Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), Industry and the University

WFS + Branch and Bound = Stable Models

by V.S. Subrahmanian, D. Nau and C. Vago

TR 92-83

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **1992**	2. REPORT TYPE	3. DATES COVERED **00-00-1992 to 00-00-1992**
4. TITLE AND SUBTITLE **WFS + Branch and Bound = Stable Models**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Maryland,Systems Research Center,College Park,MD,20742**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT **see report**		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**61**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

WFS + Branch and Bound = Stable Models *

V.S. Subrahmanian, Dana Nau[†]and Carlo Vago[‡]
Institute for Advanced Computer Studies
and
Department of Computer Science
University of Maryland
College Park, MD 20742, U.S.A.

Abstract

Though the semantics of non-monotonic logic programming has been studied extensively, relatively little work has been done on operational aspects of of these semantics. In this paper, we develop techniques to compute the well-founded model of a logic program. We describe a prototype implementation and show, based on experimental results, that our technique is more efficient than the standard alternating fixpoint computation. Subsequently, we develop techniques to compute the set of all stable models of a deductive database. These techniques first compute the well-founded semantics and then use an intelligent branch and bound strategy to compute the stable models. We report on our implementation, as well as on experiments that we have conducted on the efficiency of our approach.

*This research was supported by the Army Research Office under Grant Number DAAL-03-92-G-0225, and by NSF grants IRI-9109755, IRI-8907890 as well as NSF Grant NSFD CDR-88003012 to the Systems Research Center. Address correspondence to V.S. Subrahmanian. vs@cs.umd.edu

[†]Also affiliated with the Systems Research Center at the University of Maryland.

[‡]Universita Degli Studi di Milano, Dipartimento di Scienze della Informazione, Via Moretto da Brescia 9, 20133 Milano, Italy.

Contents

1 Introduction

In the past several years, the problem of representing negative information in logic programs and deductive databases[1] has been intensely studied. However, most of this work has concentrated on the declarative aspects of negation in logic programming – in particular, the focus has been on developing declarative semantics that are applicable to all, or at least a wide variety of logic programs, and which possess various epistemologically satisfying properties. An important research area that has been left relatively untouched is that of developing *operational semantics* and *implementation techniques* for logic programs that contain negation. It is only in the past year that a number of researchers have started working on this endeavor.

The primary contribution of this paper is the design and implementation of a bottom-up algorithm to compute:

- the well-founded model of a logic program [35]
- the set of stable models of a logic program [13]

The algorithm for computing the well-founded model is based on the observation that Fitting's Kripke-Kleene semantics for logic programming is "sound", but not complete w.r.t. well-founded semantics (WFS, for short). It is sound in the sense that if Fitting's Kripke-Kleene semantics assigns either true or false to a ground atom, WFS makes the same assignment. However, WFS may assign true/false to some atoms that are assigned "unknown" by Fitting's semantics. Our procedure first computes Fitting's Kripke-Kleene semantics (using an optimized version of Fitting's Φ_P operator) and simultaneously "compacts" the program by deleting parts of the program. It then applies an optimized version of the alternating fixpoint procedure[34, 11, 4] to the compacted program. Our alternating procedure compacts the (already compacted) program further at each step. It is well-known[34, 4, 11] that the alternating fixpoint procedure (without compaction) can compute the well-founded semantics. Experiments show that in practice, our procedure of first computing the Kripke-Kleene semantics and simultaneously compacting the program, and subsequently performing the alternating fixpoint computation with compaction, is much faster, than the naive alternating computation.

The algorithm for computation of stable models is of particular interest because stable models may be computed by first computing the well-founded model of the program and then using an intelligent branch and bound strategy. Intuitively, the search for stable models may be viewed as taking the atoms assigned "unknown" by the WFS, and making a true/false assignment to some of these atoms. This corresponds to the "branch"ing step. Two aspects are key to the success of branch and bound: first, the selection of atom(s) on which to branch plays a key role, and secondly, an efficient strategy to prune branches of the search tree needs to be found. We develop an algorithm based on branch and bound, for generating stable models. The algorithm has been implemented – we report on experimental results reflecting the efficiency of both the algorithm, as well as numerous optimizations present in the algorithm.

[1]Throughout this paper, we will consider only deductive databases, i.e. logic programs without function symbols.

There has been some debate in the deductive database community on whether top-down or bottom-up techniques should be used for query processing. Ullman [33], Zaniolo and the LDL group [31], Warren [38] and several others (like Bancilhon, Beeri, Ramakrishnan, etc.) have all argued that bottom-up computation leads to greater computational efficiency. The research reported here yields bottom-up compilation techniques for non-monotonic deductive databases.

The techniques we develop here are intended to be used primarily on those parts of a deductive database where fast run-time performance is expected and almost no time is available for performing deduction at run-time (for domains where deduction may be performed at run-time, techniques like those of [39, 14] may be used). An example of a concrete domain where this kind of database support is critically needed is control systems (e.g. plant monitoring systems, weapons guidance systems, avionics systems, etc.). The role of intelligent knowledge-based support for real-time control systems has been emphasized by the control systems community in [1] as a result of a joint IEEE-IFAC project on new directions for control theory. Kohn and Nerode (cf. their invited paper at the 1992 IEEE Symposium on Computer-Aided Control Systems Design [20, 21]) and independently, Caines [7] have argued that logic programming and deductive database support is critically needed for intelligent control applications.

The organization of this paper is as follows: Section 2 contains basic preliminaries, and sets out the required notation. A detailed discussion of the technique for computing well-founded semantics is contained in Section 3. Section 4 contains a detailed description of the branch and bound technique for computing stable models. Section 5 contains details about the implemented system, methods for storing stable models and the well-founded model, as well as the results of detailed experiments documenting the behavior of our system. Section 6 includes a description of future work on this project, as well as a comparison with works of other authors. Proofs are contained in the appendix.

2 Preliminaries

In this section, we quickly recapitulate the basic definitions of the stable and well-founded semantics for logic programs. We assume that readers are familiar with the basic ideas of constants, predicates, atoms, literals, Herbrand interpretations[2], clauses, and logic programs [22]. We assume that we have an underlying function-free first order language L containing only finitely many constant and predicate symbols. The Herbrand base of L is denoted by B_L. In many cases, we will abuse notation and use B_P to denote the Herbrand Base of the language generated by the constant and predicate symbols occurring in P.

Definition 1 Suppose P is a negation-free logic program. We may associate with P, an operator T_P that maps interpretations to interpretations. If I is an interpretation, $T_P(I)$ is the interpretation $\{A \in B_L \mid A \leftarrow B_1 \,\&\ldots\&\, B_n$ is a ground instance of a clause in P and $\{B_1, \ldots, B_n\} \subseteq I\}$.

[2]Throughout this paper, we will use the words "interpretation" and "model" to mean "Herbrand interpretation" and "Herbrand model", respectively. Recall that an Herbrand interpretation is simply a set of ground atoms of the language in question.

The set of all Herbrand interpretations of the language L is a complete lattice under the ordering of subset inclusion. For any negation-free logic program P, T_P is known to be a monotone and continuous function on the complete lattice of interpretations.

Definition 2 If $f : \mathbf{L} \to \mathbf{L}$ is a map from a complete lattice $\mathbf{L}$ to $\mathbf{L}$, then the *upward and downward iterations* of f are defined as follows:

$$\begin{array}{ll} f \uparrow 0 = \bot & f \downarrow 0 = \top \\ f \uparrow \alpha = f(f \uparrow (\alpha - 1)) & f \downarrow \alpha = f(f \downarrow (\alpha - 1)) \\ f \uparrow \lambda = \sqcup_{\beta<\lambda} f \uparrow \beta & f \downarrow \lambda = \sqcap_{\beta<\lambda} f \downarrow \beta \end{array}$$

where α is a successor ordinal whose immediate predecessor is $(\alpha - 1)$, and λ is a limit ordinal. $\bot$ and $\top$ are the bottom, and top elements, respectively, of the complete lattice $\mathbf{L}$. $\sqcup$ and $\sqcap$ are the "least upper bound" and "greatest lower bound" operators, respectively, on $\mathbf{L}$.

In the context of logic programming, the lattice $\mathbf{L}$ is the set of of all interpretations. $\bot$ is the empty interpretation, while $\top$ is the Herbrand Base of our underlying language. $\sqcup$ and $\sqcap$ are union, and intersection, respectively. We now define the Gelfond-Lifschitz transform which forms the basis of both the well-founded semantics and the stable model semantics for logic programs (cf. [4, 34]).

Definition 3 Suppose P is a logic program and $I \subseteq B_L$. The *Gelfond-Lifschitz* transformation of P, denoted P^I, is the logic program defined as follows:

$A \leftarrow B_1 \& \ldots \& B_n$, $n \geq 0$, is a clause in P^I iff there exists a clause

$$A \leftarrow B_1 \& \ldots \& B_n \& \neg D_1 \& \cdots \& \neg D_m$$

$(m \geq 0)$ in P such that $I \cap \{D_1, \ldots, D_m\} = \emptyset$. Nothing else is in P^I. Thus, P^I is a negation-free logic program.

Given a program P and an Herbrand interpretation I, we may define an operator, F_P, associated with P, as follows: $F_P(I) = T_{P^I} \uparrow \omega$, i.e. $F_P(I)$ is the least Herbrand model of the negation free logic program P^I.

Definition 4 (Gelfond and Lifschitz) I is a *stable* model of P iff $I = F_P(I)$.

Proposition 1 (van Gelder [34], Baral and Subrahmanian[3]) Let P be any logic program. Then F_P is anti-monotone, i.e. if $I_1 \subseteq I_2$, then $F_P(I_2) \subseteq F_P(I_1)$. Consequently, F_P^2, the function that applies F_P twice is monotonic. ■

We use the notation **wfs_true**(P) to denote the set of ground atoms true in the well-founded semantics of a logic program P. Likewise, **wfs_false**(P) denotes the set of ground atoms false in the well-founded semantics of P.

Proposition 2 (Baral and Subrahmanian[4]) Let P be any logic program. Then:

1. $A \in \textbf{wfs\_true}(P)$ iff $A \in \mathsf{lfp}(F_P^2)$ and
2. $A \in \textbf{wfs\_false}(P)$ iff $A \notin \mathsf{gfp}(F_P^2)$.

(Here, $\mathsf{lfp}(F_P^2)$ denotes the least fixpoint of F_P^2 and $\mathsf{gfp}(F_P^2)$ denotes the greatest fixpoint of F_P^2). ■

Note that as F_P^2 is a monotonic function on a complete lattice, it is always guaranteed to possess a least-fixpoint. Note that if F_P has a fixpoint I, then I is a fixpoint of F_P^2, but the converse is not necessarily true.

Example 1 Suppose P is the program consisting of the single clause

$$p \leftarrow \neg p.$$

Note that $F_P(\emptyset) = \{p\}$ and $F_P(\{p\}) = \emptyset$. Hence, F_P has no fixpoint (and hence no stable model). However, $\emptyset$ is a fixpoint of F_P^2 because

$$F_P^2(\emptyset) = F_P(F_P(\emptyset)) = F_P(\{p\}) = \emptyset.$$

In a similar vein, $\{p\}$ is also a fixpoint of F_P^2.

3 Computation of Well-Founded Semantics

Suppose P is a logic program. Our algorithms work with fully instantiated programs[3]. Our method for computing this set may be divided into three broad stages (cf. Figure 1).

- In the *Monotonic Iteration* stage (MI-stage, for short), we mimic the upward iteration of Fitting's Φ_P operator [10] and iteratively build up a set of ground atoms, denoted $\textbf{mi\_true}(P)$, which are known to be true, and a set $\textbf{mi\_false}(P)$ of ground atoms known to be false. However, there is one key difference from Fitting's operator that has a significant impact on efficiency: in addition to mimicing these iterations, the program P undergoes repeated simplification, resulting, in the limit, in a *target* program $\textbf{mi\_target}(P)$ that is usually considerably simpler than P. In practice, the monotonic iteration phase is efficient (cf. Experiment 5.3.1) when compared to the alternating fixpoint computation strategy described in [34, 4].
- In the *Gelfond-Lifschitz Oscillation* stage (GLO-stage, for short), we use the simplified program $\textbf{mi\_target}(P)$ produced by the MI-stage, and (recursively) oscillate by applying an optimized version of the Gelfond-Lifschitz transform. Each step of the

[3]We will argue later (Experiment 5.3.6), based both upon experimental results and theoretical results, that when computing stable models and well-founded semantics, grounding is not as great a problem as it may appear on first sight. However, further research is needed to ameliorate this problem. Nerode and Subrahmanian [28] have one solution to the problem which involves performing a limited amount of deduction at run-time. It appears [28] that there is a trade-off involved: by grounding at compile-time, it is possible to avoid deduction at run-time, thus improving run-time efficiency. By not grounding at compile-time (and hence saving space), one has to perform some deduction at run-time, thus reducing run-time efficiency.

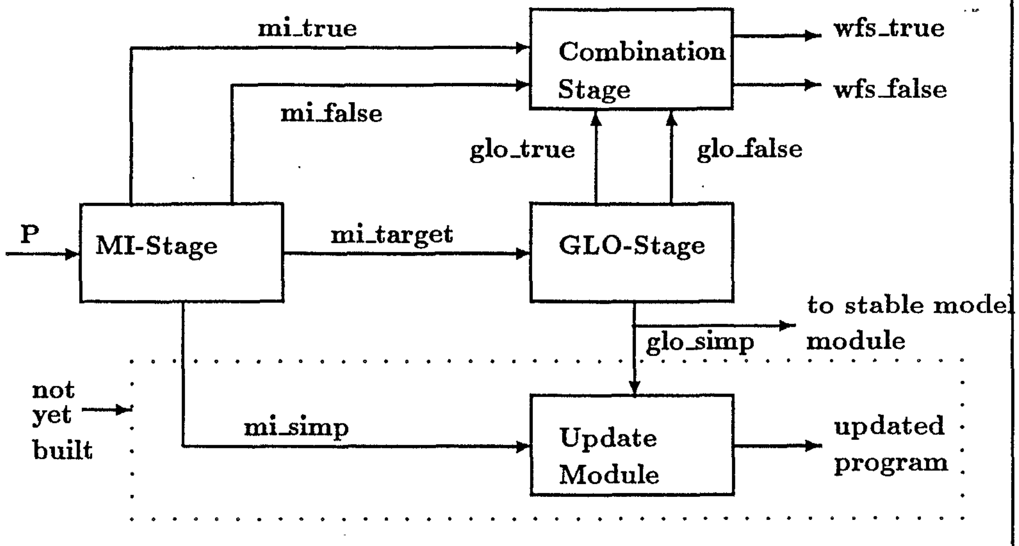


Fig. 1. Architecture of the WFS Computation Module

recursion builds up the set **glo_true**(P) of ground atoms identified to be true in the GLO-stage, and the set **glo_false**(P) of atoms identified to be false in the GLO-stage. There are two *key* differences which distinguish this method from the alternating fixpoint strategy described in [34, 4]:

- First, the GLO-stage applies only to **mi_target**(P) which is usually significantly smaller than P in size (cf. Section 5.3.2). The alternating fixpoint approach would use the program, P, which is usually much larger than **mi_target**(P).
- Second, the alternating fixpoint approach [34, 4] would proceed as follows: it would hold **mi_target**(P) fixed and start with $I_0 = \emptyset$. Given I_j, where $j \geq 1$, it would construct I_{j+1} as follows:

  (a) it would transform **mi_target**(P) w.r.t. I_j according to the Gelfond-Lifschitz transform.

  (b) it would then set I_{j+1} to the least Herbrand model of the negation-free program $G(\mathbf{mi\_target}(P), I_j)$ obtained in (a) above.

  The iteration would stop when we find a k such that $I_k = I_{k+2}$.

  Our approach adopts a different point of view. We will *not* hold **mi_target**(P) fixed. As the sequence $I_0, I_1, \ldots$ is constructed, we will keep changing the program to update previously obtained information. These changes in the program will cause the program to grow "smaller and smaller," thus leading to greater efficiency in computing the least Herbrand model (cf. Experiment 5.3.3).

  Furthermore, at any given point in time, we will *not* transform the program w.r.t. I_j, but always w.r.t. the empty-set. This can be implemented much faster because all one needs to do is to ignore all negative literals that occur in clause bodies. Both these optimizations play a significant role in reducing the time required to compute the well-founded semantics (cf. Experiment 5.3.1).

– In the *Combination* stage (C-stage, for short), we combine the results of the previous two stages (i.e. the sets **mi_true**(P), **mi_false**(P), **glo_true**(P), **glo_false**(P)) in a sound and complete manner.

Before proceeding to formally describe the details of the three-stage approach, we present a simple example to illustrate the approach, and help to fix intuitions.

Example 2 Consider the very simple program containing the following nine clauses:

$$
\begin{aligned}
p &\leftarrow q && (1)\\
p &\leftarrow r && (2)\\
q &\leftarrow \neg r \mathbin{\&} s && (3)\\
r &\leftarrow \neg q && (4)\\
s &\leftarrow t && (5)\\
t &\leftarrow && (6)\\
v &\leftarrow v && (7)\\
w &\leftarrow \neg v && (8)\\
u &\leftarrow \neg s && (9)
\end{aligned}
$$

MI-stage: The first thing to observe about this program is that t is in **wfs_true**(P) by virtue of Clause 6 and hence, so is s, by virtue of Clause 5. Thus, these two clauses may be deleted once it is realized that $s, t \in$ **wfs_true**(P). But once it is known that $s, t \in$ **wfs_true**(P), Clause 9 can be deleted as s is surely true, and similarly, s can be deleted from the body of Clause 3. In effect, then, $u \in$ **wfs_false**(P) as there is no clause left at this point with u in the head. The MI-stage mimics this kind of reasoning and leads to the computation of the following sets: **mi_true**$(P) = \{s, t\}$ and **mi_false**$(P) = \{u\}$, and the simplified target program **mi_target**(P) below:

$\boxed{\textbf{mi\_target}(P)}$

$$
\begin{aligned}
p &\leftarrow q && (10)\\
p &\leftarrow r && (11)\\
q &\leftarrow \neg r && (12)\\
r &\leftarrow \neg q && (13)\\
v &\leftarrow v && (14)\\
w &\leftarrow \neg v && (15)
\end{aligned}
$$

mi_target(P) is constructed in such a way that no atoms in either **mi_true**(P) or **mi_false**(P) occur, either positively or negatively, anywhere in **mi_target**(P).

GLO-stage: In this stage, we first realize that no atoms in **mi_true**(P) $\cup$ **mi_false**(P) occur in **mi_target**(P). We ignore P and work with **mi_target**(P), and first set $I_0 = \emptyset$ and **glo_true**(**mi_target**(P)) = **glo_false**(**mi_target**(P)) = $\emptyset$. We then compute the least model of the Gelfond-Lifschitz transformed program $(\textbf{mi\_target}(P))^{I_0}$, and denote this least model by I_1. $(\textbf{mi\_target}(P))^{I_0}$ is the program:

$\boxed{G(\textbf{mi\_target}(P), I_0)}$

$$
\begin{array}{lll}
p & \leftarrow & q \\
p & \leftarrow & r \\
q & \leftarrow & \\
r & \leftarrow & \\
v & \leftarrow & v \\
w & \leftarrow &
\end{array}
\qquad
\begin{array}{r}
(16)\\(17)\\(18)\\(19)\\(20)\\(21)
\end{array}
$$

The least model of this program is $I_1 = \{p, q, r, w\}$. The Herbrand Base of **mi_target**(P) $= \{p, q, r, v, w\}$. As $\overline{I_1} = \{v\}$, it follows that v MUST be false, and hence, we can add v to **glo_false**(**mi_target**(P)). At this point, we can use this information to simplify **mi_target**(P); as v must be false according to the WFS, Clause 14 can be deleted from **mi_target**(P) and $\neg v$ can be deleted from the body of Clause 15. Hence **mi_target**(P) now becomes the program **glo_simp**$_1(P)$ shown below:

$\boxed{\textbf{glo\_simp}_1(P)}$

$$
\begin{array}{lll}
p & \leftarrow & q \\
p & \leftarrow & r \\
q & \leftarrow & \neg r \\
r & \leftarrow & \neg q
\end{array}
\qquad
\begin{array}{r}
(22)\\(23)\\(24)\\(25)
\end{array}
$$

Recursively calling the Gelfond-Lifschitz transform w.r.t. this program yields the sets **mi_true**(**glo_simp**$_1(P)$) = **mi_false**(**glo_simp**$_1(P)$) = $\emptyset$. Thus, the GLO-stage returns, as its final output, the set **glo_false**(**mi_target**(P)) $= \{v\}$ of atoms that "false" according to WFS, and **glo_true**(**mi_target**(P)) $= \{w\}$ as the set of "true" atoms.

