUR), v.31 n.1, p.27-62, March 1999.

[25]   Ullman, J. 1989a. Principles of Database and Knowledge-Base Systems. Computer Science Press, Inc., New York, NY.

[26] Ceri, S., Gottlob, G., and Tanca, L. 1990. Logic programming and databases. Springer-Verlag, New York, NY.

[27] 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.

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

CHAPTER 9

APPENDIX: Java code used to implement our approach

9.1 Lexer:

```
/* --------------------------Usercode Section------------------------ */
import java_cup.runtime.*;
/* -----------------Options and Declarations Section----------------- */
%%
%class Lexer
%line
%column
%cup
%{
    private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
    }
    private Symbol symbol(int type, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
    }
%}
/*
  Macro Declarations
  These declarations are regular expressions that will be used latter
  in the Lexical Rules Section.
*/
LineTerminator = \r|\n|\r\n
WhiteSpace     = {LineTerminator} | [ \t\f]
NAME =  [a-z][a-zA-Z0-9_]*
VARIABLE = [_A-Z][a-zA-Z0-9_]*
NUMBER  = 0 | [-]?[1-9][0-9]*|[-]?[0-9]*\.[0-9]*
COMPARISON = > | < | (>=) | (<=) | <> | =
NOT = [Nn][Oo][Tt]
STRING = '[A-Za-z0-9_]'
COMMENT = %*\sFor Rule [a-z][a-zA-Z0-9_]*\([_A-Z][a-zA-Z0-9_]*,?]*\)
%%
<YYINITIAL> {
    "("                 { return symbol(sym.LPAREN); }
    ")"                 { return symbol(sym.RPAREN); }
    "$"                 { return symbol(sym.DOLLAR); }
    "."                 { return symbol(sym.PERIOD); }
    ":-"                { return symbol(sym.IMPLIES); }
    ","                 { return symbol(sym.COMMA);}
    {NOT}               { return symbol(sym.NOTOP);}
    {NAME}              { return symbol(sym.NAME, yytext()); }
    {VARIABLE}          { return symbol(sym.VARIABLE, yytext()); }
    {NUMBER}            { return symbol(sym.NUMBER, yytext()); }
    {STRING}            { return symbol(sym.STRING, yytext()); }
    {COMMENT}           { /* just skip what was found, do nothing */ }
      {COMPARISON}       { return symbol(sym.COMPARISON, yytext()); }
    {WhiteSpace}        { /* just skip what was found, do nothing */ }
}
[^]                     { /*throw new Error("Illegal character
<"+yytext()+">"); }*/
```

```
                         System.out.println("Syntax Error - Scanning
problem");
         }
```

9.2 Parser:

```
import java_cup.runtime.*;
import java.util.*;
parser code {:
    //public LinkedList lst = new LinkedList();
    public void report_error(String messages, Object info)
    {
        StringBuffer m = new StringBuffer("Error");
        if (info instanceof java_cup.runtime.Symbol)
        {
            java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
            if (s.left >= 0)
            {
                m.append(" in line " + (s.left+1));
                if(s.right >= 0)
                    m.append(", column " + (s.right+1));
            }
        }
        m.append(" : " + messages);
        System.err.println(m);
    }

    public void report_fatal_error(String message, Object info) {
        report_error(message, info);
        //System.exit(1);
    }
:};

/* Non terminals used in the grammar section. */
terminal                DOLLAR, IMPLIES, PERIOD, COMMA, LPAREN, RPAREN,
NOTOP;
terminal String         COMPARISON, NAME, NUMBER, STRING, VARIABLE;
non terminal Object     ddb;
non terminal Vector     idb_rules, idb_body, arg_list, arg;
non terminal Rule       idb_rule;
non terminal Predicate  predicate, literal;

/* -------------Precedence and Associatively of Terminals Section-----------
*/

/* ----------------------------Grammar Section-------------------- */

/* The grammar for our parser.

ddb          ::= rules DOLLAR
idb_rules    ::= idb_rule |  idb_rule idb_rules
idb_rule     ::= predicate IMPLIES idb_body PERIOD | predicate PERIOD
idb_body     ::= literal |  literal COMMA idb_body
literal      ::= NOTOP predicate | predicate
predicate    ::= NAME LPAREN arg_list RPAREN | arg COMPARISON arg
arg_list     ::= arg | arg COMMA arg_list
```

```
arg          ::= NAME | NUMBER | VARIABLE
constant     ::= NUMBER | STRING

*/
ddb ::= idb_rules:r DOLLAR
            {:
                //System.out.println("DDB");
                RESULT = r;
            :};
idb_rules ::= idb_rule:r1
            {:
                Vector R = new Vector();
                R.add(r1);
                //System.out.println("RULES");
                RESULT = R;
            :}
|  idb_rule:r1 idb_rules:rs
            {:
                Vector R = new Vector();
                R.add(r1);
                R.addAll(rs);
                RESULT = R;
            :};
idb_rule ::= predicate:head IMPLIES idb_body:body PERIOD
            {:
                Rule R =  new Rule(head,body,false);
                //System.out.println("RULE");
                RESULT = R;
            :}
| predicate:head PERIOD
            {:
                Rule R =  new Rule(head,null,true);
                RESULT = R;
            :};
idb_body ::= literal:l
            {:
                Vector P = new Vector();
                P.add(l);
                RESULT = P;
            :}
 |  literal:pr COMMA idb_body :prs
            {:
                Vector P = new Vector();
                P.add(pr);
                P.addAll(prs);
                RESULT = P;
            :};
literal ::= NOTOP LPAREN predicate:p RPAREN
            {:
                p.setIsNegative(true);
                RESULT = p;
            :}
|NOTOP predicate:p
            {:
                p.setIsNegative(true);
                RESULT = p;
            :}
```

```
| predicate:p1
            {:
                RESULT = p1;
            :};
predicate ::= NAME:n LPAREN arg_list:al RPAREN
            {:
                RESULT = new
Predicate(n,al,false,false,null,null,null,null,null);
            :};
arg_list ::= arg:a1
            {:
                Vector v = new Vector();
                v.add(a1);
                RESULT = v;
            :}
| arg:a2 COMMA arg_list:al
            {:
                Vector v = new Vector();
                v.add(a2);
                v.addAll(al);
                RESULT = v;
            :};

arg ::= NAME:n
            {:
                Vector v1 = new Vector();
                v1.add(n);
                v1.add("varchar");
                RESULT = v1;
            :}
| NUMBER:num
            {:
                Vector v2 = new Vector();
                v2.add(num);
                v2.add("varchar");
                RESULT = v2;
            :}
| VARIABLE:var
            {:
                Vector v3 = new Vector();
                v3.add(var);
                v3.add("variable");
                RESULT = v3;
            :};
```

9.3 Predicate Class

```
import java.util.Vector;
import java.util.*;
public class Predicate{
    String Name;
    Vector ArgList;
    boolean isNegative;
     public String getName() {
                return Name;
```

```
            }
            public void setName(String name) {
                  Name = name;
            }
            public Vector getArgList() {
                  return ArgList;
            }
            public void setArgList(Vector argList) {
                  ArgList = argList;
            }
            public boolean getIsNegative() {
                  return isNegative;
            }
            public void setIsNegative(boolean isNegative) {
                  this.isNegative = isNegative;
            }
    public Predicate( String name, Vector arglist, boolean isnegative){
        Name = name;
        ArgList = arglist;
        isNegative = isnegative;
    }
}
```

