Computation of Stable Models and its Integration with Logical Query Processing

Weidong Chen*
Computer Science and Engineering
Southern Methodist University
Dallas, TX 75275-0122

David Scott Warren †
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400

November 22, 1993

Abstract

The well-founded semantics and the stable model semantics capture intuitions of the skeptical and credulous semantics in nonmonotonic reasoning, respectively. They represent the two dominant proposals for the declarative semantics of deductive databases and logic programs. However, neither semantics seems to be suitable for all applications. We have developed an efficient implementation of goal-oriented effective query evaluation under the well-founded semantics. It produces instances of a query that are true or false, as well as a residual program for undefined instances of the query. This paper presents a simple method of stable model computation that can be applied to the residual program of a query to derive answers with respect to stable models. The method incorporates both forward and backward chaining to propagate the assumed truth values of ground atoms, and derives multiple stable models through backtracking. Users are able to request that only stable models satisfying certain conditions be computed. A prototype has been developed that provides integrated query evaluation under the well-founded semantics, the stable models, and ordinary Prolog execution. We describe the user interface of the prototype and present some experimental results.

Index Terms: Alternating Fixpoint Logic, Deductive Database, Logic Programming, Nonmonotonic Reasoning, Logical Query Evaluation, Stable Model Semantics, Well-Founded Semantics.

1 Introduction

Significant progress has been made in understanding the declarative semantics of deductive databases and logic programs with negation. Two dominant proposals are the well-founded semantics [31] and the stable model semantics [13]. For a normal logic program, where the body of each rule is a conjunction of literals, its well-founded semantics is characterized by a unique

*Supported in part by the National Science Foundation under Grant No. IRI-9212074.
†Supported in part by the National Science Foundation under Grant No. CCR-9102159.

three-valued model, called the *well-founded partial model*. It is well defined for all normal logic programs. However, the well-founded semantics is inadequate in dealing with reasoning by cases or multiple alternative situations.

Example 1.1 Consider the following program:

covered(Course) :- teach(Faculty, Course).
teach(john, cse5381) :- ∼teach(mary, cse5381).
teach(mary, cse5381) :- ∼teach(john, cse5381).

Its well-founded partial model is such that every ground atom is undefined, thus providing no useful information about the scenario being described. □

The well-founded semantics of normal logic programs has been extended by Van Gelder [30] to *general* logic programs, where the body of each rule may be an arbitrary first-order formula. The resulting semantics is called the alternating fixpoint logic [30].

The notion of stable models [13] originated from the work on autoepistemic logic [12]. Each stable model represents a set of beliefs that can be derived from itself. In Example 1.1, there are two stable models, one in which John teaches CSE 5381 and the other in which Mary teaches CSE 5381. In either case, CSE 5381 is covered. Unlike the well-founded partial model, stable models may not exist for a given program, e.g., p :- ∼p, and even if they exist, they may not be unique.

Recent research shows that the well-founded partial model and (two-valued) stable models are two extreme cases of three-valued stable models [9, 22, 26]. The well-founded partial model coincides with the smallest three-valued stable model. It corresponds to the *skeptical* semantics that includes only beliefs that are true in all possible situations. On the other hand, the notion of stable models captures the *credulous* semantics that concludes as many beliefs as possible from a normal logic program.

Separate techniques have been developed for query evaluation under the well-founded semantics and for computing stable models. The well-founded semantics has a constructive definition based upon a least fixpoint construction. For function-free programs, it has a polynomial time data complexity [31]. In addition to direct extensions of SLDNF resolution [21, 24], various mechanisms of positive and negative loop handling have been incorporated for effective query evaluation under the well-founded semantics [2, 3, 6, 7, 23, 28]. However, not all of them can be extended directly for stable model computation.

The definition of stable models requires guessing an interpretation and then verifying if it is a stable model. In fact, the problem of the existence of a stable model of a logic program is NP-complete [17]. There have been several proposals for stable model computation [10, 19, 20, 26]. Two aspects are common. One is that only two-valued stable models are computed. This is not surprising. Two-valued stable models represent the credulous semantics that does not allow any incomplete information, and they have a smaller search space from a computational point of view. The other common aspect is that only ground programs are processed. This, however, is a severe restriction in practice since almost all rules have variables.

We have developed a prototype system, called *SLG*, for logical query answering. SLG supports goal-oriented query evaluation under the well-founded semantics of normal logic programs, or more generally, the alternating fixpoint logic of general logic programs. The latter is an important

extension since a standard translation of general logic programs into normal logic programs does not always preserve the semantics. In either case, SLG has a polynomial time data complexity for function-free programs. If a query has undefined instances, SLG produces a residual program besides true and false instances of the query. The residual program can be further processed to compute its (two-valued) stable models. SLG is available by anonymous ftp from seas.smu.edu or cs.sunysb.edu.

By applying stable model computation to only the residual program of a query, SLG has two advantages. First, answers of a query that are true in the well-founded semantics can always be derived within polynomial time in the size of a database. They can be computed even more efficiently if a program satisfies certain properties such as stratification. More importantly, non-ground programs and queries can be handled. Second, the residual program of a query is often much smaller than the original program. The approach in SLG restricts the potentially expensive (two-valued) stable model computation to a small portion of the entire program. Furthermore, three-valued stable models are partially supported since the original program may not have a (two-valued) stable model even though the portion of the program that is relevant to a query has (two-valued) stable models.

The main contributions of this paper are threefold. First, we describe a simple *assume-and-reduce* algorithm for computing (two-valued) stable models of a finite ground program. It assumes the truth values of only those ground atoms whose negative counterparts occur in a program. The search space of stable models is further reduced by forward and backward propagation of the assumed truth values of ground atoms and by reduction of the program. Second, we show how to integrate query evaluation under the well-founded semantics with the computation of stable models. Two aspects are noteworthy. One is that handling negative loops by *delaying* not only avoids redundant derivations, but also leads to the residual program needed for stable model computation later. The other is that the forward chaining network set up for simplifying delayed literals in the derivation of the well-founded semantics is directly useful for stable model computation. Finally, due to the multitude of stable models, it is not clear what answers should be computed for a query. We introduce a versatile user interface for query evaluation with respect to stable models.

Section 2 describes a method of stable model computation. Section 3 presents its integration with query evaluation of the well-founded semantics. Section 4 contains some examples and performance analysis. Section 5 compares with related work.

2 Computation of Stable Models

This section reviews the terminologies of logic programs [16] and the notion of (two-valued) stable models by Gelfond and Lifschitz [13]. An *assume-and-reduce* algorithm is described for computing (two-valued) stable models of finite ground programs.

2.1 Definition of Stable Models

An *atom* is of the form $p(t_1, ..., t_n)$, where p is an n-ary predicate symbol and $t_1, ..., t_n$ are terms. For an atom A, A is a *positive literal* and $\sim A$ is a *negative literal*, and they are *complements* of

each other. A *clause* is of the form

$$A \text{ :- } L_1, ..., L_n.$$

where A, the *head* of the clause, is an atom, and $L_1, ..., L_n (n \geq 0)$, the *body* of the clause, are literals. A *definite clause* is a clause that has no negative literals in its body. A *(definite) program* is a set of (definite) clauses. A *ground* atom (literal, clause, program) is one that is variable-free.

The *Herbrand universe* of a program P is the set of all ground terms that may be constructed from the constants and function symbols appearing in P. An arbitrary constant is added if no constant occurs in P. The *Herbrand base* of P, denoted by $\mathcal{B}_P$, is the set of all ground atoms with predicates occurring in P whose arguments are in the Herbrand universe of P. The *Herbrand instantiation* of P is the (possibly infinite) set of all ground clauses obtained by substituting terms in the Herbrand universe for variables in clauses in P.

