WFS + Branch and Bound = Stable Models

V.S. Subrahmanian, Dana Nau, and Carlo Vago

***Abstract*—Though the semantics of nonmonotonic logic programming has been studied extensively, relatively little work has been done on operational aspects 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.**

***Index Terms*—Logic programming, deductive databases, nonmonotonic reasoning, negation by failure.**

I. 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 [21] and the set of stable models of a logic program [8]. 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 [20], [3] to the compacted program. Our alternating procedure compacts the (already compacted) program further at each step. It is well-known [20], [3] 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.

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 [22], [12] 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.).

II. 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 [16]. We assume that we have an underlying function-free first order language L containing only finitely many constant and predicate symbols. The Herbrand

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

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.

Manuscript received July 9, 1992; revised Sept. 7, 1993.

V.S. Subrahmanian and D. Nau are with the Institute for Advanced Computer Studies, Institute for Systems Research, Department of Computer Science, University of Maryland, College Park, MD 20742; e-mail: vs@cs.umd.edu.

C. Vago is with Universita Degli Studi di Milano, Dipartimento di Scienze della Informazione. Via Moretta da Brescia 9, 20133 Milano, Italy.

IEEECS Log Number K95035.

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 a logic program P. We will use $grd(P)$ to denote the set of ground instances of clauses in P. 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 ([3], [20]).

DEFINITION 1. *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:*

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

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

($m \geq 0$) in $grd(P)$ 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: $F_P(I)$ is defined to be the least Herbrand model of the negation free logic program P^I.

DEFINITION 2. (Gelfond and Lifschitz) *I is a stable model of P iff $I = F_P(I)$.*

PROPOSITION 1. (van Gelder [20], Baral and Subrahmanian [2]) 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.

DEFINITION 3. *Let P be any logic program. Then:*

1) $A \in$ **wfs_true**(P) *iff* $A \in \mathrm{lfp}(F_P^2)$ *and*

2) $A \in$ **wfs_false**(P) *iff* $A \notin \mathrm{gfp}(F_P^2)$.

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

III. COMPUTATION OF WELL-FOUNDED SEMANTICS

Suppose P is a logic program. Our algorithms work with fully instantiated programs. Later, in Section IV.D, we will outline how, given any technique to compute WFS/stable models for *propositional programs*, this method can be lifted to the first order case. However, the details of this first order "lifting" are left to a future paper.

In the *Monotonic Iteration* stage (MI-stage, for short), we mimic the upward iteration of Fitting's Φ_P operator [7] and iteratively build up a set of ground atoms, denoted **mi_true**(P), which are known to be true, and a set **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 mimicking these iterations, the program P undergoes repeated simplification, resulting, in the limit, in a *target* program **mi_target**(P) that is usually considerably simpler than P. In practice, the monotonic iteration phase is efficient (Experiment V.A.1) when compared to the alternating fixpoint computation strategy described in [20], [3].

In the *Gelfond-Lifschitz Oscillation* stage (GLO-stage, for short), we use the simplified program **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 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 [20], [3]:

- First, the GLO-stage applies only to **mi_target**(P) which is usually significantly smaller than P in size (Section V.A.2). The alternating fixpoint approach would use the program, P, which is usually much larger than **mi_target**(P).
- Second, the alternating fixpoint approach [20], [3] 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(**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 (Experiment V.A.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 (Experiment V.A.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.
- Last, but not least, the logic program/deductive database may be updated after its initial creation. Such updates may cause the well-founded model (or the set of stable models) to change. The purpose of the update module is to handle such changes.

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 1. Consider the very simple program containing the following nine clauses:

$$p \leftarrow q \tag{1}$$
$$p \leftarrow r \tag{2}$$
$$q \leftarrow \neg r \,\&\, s \tag{3}$$
$$r \leftarrow \neg q \tag{4}$$
$$s \leftarrow t \tag{5}$$
$$t \leftarrow \tag{6}$$
$$v \leftarrow v \tag{7}$$
$$w \leftarrow \neg v \tag{8}$$
$$u \leftarrow \neg s \tag{9}$$

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:

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

$$p \leftarrow q \tag{10}$$
$$p \leftarrow r \tag{11}$$
$$q \leftarrow \neg r \tag{12}$$
$$r \leftarrow \neg q \tag{13}$$
$$v \leftarrow v \tag{14}$$
$$w \leftarrow \neg v \tag{15}$$

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). It is important to note that according to Fitting's Kripke-Kleene semantics (which does not handle positive loops well [21]), v is concluded to have an "unknown" truth value due to the loop in clause 14). The truth value of w is the negation of "unknown," which is "unknown" too.

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 $I_0 = \varnothing$ and **glo_true**(**mi_target**(P)) = **glo_false**(**mi_target**(P)) = $\varnothing$. We then compute the least model of the Gelfond-Lifschitz transformed program $(\mathbf{mi\_target}(P))^{I_0}$, and denote this least model by I_1. $(\mathbf{mi\_target}(P))^{I_0}$ is the program:

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

$$p \leftarrow q \tag{16}$$
$$p \leftarrow r \tag{17}$$
$$q \leftarrow \tag{18}$$
$$r \leftarrow \tag{19}$$
$$v \leftarrow v \tag{20}$$
$$w \leftarrow \tag{21}$$

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 $\bar{I_1} = \{v\}$, 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:

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

$$p \leftarrow q \tag{22}$$
$$p \leftarrow r \tag{23}$$
$$q \leftarrow \neg r \tag{24}$$
$$r \leftarrow \neg q \tag{25}$$
$$w \leftarrow \tag{26}$$

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)) = $\varnothing$. Thus, the GLO-stage returns, as its final output, the set **glo_false**(**mi_target**(P)) = $\{v\}$ of atoms that are "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

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

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

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

A. The Monotone Iteration Module

In this section, we describe the technical details of the monotone iteration module. We assume that readers are familiar with the well-known Kripke-Kleene three-valued logic, and the three-valued interpretation of logic programming using Fitting's Φ_P operator [7]. Φ_P assigns to atom A if if there is a clause C in *grd*(P) such that A is the head of C and such that I satisfies the body of C. It assigns **f** to A if, for every clause C in *grd*(P) having A as the head, it is the case that I satisfies $\neg$ *Body* where *Body* is the body of C. Otherwise, it assigns **u** to A.