9.4 Rule Class:

```
import java.util.Vector;
import java.util.*;
public class Rule{
    Predicate Head;
    Vector Body;
    boolean isEDB;
    public Predicate getHead(){
        return Head;
    }
    public void setHead(Predicate Head){
        this.Head = Head;
    }
    public Vector getBody() {
        return Body;
    }
    public void setBody(Vector Body) {
        this.Body = Body;
    }
    public boolean getIsEDB() {
    return isEDB;
    }
    public void setIsEDB(boolean isEDB) {
        this.isEDB = isEDB;
    }
    public Rule(Predicate h, Vector b, boolean isedb) {
        Head =  h;
        Body =  b;
        isEDB = isedb;
    }
}
```

9.5 Main Class:

```
import java.sql.Time;
import java.util.*;
import java.io.*;
public class MAIN{
    MiscFunctions mf = new MiscFunctions();
      static public void main(String[] args){
        try{
            RandomFacts rf = new RandomFacts();
            int constants = 13;
            int t_facts = 4;
            int g_facts = 26;
            rf.setRandomFacts(constants, t_facts, g_facts);
            Vector V1 = rf.getT0_facts();
            Vector V2 = rf.getG_facts();
            CreateInput cp = new CreateInput();
            cp.CreateInputFile(V1, V2);
            System.out.println(System.getProperty("user.dir"));
            parser p = new parser(new Lexer(new FileReader(args[0])));
            Vector Rules = (Vector)(p.parse().value);
            //-----------------Safety Checks--------------------------------
            Safety sc = new Safety();
            sc.variablesCheck(Rules);
            sc.notSafety(Rules);
            sc.airtyCheck(Rules);
            //--------------------------------------------------------------
//------------Get rules for than one same IDB predicates eg: t(Z) :- t0(Z)
//------------and t(Z) :- g(X,Y,Z),t(X), not t(Y).
//Then change rule to form t1(Z) and t2(Z) and get Rules t(Z) :- t1(Z) //----
//-----;t2(Z).
                  long t1 = System.currentTimeMillis();
                  ModifyRules mr = new ModifyRules();
                  Vector MainRule = new Vector();
                  Vector Mod_Rules = (Vector)Rules.clone();
//-----------Get Transformed rules from Paraconsistent Class in TRules-------
                  Vector TRules = new Vector();
            ParaConsistentRules pr = new ParaConsistentRules();
            if(mr.hasSameHead(Rules)){//if Rules have more than one same head
modify rules
                          MainRule = mr.getMainRule(Mod_Rules);
                          Mod_Rules = mr.getNewRules(Mod_Rules);
                          TRules  = pr.getTransformedRules(Mod_Rules);
                          TRules.addAll(MainRule);
                          TRules = mr.reorderRules(MainRule,TRules);
                  }
                  else {
                  TRules = pr.getTransformedRules(Rules);
            }
                  //--------------------Send the Transformed rules to
Populate Relations Class for creating the DataBase------------------
            PopulateRelations prels = new PopulateRelations();
            prels.storeData(TRules);
                  //------Printing the Transformed rules in transformed.txt--
                  PrintRules prs = new PrintRules();
                  prs.Print(TRules);
                  //----------Compute the Weak-Well Founded Model--
                  WeakWellFoundedModel wwf = new WeakWellFoundedModel();
                  HashMap result = wwf.computeWWF(TRules);
                  System.out.println("WWF$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$");
                  long t2 = System.currentTimeMillis();
                  long diff1 = t2-t1;
                  //-----Get the ground Program for the original Rules-------
                  parser p1 = new parser(new Lexer(new FileReader(args[0])));
                  Vector Rules1 = (Vector)(p1.parse().value);
                  GroundProgram gp = new GroundProgram();
                  Vector GPRules = gp.getGroundProgram(Rules1);
//Get a Vector containing all positive Predicates and Unknown Predicates of
IDB----------
                  PredicatesForTest prt = new PredicatesForTest();
                  long t3 = System.currentTimeMillis();
                  Vector P = prt.getPositiveUnknownPredicates(Rules1,
result);
                  long t4 = System.currentTimeMillis();
                  long diff2 = t4-t3;
                  Vector EDB = prt.getEDBPredicates(Rules1);
                  //P.addAll(EDB);
                  //Populate Original data relation
                  PopulateOrgData PO = new PopulateOrgData();
                  PO.CreateDatabase(Rules1);
//-----Vector containing positive and unknown predicates from IDB.
//--Get Stable models using ground Program and data returned from Predicates
For Test--------
                  long start_time = System.currentTimeMillis();
                  StableModelTester st = new StableModelTester(GPRules,
Rules1);
                  Vector StableModels = st.Test_All(P,EDB,GPRules);
                  System.out.println("Number of stable models is
:"+StableModels.size());
                  long end_time = System.currentTimeMillis();
                  long diff = end_time - start_time;
                  diff = diff+diff1+diff2;
                  System.out.println("Time to compute:"+ diff);
                  //Calculate Stable Models with Naive Method
      System.out.println("###########################NAIVE###################
#########");
                  Vector Naive_predicates =
prt.getPredicatesForNaiveTest(Rules1);
                  StableModelTester st1 = new StableModelTester(GPRules,
Rules1);
                  long start_time_naive = System.currentTimeMillis();
                  Vector NaiveStableModels = st1.Test_All(Naive_predicates,
EDB, GPRules);
                  System.out.println("Number of stable models is :"+
NaiveStableModels.size());
                  long end_time_naive = System.currentTimeMillis();
                  long diff_naive = end_time_naive - start_time_naive;
                  System.out.println("Time to compute:"+ diff_naive);
        }
        catch(Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

9.6 Semantic Checks Class