C-stage: At this stage, we simply combine the sets of true and false atoms returned by the MI-stage and the GLO-stage to get, as final output, the sets

$$\textbf{wfs\_true}(P) = \{s, t\} \cup \{w\} = \{s, t, w\} \text{ and}$$

$$\textbf{wfs\_false}(P) = \{u\} \cup \{v\} = \{u, v\}.$$

The atoms p, q, r are all assigned "unknown" by WFS. ■

3.1 The Monotone Iteration Module

In this section, we describe the technical details of the monotone iteration module. We will describe the precise working of the module, and prove that the output generated by this module is a sound, but not complete, method for computing the well-founded semantics. Completeness will be achieved once the output of this module is sent to, and processed by, the GLO-module and the combination module.

A three-valued interpretation I of a first order language L is any map from B_L to the set of truth values $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. In the sequel, we will assume that the reader is familiar with the notion of satisfaction in Kleene's three-valued logic [15, 10].

Definition 5 (Fitting [10]) Let P be a logic program. We associate with P, an operator, Φ_P, that maps 3-valued interpretations to 3-valued interpretations. Φ_P is specified as follows:

$$\Phi_P(I)(A) = \begin{cases} \mathbf{t} & \text{if there is a clause } C \text{ in } grd(P) \\ & \text{such that } A \text{ is the head of } C \text{ and} \\ & \text{such that } I \text{ satisfies the body of } C. \\ \mathbf{f} & \text{if, for every clause } C \text{ in } grd(P) \\ & \text{having } A \text{ as the head, it is the} \\ & \text{case that } I \text{ satisfies } \neg Body \\ & \text{where } Body \text{ is the body of } C. \\ \mathbf{u} & \text{otherwise.} \end{cases}$$

Given any logic program P, Fitting [10] shows that the Φ_P operator is monotonic, but not necessarily continuous. It is easy to verify that for function-free logic programs, Fitting's operator is continuous. Fitting suggests that a ground atom A should be considered to be *true* (resp. *false*) iff the least fixpoint, denoted $\mathsf{lfp}(\Phi_P)$, of Φ_P, assigns $\mathbf{t}$ (resp. $\mathbf{f}$) to A. If neither of these situations holds, then A is assigned the truth value "unknown". Three-valued interpretations are ordered using the ordering $\preceq$ which is defined as follows: $I_j \preceq I_k$ iff for every ground atom A, either $I_j(A) = \perp$, or $I_j(A) = I_k(A)$. The set of three-valued interpretations is a complete lower semilattice [10], and hence, as shown in [10], the operator Φ_P is guaranteed to possess a unique least fixpoint.

When performing an upward iteration of Fitting's operator, the program P is held constant. In our approach, at each step of the upward iteration, we modify the program P[4] so that the modified program is smaller, in terms of the number of occurrences of literals, than P.

Definition 6 Suppose P is a ground program, and I is a three-valued interpretation. The *modified version* of P w.r.t. I is a ground logic program, denoted $mod(P, I)$ obtained as follows:

1. if A occurs in the head of a clause $C \in P$ and $I(A) \neq \mathbf{u}$, then delete clause C from P.
2. if A occurs positively in the body of a clause $C \in P$ and $I(A) = \mathbf{f}$, then delete clause C from P.
3. if A occurs positively in the body of a clause $C \in P$ and $I(A) = \mathbf{t}$, then delete A from the body of clause C.
4. if A occurs negatively in the body of a clause $C \in P$ and $I(A) = \mathbf{t}$, then delete clause C from P.
5. if A occurs negatively in the body of a clause $C \in P$ and $I(A) = \mathbf{f}$, then delete $\neg A$ from the body of clause C.

The resulting program is denoted by $mod(P, I)$.

[4]When implementing, we modify a copy of the program P.

We use $mod(P, I)$ in the computation of $\mathsf{lfp}(\Phi_P)$ in the following way: initially, we set P_0 to P (the program under consideration). We then proceed to compute $\Phi_P(I_0)$ where I_0 assigns $\mathbf{u}$ to all ground atoms. $\Phi_P(I_0)$ will make some atoms true, some atoms false, and leave others unknown. The atoms that are made true (resp. false) will stay true (resp. false) in $\mathsf{lfp}(\Phi_P)$ because Φ_P is monotone w.r.t. the $\preceq$ ordering. Suppose A is an atom that is made true in this process. Then the truth value of A is "fixed" in the sense that $\mathsf{lfp}(\Phi_P) = \mathbf{t}$. Consequently, any clause in P with A in the head can be safely deleted as it has nothing new to contribute. Likewise, any clause with $\neg A$ occurring in the body can also be deleted because it can have nothing to contribute either (the body will stay false in all further iterations). If A occurs positively in the body of C, then we can delete A from the body. Symmetric transformations occur if A's truth value had been fixed to $\mathbf{f}$ instead of $\mathbf{t}$. The following definition formalizes this informal strategy of pruning P iteratively (the word pruning is used because either whole clauses are deleted, or individual literals are deleted).

Definition 7 (Pruning Iteration) Let P be a logic program, and let $\perp$ be the interpretation that assigns $\mathbf{u}$ to all ground atoms in the language $\mathcal{L}$. We define two sequences, called the interpretation-sequence (I-sequence, for short) and a program-sequence (P-sequence, for short) as follows:

$I_0 = \perp$
$P_0 = P$
$(j+1)$ case:
$I_{j+1}(A) = \Phi_{P_j}(I_j)(A)$ if $I_j(A) = \perp$ and $I_j(A)$ otherwise.
$P_{j+1} = mod(P_j, I_j)$.

As all programs dealt with in this paper are deductive databases, it is easy to see that there is a minimal integer n such that $I_n = I_{n+1}$ and $P_n = P_{n+1}$. Hence, given any program P, there is a unique I-sequence $I_0, \ldots, I_n$ and a unique P-sequence $P_0, \ldots, P_n$ associated with P. The following result is straightforward.

Lemma 1 Suppose P is a logic program, and $I_0, \ldots, I_n$ is the I-sequence associated with P. If $1 \leq j \leq k \leq n$, then $I_j \preceq I_k$. ■

Theorem 1 (Soundness of Pruning Iteration w.r.t. WFS) Let P be a logic program, and let $I_0, \ldots, I_n$ and $P_0, \ldots, P_n$ be the I-sequence and P-sequence associated with P. Then:

1. (Soundness w.r.t. Fitting's Semantics) $I_n = \mathsf{lfp}(\Phi_P)$.
2. for all atoms A, if $I_n(A) = \mathbf{t}$ then $A \in$ **wfs_true**(P), i.e. A is true according to the well-founded semantics for P.
3. for all atoms A, if $I_n(A) = \mathbf{f}$ then $A \in$ **wfs_false**(P), i.e. A is false according to the well-founded semantics for P. ■

The MI-stage is not complete w.r.t. the well-founded semantics, as can be easily seen by the following example:

Example 3 Consider the single clause program P below:

$$a \leftarrow a.$$

The well-founded semantics for P assigns **f** to a; however, the set **mi_false**(P) generated by the MI-module does not contain a. ■

As a final remark on the computation of WFS, we observe that if the truth value of a ground atom A is determined, during the MI-stage, to be either **t** or **f**, then the atom A is completely eliminated from the target program **mi_target**(P).

Lemma 2 (1) Suppose $\mathsf{lfp}(\Phi_P)(A) \neq \mathbf{u}$. Then A does not occur either positively or negatively in **mi_target**(P).
(2) Suppose $\mathsf{lfp}(\Phi_P)(A) = \mathbf{u}$. Then there exists a clause C in **mi_target**(P) having A as the head and such that at least one literal in the body of C is assigned the truth value **u** by $\mathsf{lfp}(\Phi_P)$. ■

Before proceeding to a detailed description of the GLO-stage, we draw the reader's attention to Figure 1 and the computation of stable models. The idea is that if we want to eventually compute the stable models of a deductive database P, we first compute the well-founded semantics of P and simultaneously generate a "small" program (denoted **glo_simp**(P) in Figure 1) which is then piped to the branch and bound procedure that computes stable models. Thus, we need to be sure that the transformation performed during the WFS computation module do not compromise the stable models in any way. The following lemmas are needed to establish this property.

Lemma 3 Suppose P is a logic program and A is a ground atom which is assigned **t** by the well-founded semantics of P. Let Q be the program obtained from P by:

1. Deleting all clauses in P with head A and
2. Deleting all clauses in P with $\neg A$ in the body and
3. Deleting positive occurrences of A from the body of any clause in which it occurs.

Then: A two-valued interpretation I is a stable model of Q iff $I \cup \{A\}$ is a stable model of P. ■

The following lemma is the analog of Lemma 3 for the negative case. Its proof is similar to that of Lemma 3.

Lemma 4 Suppose P is a logic program and A is a ground atom which is assigned **f** by the well-founded semantics of P. Let Q be the program obtained from P by:

1. Deleting all clauses in P with head A and
2. Deleting all clauses in P with A occurring positively in the body and

3. Deleting negative occurrences of A from the body of any clause in which it occurs.

Then: A two-valued interpretation I is a stable model of Q iff I is a stable model of P. ■

The above two lemmas jointly indicate that as long as the three valued interpretation I is "sound" w.r.t. the well-founded semantics (in the sense that whenever I assigns false to an atom A, then $A \in$ **wfs_true**(P) and whenever I assigns true to an atom B, then $B \in$ **wfs_true**(P)), then P's stable models may be obtained from those of *mod*(P, I) by appending the true atoms in I to the stable models of *mod*(P, I).

3.2 The Gelfond-Lifschitz Oscillation Module

As seen in Example 3, the MI-stage alone is not complete w.r.t. WFS computation. However, it is sound w.r.t. WFS computation. The Gelfond-Lifschitz Oscillation (GLO, for short) stage performs some further computations with a view to computing that part of the WFS which is not already computed in the MI-stage. The GLO-module takes as input, the program **mi_target**(P) produced by the monotone iteration module. It then performs an alternating fixpoint-like computation (cf. [34, 4]). However, there are a few significant differences which allow our strategy to be much more efficient (cf. Experiment 5.3.1) than the ordinary alternating fixpoint computation strategy. The first difference is that unlike the alternating fixpoint computation, our GLO-procedure only applies to the program **mi_target**(P) which is usually much smaller than the program P. Secondly, as we perform the oscillation, we continue *pruning* the program, so that at each stage, the oscillation steps are applied to "smaller and smaller" programs. This causes the oscillation to be much more efficient than otherwise (cf. Experiment 5.3.3).

If we look carefully at the well-founded semantics, the iterations of the F_P operator exhibit the following behavior (this behavior has been observed by Baral and Subrahmanian [3, 4] and Fitting [11] and van Gelder [34]): the interpretations at *even* levels of the oscillation form a monotonically increasing sequence, and gradually build up, in the limit, the set **wfs_true**(P):

$$F_P^0(\emptyset) \subseteq F_P^2(\emptyset) \subseteq \ldots \subseteq F_P^{2i}(\emptyset) \subseteq \ldots$$

The *odd* levels of the oscillation form a monotonically decreasing sequence and gradually build up the *complement* of the set **wfs_false**(P):

$$F_P^1(\emptyset) \supseteq F_P^3(\emptyset) \supseteq \ldots \supseteq F_P^{2i+1}(\emptyset) \supseteq \ldots$$

In other words, the sequence,

$$\overline{F_P^1(\emptyset)} \subseteq \overline{F_P^3(\emptyset)} \subseteq \ldots \subseteq \overline{F_P^{2i+1}(\emptyset)} \subseteq \ldots$$

is a monotonically increasing sequence, and in the limit, it constructs the set **wfs_false**(P). Thus, when we apply F_P first to the empty set and compute $F_P^1(\emptyset)$, we know that all atoms in $\overline{F_P^1(\emptyset)}$ are *false*. Hence, we can use this information to transform the program P. In the next stage, when we apply F_P to $F_P^1(\emptyset)$, we know that all atoms in the set $F_P^2(\emptyset)$ are *true*. We may use this information to transform the program. Thus, at odd levels, we should transform the program P according to what was learned to be false, while at even levels, we should transform the program under consideration according to what has been learned to be true. These intuitions are formalized in the following definitions.

Definition 8 (Transformation Strategy) Given a program P, and a two-valued interpretation I, we now define a transformation of P w.r.t. I[5]. This transformation depends on one *extra* parameter, called *pos* or *neg*.

$trans(P, I, neg)$ is defined as follows:

1. if $A \notin I$, and A occurs in the head of a clause $C \in P$, then delete C from P.
2. if $A \notin I$, and A occurs positively in the body of a clause $C \in P$, then delete C from P.
3. if $A \notin I$, and A occurs negatively in the body of a clause $C \in P$, then delete all occurrences of $\neg A$ from the body of C.

$trans(P, I, pos)$ is defined as follows:

1. if $A \in I$ and A occurs in the head of a clause $C \in P$, then delete C from P.
2. if $A \in I$ and A occurs negatively in the body of a clause $C \in P$, then delete C from P.
3. if $A \in I$ and A occurs positively in the body of a clause $C \in P$, then delete all occurrences of A from the body of C.

Definition 9 (Pruning Oscillation) Suppose P is a logic program. Define the *GLO-iteration* of P as four sequences:

- a sequence of two-valued interpretations $I_0, \ldots, I_n, \ldots$
- a sequence of programs $P_0, \ldots, P_n, \ldots$
- a sequence of sets of *true* atoms $\mathbf{glo\_true}_0, \ldots, \mathbf{glo\_true}_n, \ldots$ and
- a sequence of sets of *false* atoms $\mathbf{glo\_false}_0, \ldots, \mathbf{glo\_false}_n, \ldots$.

These sequences are constructed as follows:

$\boxed{j = 0}$
$I_0 = \emptyset$
$P_0 = P$
$\mathbf{glo\_true}_0 = \emptyset$
$\mathbf{glo\_false}_0 = \emptyset$

$\boxed{j = 1}$
$I_1 = F_{P_0}(I_0)$
$P_1 = trans(P_0, I_1, neg)$
$\mathbf{glo\_true}_1 = \emptyset$

[5]Unlike Section 3.1 where we modified programs using three-valued interpretations, the transformation strategy described here uses two-valued interpretations.

$\textbf{glo\_false}_1 = (B_{P_0} - I_1)$

$\boxed{\text{For even } j, j > 0}$
$I_{j+2} = F_{P_{j+1}}(I_{j+1})$
$P_{j+2} = trans(P_{j+1}, I_{j+2}, pos)$
$\textbf{glo\_true}_{j+2} = \textbf{glo\_true}_j \cup I_{j+2}$
$\textbf{glo\_false}_{j+2} = \textbf{glo\_false}_j$

$\boxed{\text{For odd } j, j > 1}$
$I_{j+2} = F_{P_{j+1}}(I_{j+1})$
$P_{j+2} = trans(P_{j+1}, I_{j+2}, neg)$
$\textbf{glo\_true}_{j+2} = \textbf{glo\_true}_j$
$\textbf{glo\_false}_{j+2} = \textbf{glo\_false}_j \cup (B_{P_{j+1}} - I_{j+2})$

Note that the above definition simultaneously defines both the sequence of interpretations and the sequence of programs. It is well-defined because, each I_j is defined in terms of P_{j-1}, I_{j-1} for $j > 0$. Likewise, each P_j is defined in terms of I_j and P_{j-1}; as I_j is defined in terms of P_{j-1}, I_{j-1}, this does not lead to any circularity. Similar comments apply to $\textbf{glo\_true}_j$ and $\textbf{glo\_false}_j$.

In order to better illustrate pruning oscillations, we return to Example 2.

Example 4 Consider the program P of example 2. We focus upon **mi_target**(P) which consists of clauses (10)–(15). Our sequence of I's and P's is built as follows:

1. $I_0 = \emptyset$.
2. $P_0 = \{10, 11, 12, 13, 14, 15\}$.
3. $\textbf{glo\_false}_0 = \textbf{glo\_true}_0 = \emptyset$.
4. $I_1 = F_{P_0}(I_0) = \{p, q, r, w\}$. Note that $v \notin I$.
5. Thus, $P_1 = trans(P_0, \{p, q, r, w\}, neg)$. There are two clauses in P_0 containing occurrences of v – clause 14 and clause 15. Clause 14 gets deleted, while $\neg v$ gets deleted from the body of clause 15. Thus, P_1 consists now of clauses

$$p \leftarrow q \tag{26}$$
$$p \leftarrow r \tag{27}$$
$$q \leftarrow \neg r \tag{28}$$
$$r \leftarrow \neg q \tag{29}$$
$$w \leftarrow \tag{30}$$

6. $\textbf{glo\_false}_1 = \overline{I_1} = \{v\}$, and $\textbf{glo\_true}_0 = \emptyset$.
7. The next stage is the construction of $I_2 = F_{P_1}(I_1)$ which is equal to $\{w\}$.

8. P_2 is now set to $trans(P_1, \{w\}, pos)$. Computing $trans(P_1, \{w\}, pos)$ leads to Clause 30 being deleted from P_1. Therefore, P_2 consists of clauses (26)–(29).

9. At this stage, $\mathbf{glo\_false}_2 = \mathbf{glo\_false}_1$, but $\mathbf{glo\_true}_1 = \{w\}$.

10. The next stage is the construction of $I_3 = F_{P_2}(I_2)$ which is equal to $\{p, q, r\}$.

11. P_3 is now set to $trans(P_2, \{p, q, r\}, neg)$. No clauses are deleted nor modified in this step, and we have $P_3 = P_2$.

12. $\mathbf{glo\_false}_3 = \mathbf{glo\_false}_2 \cup \overline{I_3} = \{v\}$. Note, in particular, that complement of I_3 is w.r.t. the Herbrand Base of P_2, and hence, $\overline{I_3} = \emptyset$.

13. The next stage is the construction of $I_4 = F_{P_3}(I_3)$ which is equal to $\emptyset$.

14. P_4 is now set to $trans(P_3, \emptyset, pos)$ and leads to no change.

15. The values of both $\mathbf{glo\_false}_4$ and $\mathbf{glo\_true}_4$ are the same as the values of $\mathbf{glo\_false}_3$ and $\mathbf{glo\_true}_3$ respectively. As there are no changes in the values of *both* $\mathbf{glo\_true}_3$ and $\mathbf{glo\_true}_4$, we may terminate construction of the sequence. ■

The alternating fixpoint approach [34, 4] allows us to stop constructing our sequence(s) as soon as we find the smallest n such that $\mathbf{glo\_true}_n = \mathbf{glo\_true}_{n+2}$. It turns out that in this case, $\mathbf{glo\_true}_n = \mathsf{lfp}(F_P^2) = \mathbf{wfs\_true}(P)$ and that $\mathbf{glo\_false}_{n+1} = \mathsf{gfp}(F_P^2) = \mathbf{wfs\_false}(P)$. The equality $\mathsf{lfp}(F_P^2) = \mathbf{wfs\_true}(P)$ has been proved in [4], as has the equality $\mathsf{gfp}(F_P^2) = \mathbf{wfs\_false}(P)$. What remain to be established are the equalities $\mathbf{glo\_true}_n = \mathsf{lfp}(F_P^2)$ and $\mathbf{glo\_true}_{n+1} = \mathsf{gfp}(F_P^2)$. We show this below.