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 [3] so that the modified program is smaller, in terms of the number of occurrences of literals, than P.

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

We use $mod(P, I)$ in the computation of $\mathrm{lfp}(\Phi_P)$ 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 **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 (respectively, false) will stay true (respectively, false) in $\mathrm{lfp}(\Phi_P)$ because Φ_P is monotone w.r.t. the $\preceq$ ordering ($I_j \preceq I_k$ iff for all ground atoms A, $I_1(A) =$ **t** (respectively, **f**) implies that $I_2(A) =$ **t** (respectively, **f**)). Suppose A is an atom that is made true in this process. then 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 **f** instead of **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 5. (Pruning Iteration) *Let P be a logic program, and let $\perp$ be the interpretation that assigns* **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:*

$I_0 = \perp$ $\qquad\qquad P_0 = P$

$I_{j+1}(A) = \Phi_{P_j}(I_j)(A)$ *if* $I_j(A) =$ **u** *and* $I_j(A)$ *otherwise* $\qquad P_{j+1} = mod(P_j, I_j)$.

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 $1 \le j \le k \le n$, then $I_j \preceq I_k$.* □

3. When implementing, we modify a copy of the program P.

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)* $I_n = \mathrm{lfp}(\Phi_P)$.
2) *for all atoms A, if $I_n(A) =$* **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 $I_n(A) = f$ 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 2. Consider the single clause program $P = \{a \leftarrow a\}$. 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* $\mathrm{lfp}(\Phi_P)(A) \neq$ **u**. *Then A does not occur either positively or negatively in* **mi_target**(P). 2) *Suppose* $\mathrm{lfp}(\Phi_P)(A) =$ **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* $\mathrm{lfp}(\Phi_P)$. □

Before proceeding to a detailed description of the GLO-stage, we draw the reader's attention to Fig. 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 Fig. 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.

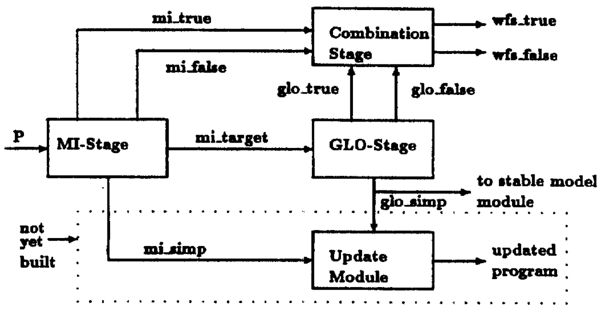


Fig. 1. Architecture of the WFS computation module.

LEMMA 3. *Suppose P is a logic program and A (respectively, B) is a ground atom which is assigned* **t** *(respectively,* **f**) *by the well-founded semantics of P. Let Q (respectively, Q^*) be the program obtained from P by:*

1) *deleting all clauses in P with head A (respectively, B), and*

2) *deleting all clauses in P with $\neg A$ (respectively, B) in the body, and*
3) *deleting positive (respectively, negative) occurrences of A (respectively, B) 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 above lemma indicates 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 true to an atom A, then $A \in$ **wfs_true**(P) and whenever I assigns false 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).

B. The Gelfond-Lifschitz Oscillation Module

As seen in Example 2, 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 ([20], [3]). However, there are a few significant differences which allow our strategy to be much more efficient (Experiment V.A.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 (Experiment V.A.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 [2], [3] and van Gelder [20]): 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): $F_P^0(\varnothing) \subseteq F_P^2(\varnothing) \subseteq \ldots \subseteq F_P^{2i}(\varnothing) \subseteq \ldots$. The *odd* levels of the oscillation form a monotonically decreasing sequence and gradually build up the *complement* of the set **wfs_false**(P): $F_P^1(\varnothing) \supseteq F_P^3(\varnothing) \supseteq \ldots \supseteq F_P^{2i+1}(\varnothing) \supseteq \ldots$. In other words, the sequence, $\overline{F_P^1(\varnothing)} \subseteq \overline{F_P^3(\varnothing)} \subseteq \ldots \subseteq \overline{F_P^{2i+1}(\varnothing)} \subseteq \ldots$ 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 $F_P^1(\varnothing)$, we know that all atoms in $\overline{F_P^1(\varnothing)}$ are *false*. Hence, we can use this information to transform the program P. In the next stage, when we apply F_P to $F_P^1(\varnothing)$, we know that all atoms in the set $F_P^2(\varnothing)$ 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 6. (Transformation Strategy) *Given a program P, and a two-valued interpretation I, we now define a transformation of P w.r.t. I.*[4] *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 7. (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* **glo_true**$_0$, …, **glo_true**$_n$, …, *and a sequence of sets of false atoms* **glo_false**$_0$, …, **glo_false**$_n$, …. *These sequences are constructed as follows:*