```
import java.util.*;
import java.io.*;
public class Safety{
    public void variablesCheck(Vector Rules){
        for(int i= 0; i<Rules.size(); i++){
            Rule r = (Rule)(Rules.elementAt(i));
            if(!(r.getIsEDB())){
                Predicate head = r.getHead();
                Vector headArgList = (Vector)(head.getArgList());
                Vector bodyArgList = new Vector();
                Vector body = (Vector)(r.getBody());
                for(int j =0 ; j<body.size();j++){
                    Predicate b = (Predicate)(body.elementAt(j));
                    if(!(b.getIsNegative()) && !(b.getIsComparision())){
                        Vector bArgList = (Vector)(b.getArgList());
                        bodyArgList.addAll(bArgList);
                    }
                }
                for(int k=0; k<headArgList.size();k++){
                    String val1 = (headArgList.elementAt(k)).toString();
                    int flag = 0;
                    for(int m=0; m<bodyArgList.size(); m++){
                        String val2 = (bodyArgList.elementAt(m)).toString();
                        if(val1.equals(val2)){
                            flag = 1;
                        }
                    }
                    if(flag == 0){
                        System.out.println("Safety Error:"+ val1+" does not
occur in any positive body predicate.");
                    }
                }
            }
        }
    }
    public void notSafety(Vector Rules){
        for(int i= 0; i<Rules.size(); i++){
            Rule r = (Rule)(Rules.elementAt(i));
            if(!(r.getIsEDB())){
                Vector positiveArgList = new Vector();
                Vector negativeArgList = new Vector();
                Vector body = (Vector)(r.getBody());
                for(int j =0 ; j<body.size();j++){
                    Predicate b = (Predicate)(body.elementAt(j));
                    if(!(b.getIsNegative()) && !(b.getIsComparision())){
                        Vector bArgList = (Vector)(b.getArgList());
                        positiveArgList.addAll(bArgList);
                    }
                    else if(!(b.getIsComparision())){
                        Vector nArgList = (Vector)(b.getArgList());
                        negativeArgList.addAll(nArgList);
                    }
                }
                for(int k = 0; k<negativeArgList.size(); k++){
                    Vector in = (Vector)negativeArgList.elementAt(k);
                    String type = (String)(in.elementAt(1));
                    String val = (in.elementAt(0)).toString();
                    if(type.equals("variable")){
                        int flag = 0;
                        for(int m=0; m<positiveArgList.size(); m++){
                            Vector inner =
(Vector)(positiveArgList.elementAt(m));
                            String type1 = (String)(inner.elementAt(1));
                            String val1 = (inner.elementAt(0)).toString();
                            if(type1.equals("variable")){
                                if(val1.equals(val)){
                                    flag = 1;
                                }
                            }
                        }
                        if(flag == 0){
                            System.out.println("Safety Error: Variable " +
val +" does not appear in any positive predicates.");
                        }
                    }
                }
            }
        }
    }
    public void airtyCheck(Vector Rules)  {
       Vector Airty = new Vector();
       Vector IDBAirty = new Vector();
       for(int i= 0; i<Rules.size(); i++){
            Rule r = (Rule)(Rules.elementAt(i));
            if(r.getIsEDB()){
                Predicate head = (Predicate)r.getHead();
                Vector val = new Vector();
                val.add(head.getName());
                val.add(((Vector)(head.getArgList())).size());
                Airty.add(val);
            }
        }
        for(int i= 0; i<Rules.size(); i++)
        {
            Rule r = (Rule)(Rules.elementAt(i));
            if(!(r.getIsEDB())){
                Predicate head = (Predicate)r.getHead();
                Vector v = new Vector();
                v.add(head.getName());
                v.add(((Vector)(head.getArgList())).size());
                IDBAirty.add(v);
                Vector body = (Vector)(r.getBody());
                for(int j=0; j<body.size(); j++){
                    Predicate b = (Predicate)(body.elementAt(j));
                    Vector bv = new Vector();
                              if(!(b.getIsComparision())){
                    bv.add(b.getName());
                    bv.add(((Vector)(b.getArgList())).size());
                    IDBAirty.add(bv);
                              }
                }
            }
        }
        for(int i=0; i<Airty.size(); i++){
            Vector in = (Vector)(Airty.elementAt(i));
            String name = (String)(in.elementAt(0));
            String s = (in.elementAt(1)).toString();
            int size = Integer.parseInt(s);
            for(int j=0; j<IDBAirty.size();j++){
                Vector inner = (Vector)(IDBAirty.elementAt(j));
                String pname = (String)(inner.elementAt(0));
                String s1 = (inner.elementAt(1)).toString();
                int size1 = Integer.parseInt(s1);
                if(name.equals(pname)){
                    if(size != size1){
                        System.out.println("Airty mismatch for predicate " +
pname +" .");
                    }
                }
            }
        }
        for(int i=0; i<IDBAirty.size(); i++){
            Vector in = (Vector)(IDBAirty.elementAt(i));
            String name = (String)(in.elementAt(0));
            String s = (in.elementAt(1)).toString();
            int size = Integer.parseInt(s);
            for(int j=0; j<IDBAirty.size();j++){
                Vector inner = (Vector)(IDBAirty.elementAt(j));
                String pname = (String)(inner.elementAt(0));
                String s1 = (inner.elementAt(1)).toString();
                int size1 = Integer.parseInt(s1);
                if(name.equals(pname)){
                    if(size != size1){
                        System.out.println("Airty mismatch for predicate " +
pname +" .");
                    }
                }
            }
        }
    }
}
```

9.7 Transformation Class:

```
import java.util.*;
import java.io.*;
import java.lang.*;
public class ParaConsistentRules
{
    public Vector getTransformedRules(Vector Rules){
        Vector Constants = new Vector(); // Vector containing all constants
fro making dom rules.
        Vector EDBRules = new Vector(); // Vector containing all rules of
type r(1,2) to r_plus(1,2).
        Vector TransformedRules = new Vector();
        for(int i = 0; i < Rules.size(); i++){
            Rule R = (Rule)(Rules.elementAt(i));
            if(R.getIsEDB()){
                Predicate fact = R.getHead();
                String pname = fact.getName() + "_plus";
                Vector AL = new Vector();
                Vector ArgList = fact.getArgList();
                for(int j = 0; j<ArgList.size(); j++){
                    if(!(Constants.contains(ArgList.elementAt(j)))){
                        Constants.add(ArgList.elementAt(j));
                    }
                    Vector inner = (Vector)(ArgList.elementAt(j));
                    AL.add(inner.elementAt(0));
                }
                Predicate P = new Predicate(pname, AL,
false,false,null,null,null,null,null);
                Rule R1 = new Rule(P,null,true);
                EDBRules.add(R1);
            }
        }
        // Code to add all dom rules.
        // Eg: dom(1), dom(2), dom(3)......
        for(int k=0; k<Constants.size(); k++){
            Vector inner = (Vector)(Constants.elementAt(k));
            Vector arglist = new Vector();
            arglist.add(inner.elementAt(0));
            Predicate P = new
Predicate("dom",arglist,false,false,null,null,null,null,null);
            Rule R = new Rule(P,null,true);
            TransformedRules.add(R);
        }
        Vector minusRules = AddEDBMinus(TransformedRules, EDBRules,Rules);
        TransformedRules.addAll(EDBRules);
        TransformedRules.addAll(minusRules);
        Vector IDBRules = TransformedIDBRules(Rules);
        TransformedRules.addAll(IDBRules);
        return TransformedRules;
    }
        // Code to add minus predicate for non-dom predicates
        // Eg : r_minus(X,Y) :- dom(X), dom(Y), not r_plus(X,Y).
    public Vector AddEDBMinus(Vector domRules, Vector EDBRules, Vector
Rules){
        Vector minusRules = new Vector();
        for(int k =0; k<EDBRules.size();k++){
            Rule EDBrule = (Rule)(EDBRules.elementAt(k));
            Predicate head = (Predicate)EDBrule.getHead();
            String name = head.getName();
            int ind = name.indexOf("_plus",0);
            name = name.substring(0,ind);
            int flag = 0;
            for(int q=0; q< minusRules.size();q++){
                Rule minusRule = (Rule)(minusRules.elementAt(q));
                Predicate minushead = minusRule.getHead();
                if(minushead.getName().equals(name+"_minus")){
                    flag =1;
                }
            }
            if(flag == 0)   {
                Vector newAL = new Vector();
                Vector minusBody = new Vector();
                for(int i = 0;i<Rules.size();i++){
                    Rule R = (Rule)(Rules.elementAt(i));
                    if(!(R.isEDB)){
                        Vector body = R.getBody();
                        for(int j=0; j< body.size(); j++){
                            Predicate P = (Predicate)(body.elementAt(j));
                                          if(!(P.getIsComparision())){
                                if(P.getName().equals(name)){
                                    Vector AL = P.getArgList();
                                    if(newAL.size() == 0){
                                        for(int a =0; a<AL.size(); a++){