A set I of ground literals is *consistent* if for no ground atom A, both A and $\sim A$ are in I. We denote by $Pos(I)$ the set of positive literals in I and $Neg(I)$ the set of ground atoms whose complements are in I. A *partial interpretation* (or just *interpretation*) I is a consistent set of ground literals. A *total* interpretation is an interpretation where $Pos(I) \cup Neg(I) = \mathcal{B}_P$. A ground literal L is true in an interpretation I if and only if $L \in I$.

Definition 2.1 ([13]) Let P be a program and I be an interpretation. The *Gelfond-Lifschitz transformation* of P with respect to I, denoted by P_I, is the program obtained from the Herbrand instantiation of P by deleting

- each clause that has a negative literal $\sim B$ in its body with $B \in I$, and
- all negative literals $\sim B$ in the bodies of the remaining clauses with $\sim B \in I$.

If I is a total interpretation, the resulting program P_I is a definite program. According to [1], every definite program P has a unique minimal model, which we will denote by $\mathcal{M}(P)$.

Definition 2.2 ([13]) Let I be a total interpretation, and P be a logic program. I is a (two-valued) *stable model* of P if and only if I coincides with $\mathcal{M}(P_I)$.

2.2 Derivation of Stable Models

For the derivation of stable models, we consider finite ground programs. The application for goal-oriented query evaluation of non-ground logic programs will be discussed in Section 3.

Let P be a finite ground program. We can restrict interpretations to ground atoms that occur in P as every ground atom that does not occur in P is definitely false. According to Definition 2.2, stable models of P can be computed by enumerating every possible total interpretation I and checking if I coincides with the unique minimal model $\mathcal{M}(P_I)$ of P_I. The number of possible total interpretations is obviously exponential in the size of the Herbrand base. Fortunately there are often mutual dependencies among ground atoms in a program, which can be used to reduce the search space and to speed up the computation of stable models substantially.

2.2.1 Assuming Negative Literals Only

Our first observation is that we have to guess only the truth values of ground atoms A such that $\sim A$ occurs in P.

Example 2.1 Consider the following ground program:

```
covered(cse5381) :- teach(john,cse5381).
covered(cse5381) :- teach(mary,cse5381).
teach(john,cse5381) :- ~teach(mary,cse5381).
teach(mary,cse5381) :- ~teach(john,cse5381).
```

Two negative literals, ~teach(mary,cse5381) and ~teach(john,cse5381), occur in the program. As soon as their truth values are determined, the truth value of covered(cse5381) can be derived. There is no need to make assumptions about the truth value of covered(cse5381). This reduces the search space for stable models. □

Let $\mathcal{N}(P)$ denote the set of ground atoms A such that $\sim A$ occurs in P.

Lemma 2.1 Let P be a ground logic program, and I be a total interpretation. Then I is a stable model of P if and only if for some interpretation J, where $Pos(J) \cup Neg(J) = \mathcal{N}(P)$, $J \subseteq I$, and I coincides with $\mathcal{M}(P_J)$.

Proof: Let I be a total interpretation and let J be the restriction of I to $\mathcal{N}(P)$, i.e., $Pos(J) = Pos(I) \cap \mathcal{N}(P)$ and $Neg(J) = Neg(I) \cap \mathcal{N}(P)$. Then $P_I = P_J$, and the lemma follows from the definition of stable models. □

2.2.2 Propagation of Assumed Truth Values

The truth values of ground atoms in $\mathcal{N}(P)$ are not independent of each other either. In Example 2.1, if we assume that teach(mary,cse5381) is true, the clause for teach(john,cse5381) can be deleted since its body is false according to the assumption. Hence teach(john,cse5381) must be false. Similarly, if teach(mary,cse5381) is assumed to be false, teach(john,cse5381) can be derived to be true. Therefore it is not necessary to enumerate all the four possible truth assignments for teach(john,cse5381) and teach(mary,cse5381).

Our second observation is that the assumed truth value of a ground atom should be used to simplify the program being considered in order to reduce the search space for truth assignments of ground atoms in $\mathcal{N}(P)$ that lead to stable models.

Let P be a finite ground program, and $\sim A$ be a ground literal that occurs in P. There are two possible choices: either A is true or $\sim A$ is true. We derive two programs from P, namely P_A where P is simplified based upon the assumption that A is true and $P_{\sim A}$ where P is simplified based upon the assumption that $\sim A$ is true. The objective is to derive stable models of P from those of P_A and $P_{\sim A}$.

The simplification of a program P based upon the assumed truth value of a literal should be done in such a way that avoids the generation of models that are supported but not stable.

Example 2.2 Let P be the following program:

q :- ∼p.
p :- p.

The only stable model for P, which is also the perfect model in this case, is that q is true and p is false. Suppose that p is assumed to be true. Based upon the assumption that p is true, we can delete the clause for q since $\sim p$ is false according to the assumption. However, the assumption that p is true cannot be used to simplify away the positive occurrence of p in the body of the clause for p. Otherwise, we would derive a model, $\{p, \sim q\}$, that is not a stable model. Therefore the simplified program, P_p, should contain one clause, p :- p. In this particular case, assuming that p is true does not lead to any stable model since p is false in the stable model of P_p, which is inconsistent with the assumption.

On the other hand, suppose that ∼p is assumed to be true. The simplified program, $P_{\sim p}$, should contain one fact, q. That is, the assumption that ∼p is true can be used to delete the clause for p since its body is false according to the assumption. □

Example 2.2 indicates that the assumed truth value of a ground atom can be used to delete every clause whose body is false and every negative body literal that is true according to the assumption, but it cannot be used to delete a *positive* body literal that is assumed to be true.

Definition 2.3 Let P be a ground program, and L be a ground literal. Then P_L is the program obtained from P by deleting

- every clause in P whose body contains the complement of L, and
- every occurrence of L in P if L is a negative literal.

Lemma 2.2 Let P be a ground program, A be a ground atom, and I be a total interpretation. Then I is a stable model of P if and only if either A is true in I and I is a stable model of P_A, or $\sim A$ is true in I and I is a stable model of $P_{\sim A}$.

Proof: Suppose that A is true in I. Then $P_I = (P_A)_I$. Therefore I is a stable model of P if and only if I is a stable model of P_A.

Now assume that $\sim A$ is true in I. Compared with Gelfond-Lifschitz transformation, the simplification of P to $P_{\sim A}$ also deletes all clauses that have a positive literal A in the body. Therefore P_I is identical to $(P_{\sim A})_I$, except that P_I may contains some additional clauses with positive literal A in the body. By definition, I is a stable model of P if and only if I coincides with $\mathcal{M}(P_I)$. Since $\sim A$ is true in I, A must be false in $\mathcal{M}(P_I)$. Therefore $\mathcal{M}(P_I) = \mathcal{M}((P_{\sim A})_I)$, and so I is a stable model of P if and only if I is a stable model of $P_{\sim A}$. □

2.2.3 Reduction of a Program

Our third observation is that the simplification carried out by the construction of P_L may determine the truth values of other ground atoms, which should be propagated as much as possible to reduce the program for which stable models are being sought. The propagation allows us to avoid choice points for guessing truth values of ground atoms whose values are already determined by previous assumptions.

Example 2.3 Let P be the following program on the left:

p :- ∼q, ∼r.	p :- ∼q, ∼r.
p :- ∼u.	p.
q :- ∼s.	q :- ∼s.
q :- ∼u.	q.
s :- ∼q.	s :- ∼q.
r :- ∼t.	r :- ∼t.
t :- ∼r.	t :- ∼r.
u :- ∼v.	u :- ∼v.
v :- ∼u.	v.
(P)	($P_{\sim u}$)

Suppose that ∼u is assumed to be true. The program, $P_{\sim u}$, obtained from P, is shown above on the right. Further propagation of known truth values of ground atoms leads to a partial interpretation, namely $\{p, q, \sim s, \sim u, v\}$, and a much simpler program:

r :- ∼t.
t :- ∼r.