Theorem 2 Suppose P is a logic program. Then, for all even integers i, it is the case that

1. $\mathbf{wfs\_true}(P) = \mathbf{wfs\_true}(P_i) \cup \mathbf{glo\_true}_i(P)$ and

2. $\mathbf{wfs\_false}(P) = \mathbf{wfs\_false}(P_i) \cup \mathbf{glo\_false}_i(P)$ ■

Part (1) of Theorem 2 says that to compute $\mathbf{wfs\_true}(P)$, we can perform pruning oscillations for i stages. At the end of these i stages, we have a set $\mathbf{glo\_true}_i(P)$ of ground atoms, and a "pruned" program P_i. $\mathbf{wfs\_true}(P)$ may be obtained by computing $\mathbf{wfs\_true}(P_i)$ and then adding all the atoms in $\mathbf{glo\_true}_i(P)$ to this set. Part (2) of the theorem is similar. Theorem 2 has, as an important corollary, the following result:

Corollary 1 (van Gelder [34], Baral and Subrahmanian[4]) Suppose P is a logic program. Then $\mathbf{wfs\_true}(P) = \mathbf{glo\_true}(P)$ and $\mathbf{wfs\_false}(P) = \mathbf{glo\_false}(P)$. ■

Though the above corollary says that the GLO-module alone is sufficient to compute the well-founded semantics of any program P, it turns out that using the GLO-oscillation on a program P is relatively inefficient (cf. Experiments 5.3.1 and 5.3.3). Instead, it is computationally faster, in practice, to run the MI-module first on program P, and generate the sets $\mathbf{mi\_true}(P)$ and $\mathbf{mi\_false}(P)$ and the modified program $\mathbf{mi\_target}(P)$.

mi_target(P) is usually much "smaller" than P (cf. Experiment 5.3.2); applying the GLO module on **mi_target**(P) leads to the computation of the sets **glo_true**(**mi_target**(P)) and **glo_false**(**mi_target**(P)) which may then be combined using the combination module below.

3.3 The Combination Module

The combination module takes as input, the sets **mi_true**(P) and **mi_false**(P) returned by the monotone iteration module, and the sets **glo_true**(**mi_target**(P)) and **glo_false**(**mi_target**(P)) returned by the GLO-module. It returns, as output, the set **mi_true**$(P) \cup$ **glo_true**(**mi_target**(P)) of "true" atoms, and **mi_false**$(P) \cup$ **glo_false**(**mi_target**(P)) of "false" atoms. The following result now follows immediately from Theorem 2 and Corollary 1.

Theorem 3 Let P be any logic program. Then:

1. **wfs_true**$(P) =$ **mi_true**$(P) \cup$ **glo_true**(**mi_target**(P))
2. **wfs_false**$(P) =$ **mi_false**$(P) \cup$ **glo_false**(**mi_target**(P)) ■

Given a logic program P, once the MI-module, GLO-module and the combination modules have been executed, the sets **wfs_true**(P) and the sets **wfs_false**(P) are fully computed. A simplified version, **glo_simp**(**mi_target**(P)), of P is also computed. This simplified program is now fed into the stable model computation module (described below).

4 Computation of Stable Semantics

It is well-known [34, 4] that the well-founded model approximates the stable models of a logic program in the following sense: for any logic program, P, and for any stable model, M, of P:

- **wfs_true**$(P) \subseteq M$, i.e. the set of ground atoms true in the well-founded semantics of P is a subset of the set of atoms true in M and
- **wfs_false**$(P) \subseteq (B_P - M)$, i.e. the set of ground atoms false in the well-founded semantics of P is a subset of the set of atoms false in M.

4.1 Informal Description of Branch and Bound Algorithm

Given a logic program P, we compute its stable models as follows:

1. First, we compute the well-founded semantics of P using the procedure outlined in the preceding section. The WFS computation module (cf. Figure 1) returns the following: the sets **wfs_true**(P) and **wfs_false**(P), as well as the program **glo_simp**(P), which is a simplified version of **mi_target**(P). **glo_simp**(P) is the final element P_n of the sequence $P_0, \ldots, P_n$ specified in Definition 9. (As we are only dealing with deductive databases, there must exist an integer n such that $P_n = P_{n+1}$).

```
$L = \langle(P, \emptyset, \emptyset, B_P)\rangle$; (* $B_P$ is the Herbrand Base of $P$ *)          (1)
$S = \emptyset$; (* $S$ is the set of stable models obtained so far *)                               (2)
while ($L \neq \emptyset$) do                                                                         (3)
  select the first node $Q = (q, T, F, U)$ from list $L$;                                             (4)
  Remove $Q$ from $L$;                                                                                (5)
  if there is no $T_0 \in S$ such that $T_0 \subseteq T$ then                                         (6)
    Select ground atom $A$ from $U$;                                                                  (7)
    $Q^- = (q^-, T^-, F^-, U^-)$ where                                                                (8)
        $q^-$ is $q$ modified by $\neg A$ and                                                         (9)
        $T^-$ is $T \cup$ the set of atoms true in WFS($q^-$) and                                     (10)
        $F^-$ is $F \cup \{A\} \cup$ the set of atoms false in WFS($q^-$) and                         (11)
        $U^-$ is the set $(U - \{A\}) - (T^- \cup F^-)$                                               (12)
    if $T^-$ is not a superset of any $T_0 \in S$ then                                                (13)
      if $Q^-$ is consistent then                                                                     (14)
        if $U^- = \emptyset$ then                                                                     (15)
          add $T^-$ to $S$                                                                            (16)
      else append $Q^-$ to the end of list $L$;                                                       (17)
    $Q^+ = (q^+, T^+, F^+, U^+)$ where                                                                (18)
      $q^+$ is $q$ modified by $A$ and                                                                (19)
      $T^+$ is $T \cup \{A\} \cup$ the set of atoms true in WFS($q^+$) and                            (20)
      $F^+$ is $F \cup$ the set of atoms false in WFS($q^+$) and                                      (21)
      $U^+$ is the set $(U - \{A\}) - (T^+ \cup F^+)$                                                 (22)
    if $T^+$ is not a superset of any $T_0 \in S$ then                                                (23)
      if $Q^+$ is consistent then                                                                     (24)
        if $U^+ = \emptyset$ then                                                                     (25)
          add $T^+$ to $S$                                                                            (26)
      else append $Q^+$ to the end of list $L$;                                                       (27)
end while                                                                                             (28)
return S;                                                                                             (29)
```

Fig. 2: Branch and Bound Algorithm for Computing Stable Models

2. Our branch and bound algorithm for computing stable models takes **glo_simp**(P) as input, and returns the set, S, of all stable models of **glo_simp**(P) as output.

3. The set of stable models of the original program P is then $\{\mathbf{wfs\_true}(P) \cup I \mid I \in S\}$.

An important point to note is that the program, P, whose stable models we wish to compute should not be fed directly to the branch and bound algorithm (doing so may lead to incorrect results). Only **glo_simp**(P) may be fed to the branch and bound algorithm. The example below illustrates the working of the algorithm. Formal definitions are given after the example.

Example 5 Suppose $q \equiv$ **glo_simp**(P) is the program:

$$a \leftarrow \neg b \qquad (31)$$

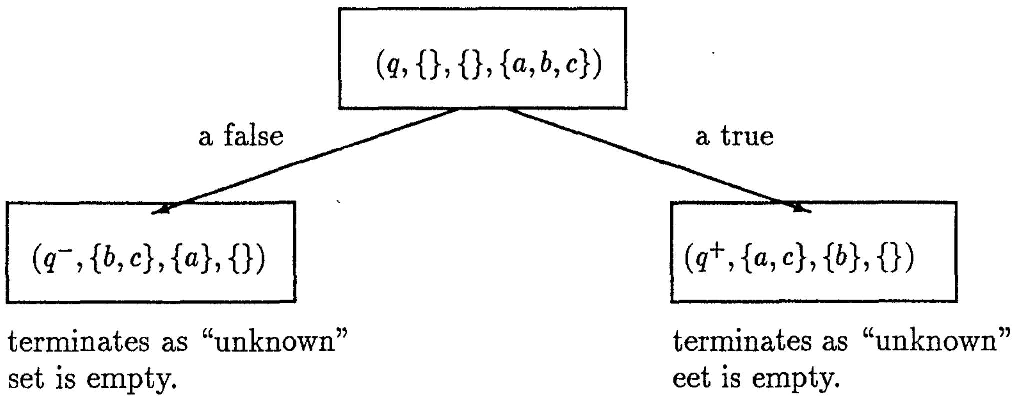

$(q, \{\}, \{\}, \{a, b, c\})$
a false
a true
$(q^-, \{b, c\}, \{a\}, \{\})$
$(q^+, \{a, c\}, \{b\}, \{\})$
terminates as "unknown" set is empty.
terminates as "unknown" eet is empty.

Fig. 3. Branch and Bound Example

$$b \leftarrow \neg a \tag{32}$$
$$c \leftarrow a \tag{33}$$
$$c \leftarrow b \tag{34}$$

All the atoms a, b, c are "unknown" according to the well-founded semantics. In our branch and bound algorithm, we process this program as follows: we first initialize the list S (of stable models found thus far) to $\emptyset$ and we have a list L containing one node – the 4-tuple

$$Q = (\textbf{glo\_simp}(P), \emptyset, \emptyset, \{a, b, c\}).$$

L points to a list of nodes that are yet to be processed. The four-tuple consists of the program to be processed, atoms assumed to be true, atoms assumed to be false, and atoms currently "unknown". We select an atom that is unknown (let us say we select a) and branch by assigning either *false* or *true* to a. How best to select an atom from the set of currently "unknown" atoms is a significant problem; one method of doing so is described in Section 4.3. Figure 3 shows the branching process once the atom a has been chosen as the atom on which to branch. The left branch assumes a to be false, the right branch assumes a to be true.

In the left branch, which assumes a to be false, we replace occurrences of a (positive and negative) in the body of clauses in **glo_simp**(P) as follows: If a occurs positively in the body of a clause, replace it by *false*, and if a occurs negatively in the body of a clause, then delete that negative occurrence of a from the body. This leads to a new node consisting of

- q^-: the modified program – in this case, it consists of the clauses:

$$a \leftarrow \neg b \tag{35}$$
$$b \leftarrow \tag{36}$$
$$c \leftarrow \mathit{false} \tag{37}$$
$$c \leftarrow b \tag{38}$$

A recursive call is made to the WFS computation algorithm. The set of atoms true in the well-founded semantics of this new program is $\{b, c\}$ and the set of atoms false in the well-founded semantics of this new program is $\{a\}$.

- T^-: The true atoms consist of the true atoms from the parent node ($\emptyset$ in this case) plus the atoms determined to be true in the well-founded semantics of the new program. Hence, the set of true atoms in the new node is $\{b, c\}$.
- F^-: The false atoms consist of the false atoms from the parent node ($\emptyset$ in this case) plus the atoms determined to be false in the well-founded semantics of the new program. Hence, the set of false atoms in the new node is $\{a\}$.
- U^-: The set of unknown atoms in the new node is is $\emptyset$ (all atoms' truth values have been "fixed" as above).

  We then check if T^- is a superset of anything in S. It is not. Furthermore, we observe that $T^- \cap F^- = \emptyset$, i.e. the assumption that a is false has not led to inconsistency.

  Finally, we observe that nothing is now unknown, i.e. U^- is empty. Hence, all atoms have been assigned truth values, and no inconsistency results. Consequently, we know that T^- is stable, and we add it to S. (Had U^- been non-empty, we would have added the tuple (q^-, T^-, F^-, U^-) to the list L.)

In the right branch, which assumes a to be true, we delete positive occurrences of a in clause bodies, and replace occurrences of $\neg a$ in clause bodies by $false$. This leads to a new node consisting of:

- q^+: The modified program consisting of the clauses

$$a \leftarrow \neg b \tag{39}$$
$$b \leftarrow false \tag{40}$$
$$c \leftarrow \tag{41}$$
$$c \leftarrow b \tag{42}$$

  When the well-founded computation module is called with this program as input, the set $\{c\}$ is determined to be true and $\{b\}$ is determined to be false.

- T^+: Consists of the assumption, a, and c, and hence is the set $\{a, c\}$.
- $F+$: Consists of $\{b\}$
- U^+: This set is empty.

We then check if T^+ is a superset of something in S. It is not. Furthermore, $T^+ \cap F^+ = \emptyset$ and hence, there is no inconsistency. Furthermore, U^+ is empty. Consequently, we add T^+ to S.

At this point, L contains no nodes, and we are done. S contains the two stable models of this program $\{a, c\}$ and $\{b, c\}$. ■

4.2 Formal Properties of Branch and Bound Algorithm

In this section, we develop the formal theory of computing stable models using the branch and bound strategy of Figure 2. As can be observed by a cursory glance at the algorithm of Figure 2, various expressions used in the description need to be formally defined. The first is the concept of what expressions like "q^- is q modified by $\neg A$" and "q^+ is q modified by A" mean. These modifications are similar, but not identical to, the transformation strategy given in Definition 8.

Definition 10 Suppose q is a logic program, and A is a ground atom. The result of *modifying q w.r.t.* $\neg A$, denoted $\mathbf{CH}(q, \neg A)$, is the logic program obtained as follows:

1. If A occurs in the body of a clause in q, then A is replaced by the atom *false*.
2. If $\neg A$ occurs in the body of a clause in q, then that occurrence of $\neg A$ is deleted.

The result of *modifying q w.r.t.* A, denoted $\mathbf{CH}(q, A)$ is the logic program obtained as follows:

1. If $\neg A$ occurs in the body of a clause in q, then $\neg A$ is replaced by the atom *false*.
2. If A occurs in the body of a clause in q, then that occurrence of A is deleted.

We assume that the proposition *false* is an artificial atom that is not considered (for ease of presentation) to occur in the Herbrand Base of the program. The key difference between the modification $mod(-,-)$ and $\mathbf{CH}(-,-)$ is that the latter never causes any clause to be deleted and never affects the head of any clause.

Definition 11 Suppose T is a binary tree. The root of T is said to be a *level 1* node. If N is a level i node, and N' is a child of N, then N' is said to be a *level* $(i+1)$ node.

If T contains finitely many nodes, then the height of T is defined to be

$$\max\{\text{level}(N) \mid N \in T\}.$$

Definition 12 Suppose P is a logic program. Let B_P be the Herbrand Base of P, and let $a_1, \ldots, a_n$ be an enumeration of B_P. The *abstract computation tree*, denoted $ACT(P)$, associated with P and the enumeration ordering $a_1, \ldots, a_n$ is a full binary tree of height $(n+1)$ defined as follows:

1. The root of $ACT(P)$ is labeled with
$$(P, \emptyset, \emptyset, B_P).$$
2. If N is a level i node in $ACT(P)$ labeled with (q, T, F, U), and $i \leq n$ then N has two children, N^- and N^+. The link from N to N^- is labeled with $\neg a_i$, and the link from N to N^+ is labeled with a_i.

3. The label of N^- is

$$(q^-, T^-, F^-, U^-)$$

where:

   (a) $q^- = \mathbf{CH}(q, a_i)$
   (b) $T^- = T \cup \mathbf{wfs\_true}(\mathbf{glo\_simp}(\mathbf{CH}(q, \neg a_i)))$
   (c) $F^- = F \cup \mathbf{wfs\_false}(\mathbf{glo\_simp}(\mathbf{CH}(q, \neg a_i)))$
   (d) $U^- = U - (\{a_i\} \cup T^- \cup F^-)$

4. The label of N^+ is

$$(q^+, T^+, F^+, U^+)$$

where:

   (a) $q^+ = \mathbf{CH}(q, a_i)$
   (b) $T^+ = T \cup \mathbf{wfs\_true}(\mathbf{glo\_simp}(\mathbf{CH}(q, a_i)))$
   (c) $F^+ = F \cup \mathbf{wfs\_false}(\mathbf{glo\_simp}(\mathbf{CH}(q, a_i)))$
   (d) $U^+ = U - (\{a_i\} \cup T^+ \cup F^+)$

Pruning Strategy. The abstract computation tree associated with a program P is, in general, very large. The reason for this is that $ACT(P)$ is of height $||B_P|| + 1$ where $||B_P||$ is the number of ground atoms in the language being considered. Thus, as $ACT(P)$ is a full binary tree, it contains $(2^{(||B_P||+1)} - 1)$ nodes: a potentially very large number. The stable model algorithm, as envisaged in Figure 2, would attempt to alleviate this problem by the following methods:

1. First, given a logic program P, we would call the branch and bound algorithm with the program $\mathbf{glo\_simp}(P)$ which is typically much smaller than P and has a much smaller Herbrand Base. In other words, we would study the abstract computation tree $ACT(\mathbf{glo\_simp}(P))$ as opposed to $ACT(P)$. This reduces the number of nodes from $(2^{(||B_P||+1)} - 1)$ to $(2^{||B_{\mathbf{glo\_simp}(P)}||+1} - 1)$. In practice the size of the program $\mathbf{glo\_simp}(P)$ as compared to the size of P is very small indeed.

2. Second, many branches in $ACT(\mathbf{glo\_simp}(P))$ can be *pruned* away. If N is a node with label $Q = (q, T, F, U)$ such that $T \cap F \neq \emptyset$ then Q is said to be *inconsistent* and the left and right subtrees are pruned away via the if-tests in lines 14 and 24 of the branch and bound algorithm.

3. Third, further pruning can be done based upon the set U. As soon as a node's label has an empty U-component, there is no need to expand that node any further, so it is pruned in lines 15 and 25 of the algorithm.

4. Fourthly, it is not difficult to see that if we consider any branch in $ACT(P)$, the T-components of the nodes in this branch are monotonically increasing as we get further away from the root, i.e. if $N_1, \ldots, N_k$ is the branch in question, and T_i is the T-component of the label of node i, then

$$T_1 \subseteq T_2 \subseteq \cdots.$$

Furthermore, Marek and Truszczynski [25] have shown that every stable model I of a logic program P is minimal in the sense that no strict subset $J \subset I$ can be a stable model of P. Consequently, if we already know when, exploring a particular branch, that I is a stable model, and if we find that T_j is a label in that branch such that $I \subseteq T_j$, then we can prune away all subtrees rooted at node N_j. This is done in lines 13 and 23 of the branch and bound algorithm.

5. Fifth, the specification of $ACT(P)$ is non-deterministic in the sense that there are many possible ways of *selecting* which atom to branch on. A *judicious* choice of the atoms on which to branch on may well lead to:

   (a) the set of "unknown" atoms being quickly disposed of and/or

   (b) pruning of a subtree below the current node.

Given a logic program P, and an enumeration $a_1, \ldots, a_n$ of the Herbrand base of P, we use $PRUNE\_ACT(P)$ to denote the tree obtained by pruning $ACT(P)$ as much as possible using conditions (1)–(4) above.

Definition 13 Suppose P is a logic program. Let $\mathbf{LEAF}(\mathbf{glo\_simp}(P))$ denote the set

$$\{T \mid \text{ there exists a leaf node in } PRUNE\_ACT(\mathbf{glo\_simp}(P)) \text{ having, as its label, } (q, T, F, \emptyset) \text{ such that } T \cap F = \emptyset\}.$$

Let $\mathbf{MIN\_LEAF}(\mathbf{glo\_simp}(P))$ be the set of all $\subseteq$-minimal elements of $\mathbf{LEAF}(\mathbf{glo\_simp}(P))$.