                                            newAL.add(AL.elementAt(a));
                                        }
                                    }
                                }
                                          }
                        }
                    }
                }
                for(int a1 =0; a1<newAL.size(); a1++) {
                    Vector domlist = new Vector();
                    domlist.add(newAL.elementAt(a1));
                    Predicate Pdom = new
Predicate("dom",domlist,false,false,null,null,null,null,null);
                    minusBody.add(Pdom);
                }
                Predicate P1 = new
Predicate(name+"_plus",newAL,true,false,null,null,null,null,null);
                minusBody.add(P1);
                Predicate head1 = new
Predicate(name+"_minus",newAL,true,false,null,null,null,null,null);
                Rule R1 = new Rule(head1, minusBody, false);
                minusRules.add(R1);
            }
        }
        return minusRules;
    }
    public Vector TransformedIDBRules(Vector Rules){
        Vector TransRules = new Vector();
        for(int i=0; i< Rules.size(); i++){
           Vector CPredicates = new Vector();//Vector to store comparision
predicates of a rule.
               boolean hasComparision = false;
                  Rule R = (Rule)(Rules.elementAt(i));
            if(!(R.isEDB)){
                Vector Body = R.getBody();
                for(int j = 0; j < Body.size(); j++){
                    Predicate P = (Predicate)(Body.elementAt(j));
                    if(P.getIsNegative()) {
                        Predicate b1 = new
Predicate(P.getName()+"_minus",P.getArgList(),false,false,null,null,null,null
,null);
                        Vector b1body = new Vector();
                        b1body.add(b1);
                        Predicate h1 = new
Predicate(P.getName()+"_complementplus",
P.getArgList(),false,false,null,null,null,null,null);
                        Predicate b2 = new
Predicate(P.getName()+"_plus",P.getArgList(),false,false,null,null,null,null,
null);
                        Vector b2body = new Vector();
                        b2body.add(b2);
                        Predicate h2 = new
Predicate(P.getName()+"_complementminus",P.getArgList(),false,false,null,null
,null,null,null);
                        Rule r1 = new Rule(h1,b1body,false);
                        Rule r2 = new Rule(h2,b2body,false);
                        TransRules.add(r1);
                        TransRules.add(r2);
                    }
                }
                Predicate Head = (Predicate)R.getHead();
                Vector headAL = Head.getArgList();
                boolean projection = false;
                for(int m=0; m<Body.size(); m++){
                    Predicate P1 = (Predicate)(Body.elementAt(m));
                              Vector bodyAL = new Vector();
                              if(!(P1.getIsComparision())){
                                    bodyAL= P1.getArgList();
                              }
                    if(bodyAL.size() > headAL.size()){
                        projection = true;
                    }
                }
                        for(int g =0; g< Body.size(); g++){
                              Predicate P = (Predicate)(Body.elementAt(g));
                              if(P.getIsComparision()){
                                    CPredicates.add(P);
                                    hasComparision = true;
                              }
                        }
                Vector plusBody = new Vector();
                if(projection){
                    Vector ag = getArgumentsforTemp(R);
                              for(int n = 0; n<Body.size(); n++){
                        Predicate P = (Predicate)(Body.elementAt(n));
```

```
                                    if(!(P.getIsComparision())){
                                          Vector ArgList = new Vector();
                                          ArgList = P.getArgList();
                                          Vector bodyminus = new Vector();
                                          Predicate hminus ;
                                          Predicate b1;
                                          Vector newArgList = ArgList;
                                          if(P.getIsNegative()){
      b1 = new Predicate(P.getName()+"_complementplus", newArgList, false,
      false, null,null,null,null,null);
      hminus = new Predicate("temp"+i+"_minus" ,ag, false, false,
      null,null,null,null,null);
      Predicate bminus = new Predicate(P.getName()+"_complementminus",
      newArgList, false, false, null,null,null,null,null);
      bodyminus.add(bminus);
}
else{
      b1 = new Predicate(P.getName()+"_plus",newArgList, false, false,
      null,null,null,null,null);
      hminus = new Predicate("temp"+i+"_minus" , ag, false, false,
      null,null,null,null,null);
      Predicate bminus = new Predicate(P.getName()+"_minus", newArgList,
      false, false, null,null,null,null,null);
      bodyminus.add(bminus);
}
if(hminus.getArgList().size() > P.getArgList().size()){
      Vector T = hminus.getArgList();
      Vector Dom = new Vector();
      for(int b = 0; b<T.size();b++){
      if(!(newArgList.contains(T.elementAt(b)))){
                  Dom.add(T.elementAt(b));
                                          }
                              }
            for(int c= 0; c< Dom.size(); c++){
                  Vector domlist = new Vector();
                  domlist.add(Dom.elementAt(c));
                  Predicate pdom = new
Predicate("dom",domlist,false,false,null,null,null,null,null);
                  bodyminus.add(pdom);
                  }
      }
      Rule mRuleminus = new Rule(hminus, bodyminus, false);
      TransRules.add(mRuleminus);
      plusBody.add(b1);
      }
}
    Predicate hplus = new Predicate("temp"+i+"_plus",ag,false, false,
null,null,null,null,null);
    Rule mRuleplus = new Rule(hplus, plusBody, false);
```

```
                    TransRules.add(mRuleplus);
                    Vector projectionRules = 
getProjectionRules(R,i,ag,hasComparision,CPredicates);
                    TransRules.addAll(projectionRules);
                    //Add plus body vector rule
                }
                else{
                    for(int n = 0; n<Body.size(); n++){
                        Predicate P = (Predicate)(Body.elementAt(n));
                                if(!(P.getIsComparision())){
                                    Vector ArgList = P.getArgList();
                                    Vector bodyminus = new Vector();
                                    Predicate hminus;
                                    Predicate b1;
                                    Vector newArgList = P.getArgList();
                                    if(P.getIsNegative()){
      b1 = new Predicate(P.getName()+"_complementplus", newArgList, false, 
false, null,null,null,null,null);
      hminus = new Predicate(Head.getName()+"_minus" ,Head.getArgList(), 
false, false, null,null,null,null,null);
      Predicate bminus = new Predicate(P.getName()+"_complementminus", 
newArgList, false, false, null,null,null,null,null);
      bodyminus.add(bminus);
      }
      else{
      b1 = new Predicate(P.getName()+"_plus",newArgList, false, false, 
null,null,null,null,null);
      hminus = new Predicate(Head.getName()+"_minus" ,Head.getArgList(), 
false, false, null,null,null,null,null);
      Predicate bminus = new Predicate(P.getName()+"_minus", newArgList, 
false, false, null,null,null,null,null);
      bodyminus.add(bminus);
      }
      if(Head.getArgList().size() > P.getArgList().size()){
      Vector T = new Vector();
      Vector Dom = new Vector();
      for(int z = 0; z< Head.getArgList().size();z++){
            try{
                  T.add(Head.getArgList().elementAt(z));
            }
            catch(Exception e){
                        }
            }
            for(int b = 0; b<T.size();b++){
            if(!(newArgList.contains(T.elementAt(b)))){
            Dom.add(T.elementAt(b));
            }
            }
            for(int c= 0; c< Dom.size(); c++){
```