In this case, the derived truth value of u is consistent with the assumption that ∼u is true. □

Propagation of previously known or assumed truth values is essentially a process of forward chaining. The result is a partial interpretation and a reduced program.

Definition 2.4 Let P be a ground program, and U be a set of ground atoms that contains all those occurring in P. We define $(P, U) \xrightarrow{I} (P', U')$ if and only if

- I is a non-empty interpretation, where $Pos(I)$ is the set of all ground atoms A such that P contains a fact A, and $Neg(I)$ is the set of all ground atoms A in U such that there is no clause in P with A in the head, and
- $U' = U - (Pos(I) \cup Neg(I))$, and
- P' is obtained from P by deleting
  - every clause that has a literal L in the body that is false in I, and
  - every literal L in the bodies of remaining clauses such that L is true in I, and then
  - every clause with A in the head where $A \in Pos(I)$.

(P, U) is *reduced* if there exists no interpretation I such that $(P, U) \xrightarrow{I} (P', U')$ for some P' and U'.

Notice that if (P, U) is not reduced, there exists a unique non-empty interpretation I, a unique (P', U') such that $(P, U) \xrightarrow{I} (P', U')$. If (P, U) is reduced, then U must be exactly the set of all atoms occurring in P.

In Example 2.3, let U_0 be $\{p, q, r, s, t, u, v\}$. Then $(P_{\sim u}, U_0) \xrightarrow{I_1} (P_1, U_1)$, where I_1 is $\{p, q, v\}$, and P_1 contains the following two clauses:

r :- ∼t.
t :- ∼r.

and U_1 is $\{r, s, t, u\}$. With another step of reduction, we have $(P_1, U_1) \xrightarrow{I_2} (P_2, U_2)$, where $I_2 = \{\sim s, \sim u\}$, $P_2 = P_1$, and $U_2 = \{r, t\}$. No further reduction is possible since (P_2, U_2) is reduced.

Definition 2.5 Let P be a ground program, and $\mathcal{U}(P)$ be the set of all ground atoms occurring in P. P is *reduced* to an interpretation I and a program P' if and only if for some $n \geq 0$,

$$(P_0, U_0) \xrightarrow{I_1} (P_1, U_1) \xrightarrow{I_2} ... \xrightarrow{I_n} (P_n, U_n)$$

where $P_0 = P$, $U_0 = \mathcal{U}(P)$, $P_n = P'$ and (P_n, U_n) is reduced, and $I = \cup_{1 \leq i \leq n} I_i$.

Notice that every finite ground program can be reduced to an interpretation and a (possibly simpler) program. The following lemma shows that reduction preserves stable models.

Lemma 2.3 Let P be a finite ground program, and P be reduced to an interpretation I^* and a program P^*. Then every stable model I of P is equal to $I^* \cup J$ for some stable model J of P^* and vice versa. Furthermore, if P is a definite program, then $Pos(I^*) = Pos(\mathcal{M}(P))$.

Proof: We show that every step of reduction preserves stable models, and the lemma follows by a simple induction.

Suppose that $(P_k, U_k) \xrightarrow{I_{k+1}} (P_{k+1}, U_{k+1})$. By definition, $Pos(I_{k+1})$ consists of all ground atoms A where A is a fact in P_k, and $Neg(I_{k+1})$ consists of all ground atoms $A \in U$ where there is no clause in P_k with A in the head.

Let I be any interpretation such that $Pos(I) \cup Neg(I) = U_k$, and $I = I_{k+1} \cup J$ for some interpretation J where $Pos(J) \cup Neg(J) = U_{k+1}$. Then $\mathcal{M}((P_k)_I) = I_{k+1} \cup \mathcal{M}((P_{k+1})_J)$.

For any interpretation I such that $Pos(I) \cup Neg(I) = U_k$, I is a stable model of P_k if and only if $I = \mathcal{M}((P_k)_I)$, if and only if $I = I_{k+1} \cup J$ for some J where $Pos(J) \cup Neg(J) = U_{k+1}$, and $J = \mathcal{M}((P_{k+1})_J)$, i.e., J is a stable model of P_{k+1}. □

The reduction of a program with respect to a partial interpretation differs from the simplification of a program according to the assumed truth value of a ground atom. Recall that if a ground atom A is *assumed* to be true, this assumption cannot be used to delete any occurrence of A in a program as a clause body literal. On the other hand, the reduction of a program P with respect to a partial interpretation I is similar to the bottom-up computation embodied in the transformation $\mathcal{T}_P(I)$ [1].

Reduction, however, does not attempt to compute the well-founded semantics. It derives only literals that are true or false with respect to Clark's completion of a program. For instance, the following program

p :- q.
q :- p.

cannot be reduced further. In our framework, reduction is used in stable model computation, which is carried out after a query is evaluated under the well-founded semantics. There is no interleaving of computations of stable models and the well-founded semantics, in the sense that our algorithm of stable model computation does not call any general procedure for computing the well-founded semantics.

2.3 Assume-and-Reduce Algorithm

Figure 1 shows the *assume-and-reduce* algorithm for computing stable models of a finite ground program. The algorithm is non-deterministic in the sense that certain choices have to be made at some point. However, all stable models can be enumerated through backtracking.

```
Input:  a finite ground program P
Output: a stable model or failure
begin
(1)     Let P be reduced to an interpretation I_0 and a program P_0;
(2)     DI := I_0; Pgm := P_0;
(3)     AI := ∅; N := 𝒩(Pgm);
(4)     while N ≠ ∅ do begin
(5)         Delete an arbitrary element, A, from N;
(6)         if A ∉ DI and ~A ∉ DI then begin
(7)             choice(A, L); /* choice point: L can be either A or ~A */
(8)             AI := AI ∪{L};
(9)             Let Pgm_L be reduced to an interpretation I* and a program P*;
(10)            DI := DI ∪I*; Pgm := P*;
(11)            if DI ∪ AI is inconsistent then
(12)                fail (and backtrack)
(13)        end
(14)    end;
(15)    if A ∈ AI and A ∉ DI for some ground atom A then
(16)        fail (and backtrack)
(17)    else begin
(18)        for every A that occurs in Pgm, add ~A to DI;
(19)        return AI ∪ DI as a stable model of P;
(20)    end
end
```

Figure 1: The assume-and-reduce algorithm for computing stable models

Theorem 2.4 Let P be a finite ground program. Then an interpretation I is a stable model of P if and only if I is returned by an execution of the *assume-and-reduce* algorithm.

Proof: In the algorithm, DI represents the set of ground literals that have been *derived* to be true (possibly from previous assumptions), and AI represents the set of ground literals that are *assumed* to be true. The algorithm explores a tree of search space for stable models in a non-deterministic and backtracking manner, where each node can be represented by a triple (AI, DI, Pgm). It terminates for finite ground programs.

We prove that the search space explored by the assume-and-reduce algorithm is complete. Initially, the root node of the search tree is $(AI, DI, Pgm) = (\emptyset, I_0, P_0)$, where P is reduced to

I_0 and P_0. By Lemma 2.3, I is a stable model of P if and only if $I = I_0 \cup J$ and J is a stable model of P_0, i.e., $I = AI \cup DI \cup J$, where J is a stable model of Pgm.