In other words, $\mathbf{LEAF}(\mathbf{glo\_simp}(P))$ is simply the set of all T-components of the labels of consistent leaves of $PRUNE\_ACT(\mathbf{glo\_simp}(P))$. Similarly, $\mathbf{MIN\_LEAF}(\mathbf{glo\_simp}(P))$ is the set of *minimal* elements of $\mathbf{LEAF}(\mathbf{glo\_simp}(P))$. The following example shows the tree $PRUNE\_ACT(P)$, and how stable models may be generated.

Example 6 Consider a program P containing the following clauses:

$$\begin{array}{lcl} a & \leftarrow & \neg b \\ b & \leftarrow & \neg a \\ c & \leftarrow & a \\ c & \leftarrow & b \end{array}$$

Figure 4 shows the tree $PRUNE\_ACT(P)$ corresponding to this program P. Note that in this case, $P = \mathbf{glo\_simp}(P)$.

If one looks carefully at this figure, the *strategy* to select a literal is c, b, a. In other words, branching at the root is based on c, branching at level 1 nodes is based on b. It turns out that we never need to branch on a.

Suppose we choose, instead, to consider selection of the branch literals to occur in the order b, a, c. In that case, Figure 5 shows the tree $PRUNE\_ACT(P)$. One will observe that using this selection order causes $PRUNE\_ACT(P)$ to contain fewer nodes. Hence, this ordering is preferable to the ordering c, b, a. Section 4.3 provides an outline of how to make such selections *a priori*.

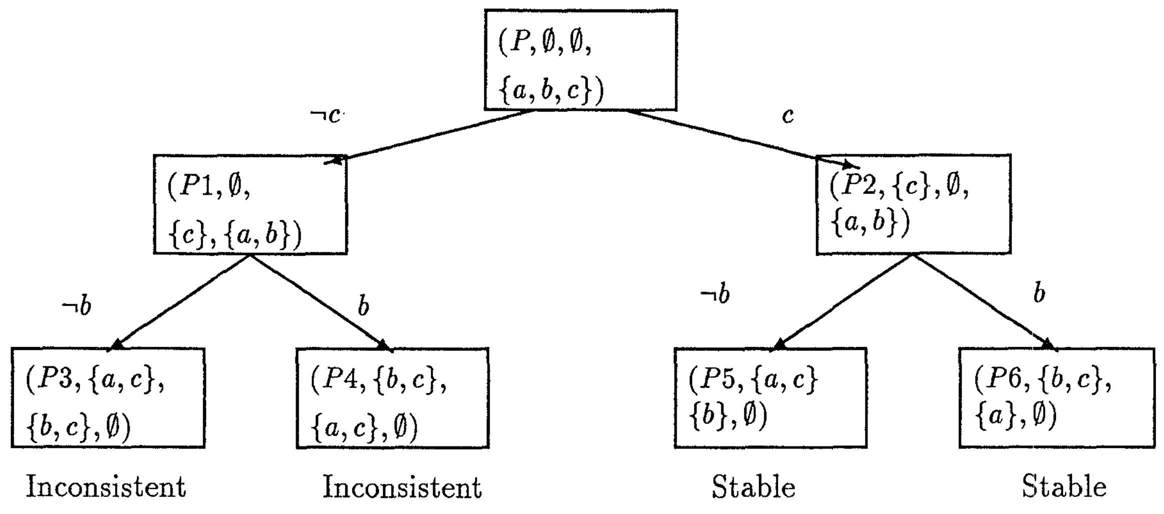

$(P, \emptyset, \emptyset, \{a,b,c\})$
$\neg c$
c
$(P1, \emptyset, \{c\}, \{a,b\})$
$(P2, \{c\}, \emptyset, \{a,b\})$
$\neg b$
b
$\neg b$
b
$(P3, \{a,c\}, \{b,c\}, \emptyset)$
$(P4, \{b,c\}, \{a,c\}, \emptyset)$
$(P5, \{a,c\} \{b\}, \emptyset)$
$(P6, \{b,c\}, \{a\}, \emptyset)$
Inconsistent
Inconsistent
Stable
Stable

Fig. 4 The (Pruned) Tree ACT(P) for Example 4 Using Selection Ordering c,b,a

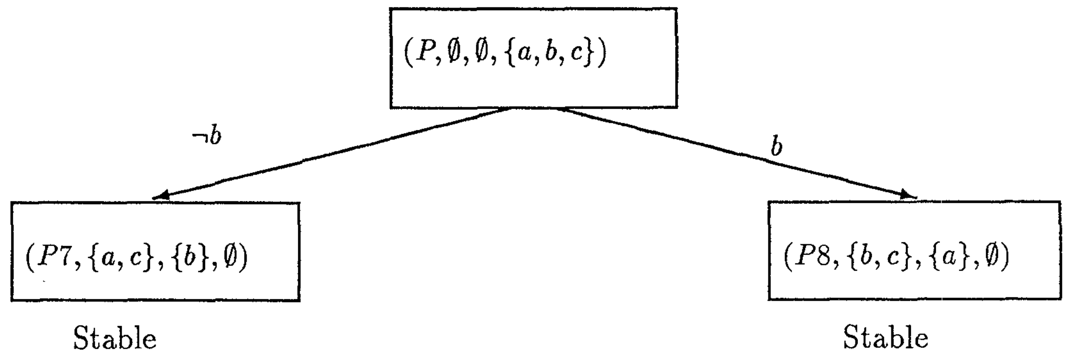

$(P, \emptyset, \emptyset, \{a,b,c\})$
$\neg b$
b
$(P7, \{a,c\}, \{b\}, \emptyset)$
$(P8, \{b,c\}, \{a\}, \emptyset)$
Stable
Stable

Fig. 5 The (Pruned) Tree ACT(P) for Example 5 Using Selection Ordering b,a,c

Note that once a specific literal ordering is given, the abstract (un-pruned) computation tree $ACT(P)$ is uniquely determined. Strictly speaking, the depth of $ACT(P)$ remains the same irrespective of the specified literal ordering because technically, $ACT(P)$ contains branching nodes for all atoms. The effect of pruning is to cut down $ACT(P)$ by refusing to branch on nodes that are either:

1. completely determined, i.e. the node's label is of the form $(q, T, F, \emptyset)$ or
2. subsumed, i.e. $T \supseteq I$ for some I that is already known to be stable, or
3. inconsistent, i.e. $T \cap F = \emptyset$.

The following result is straightforward and is of great utility in proving the soundness and completeness of the branch and bound algorithm.

Lemma 5 Suppose P is a logic program and A is a ground atom. Then:

1. If A is "unknown" according to WFS, then there exists a clause $C \in$ **glo_simp**(P) with A in the head such that
   (a) some literal L in the body of C is "unknown" according to WFS and
   (b) there is no clause $C' \in$ **glo_simp**(P) such that all literals in the body of C' are true in WFS.
2. If A occurs (positively or negatively) either in the head or in the body of any clause in **glo_simp**(P), then A is assigned **f** by WFS.

Note that the branch and bound algorithm should *not* be applied directly to a deductive database P. It works only after P has been converted to **glo_simp**(P) – if applied directly to P, incorrect results may be obtained. The reason why the branch and bound algorithm should not be directly applied to P is that all atoms occurring in **glo_simp**(P) are "unknown" according to the well-founded semantics of **glo_simp**(P). It is precisely to preserve this property that the programs occurring in labels of nodes are of the form **glo_simp**$(\mathbf{CH}(q, \pm a))$ rather than just $\mathbf{CH}(q, \pm a)$.

Theorem 4 I is a stable model of **glo_simp**(P) iff $I \in$ **MIN_LEAF**(**glo_simp**(P)). ■

Before proceeding to prove the soundness and completeness of our branch and bound algorithm in Theorem 5 below, a number of technical lemmas need to be established.

Lemma 6 The branch and bound algorithm generates the nodes of $PRUNE\_ACT(P)$ in pre-order (cf. Knuth [16]). ■

Corollary 2 (Termination of Branch and Bound Algorithm) The branch and bound algorithm always terminates. ■

Corollary 3 The branch and bound algorithm generates the nodes in **LEAF**(**glo_simp**(P)) in left to right order. ■

Lemma 7 If N and N' are nodes of $PRUNE\_ACT(P)$ with labels (q, T, F, U) and (q', T', F', U'), respectively, and if N is to the left of N', then $T' \not\subseteq T$. ■

Theorem 5 (Soundness and Completeness of Branch and Bound Algorithm) When called with **glo_simp**(P) as input, the Branch and Bound Algorithm returns as output, the set **MIN_LEAF(glo_simp**(P)**)** which is identical to the set of stable models of **glo_simp**(P). ■

4.3 Intelligent Branching

As described earlier (cf. Example 6), the selection of atoms on which to branch makes a significant difference in the height of $PRUNE\_ACT(P)$. We describe below, a simple methodology for selecting atoms on which to branch which, in practice, causes $PRUNE\_ACT(P)$ to be relatively "small." We will heavily use the "dependency graph" of Apt, Blair and Walker[2] for this purpose.

Definition 14 The graph associated with a logic program P is defined as follows:

- The nodes of the graph are the ground atoms in our underlying language and
- There is a (directed) edge from A to B if there is a clause in $grd(P)$ with A in the head such that B occurs either positively or negatively in the body.

Definition 15 Suppose P is a logic program. A ground atom A is said to *depend on* ground atom B iff there is a path of length 0 or more from A to B in the dependency graph of P.

Apt, Blair and Walker[2] use the above dependency graph (together with a labeling of the edges) to develop a notion of stratification. We will use this graph in a different way. It is well known [2] that "depends on" is a reflexive and transitive relation. Using the "depends on" relationship, we will build a quotient algebra in the usual way.

- given a ground atom A, the *equivalence class of* A, denoted $||A||$ is the set $\{B \mid B$ is a ground atom such that A depends on B and B depends on $A\}$.
- We define an ordering, denoted $\trianglelefteq$, on equivalence classes as follows: $||A|| \trianglelefteq ||B||$ iff there exists an atom $a \in ||A||$ and an atom $b \in ||B||$ such that b depends upon a.

It is not difficult to see that the relation $\trianglelefteq$ on equivalence classes is a partial ordering.

Example 7 Consider the program of Example 6. Here, the equivalence classes are:

$$\begin{aligned} ||a|| &= \{a, b\} \\ ||c|| &= \{c\} \end{aligned}$$

In particular, $||b|| = ||a||$. It is easy to see that $\{a, b\} \trianglelefteq \{c\}$. The reason is that c depends on a.

In fact, it is not difficult to see that if $||A||$ and $||B||$ are equivalence classes such that $||A|| \trianglelefteq ||B||$, then *every* atom in B must depend on every atom in A.

Given a logic program P, we may use the ordering $\trianglelefteq$ on the equivalence classes defined above to list the equivalence classes in "layers." This can be done as follows: define $\mathbf{E}_0$ to be the set of all $\trianglelefteq$-minimal equivalence classes of P. For $i \geq 0$, define $\mathbf{E}_{i+1}$ to be the set of all $\trianglelefteq$-minimal members of the set

$$\{||A|| \mid A \in B_L\} - \bigcup_{j \leq i} \mathbf{E}_j.$$

Example 8 Continuing with the program of example 6 and example 7, we note that:

$$\begin{aligned} \mathbf{E}_0 &= \{\{a, b\}\} \\ \mathbf{E}_1 &= \{\{c\}\} \end{aligned}$$

Intelligent Branching Strategy. The strategy for selecting atoms on which to branch may now be described as follows: Suppose N is the node we are currently attempting to branch from, and the label of N is (q, T, F, U). An atom $a \in U$ is *selected for branching* iff $||a|| \in \mathbf{E}_i$ implies that there is no ground atom $b \in U$ such that $||b|| \in \mathbf{E}_j$ where $j < i$.

In other words, the candidates for branching are picked from the "lowest" possible levels of the $\mathbf{E}_0, \mathbf{E}_1, \ldots$ hierarchy. Thus, in the case of the root of the tree associated with the program example 6 and example 7, we would choose to branch on either a or b instead of choosing to branch on c. This leads to a "shorter" tree.

Experiment 5.3.5 reports on some experiments that we have run to determine the value of intelligent branching.

5 Implementation and Experimentation

All the components of Figure 1 (except for the update module) as well as the entire branch and bound procedure and the procedure for selecting atoms have been implemented in a prototype compiler.

The prototype compiler is written in C running under the Unix environment on a Dec-2100 workstation. It has roughly 6200 lines of C code implementing the pruning iteration strategy described in Sections 3.1, the transformation strategy, the pruning oscillation described in Section 3.2, the branch and bound procedure of Section 4, and the intelligent branching strategy of Section 4.3.

In the rest of this section, we present some small examples showing the working of our implementation, as well as results of detailed experimentation on larger examples.

5.1 Sample Programs

5.1.1 The (Locally Stratified) Missile Example

The anti-tank missile knowledge base is shown in Figure 6. It contains rules on what kind of missile should be fired at different types of tanks. Five types of tanks are within the scope of this example: The M-1 and M-60 tanks (which are friendly) and the T-72,T-78 and T-80 (which are enemy tanks). Each tank is in one of three classes $c1, c2, c3$. As far as tanks in class $c1$ are concerned, the best anti-tank missile to use is the Tow-1, followed by the Tow-2, with the Tow-3 being the least effective. For tanks in class $c2$, the Tow-1 is the best (and the others are not really best, but are better than nothing). For class $c3$, the Tow-3 is most effective followed by the Tow-2. Finally, the rules on what to fire say the following: "If the enemy tank is approaching, but is not attacking, then choose the best anti-tank missile to use and fire it. On the other hand, if the enemy tank is attacking, then choose an effective anti-tank missile (even if it is not the best) and fire it." These rules on what to fire make sense because if the enemy tank is firing at the autonomous agent that is using our knowledge base, then there really may be no time to choose the best; it may be better to fire something effective. The CPU times taken by our implementation are given below[6]:

Well-Founded Model:	237 milliseconds
Set of Stable Models:	243 milliseconds

In contrast, the times taken by Prolog are:

Compile-Time for Prolog:	761 milliseconds
Consult-Time for Prolog:	281 milliseconds

Note that in contrast, the *entire-time* to compute the well-founded model is 237 milliseconds. This time is considerably smaller than the corresponding consult/compile times for Prolog.

At run-time queries in our framework are processed using a standard relational query module, and hence, to process the query "Find the set of all (X, Y) such that $fire(X, Y)$ is true in the well-founded model" is encoded as a standard relational query. This can be done very fast in practice.

5.1.2 The Plant Control Example

Figure 7 contains a logic program that describes a small knowledge base that may be used in plant control situations. It describes the status of a plant (or a part of a plant) based on certain temperature (warm,hot,melting) and pressure (high,low) readings of three components $c1, c2, c3$. The plant status may, at any given point of time, be either normal,

[6]All times reported here include the total time taken to read the logic program under consideration, to compute the well-founded semantics (or set of stable models, as appropriate), as well as the time taken to write, as output, the well-founded model (or set of stable models, as appropriate). In the case of Prolog, timings were obtained using the *statistics* predicate.

```
fire(M,X) :- ~attacking(X), approaching(X),missile(M), best(M,X), ~friend(X).
fire(M,X) :- attacking(X), effective(M,X), available(M).
best(tow1,X) :- c1(X), effective(tow1,X), available(tow1).
best(tow2,X) :- c1(X), effective(tow2,X), available(tow2),~best(tow1,X).
best(tow3,X) :- c1(X), effective(tow3,X), available(tow3),~best(tow1,X),
                ~best(tow2,X).
best(tow1,X) :- c2(X), effective(tow1,X), available(tow1).
best(tow3,X) :- c3(X), effective(tow3,X), available(tow3).
best(tow2,X) :- c3(X), effective(tow2,X), available(tow2),~best(tow3,X).
missile(tow1).
missile(tow2).
missile(tow3).
available(tow1).
available(tow3).
friend(m1).
friend(m60).
attacking(t72).
approaching(t72).
approaching(t80).
effective(tow1,t72).
effective(tow2,t72).
effective(tow1,t80).
effective(tow3,t80).
effective(tow3,t78).
c1(t72).
c2(t80).
c3(t78).
```

Fig. 6: Missile Example

```
status(X,danger) :- temp(X,melting).
status(X,danger) :- pressure(X,low).
status(X,warning) :- temp(X,hot), pressure(X,high).
status(X,normal) :- ~status(X,danger), ~status(X,warning), component(X).
shutdown(plant) :- status(X,danger).
sound_alarm(X) :- status(X,warning), ~shutdown(plant).
pressure(c1,high).
pressure(c2,high).
pressure(c3,low).
temp(c1,warm).
temp(c2,hot).
temp(c3,melting).
component(c1).
component(c2).
component(c3).
```

Figure 7: Plant Control Example

warning, or danger. Each component may itself sound an alarm if it is malfunctioning. Again, this example constitutes a locally stratified logic program [29]. The times taken by our implementation are given below:

Well-Founded Model:	68 milliseconds
Set of Stable Models:	73 milliseconds

In contrast, the times taken by Prolog are:

Compile-Time for Prolog	465 milliseconds
Consult-Time for Prolog	210 milliseconds

As in the case of the missile example, the times reported here for Prolog are significantly larger than the corresponding times for our well-founded and stable model computations.

5.1.3 The (Non-Locally Stratified) Animal Example

The missile example and the plant control example are both locally stratified logic programs. Consequently, they have a unique stable model which coincides with the well-founded model of the program. Figure 8 below shows a non-locally stratified logic program that describes some animals and their eating properties. This program has eight stable models. The times taken by our implementation are given below:

Well-Founded Model:	340 milliseconds
Set of Stable Models:	384 milliseconds

In contrast, the time taken by Prolog is given below:

Compile-Time for Prolog	1054 milliseconds
Consult-Time for Prolog	445 milliseconds

As in the case of the missile and plant control examples, the times reported here for Prolog are significantly larger than the corresponding times for our well-founded and stable model computations.

5.2 Storage and Access of Models

One reason why deductive databases are elegant is because they can be developed much more quickly: when creating a relational database, the database creator(s) must insert all tuples in each relation, one by one, into the database. This method of creating a relational database is consequently error-prone. Deductive databases, on the other hand, can be created much more quickly than relational database because instead of inserting all tuples, one by one, into a relational, the presence of a tuple in a relation may be implied by a rule in the database. A second advantage is that deductive databases use up less storage space than relational databases. Both these advantages (rapid database creation, lower storage requirements) are offset by the fact that at run-time, query processing takes much longer than in the relational model.

When (parts of) a database is used to provide support, in real-time, to a real-time control system, the run-time, resolution-based theorem proving approach used by deductive databases is infeasible in practice. Hence, our proposal is that those parts of a database that are expected to provide such support be compiled into a relational database format. After a deductive database is compiled, the model(s) of interest (well-founded/stable) are stored in relational format so that queries against the deductive database can be answered by checking with the stored model(s). (In the next two subsections, we show how to store and access the well-founded model, as well as the set of stable models.)

In other words, we are proposing a trade-off: by compiling those parts of a deductive database that need to provide intelligent real-time support, we retain the advantage of rapid database creation (as the creator of the database still proceeds in the same way as for deductive DBs), but lose the advantage of lower storage requirements. In return, we gain the advantage of rapid query-processing at run-time. These trade-offs may be summed up in the following table.

