Efficient Implementation of the Well-founded and Stable Model Semantics

Ilkka Niemelä* and Patrik Simons†

Abstract

An implementation of the well-founded and stable model semantics for range-restricted function-free normal programs is presented. It includes two modules: an algorithm for implementing the two semantics for ground programs and an algorithm for computing a grounded version of a range-restricted function-free normal program. The latter algorithm does not produce the whole set of ground instances of the program but a subset which is sufficient in the sense that no stable models are lost. The implementation of the stable model semantics for ground programs is based on bottom-up backtracking search. It works in linear space and employs a powerful pruning method based on an approximation technique for stable models which is closely related to the well-founded semantics. The implementation includes an efficient algorithm for computing the well-founded model of a ground program. The implementation has been tested extensively and compared with a state of the art implementation of the stable model semantics, the SLG system. In tests involving ground programs it clearly outperforms SLG.

Keywords: stable model semantics, well-founded semantics, normal programs, implementation.

1 Introduction

In recent years the semantical underpinnings of logic programs have been studied intensively. This work has lead to the development of new declarative semantics for logic programs with the well-founded semantics [VGRS91] and the stable model semantics [GL88] as the two most influential representatives. However, there is limited amount of work where actual implementations of these semantics have been developed.

In this paper we aim to develop an implementation of the stable model semantics for normal logic programs. We restrict ourselves to a decidable fragment consisting of range-restricted function-free normal programs. Our implementation includes two modules: (i) an algorithm for implementing the stable model semantics for ground programs and (ii) an algorithm for computing a grounded version of a range-restricted function-free normal program. The latter algorithm does not generate the whole set of ground instances of the program but a subset which is sufficient in the sense that no stable models are lost.

*Dept. of Computer Science, University of Koblenz, Rheinau 1, D-56075 Koblenz, Germany, ini@informatik.uni-koblenz.de, tel: +49-261-9119-649, fax: +49-261-9119-496

†Dept. of Computer Science, Helsinki University of Technology, Otakaari 1, FIN-02150 Espoo, Finland, Patrik.Simons@hut.fi

The algorithm for implementing the stable model semantics for ground programs is developed using an approach introduced for implementing default logic [Nie95]. This approach provides a framework for developing bottom-up methods for computing extensions of default theories. In this paper we use the framework to devise an efficient implementation for the stable model semantics that possesses some interesting properties.

- The implementation is able to solve a range of decision problems related to the stable model semantics. It can decide whether a program has a stable model, compute all stable models of a program and handle two basic query-answering tasks: decide whether a given formula is satisfied in some or all of the stable models of a program.
- The implementation is based on bottom-up backtracking search and employs a powerful pruning method based on an approximation technique for stable models which is closely related to the well-founded semantics.
- The implementation includes an efficient (quadratic time) algorithm for computing the well-founded model of a ground program.
- The implementation runs in linear space.

The new implementation opens up the possibility to apply the stable model semantics in areas where resulting programs are highly non-stratified and possess a potentially large number of stable models. Because of the polynomial space complexity our implementation is able to solve these kinds of hard instances provided that adequate amount of running time is allocated. We have tested the implementation extensively. In order to obtain challenging test cases we have used, e.g., combinatorial graph problems as a test domain. This domain has also been used in TheoryBase [CMMT95] which is a system for generating test cases for nonmonotonic reasoning and logic programs.

Our implementation compares favorably with other existing implementations of the stable model semantics [BNNS94, CW94, SNV94]. The methods described in [BNNS94, SNV94] do not seem to be able to handle programs with a large number of stable models because of exponential worst-case space requirements and when considering programs with a few stable models, our implementation seems to outperform them.

The SLG system [CW93, CW94] appears to be the best currently available implementation of the stable model semantics. SLG also exploits the well-founded semantics: first a residual ground program is produced from a program and a given query using a top-down query evaluation method for the well-founded semantics and then an assume-and-reduce algorithm is employed to compute stable models of the residual program. However, SLG differs in several respects from our approach. The residual program does not preserve stable model semantics, i.e., the stable models of the residual program do not coincide with the stable models of the original program [CW94]. On the other hand, the grounded version computed by our implementation preserves the stable model semantics. When searching for the stable models of a ground program, our algorithm seems to prune the search space much more efficiently than the assume-and-reduce algorithm proposed in [CW94]. The experiments we have run clearly confirm this and show that for ground programs the performance of our implementation is superior to that of SLG.

The paper is organized as follows. In the next section we briefly introduce the stable model semantics and the well-founded semantics. In Section 3 the algorithm for implementing the stable model semantics of ground programs is developed. Section 4 discusses the implementation of the

deductive closures needed in the the implementation. Section 5 presents a method to compute a grounded version of a range-restricted function-free program. In Section 6 we discuss our implementation. The test results and comparisons are presented in Section 7.

2 Preliminaries

In this section we briefly introduce the stable model semantics and the well-founded semantics for normal logic programs. We present a characterization of stable models that serves as the basis of the novel algorithm for computing stable models and give an alternative characterization for the well-founded model of a program using the least fixed point of a monotonic function.

We consider normal logic programs consisting of rules of the form

$$c \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m). \tag{1}$$

where a_i, b_j and c are atoms. For normal logic programs the *stable model semantics* is defined as follows [GL88]. Given a logic program P we first take its set of ground instances which we often denote by P_{g}. For a ground program P_{g}, the *reduct* P_{g}^S with respect to a set of atoms S is obtained from P_{g} by deleting (i) each rule that has $\mathrm{not}(b)$ in its body with $b \in S$ and (ii) all $\mathrm{not}(l)$ in the bodies of the remaining rules. Now a set of atoms S is a *stable model* of P iff S is the deductive closure of P_{g}^S when rules in P_{g}^S are seen as inference rules.

We reformulate the definition of a stable model in terms of a deductive closure operator $\mathrm{Dcl}(\cdot,\cdot)$ that we use for describing our approach to computing stable models. We call atoms and not-atoms (formulae of the form $\mathrm{not}(a)$) literals. For a set of literals S, by S^+ (S^-) we denote the set of atoms (not-atoms) in S. For example, if $S = \{a, \mathrm{not}(b), \mathrm{not}(c)\}$, then $S^+ = \{a\}$ and $S^- = \{\mathrm{not}(b), \mathrm{not}(c)\}$. For a set of atoms S, let $\mathrm{not}(S) = \{\mathrm{not}(\phi) \mid \phi \in S\}$. We denote by $\mathrm{Dcl}(P, L)$ the *deductive closure* of a set of ground rules P of the form (1) and a set of literals L. $\mathrm{Dcl}(P, L)$ is the smallest set of atoms which contains L^+ and which is closed under the inference rules $\mathrm{R}(P, L)$ where

$$\begin{aligned} \mathrm{R}(P,L) \quad = \quad \{\, c \leftarrow a_1, \ldots, a_n \mid \quad & c \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m) \in P, \\ & \{\mathrm{not}(b_1), \ldots, \mathrm{not}(b_m)\} \subseteq L^-\}. \end{aligned}$$

For example, if $P = \{\ q \leftarrow \mathrm{not}(p);\ \ r \leftarrow q\}$, then $\mathrm{R}(P, \{p, \mathrm{not}(p)\}) = \{\ q \leftarrow ;\ \ r \leftarrow q\}$ and $\mathrm{Dcl}(P, \{p, \mathrm{not}(p)\}) = \{p, q, r\}$. It is straightforward to see that Δ is a *stable model* of a set of ground rules P iff

$$\Delta = \mathrm{Dcl}(P, \mathrm{not}(\overline{\Delta}))$$

where $\overline{\Delta}$ is the complement of Δ, i.e., the atoms not in Δ. This is because $\mathrm{R}(P, \mathrm{not}(\overline{\Delta})) = P^{\Delta}$.

In [Nie95] we presented a compact characterization of extensions in default logic. Because of the close relationship this approach is applicable to stable models. We present the characterization here directly in terms of logic programs. The characterizing sets for stable models are called *full sets* and they are sets of not-atoms built from atoms in $\mathrm{NAnt}(P)$, the *negative antecedents* in P, i.e., atoms b such that $\mathrm{not}(b)$ appears in P. For example, $\mathrm{NAnt}(\{\ q \leftarrow \mathrm{not}(p);\ r \leftarrow q\}) = \{p\}$. The crucial observation is that for the deductive closure of P, a premise $\mathrm{not}(b)$ plays a role only when $b \in \mathrm{NAnt}(P)$, i.e., $\mathrm{Dcl}(P, \mathrm{not}(\overline{\Delta})) = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta))$. Thus, a set of atoms Δ is a stable model of P iff $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta))$.

Definition 2.1 *A set* Λ *of* not*-atoms is called* P*-full iff for all* $\phi \in \mathrm{NAnt}(P)$, $\mathrm{not}(\phi) \in \Lambda$ *iff* $\phi \notin \mathrm{Dcl}(P, \Lambda)$.

The following characterization theorem provides the basis for the novel algorithm.

Theorem 2.2 *(i) If* Λ *is* P*-full, then* $\mathrm{Dcl}(P, \Lambda)$ *is a stable model of a set of ground rules* P.

(ii) If there is a stable model Δ *of* P, *then* $\Lambda = \mathrm{not}(\mathrm{NAnt}(P) - \Delta)$ *is a* P*-full set such that* $\Delta = \mathrm{Dcl}(P, \Lambda)$.

Proof. (i) Let $\Delta = \mathrm{Dcl}(P, \Lambda)$. If $\phi \in \mathrm{NAnt}(P) - \Delta$, then as Λ is P-full, $\mathrm{not}(\phi) \in \Lambda$. If $\mathrm{not}(\phi) \in \Lambda$, then as Λ is P-full, $\phi \in \mathrm{NAnt}(P) - \Delta$. Thus $\mathrm{not}(\mathrm{NAnt}(P) - \Delta) = \Lambda$ holds. Hence, $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta))$ holds which implies that $\mathrm{Dcl}(P, \Lambda)$ is a stable model of P.