Given a node v represented by (AI, DI, Pgm), let A be any ground atom in $\mathcal{N}(P)$, i.e., $\sim A$ occurs in P, such that neither A or $\sim A$ is in $AI \cup DI$. Then $\sim A$ is still in Pgm. There are two choices, either A is true or $\sim A$ is true. By Lemma 2.2, an interpretation J is a stable model of Pgm if and only if either A is true in J and J is a stable model of Pgm_A, or $\sim A$ is true in J and J is a stable model of $Pgm_{\sim A}$. Let Pgm_A ($Pgm_{\sim A}$) be reduced to an interpretation I^*_A ($I^*_{\sim A}$) and a program P^*_A ($P^*_{\sim A}$). Again, by Lemma 2.3, J is a stable model of Pgm_L, where L can be either A or $\sim A$, if and only if $J = I^*_L \cup J^*_L$, where J^*_L is a stable model of P^*_L. Then v has two child nodes. One is $(AI \cup \{A\}, DI \cup I^*_A, P^*_A)$ and the other is $(AI \cup \{\sim A\}, DI \cup I^*_{\sim A}, P^*_{\sim A})$. By the arguments above, for any interpretation I, $I = AI \cup DI \cup J$ for some stable model J of Pgm if and only if for L that can be either A or $\sim A$, $L \in J$ and $J = I^*_L \cup J^*_L$ for some stable model J^*_L of P^*_L, if and only if $I = AI \cup \{L\} \cup DI \cup I^*_L \cup J^*_L$ for some stable model J^*_L of P^*_L.

For a leaf node v represented by (AI, DI, Pgm) in the search tree for stable models, either $AI \cup DI$ is inconsistent, or for every $A \in \mathcal{N}(P)$, A or $\sim A$ is in $AI \cup DI$. In the latter case, Pgm does not contain any negative literals. Furthermore Pgm has no facts since it can be further reduced otherwise. Therefore the only stable model of Pgm, which is also the unique minimal model $\mathcal{M}(Pgm)$, is that every ground atom occurring in Pgm is false. If $A \in AI$ and $A \notin DI$ for some ground atom A, $\sim A$ must be true either in DI or in $\mathcal{M}(Pgm)$, which is inconsistent with the assumption in AI that A is true. □

2.4 Backward Propagation of Assumed Truth Values

According to the definition of stable models, the *assumed* truth values of ground atoms should coincide with the *derived* truth values. The assume-and-reduce algorithm uses forward chaining to propagate the assumed truth values of ground atoms. This propagation may derive the truth values of more ground atoms so that there is no need to lay down a choice point for guessing their truth values.

SLG incorporates backward propagation of assumed truth values of ground atoms under certain conditions. Unlike forward propagation, which computes the derived truth values of ground atoms, backward propagation of assumptions may lead to more assumptions, thus reducing the search space for stable models.

Example 2.4 One application of stable models is to provide a semantics for programs with *choice* constructs [26]. Suppose that three students are taking the AI class.

take(sean, ai). take(irene, ai). take(chris, ai).

The following ground program chooses exactly one student taking the AI class:

choose(sean,ai) :- ∼diff(sean,ai).
diff(sean,ai) :- choose(irene,ai).
diff(sean,ai) :- choose(chris,ai).

choose(irene,ai) :- ∼diff(irene,ai).
diff(irene,ai) :- choose(sean, ai).

diff(irene,ai) :- choose(chris,ai).

choose(chris,ai) :- ∼diff(chris,ai).
diff(chris,ai) :- choose(sean,ai).
diff(chris,ai) :- choose(irene,ai).

There are three ground negative literals in the program. Suppose that diff(sean,ai) is assumed to be false. By backward propagation, we can infer that both choose(irene,ai) and choose(chris,ai) must be assumed to be false too. All three assumptions can be used to simplify the program to the following:

choose(sean,ai).
diff(irene,ai) :- choose(sean,ai).
diff(chris,ai) :- choose(sean,ai).

A reduction of the program derives that diff(irene,ai) and diff(chris,ai) are true. □

Let P be a finite ground program. SLG supports backward propagation under two situations:

- If a ground atom A is assumed to be true, and P contains exactly one clause with A in the head, of the form
$$A \text{ :- } L_1, ..., L_n$$
then every $L_i (1 \leq i \leq n)$ is assumed to be true;
- If a ground atom A is assumed to be false, then for every clause in P with A in the head and a single literal L in the body, L is assumed to be false.

The backward propagation may be repeated several times. The correctness of backward propagation is obvious according to the definition of stable models. The *assume-and-reduce* algorithm can be modified to include backward propagation, the details of which are omitted.

3 Integration with Computation of the Well-Founded Semantics

The *assume-and-reduce* algorithm deals with only finite ground programs and computes (two-valued) stable models. This section shows how to integrate computations of the well-founded semantics and stable models to provide query evaluation of non-ground programs for practical applications.

It is known that for logic programs without loops through negation, e.g., modularly stratified programs [25], the well-founded partial model is total and coincides with the unique stable model of the program. In that case, computation of the well-founded semantics is sufficient. For programs with literals involved in loops through negation, the well-founded partial model is in general three-valued. We discuss how negative loops should be handled in order to facilitate computation of stable models and to ensure the polynomial time data complexity of query evaluation under the well-founded semantics at the same time.

3.1 Handling Negative Loops

Negative loops occur due to recursion through negation. There are two main issues, namely how to detect negative loops and how to treat literals that are involved in negative loops so that query evaluation can proceed.

Example 3.1 Consider the following program [31]:

```
win(X)  :- move(X,Y), ~win(Y).
move(a,b).   move(b,a).   move(b,c).   move(c,d).
```

Figure 2 shows a portion of the SLDNF tree for the query win(a), which contains an infinite branch through negation. □

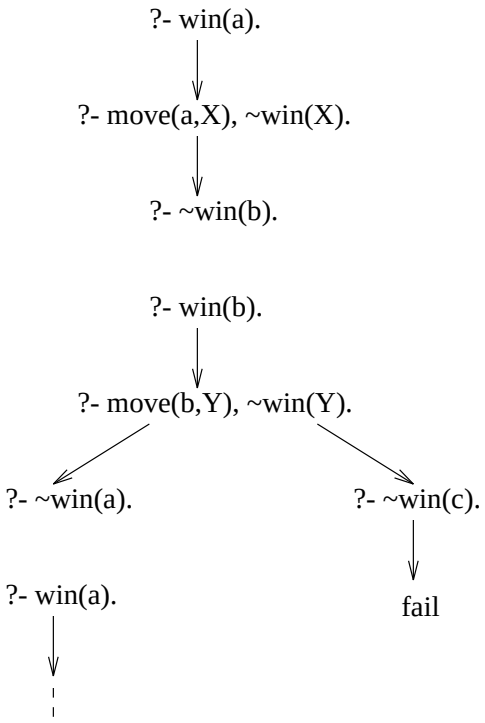

Figure 2: SLDNF tree for $win(a)$

A simple mechanism for negative loop detection is to associate with each call a *negative context*. This approach has been adopted in Well! [2] and in XOLDTNF resolution [6]. Consider a branch through negation in an SLDNF tree. The negative context of a call on the branch is the set of ground negative literals encountered along the path from the root to the call. In Figure 2, the initial call $win(a)$ has an empty negative context. The negative context for $win(b)$ is $\{\sim win(b)\}$, and the negative context for the second call of $win(a)$ is $\{\sim win(b), \sim win(a)\}$.

In the tree for the second call $win(a)$, when $\sim win(b)$ is selected, it is in the negative context of $win(a)$, indicating that there is a negative loop. The approach in XOLDTNF resolution [6] is to treat the selected ground negative literal $\sim win(b)$ as *undefined*. In general, an answer consists of not only an instance of a query atom, but also a truth value indicating whether the answer

is true or undefined. Figure 3 shows a portion of the XOLDTNF forest for query $win(a)$. Each node is labeled by a pseudo-clause. The head captures bindings of relevant variables that have been accumulated and the truth value, and the body contains literals that are yet to be solved.