$\boxed{j=0}$	$\boxed{j=1}$
$I_0 = \varnothing$	$I_1 = F_{P0}(I_0)$
$P_0 = P$	$P_1 = trans(P_0, I_1, neg)$
glo_true$_0 = \varnothing$	**glo_true**$_1 = \varnothing$
glo_false$_0 = \varnothing$	**glo_false**$_1 = (B_{P0} - I_1)$
$\boxed{\text{For even } j, j > 0}$	$\boxed{\text{For odd } j, j > 1}$
$I_{j+2} = F_{Pj+1}(I_{j+1})$	$I_{j+2} = F_{Pj+1}(I_{j+1})$
$P_{j+2} = trans(P_{j+1}, I_{j+2}, pos)$	$P_{j+2} = trans(P_{j+1}, I_{j+2}, neg)$
glo_true$_{j+2}$ = **glo_true**$_j \cup I_{j+2}$	**glo_true**$_{j+2}$ = **glo_true**$_j$
glo_false$_{j+2}$ = **glo_false**$_j$	**glo_false**$_{j+2}$ = **glo_false**$_j \cup (B_{Pj+1} - I_{j+2})$

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 **glo_true**$_j$ and **glo_false**$_j$.

In order to better illustrate pruning oscillations, we return to

4. Unlike Section III.A where we modified programs using three-valued interpretations, the transformation strategy described here uses two-valued interpretations.

Example 1.

EXAMPLE 3. Consider the program P of Example 1. We focus upon **mi_target(*P*)** which consists of clauses 10–15. Our sequence of Is and Ps is built as follows:

1) $I_0 = \varnothing$.
2) $P_0 = \{10, 11, 12, 13, 14, 15\}$.
3) $\mathbf{glo\_false}_0 = \mathbf{glo\_true}_0 = \varnothing$.
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

$$p \leftarrow q \quad (27)$$
$$p \leftarrow r \quad (28)$$
$$q \leftarrow \neg r \quad (29)$$
$$r \leftarrow \neg q \quad (30)$$
$$w \leftarrow \quad (31)$$

6) $\mathbf{glo\_false}_1 = \overline{I_1} = \{v\}$, and $\mathbf{glo\_true}_1 = \varnothing$.
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 31 being deleted from P_1. Therefore, P_2 consists of clauses 27-30.
9) At this stage, $\mathbf{glo\_false}_2 = \mathbf{glo\_false}_1$, but $\mathbf{glo\_true}_2 = \{w\}$.
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, $\overline{I_3} = \varnothing$.
13)The next stage is the construction of $I_4 = F_{P_3}(I_3)$ which is equal to $\varnothing$.
14)P_4 is now set to $trans(P_3, \varnothing, pos)$ 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 [20], [3] 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, $\mathbf{glo\_true}_n = \mathrm{lfp}(F_P^2) = \mathbf{wfs\_true}(P)$ and that $\mathbf{glo\_false}_{n+1} = \mathrm{gfp}(F_P^2) = \overline{\mathbf{wfs\_false}(P)}$. The equality $\mathrm{lfp}(F_P^2) = \mathbf{wfs\_true}(P)$ has been proved in [3], as has the equality $\mathrm{gfp}(F_P^2) = \overline{\mathbf{wfs\_false}(P)}$. What remain to be established are the equalities $\mathbf{glo\_true}_n = \mathrm{lfp}(F_P^2)$ and $\mathbf{glo\_true}_{n+1} = \mathrm{gfp}(F_P^2)$. 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 **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. **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 [20], Baral and Subrahmanian [3]) *Suppose P is a logic program. Then* **wfs_true(*P*)** = **glo_true(*P*)** *and* **wfs_false(*P*)** = **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 (Experiments V.A.1 and V.A.3). Instead, it is computationally faster, in practice, to run the MI-module first on program P, and generate the sets **mi_true(*P*)** and **mi_false(*P*)** and the modified program **mi_target(*P*)**. **mi_target(*P*)** is usually much "smaller" than P (Experiment V.A.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.

C. 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*)** ∪ **glo_true(mi_target(*P*))** of "true" atoms, and **mi_false(*P*)** ∪ **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*)** ∪ **glo_true(mi_target(*P*))**
2) **wfs_false(*P*)** = **mi_false(*P*)** ∪ **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).

IV. COMPUTATION OF STABLE SEMANTICS

It is well-known [20], [3] 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*)** ⊆ 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.

A. 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 (Fig. 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, ..., P_n specified in Definition 7. (As we are only dealing with deductive databases, there must exist an integer n such that $P_n = P_{n+1}$).
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 $\{$**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 4. Suppose $q \equiv$ **glo_simp**(P) is the program: $\{a \leftarrow \neg b;\ b \leftarrow \neg a;\ c \leftarrow a;\ c \leftarrow b\}$. 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 $\varnothing$ and we have a list L containing one node—the four-tuple Q = (**glo_simp**(P), $\varnothing$, $\varnothing$, $\{a, b, c\}$) 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 IV.C. Fig. 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: $\{a \leftarrow \neg b;\ b \leftarrow\ ;\ c \leftarrow false;\ c \leftarrow b\}$. 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 ($\varnothing$ 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^-: First of all, a must be in F^- because we are branching on the assumption that a is false. In addition, F^- includes all the false atoms from the parent node ($\varnothing$ in this case) plus the atoms determined to be false in the well-founded semantics of the new program (also $\varnothing$ in this case). Hence, the set of false atoms in the new node is $\{a\}$.
- U^-: The set of unknown atoms in the new node is $\varnothing$ (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 $T^- \cap F^- = \varnothing$, 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 $\{a \leftarrow \neg b;\ b \leftarrow false;\ c \leftarrow\ ;\ c \leftarrow b\}$. 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, $T^+ \cap F^+ = \varnothing$ 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 $\{b, c\}$. □

B. 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 Fig. 2. As can be observed by a cursory glance at the algorithm of Fig. 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 6.