(ii) Let Δ be a stable model of P. Then $\Delta = \mathrm{Dcl}(P, \Lambda)$ for $\Lambda = \mathrm{not}(\mathrm{NAnt}(P) - \Delta)$. Let $\phi \in \mathrm{NAnt}(P)$. If $\phi \in \mathrm{Dcl}(P, \Lambda)$, then $\phi \in \Delta$ and $\mathrm{not}(\phi) \notin \Lambda$. If $\phi \notin \mathrm{Dcl}(P, \Lambda)$, then $\phi \notin \Delta$ and $\mathrm{not}(\phi) \in \Lambda$. Hence, Λ is P-full. □

Theorem 2.2 suggests the following straightforward method for finding all stable models. For each subset S of $\mathrm{NAnt}(P)$, test whether $\mathrm{not}(S)$ is P-full. If a full set Λ is found, $\mathrm{Dcl}(P, \Lambda)$ is a stable model of P by Theorem 2.2 (i) and by Theorem 2.2 (ii) for each stable model Δ, there is a corresponding full set Λ such that $\Delta = \mathrm{Dcl}(P, \Lambda)$. However, this method is not very practical: if there are n not-atoms in P, 2^n fullness tests are needed. The novel algorithm presented in the next section is also based on finding full sets for the stable models but it offers significant improvements over the straightforward method.

The original definition of the well-founded semantics [VGRS91] is somewhat complicated and we present here an alternative characterization using our closure operator. This kind of a characterization has been discussed, for example in [BS93]. Given a ground program P and a set of ground atoms S we denote by $\Gamma_P(S)$ the deductive closure $\mathrm{Dcl}(P, \mathrm{not}(\overline{S}))$. Now a stable model is a set of atoms S such that $S = \Gamma_P(S)$. We define the well-founded model of a program P using the fixed points of the operator Γ_P^2 ($\Gamma_P^2(S) = \Gamma_P(\Gamma_P(S))$). The operator Γ_P is anti-monotonic and, hence, Γ_P^2 is monotonic. Thus it has the least and greatest fixed point which we denote by $\mathrm{lfp}(\Gamma_P^2)$ and $\mathrm{gfp}(\Gamma_P^2)$, respectively. The *well-founded model* I of a ground program P can be defined as follows [BS93].

$$I^+ = \mathrm{lfp}(\Gamma_P^2), \quad I^- = \mathrm{not}(\overline{\mathrm{gfp}(\Gamma_P^2)})$$

It can be shown that the greatest fixed point is obtained from the least fixed point by applying the operator Γ_P: $\mathrm{gfp}(\Gamma_P^2) = \Gamma_P(\mathrm{lfp}(\Gamma_P^2))$. Hence, also the negative part of the well-founded model can be characterized in terms of the least fixed point, i.e., $I^- = \mathrm{not}(\overline{\Gamma_P(\mathrm{lfp}(\Gamma_P^2))})$.

Proposition 2.3 $\mathrm{gfp}(\Gamma_P^2) = \Gamma_P(\mathrm{lfp}(\Gamma_P^2))$.

Proof. Let $I = \mathrm{lfp}(\Gamma_P^2)$ and $J = \mathrm{gfp}(\Gamma_P^2)$. First note that $\Gamma_P(I)$ is also a fixed point of Γ_P^2 because $\Gamma_P^2(\Gamma_P(I)) = \Gamma_P(\Gamma_P^2(I)) = \Gamma_P(I)$. Hence, $\Gamma_P(I) \subseteq J$ as J is the greatest fixed point. Then it can be shown that for all $n = 0, 1, 2, \ldots$, $J \subseteq \Gamma_P(\Gamma_P^{2n}(\emptyset))$. This implies that $J \subseteq \Gamma_P(I)$. Thus $J = \Gamma_P(I)$ holds. □

3 An Algorithm for Computing Stable Models

In this section we develop an algorithm for computing the stable model semantics of ground programs using a framework based on an approach introduced for default logic [Nie95]. The framework consists of a basic algorithm where two key functions are not given directly but they are specified by stating conditions that these functions have to satisfy in order to guarantee the soundness and completeness of the algorithm. The framework can be used to develop a range of algorithms for computing the stable model semantics by varying the implementations of the two functions. In order to prove correctness of the optimizations it is sufficient to establish that the implementations of the two function satisfy the given conditions. We first introduce the basic algorithm and then develop optimized implementations of the two functions.

The basic algorithm is actually a skeleton for a range of decision procedures for solving reasoning problems concerning the stable model semantics. By changing a certain test function, the same algorithm can be adapted to deciding whether a program has a stable model, computing all stable models of a program and handling two basic query-answering tasks: decide whether a given formula is satisfied in some or all of the stable models of a program. When describing the algorithm we use the following concepts.

- We say that a set of atoms S *agrees* with a set of literals Λ (i) if $\phi \in \Lambda^+$ implies $\phi \in S$ and (ii) $\text{not}(\phi) \in \Lambda^-$ implies $\phi \notin S$. For example, the set $\{a, b\}$ agrees with $\{a, \text{not}(c)\}$ but not with $\{a, \text{not}(b)\}$.
- A set of literals Λ *covers* an atom ϕ if either $\phi \in \Lambda$ or $\text{not}(\phi) \in \Lambda$. For example, the set $\{\text{not}(a), b\}$ covers a. A set of atoms S is covered by Λ if each atom in S is covered by Λ.

The basic algorithm is presented in Figure 1 as a function **sm**. It takes as input a set of rules P, a set B which represents the common part of the stable models to be considered and a formula ϕ which is just passed as an argument to the function **test**. The set B represents a partially constructed full set in the following way: B^- are those not-atoms already included in the full set and an atom $\phi \in B$ indicates that $\text{not}(\phi)$ cannot be included in the full set. The function **sm** performs *focused model search*: it returns true iff there is a stable model Δ of P *agreeing with* B such that $\textbf{test}(\Delta, \phi)$ returns true. This is accomplished by constructing P-full sets agreeing with B until a full set Λ is found such that for the corresponding stable model $\Delta = \text{Dcl}(P, \Lambda)$, $\textbf{test}(\Delta, \phi)$ returns true.

The function **sm** uses three additional functions: **expand** (for expanding B), **conflict** (for detecting conflicts), and **test** (for testing stable models). By changing the function **test** the various decision procedures are obtained. For example, to decide whether the program P has a stable model, it is sufficient to use a test function that always returns true. The test functions for the other reasoning task are explained below. For the first two functions we present the necessary requirements (E1,E2,C1,C2) that the implementations of these functions have to satisfy in order to guarantee the soundness and completeness of the decision procedures. These two functions form the crucial points in the algorithm where optimization techniques can be applied.

In order to guarantee soundness and completeness **expand** has to fulfill the following conditions where $B_n = \textbf{expand}(P, B)$.

E1: $B \subseteq B_n$.

function sm(P, B, ϕ)
$B_n :=$ **expand**(P, B);
if conflict(P, B_n) returns true **then**
    return false
else if NAnt(P) is covered by B_n **then**
    return **test**$(\mathrm{Dcl}(P, B_n), \phi)$
else
    take some $\chi \in \mathrm{NAnt}(P)$ not covered by B_n;
    if sm$(P, B_n \cup \{\mathrm{not}(\chi)\}, \phi)$ returns true **then**
        return true
    else
        return **sm**$(P, B_n \cup \{\chi\}, \phi)$
    endif
endif

Figure 1: The Skeleton for the Decision Procedures for the Stable Model Semantics

E2: Every stable model Δ of P agreeing with B agrees with B_n.

The function **conflict** is assumed to fulfill the following conditions.

C1: If B covers NAnt(P) and for some $\mathrm{not}(\chi) \in B$, $\chi \in \mathrm{Dcl}(P, B^-)$ or for an atom $\chi \in B$, $\chi \notin \mathrm{Dcl}(P, B^-)$, then **conflict**$(P, B)$ returns true.

C2: If **conflict**(P, B) returns true, then there exists no stable model Δ of P agreeing with B.

The function **sm** starts by expanding cautiously the set B. Condition (E1) ensures that B is really extended and (E2) guarantees that no stable models are lost. Then a conflict test is performed. Condition (C1) ensures that if all negative antecedents are already covered and there is an apparent conflict, it is detected and (C2) guarantees that if a conflict is reported, there are no stable models agreeing with B_n left. If there are no conflicts and B_n covers every negative antecedent, B_n^- is a full set for the stable model $\mathrm{Dcl}(P, B_n)$ and **sm** returns what **test**$(\mathrm{Dcl}(P, B_n), \phi)$ returns. If there is an atom $\chi \in \mathrm{NAnt}(P)$ not covered by B_n, then either $\mathrm{not}(\chi)$ belongs to the full set to be constructed or not. The two alternatives are handled by backtracking. First B_n is extended by $\mathrm{not}(\chi)$ but if **sm**$(P, B_n \cup \{\mathrm{not}(\chi)\}, \phi)$ returns false, $\mathrm{not}(\chi)$ cannot belong to the full set. Then **sm** returns what **sm**$(P, B_n \cup \{\chi\}, \phi)$ returns. It can be shown that **sm** is sound and complete.

Theorem 3.1 (Soundness) *Let P be a set of rules and B a set of literals. If* **sm**(P, B, ϕ) *returns true, then there exists a stable model Δ of P agreeing with B and for which* **test**(Δ, ϕ) *returns true.*

Proof. We prove the theorem by induction on the number of atoms in NAnt(P) but not covered by B. We denote this set by $\mathrm{NC}(P, B)$. Let **sm**(P, B, ϕ) return true. Then **conflict**(P, B_n) does not return true and if **test**$(\mathrm{Dcl}(P, B_n), \phi)$ is called, it returns true. Using (E1), (C1) and Theorem 2.2 we can first show that

$$\text{if } \mathrm{NC}(P, B_n) = \emptyset, \text{ then } \mathrm{Dcl}(P, B_n) \text{ is a stable model of } P \text{ agreeing with } B. \tag{2}$$

First we establish that B_n^- is P-full. Let $\chi \in \mathrm{NAnt}(P)$. If $\mathrm{not}(\chi) \in B_n^-$, then by (C1) $\chi \notin \mathrm{Dcl}(P, B_n^-)$. If $\mathrm{not}(\chi) \notin B_n^-$, then $\chi \in B_n^-$ as $\mathrm{NC}(P, B_n) = \emptyset$ and by (C1) $\chi \in \mathrm{Dcl}(P, B_n^-)$.

Hence, B_n^- is P-full and by Theorem 2.2 $\Delta = \mathrm{Dcl}(P, B_n^-)$ is a stable model of P. Now Δ agrees with B because (i) if $\chi \in B$, then $\chi \in B_n$ by (E1) and by (C1) $\chi \in \Delta = \mathrm{Dcl}(P, B_n^-)$ and (ii) if $\mathrm{not}(\chi) \in B$, then $\mathrm{not}(\chi) \in B_n$ by (E1) and by (C1) $\chi \notin \Delta = \mathrm{Dcl}(P, B_n^-)$. Furthermore, it can be shown that $\mathrm{Dcl}(P, B_n^-) = \mathrm{Dcl}(P, B_n)$. Hence, (2) holds.