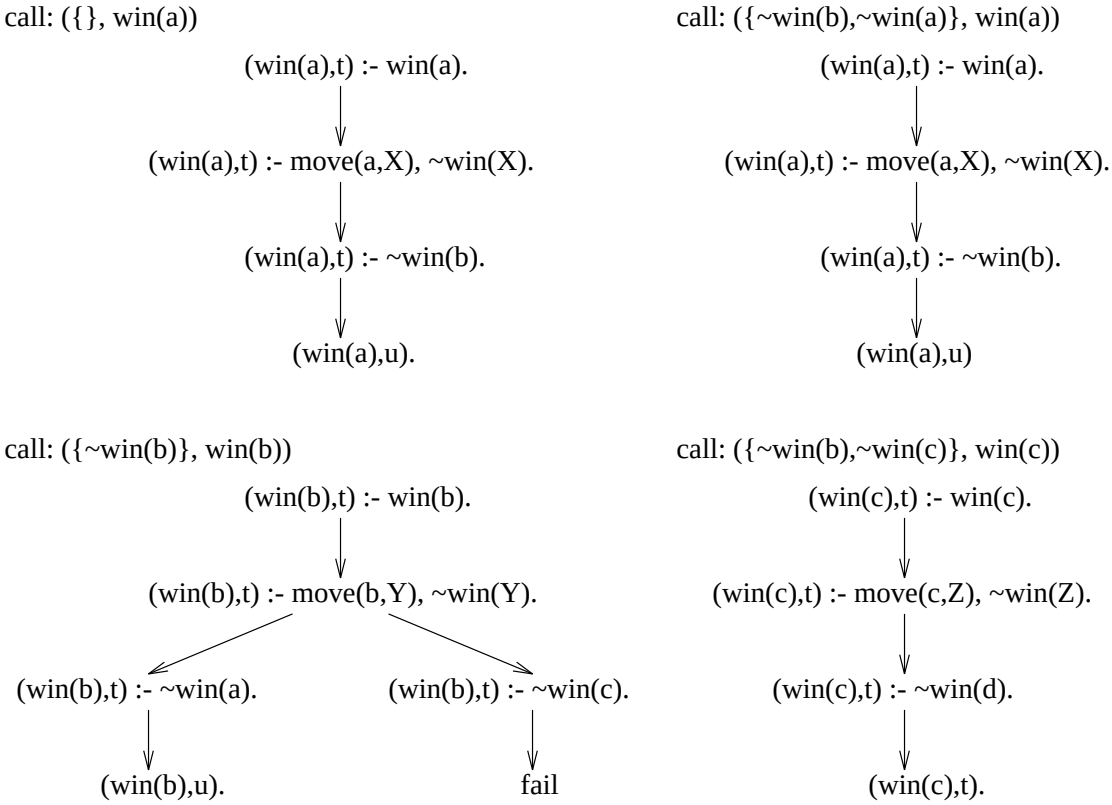

Figure 3: XOLDTNF forest for $win(a)$

The detection of negative loops using negative contexts is easy to implement in goal-oriented query evaluation. However, associating with each call a negative context prevents the full sharing of answers of a call across different negative contexts. Examples can be constructed in which a subgoal may be evaluated in an exponential number of negative contexts [7], even though the well-founded semantics is known to have a polynomial time data complexity.

Treating negative literals involved in negative loops as *undefined* is appropriate for query evaluation under the well-founded semantics. But it destroys the mutual dependencies among the negative literals. If a query turns out to be undefined in the well-founded semantics, there is little information that can be used for computation of stable models.

In [7], we developed a framework called *SLG resolution*. It detects potential negative loops by maintaining dependency information among calls incrementally. Each call (up to renaming of variable) is evaluated at most once, allowing the full sharing of answers. When a potential negative loop is detected, negative literals that are involved are *delayed* so that other literals in the body of a clause can be evaluated. These delayed literals may be simplified later if they become known to be true or false, or they may be returned as part of a conditional answer otherwise.

Figure 4 shows a portion of the SLG forest for query $win(a)$. A vertical bar (|) separates delayed literals on the left and the remaining literals on the right in the body of a clause. Notice

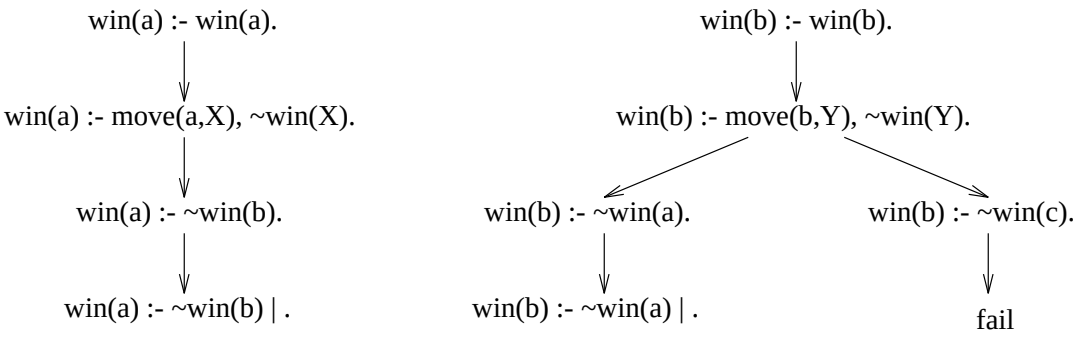

Figure 4: SLG forest for $win(a)$

that there is a conditional answer for $win(a)$, namely win(a) :- ~win(b), and similarly for $win(b)$, win(b) :- ~win(a). These conditional answers constitute a residual program, to which the assume-and-reduce algorithm can be applied to derive stable models and the answers of the original query in each stable model.

3.2 Simplification of Delayed Literals

Given an arbitrary but fixed computation rule, there are programs in which ground negative literals must be delayed before their truth or falsity is known.

Example 3.2 Assume that a left-most computation rule is used and that s is to be solved with respect to the following program:

s :- ~p, ~q, ~r.
p :- ~s, ~r, q.
q :- ~s, ~p, r.
r :- ~s, ~q, p.

The first negative loop involves s and p, which is processed by delaying $\sim p$ and $\sim s$. In the clause of s, the next body literal $\sim q$ is then selected, which leads to the second negative loop involving s and q. Delaying is applied again so that query evaluation can proceed. The computation rule selects the next body literal, namely $\sim r$, in the clause of s, whose evaluation results in the third negative loop involving s and r. The literal $\sim r$ in the clause of s and the literal $\sim s$ in the clause of r are delayed. At this point, the clause of s does not have any body literals that are not delayed. Thus we derive a conditional answer for s, namely s :- ~p, ~q, ~r. The evaluation of p, q, and r continues, leading to a negative loop involving p, q, and r. The corresponding negative literals are delayed. Computation continues, and a positive loop is detected among p, q, and r. They become completely evaluated without any answers, and so they are false. The falsity of p, q, and r is then propagated to derive a true answer for s. □

To facilitate the simplification of delayed literals, SLG sets up forward chaining links among calls when a conditional answer is derived. When the truth value of a ground atom A becomes known, all conditional answers with delayed literals A or $\sim A$ are simplified.

Not all delayed literals can be simplified as the well-founded semantics is in general three-valued. If a query has undefined instances under the well-founded semantics, its evaluation

produces a residual program consisting of all relevant conditional answers, as well as forward chaining links for simplification of delayed literals.

For the program and query $win(a)$ in Example 3.1, Figure 5 shows the residual program for $win(a)$ that consists of two conditional answers and the corresponding forward chaining links. A link from $win(b)$ to $win(a)$ indicates that if $win(b)$ is true or false, some conditional answer of $win(a)$ can be simplified. These forward chaining links are used directly in SLG for computation of stable models, for propagation of assumed truth values of ground atoms and for reduction of the residual program.

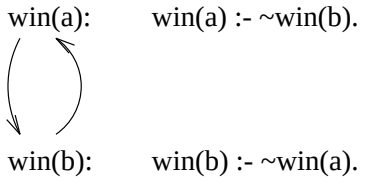

Figure 5: Residual program and forward chaining links for $win(a)$

3.3 General Logic Programs

Van Gelder has generalized the well-founded semantics of normal logic programs to the alternating fixpoint logic of general logic programs [30]. The SLG resolution developed in [7] has been extended for goal-oriented query evaluation of general logic programs [4]. Similarly it produces a residual program and corresponding forward chaining links for queries that have undefined instances under the alternating fixpoint logic.