DEFINITION 8. *Suppose q is a logic program, and A is a ground atom. The result of modifying q w.r.t. $\neg A$, denoted* **CH**(q, $\neg A$), *is the logic program obtained as follows:*

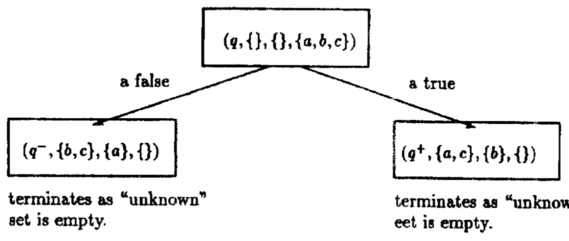
1) *If A occurs in the body of a clause in q, then A is replaced by the atom false.*
2) *If $\neg A$ occurs in the body of a clause in q, then that occurrence of $\neg A$ is deleted.*

The result of modifying q w.r.t. A, denoted **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.*

```
$L = \langle (P,\emptyset,\emptyset,B_P) \rangle$; (* $B_P$ 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.



(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.

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 **CH**(–, –) is that the latter never causes any clause to be deleted and never affects the head of any clause.

DEFINITION 9. *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 10. *Suppose P is a logic program. Let B_P be the Herbrand Base of P. Furthermore, suppose the cardinality of B_P is n 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) $q^- =$ **CH**$(q, \neg a_i)$
    b) $T^- = T \cup$ **wfs_true(glo_simp(CH**$(q, \neg a_i)$**))**
    c) $F^- = F \cup \{a_i\} \cup$ **wfs_false(glo_simp(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) $q^+ =$ **CH**(q, a_i)
    b) $T^+ = T \cup \{a_i\} \cup$ **wfs_true(glo_simp(CH**(q, a_i)**))**
    c) $F^+ = F \cup$ **wfs_false(glo_simp(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 Fig. 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 **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(**glo_simp**(P)) as opposed to $ACT(P)$. This reduces the number of nodes from $(2^{(\|B_P\|+1)} - 1)$ to $(2^{\|B_{\text{glo\_simp}(P)}\|+1} - 1)$. In practice the size of the program **glo_simp**(P) as compared to the size of P is very small indeed.
2) Second, many branches in ACT(**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) Fourth, 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 $T_1 \subseteq T_2 \subseteq$. Furthermore, Marek and Truszczynski [17] 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 11. *Suppose P is a logic program. Let* **LEAF(glo_simp(*P*))** $=$ *{T | there exists a leaf node in PRUNE_ACT(***glo_simp***(P)) having, as its label,* $(q, T, F, \emptyset)$ *such that* $T \cap F = \emptyset$}.

Let **MIN_LEAF(glo_simp(*P*))** *be the set of all $\subseteq$-minimal elements of* **LEAF(glo_simp(*P*))**.

In other words, **LEAF(glo_simp(*P*))** is simply the set of all T-components of the labels of consistent leaves of *PRUNE_ACT*(**glo_simp**(*P*)). Similarly, **MIN_LEAF (glo_simp(*P*))** is the set of *minimal* elements of **LEAF(GLO_SIMP(*P*))**. The following example shows the tree *PRUNE_ACT*(*P*), and how stable models may be generated.

EXAMPLE 5. Consider a program P containing the following clauses:

$$a \leftarrow \neg b \qquad b \leftarrow \neg a$$
$$c \leftarrow a \qquad c \leftarrow b$$

Fig. 4 shows the tree *PRUNE_ACT*(*P*) corresponding to this program P. Note that in this case, $P =$ **glo_simp**(*P*).

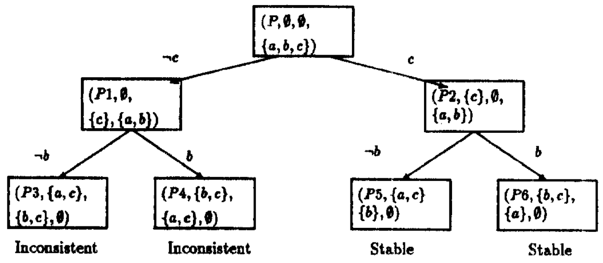


Fig 4. The (pruned) tree ACT(P) for Example 6 using selection ordering c,b,a.

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, Fig. 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 IV.C provides an outline of how to make such selections a priori. □

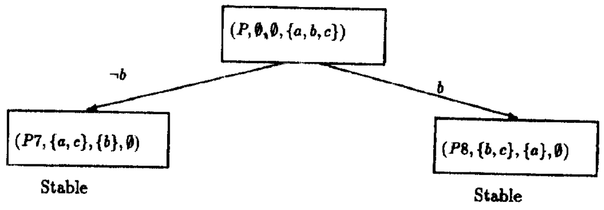


Fig. 5. The (pruned) treee ACT(P) for Example 6 using selection ordering b,a,c.

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 4. *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*) *with A in the head 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* **u** *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**(**CH**(q, $\pm a$)) rather than just **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 5. *The branch and bound algorithm generates the nodes of PRUNE_ACT(P) in pre-order (Knuth* [13]*).* □

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 6. *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*)**. □

Before proceeding to discuss intelligent branching strategies, we observe that sometimes, we may be interested in truth in *some* stable model of *P*. Stable models reflect multiple possible ways of completing an "incomplete" description of the world. Any one of these may be the "right" one, but based on the available intuition, we do not know which. To determine truth of a query in some stable model of *P*, the branch and bound method can be modified as follows: as soon as the first stable model M_1 is discovered by the branch and bound algorithm, write down the tuples $\{p(1, \vec{t}) \mid p(\vec{t}) \in M_1\}$. Basically, the tuple $p(i, \vec{t})$ says that the ground atom $p(\vec{t})$ is true in the *i*th stable model of *P*. When wanting to know if the query $(\exists x_1, \ldots x_k)(p_1, (\vec{t}_1) \& \ldots \& p_m(\vec{t}_m))$ is true in some stable model of *P*, the above set of tuples can be queried as: $(\exists i)(\exists x_1, \ldots x_k)(p_1 i, (\vec{t}_1) \& \ldots \& p_m(i, \vec{t}_m))$. Alternatively, should we so desire, the branch and bound algorithm can be easily modified to terminate as soon as one stable model has been discovered. Whether this non-deterministic way of selecting a stable model (and committing to it) is appropriate would depend on the application.