```
            Vector domlist = new Vector();

      domlist.add(Dom.elementAt(c));
      Predicate pdom = new
      Predicate("dom",domlist,false,false,null,null,null,null,null);
      bodyminus.add(pdom);
      }
      }
      Vector list = Head.getArgList();
       Predicate hplus = new Predicate(Head.getName()+"_plus",list,false,
false, null,null,null,null,null);
   Rule mRuleplus = new Rule(hplus, plusBody, false);
    TransRules.add(mRuleplus);
                }//Get Unknown Rule
            }
        }
            for(int j= 0; j<Rules.size(); j++){
                  Rule R = (Rule)(Rules.elementAt(j));
                  if(!(R.getIsEDB())){
                        Rule U_rule = getUnknownRule(R);
                        TransRules.add(U_rule);
                  }
            }
    return TransRules;
    }
    public Rule getUnknownRule(Rule R){
            Predicate head = (Predicate)(R.getHead());
            Vector RBody = new Vector();
            Vector Arglist = head.getArgList();
            for(int j=0; j<Arglist.size(); j++){
                  Vector AL = new Vector();
                  AL.add(Arglist.elementAt(j));
                  Predicate dom = new
Predicate("dom",AL,false,false,null,null,null,null,null);
                  RBody.add(dom);
            }
            Predicate R_plus = new
Predicate(head.getName()+"_plus",Arglist,true,false,null,null,null,null,null)
;
            Predicate R_minus = new Predicate(head.getName()+"_minus",
Arglist,true,false,null,null,null,null,null);
            RBody.add(R_plus);
            RBody.add(R_minus);
            Predicate R_head = new
Predicate(head.getName()+"_unknown",Arglist,false,false,null,null,null,null,n
ull);
            Rule R1 = new Rule(R_head,RBody,false);
            return R1;
      }
    public Vector getArgumentsforTemp(Rule R){
        Predicate head = (Predicate)(R.getHead());
        Vector body = (Vector)(R.getBody());
        Vector argTemp = new Vector();
        for(int i= 0; i<body.size(); i++){
            Predicate P = (Predicate)(body.elementAt(i));
                  if(!(P.getIsComparision())){
                        Vector parg = P.getArgList();
                        for(int j= 0; j< parg.size(); j++){
                              if(!(argTemp.contains(parg.elementAt(j)))){
                                    argTemp.add(parg.elementAt(j));
                              }
                        }
                  }
        }
            Vector variables = new Vector();
            for(int k=0; k<argTemp.size(); k++){
                  Vector inner = (Vector)(argTemp.elementAt(k));
                  variables.add(inner.elementAt(0));
            }
            Object[] arr = new String[argTemp.size()];
            arr = variables.toArray();
            String[] sa = new String[argTemp.size()];
            for(int m=0; m<arr.length;m++){
                  sa[m] = arr[m].toString();
            }
            Arrays.sort(sa);
            Vector argtemp1 = new Vector();
            for(int k=0; k<sa.length; k++){
                  Vector v  = new Vector();
                  v.add(sa[k]);
                  v.add("variable");
                  argtemp1.add(v);
            }
        return argtemp1;
    }
    public Vector getProjectionRules(Rule R, int i, Vector arguments, boolean
hasComparision, Vector CP){
        Vector Rules = new Vector();
        Predicate head = (Predicate)(R.getHead());
        Vector headargs = head.getArgList();
       //For Positive Rule
        Vector plusargs = new Vector();
        //plusargs.addAll(headargs);
        for(int k =0; k< arguments.size();k++){
            if(!(headargs.contains(arguments.elementAt(k)))) {
                Vector plusargs1 = new Vector();
                        plusargs1.add("_");
                        plusargs1.add("variable");
                        plusargs.add(plusargs1);
            }
                  else{
                        plusargs.add(arguments.elementAt(k));
                  }
        }
        Predicate bodyplus = new
Predicate("temp"+i+"_plus",plusargs,false,false, null,null,null,null,null);
        Vector bodyp = new Vector();
        bodyp.add(bodyplus);
            if(hasComparision){
                  bodyp.addAll(CP);
            }
        Predicate plushead = new
Predicate(head.getName()+"_plus",headargs,false,false,
null,null,null,null,null);
        Rule plusRule = new Rule(plushead, bodyp, false);
        Rules.add(plusRule);
       //For Minus Rules ----Rule 1----------------------------------------
        Vector b1 = new Vector();
        for(int l = 0 ; l< arguments.size(); l++) {
            Vector domList= new Vector();
            domList.add(arguments.elementAt(l));

            Predicate P = new Predicate("dom", domList, false,false,
null,null,null,null,null);
            b1.add(P);
        }
        Predicate h1 = new
Predicate("temp"+i+1,arguments,false,false,null,null,null,null,null);
        Rule R1 = new Rule(h1,b1,false);
        Rules.add(R1);
        ///For Rule 2-----------------------------------------------------
        Vector b2 = new Vector();
        b2.add(R1.getHead());
        Predicate body2 = new Predicate("temp"+i+"_minus",arguments,
true,false,null,null,null,null,null);
        b2.add(body2);
        Predicate head2 = new
Predicate("temp"+i+2,headargs,false,false,null,null,null,null,null);
        Rule R2 = new Rule(head2, b2, false);
        Rules.add(R2);
        //For Rule 3------------------------------------------------------
        Vector b3 = new Vector();
        for(int j= 0; j < headargs.size(); j++){
            Vector domList = new Vector();
            domList.add(headargs.elementAt(j));
            Predicate P = new Predicate("dom", domList, false,false,
null,null,null,null,null);
```

```
            b3.add(P);
        }
        Predicate body3 = new
Predicate("temp"+i+2,headargs,true,false,null,null,null,null,null);
        b3.add(body3);
        Predicate head3 = new
Predicate(head.getName()+"_minus",headargs,false,false,null,null,null,null,nu
ll);
            if(hasComparision){
                  Vector CP1 = getCompPredicates(CP);
                  b3.addAll(CP1);
            }
        Rule R3 = new Rule(head3, b3, false);
        Rules.add(R3);
        return Rules;
    }
}
```

9.8 Weak Well Founded Model:

```
import java.util.*;
import java.io.*;

public class WeakWellFoundedModel
{
      public HashMap computeWWF(Vector Rules)
      {
            HashMap resultMap = new HashMap();
            try{
                  Relation Rel = null;
                  Rel.initializeDatabase("DATA");
                  boolean flag = true;
                  Vector wffPredicates = new Vector();
                  for(int j=0; j<Rules.size();j++){
                        Rule r = (Rule)(Rules.elementAt(j));
                        if(r.getIsEDB()){
                              Predicate p = (Predicate)(r.getHead());
                              String name = p.getName();
                              Vector argList = p.getArgList();
                              if(!(resultMap.containsKey(name))){
                                    Vector val = new Vector();
                                    val.add(argList);
                                    resultMap.put(name,val);
                              }
                              else{
                                    Vector val1 =
(Vector)(resultMap.get(name));
                                    val1.add(argList);
                                    resultMap.put(name,val1);
                              }
                              wffPredicates.add(p);
                        }
                  }
            //While Loop to find the Well Founded Model until no more changes
occur to data.
                  while(flag){
                        flag= false;
                        for(int m = 0; m< Rules.size(); m++){
                              Rel.initializeDatabase("DATA");
                              Rule R = (Rule)(Rules.elementAt(m));
                              if(!(R.getIsEDB())){
                                    Predicate head =
(Predicate)(R.getHead());

      if(!(head.getName().endsWith("unknown"))){
                                          Vector body =
(Vector)(R.getBody());
                                          Relation R1 = null;
                                          for(int j=0; j< body.size(); j++){
                                                Predicate P =
(Predicate)(body.elementAt(j));
if((!(P.getIsNegative())) && (!(P.getIsComparision())) && R1 == null){
      R1 = Rel.getRelation(P.getName().toUpperCase());
      R1 = R1.rename(getVarList(P.getArgList()));
      }
else if((!(P.getIsNegative())) && (!(P.getIsComparision()))){
      Relation R2 = Rel.getRelation(P.getName().toUpperCase());
      Relation R3 = R2.rename(getVarList(P.getArgList()));
      R1 = R1.join(R3);
      }
      }
      for(int k=0; k<body.size();k++){
            Predicate P = (Predicate)(body.elementAt(k));
            if(P.getIsNegative()){
            Relation minusRel = Rel.getRelation(P.getName().toUpperCase());
            minusRel = minusRel.rename(getVarList(P.getArgList()));
            Relation plusRel = R1.projection(getVarList(P.getArgList()));
            minusRel = plusRel.minus(minusRel);
            R1 = R1.join(minusRel);
            }
      }
      R1 = R1.projection(getVarList(head.getArgList()));
      Vector RelTable = R1.getTable();
      if(!(resultMap.containsKey(head.getName()))){
      resultMap.put(head.getName(),RelTable);
            if(RelTable != null){
                  flag = true;
                  }
            }
            else{
```

```
            Vector V = (Vector)(resultMap.get(head.getName()));
            for(int c=0; c< RelTable.size(); c++){
            Vector inner = (Vector)(RelTable.elementAt(c));

      if(!(V.contains(inner))){
            V.add(inner);
            flag = true;
            }
      }
      resultMap.put(head.getName(),V);
      }
      Set keyset = resultMap.keySet();
      Iterator i1 = keyset.iterator();
      while(i1.hasNext()){
      String RelName = (String)(i1.next());
      Vector v1 = (Vector)(resultMap.get(RelName));
      FileOutputStream fs1 = new
FileOutputStream("DATA\\"+RelName.toUpperCase()+".dat");
      PrintStream ps1 = new PrintStream(fs1);
      ps1.println(v1.size());
      for(int d = 0; d<v1.size(); d++){
            Vector v2 = (Vector)(v1.elementAt(d));
            for(int e =0; e< v2.size(); e++){
                  ps1.println(v2.elementAt(e));
                  }
                  }
            ps1.close();
            fs1.close();
            }
      }
}
}}
for(int m=0; m<Rules.size();m++){
      Rel.initializeDatabase("DATA");
      Rule R = (Rule)(Rules.elementAt(m));
      if(!(R.getIsEDB())){
            Predicate head = (Predicate)(R.getHead());
            if(head.getName().endsWith("unknown")){
            System.out.println(head.getName());
            Vector body = (Vector)(R.getBody());
            Relation R1 = null;
            for(int j=0; j< body.size(); j++){
            Predicate P = (Predicate)(body.elementAt(j));
            if((!(P.getIsNegative())) && (!(P.getIsComparision())) && R1 ==
null){
            R1 = Rel.getRelation(P.getName().toUpperCase());
            R1 = R1.rename(getVarList(P.getArgList()));
            }
            else if((!(P.getIsNegative())) && (!(P.getIsComparision()))){
```

```
            Relation R2 = Rel.getRelation(P.getName().toUpperCase());
            Relation R3 = R2.rename(getVarList(P.getArgList()));
                                    R1 = R1.join(R3);
                              }
                        }
                        for(int k=0; k<body.size();k++){
                              Predicate P =
(Predicate)(body.elementAt(k));
      if(P.getIsNegative()){
      Relation minusRel = Rel.getRelation(P.getName().toUpperCase());
      minusRel = minusRel.rename(getVarList(P.getArgList()));
      Relation plusRel = R1.projection(getVarList(P.getArgList()));
      minusRel = plusRel.minus(minusRel);
      R1 = R1.join(minusRel);
            }
      }
      R1 = R1.projection(getVarList(head.getArgList()));
      R1.displayRelation();
      Vector RelTable = R1.getTable();
      if(!(resultMap.containsKey(head.getName()))){
      resultMap.put(head.getName(),RelTable);
      }
      else{
      Vector V = (Vector)(resultMap.get(head.getName()));
      for(int c=0; c< RelTable.size(); c++){
            Vector inner = (Vector)(RelTable.elementAt(c));
                                    if(!(V.contains(inner))){
                                          V.add(inner);
                                    }
                              }
                              resultMap.put(head.getName(),V);
                        }
                        Set keyset = resultMap.keySet();
                        Iterator i1 = keyset.iterator();
                        while(i1.hasNext()){
                        String RelName = (String)(i1.next());
                  Vector v1 = (Vector)(resultMap.get(RelName));
                  FileOutputStream fs1 = new
FileOutputStream("DATA\\"+RelName.toUpperCase()+".dat");
            PrintStream ps1 = new PrintStream(fs1);
                        ps1.println(v1.size());
                              for(int d = 0; d<v1.size(); d++){
                                    Vector v2 =
(Vector)(v1.elementAt(d));
                                    for(int e =0; e< v2.size();
e++){

      ps1.println(v2.elementAt(e));
                                    }
                                          }
                                          ps1.close();
                                          fs1.close();
                                    }
                              }
                        }
                  }
                  MiscFunctions mf = new MiscFunctions();
                  mf.printWeakWellFounded(resultMap);
            }
            catch(Exception ex){
                  ex.printStackTrace();
            }
            return resultMap;
      }
      public Vector getVarList(Vector argList) {
        Vector T =  new Vector();
        for(int i = 0; i<argList.size(); i++) {
                  if(argList.elementAt(i).equals("_")){
                        T.add("_");
                  }
                  else{
                        Vector in = (Vector)(argList.elementAt(i));
                        T.add(in.elementAt(0));
                  }
        }
        return T;
    }
      public Relation evaluateCols(Relation Rel,Vector ArgList){
            Vector variables = new Vector();
            Vector var1 = new Vector();
            for(int i=0;i<ArgList.size();i++){
                  Vector inner = (Vector)(ArgList.elementAt(i));
                  if(inner.elementAt(1).equals("varchar")){
                        String LOT = "col";
                        String ROT = "str";
                        int j= i+1;
                        String LOP = "C"+ Integer.toString(j);
                        String ROP = (String)inner.elementAt(0);
                        String OP = "=";
                        Rel = Rel.selection(LOT,LOP,OP,ROT,ROP);
                  }
                  else{
                        int j = i+1;
                        variables.add("C"+Integer.toString(j));
                        var1.add(inner.elementAt(0));
                  }
            }
            Rel = Rel.projection(variables);
```