Example 3.3 The following program describes a coloring of nodes in a graph in such a way that two adjacent nodes cannot be both colored.

color(X) :- $\forall$ Y. ($\sim$edge(X,Y) $\vee$ $\sim$color(Y)).
edge(a,b). edge(b,a). edge(b,c). edge(c,d).

Consider the query color(a). By By resolving (color(a) :- color(a)) with the program clause, we obtain a new clause for color(a):

color(a) :- $\forall$ Y. ($\sim$edge(a,Y) $\vee$ $\sim$color(Y)).

The literal, $\sim$edge(a,Y), is selected. The corresponding positive subgoal, edge(a,Y), is evaluated completely and has one answer, namely edge(a,b). Hence $\sim$edge(a,Y) is true for all Y that is distinct from b. We return the answer of edge(a,Y) to the clause for color(a). By resolving the universal disjunction

$$\forall \text{ Y. } (\sim\text{edge(a,Y)} \vee \sim\text{color(Y)})$$

with the answer edge(a,b), we derive a unit clause, color(b). The clause for color(a) is replaced by the following clause:

color(a) :- $\sim$color(b).

The literal, ∼color(b), is then selected. A new subgoal, color(b), is created and evaluated. Figure 6 shows the SLG forest resulted from the evaluation of color(a), where ∀ is represented by *All* and disjunction ∨ is represented by ';'. Notice that color(d) is derived to be true and color(c) is derived to be false, while both color(a) and color(b) are undefined. The residual program consisting of

color(a) :- ∼color(b).
color(b) :- ∼color(a).

can be further processed for stable model computation. □

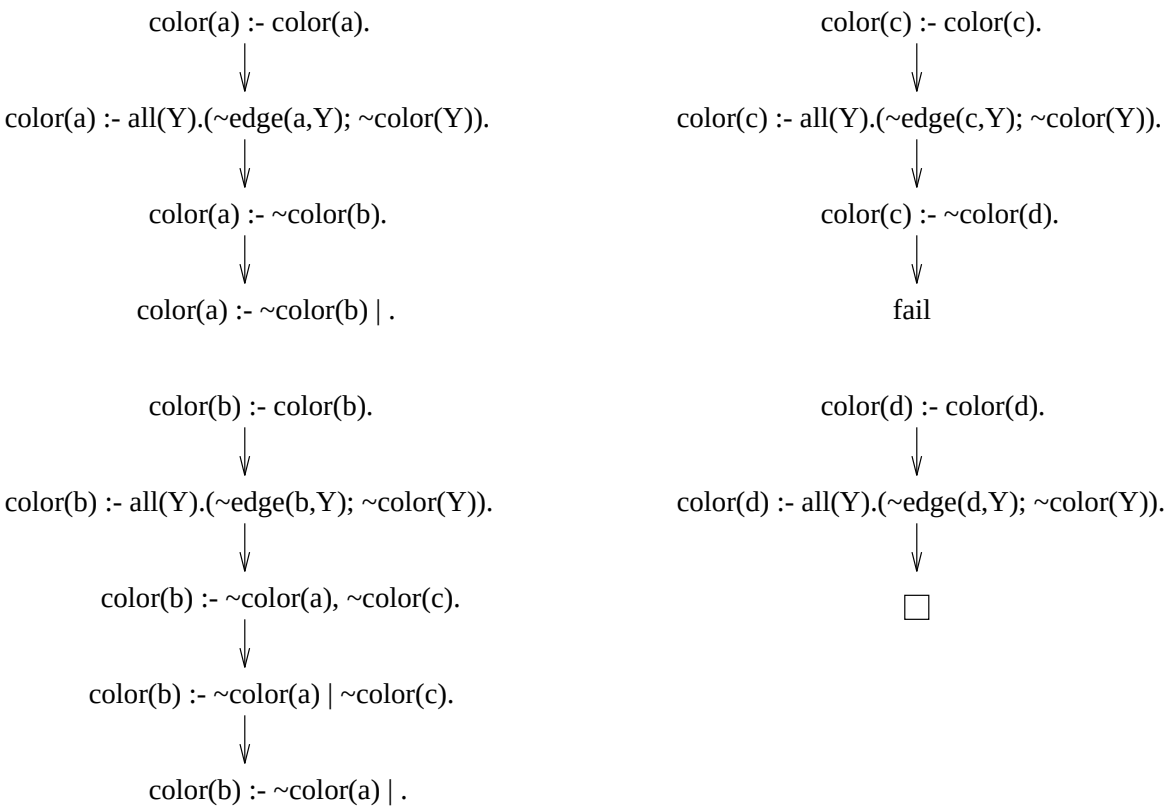

Figure 6: SLG forest for color(a)

3.4 Stable Models: Two-Valued versus Three-Valued

Let P be a program and Q be a query. SLG first evaluates Q with respect to the well-founded semantics of P. The result includes a set of true and false instances of Q, and in general, a residual program $P_{und(Q)}$ for undefined instances of Q. (Two-valued) stable models for the residual program, $P_{und(Q)}$, can be derived by using the *assume-and-reduce* algorithm. However, a (two-valued) stable model of $P_{und(Q)}$ may or may not be extended to a (two-valued) stable model of P.

Example 3.4 Consider the following simple program P:

a :- ∼b.
b :- ∼a.
p :- ∼p.
p :- ∼b.

and query b. The residual program for b contains two clauses:

b :- ∼a.
a :- ∼b.

There are two stable models for the residual program, one in which b is true and the other in which a is true. However, the one in which b is true cannot be extended to a stable model of the original program, even though there is a three-valued stable model of the original program in which b is true. □

In general, answers of a query computed by SLG are answers with respect to three-valued stable models of a given program P. SLG does not enumerate all possible three-valued stable models of P.

To compute (two-valued) stable models of a program P in SLG, one may define a new predicate that calls all predicates in P with distinct variables as arguments. By evaluating the new predicate (with distinct variables as arguments), SLG derives a residual program P_{und} for all undefined atoms of the original program. All (two-valued) stable models of P can be derived by applying the *assume-and-reduce* algorithm to P_{und}.

4 Integration with Prolog

All normal logic programs are obviously syntactically correct Prolog programs, even though their execution under Prolog's strategy may not terminate. One of the objectives of the SLG system is to integrate query evaluation with ordinary Prolog execution so that existing Prolog environments can be readily used for knowledge-based applications. This section describes the interface of SLG from a user's point of view.

4.1 Syntax

The syntax of Prolog is used for input programs, with additional directives for predicate declarations. Predicates can be declared as *tabled* or *prolog*. Tabled predicates are evaluated using SLG resolution. Prolog predicates are solved by calling Prolog directly. Calls to tabled predicates are remembered in a table with their corresponding answers. Future calls to tabled predicates that are renaming variants of previous calls are not re-evaluated, but will be satisfied using answers that are computed as a result of the previous calls.

It is actually legal for a tabled predicate to call a Prolog predicate which in turn calls a tabled predicate. However, the two invocations of tabled predicates will not share the same table, and Prolog's infinite loops will not be terminated.

There are also certain constraints on the form of clauses that can be used to define tabled predicates. In particular, the body of a clause for a tabled predicate should be a conjunction of literals. Cuts are allowed in the body before any occurrence of a tabled predicate. Common uses

of cuts for selection of clauses according to certain guard conditions are supported for tabled predicates.