C. Intelligent Branching

As described earlier (Example 5), 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 [1] for this purpose.

DEFINITION 12. *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 13. *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 [1] 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 [1] 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 $A\}$. (The equivalence classes correspond to the strongly connected components of the dependency graph.)
- 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 6. Consider the program of Example 5. Here, the equivalence classes are: $\|a\| = \{a, b\}$ and $\|c\| = \{c\}$. 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 7. Continuing with the program of Example 5 and Example 6, we note that $\mathbf{E}_0 = \{\{a, b\}$ and $\mathbf{E}_1 = \{\{c\}\}$.

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 $\mathbf{E}_0$, $\mathbf{E}_1$, ... hierarchy. Thus, in the case of the root of the tree associated with the program Example 5 and Example 6, we would choose to branch on either *a* or *b* instead of choosing to branch on *c*. This leads to a "shorter" tree.

Experiment V.A.5 reports on some experiments that we have run to determine the utility of intelligent branching.

An alternative formulation of the intelligent branching strategy is to partition the logic program being processed by the branch and bound module according to the equivalence classes generated by the ≤-ordering. The ≤-minimal components' stable models can then be computed first; stable models of components that are not ≤-minimal may be done once all the stable models of all (programs corresponding to) components "strictly below" have been computed. This is equivalent to the intelligent branching strategy.

D. Partial Instantiation: The Non-Ground Case

A valid critique of the work presented this far in this paper is that it applies to *ground* programs. This is a drawback because the ground instantiation of a logic program is significantly "larger" than the original program. In [10], we have developed techniques that, given a definite (i.e., negation-free) logic program P, and any method for computing the semantics of a *propositional* (i.e., grounded out) logic program, will show how to instantiate P on an "instantiate by need" basis so that the set of atomic logical consequences of the non-ground program P can be computed.

Basically, this partial instantiation method for evaluating logic programs proceeds as follows—first, a (non-ground) logic program P is treated as if it were a propositional logic program $P^{\star}$(i.e., an atom A occurring in P is considered to be a proposition $\mathbf{p}_A$). Program $P^{\star}$ may then be evaluated using *any known mechanism for evaluating propositional logic programs* [4], [5]. Assignments of true/false to different propositions $\mathbf{p}_A$ and $\mathbf{p}_B$ in $P^{\star}$ may lead to "conflicts" when A and B are unifiable, but $\mathbf{p}_A$ and $\mathbf{p}_B$ are assigned different truth values. If there are no such conflicts, then we are done. When such conflicts are present, then [10] articulates a precise strategy for removing such conflicts and shows that this strategy of

Evaluate Propositional Program $\rightarrow$ Identify Conflicts $\rightarrow$ Partially Instantiate

yields a soundness and completeness theorem for the computation of answer substitutions [16].

The extension of the partial instantiation strategy for definite programs to apply to well-founded and stable models is being studied in two separate efforts [11], [9]. As in [10], both these efforts assume the existence of two methods, $\mathbf{M}_w$ and $\mathbf{M}_s$ that, given any ground logic program will compute the well-founded semantics and the set of stable models, respectively of the ground program. *The methods described in the preceding sections perform these computations in the ground case.* Subsequently, "conflicts" will be identified and partial instantiation will be used to remove these conflicts. Neither of the two papers [11], [9] in preparation show how to compute the well-founded (or stable models) semantics of propositional programs—rather, they show how to use a propositional stable/well-founded semantics computation strategy to generate a partial instantiation strategy that will instantiate non-ground programs on a "need-to-instantiate" basis. *Consequently, the methods developed in this paper can be used in conjunction with the partial instantiation strategies being developed in* [9], [11] *to yield computational paradigms for nonmonotonic logic programming semantics in the non-ground case.*