Let $\mathrm{NC}(P, B) = \emptyset$. By (E1) $\mathrm{NC}(P, B_n) = \emptyset$. Thus, by (2) there exists a stable model $\Delta = \mathrm{Dcl}(P, B_n)$ of P agreeing with B and $\mathbf{test}(\Delta, \phi)$ returns true.

Let $\mathrm{NC}(P, B) \neq \emptyset$. If $\mathrm{NC}(P, B_n) = \emptyset$, then there exists a stable model Δ of P agreeing with B such that $\mathbf{test}(\Delta, \phi)$ returns true by (2). If $\mathrm{NC}(P, B_n) \neq \emptyset$, then as $\mathbf{sm}(P, B, \phi)$ returns true, some $\chi \in \mathrm{NC}(P, B_n)$ is chosen and either $\mathbf{sm}(P, B_n \cup \{\mathrm{not}(\chi)\}, \phi)$ or $\mathbf{sm}(P, B_n \cup \{\chi\}, \phi)$ returns true. As $\mathrm{NC}(P, B_n \cup \{\mathrm{not}(\chi)\}) = \mathrm{NC}(P, B_n \cup \{\chi\}) \subset \mathrm{NC}(P, B_n) \subseteq \mathrm{NC}(P, B)$ holds, then by the inductive hypothesis there exists a stable model Δ of P agreeing with $B_n \cup \{\mathrm{not}(\chi)\}$ or $B_n \cup \{\chi\}$. By (E1) Δ agrees with B, as well. □

Theorem 3.2 (Completeness) *Let P be a set of rules and B a set of literals. If there exists a stable model Δ of P agreeing with B and for which $\mathbf{test}(\Delta, \phi)$ returns true, then $\mathbf{sm}(P, B, \phi)$ returns true.*

Proof. We prove the theorem by induction on the number of atoms in $\mathrm{NAnt}(P)$ but not covered by B. We denote this set by $\mathrm{NC}(P, B)$.

Let there exist a stable model Δ of P agreeing with B and for which $\mathbf{test}(\Delta, \phi)$ returns true. By (E2) Δ agrees with B_n and by (C2) $\mathbf{conflict}(P, B_n)$ does not return true. Using (E2) and Theorem 2.2 we can establish first that

$$\text{if } \mathrm{NC}(P, B_n) = \emptyset, \text{ then } \mathbf{sm}(P, B, \phi) \text{ returns true.} \tag{3}$$

This can be proved by showing that $\Delta = \mathrm{Dcl}(P, B_n)$ holds. We start by establishing

$$\mathrm{not}(\mathrm{NAnt}(P) - \Delta) = B_n^-. \tag{4}$$

If $\chi \in \mathrm{NAnt}(P) - \Delta$, then as by (E2) Δ agrees with B_n, $\chi \notin B_n$. Then $\mathrm{not}(\chi) \in B_n$ since $\mathrm{NC}(P, B_n) = \emptyset$. If $\mathrm{not}(\chi) \in B_n$, $\chi \in \mathrm{NAnt}(P) - \Delta$ by (E2). Hence, (4) holds. By Theorem 2.2 $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta))$ holds. Hence, $\Delta = \mathrm{Dcl}(P, B_n^-)$ holds which implies that $\Delta = \mathrm{Dcl}(P, B_n)$ holds. Thus, (3) holds.

Let $\mathrm{NC}(P, B) = \emptyset$. By (E1) $\mathrm{NC}(P, B_n) = \emptyset$. Thus, by (3) $\mathbf{sm}(P, B, \phi)$ returns true. Assume $\mathrm{NC}(P, B) \neq \emptyset$. If $\mathrm{NC}(P, B_n) = \emptyset$, then $\mathbf{sm}(P, B, \phi)$ returns true by (3). If $\mathrm{NC}(P, B_n) \neq \emptyset$, then some $\chi \in \mathrm{NC}(P, B_n)$ is chosen. But Δ agrees with either $B_n \cup \{\mathrm{not}(\chi)\}$ or $B_n \cup \{\chi\}$, Since $\mathrm{NC}(P, B_n \cup \{\mathrm{not}(\chi)\}) = \mathrm{NC}(P, B_n \cup \{\chi\}) \subset \mathrm{NC}(P, B)$ holds, by the inductive hypothesis either $\mathbf{sm}(P, B_n \cup \{\mathrm{not}(\chi)\}, \phi)$ or $\mathbf{sm}(P, B_n \cup \{\chi\}, \phi)$ returns true. Thus, $\mathbf{sm}(P, B, \phi)$ returns true. □

The theorems imply that decision procedures for the different reasoning tasks can be obtained by (i) taking the empty set as the common part of the stable models to be constructed and (ii) choosing an appropriate **test** function as shown in the following corollary.

Corollary 3.3 *Let P be a set of rules and ϕ an atom.*

1. Let $\mathbf{test}(\Delta, \phi)$ return true for all Δ and ϕ. Then $\mathbf{sm}(P, \emptyset, \phi)$ returns true iff P has a stable model.

2. Let $\mathbf{test}(\Delta, \phi)$ *return true iff* ϕ *is satisfied by* Δ*. Then* $\mathbf{sm}(P, \emptyset, \phi)$ *returns true iff there exists a stable model of* P *satisfying* ϕ*.*

3. Let $\mathbf{test}(\Delta, \phi)$ *return true iff* ϕ *is not satisfied by* Δ*. Then* $\mathbf{sm}(P, \emptyset, \phi)$ *returns true iff there exists a stable model of* P *not satisfying* ϕ *(and, in other words,* $\mathbf{sm}(P, \emptyset, \phi)$ *returns false iff* ϕ *is satisfied by every stable model of* P*).*

It is quite straightforward to devise implementations of **expand** and **conflict** satisfying conditions E1–E2, C1–C2. For example, the following would do:

- Let $\mathbf{expand}(P, B)$ return $B \cup \{\chi \in \mathrm{NAnt}(P) \mid \chi \text{ not covered by } B, \chi \in \mathrm{Dcl}(P, B^-)\}$.
- Let $\mathbf{conflict}(P, B)$ return true iff B covers $\mathrm{NAnt}(P)$ and there exists $\mathrm{not}(\chi) \in B$ such that $\chi \in \mathrm{Dcl}(P, B^-)$ or an atom $\chi \in B$ such that $\chi \notin \mathrm{Dcl}(P, B^-)$.

Example 1 Let us consider an example that illustrates the basic algorithm. Let

$$P = \{\ a \leftarrow c, \mathrm{not}(b);\ b \leftarrow c, \mathrm{not}(a);\ c \leftarrow \mathrm{not}(a)\}. \tag{5}$$

We use the decision procedure to determine whether P has a stable model satisfying $a \wedge c$. Hence, we use a **test** function such that $\mathbf{test}(\Delta, \phi)$ returns true iff ϕ is satisfied by Δ and take the empty set as the initial common part of the stable models to be considered. So we consider the execution of $\mathbf{sm}(P, \emptyset, a \wedge c)$.

First $\mathbf{expand}(P, \emptyset)$ returns $\emptyset$ and $\mathbf{conflict}(P, \emptyset)$ returns false. Since $\mathrm{NAnt}(P) = \{a, b\}$ is not covered by $\emptyset$, we must choose either a or b in order to continue. Let us take a. Then $\mathbf{sm}(P, \{\mathrm{not}(a)\}, a \wedge c)$ is executed.

Now $\mathbf{expand}(P, \{\mathrm{not}(a)\})$ returns $\{\mathrm{not}(a), b\}$ and $\mathbf{conflict}(P, \{\mathrm{not}(a), b\})$ returns false. Because $\mathrm{NAnt}(P)$ is covered by $\{\mathrm{not}(a), b\}$, we have found a stable model $\mathrm{Dcl}(P, \{\mathrm{not}(a), b\}) = \{b, c\}$ of P but $\mathbf{test}(\{b, c\}, a \wedge c)$ returns false. Hence, $\mathbf{sm}(P, \{a\}, a \wedge c)$ is executed.

Now $\mathbf{expand}(P, \{a\})$ returns $\{a\}$, $\mathbf{conflict}(P, \{a\})$ returns false and $\mathbf{sm}(P, \{a, \mathrm{not}(b)\}, a \wedge c)$ is executed. Now $\mathbf{expand}(P, \{a, \mathrm{not}(b)\})$ returns $\{a, \mathrm{not}(b)\}$. Then $\mathbf{conflict}(P, \{a, \mathrm{not}(b)\})$ returns true because $a \notin \mathrm{Dcl}(P, \{\mathrm{not}(b)\})$ and $\mathbf{sm}(P, \{a, b\}, a \wedge c)$ is called. Now $\mathbf{expand}(P, \{a, b\})$ returns $\{a, b\}$. Then $\mathbf{conflict}(P, \{a, b\})$ returns true as $a \notin \mathrm{Dcl}(P, \{\})$. Hence, $\mathbf{sm}(P, \{b, a\}, a \wedge c)$ returns false and finally also $\mathbf{sm}(P, \emptyset, a \wedge c)$. Thus, no stable model of P satisfies $a \wedge c$. ■

Already the given implementations lead to quite an efficient decision method for the stable model semantics. However, we aim to develop further optimized implementations of the two functions where the search space of remaining stable models is more effectively pruned. The idea is to strengthen **expand** so that it extends the common part of the models as far as possible without loosing any models. Hence, the number of choice points for the backtracking search is reduced. On the other hand, detecting conflicts as early as possible prunes the search space effectively. There is an interesting *trade-off*: the work spent in **expand** and **conflict** saves work in the remaining backtracking search. Our solution to the problem of dividing the work between the two functions and the search is that we spend polynomial time (actually quadratic time) in the two functions for each recursive call and the rest of the work is done by the backtracking search.

We start by optimizing the function **expand**. First we note that we can use the whole set B (instead of just B^-). We can make the deductive closure even stronger without compromising the property (E2) by employing a Fitting operator F_P. We define $\mathrm{F}_P(B)$ to be the smallest set S containing B such that

function expand(P, B)
repeat
  $B' := B$;
  $B := B \cup \{\phi \in \mathrm{Dcl}^{\mathrm{F}}(P, B) \mid$ the atom in ϕ is in $\mathrm{NAnt}(P)$ but not covered by $B\}$;
  $B := B \cup \{\mathrm{not}(\phi) \mid \phi \in \mathrm{NAnt}(P), \phi$ is not covered by B and $\phi \notin \mathrm{Dcl}(P^B, \emptyset)\}$;
  Mark all new literals added to B (i.e. in $B - B'$)
until $B = B'$;
return B.

function conflict(P, B)
if for some unmarked $\mathrm{not}(\phi) \in B$,
    $\phi \in \mathrm{Dcl}^{\mathrm{F}}(P, B)$ or
    for some unmarked atom $\phi \in B$,
    $\phi \notin \mathrm{Dcl}(P^B, \emptyset)$ **then**
  return true
else
  return false.