Criterion	Relational	Deductive	Our
Database Creation Time	Slow Error-prone	Fast Fewer Errors	Fast Fewer Errors
Storage Requirements	High	Much Smaller	High
Run-Time Efficiency	High	Poor	High

Some important advantages of storing the well-founded model and the set of stable models in a relational format are:

```
swim(X) :- is_fish(X).
swim(X) :- is_mammal(X), lives_in_sea(X).
swim(X) :- is_bird(X), has_webbed_feet(X).
lives_in_sea(X) :- is_fish(X), ~ab_fish(X).
lives_in_sea(X) :- is_whale(X).
flys(X) :- is_bird(X), ~ab_bird(X).
flys(X) :- is_mammal(X), has_wings(X).
has_wings(X) :- has_feathers(X).
has_webbed_feet(X) :- is_duck(X).
has_feathers(X) :- is_bird(X).
has_eggs(X) :- is_fish(X).
has_eggs(X) :- is_bird(X).
lives_on_land(X) :- is_mammal(X), ~ab_mammal(X).
lives_on_land(X) :- is_bird(X).
carnivore(X) :- has_fangs(X), is_mammal(X).
carnivore(X) :- is_fish(X), large_mouth(X).
herbivore(X) :- is_fish(X), small_mouth(X).
herbivore(X) :- has_molars(X), is_mammal(X).
has_fangs(X) :- is_cat(X).
has_molars(X) :- is_cow(X).
has_fins(X) :- lives_in_sea(X), ~ab_sea_creature(X).
large_mouth(X) :- is_whale(X).
small_mouth(X) :- is_cat(X).
large_mouth(X) :- ~small_mouth(X).
small_mouth(X) :- ~large_mouth(X).
eats(X,Y) :- lives_together(X,Y), carnivore(X), herbivore(Y).
eats(X,Y) :- is_cat(X), is_bird(Y).
lives_together(X,Y) :- flys(X), flys(Y).
lives_together(X,Y) :- lives_on_land(X), lives_on_land(Y).
lives_together(X,Y) :- lives_in_sea(X), lives_in_sea(Y).
is_whale(moby_dick).
is_cat(garfield).
is_bird(tweety).
is_duck(donald).
is_platypus(pogo).
ab_bird(tweety).
ab_mammal(X) :- is_platypus(X).
is_mammal(X) :- is_platypus(X).
ab_sea_creature(X) :- is_whale(X).
```

Fig. 8: Animal Example

- At run-time, queries are processed by executing standard relational operations like PROJECTs, JOINs, and SELECTs. This is usually much faster than doing resolution theorem-proving at run-time.
- At run-time, the user may interact directly with the database using a standard relational query language like SQL. Such languages are typically more expressive than PROLOG's run-time query language, and allow the easy expression, and processing, of aggregate queries like "What is the average salary of secretaries in this company ?" which are hard to process in PROLOG.
- Techniques for storing large amounts of relational data on auxiliary storage are well-developed. The US Census Bureau's database is on the order of 15 Gigabytes.

5.2.1 Well-Founded Models

For each n-ary predicate symbol p, there is a corresponding relation:

$$p(truthval, field_1, \ldots, field_n).$$

In practice, for most logic programs, the set of ground atoms that are assigned the truth value "unknown" is relatively small. Hence, we may store either

1. $\mathbf{wfs\_true}(P)$ and $(B_P - (\mathbf{wfs\_true}(P) \cup \mathbf{wfs\_false}(P))$ or
2. $\mathbf{wfs\_false}(P)$ and $(B_P - (\mathbf{wfs\_true}(P) \cup \mathbf{wfs\_false}(P))$.

The *truthval* field contains either **t** or **f** or **u**. If representation (1) above is used, then we store all the "true" ground atoms and all the ground atoms that receive the truth value "unknown". We will illustrate below, how representation (1) is used to store the well-founded model of the missile example.

The missile example contains the relations *fire, best, missile, available, friend,attacking, approaching, effective, c1,c2,c3.* For each of these relations, we create a table. Consider the relation *best. best* is a binary predicate; hence, we store it as a ternary relation – the extra argument is for the *truthval* field. In representation (1), we store all the "true" ground atoms and all the ground atoms that receive the truth value "unknown". The following table is the storage table for the relation *best*:

truthval	Missile	Tank
t	tow1	t72
t	tow3	t78
t	tow1	t80

As this is a locally stratified program, the well-founded model is "total" in the sense that nothing is "unknown" in the well-founded model. On the other hand, the animal example does contain atoms that are "unknown" according to the well-founded semantics. For example, the atoms

$$large\_mouth(platypus), small\_mouth(platypus)$$

are unknown. The table below shows how the (unary) relation *large_mouth* is stored as a 2-column relational table.

truthval	Animal
t	*moby_dick*
u	*tweety*
u	*donald*
u	*pogo*

Note that as *garfield* does not occur in the second column of the above table, we know that *large_mouth(garfield)* is false.

5.2.2 Set of Stable Models

As a deductive database may have multiple stable models, we now present a technique for storing multiple stable models[7]. One advantage of our storage method is that at run-time, the user can specify whether he is interested in the truth of a query in all stable models or in some. We will show below how to use SQL to find all answers to queries such as "Is $p(X)$ & $q(X)$ true in some stable model?" and "Is $p(X)$ & $q(X)$ true in all stable models?"

A straightforward way to store an atom $p(a_1, \ldots, a_n)$ is to use a relation p whose schema is:

$$p(modelnumber,\ field_1,\ \ldots,\ field_n).$$

The problem with this simple way is the classical one of normalization [32]. For instance, if the same atom appears in more than one stable model, many problems such as unnecessary duplication arise. Thus, a more appropriate way of storage is as follows:

$$p(tupleid,\ field_1,\ \ldots,\ field_n)$$
$$model(number,\ tupleid).$$

The relation *model* is used to specify the stable models and which tuples are in them. For example, the set of all rows of the form (2, *tupleid*) in the *model* relation specifies the tuple-ids of all tuples in stable model number 2. As tuple-ids are unique, they refer to a specific ground atom. Thus, tuple-id 27 refers to a specific row in a specific relational table.

The following example shows an SQL query that retrieves all answers that are true in some stable model of the deductive database being considered.

Example 9 Suppose p and q are two unary predicates in our deductive database. The following SQL query finds all answers X such that $p(X)\&q(X)$ is true in some stable model.

[7]For applications that only consider the intersection of all stable models, it suffices to store the intersection in the following way. To store an n-ary atom $p(a_1, \ldots, a_n)$, we can use a relation p whose schema is simply: $p(field_1,\ \ldots,\ field_n)$. Then querying the deductive database is exactly the same as querying a conventional relational database.

```
select p.field1
from p, q
where p.field1 = q.field1
and exists
      ( select number
        from model M1, model M2
        where M1.number = M2.number
        and M1.tupleid = p.tupleid
        and M2.tupleid = q.tupleid )
```

■

5.3 Experimental Results

We have conducted a number of experiments testing the efficiency of our prototype compiler. First of all, we have experimented with the programs considered in the literature (e.g. [34]). These include definite, stratified, locally-stratified, as well as non-locally stratified programs. Our prototype compiler handles all those programs correctly, and given the relatively small sizes of those programs, our compiler finishes all computations very rapidly. Unless otherwise stated, the computation times of our prototype compiler presented below include all computations [8] including the total time taken to: read a (ground) program, perform the MI-stage and GLO-stage computations and output the results. In cases where stable models are considered, the time to execute the branch and bound procedure is also included.

Though we have experimented with a number of alternative examples, we will only report here on experiments conducted with the "win-move" example of van Gelder [34]. These results are representative of our other results. The "win-move" example consists of the single rule

$$win(X) \leftarrow move(X,Y) \,\&\, \neg win(Y)$$

together with a set of facts of the form $move(-,-)$. This set of facts represents a directed graph (which we call the "game graph") representing the moves in a game. We ran an extensive set of experiments with the win-move example. In our experimentation, we varied the number of nodes in the game graph from 50 to 100 in steps of 10. Once the number of nodes was fixed, we randomly generated edges between these nodes. We generated 60 to 200 edges, in steps of 20. Once both the number of nodes and the number of arcs was fixed, we generated 75 *sets of edges.* In other words, once the number of nodes and number of arcs was fixed, 75 different extensional databases containing *move* predicates were generated. Each of these was run 8 times to average out variations in timing. In total, we ran

$$6 \times 8 \times 75 \times 8 = 28,800$$

logic programs altogether to get these readings.

[8]The Unix utility program *profile* is used to record computation times.

5.3.1 Our Approach vs. Alternating Approach to WFS Computation

The main aim of this experiment was to determine how our approach compared with the alternating approach as described by van Gelder [34]. We wished to compare the rate at which performance in both approaches degraded as the programs got larger in size (in terms of having more constants and more clauses in them). Our approach consists of running the (ground version of) a program P through the MI, GLO, and C-modules described in Figure 1. The naive alternating approach would run the entire program through the GLO module alone.

Figure 9 shows how our approach performed vis-a-vis the alternating fixpoint approach. The x-axis specifies the number of nodes. The dotted lines denote the times taken by our approach when the number of arcs in the graph differ. Thus, for example, the dotted line marked $n = 100$ denotes the time taken by our approach when the number of nodes varies from 50 to 100. The bold lines denote the times taken by the alternating approach. The y-axis denotes time in milliseconds.

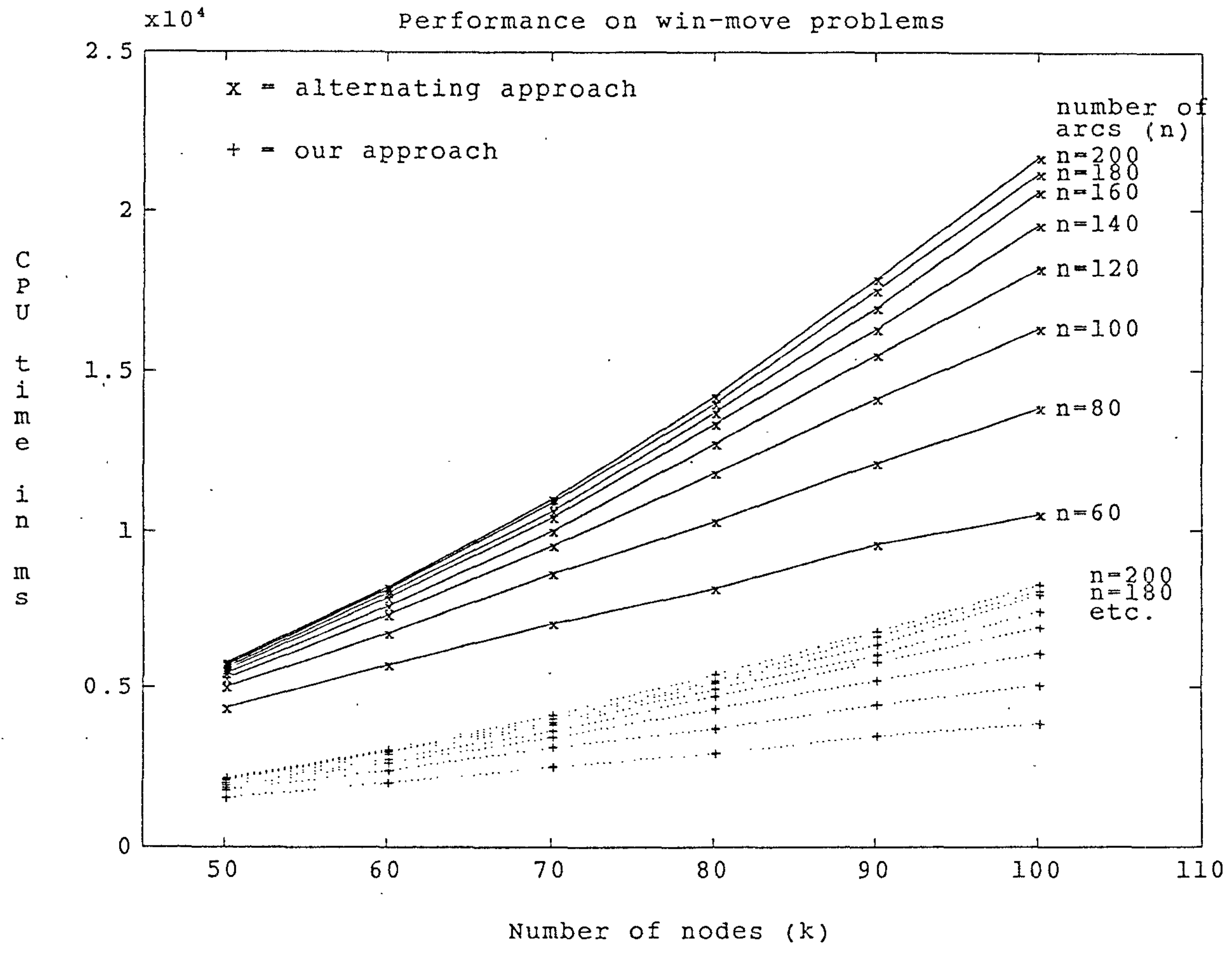
Two conclusions may be drawn from the graph of Figure 9.

- The first is that our approach takes considerably less time than the alternating approach. For each value of n, the dotted line representing our approach is completely below the bold line (for the alternating approach) that is marked with the same value of n.
- The second conclusion that may be drawn is that our approach degrades at a lower degree than does the alternating approach. Why ? Consider the slopes of the lines involved (take, for example, the dotted line $n = 100$ and the bold line $n = 100$). The slope of the dotted line is smaller than the corresponding slope for the bold line.

The second conclusion is further reinforced by the graph of Figure 9 which compares the time taken by our procedure with the time taken by the alternating procedure.

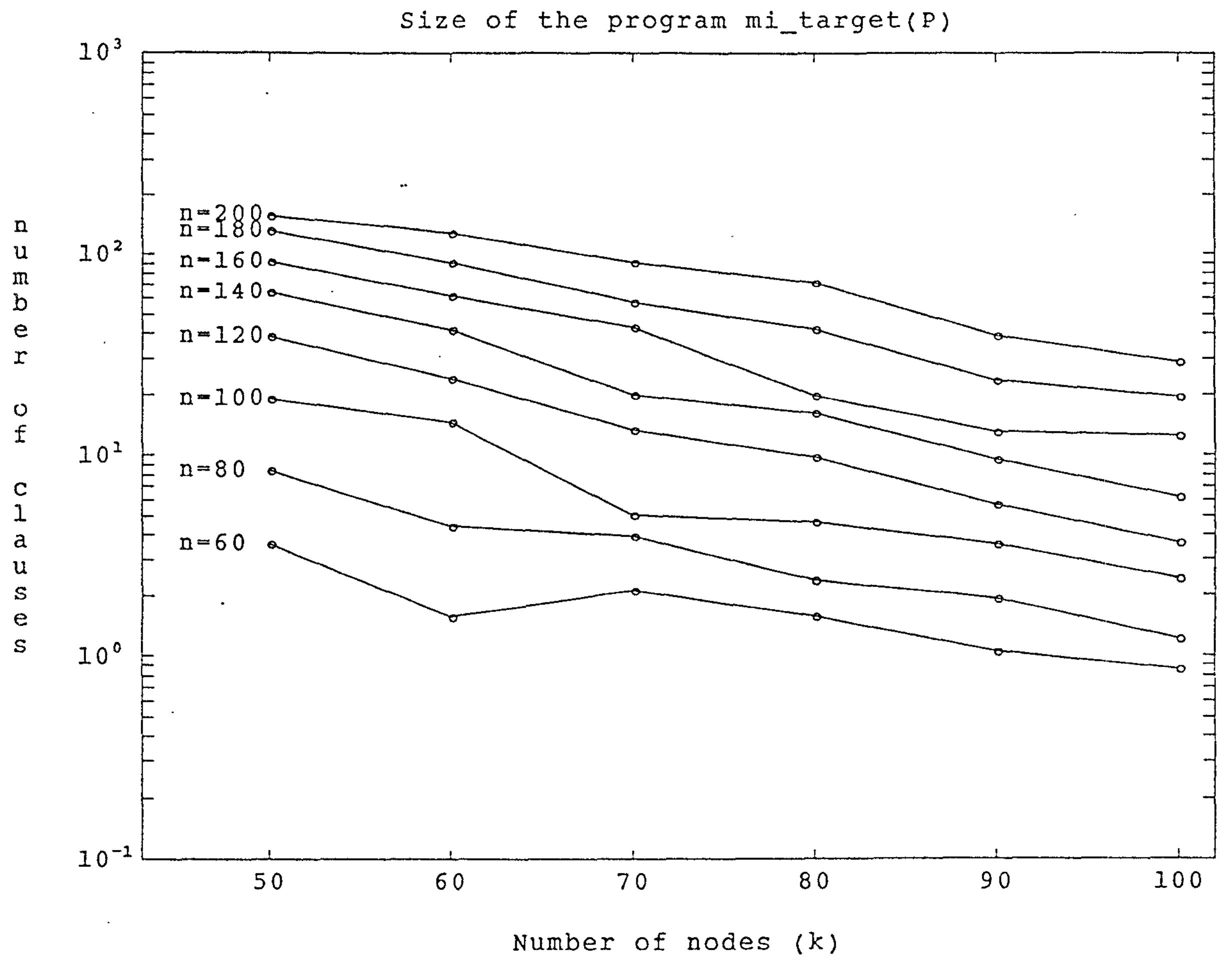
5.3.2 Size of mi_target(P) compared to the Size of P

Figure 10 below shows the number of clauses in **mi_target**(P) as the number of nodes (represented by constants in P) in the game graph is increased. The graph is plotted on a logarithmic scale which means that a linear downward slope on the log-scale means an exponential downward slope on an ordinary scale. As Figure 10 shows, for each of the values of n (the number of arcs) in the game-graph, there is a clear downward slope on the log-scale graph, showing that in practice, the effect of pruning iterations causes the size of **mi_target**(P) to decrease exponentially as a function of the number of constants. This means that pruning iterations have a more and more significant impact on the size of **mi_target**(P) as the number of constants gets larger.



x10⁴
Performance on win-move problems
x = alternating approach
+ = our approach
number of arcs (n)
n=200
n=180
n=160
n=140
n=120
n=100
n=80
n=60
n=200
n=180
etc.
CPU time in ms
2.5
2
1.5
1
0.5
0
50
60
70
80
90
100
110
Number of nodes (k)

Fig.9



Size of the program mi_target(P)
10^3
10^2
10^1
10^0
10^{-1}
number of clauses
n=200
n=180
n=160
n=140
n=120
n=100
n=80
n=60
50
60
70
80
90
100
Number of nodes (k)

Fig.10

5.3.3 Effect of Pruning Oscillation

Finally, we ran experiments to verify the effectiveness of pruning oscillations. Figure 11 shows that alternating fixpoint computation with *pruning* oscillations is an improvement on the naive alternating fixpoint computation. In the figure, the dashed lines denote the time-lines for the computation using pruning oscillations, while the bold lines denote the times taken for the naive alternating fixpoint computations. However, simply performing alternating fixpoint computation with pruning oscillations does not produce the best results.

Figure 11 shows also that our approach of first processing P through the MI-module simplifies the program, producing **mi_target**(P) and the sets **mi_true**(P) and and **mi_false**(P). Subsequently executing the GLO-program on **mi_target**(P) leads to better results than executing the GLO-program on the larger program P.

5.3.4 Stable Model Computation

Figure 12 shows the total time taken to compute all the stable models of a logic program using our approach. As can be seen from the graph, the performance of our procedure did not appear to explode exponentially as a function of the number of nodes in the game graph. Beyond that, the results indicate that the time taken to compute stable models increases as a function of n.