Clauses with a universal disjunction of literals in the body are allowed. They are indicated by an operator ←—. For the program in Example 3.3, it will be written as follows:

color(X) ←— ∼edge(X,Y); ∼color(Y)).

where all variables that occur only in the body are universally quantified in the body, and disjunction is indicated by ';'. Standard safety conditions are assumed [32]. To determine if the body of a clause is safe to evaluate, we convert the universal quantification into existential quantification:

$$color(X) \longleftarrow \sim\exists Y.(edge(X,Y) \wedge color(Y))$$

The notion of safety requires that all free variables in the body must be bound when the negation in the body is evaluated. For the conjunction inside the existential quantification, all variables in a negative literal must also occur in the a positive literal. Accordingly we require that for a clause with a universal disjunction of literals in the body, the head must be ground when the clause is used, and all variables that occur in positive literals in the body also occur in the head or in negative literals in the body.

4.2 Query Interface

Tabled predicates are evaluated with respect to the well-founded semantics by default. Both true and conditional answers can be returned.

Example 4.1 The following is the *winning* program in Prolog syntax with a tabling directive of SLG:

```
:- tabled win/1.
win(X) :- move(X,Y), \+win(Y).
move(a,a). move(a,b). move(b,a). move(b,c).
```

By the initial default, all predicates are Prolog predicates unless declared otherwise. The default can be changed to *tabled* by users if needed. The Prolog interface is also used for queries. In the following, the first query asks for true answers under the well-founded semantics, and the second returns also conditional answers, where each condition is a list of delayed literals.

```
| ?- win(X).

X = b ? ;

no
| ?- win(X) <- C.

C = [\+win(a)],
X = a ? ;
```

```
C = [],
X = b ? ;

no
```

□

By applying the assume-and-reduce algorithm to the residual program produced by the computation of the well-founded semantics, SLG derives answers of queries under (two-valued) stable models of the residual program (or three-valued stable models of the original program). (See Section 3.4 for more discussions.) In general, there may be multiple stable models of the residual program, and answers of queries have to be qualified by the corresponding stable model. SLG provides a versatile interface for query evaluation under stable models of the residual program. It includes the following predicates:

- `st(Call,PSM)` (or `stnot(Call,PSM)`): It succeeds if `Call` is a ground atom and there is a stable model `PSM` in which `Call` is true (or false);
- `stall(Call,Anss,PSM)`: It computes a stable model `PSM` and collects all answers of `Call` in a list `Anss`.
- `stselect(Call,PSM0,Anss,PSM)`: It is similar to `stall/3`, except that it computes only those stable models in which all ground literals in `PSM0` are true. This allows the user to select only those stable models that satisfy certain conditions.

Alternative stable models of the residual program and the corresponding answers of `Call` are returned upon backtracking.

Example 4.2 The following is a program that selects exactly one student for each course [26]:

```
:- tabled choose/2, diff/2.

choose(S,C) :- take(S,C), \+diff(S,C).
diff(S,C) :- choose(S1,C), \+same(S1,S).

take(sean,ai). take(irene,ai). take(chris,ai).
take(brad,db). take(irene,db). take(jenny,db).

same(X,X).
```

The query below selects those stable models in which `choose(sean,ai)` and `\+choose(irene,db)` are true:

```
| ?- stselect(choose(_,_),[choose(sean,ai),\+choose(irene,db)],Anss,PSM).

Anss = [choose(brad,db),choose(sean,ai)],
PSM = [\+choose(irene,db),choose(sean,ai),\+diff(brad,db),diff(chris,ai),
       diff(irene,ai),diff(irene,db),diff(jenny,db),\+diff(sean,ai)] ? ;
```

```
Anss = [choose(jenny,db),choose(sean,ai)],
PSM = [\+choose(irene,db),choose(sean,ai),diff(brad,db),diff(chris,ai),
       diff(irene,ai),diff(irene,db),\+diff(jenny,db),\+diff(sean,ai)] ? ;

no
```

□

5 Related Work and Experimental Results

SLG seems to be the first work that provides integrated query evaluation under various semantics, including the well-founded semantics and stable models of normal logic program, the alternating fixpoint logic of general logic programs, and SLDNF resolution in Prolog execution. The combination of Prolog's programming environment and SLG's query processing capabilities makes it easier to develop knowledge-based applications.

The delaying mechanism for handling negative loops in SLG has two important implications. First, it avoids redundant derivations in the computation of the well-founded semantics as delayed literals are simplified away as needed using forward chaining links. Second, it allows SLG to produce a residual program for undefined instances of a query, which can be used directly for stable model computation. Most of existing techniques for query evaluation under the well-founded semantics replace looping negative literals with an *undefined* truth value [2, 3, 6], or use the alternating fixpoint method to compute possibly true or false facts [28]. In either case, little information is saved for later computation of stable models.

Goal-oriented query evaluation with respect to stable models was studied by Dung in an abductive framework [9]. It is a refinement of Eshghi and Kowalski's abductive procedure [11]. A ground negative literal can be assumed to be true if it does not lead to any inconsistency. It is not clear how to specialize the abductive procedure to compute only answers that are valid in the well-founded semantics. Pereira *et al.* [19] developed derivation procedures for goal-oriented evaluation of ground programs under the well-founded semantics or stable models.

A bottom-up procedure, called *backtracking fixpoint*, was developed by Sacca and Zaniolo [26], which non-deterministically constructs a stable model if one exists. In [15], stable models are characterized by a transformation of normal logic programs into semantically equivalent positive disjunctive programs, with integrity constraints in the denial form $\leftarrow B' \wedge B$ for each atom B, where B' is a *new* atom for the negation of B. Stable models are constructed using the *model generation theorem prover* (MGTP), which is a bottom-up forward chaining system. Starting with the set containing the empty interpretation, MGTP either expands an interpretation according to a disjunctive clause or discards an interpretation if it violates some integrity constraints. Other methods that construct all stable models simultaneously include [10, 20, 29].

The work most closely related to ours is the branch-and-bound method by Subrahmanian *et al.* [29]. Their approach first computes Fitting's Kripke-Kleene semantics and at the same time "compacts" the program by deleting parts of the program. The program is then processed and further compacted by an alternating fixpoint procedure that computes the well-founded semantics. The resulting program is used for computation of stable models using a branch-and-bound method.

The branch-and-bound method and SLG are similar in the sense that both assume the truth values of some atoms and compact or simplify the program as computation proceeds. However, there are several major differences. First, the branch-and-bound method in [29] computes and stores all stable models simultaneously. As the number of stable models can be exponential, storing all stable models at the same time may require a substantial amount of memory. SLG, on the other hand, computes alternative stable models through backtracking. Second, the branch-and-bound method interleaves the assumption of the truth value of an atom with the computation of the well-founded semantics. After the truth value of an atom is assumed, the resulting program is processed with respect to the well-founded semantics. SLG only attempts to reduce the program in such a way that ground atoms that are true or false in Clark's completion are derived, which is simpler than a full-fledged computation of the well-founded semantics. Third, the branch-and-bound method is intelligent in choosing which atom to make an assumption about its truth value. It selects an atom in a leaf strongly connected component according to the dependency graph. SLG uses a very simple criterion, namely only those atoms whose complements occur in a program can be assumed. Finally, SLG integrates query evaluation with ordinary Prolog execution and accepts programs with variables, while the method in [29] assumes a finite ground program.

To get a rough idea how SLG performs, we took two benchmark programs reported in [29] together with their timing information, and ran them using SLG. It should be pointed out that a systematic study of benchmark programs have to be conducted before a clear picture of the relative performance of the various systems can be obtained. The prototype compiler in [29] was written in C running under the Unix environment on a Decstation 2100. SLG was written in Prolog running under SICStus Prolog in the Unix environment on a Decstation 2100. The timing information of SLG was obtained by Prolog builtin predicate *statistics*. All timing data are in milliseconds.