Figure 2: Functions **expand** and **conflict**

1. if $c \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m) \in P$ and $\{a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m)\} \subseteq S$, then $c \in S$ and

2. if for each rule with c as the head in P, some atom a is in the body but $\mathrm{not}(a) \in S$ or some $\mathrm{not}(b)$ is in the body but $b \in S$, then $\mathrm{not}(c) \in S$.

Now F_P is monotonic for any program P and we define $\mathrm{Dcl}^{\mathrm{F}}(P, B)$ to be the least fixed point of the series $S_0 = B$ and for all $i > 0$, $S_i = \mathrm{F}_P(S_{i-1})$. Notice that $\mathrm{Dcl}^{\mathrm{F}}(P, B)$ is a superset of $\mathrm{Dcl}(P, B)$ but often considerably larger. For example, let P be as in (5) and $B = \{a\}$. Then $\mathrm{Dcl}(P, B) = \{\}$ but $\mathrm{Dcl}^{\mathrm{F}}(P, B) = \{a, \mathrm{not}(c), \mathrm{not}(b), \mathrm{not}(a)\}$.

Second, we notice that we can also extend B by not-atoms by borrowing an idea from the well-founded computation and still satisfy condition (E2): we can add $\mathrm{not}(b)$ if $b \notin \Gamma_P(B^+) = \mathrm{Dcl}(P^{B^+}, \emptyset)$. In fact, we can further optimize this by exploiting the not-atoms in B. For this we generalize the reduct P^S of a program P to the case where S can contain also not-atoms and use P^B instead of P^{B^+}. The idea is that if we are interested in a stable model not containing a, i.e. $\mathrm{not}(a) \in B$, then a rule with a in the body cannot be used for deriving any atom in the stable model. Hence, we define P^S, a reduct of P with respect to a set of literals S, as a set of rules obtained from P by deleting (i) each rule that has $\mathrm{not}(b)$ in its body but $b \in S$ or a in the body but $\mathrm{not}(a)$ in S and (ii) all $\mathrm{not}(l)$ in the bodies of the remaining rules. For example, let P be as in (5) and $B = \{\mathrm{not}(c)\}$. Then $P^B = \{\ c \leftarrow\}$ but $P^{B^+} = \{\ a \leftarrow c;\ b \leftarrow c;\ c \leftarrow\}$.

Similar considerations are applicable when implementing **conflict**. The resulting implementations of **expand** and **conflict** are presented in Figure 2. Obviously, (E1) is satisfied by **expand** in Figure 2 and the following two propositions imply that also (E2) is satisfied.

Proposition 3.4 *If a stable model Δ of P agrees with B, then it agrees with* $\mathrm{Dcl}^{\mathrm{F}}(P, B)$.

Proposition 3.5 *If a stable model Δ of P agrees with B, then* $\Delta \subseteq \mathrm{Dcl}(P^B, \emptyset)$.

The two propositions also imply that (C2) holds for the implementation of **conflict** in Figure 2. The condition (C1) is implied by the fact that (i) $\mathrm{Dcl}(P, B^-) \subseteq \mathrm{Dcl}(P, B) \subseteq \mathrm{Dcl}^{\mathrm{F}}(P, B)$ and (ii) if B covers $\mathrm{NAnt}(P)$, then $\mathrm{Dcl}(P^B, \emptyset) \subseteq \mathrm{Dcl}(P, B^-)$.

It turns out that the optimized implementation of **expand** can be seen as a generalization of well-founded model computation in the sense that when the initial part of the model is empty, **expand** computes the well-founded model of the program as shown in the next theorem. When the initial part of the model is not empty, **expand** computes a well-founded type approximation for all the stable models agreeing with the initial part.

Theorem 3.6 *Let* $B = \mathbf{expand}(P, \emptyset)$. *Then* $I = \mathrm{Dcl}(P, B) \cup \mathrm{not}(\overline{\mathrm{Dcl}(P^B, \emptyset)})$ *is the well-founded model of* P.

Proof. Let M be the well-founded model of P. First we note that for every $B' \subseteq M$,

$$\mathrm{Dcl}^{\mathrm{F}}(P, B') \subseteq M \tag{6}$$
$$\Gamma_P(M^+) \subseteq \mathrm{Dcl}(P^{B'}, \emptyset) \tag{7}$$

Now

$$B = \mathbf{expand}(P, \emptyset) \subseteq M \tag{8}$$

holds because initially $B = \emptyset$ and after each step where B is extended in **expand** condition (8) holds by (6) and (7). Hence, by (8), (6), and (7),

$$\mathrm{Dcl}(P, B) \subseteq \mathrm{Dcl}^{\mathrm{F}}(P, B) \subseteq M, \tag{9}$$
$$\Gamma_P(M^+) \subseteq \mathrm{Dcl}(P^B, \emptyset) \tag{10}$$

hold. To complete the proof we first establish that for all $i = 0, 1, \ldots$,

$$M_i^+ \cap \mathrm{NAnt}(P) \subseteq B \tag{11}$$

holds where $M_0^+ = \emptyset$ and $M_{i+1}^+ = \Gamma_P^2(M_i^+)$ for all $i > 0$.

Assume that $M_i^+ \cap \mathrm{NAnt}(P) \subseteq B$ holds. Then $\mathrm{Dcl}(P^B, \emptyset) \subseteq \Gamma_P(B^+) \subseteq \Gamma_P(M_i^+)$ holds. Let $\chi \in M_{i+1}^+ = \Gamma_P^2(M_i^+) = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Gamma_P(M_i^+)))$. Let $\mathrm{not}(b) \in \mathrm{not}(\mathrm{NAnt}(P) - \Gamma_P(M_i^+))$. Then $b \notin \Gamma_P(M_i^+)$ and, hence, $b \notin \mathrm{Dcl}(P^B, \emptyset)$. But then $\mathrm{not}(b) \in B$ and $\chi \in \mathrm{Dcl}^{\mathrm{F}}(P, B)$ which implies $\chi \in B$ if $\chi \in \mathrm{NAnt}(P)$. Hence, (11) holds.

Now by (11) $\mathrm{Dcl}(P^B, \emptyset) \subseteq \Gamma_P(B^+) \subseteq \Gamma_P(M^+)$ holds. Hence, by (10) $M^- = \Gamma_P(M^+) = \mathrm{Dcl}(P^B, \emptyset)$ holds. Then

$$\begin{aligned} M^+ = \Gamma_P^2(M^+) &= \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P)) - \Gamma_P(M^+)) \\ &= \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P)) - \mathrm{Dcl}(P^B, \emptyset)) \\ &\subseteq \mathrm{Dcl}(P, B). \end{aligned}$$

Hence, by (9) $M^+ = \mathrm{Dcl}(P, B)$. □

The algorithm in Figure 1 together with the implementations of **expand** and **conflict** in Figure 2 form the basis of our implementation. In order to obtain an efficient implementation two important issues must be addressed:

- implementations of the deductive closures $\mathrm{Dcl}^{\mathrm{F}}(P, B)$ and $\mathrm{Dcl}(P^B, \emptyset)$ and
- search heuristics

Implementations of the deductive closures are needed in the functions **expand** and **conflict** in Figure 2. In the next section we show that both of the closures can be computed in linear time for a ground program. This implies that **expand** can be implemented in quadratic time yielding a quadratic time implementation of the well-founded semantics for ground programs. Actually our implementation of **expand** frequently seems to have a linear time performance. This is

because the Fitting operator often yields (for programs without positive loops) the well-founded model in linear time and in these cases the other closure is not needed at all. For example, let

$$P = \{\ a_1 \leftarrow \mathrm{not}(a_0);\ a_2 \leftarrow \mathrm{not}(a_1); \ldots;\ a_n \leftarrow \mathrm{not}(a_{n-1})\}.$$

Now $\mathrm{Dcl}^{\mathrm{F}}(P, \emptyset) = \{\mathrm{not}(a_0), a_1, \mathrm{not}(a_2), a_3, \ldots\}$. Hence $\mathrm{NAnt}(P)$ is covered (in linear time) and the computation of $\mathrm{Dcl}(P^B, \emptyset)$ does not need to be invoked at all to compute the unique stable model (well-founded model) of P. This positive effect of the Fitting operator has also been pointed out in [SNV94, BSF95].

The used *search heuristic* has a major influence on the performance of the algorithm **sm** in Figure 1. The heuristic can be incorporated in the phase where there are atoms in $\mathrm{NAnt}(P)$ not yet covered by B_n and one of them has to be chosen in order to extend the full set under construction. In Section 6 we explain what kind of a heuristic we employ in the implementation.

4 Computing Deductive Closures

In this section we describe two linear time algorithms for computing the two deductive closures $\mathrm{Dcl}^{\mathrm{F}}(P, B)$ and $\mathrm{Dcl}(P^B, \emptyset)$ needed to implement the functions **expand**(P, B) and **conflict**(P, B). The deductive closures are calculated using a modified version of the first linear time algorithm of Dowling and Gallier [DG84] for computing the deductive closure of a set of Horn clauses. The modification is necessary as the rules contain, apart from ordinary atoms, also not-atoms. In addition, we exploit the fact that the set $\mathrm{R}(P, B)$ diminish or grow when the set of literals B diminish or grow to compute the deductive closure $\mathrm{Dcl}^{\mathrm{F}}(P, B)$ incrementally.

We assume that the rules and atoms are stored in arrays. In order to guarantee constant access times we let the atoms and rules be indices into the arrays. If r is the rule $h \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m)$, we let *r.head* denote the head of the rule, i.e., the atom h. In addition, we join three counters *poslit*, *neglit*, and *inactive* to each rule. The first two counters are initially initialized with the number of respectively atoms and not-atoms in the body of the rule and the counter *inactive* is initialized with a zero. For each atom a we initialize two lists *a.poslitlist* and *a.neglitlist* with all rules whose bodies contain the atom a as respectively a positive and a negative literal. Similarly, for each atom a we initialize a list *a.headlist* with all rules whose heads are a and a counter *a.headof* with the number of rules in *a.headlist*. An array *closure* is used to tell which literals belong to the closure. If $closure[a] = \mathrm{true}$ then the atom a belongs to the closure and if $closure[a] = \mathrm{false}$ then the literal $\mathrm{not}(a)$ belongs to the closure. We let $closure[a] = \mathrm{unknown}$ if none of the previous hold.