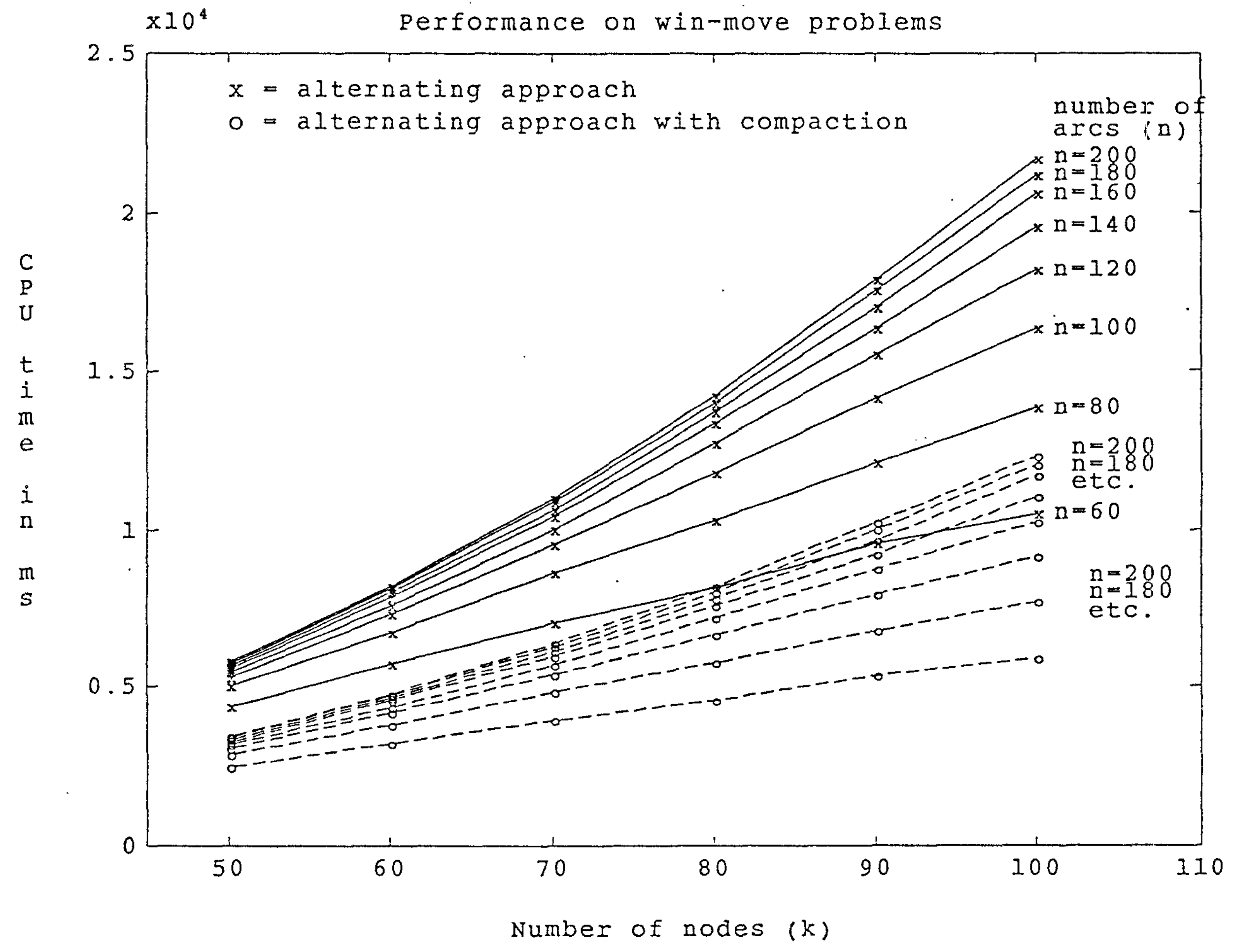
5.3.5 The Impact of Intelligent Branching

In order to determine the effect of intelligent branching, we conducted experiments with two programs. The two programs both had non-trivial dependency graph structures. In both cases, we increased the number of constants while keeping the number of rules constant.

Program 1. This program consisted of the rules shown below.

$$\begin{array}{rcl}
z1(X) & \leftarrow & v1(X), w1(X). \\
z2(X) & \leftarrow & v1(X), w2(X). \\
z3(X) & \leftarrow & v2(X), w1(X). \\
z4(X) & \leftarrow & v2(X), w2(X). \\
v1(X) & \leftarrow & s(X). \\
v2(X) & \leftarrow & t(X). \\
w1(X) & \leftarrow & p(X). \\
w2(X) & \leftarrow & q(X). \\
t(X) & \leftarrow & \neg s(X). \\
s(X) & \leftarrow & \neg t(X). \\
p(X) & \leftarrow & \neg q(X). \\
q(X) & \leftarrow & \neg p(X).
\end{array}$$

The above set of rules was augmented by adding facts of the form $y(-)$ where y is a unary predicate symbol. The predicate y was used solely to introduce constant symbols in the



x10⁴
Performance on win-move problems
x = alternating approach
o = alternating approach with compaction
number of arcs (n)
n=200
n=180
n=160
n=140
n=120
n=100
n=80
n=200
n=180
etc.
n=60
n=200
n=180
etc.
CPU time in ms
Number of nodes (k)
0
0.5
1
1.5
2
2.5
50
60
70
80
90
100
110

Fig.11

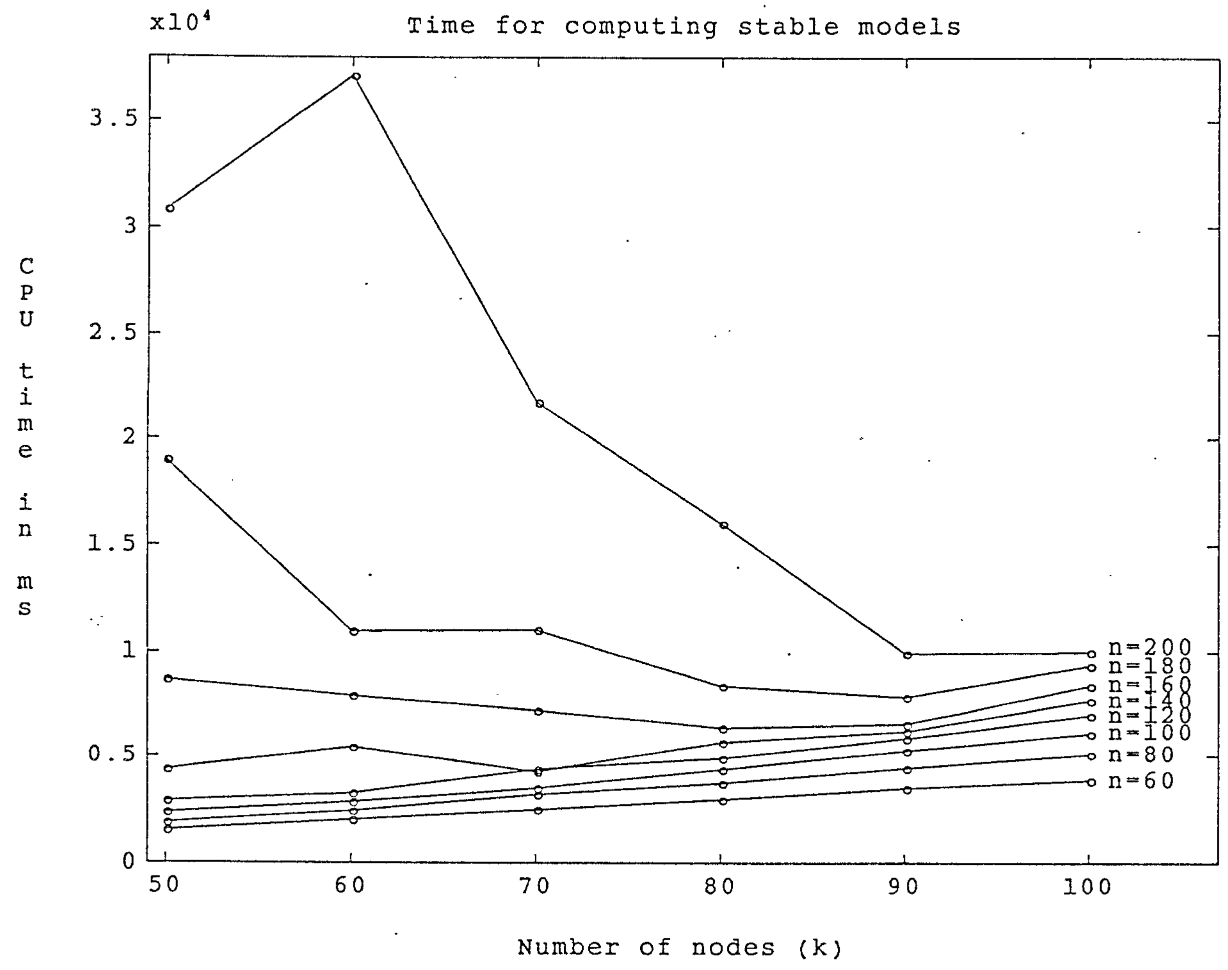

x10^4
Time for computing stable models
CPU time in ms
3.5
3
2.5
2
1.5
1
0.5
0
50
60
70
80
90
100
Number of nodes (k)
n=200
n=180
n=160
n=140
n=120
n=100
n=80
n=60

Fig.12

language. This program has 4^n stable models where n is the number of constants in our language. The table below shows the results of using the naive branch and bound strategy as opposed to the intelligent branching strategy. It is clear that the intelligent branching significantly speeds up the computation. All times given below are in milliseconds. The times reported below include the times taken to construct the dependency graph associated with a program, and to compute the sets $\mathbf{E}_0, \mathbf{E}_1, \ldots$ described in Section 4.3.

Number of Constants	1	2	3	4	5
Naive Branch and Bound	101	637	3165	16744	129186
Intelligent Branch and Bound	43	262	1413	9431	95766
Number of Stable Models	4	16	64	256	1024

Program 2. This program consisted of the rules shown below.

$$\begin{array}{lcl} s(X) & \leftarrow & p(X), q(X). \\ s(X) & \leftarrow & p(X), r(X). \\ s(X) & \leftarrow & q(X), r(X). \\ p(X) & \leftarrow & \neg q(X). \\ q(X) & \leftarrow & \neg r(X). \\ r(X) & \leftarrow & \neg p(X). \end{array}$$

As before, the above set of rules was augmented by adding facts of the form $y(-)$ where y is a unary predicate symbol. The predicate y was used solely to introduce constant symbols in the language. The program has *no* stable models at all, and hence, both the naive branch and bound strategy, as well as the intelligent branching strategy need to search almost the whole of $ACT(P)$. The table below shows the results of using the naive branch and bound strategy as opposed to the intelligent branching strategy. It is clear that the intelligent branching significantly speeds up the computation. All CPU times given below are in milliseconds.

Number of Constants	Without Intelligent Branching	With Intelligent Branching
5	105	54
10	224	117
15	346	198
20	482	303
25	668	431
30	873	586
35	1117	755
40	1379	972
45	1691	1203
50	2008	1475

On programs that generated dependency graphs with little or no structure, we found that the effect of intelligent branching was relatively minor.

5.3.6 The Impact of Grounding

Figure 13 plots the ratio of the time taken to ground a program, as compared to the total time taken to compute the well-founded semantics of the program (including the time taken to ground the program). In all cases, it can be seen that:

- the time taken to ground the program is less than half the total time taken to compute the well-founded semantics and
- as n increases, the time taken to ground the program becomes an increasingly small percentage of the total time.

Exactly the same conclusions may be drawn when we plot the ratio of the time taken to ground a program, as compared to the total time taken to compute the set of stable models of the program (which includes the time taken to ground the program). Figure 14 shows this graph.

There is a theoretical explanation for why grounding is not such a major problem. Suppose b is a *fixed* integer representing an upper bound on the number of distinct variable symbols that are allowed to occur in a clause. Almost no logic program that we have seen in practice contains clauses having more than, say, 10-12 distinct variables in it. In such a case, the total number of ground clauses of a logic program P is bounded above by $k \times c^b$ where k is the number of clauses in P, and c is the number of constants in P. This is a polynomial-time expression when b is fixed in advance.

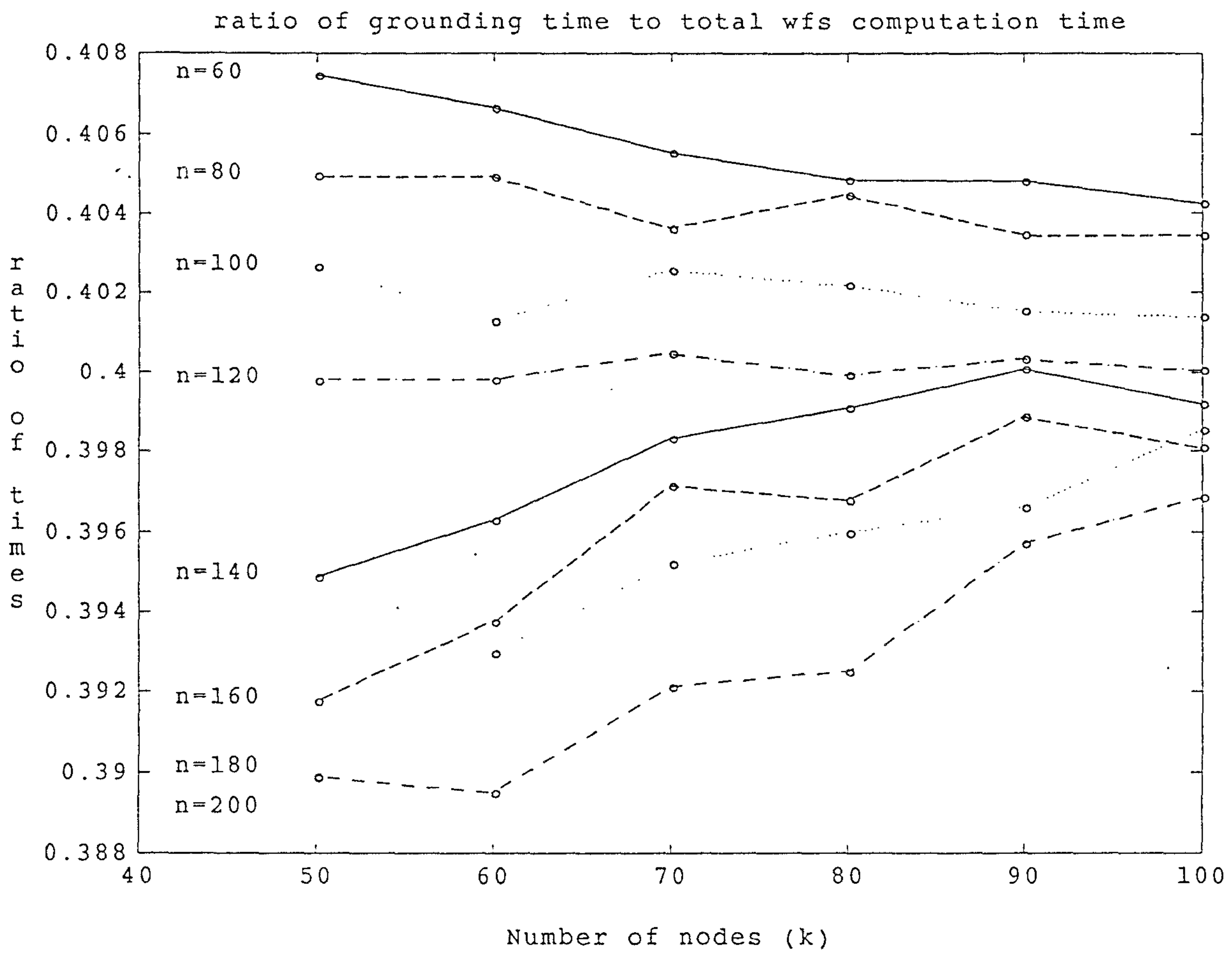
Almost all work (cf. Ullman [32], Vardi [37]) on complexity of deductive database ("data-complexity" and "expression-complexity") make similar assumptions. The standard assumption is that predicates are of bounded arity, i.e. we assume a priori that there is a fixed upper bound, b_0, that bounds the arity of any predicate allowed to occur in a program. Thus, when statements like "Query answering in definite datalog programs can be can be answered in polynomial time" are made, they implicitly make the bounded arity assumption. Without this assumption, it is not difficult to show that definite (i.e. negation-free) datalog programs lead to EXPTIME-completeness [37].

6 Discussion

Though it is now almost five years since the development of the well-founded semantics and stable semantics, relatively little work has been done on implementing these alternative semantics. To our knowledge, this is the first work which shows precisely how to compute the stable semantics by using computation of the well-founded semantics as a first step.

Computation of well-founded semantics of logic programs has been studied by Kemp et. al. [14] and Warren [8, 39]. Kemp et. al. show how, given a query Q to a logic program P, and a sideways information passing strategy[9] S, it is possible to create a new program

[9] See [14] for an explanation of this expression



ratio of grounding time to total wfs computation time
ratio of times
Number of nodes (k)
n=60
n=80
n=100
n=120
n=140
n=160
n=180
n=200
0.408
0.406
0.404
0.402
0.4
0.398
0.396
0.394
0.392
0.39
0.388
40
50
60
70
80
90
100

Fig.13

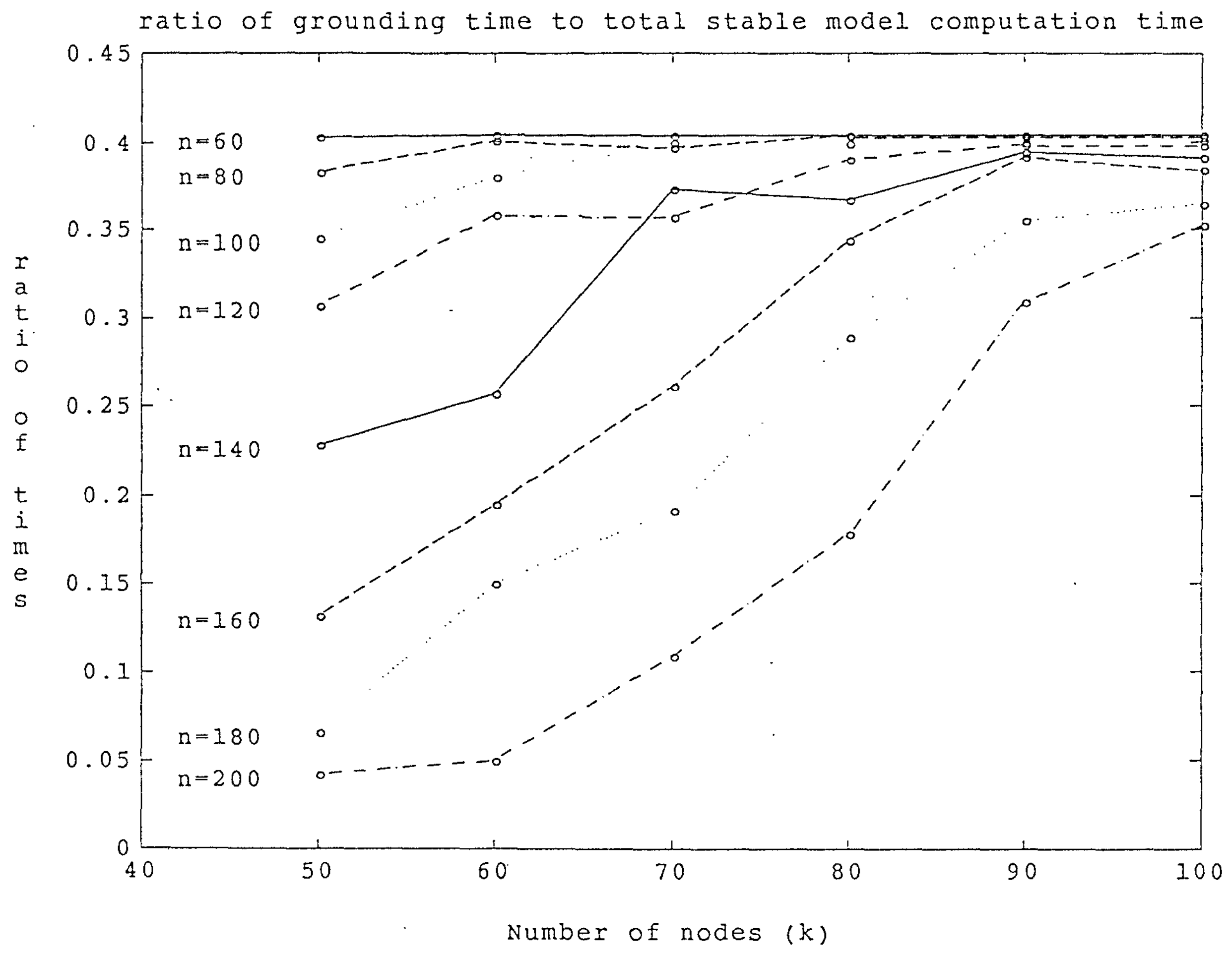

ratio of grounding time to total stable model computation time
ratio of times
Number of nodes (k)
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
40
50
60
70
80
90
100
n=60
n=80
n=100
n=120
n=140
n=160
n=180
n=200

Fig.14

$Magic(P, S, Q)$. More importantly, this new program has the same well-founded semantics as the original program P, and has a particular syntactic form. Kemp et. al.[14] show how the query Q may be answered w.r.t. the new program $Magic(P, S, Q)$. Warren[39] shows how to construct a Prolog meta-interpreter for the well-founded semantics based on OLDT-resolution. Warren's technique uses a table to tabulate previously solved goals – this avoids redundant computation. Chen and Warren [8] extend the work in [39] and develop a sound and complete technique for computing WFS called XOLDTNF-resolution.