We give below, an outline of how the partial instantiation strategy can be used to compute the well-founded semantics. The detailed description of the scheme and its soundness and completeness results are contained in [11].

EXAMPLE 8. Let P be the (non-ground) logic program below.

$$p(X_1,Y_1)\leftarrow \neg q(X_1,Y_1). \qquad q(X_2,Y_2)\leftarrow \neg p(X_2,Y_2).$$
$$r(a)\leftarrow. \qquad r(b)\leftarrow.$$
$$q(a,a)\leftarrow.$$

According to the well-founded semantics, the ground atoms $r(a)$, $r(b)$, $q(a, a)$ are true, the atom $p(a, a)$ is false, and all other ground atoms are "unknown."

The partial instantiation strategy works by considering all the atoms occurring in P to be distinct propositional symbols – thus, for instance, $p(X_1, Y_1)$ and $q(X_1, Y_1)$ are considered to be distinct propositional symbols. The well-founded semantics of this "propositional" version of P says that $r(a)$, $r(b)$, $q(a, a)$ are true, and $p(X_1, Y_1)$, $q(X_1, Y_1)$, $p(X_2, Y_2)$, $q(X_2, Y_2)$ are unknown. Nothing is assigned false. At this stage, we notice that there is a conflict—$q(a, a)$ is unifiable with both $q(X_1, Y_1)$, $q(X_2, Y_2)$ via unifiers $\theta_1 = \{X_1 = a, Y_1 = a\}$ and $\theta_2 = \{X_2 = a, Y_2 = a\}$, respectively. The conflict exists because $q(a, a)$ is "true" according to the well-founded semantics, but $q(X_1, Y_1)$, $q(X_2, Y_2)$ are assigned the truth value "unknown." We instantiate the clauses in P by θ_1 and θ_2, respectively, leading to two new clauses: $p(a, a)\leftarrow \neg q(a, a)$ and $q(a, a)\leftarrow \neg p(a, a)$. These are then added back into P and the process repeated. At this stage, $r(a)$, $r(b)$, $q(a, a)$ are assigned "true" by the propositional WFS computation process, $p(a, a)$ is assigned "false" and all other atoms are assigned "unknown." The only conflicts that occur now generate the same substitutions θ_1 and θ_2, that we saw before, and hence, we can terminate.

V. IMPLEMENTATION AND EXPERIMENTATION

All the components of Fig. 1 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 Section III.A, the transformation strategy, the pruning oscillation described in Section III.B, the branch and bound procedure of Section IV, and the intelligent branching strategy of Section IV.C.

A. 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., [20]). 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[5] 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. *All times are reported in milliseconds.*

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 [20]. Other experiments and examples are described in the longer technical

5. The Unix utility program *profile* is used to record computation times.

report [19]. These results are representative of our other results. The "win-move" example consists of the single rule *win*(*X*) ← *move*(*X*, *Y*) & ¬*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 eight 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.

A.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 [20]. 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 Fig. 1. The naive alternating approach would run the entire program through the GLO module alone.

Fig. 6 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.

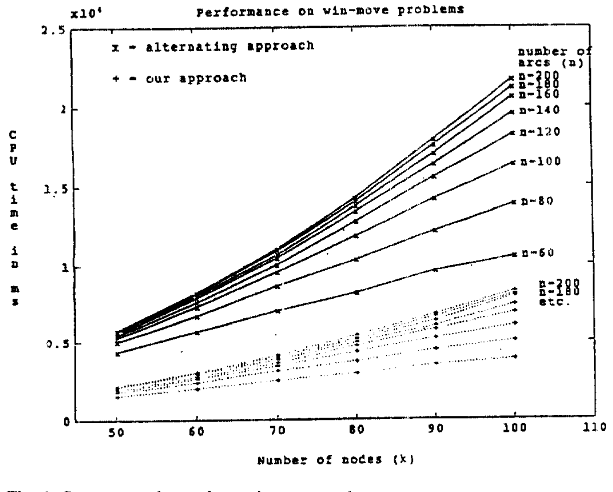


Fig. 6. Our approach vs. alternating approach.

Two conclusions may be drawn from the graph of Fig. 6.

- 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 Fig. 6 which compares the time taken by our procedure with the time taken by the alternating procedure.

A.2. Size of **mi_target**(*P*) *compared to the Size of P*

Fig. 7 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 Fig. 7 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.

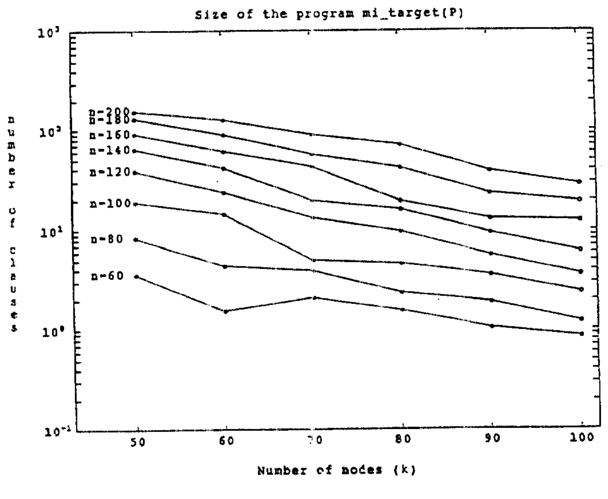


Fig. 7. Growth in size of **Mi_target**(*P*).

A.3. Effect of Pruning Oscillation

Finally, we ran experiments to verify the effectiveness of pruning oscillations. Fig. 8 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.

Fig. 8 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 **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.

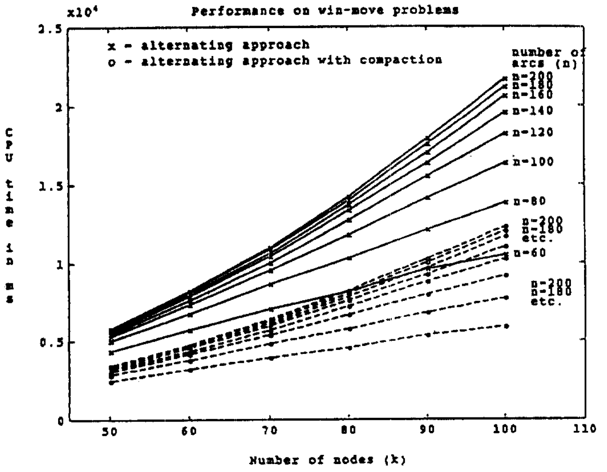


Fig. 8. Effect of compaction.

A.4. Stable Model Computation

Fig. 9 shows the total time taken to compute all the stable models of a logic program using our approach. (Again, as before, the "win-move" example is being used here.) 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.

A.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.

$z1(X) \leftarrow v1(X), w1(X).$	$z2(X) \leftarrow v1(X), w2(X).$
$z3(X) \leftarrow v2(X), w1(X).$	$z4(X) \leftarrow v2(X), w2(X).$
$v1(X) \leftarrow s(X).$	$v2(X) \leftarrow t(X).$
$w1(X) \leftarrow p(X).$	$w2(X) \leftarrow q(X).$
$t(X) \leftarrow \neg s(X).$	$s(X) \leftarrow \neg t(X).$
$p(X) \leftarrow \neg q(X).$	$q(X) \leftarrow \neg p(X).$