```
            return Rel;
      }
}
```

9.9 Generate Models for Stability Testing:

```
import java.util.*;
import java.io.*;
public class PredicatesForTest {
      public Vector getPositiveUnknownPredicates(Vector Rules, HashMap
result){
            ModifyRules mrf = new ModifyRules();
            Vector headNames = new Vector();
            headNames = mrf.getUniqueHeads(Rules);
            Vector P = new Vector();
            for(int i=0; i< headNames.size();i++){
                  String name = (String)headNames.elementAt(i);
                  String headplus = name + "_plus";
                  String headunknown = name + "_unknown";
                  Vector plusArgs = (Vector)(result.get(headplus));
                  Vector unknownArgs = (Vector)(result.get(headunknown));
                  for(int m=0;m<unknownArgs.size();m++){
                        if(!(plusArgs.contains(unknownArgs.elementAt(m)))){
                              plusArgs.add(unknownArgs.elementAt(m));
                        }
                  }
                  //System.out.println("HEAD:"+head.getName());
                  for(int j=0; j< plusArgs.size();j++){
                        Vector args = (Vector)(plusArgs.elementAt(j));
                        Predicate p1 = new
Predicate(name,args,false,false,null,null,null,null,null);
                        int index = P.indexOf(p1);
                        if(!(P.contains(p1))){
                              P.add(p1);
                        }
                  }
            }
            return P;
      }
      public Vector getEDBPredicates(Vector Rules){
            Vector EDB = new Vector();
            for(int i=0;i<Rules.size();i++){
                  Rule R = (Rule)(Rules.elementAt(i));
                  if(R.getIsEDB()){
                        Predicate head = R.getHead();
                        EDB.add(head);
                  }
            }
            return EDB;
      }
      public Vector getPredicatesForNaiveTest(Vector Rules){
            Vector NaivePredicates = new Vector();
            Vector args = new Vector();
            Vector EDB1 = getEDBPredicates(Rules);
            for(int i=0; i<EDB1.size(); i++){
                  Predicate p = (Predicate)(EDB1.elementAt(i));
                  Vector ArgList = p.getArgList();
                  for(int j=0; j< ArgList.size(); j++){
                        if(!args.contains(ArgList.elementAt(j))){
                              args.add(ArgList.elementAt(j));
                        }
                  }
            }
            for(int i=0; i< args.size(); i++){
                  Vector v = new Vector();
                  v.add(args.elementAt(i));
                  Predicate P = new Predicate("t",v,false,false,
null,null,null,null,null);
                  NaivePredicates.add(P);
            }
            return NaivePredicates;
      }
}
```

9.10 Ground Program Generation:

```
import java.util.*;
import java.io.*;
//A class to transform a Program into its ground program i.e. replacing all
the variables with
// constants.
public class GroundProgram{
      public Vector Perms = new Vector();
      MiscFunctions mf = new MiscFunctions();
      public Vector getGroundProgram(Vector Rules){
            Vector GP = new Vector();
            String[] constants = getConstants(Rules);
            for(int i = 0; i<Rules.size();i++){
                  Rule R = (Rule)(Rules.elementAt(i));
                  if(!(R.getIsEDB())){
                        Predicate head = R.getHead();
                        Vector body = R.getBody();
                        Perms = new Vector();
                        Predicate p = getMaxPredicate(R);
                        int size = ((Vector)(p.getArgList())).size();
                        getCombos(size,constants);
                        for(int j=0;j<Perms.size();j++){
                              Vector argList = mf.getVarList(p.getArgList());
                              String[] arr = (String[])(Perms.elementAt(j));
                              HashMap map = new HashMap();
                              for(int k = 0; k<arr.length;k++){
```

```
                              map.put(argList.elementAt(k),arr[k]);
                        }
                        //-------Create new Head with new constant 
arguments
                        Vector headArgs = 
mf.getVarList(head.getArgList());//get original head arguments
                        Vector newHeadArgs = new Vector();//place new 
contants arguments keeping the mapping as X to 1st argument, Y to
                        // 2nd argument
                        for(int m=0; m<headArgs.size();m++){
                              String val = 
(String)(map.get(headArgs.elementAt(m)));
                              newHeadArgs.add(val);
                        }
                        Predicate newHead = new 
Predicate(head.getName(),newHeadArgs,false,false,null,null,null,null,null);
                        //-------------Create new Body Predicate for 
each permutation------------------
                        Vector newBody = new Vector();
                        for(int bp =0 ;bp<body.size(); bp++){
                              Predicate P_body = 
(Predicate)(body.elementAt(bp));
                              if(!(P_body.getIsComparision())){
                                    Vector b_args = 
mf.getVarList(P_body.getArgList());
                                    Vector newb_args = new Vector();
                                    for(int a=0; a<b_args.size();a++){
                                          String val1 = 
(String)(map.get(b_args.elementAt(a)));
                                          newb_args.add(val1);
                                    }
                                    Predicate newP_body = new 
Predicate(P_body.getName(),newb_args,P_body.getIsNegative(),false,null,null,n
ull,null,null);
                                    newBody.add(newP_body);
                              }
                        }
                        Rule new_R = new Rule(newHead,newBody,false);
                        GP.add(new_R);
                  }
            }
            else{
                  Predicate head = R.getHead();
                  head.setArgList(mf.getVarList(head.getArgList()));
                  GP.add(R);
            }
      }
      //mf.PrintRules_2(GP);
      return GP;
```