The main difference between our work and that of Warren and Kemp et. al. is that our compilation technique is *query-independent*, while in their case, the query plays a key role in transforming the program P. Thus, our technique may be applied at compile-time, and hence is more suitable in situations where very quick run-time responses are desired: in our overall architecture, run-time query evaluation is done by a standard run-time query language implementation. In contrast, the methods of Kemp et. al. are query-dependent, and hence, the work of creating $Magic(P, S, Q)$ is done after the query Q has been asked, i.e. at run-time.

Another advantage of computing the well-founded semantics at compile-time and storing it in a relational format is that more expressive queries, such as aggregate queries, need not be specially developed for this purpose. Furthermore, standard techniques developed by relational database researchers for run-time query optimization may now be used. On the other hand, aggregate query processing techniques need to be specifically developed for the magic set approach [36, 30]. These techniques involve deduction at run-time.

A disadvantage of our approach vis-a-vis the approach of Kemp et. al. is that we do more work at compile-time, and as we are storing the well-founded model, we have larger space requirements. A lot of work has been done by the relational database community on storing very large databases on auxiliary storage. For instance, the US Census Bureau's database is approximately 15 Gigabytes in size. NASA's EOS database (Earth Observing System) is approximately 10^{15} bytes in size. Hence, we believe that storage is not such a major problem. It is possible that a suitable trade-off between the two approaches is desirable in a full-fledged working system: use our approach to compile those parts of the database involving predicates that require "rapid" run-time responses, and use the Kemp. et. al. approach to handle other predicates.

To summarize, we believe that those parts of a database involving "real-time" predicates (cf. Maler, Manna and Pnueli [23]) need to be processed at compile-time using techniques such as ours. Those parts of a database that do *not* involve real-time predicates do not need to be pre-processed, and in such cases, the techniques of Kemp. et. al. [14] and Warren [39] are perhaps more appropriate.

This work is part of the LOPS ("Logic and Optimization for Problem Solving") project between Cornell, Maryland and Intermetrics, Inc. The overall goal of the project is to develop intelligent support for real-time control systems. Kohn and Nerode (cf. their invited paper at the 1992 IEEE Symposium on Computer-Aided Control Systems Design [20, 21]) have argued that logic programming and deductive database support is critically needed for intelligent control applications. The point has been argued independently by

Kohn [17, 18, 19]. The techniques reported here are intended to be used on that part of a deductive database where fast run-time performance is expected and no time is available for performing deduction at run-time. Techniques such as those of Kemp et. al. [14] and Warren [39] are appropriate on those parts of the database where this is not required (thus leading to storage gains).

7 Concluding Remarks

Though non-monotonic modes of negation have been studied extensively in deductive databases and logic programming, relatively little work has been done on the computation and implementation of non-monotonic semantics. In this paper, we take a first step towards developing a compiled approach for computing the

- well-founded model of non-monotonic deductive databases and
- the set of stable models of non-monotonic deductive databases.

We believe that the desired run-time performance of different parts of a deductive database system is likely to vary. A database system that interacts with a real-time control system, for instance, is likely to contain predicates, some of which need to be processed in real-time, others which do not need to be processed particularly rapidly, and still others that fall between these two extremes. Those parts of the database that deal with "real-time" predicates need to be pre-compiled in advance. Run-time efficiency compromises are *not* an option in such cases. In such cases, the fastest known technology for run-time query processing is the relational database scheme. We suggest, therefore, that the part of a database dealing with predicates whose run-time responses are of critical importance, be completely compiled in advance. One way of doing such compilation is described in this paper when the desired semantics is the well-founded semantics/stable model semantics.

Future research will concentrate on the development of the update module shown in Figure 1, and the development of optimal representations (in relational format) for storing the well-founded model and/or the set of stable models. The update module must not only recompute the new well-founded model (or new set of stable models) when an update occurs, but also update the *relational representation* of the well-founded model (resp. set of stable models). We plan to study these topics.

Acknowledgements. We have benefited from conversations with Christos Faloutsos, Wiktor Marek, Anil Nerode, Raymond Ng, Timos Sellis and David S. Warren.

References

[1] K.J. Astrom, A. Benveniste, P. E. Caines, G. Cohen, L. Ljung, and P. Varaiya. (1991) "Facing the Challenge of Computer Science in the Industrial Applications of Control." IEEE Control Systems, Vol. 11, No. 4.

[2] K. R. Apt, H. Blair and A. Walker. (1988) *Towards a Theory of Declarative Knowledge*, in: J. Minker (ed.) Foundations of Deductive Databases and Logic Programming, pps 89–148, Morgan Kaufmann.

[3] C. Baral and V.S. Subrahmanian. (1990) *Stable and Extension Class Theory for Logic Programs and Default Logics*, JOURNAL OF AUTOMATED REASONING, 8, pps 345–366, 1992. Preliminary version in: Proc. 1990 Intl. Workshop on Non-Monotonic Reasoning, Lake Tahoe, June 1990.

[4] C. Baral and V.S. Subrahmanian. (1991) *Dualities between Alternative Semantics for Logic Programming and Non-Monotonic Reasoning*, to appear in JOURNAL OF AUTOMATED REASONING. Preliminary version in: Proc. 1991 Intl. Workshop on Logic Programming and Non-Monotonic Reasoning (eds. A. Nerode, W. Marek and V.S. Subrahmanian), MIT Press.

[5] C. Bell, A. Nerode, R. Ng and V.S. Subrahmanian (1991) *Implementing Deductive Databases by Linear Programming*, CS-TR-2747, University of Maryland. Also available as Math. Sciences Institute TR. Aug. 1991. Preliminary version to appear in Proc. ACM Symposium on Principles of Database Systems, June 1992, San Diego.

[6] C. Bell, A. Nerode, R. Ng and V.S. Subrahmanian (1991) *Computation and Implementation of Non-Monotonic Deductive Databases*, Univ. of Maryland CS-TR-2801 and Math. Sciences Institute TR-91-67, Dec. 1991.

[7] P.E. Caines and S. Wang. (1991) *COCOLOG: A Conditional Observer and Controller Logic for Finite Machines*, Proc. 29th IEEE Conf. on Decision and Control.

[8] W. Chen and D.S. Warren. (1992) *A Practical Approach to Computing the Well-Founded Semantics*, to appear in Proc. 1992 Intl. Conf. on Logic Programming, Nov. 1992.

[9] J. Fernandez, J. Lobo, J. Minker and V.S. Subrahmanian. (1991) *Disjunctive LP + Integrity Constraints = Stable Model Semantics*, submitted for journal publication, Preliminary version was presented at the Intl. Symp. on Artificial Intelligence and Mathematics, Fort Lauderdale, Jan. 1992.

[10] M.C. Fitting. (1985) *A Kripke-Kleene Semantics for Logic Programming*, J. of Logic Programming, 4, pps 295–312.

[11] M.C. Fitting. (1991) *Well-Founded Semantics, Generalized*, Proc. 1991 Intl. Logic Programming Symp., (eds. V.Saraswat and K. Ueda), pps 71–84, MIT Press.

[12] L. O. Fuentes. (1991) *Applying Uncertainty Formalisms to Well-Defined Problems*, manuscript.

[13] M. Gelfond and V. Lifschitz. (1988) *The Stable Model Semantics for Logic Programming*, in: Proc. 5th International Conference and Symposium on Logic Programming, ed R. A. Kowalski and K. A. Bowen, pp 1070–1080.

[14] D. Kemp, P.J. Stuckey and D. Srivastava. (1991) *Magic Sets and Bottom-up Evaluation of Well-Founded Models*, Proc. 1991 Intl. Logic Programming Symp. (eds. V. Saraswat and K. Ueda), pps 337–351, MIT Press.

[15] S.C. Kleene. (1952) *Introduction to Metamathematics*, North Holland.

[16] D. E. Knuth. (1973) *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison Wesley.

[17] W. Kohn. (1988) *A Declarative Theory for Rational Controllers*, Proc. 27th Conference on Decision and Control, pps 130-136, Austin, Texas, 1988.

[18] W. Kohn. (1990) *Declarative Multiplexed Rational Controllers*, Proc. 5th IEEE International Symposium on Intelligent Control, pps 794–803, Philadelphia, 1990.

[19] W. Kohn. (1991) *Declarative Control Architecture*, Communications of the ACM, 34, 8, pps 64–79, 1991.

[20] W. Kohn and A. Nerode. (1992) *An Autonomous Systems Control Theory: An Overview*, Invited Address. Proc. 1992 IEEE Symp. on Computer-Aided Control Systems Design, pps 204–210.

[21] W. Kohn and A. Nerode. (1992) manuscript in preparation for submission to 1992 Conf. on Decision and Control.

[22] J.W. Lloyd. (1987) *Foundations of Logic Programming*, Springer.

[23] O. Maler, Z. Manna and A. Pnueli. (1991) *From Timed to Hybrid Systems*, Proc. Symp. on Real-Time Systems.

[24] W. Marek, A. Nerode and J. Remmel. (1990) *A Theory of Non-Monotonic Rule Systems, I*, Annals of Mathematics and Artificial Intelligence, 1991. Preliminary version in: Proc. 1990 Conf. on Logic in Computer Science.

[25] W. Marek and M. Truszczynski. (1989) *Stable Semantics for Logic Programs and Default Theories*, Proc. 1989 North American Conf. on Logic Programming (eds. E. Lusk and R. Overbeek), pps 243–256, MIT Press.

[26] W. Marek and M. Truszczynski. *Relating Autoepistemic and Default Logics*, Technical report, Department of Computer Science, University of Kentucky at Lexington, 1989. Preliminary version in: Proceedings of the 1989 Conf. on Knowledge Representation and Reasoning, Morgan-Kaufmann.

[27] A. Nerode, R. Ng and V.S. Subrahmanian. (1991) *Computing Circumscriptive Deductive Databases*, MSI TR 91-66, Dec. 1991.

[28] A. Nerode and V.S. Subrahmanian. (1992) *Hybrid Knowledge Bases*, Tech. Report, Math. Sciences Institute, Cornell University.

[29] T. Przymusinski. (1988) *On the Declarative Semantics of Deductive Databases and Logic Programs*, in: J. Minker (ed.) "Foundations of Deductive Databases and Logic Programming," pps 193–216. Morgan-Kaufmann.

[30] K. Ross and Y. Sagiv. (1992) *Monotonic Aggregation in Deductive Databases*, to appear: Proc. 11th ACM Symp. on Principles of Database Systems, San Diego, CA, June 1992.

[31] O. Shmueli, S. Tsur and C. Zaniolo. (1988) *Rewriting of Rules Containing Set Terms in a Logic Data Language*, in: Proc. 7th Symp. on Principles of Database Systems, pps 15–28.

[32] J. D. Ullman. (1988) *Principles of Database and Knowledge-Base System, Vol. 1*, Computer Science Press.

[33] J. D. Ullman. (1989) *Bottom-Up Beats Top-Down for Datalog*, Proc. 1989 Symp. on Principles of Database Systems, pps 140–149.

[34] A. van Gelder. (1989) *The Alternating Fixpoint of Logic Programs with Negation*, Proc. 8th ACM Symp. on Principles of Database Systems, pps 1 – 10.

[35] A. van Gelder, K. Ross and J. Schlipf. (1988) *Unfounded Sets and Well-founded Semantics for General Logic Programs*, in Proc. 7th Symposium on Principles of Database Systems, pp 221-230.

[36] A. van Gelder. (1989) *The Well-Founded Semantics of Aggregation*, to appear: Proc. 11th ACM Symp. on Principles of Database Systems, San Diego, CA, June 1992.

[37] M. Vardi. (1982) *The Complexity of Relational Query Languages*, Proc. 14th ACM Symp. on Theory of Computing, San Francisco, 1982, pp. 137–146.

[38] D.S. Warren. (1989) *The XWAM: A Machine that Integrates Prolog and Deductive Database Query Evaluation*, Tech Report #89-25, SUNY at Stony Brook, 1989.

[39] D.S. Warren. (1991) *Computing the Well-Founded Semantics of Logic Programs*, SUNY/Stonybrook Tech. Report TR 91-12, June 1991.

Appendix: Proofs of Theorems and Lemmas

Proof of Theorem 1.

(1) Suppose $I_n(A) = \mathsf{lfp}(\Phi_P)$. Then there exists a smallest integer $k \leq n$ such that $I_k(A) = \mathsf{lfp}(\Phi_P)$. We proceed by induction on k and show that $I_n(A) = \mathsf{lfp}(\Phi_P)(A)$.

Base Case. ($k = 0$) In this case, $I_n(A) = \bot$. It follows that for all $0 \leq j \leq n$, $I_j(A) = \bot$. Suppose $\mathsf{lfp}(\Phi_P)(A) \neq \bot$. Then there exists an integer k such that $\Phi_P \uparrow r(A) = \mathsf{lfp}(\Phi_P)(A) \neq \bot$. It follows by a straightforward induction on r that this will always lead to a contradiction. The induction hypothesis is that for all integer r', if $\Phi_P \uparrow r'(A) = \mathsf{lfp}(\Phi_P)(A)$, then $I_n(A) = \mathsf{lfp}(\Phi_P)(A)$.

Inductive Case. (k+1) There are three cases to consider, depending upon whether $I_{k+1}(A) =$ **t** or **f** or **u**. We consider these cases one by one.

$\boxed{I_n(A) = I_{k+1}(A) = \mathbf{t}}$ Suppose $I_n(A) = I_{k+1}(A) = \mathbf{t}$. By definition of I_{k+1}, there exists a clause

$$A \leftarrow L_1 \,\&\ldots\&\, L_s$$

in P_k such that $I_k(L_i) = \mathbf{t}$ for all $1 \leq i \leq s$. By the induction hypothesis, $\mathsf{lfp}(\Phi_P)(L_i) = \mathbf{t}$ for all $1 \leq i \leq s$. But then $\Phi_P(\mathsf{lfp}(\Phi_P))(A) = \mathbf{t}$, i.e. $\mathsf{lfp}(\Phi_P)(A) = \mathbf{t}$.

$\boxed{I_n(A) = I_{k+1}(A) = \mathbf{f}}$ Suppose $I_n(A) = \mathbf{f}$. By definition of I_{k+1}, it follows that for every clause

$$A \leftarrow L_1 \,\&\ldots\&\, L_s$$

in P_k, there exists a literal L_i such that $I_k(L_i) = \mathbf{f}$. By the induction hypothesis, $\mathsf{lfp}(\Phi_P)(L_i) = \mathbf{f}$. But then $\Phi_P(\mathsf{lfp}(\Phi_P))(A) = \mathbf{f}$, i.e. $\mathsf{lfp}(\Phi_P)(A) = \mathbf{f}$.

$\boxed{I_n(A) = I_{k+1}(A) = \bot}$ Similar to the base case.

As $I_n(A)$ must be either **t** or **f** or **u**, it follows from the above that in each of these three cases, $I_n(A) = \mathsf{lfp}(\Phi_P)(A)$.

This completes the proof of **(1)**.

(2) The proof is similar to (and easier than) the proof of **(3)** below.

(3) By **(1)** above, $I_n(A) = \mathbf{f}$ implies that $\mathsf{lfp}(\Phi_P)(A) = \mathbf{f}$. Hence, there is a smallest integer $m > 0$ such that $\Phi_P \uparrow m(A) = \mathbf{f}$. We proceed by induction on m.

If ($m = 1$), then there is no clause in P with A as the head. Consequently, $A \notin F_P(\emptyset)$, and hence, as $\mathsf{gfp}(F_P^2) \subseteq F_P^{2r+1}(\emptyset)$ for all $r \geq 0$[3, 4, 34], it follows that $A \notin \mathsf{gfp}(F_P^2)$. Hence, $A \in$ **wfs_false**(P).

If ($m > 1$), then for every clause C_i of the form

$$A \leftarrow L_1^i \,\&\ldots\&\, L_{k_i}^i$$

in P, there exists a literal $L_{\alpha(i)}^i$ such that $\Phi_P \uparrow (m-1)(L_{\alpha(i)}^i) = \mathbf{f}$. By the induction hypothesis, $L_{\alpha(i)}^i \in$ **wfs_false**(P), i.e. $L_{\alpha(i)}^i \notin \mathsf{gfp}(F_P^2)$. Let $C_1, \ldots, C_r$ be all clauses with

A in the head of the above form. Then there is an odd integer s such that

$$F_P \uparrow s \cap \{L^1_{\alpha(i)}, \ldots, L^s_{\alpha(s)}\} = \emptyset.$$

As s is odd, we know that $F_P \uparrow (s+2) \subseteq F_P \uparrow s$, and consequently, we may conclude that $A \notin F_P^2 \uparrow (s+2)$. Hence, $A \notin \mathsf{gfp}(F_P^2)$; this establishes that $A \in$ **wfs_false**(P). ∎

Proof of Lemma 2. (1) Suppose $\mathsf{lfp}(\Phi_P)(A) \neq \mathbf{u}$. Then, by Theorem 1, we may conclude that there is a smallest integer k such that $I_k(A) \neq \mathbf{u}$. In this case, A does not occur anywhere, either positively or negatively, in P_{k+1}. To see this, observe that $P_{k+1} = mod(P_k, I_k)$. If A occurs in the head of a clause in P_k, then that clause is deleted by part (1) of the definition of $mod(P_k, I_k)$. Four cases arise depending upon whether $I(A)$ is **t** or **f**, and depending upon whether A occurs positively, or negatively, in the body of a clause C. Parts (2) – (5) of the definition of $mod(P_k, I_k)$ handle these four cases. Occurrences of A are eliminated by either deleting the occurrences, or by deleting the entire clause.

(2) Follows immediately from the definition of Φ_P and the construction of **mi_target**(P). ∎

Proof of Lemma 3. Suppose $A \in$ **wfs_true**(P). Then A is in every stable model of P by results of [4, 34].

$(\rightarrow)$ Suppose I is a stable model of Q. We need to show that $I \cup \{A\}$ is a stable model of P, i.e. we need to show that $I \cup \{A\} = F_P(I \cup \{A\})$.