The first program consists of the following rules:

```
z1(X) :- v1(X), w1(X).        z2(X) :- v1(X), w2(X).
z3(X) :- v2(X), w1(X).        z4(X) :- v2(X), w2(X).
v1(X) :- s(X).                v2(X) :- t(X).
w1(X) :- p(X).                w2(X) :- q(X).
t(X) :- ~s(X).                s(X) :- ~t(X).
p(X) :- ~q(X).                q(X) :- ~p(X).
```

An additional unary predicate y(_) is used to introduce constants in the program. To test the program in SLG, we added the following rules:

```
m(X) :- y(X),z1(X).
m(X) :- y(X),z2(X).
m(X) :- y(X),z3(X).
m(X) :- y(X),z4(X).
```

The query m(X) is then evaluated by calling `stall(m(X),Anss,PSM)`. A failure loop is used to get all answers of the call. Table 7 shows the timing of SLG and the intelligent branch-and-bound in [29]. The relative rate of increase in execution time in SLG seems closer to the rate of increase of the number of stable models. The execution time of SLG falls below that of the intelligent

Number of constants	1	2	3	4	5
Number of stable models	4	16	64	256	1024
SLG	251	891	3162	12965	55276
Branch and bound	43	262	1413	9431	95766

Figure 7: SLG and branch-and-bound for enumerating all stable models

branch and bound when the number of constants reaches 5, probably due to the large number of stable models that have to be stored in the latter.

The second program, also taken from [29], is as follows:

s(X) :- p(X), q(X).
s(X) :- q(X), r(X).
q(X) :- ∼r(X).

s(X) :- p(X), r(X).
p(X) :- ∼q(X).
r(X) :- ∼p(X).

It is augmented by a unary predicate y(_) whose sole purpose is to introduce constant symbols into the program. For SLG, we added y(X) at the beginning of the body of each rule for s(X). The query s(X) is then evaluated by calling `stall(s(X),Anss,PSM)`, and a failure loop is used to check all possibilities. Table 8 shows the timing information of SLG versus the intelligent branch and bound in [29]. In this case, there is no stable model for the program, which can be detected as soon as the truth value of p(X), q(X), or r(X) for some X is assumed. Thus most of the time is spent on the computation of the well-founded semantics.

Number of constants	5	10	15	20	25	30	35	40	45	50
SLG	347	721	1100	1506	1991	2220	2733	3158	3591	4002
Branch and bound	54	117	198	303	431	586	755	972	1203	1475

Figure 8: SLG and branch-and-bound for checking non-existence of stable models

The notion of stable models provides a declarative semantics for the *choice* construct in LDL [18]. It has been shown by Greco *et al.* [14] that for certain classes of programs with choice, the data complexity for computing a stable model is polynomial time. The choice construct has been used to model a variety of applications where only one stable model is needed [14].

We tested SLG on a classical choice program [14]:

choose(X,Y) :- base(X,Y), choice(X,Y).

It is translated into a normal logic program:

choose(X,Y) :- base(X,Y), ∼diffchoice(X,Y).
diffchoice(X,Y) :- choose(X,Z), ∼same(Y,Z).
same(X,X).

The *base* relation contains a set of facts of the form:

base(i,a). base(i, b). base(i, c). base(i, d).

where i ranges from 1 to N, and N is used as a parameter. The query $choose(X, Y)$ is evaluated by calling `stall(choose(X,Y),Anss,PSM)`. We measured the time (on a Decstation 2100) for computing the first solution for programs of different sizes by varying N from 2 to 10. Table 9 shows the timing of SLG for different values of N. The execution time of SLG seems polynomial in the size of the base relation.

N	1	2	3	4	5	6	7	8	9	10
SLG	221	449	678	917	1161	1410	1840	2120	2400	2700

Figure 9: SLG for computing the first stable model

SLG is currently implemented as a Prolog meta interpreter [8], and therefore carries significant overhead. A compiler implementation of SLG by extending the Warren Abstract Machine is being carried out in the XSB project led by the second author [27]. XSB currently handles modularly stratified programs. A preliminary performance analysis shows that XSB is over an order of magnitude faster than SLG [5].

6 Conclusion

We have presented an assume-and-reduce algorithm for computing stable models and its integration in SLG with goal-oriented query evaluation under the well-founded semantics, or more generally the alternating fixpoint logic of general logic programs. The synergism exemplified by SLG between Prolog on the one hand and deductive query processing and nonmonotonic reasoning on the other offers an ideal environment for developing knowledge-based applications.

References

[1] K.R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *JACM*, 29(3):841–862, July 1982.

[2] N. Bidoit and P. Legay. Well!: An evaluation procedure for all logic programs. In *Intl. Conference on Database Theory*, pages 335–348, 1990.

[3] R. Bol and L. Degerstedt. Tabulated resolution for well founded semantics. In *Intl. Logic Programming Symposium*, October 1993.

[4] W. Chen. Query evaluation of alternating fixpoint logic. submitted, October 1993.

[5] W. Chen, T. Swift, and D.S. Warren. Efficient top-down computation of queries under the well-founded semantics. Technical Report 93-CSE-33, Computer Science and Engineering, Southern Methodist University, August 1993. submitted.

[6] W. Chen and D.S. Warren. A goal-oriented approach to computing well founded semantics. *Journal of Logic Programming*, 17, 1993.

[7] W. Chen and D.S. Warren. Query evaluation under the well founded semantics. In *The Twelfth ACM Symposium on Principles of Database Systems*, 1993.

[8] W. Chen and D.S. Warren. *The SLG System*, November 1993. available by anonymous FTP from seas.smu.edu or cs.sunysb.edu.

[9] P.M. Dung. Negation as hypotheses: An abductive foundation for logic programming. In *Intl. Conference on Logic Programming*, June 1991.

[10] K. Eshghi. Computing stable models by using the ATMS. In *National Conference on Artificial Intelligence*, pages 272–277, 1990.

[11] K. Eshghi and R.A. Kowalski. Abduction compared with negation by failure. In *Intl. Conference on Logic Programming*, pages 235–254, 1989.

[12] M. Gelfond. On stratified autoepistemic theories. In *National Conference on Artificial Intelligence*, pages 207–211, 1987.

[13] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R.A. Kowalski and K.A. Bowen, editors, *Joint Intl. Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.

[14] S. Greco, C. Zaniolo, and S. Ganguly. Greedy by choice. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 105–113, 1992.

[15] K. Inoue and C. Sakama. Transforming abductive logic programs to disjunctive programs. In *Intl. Conference on Logic Programming*, 1993.

[16] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, New York, second edition, 1987.

[17] W. Marek and M. Truszczynski. Autoepistemic logic. *Journal of ACM*, 38(3):588–619, 1991.

[18] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, New York, 1989.

[19] L.M. Pereira, J.N. Aparicio, and J.J. Alferes. Derivation procedures for extended stable models. In *Intl. Joint Conference on Artificial Intelligence*, pages 863–868, 1991.

[20] S.G. Pimental and J.L. Cuadraro. A truth maintenance system based on stable models. In *North American Conference on Logic Programming*, pages 274–290, 1989.

[21] T.C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 11–21, 1989.

[22] T.C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13:445–463, 1990.

[23] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Joint Intl. Conference and Symposium on Logic Programming*, pages 273–287, 1992.

[24] K.A. Ross. A procedural semantics for well-founded negation in logic programs. *Journal of Logic Programming*, 13(1):1–22, 1992.

[25] K.A. Ross. *The Semantics of Deductive Databases*. PhD thesis, Department of Computer Science, Stanford University, August 1991.

[26] D. Saccà and C. Zaniolo. Stable models and non-determinism for logic programs with negation. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 205–217, 1990.

[27] P. Sagonas, T. Swift, and D.S. Warren. *The XSB Programmers Manual*, April 1993.

[28] P. Stuckey and S. Sudarshan. Well-founded ordered search. Manuscript, May 1993.

[29] V.S. Subrahmanian, D. Nau, and C. Vago. Wfs + branch and bound = stable models. submitted, 1993.

[30] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47:185–221, 1993.

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

[32] A. Van Gelder and R.W. Topor. Safety and translation of relational calculus queries. *ACM Transactions on Database Systems*, 16(2):235–278, June 1991.