```
	}
	//Returns a predicate with longest argument list from a Rule
	public Predicate getMaxPredicate(Rule R){
		Predicate P = null;
		Predicate head = R.getHead();
		Vector arglist = head.getArgList();
		int len = arglist.size();
		P = head;
		Vector body = (Vector)(R.getBody());
		for(int i =0; i< body.size(); i++){
			Predicate pd = (Predicate)(body.elementAt(i));
			Vector arg = pd.getArgList();
			if(arg != null){
				if(arg.size()>len){
					P = pd;
					len = arg.size();
				}
			}
		}
		return P;
	}
	//Retuns a vector containing all the permutations of given arguments of
given size.
	//eg: give arguments 1,2 and size 2, returns |1,2|,|2,1|
	public void printCombos(int[] digits,int count, String[][] arr,String[]
temp,int number, Vector Perms){
		if(count == number){
			String[] val = new String[temp.length];
			for(int j=0; j<temp.length;j++){
				val[j] = temp[j];
				//System.out.print(temp[j]);
			}//temp = temp.substring(0,1);
			//System.out.println();
			Perms.add(val);
			return;
		}
		for(int i=0;i<(arr[digits[count]]).length;i++){
			temp[count] = arr[digits[count]][i];
			printCombos(digits,count+1,arr,temp,number,Perms);
		}
	}
	public void getCombos(int number, String[] arr){
		String[][] arr1 = new String[number][arr.length];
		for(int k = 0;k<number;k++)
		{
			arr1[k] = arr;
		}
	  int val  = 0;
		int[] digits = new int[number];
```

```
            for(int j = 0 ; j <number;j++){
                  digits[j] = j;
            }
            String[] temp = new String[number];
            printCombos(digits,0,arr1,temp,number,Perms);
      }
      //Gets all the constants from the EDB predicates in a String array
      public String[] getConstants(Vector Rules){
            Vector constants = new Vector();
            for(int i=0; i<Rules.size();i++){
                  Rule R = (Rule)(Rules.elementAt(i));
                  if(R.getIsEDB()){
                        Predicate p = R.getHead();
                        Vector args = mf.getVarList(p.getArgList());
                        for(int j=0; j<args.size();j++){
                              if(!(constants.contains(args.elementAt(j))))
                                    constants.add(args.elementAt(j));
                        }
                  }
            }
            String[] dom = new String[constants.size()];
            for(int k=0; k<constants.size();k++){
                  dom[k] = constants.elementAt(k).toString();
            }
            return dom;
      }
}
```

9.11 Stable Model Tester:

```
import java.util.*;
public class StableModelTester {
      public Vector GP = new Vector();
      public Vector OrgRules = new Vector();
      public Vector Combos = new Vector();
      MiscFunctions mfs = new MiscFunctions();
      public StableModelTester(Vector _GP, Vector _OrgRules){
            GP =  _GP;
            OrgRules = _OrgRules;
            //System.out.println("GROUND PROGRAM");
            //mfs.PrintRules_2(GP);}
      public boolean Test(Vector TP1, Vector EDB, Vector _GP)//TP =
TestPredicates, GP = Ground Program{
            GroundProgram PG = new GroundProgram();
            GP = PG.getGroundProgram(OrgRules);
            boolean flag = false;
            Vector GPNew = new Vector();
            for(int i=0;i<GP.size();i++){
                  Rule R = (Rule)(GP.elementAt(i));
                  if(R.getIsEDB()){
                        GPNew.add(R);
            }
                  else{
                        Vector body = R.getBody();
                        boolean hasNegative = false;
                        for(int j=0;j<body.size();j++){
                              Predicate P1 = (Predicate)(body.elementAt(j));
                              for(int a=0;a<TP1.size();a++){
                                    Predicate P2 =
(Predicate)(TP1.elementAt(a));
                                    if(P1.equalsPositive(P2) &&
P1.getIsNegative()){
                                          hasNegative = true;
                                          break;
                                    }
                                    if(hasNegative){
                                          break;
                                    }
                              }
                        }
                        if(!hasNegative){
                              Rule R1 = new
Rule(R.getHead(),R.getBody(),R.getIsEDB());
                              GPNew.add(R1);
                        }
                  }
            }
            Vector TransGP = (Vector)GPNew.clone();
            for(int i=0;i<TransGP.size();i++){
                  Rule R1 = (Rule)(TransGP.elementAt(i));
                  if(!(R1.getIsEDB())){
                        Vector body1 = R1.getBody();
                        Vector remove = new Vector();
                        for(int j=0; j<body1.size();j++){
                              Predicate P1 = (Predicate)(body1.elementAt(j));
                              if(P1.getIsNegative()){
                                    remove.add(j);
                              }
                        }
                        if(remove.size()>0){
                              for(int a=0;a<remove.size();a++)
                              {

      body1.removeElementAt((Integer)(remove.elementAt(a)));
                              }
                        }
                  }
            }
            EvaluateStableModel SM = new EvaluateStableModel(TransGP);
            Vector SMPredicates = SM.evaluate();
            for(int k=0;k<SMPredicates.size();k++){
                  Predicate P = (Predicate)(SMPredicates.elementAt(k));
                  P.PrintPredicate();
            }
            System.out.println("----------------Stable Model End-------------
-------");*/
            boolean comp = ComparePredicates(SMPredicates, TP1);
            GPNew.removeAllElements();
            return comp;
      }
      public Vector Test_All(Vector TP,Vector EDB, Vector _GP){
            Vector StableModels = new Vector();
            GenerateAllCombinations(TP);
            for(int i=0; i<Combos.size();i++){
                  Vector TP1 = new Vector();
                  int[] arr = (int[])Combos.elementAt(i);
                  for(int j = 0;j<arr.length;j++){
                        TP1.add(TP.elementAt(arr[j]));
                  }
                  TP1.addAll(EDB);
                  PopulateOrgData PO = new PopulateOrgData();
                  PO.CreateDatabase(OrgRules);
                  boolean flag = Test(TP1, EDB, _GP);
                  if(flag == true){
                        StableModels.add(TP1);
                        for(int m=0;m<TP1.size();m++){
                              Predicate P = (Predicate)(TP1.elementAt(m));
                              P.PrintPredicate();
                        }
                        System.out.println("STABLE MODEL ADDED---------------
-------------");
                  }
            }
            return StableModels;
      }
      public void GenerateAllCombinations (Vector TP)//PT= Predicates for
test{
            //System.out.println("Generate ALL Combinations");
            int[] in = new int[TP.size()];
            int[] out = new int[in.length+1];
            for(int i= 0;i<TP.size();i++){
                  in[i] = i;
                  //System.out.println(in[i]);
            }
            DoCombine(in,out,in.length,0,0);
      }
      public void DoCombine(int[] in, int[] out,int length, int recLevel, int
start){
            int i;
```

```
		for(i= start;i<length;i++){
			out[recLevel] = in[i];
			out[recLevel+1] = '\u0000';
			int[] c= new int[recLevel+1];
			for(int j=0; j<recLevel+1;j++){
				//System.out.print(out[j]+",");
				c[j] = out[j];
			}
			if(!Combos.contains(c)){
				Combos.add(c);
			}
			if(i<length -1)
			DoCombine(in,out,length,recLevel+1,i+1);
		}
	}
	public boolean ComparePredicates(Vector P1, Vector P2)
	{
		boolean b = false;
		if(P1.size() != P2.size())
			return false;
		else{
			for(int i=0;i<P1.size();i++){
				Predicate p1 = (Predicate)P1.elementAt(i);
				b =  false;
				for(int j=0;j<P2.size();j++){
					Predicate p2 = (Predicate)P2.elementAt(j);
					if(p1.equalsPositive(p2)){
						b = true;
					}
				}
				if(b == false)
					break;
			}
			if(b)
				return true;
			else
				return false;
		}
	}
}
```