Since we know all possible members of the set B, we implement it as an array, as well. The elements of B are represented by indices into the array and the value of the array at index n tells if element n belongs to the set or not. If the value is false, the element does not belong to the set and if the value is true it does.

The algorithm that computes the deductive closure $\mathrm{Dcl}^{\mathrm{F}}(P, B)$ of a set of ground rules P and a set of literals B is described in Figure 3. The algorithm works by putting the atoms which belong to the closure as positive literals on the queue *posqueue* and which belong to the closure as negative literals on the queue *negqueue*. When an atom is removed from a queue and when it is not in the closure, the algorithm adds the atom to the closure and then puts the head of every newly activated rule on the queue *posqueue* and the head of every inactivated rule on the queue *negqueue*. A rule is activated when both of its counters *poslit* and *neglit* reach zero and

inactivated when its counter *inactive* goes from zero to one. In other words, an activated rule has its body in the closure and should therefore have its head in the closure. Similarly, since an inactivated rule cannot be activated, its head should be put as a not-atom in the closure. The counter *r.poslit* of a rule r holds the number of atoms in the body of r whose values in the closure array are not true and the counter *r.neglit* the number of not-atoms in the body whose values are not false. In order to compute the deductive closure $\mathrm{Dcl}^{\mathrm{F}}(P, B)$, we have to initialize the queue *posqueue* with the atoms in B^+ and the queue *negqueue* with the literals in B^-.

The algorithm computes the deductive closure linearly in the size of the set of rules P as an atom is added to a queue at most as many times as it is a head of a rule.

The algorithm can be used to compute the deductive closure incrementally. Suppose the algorithm has computed the closure $\mathrm{Dcl}^{\mathrm{F}}(P, B)$. Suppose we want to compute $\mathrm{Dcl}^{\mathrm{F}}(P, B')$ and suppose $B \subseteq B'$. In order to compute the larger closure starting from the smaller one, we only have to initialize the queue with the heads of the rules whose bodies are in B' and which are not in the closure. The algorithm will then compute the new closure when invoked.

A slightly different algorithm is needed if we want to compute the other closure $\mathrm{Dcl}(P^B, \emptyset)$. We assume that the closure array and the counters have been initialized by $\mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$. This initialization is only needed once, before the absolutely first call to $\mathrm{Dcl}(P^B, \emptyset)$. The queue is initialized with the heads of every newly inactivated rule, i.e., with the heads of the rules whose *inactive* counters went from zero to one in the latest computation of $\mathrm{Dcl}^{\mathrm{F}}(P, B)$. The idea of the algorithm is that instead of propagating the fact that an atom should belong to the closure, we propagate the fact that an atom should be removed from the closure. This is done by pushing atoms which should be removed from the deductive closure onto the queue $queue_2$. When an atom a is removed from the queue, we check if it is in the closure. If this is the case, then the atom a is removed from the closure and for each rule who has a in its body, the counter *poslit* is incremented. Finally, the heads of those active rules whose *poslit* counters are thus set to one and whose *inactive* counters are zero are added to the queue as they no longer imply that their heads should belong to the closure.

The problem with this approach is that too many atoms are possibly removed from the closure. When the algorithm removes an atom from the closure, it removes the atom even if the atom is the head of a rule which is active in the final closure. To remedy this problem we store the atoms removed from the closure in a temporary set N and then add the atoms wrongly removed to the closure. We can not abstain from removing atoms which are heads of active rules, as we do not know if the rules will still be active in the final closure.

The set B is used to determine which rules are active and which are not. A rule is active if none of its positive literals is covered by B^- and none of its negative literals cover B^+.

The part of the algorithm that propagates the fact that atoms should be removed from the closure works in linear time in the size of the set of rules P, as each atom is added to the queue at most once for every rule it is head of. The part that adds atoms which have been wrongly removed from the closure to the queue $queue_2$ is also linearly bounded by the size of P, as the size of N is bounded by the number of atoms, as the number of rules in the *headlist* lists are bounded by the number of rules in P, and as each atom is added at most once for every rule in which it is a head. Thus, the time complexity of the algorithm is linear in the size of P.

```
function DclF(posqueue, negqueue, P)
while neither queue is empty do
    while posqueue is not empty do
        a := pop(posqueue)
        if closure[a] = unknown then
            closure[a] := true
            for each rule r ∈ a.poslitlist do
                r.poslit := r.poslit − 1
                if r.poslit = 0 and r.neglit = 0
                    and r.inactive = 0 then
                    push(r.head, posqueue)
                endif
            end for
            for each rule r ∈ a.neglitlist do
                r.inactive := r.inactive + 1
                if r.inactive = 1 then
                    b := r.head
                    b.headof := b.headof − 1
                    if b.headof = 0 then
                        push(b, negqueue)
                    endif
                endif
            end for
        endif
    end while
    while negqueue is not empty do
        a := pop(negqueue)
        if closure[a] = unknown then
            closure[a] := false
            for each rule r ∈ a.neglitlist do
                r.neglit := r.neglit − 1
                if r.poslit = 0 and r.neglit = 0
                    and r.inactive = 0 then
                    push(r.head, posqueue)
                endif
            end for
            for each rule r ∈ a.poslitlist do
                r.inactive := r.inactive + 1
                if r.inactive = 1 then
                    b := r.head
                    b.headof := b.headof − 1
                    if b.headof = 0 then
                        push(b, negqueue)
                    endif
                endif
            end for
        endif
    end while
end while
```

```
function Dcl (queue_2, P)
N := ∅
while queue_2 is not empty do
    a := pop(queue_2)
    if closure[a] = true then
        closure[a] := unknown
        N := N ∪ {a}
        for each rule r ∈ a.poslitlist do
            r.poslit := r.poslit + 1
            if r.poslit = 1 and r.inactive = 0 then
                push(r.head, queue_2)
            endif
        end for
    endif
end while
for each atom a ∈ N do
    if there exists a rule r ∈ a.headlist
        such that r.poslit = 0,
        and r.inactive = 0 then
        push(r, head, queue_2)
    endif
end for
while queue_2 is not empty do
    a := pop(queue_2)
    if closure[a] = unknown then
        closure[a] := true
        for each rule r ∈ a.poslitlist do
            r.poslit := r.poslit − 1
            if r.poslit = 0 and r.inactive = 0 then
                push(r.head, queue_2)
            endif
        end for
    endif
end while
```

Figure 3: The Procedures for Computing the Deductive Closures

5 Handling Rules with Variables

In this section we develop a method to compute a grounded version P' of a range-restricted function-free program P such that P and P' have exactly the same stable models agreeing with a given set of literals B but P' is often considerably smaller than the whole set of ground instances of P. We first define the set P' and then present an algorithm for computing it directly from P without considering all of its ground instances. To define P' we first let

$$A = B \cup B_n \cup B_p$$

where $B_n = \{\mathrm{not}(c) \mid c \in \mathrm{NAnt}(P_\mathrm{g}) - B$ and there is no rule in P having the same predicate as c in the head $\}$ and $B_p = \{c \mid$ there is a ground rule in P with c as its head and all literals in the body in $B_n \cup B\}$. Then let

$$A' = \mathrm{Dcl}(P_\mathrm{g}, A \cup \mathrm{not}(\mathrm{NAnt}(P_\mathrm{g}) - A))$$

and finally let

$$\begin{aligned} P' \;=\; & \{\, c \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m) \in P_\mathrm{g} \mid \\ & \{a_1, \ldots, a_n\} \subseteq A', \{b_1, \ldots, b_m\} \cap A = \emptyset\}. \end{aligned} \tag{12}$$

Figure 4 presents an algorithm that computes P' directly from P and A without grounding P first when P is a range-restricted function-free program. A program is range-restricted when for each rule holds that if a variable appears in the head or in a not-atom in the body, it also appears in a positive literal in the body. When using P' instead of the whole P_g no stable models are lost. Notice that when invoking the algorithm to compute the stable models for the grounded version P' we can give as the initial part of the stable models to be considered not only B but $A \cup \mathrm{not}(\mathrm{NAnt}(P') - A')$.

Proposition 5.1 *Δ is a stable model of P_g agreeing with B iff Δ is a stable model of P' agreeing with $A \cup \mathrm{not}(\mathrm{NAnt}(P') - A')$.*

Proof. First we present two facts which are straightforward to establish.

For any stable model Δ of P_g agreeing with B (of P' agreeing with $A \cup \mathrm{not}(\mathrm{NAnt}(P') - A')$),

$$\Delta \text{ agrees with } A, \tag{13}$$

$$\Delta \subseteq A'. \tag{14}$$

($\Rightarrow$) Let Δ be a stable model of P_g agreeing with B. We show that

$$\mathrm{Dcl}(P_\mathrm{g}, \mathrm{not}(\overline{\Delta})) = \mathrm{Dcl}(P', \mathrm{not}(\overline{\Delta})) \tag{15}$$

holds which is implies that Δ be a stable model of P'. By (13) and (14) Δ agrees also with $A \cup \mathrm{not}(\mathrm{NAnt}(P') - A')$.

As $P' \subseteq P_\mathrm{g}$, $\mathrm{Dcl}(P', \mathrm{not}(\overline{\Delta})) \subseteq \mathrm{Dcl}(P_\mathrm{g}, \mathrm{not}(\overline{\Delta}))$ holds. Let $\chi \in \mathrm{Dcl}(P_\mathrm{g}, \mathrm{not}(\overline{\Delta}))$. Then there is a proof of χ from $\mathrm{R}(P_\mathrm{g}, \mathrm{not}(\overline{\Delta}))$. Assume that a rule

$$r = \; c \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m) \in P_\mathrm{g}$$

```
function ground(P, A)
P_2 := {r ∈ P | the positive literals of the body of r is in
    A and the negative literals are in A^- or not in not(A^+)}
P := P − P_2; U := ∅;
N := A^+ ∪ {a | there exists a rule r ∈ P_2 whose head is a}
while N is not empty do
    Choose a ∈ N; N := N − {a}; U := U ∪ {a};
    for each rule r in P do
        for every rule r' resulting from applying a substitution
            to r such that a is in the body of the rule r' do
            if no negative literal in the body of r' is in not(A^+) then
                if r' is grounded and all positive literals
                    in the body of r' is in N ∪ U then
                    P_2 := P_2 ∪ {r'}
                    if the head of r' is not in U then
                        N := N ∪ {the head of r'}
                    endif
                else
                    P := P ∪ {r'}
                endif
            endif
        end for
    end for
end while
return P_2
```

Figure 4: A Procedure for Grounding Range Restricted Programs

is needed in the proof. Then for all a_i in the body, $a_i \in \Delta$ and for all b_j in the body, $b_j \notin \Delta$. By (13) $b_j \notin A$ and by (14) $a_i \in A'$. Hence, $r \in P'$ and $\mathrm{Dcl}(P_g, \mathrm{not}(\overline{\Delta})) \subseteq \mathrm{Dcl}(P', \mathrm{not}(\overline{\Delta}))$. Thus (15) holds.