I. $(I \cup \{A\} \subseteq F_P(I \cup \{A\}))$ Suppose $B \in I \cup \{A\}$. There are two cases:

Case 1: $(B \neq A)$ In this case, $B \in I$ and hence, as I is a stable model of Q, $B \in F_Q(I) = T_{Q^I} \uparrow \omega$. Hence, there is a smallest integer $n > 1$ such that $B \in T_{Q^I} \uparrow n$. We proceed by induction on n.

$\boxed{\text{Base Case: } (n=1)}$ Thus, $B \leftarrow$ is in Q^I and hence, Q contains a clause C of the form

$$B \leftarrow \neg B_1 \,\&\ldots\&\, \neg B_k$$

such that $\{B_1, \ldots, B_k\} \cap I = \emptyset$. As $C \in Q$, and as Q is obtained from P using the transformation specified in the statement of the Lemma, we know that for all $1 \leq i \leq k$, $A \neq B_i$. Hence, $\{B_1, \ldots, B_k\} \cap (I \cup \{A\}) = \emptyset$. At this stage, there are two possibilities: (1) either $C \in P$, or (2) C was obtained from a clause $C' \in P$ by deleting positive occurrences of A from the body of C'. In the first case, as none of the B_i's, $1 \leq i \leq k$, is in $I \cup \{A\}$, it follows that $B \leftarrow \in P^{I \cup \{A\}}$ and hence, $B \in F_P(I \cup \{A\})$.

In the second case,

$$B \leftarrow A \,\&\, \neg B_1 \,\&\ldots\&\, \neg B_k$$

is in P. As $\{B_1, \ldots, B_k\} \cap (I \cup \{A\}) = \emptyset$, it follows that $B \leftarrow A$ is in $P^{(I \cup \{A\})}$. To show that $B \in F_P(I \cup \{A\})$, it suffices to show that $A \in F_P(I \cup \{A\})$ because of the presence of the rule $B \leftarrow A$ in $P^{(I \cup \{A\})}$. As $A \in$ **wfs_true**(P), it follows, by results of [4, 34], that there is a smallest integer s such that $A \in (F_P^2) \uparrow s$. It follows by a straightforward induction on s that $A \in F_P(I \cup \{A\})$. Hence, $B \in F_P(I \cup \{A\})$.

$\boxed{\text{Inductive Case } (n = m+1)}$ In this case, there is a clause $B \leftarrow D_1 \,\&\ldots\&\, D_w$ in Q^I such that $\{D_1, \ldots, D_w\} \subseteq T_{Q^I} \uparrow m$. By the induction hypothesis, we may assume that for all

$1 \leq i \leq w$, $D_i \in F_P(I \cup \{A\})$. Thus, Q contains a clause C of the form

$$B \leftarrow \neg B_1 \& \ldots \& \neg B_k \& D_1 \& \ldots \& D_w$$

such that $\{B_1, \ldots, B_k\} \cap I = \emptyset$. The rest of the proof is identical to the base case.

This completes the proof that $I \cup \{A\} \subseteq F_P(I \cup \{A\})$.

II. $(F_P(I \cup \{A\}) \subseteq I \cup \{A\})$ Suppose $B \in F_P(I \cup \{A\})$. Then there is a smallest integer v such that $B \in T_{P(I \cup \{A\})} \uparrow v$. The proof is by a straightforward induction on v.

$(\leftarrow)$ $I \cup \{A\}$ is a stable model of P. We need to show that I is a stable model of Q, i.e. we need to show that $I = F_Q(I)$. Both inclusions are proved in the same way as in the $(\rightarrow)$ case. ■

Proof of Theorem 2. We prove the result by a simultaneous induction on (1) & (2).

Base Case. $(i = 0)$ In this case, $\mathbf{wfs\_true}(P) = \emptyset = \mathbf{wfs\_true}(P_0) \cup \mathbf{glo\_true}_0(P) = \emptyset \cup \emptyset$. The same holds of **wfs_false**. This completes this case.

Inductive Case. $(i+2)$ We assume, without loss of generality, that i is an even number. The induction hypothesis is that for all even numbers $j \leq i$, $\mathbf{wfs\_true}(P) = \mathbf{wfs\_true}(P_j) \cup \mathbf{glo\_true}_j(P)$ and $\mathbf{wfs\_false}(P) = \mathbf{wfs\_false}(P_j) \cup \mathbf{glo\_false}_j(P)$.

[$\mathbf{wfs\_true}(P) \subseteq \mathbf{wfs\_true}(P_{i+2}) \cup \mathbf{glo\_true}_{i+2}(P)$ and $\mathbf{wfs\_false}(P) \subseteq \mathbf{wfs\_false}(P_{i+2}) \cup \mathbf{glo\_false}_{i+2}(P)$] By the induction hypothesis, $\mathbf{wfs\_true}(P) \subseteq \mathbf{wfs\_true}(P_i) \cup \mathbf{glo\_true}_i(P)$ and $\mathbf{wfs\_false}(P) \subseteq \mathbf{wfs\_false}(P_i) \cup \mathbf{glo\_true}_i(P)$.

Suppose $A \in \mathbf{wfs\_true}(P)$. Then $A \in \mathbf{wfs\_true}(P_i) \cup \mathbf{glo\_true}_i(P)$. If $A \in \mathbf{glo\_true}_i(P)$, then, as $\mathbf{glo\_true}_i(P) \subseteq \mathbf{glo\_true}_{i+2}(P)$, $A \in \mathbf{glo\_true}_{i+2}(P)$.

On the other hand, suppose $A \in \mathbf{wfs\_true}(P_i) - \mathbf{glo\_true}_i(P)$. Then $A \in F^2_{P_i} \uparrow k$ for some minimal integer k. It can be established, by a straightforward induction on k, that then $A \in \mathbf{wfs\_true}(P_{i+2}) \cup \mathbf{glo\_true}_{i+2}(P)$. Intuitively, the reason for this is the following: as i is even, $\mathbf{glo\_true}_{i+1}(P) = \mathbf{glo\_true}_i(P)$, and hence, $A \notin \mathbf{glo\_true}_{i+1}(P)$. Furthermore, $A \notin I_i$ as $I_i \subseteq \mathbf{glo\_true}_i(P)$. Furthermore, $A \notin \mathbf{glo\_false}_{i+1}(P)$; were this the case, we would have $A \in \mathbf{glo\_false}_i(P) \cup (B_{P_i} - F_{P_i}(I_i))$. $A \in \mathbf{glo\_false}_i(P)$ would contradict $A \in \mathbf{wfs\_true}(P_i)$ and $A \in (B_{P_i} - F_{P_i}(I_i))$ would do the same. Consequently, as $A \in \mathbf{wfs\_true}(P_i)$, A must either be in $\mathbf{wfs\_true}(P_{i+2})$ or in $\mathbf{glo\_true}_{i+2}(P)$.

The proof that $\mathbf{wfs\_false}(P) \subseteq \mathbf{wfs\_false}(P_{i+2}) \cup \mathbf{glo\_false}_{i+2}(P)$ is symmetric.

[$\mathbf{wfs\_true}(P) \supseteq \mathbf{wfs\_true}(P_{i+2}) \cup \mathbf{glo\_true}_{i+2}(P)$ and $\mathbf{wfs\_false}(P) \supseteq \mathbf{wfs\_false}(P_{i+2}) \cup \mathbf{glo\_false}_{i+2}(P)$] The proof is by a similar induction on i. ■

Proof of Theorem 3. We wish to prove that

$$\mathbf{wfs\_true}(P) = \mathbf{mi\_true}(P) \cup \mathbf{glo\_true}(\mathbf{mi\_target}(P))$$

and

$$\mathbf{wfs\_false}(P) = \mathbf{mi\_false}(P) \cup \mathbf{glo\_false}(\mathbf{mi\_target}(P)).$$

By theorem 1, we know that $\mathbf{mi\_true}(P) \subseteq \mathbf{wfs\_true}(P)$ and $\mathbf{mi\_false}(P) \subseteq \mathbf{wfs\_false}(P)$. $\mathbf{mi\_target}(P)$ is obtained from P by performing the transformations specified in Definition 6. All changes made in the five cases of Definition 6 preserve the well-founded semantics when the interpretation I is an interpretation that occurs in the I-sequence associated with P[10]. An important point to note is that in the pruning iteration, we only perform transformations w.r.t. interpretations I such that $I \preceq \mathsf{lfp}(\Phi_P)$.

Case 1: According to this case, if A occurs in the head of a clause C, and I_j is an interpretation in the I-sequence associated with P which assigns either $\mathbf{t}$ or $\mathbf{f}$ to A, then C gets deleted. If I_j makes A true, then $\mathsf{lfp}(\Phi_P)(A) = \mathbf{t}$ and so $A \in \mathbf{wfs\_true}(P)$ by Theorem 1. But then, the clause C cannot make anything else true and hence can be deleted. Similarly, if $I_j(A) = \mathbf{f}$, then $\mathsf{lfp}(\Phi_P)(A) = \mathbf{f}$ and so $A \in \mathbf{wfs\_false}(P)$ by Theorem 1. But then, C cannot make anything else false (by itself) and hence can be deleted.

Cases 2,4: These cases delete clauses whose bodies contain literals that are false in Fitting's semantics (and hence in WFS by Theorem 1). This is clearly sound.

Cases 3,5: These cases delete literals that are true. This is clearly sound.

Consequently, every atom A that is assigned "unknown" by $\mathsf{lfp}(\Phi_P)$ is unaffected by the five transformations of Definition 6 in the sense that the well-founded semantics of P and the well-founded semantics of $\mathbf{mi\_target}(P)$ agree on A. But the well-founded semantics of $\mathbf{mi\_target}(P)$ is given by the sets $\mathbf{glo\_true}(\mathbf{mi\_target}(P))$ and $\mathbf{glo\_false}(\mathbf{mi\_target}(P))$ by Corollary 1. This completes the proof. ■

Proof of Lemma 5. (1) follows immediately from the fact that if this were not the case, then A would be assigned either the truth value $\mathbf{t}$ or $\mathbf{f}$. (2) follows immediately from the construction of $\mathbf{glo\_simp}(P)$. ■

Proof of Theorem 4. ($\leftarrow$) Suppose $I \in \mathbf{MIN\_LEAF}(\mathbf{glo\_simp}(P))$. We need to show that I is a stable model of $\mathbf{glo\_simp}(P)$, i.e. we need to show that $I = T_{\mathbf{glo\_simp}(P)^I} \uparrow \omega$.

Part 1: $[I \subseteq T_{\mathbf{glo\_simp}(P)^I} \uparrow \omega]$ Suppose $a \in I$. Let $N_0, \ldots, N_k$ be the nodes from the root of $PRUNE\_ACT(P)$ to the leaf in question, and let

$$(q_0, T_0, F_0, U_0), \ldots, (q_k, T_k, F_k, U_k)$$

be the sequence of node-labels in the path from the root of $PRUNE\_ACT(P)$ to the leaf node having $T_k = I$. In particular, note that the following relations hold:

$$\emptyset = T_0 \subseteq T_1 \subseteq T_2 \ldots \subseteq T_k$$

$$\emptyset = F_0 \subseteq F_1 \subseteq F_2 \ldots \subseteq F_k$$

$$U_0 \supseteq U_1 \supseteq U_2 \ldots \supseteq U_k = \emptyset.$$

Furthermore, suppose $a_1, \ldots, a_k$ are the atoms on which we branch when following the branch from the root of $PRUNE\_ACT(P)$ to the leaf node having $T_k = I$. Let $i > 0$ be the smallest integer such that $a \in T_i$. We show, by induction on i, that $a \in T_{\mathbf{glo\_simp}(P)^I} \uparrow \omega$.

Base Case ($i = 1$) Two possibilities arise, depending upon whether the branch from N_0 to N_1 is obtained by branching positively, or negatively, on a_1. If it is a positive branch, then $q_1 = \mathbf{CH}(q_0, a_1)$, and $a \in \mathbf{wfs\_true}(\mathbf{CH}(q_0, a_1))$. As $a \in \mathbf{wfs\_true}(\mathbf{glo\_simp}(\mathbf{CH}(q_0, a_1)))$, it

[10]Note that our proof hinges critically upon the fact that the pruning iteration only prunes w.r.t. interpretations I_j occurring in the I-sequence associated with P. Transforming P w.r.t. interpretations that do not occur in the I-sequence associated with P may not preserve well-founded semantics.

follows that $a \in (F^2_{\mathbf{glo\_simp}(\mathbf{CH}(q_0,a_1))}) \uparrow r$ for some integer r. It follows, by a straightforward induction on r, that $a \in T_{\mathbf{glo\_simp}(P)^I} \uparrow \omega$. The inductive hypothesis is that for all ground atoms A, if $A \in (F^2_{\mathbf{glo\_simp}(\mathbf{CH}(q_0,a_1))}) \uparrow k$ then $A \in T_{\mathbf{glo\_simp}(P)^I} \uparrow \omega$ AND if $A \notin (F_{\mathbf{glo\_simp}(\mathbf{CH}(q_0,a_1))}) \uparrow 2k+1$, then $A \notin T_{\mathbf{glo\_simp}(P)^I} \uparrow \omega$.

Inductive Case $(i+1)$: Similar to the base case.

$(\rightarrow)$ Suppose I is a stable model of $\mathbf{glo\_simp}(P)$. Note that each Herbrand interpretation of the language generated by the constant symbols and predicate symbols of $\mathbf{glo\_simp}(P)$ corresponds to a path in $ACT(P)$. We need to show only that if I is a stable model of $\mathbf{glo\_simp}(P)$, then:

1. there is a branch in $ACT(P)$ that corresponds to I in the sense that I is the T-component of the leaf node of the branch.

2. if $N_0, \ldots, N_k$ are the nodes along the branch in $PRUNE\_ACT(P)$ corresponding to I, and if the label of N_i is (q_i, T_i, F_i, U_i), then $U_k = \emptyset$ and $T_k \cap F_k = \emptyset$.

Let $a_1, \ldots, a_k$ be the sequence of atoms on which branches occur in $ACT(P)$. Then we can inductively construct the branch $\mathcal{B}(I)$ corresponding to I as follows: The root of $ACT(P)$ is in $\mathcal{B}(I)$. Suppose N is a node in $\mathcal{B}(I)$, and the arcs emanating from N are labeled with $\neg a_i$ (to the left child N_l) and a_i (to the right child N_r) respectively. If $a_i \in I$, then $N_r \in \mathcal{B}(I)$, else $N_l \in \mathcal{B}(I)$.

Let $N'_0, \ldots, N'_k$ be the sequence of nodes in $\mathcal{B}(I)$. Let s be the smallest integer $1 \leq s \leq k$ such that $U'_k = \emptyset$ where we use (q'_i, T'_i, F'_i, U'_i) to denote the label of the node N'_i. Then the branch $N'_0, \ldots, N'_s$ is in $\mathbf{LEAF}(P)$ *unless* $T_j \cap F_j \neq \emptyset$ for some $1 \leq j \leq s$. The only way $T_j \cap F_j$ could contain something is if assuming one of the $\pm a_w$'s, $1 \leq w \leq j$, led to a previous assumption being contradicted. This is impossible, given the way we are transforming P using the function $\mathbf{CH}$. This completes establishing that I is the T-component of a leaf node in $PRUNE\_ACT(P)$, i.e. $I \in \mathbf{LEAF}(P)$.

$I \in \mathbf{MIN\_LEAF}(\mathbf{glo\_simp}(P))$ for the following reason: when branching on an atom a, the branch and bound algorithm we have specified first branches by assuming that a is false. Thus, if some strict subset I' of I were in $\mathbf{LEAF}(\mathbf{glo\_simp}(P))$, then by the preceding argument, this leaf would have been generated before I, i.e. I' would be in S (where S is the set of stable models found "so far", cf. the branch and bound algorithm of Example 2). But then, by the $(\leftarrow)$ part of this theorem, I' is a stable model. It is known [25] that if I_1, I_2 are distinct stable models, then $I_1 \not\subseteq I_2$ and $I_2 \not\subseteq I_1$. Hence, it is impossible that there is a leaf such that $I' \subset I$, and hence, $I \in \mathbf{MIN\_LEAF}(\mathbf{glo\_simp}(P))$. ■

Proof of Lemma 6. Whenever the algorithm selects from L a node Q of $PRUNE\_ACT(P)$, it removes Q from L; and the nodes Q^- and Q^+ generated in lines 8 and 18 of the algorithm are precisely Q's children in $PRUNE\_ACT(P)$. From this, it is easy to prove by induction that:

1. every node generated by the algorithm is in $PRUNE\_ACT(P)$;

2. every node in $PRUNE\_ACT(P)$ is generated by the algorithm;

3. no node is generated more than once.

From this, it follows that the algorithm generates each node of $PRUNE\_ACT(P)$ exactly once; and thus, from lines 4, 17 and 27 of the algorithm, it follows that it generates them in pre-order. ■

Proof of Corollary 2. $PRUNE\_ACT(P)$ has finitely many nodes. By the proof of Lemma 6, no node is generated twice. The result follows. ■

Proof of Corollary 3. Every node in **LEAF(glo_simp**(P)**)** is a leaf of $PRUNE\_ACT(P)$, and from Lemma 6, it follows that the algorithm generates the leaves of $PRUNE\_ACT(P)$ in left-to-right order. ■

Proof of Lemma 7. If N is to the left of N', then there is a node M somewhere "above" N and N' such that N is a descendant of M's left branch and N' is a descendant of M's right branch. Let a be the atom on which we branch at node M. Then $a \notin T$ and $a \in T'$, so $T' \not\subseteq T$. ■

Proof of Theorem 5. From lines 14–15 and 24–25 of the branch and bound algorithm, it follows that every element added to S is in **LEAF(glo_simp**(P)**)**. Thus, upon termination of the algorithm, $S \subseteq$ **LEAF(glo_simp**(P)**)**. Let $T_1, \ldots, T_n$ be the elements of **LEAF(glo_simp**(P)**)** in left-to-right order. We now prove by induction on i that for $i = 1, \ldots, n$, T_i is added to S by the branch and bound algorithm iff $T_i \in$ **MIN_LEAF(glo_simp**(P)**)**.

Base Case. At the time that the algorithm decides whether to add T_1, $S = \emptyset$, and so T_i is added to S. From Lemma 7, $T_1 \not\subseteq T_i$ for all $i > 1$ and so $T_1 \in$ **MIN_LEAF(glo_simp**(P)**)**.

Inductive Case. Suppose the algorithm is ready to decide whether to add T_i for some $i > 1$, and suppose $S = \{T_j \mid j < i$ and $T_j \in$ **MIN_LEAF(glo_simp**(P)**)**$\}$. Then the algorithm will add T_i to S iff for all $T \in S$, $T \not\subseteq T_i$. There are two cases:

1. $T \subseteq T_i$ for some $T \in S$. Then the algorithm does not add T_i to S and $T_i \notin$ **MIN_LEAF(glo_simp**(P)**)**.
2. $T \not\subseteq T_i$ for all $T \in S$. Then the algorithm will add T_i to S. Suppose there is some T_j such that $T_j \subseteq T_i$ and $j \neq i$. Then, from Lemma 7, it follows that $j < i$. Thus, from the induction hypothesis, $T \subseteq T_j$ for some $T \in S$, whence $T \subseteq T_i$, which is a contradiction. Thus, $T_j \not\subseteq T_i$ for all $j \neq i$, and so $T_i \in$ **MIN_LEAF(glo_simp**(P)**)**.

This completes the proof that the output returned by the branch and bound algorithm is **MIN_LEAF(glo_simp**(P)**)**. That this output coincides with the set of stable models of **glo_simp**(P) follows immediately from Theorem 4. ■