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. This program has 4^n stable models where n is the number of constants in our language. Table I 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 $\mathbf{E}_0$, $\mathbf{E}_1$, ... described in Section IV.C.

TABLE I
NAIVE VS. INTELLIGENT BRANCH AND BOUND

Number of Constants	1	2	3	4	5
Naive Branch and Bound	101	637	3,165	16,744	129,186
Intelligent Branch and Bound	43	262	1,413	9,431	95,766
Number of Stable Models	4	16	64	256	1,024

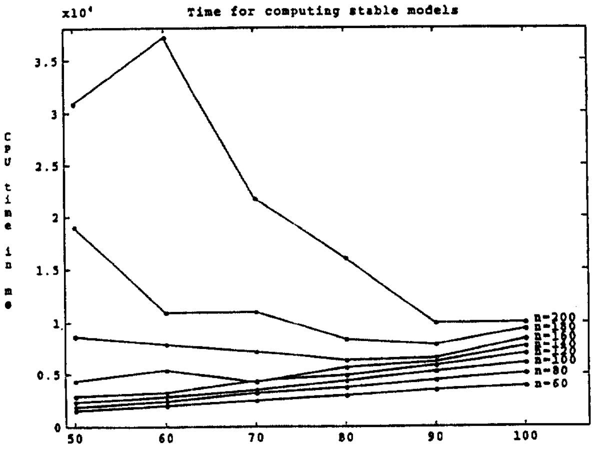


Fig. 9. Time for stable model computation.

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

$s(X) \leftarrow p(X), q(X).$	$s(X) \leftarrow p(X), r(X).$
$s(X) \leftarrow q(X), r(X).$	$p(X) \leftarrow \neg (X).$
$q(X) \leftarrow \neg (X).$	$r(X) \leftarrow \neg (X).$

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*). Table II 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.

TABLE II
NAIVE VS. INTELLIGENT BRANCH AND BOUND AS CONSTANTS INCREASE

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	1,117	755
40	1,379	972
45	1,691	1,203
50	2,008	1,475

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

B. 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 databases because instead of inserting all tuples, one by one, into a relation, 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 say a real-time control system, then 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 Table III.

TABLE III
PROS AND CONS OF DIFFERENT DATA MODELS

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

VI. 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. [12], Chen and Warren [6], Warren [22], and by Leone and Rullo [14]. Kemp et al. show how, given a query Q to a logic program P, and a sideways information passing strategy[6] S, it is possible to create a new program *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.[12] show how the query Q may be answered w.r.t. the new program *Magic*(P, S, Q). Warren [22] 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 —his avoids redundant computation. Chen and Warren [6] extend the work in [22] and develop a sound and complete technique for computing WFS called XOLDTNF-resolution. Leone and Rullos's technique is similar to the above techniques in spirit, and deals with safe computations in a datalog language containing well-founded negation. They do not present an implementation, however.

Computation of the set of stable models has also been studied by Sacca and Zaniolo [18]. Their method is based on a backtracking technique which assumes an undefined atom to be false and then continues the computation on this assumption until it computes a stable model or discovers a contradiction in which case it backtracks. The branch and bound technique developed here may be viewed as an improvement of the Sacca-Zaniolo technique—especially as various pruning (i.e., bounding) techniques we use speed up the computation. A new and important feature of our work is that our computations are based on a prior computation of the well-founded model which the backtracking method does not do. Last, but not least, Leone et al. [15] study computation of nonmonotonic negation in logic programming. In contrast to our work, their work makes use of choice constructs in its computation.

The main difference between our work and that of Warren

6. See [12] for an explanation of this expression.

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. 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 U.S. 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 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. [12] and Warren [22] are perhaps more appropriate.

VII. Concluding Remarks

Though nonmonotonic 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 nonmonotonic semantics. In this paper, we take a first step towards developing a compiled approach for computing the

- well-founded model of nonmonotonic deductive databases and
- the set of stable models of nonmonotonic 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 Fig. 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 re-compute 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 (respectively, set of stable models). We plan to study these topics.

Acknowledgments

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

This research was supported by the U.S. Army Research Office under Grant Number DAAL-03-92-G-0225, by National Science Foundation grants IRI-9109755, IRI-9357756, IRI-8907890 as well as National Science Foundation Grant NSFD CDR-88003012 to the Systems Research Center, by ARPA/Rome Labs contract F30602-93-C-0241 (ARPA Order Nr. A716).

References

[1] K.R. Apt, H. Blair, and A. Walker, "Towards a theory of declarative knowledge," J. Minker, ed., *Foundations of Deductive Databases and Logic Programming*, pp. 89–148, Morgan Kaufmann, 1988.