($\Leftarrow$) Let Δ be a stable model of P_g agreeing with $A \cup \mathrm{not}(\mathrm{NAnt}(P') - A')$. Similarly, (13) and (14) imply that (15) holds. Hence, Δ is a stable model of P_g agreeing with B. □

Example 2 Consider a range-restricted program

$$\begin{aligned} P \;=\; & \{\, q(a) \leftarrow;\; s(b) \leftarrow;\; p(a) \leftarrow \mathrm{not}(r(a));\\ & \quad t(X) \leftarrow q(X), \mathrm{not}(p(X));\; q(X) \leftarrow s(X), \mathrm{not}(t(X))\}. \end{aligned}$$

Let $B = \{t(b)\}$. Now the grounding $P' = \{\, q(a) \leftarrow;\; s(b) \leftarrow;\; p(a) \leftarrow \mathrm{not}(r(a))\}$. Observe that, e.g., $t(a) \leftarrow s(a), \mathrm{not}(p(a))$ is not included because $p(a) \in A$ and $t(b) \leftarrow q(b), \mathrm{not}(p(b))$ is not included because $q(b) \notin A'$. ■

6 Implementation

We have implemented the **sm** algorithm in C++ and the resulting implementation of the stable model and well-founded semantics can be obtained from `ftp://saturn.hut.fi/pub/smodels/`. In this section we describe the implementation specific choices we made to ensure an efficient

implementation of the **sm** algorithm. In particular, this concerns the detection of conflicts, and the way backtracking is handled. As the algorithm contains a non-deterministic choice, we describe a heuristic for making a hopefully good one.

The order in which the atoms are picked from $\mathrm{NAnt}(P)$ drastically affects the performance of the stable model algorithm. If only atoms that appear in a specific stable model are chosen, the algorithm finds the first stable model without backtracking. It is therefore important to use a good heuristic when selecting atoms.

We employ a heuristic which selects a not-atom from the body of an applicable rule with the least number of not-atoms not in B^- in its body. An applicable rule is a rule that might change the set B, i.e., the head of an applicable rule is not in $\mathrm{Dcl}^{\mathrm{F}}(P, B)$, all of the positive literals in the body of the rule are in the closure, and none of the negative literals cover any atom in B^+. Thus, the heuristic tries to select the atom that has the greatest possibility of changing B, thereby helping the algorithm to find conflicts or full sets with a minimal number of choices. If there are no applicable rules left, the atoms in $\mathrm{NAnt}(P)$ not covered by B cannot affect the deductive closures. Therefore, they are immediately added as not-atoms to B^-.

The heuristic can be implemented by examining the rules that have not-atoms in their bodies that are not covered by B. The counter *poslit* provides an easy way of detecting whether all atoms in the body of a rule are in $\mathrm{Dcl}^{\mathrm{F}}(P, B)$. Hence, computing the heuristic is linear in the number of rules in the logic program in the worst case.

We want to reduce the amount of computation as much as possible. Therefore, we make the conflict test each time an atom or not-atom is added to B in the **expand** function. This incurs very little overhead as the deductive closures are already calculated when the literal is added to the set B and as we only have to test the literal being added. Thus, we gain the ability to interrupt the computations of the deductive closures and the **expand** function as soon as a conflict is detected.

The algorithm **sm** handles backtracking by recursively passing along the changing set B as one of its parameters. This practice would in an implementation require a lot of memory. To avoid this inconvenience we use iteration instead of recursion and use a stack to record the information necessary for backtracking. All that has to be stored is the changing state of the two deductive closures that are used in the **expand** function. The two arrays reflect which literals have been added to B and how the counters of the rules have been incremented or decremented. When the algorithm is backtracking the correct state is recreated.

7 Experimental Results

We have tested our implementation extensively. To get challenging test cases we have employed combinatorial graph problems as a test domain. Test cases from the literature and from the domain of diagnostics have also been used. All test cases can be obtained from `ftp://saturn.hut.fi/pub/smodels/tests`. In this section we report some of the test results and compare the performance of our implementation of the stable model semantics with the SLG system [CW94, CW93] which is a state of the art implementation of the stable model semantics.

We compare the algorithm **sm** and the SLG system by using the two algorithms to search for 3-colorings and Hamiltonian circuits in different graphs. The graphs were created using

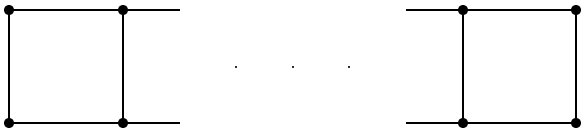

Figure 5: Ladder Graph

the Stanford GraphBase [Knu93], which is a highly portable collection of programs and data that serves as a platform for combinatorial algorithms. In addition, two benchmark programs in [CW94] and the win-move example [SNV94] are used. Furthermore, **sm** has been tested on two diagnosis problems: a ripple carry adder and a parity circuit.

Three types of graphs have been used in the tests. A special type of planar graph, created by the *plane* function found in the Stanford GraphBase [Knu93], a random graph created by the *random_graph* function, and a 'ladder'-graph, see Figure 5. The planar graphs are constructed by Delaunay triangulation of randomly inserted points in a plane. The test cases are generated from the graphs by translating the graphs into ground programs such that the stable models of the programs provide the solutions to the graph problems.

The 3-coloring problem is translated from a graph into a logic program in the following way. For each vertex a with neighbors $p_1, \ldots, p_j$ and each color $i \in \{0, 1, 2\}$, include the rule

$$\begin{array}{rcl} color(a,i) & \leftarrow & \mathrm{not}(color(p_1,i)), \ldots, \mathrm{not}(color(p_j,i)), \\ & & \mathrm{not}(color(a, i+1 \bmod 3)), \\ & & \mathrm{not}(color(a, i+2 \bmod 3)) \end{array}$$

and for each vertex a include the rule

$$h \leftarrow \mathrm{not}(color(a,0)), \mathrm{not}(color(a,1)), \mathrm{not}(color(a,2)).$$

Finally add the rules $h_2 \leftarrow \mathrm{not}(h)$ and $h_2 \leftarrow \mathrm{not}(h_2)$. The number of rules is thus $4V + 2$ if V is the number of vertices.

The translation of the Hamiltonian circuit problem is somewhat more complicated. For each pair (a, b) of vertices in the graph such that there is an edge between a and b, and where $b_1, \ldots, b_i$ are the neighbors of a excluding b, and $a_1, \ldots, a_j$ the neighbors of b excluding a, include the rule

$$\begin{array}{rcl} edge(b,a) & \leftarrow & \mathrm{not}(edge(b_1,a)), \ldots, \mathrm{not}(edge(b_i,a)), \\ & & \mathrm{not}(edge(b,a_1)), \ldots, \mathrm{not}(edge(b,a_j)), \\ & & \mathrm{not}(edge(a,b)). \end{array}$$

Fix a vertex d. For each pair (a, b) of vertices in the graph such that there is an edge between a and b, if $b \neq d$ include the rule

$$b \leftarrow a, edge(a,b)$$

and if $b = d$ include the rule

$$d' \leftarrow a, edge(a,d).$$

Finally, for each vertex $a \neq d$ add $a \leftarrow \mathrm{not}(a)$, and then add the rules $d \leftarrow$ and $d' \leftarrow \mathrm{not}(d')$. If the number of vertices is denoted by V and the number of edges by E, the number of rules in this problem is $4E + V + 1$.

The win-move examples consists of the rule

$$win(X) \leftarrow move(X,Y), \mathrm{not}(win(Y))$$

$$
\begin{array}{ll}
z1(X) \leftarrow v1(X), w1(X) & z2(X) \leftarrow v1(X), w2(X) \\
z3(X) \leftarrow v2(X), w1(X) & z4(X) \leftarrow v2(X), w2(X) \\
v1(X) \leftarrow s(X) & v2(X) \leftarrow t(X) \\
w1(X) \leftarrow p(X) & w2(X) \leftarrow q(X) \\
t(X) \leftarrow \mathrm{not}(s(X)) & s(X) \leftarrow \mathrm{not}(t(X)) \\
p(X) \leftarrow \mathrm{not}(q(X)) & q(X) \leftarrow \mathrm{not}(p(X)) \\
m(X) \leftarrow z1(X) & m(X) \leftarrow z2(X) \\
m(X) \leftarrow z3(X) & m(X) \leftarrow z4(X)
\end{array}
$$

Figure 6: SLG Benchmark 1

$$
\begin{array}{ll}
s(X) \leftarrow p(X), q(X) & s(X) \leftarrow p(X), r(X) \\
s(X) \leftarrow q(X), r(X) & p(X) \leftarrow \mathrm{not}(q(X)) \\
q(X) \leftarrow \mathrm{not}(r(X)) & r(X) \leftarrow \mathrm{not}(p(X))
\end{array}
$$

Figure 7: SLG Benchmark 2

together with some rules of the form $move(\cdot,\cdot) \leftarrow$. The *move* rules represents the moves in a game and are chosen from random graphs with a varying number of vertices and edges.

Since the SLG system is goal-directed we must add some rules to the translations of the logic circuits and the graph problems to make the system compute the whole stable model. We add the rule $s \leftarrow d'$ to the Hamiltonian circuit problem and for each vertex a and each color i we add the rule $s \leftarrow color(a, i)$ to the 3-coloring problem. Finally, for every vertex a in the random graphs we add the rule $s \leftarrow win(a)$ to the win-move examples and for each atom ab that determines whether a gate is abnormal we add a rule $s \leftarrow ab$ to the diagnosis problem. The resulting set of rules is then used when testing the SLG system, using the goal s, and the algorithm **sm**.

We have also used the two benchmark programs found in [CW94] to generate test cases, see Figures 6–7. Before the stable models are computed a number of constants are introduced and the program is grounded, i.e., the variables in the rules are instantiated with the constants. If the number of constants is n, the number of stable models of the first benchmark is 2^{2n}. The second benchmark has no stable models.

Both benchmark programs are very simple as there are few interdependencies between the ground atoms. This fact is clearly reflected in the timing results.

The circuit diagrams of the ripple carry adder and the parity circuit is shown in Figure 8. We set all inputs of the ripple carry adder and the parity circuit to zero. In order to get diagnoses we set the last output of the adder and the output of the parity circuit to one.

Given a description and some observations of a system, a diagnosis is a conjecture that some minimal set of components are faulty, the rest being normal, that explains the behavior of the system, i.e., that explains the discrepancy between the observations and the correct behavior of the system [Rei87]. We use this definition to encode logic gates, their connections, and their inputs and outputs as logic programs in such a way that each diagnosis determines at least one stable model and that each stable model determines one diagnosis.

We proceed with the encoding of the not-, xor-, and- and or-gates. The not- and xor-gates are easily encoded as three sets of rules. The first set encodes the normal behavior of the gate from the input to the output side. The second set encodes the normal behavior from the output to the

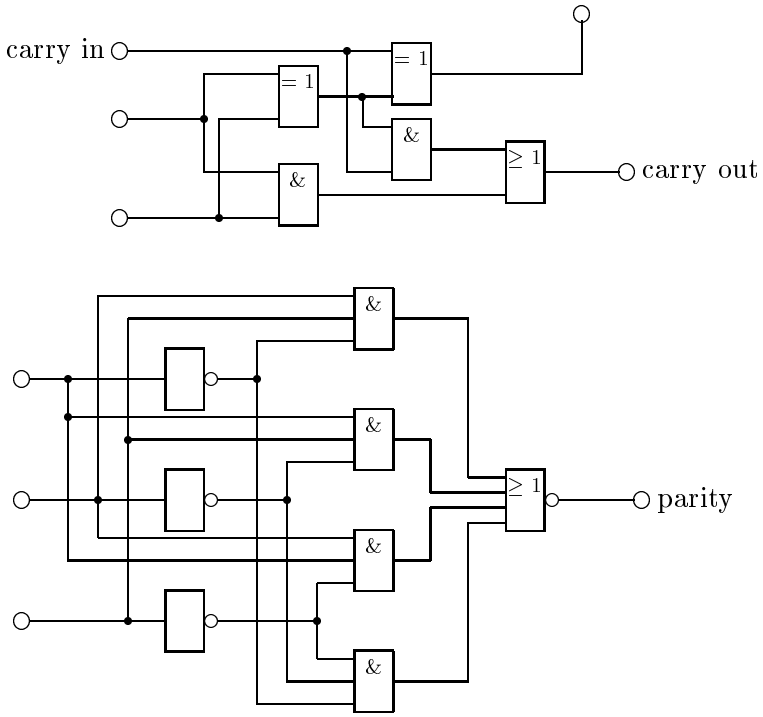

Figure 8: Ripple Carry Adder and Parity Circuit

input side. Finally, the third set encodes input and output combinations that lead to abnormal behavior of the gate. The rules corresponding to the not-gate and the xor-gate are presented in Figures 9 and 10. We use a notation containing the variable X. The variable X should be replaced by a constant representing a component of the logic circuit in an actual encoding. The resulting construct then denotes a ground atom. For example, if xor_1 is a constant then $ab(xor_1)$ is a ground atom that determines whether the component xor_1 is abnormal. The rules are the encoding of a system description where negation is replaced by *not* and the $\leftarrow$ symbol replaces the implication sign.

We encode the and- and or-gates in a similar way to the not- and xor-gates. However, we have to add some rules as some outputs of these two gates does not uniquely determine their inputs. The problem lies in that we would need to use disjunction to encode the and- and or-gates correctly. For instance, if the output of an and-gate is zero, the first or the second input should be zero. We simulate the disjunction by using two rules to determine two different stable models in which one or the other of the inputs is zero. This technique has its drawbacks, however, as it in some cases leads to multiple stable models per diagnosis. The and- and or-gates are described in Figures 11 and 12.

The interconnections between different gates are encoded using four rules as follows. Suppose the first input of the and-gate and_1 and the output of the or-gate or_1 are connected. This is encoded using the rules

$$\{\ out(or_1, 0) \leftarrow in1(and_1, 0);\quad out(or_1, 1) \leftarrow in1(and_1, 1);$$
$$in1(and_1, 0) \leftarrow out(or_1, 0);\quad in1(and_1, 1) \leftarrow out(or_1, 1)\}.$$

Forward

$out(X,1) \leftarrow nt(X), in(X,0), \text{not}(ab(X))$
$out(X,0) \leftarrow nt(X), in(X,1), \text{not}(ab(X))$

Backward

$in(X,1) \leftarrow nt(X), out(X,0), \text{not}(ab(X))$
$in(X,0) \leftarrow nt(X), out(X,1), \text{not}(ab(X))$

Abnormal

$ab(X) \leftarrow nt(X), in(X,0), out(X,0)$
$ab(X) \leftarrow nt(X), in(X,1), out(X,1)$

Figure 9: Not-gate

Forward

$out(X,0) \leftarrow xor(X), in1(X,0), in2(X,0), \text{not}(ab(X))$
$out(X,1) \leftarrow xor(X), in1(X,0), in2(X,1), \text{not}(ab(X))$
$out(X,1) \leftarrow xor(X), in1(X,1), in2(X,0), \text{not}(ab(X))$
$out(X,0) \leftarrow xor(X), in1(X,1), in2(X,1), \text{not}(ab(X))$

Backward

$in1(X,0) \leftarrow xor(X), in2(X,0), out(X,0), \text{not}(ab(X))$
$in1(X,1) \leftarrow xor(X), in2(X,0), out(X,1), \text{not}(ab(X))$
$in1(X,1) \leftarrow xor(X), in2(X,1), out(X,0), \text{not}(ab(X))$
$in1(X,0) \leftarrow xor(X), in2(X,1), out(X,1), \text{not}(ab(X))$
$in2(X,0) \leftarrow xor(X), in1(X,0), out(X,0), \text{not}(ab(X))$
$in2(X,1) \leftarrow xor(X), in1(X,0), out(X,1), \text{not}(ab(X))$
$in2(X,1) \leftarrow xor(X), in1(X,1), out(X,0), \text{not}(ab(X))$
$in2(X,0) \leftarrow xor(X), in1(X,1), out(X,1), \text{not}(ab(X))$

Abnormal

$ab(X) \leftarrow xor(X), in1(X,0), in2(X,0), out(X,1)$
$ab(X) \leftarrow xor(X), in1(X,0), in2(X,1), out(X,0)$
$ab(X) \leftarrow xor(X), in1(X,1), in2(X,0), out(X,0)$
$ab(X) \leftarrow xor(X), in1(X,1), in2(X,1), out(X,1)$

Figure 10: Xor-gate

Forward

$out(X,0) \leftarrow and(X), in1(X,0), in2(X,0), \text{not}(ab(X))$
$out(X,0) \leftarrow and(X), in1(X,0), in2(X,1), \text{not}(ab(X))$
$out(X,0) \leftarrow and(X), in1(X,1), in2(X,0), \text{not}(ab(X))$
$out(X,1) \leftarrow and(X), in1(X,1), in2(X,1), \text{not}(ab(X))$

Backward

$in1(X,1) \leftarrow and(X), out(X,1), \text{not}(ab(X))$
$in2(X,1) \leftarrow and(X), out(X,1), \text{not}(ab(X))$
$in1(X,0) \leftarrow and(X), in2(X,1), out(X,0), \text{not}(ab(X))$
$in2(X,0) \leftarrow and(X), in1(X,1), out(X,0), \text{not}(ab(X))$
$p(X) \leftarrow and(X), out(X,0), \text{not}(q(X)), \text{not}(r(X)), \text{not}(ab(X))$
$q(X) \leftarrow and(X), out(X,0), \text{not}(p(X)), \text{not}(r(X)), \text{not}(ab(X))$
$r(X) \leftarrow and(X), out(X,0), in1(X,1), \text{not}(ab(X))$
$r(X) \leftarrow and(X), out(X,0), in2(X,1), \text{not}(ab(X))$
$in1(X,0) \leftarrow and(X), p(X)$
$in2(X,0) \leftarrow and(X), q(X)$

Abnormal

$ab(X) \leftarrow and(X), in1(X,0), out(X,1)$
$ab(X) \leftarrow and(X), in2(X,0), out(X,1)$
$ab(X) \leftarrow and(X), in1(X,1), in2(X,1), out(X,0)$

Figure 11: And-gate

Forward

$out(X,0) \leftarrow or(X), in1(X,0), in2(X,0), \text{not}(ab(X))$
$out(X,1) \leftarrow or(X), in1(X,0), in2(X,1), \text{not}(ab(X))$
$out(X,1) \leftarrow or(X), in1(X,1), in2(X,0), \text{not}(ab(X))$
$out(X,1) \leftarrow or(X), in1(X,1), in2(X,1), \text{not}(ab(X))$

Backward

$in1(X,0) \leftarrow or(X), out(X,0), \text{not}(ab(X))$
$in2(X,0) \leftarrow or(X), out(X,0), \text{not}(ab(X))$
$in1(X,1) \leftarrow or(X), in2(X,0), out(X,1), \text{not}(ab(X))$
$in2(X,1) \leftarrow or(X), in1(X,0), out(X,1), \text{not}(ab(X))$
$p(X) \leftarrow or(X), out(X,1), \text{not}(q(X)), \text{not}(r(X)), \text{not}(ab(X))$
$q(X) \leftarrow or(X), out(X,1), \text{not}(p(X)), \text{not}(r(X)), \text{not}(ab(X))$
$r(X) \leftarrow or(X), out(X,1), in1(X,0), \text{not}(ab(X))$
$r(X) \leftarrow or(X), out(X,1), in2(X,0), \text{not}(ab(X))$
$in1(X,1) \leftarrow or(X), p(X)$
$in2(X,1) \leftarrow or(X), q(X)$

Abnormal

$ab(X) \leftarrow or(X), in1(X,0), in2(X,0), out(X,1)$
$ab(X) \leftarrow or(X), in1(X,1), out(X,0)$
$ab(X) \leftarrow or(X), in2(X,1), out(X,0)$

Figure 12: Or-gate

If an input or an output is set to zero or one, we encode this using one rule. Assume that the second input of the and-gate and_1 is zero and that the output is one. This is encoded as $\{\ in2(and_1, 0) \leftarrow;\ \ out(and_1, 1) \leftarrow\}$.