[2] C. Baral and V.S. Subrahmanian, "Stable and extension class theory for logic programs and default logics," *J. Automated Reasoning*, vol. 8, pp. 345–366, 1992.

[3] C. Baral and V.S. Subrahmanian, "Dualities between alternative semantics for logic programming and nonmonotonic reasoning," *J. Automated Reasoning*, vol. 10, pp. 399–420, 1993.

[4] C. Bell, A. Nerode, R. Ng, and V.S. Subrahmanian, "Implementing deductive databases by linear programming," accepted for publication in *ACM Trans. Database Systems*.

[5] C. Bell, A. Nerode, R. Ng, and V.S. Subrahmanian, "Computation and implementation of nonmonotonic deductive databases," *J. ACM*, vol. 41, no. 6, pp. 1,178-1215, Nov. 1994.

[6] W. Chen and D.S. Warren, "A practical approach to computing the well-founded semantics," *Proc. 1992 Int'l Conf. Logic Programming*, Nov. 1992.

[7] M.C. Fitting, "A Kripke-Kleene semantics for logic programming," *J. Logic Programming*, vol. 4, pp. 295–312, 1985.

[8] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," *Proc. Fifth Int'l Conf. and Symp. on Logic Programming*, R.A. Kowalski and K.A. Bowen, eds., pp. 1,070–1,080, 1988.

[9] G. Gottlob, S. Marcus, A. Nerode, and V.S. Subrahmanian, "Non-ground stable and well-founded demantics," manuscript in preparation, 1993.

[10] V. Kagan, A. Nerode, and V.S. Subrahmanian, "Computing definite logic programs by partial instantiation," *Annals of Pure and Applied Logic*, vol. 67, pp. 161–182, 1993.

[11] V. Kagan, A. Nerode, and V.S. Subrahmanian, "Computing minimal models by partial instantiation," 1993, accepted for publication in *Theoretical Computer Science*.

[12] D. Kemp, P.J. Stuckey, and D. Srivastava, "Magic sets and bottom-up evaluation of well-founded models," *Proc. 1991 Int'l Logic Programming Symp.*, V. Saraswat and K. Ueda, eds., pp. 337–351, MIT Press, 1991.

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

[14] N. Leone and P. Rullo, "Safe computation of the well-founded semantics of DATALOG queries," *Information Systems*, vol. 17, no. 1, 1992.

[15] N. Leone, M. Romeo, P. Rullo, and D. Sacca, "Effective implementation of negation in database logic query languages," *LOGIDATA+: Deductive Databases with Complex Objects*, LNCS vol. 701, pp. 159–175, Springer, 1993.

[16] J.W. Lloyd, *Foundations of Logic Programming*, Springer, 1987.

[17] W. Marek and M. Truszczynski, "Stable semantics for logic programs and default theories," *Proc. 1989 North Am. Conf. Logic Programming*, E. Lusk and R. Overbeek, eds., pp. 243–256, MIT Press, 1989.

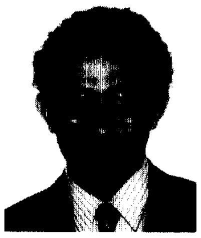
[18] D. Sacca and C. Zaniolo, "Stable models and nondeterminism in logic programs with negation," *Proc. 1990 ACM Symp. Principles of Database Systems*.

[19] V.S. Subrahmanian, D.S. Nau, and C. Vago, "WFS + branch and bound = stable models," Tech. Report CS-TR-2935, July 1992, Univ. of Maryland. Tech. report version of this paper.

[20] A. van Gelder, "The alternating fixpoint of logic programs with negation," *Proc. Eighth ACM Symp. Principles of Database Systems*, pp. 1–10, 1989.

[21] A. van Gelder, K. Ross, and J. Schlipf, "Unfounded sets and well-founded semantics for general logic programs," *Proc. Seventh Symp. Principles of Database Systems*, pp. 221-230, 1988.

[22] D.S. Warren, "Computing the well-founded semantics of logic programs," SUNY Stonybrook Tech. Report TR 91-12, June 1991.



V.S. Subrahmanian received the PhD degree in computer science from Syracuse University in 1989. Since then, he has been assistant professor of computer science at the University of Maryland, College Park.

Dr. Subrahmanian received a National Science Foundation Young Investigator award in 1993. He has worked extensively on the theory and implementation of logic programming and deductive database systems. In particular, he has developed theories of probabilistic logic programming, nonmonotonic logic programming, and frameworks for hybrid reasoning using distributed hereogeneous databases. Many of these theories have been implemented in prototype experimental systems and are being used in a variety of civilian and military applications. Dr. Subrahmanian has published more than 30 papers and is principal investigator on research grants from NSF, ARO, AFOSR, and ARPA.

Dana Nau received his PhD in computer science in 1979 from Duke University, where he was a U.S. National Science Foundation graduate fellow.

Dr. Nau is a professor at the University of Maryland in the Department of Computer Science and the Institute for Systems Research. He received a National Science Foundation Presidential Young Investigator award (1984-1989) and the ISR Outstanding Systems Engineering Faculty award (1993-1994), and has received various other awards. His current research interests include AI planning and searching techniques, and computer-integrated design and manufacturing. Dr. Nau has published nearly 150 technical papers.

Carlo Vago received his master's degree in computer science from the University of Milan. He is with Olivetti in Italy.