The results are tabulated in Figures 13–15. They have been calculated as the minimum, maximum and average times for ten different runs on a pseudo-randomly shuffled set of rules. The rationale for shuffling the set of rules is that a particular ordering of the rules might help the algorithm to avoid backtracking and thus give a skewed picture of its behavior. All times are in seconds and they represent the time to find one stable model if one exists, or the time to decide that there are no stable models. As an exception all stable models of the first benchmark program were computed. A blank entry indicates that the corresponding test was not performed or could not be completed.

All tests were run on a SPARCstation IPX workstation, with 16 MB of memory and the SunOS 4.1.3 operating system. The SLG system was compiled in SICStus 2.1 #5 with the fastcode compilation-flag set. The C++ program **sm** was compiled using gcc version 2.5.8. The timing was handled by the SICStus command 'statistics' in SLG, and by the C function 'gettimeofday' in **sm**.

The test results show clearly that **sm** computes the stable model semantics significantly faster than SLG. We can explain some of the performance dissimilarities by the different implementation techniques; **sm** is implemented in C++ while SLG is implemented as a meta-interpreter on top of Prolog. However, the execution times display, in spite of this, drastic differences.

8 Conclusions

We have developed an implementation of the well-founded and stable model semantics for range-restricted function-free normal programs. It includes two modules: an algorithm for implementing the two semantics for ground programs and an algorithm for computing a grounded version of a range-restricted function-free normal program. The emphasis has been in the implementation of the stable model semantics for ground programs. We have tested the implementation with a wide variety of test cases and compared it with a state of the art implementation of the stable model semantics, the SLG system. The tests show that for ground programs our implementation clearly outperforms SLG. The key to our success seems to lie in the new approximation technique for stable models which is a generalization of the well-founded semantics. SLG exploits the well-founded semantics, too. But the crucial difference is that we use our well-founded type approximation recursively during the whole search whereas in SLG the well-founded model is computed only once in the beginning. Our implementation of the approximation employs the Fitting operator and is quadratic time in the worst case. It provides an efficient implementation for computing the well-founded model of a ground program and often the well-founded model is computed in linear time. It seems that our implementation advances the state of the art in computing the stable model semantics for ground programs to a level where highly non-stratified ground programs with several hundreds of rules can be handled quite efficiently.

Our implementation is based on a bottom-up method and is most suitable when a complete stable model (or all stable models) should be computed. A certain degree of goal-directedness can be introduced by exploiting the capability of the implementation to perform *focused model search*, i.e., to concentrate the search on models with given characteristics, e.g., not containing a given set of atoms. Further work is needed in handling non-ground programs. In our current

Ladder graphs, 3-coloring							
Vertices	Rules	SLG			**sm**		
		min	max	ave	min	max	ave
10	73	0.48	1.24	0.72	0.02	0.05	0.03
12	87	0.59	5.26	1.17	0.02	0.03	0.02
14	101	0.74	18.14	3.72	0.02	0.06	0.03
16	115	0.86	79.24	14.06	0.03	0.05	0.03
18	129	1.08	508.85	88.82	0.03	0.04	0.03
20	143	1.14	1098.86	132.89	0.03	0.04	0.04
22	157	1.30	659.86	278.38	0.04	0.08	0.05
100	703				0.22	0.30	0.24
200	1403				0.61	0.66	0.63
300	2103				1.41	1.50	1.45
400	2803				2.59	2.73	2.65
500	3503				3.60	3.75	3.65

Planar graphs, 3-coloring							
Vertices	Rules	SLG			**sm**		
		min	max	ave	min	max	ave
10	73	7.57	34.45	23.81	0.03	0.11	0.04
12	87	2.68	140.01	82.53	0.03	0.04	0.03
14	101	35.09	657.48	390.56	0.03	0.06	0.04
16	115	3.75	2444.90	1421.19	0.04	0.06	0.05
18	129				0.04	0.08	0.05
20	143				0.04	0.06	0.05
100	703				0.19	0.23	0.21
200	1403				0.39	0.49	0.44
300	2103				0.63	0.86	0.71
400	2803				0.85	1.14	1.00
500	3503				1.03	1.35	1.12

Figure 13: Test Results

Ladder graphs, Hamiltonian circuit							
Vertices	Rules	SLG			**sm**		
		min	max	ave	min	max	ave
10	64	1.03	1.39	1.21	0.02	0.05	0.03
14	92	2.87	6.02	4.70	0.03	0.05	0.04
18	120	5.62	25.66	19.03	0.05	0.13	0.07
22	148	23.98	89.19	67.78	0.06	0.17	0.10
26	176	18.13	419.64	240.77	0.08	0.34	0.20
30	204	66.70	1649.53	899.46	0.11	0.52	0.25
40	274				0.16	1.39	0.60
50	344				0.22	2.88	0.85
60	414				0.30	6.31	2.61
70	484				0.38	12.92	2.43
80	554				0.50	23.08	12.82
90	624				0.60	33.89	13.50
100	694				0.73	48.67	21.55

Planar graphs, Hamiltonian circuit							
Vertices	Rules	SLG			**sm**		
		min	max	ave	min	max	ave
10	104	33.90	60.61	47.62	0.04	0.10	0.06
12	130	139.40	522.98	343.52	0.06	0.16	0.08
14	156	1140.33	3746.17	2132.48	0.08	0.15	0.10
16	174				0.09	0.25	0.12
18	200				0.12	4.68	0.59
20	226				0.13	97.65	10.10
22	252				0.15	1.73	0.34
24	274				0.17	4.62	0.84
26	296				0.20	905.58	163.73
28	322				0.23	33087.20	3310.88
30	348				0.25	49566.81	5028.84

SLG benchmark 1, all stable models							
Constants	Rules	SLG			**sm**		
		min	max	ave	min	max	ave
1	16	0.09	0.12	0.10	0.01	0.02	0.01
2	32	0.27	0.32	0.30	0.03	0.06	0.04
3	48	0.94	0.99	0.96	0.13	0.19	0.15
4	64	3.56	3.66	3.60	0.53	0.64	0.59
5	80	14.86	15.35	15.12	2.42	2.62	2.53
6	96	66.10	67.16	66.64	10.53	10.78	10.63
7	112	285.03	289.57	287.21	44.84	46.61	45.40

Figure 14: Test Results

SLG benchmark 2							
Constants	Rules	SLG			**sm**		
		min	max	ave	min	max	ave
50	300	1.36	1.43	1.40	0.05	0.11	0.05
100	600	2.82	2.92	2.88	0.09	0.11	0.09
150	900	4.38	4.50	4.44	0.13	0.20	0.15
200	1200	5.96	6.07	6.01	0.18	0.24	0.19
250	1500	7.49	7.62	7.57	0.22	0.26	0.23
300	1800	9.09	9.23	9.19	0.27	0.32	0.28

Win-move example, 200 edges							
Vertices	Rules	SLG			**sm**		
		min	max	ave	min	max	ave
50	450	2.20	4.56	3.16	0.16	0.57	0.40
60	460	2.35	4.30	3.34	0.09	0.86	0.44
70	470	1.91	1.99	1.94	0.07	0.29	0.16
80	480	1.14	1.35	1.23	0.07	0.10	0.08
90	490	1.15	1.28	1.21	0.08	0.13	0.09
100	500	1.03	1.28	1.16	0.08	0.12	0.09

Ripple carry adder							
Bits	Rules	SLG			**sm**		
		min	max	ave	min	max	ave
1	138	1.47	1.56	1.50	0.03	0.03	0.03
2	279	3.35	3.55	3.44	0.05	0.08	0.06
3	420	5.61	5.83	5.68	0.08	0.14	0.10
4	561	8.03	8.51	8.18	0.12	0.14	0.13
5	702	10.47	10.82	10.64	0.15	0.17	0.16
6	843	13.32	13.62	13.43	0.19	0.24	0.20
7	984	16.09	16.46	16.32	0.22	0.30	0.24
8	1125	19.35	19.81	19.60	0.27	0.33	0.29
9	1266	22.58	22.96	22.81	0.31	0.37	0.33
10	1407	25.79	26.52	26.28	0.37	0.40	0.38

Parity circuit							
Bits	Rules	SLG			**sm**		
		min	max	ave	min	max	ave
3	233	3.39	8.48	4.60	0.05	0.13	0.07
5	468	7.54	804.77	272.68	0.13	0.16	0.14
7	703	12.35	16415.75	2211.88	0.18	0.28	0.24
9	938				0.30	0.41	0.36
11	1173				0.38	0.57	0.50

Figure 15: Test Results

approach for a non-ground program, a (sometimes rather large) grounded version is computed and then the efficient algorithm for the ground case is exploited. For computing complete stable models this method can be quite competitive. But for query-answering, especially with respect to the well-founded semantics, goal-directed methods such as that in SLG could be more suitable. An interesting topic for further research would be to couple the goal-directed query-answering method in SLG, which computes a ground residual program, and our implementation of computing the stable model semantics for the ground program.

References

[BNNS94] C. Bell, A. Nerode, R.T. Ng, and V.S. Subrahmanian. Mixed integer programming methods for computing nonmonotonic deductive databases. *Journal of the ACM*, 41(6):1178–1215, November 1994.

[BS93] C. Baral and V.S. Subrahmanian. Dualities between alternative semantics for logic programming and non-monotonic reasoning. *Journal of Automated Reasoning*, 10:399–420, 1993.

[BSF95] K.A. Berman, J.S. Schlipf, and J.V. Franco. Computing the well-founded semantics faster. In *Proceedings of the 3rd International Workshop on Logic Programming and Non-monotonic Reasoning*, pages 113–126, Lexington, KY, June 1995. The MIT Press.

[CMMT95] P. Cholewiński, V.W. Marek, A. Mikitiuk, and M. Truszczyński. Experimenting with nonmonotonic reasoning. In *Proceedings of the 12th International Conference on Logic Programming*, Tokyo, June 1995.

[CW93] W. Chen and D.S. Warren. The SLG system, 1993. Available by ftp from seas.smu.edu:pub or sbcs.sunysv.edu:pub/XSB.

[CW94] W. Chen and D.S. Warren. Computation of stable models and its intergration with logical query processing. Available by ftp from ftp.ms.uky.edu:pub/lpnmr/chen2.ps, 1994.

[DG84] W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. The MIT Press.

[Knu93] D.E. Knuth. Stanford GraphBase, 1993. Available by ftp from labrea.stanford.edu.

[Nie95] I. Niemelä. Towards efficient default reasoning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 312–318, Montreal, Canada, August 1995. Morgan Kaufmann Publishers.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[SNV94] V.S. Subrahmanian, D. Nau, and C. Vago. WFS + branch bound = stable models. *IEEE Transactions on Knowledge and Data Engineering*, to appear, 1994.

[VGRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.