# Smodels – an implementation of the stable model and well-founded semantics for normal logic programs

Ilkka Niemelä and Patrik Simons

Helsinki University of Technology
Dept. of Computer Science and Engineering
Digital Systems Laboratory
P.O. 1100, FIN-02015 HUT, Finland
{Ilkka.Niemela,Patrik.Simons}@hut.fi

**Abstract.** The Smodels system is a C++ implementation of the well-founded and stable model semantics for range-restricted function-free normal programs. The system includes two modules: (i) *smodels* which implements the two semantics for ground programs and (ii) *parse* which computes a grounded version of a range-restricted function-free normal program. The latter module does not produce the whole set of ground instances of the program but a subset that 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 where a powerful pruning method is employed. The pruning method exploits an approximation technique for stable models which is closely related to the well-founded semantics. One of the advantages of this novel technique is that it can be implemented to work in linear space. This makes it possible to apply the stable model semantics also in areas where resulting programs are highly non-stratified and can possess a large number of stable models. 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.

## 1 Introduction

In recent years there has been a considerable amount of work on the formal underpinnings of *declarative* logic programming. This has led to the development of several alternative semantics for the procedural SLDNF semantics employed in standard Prolog implementations. For normal logic programs the leading declarative approaches are the stable model semantics [8] and the well-founded semantics [15]. Until quite recently there has been relatively little work on developing serious implementations for these new semantics with the notable exception of the work of D. Warren's group on implementing the well-founded semantics. This has led to a WAM implementation [13]. Our aim is to make a serious attempt at implementing the stable model semantics. We focus on range-restricted function-free normal programs for which the stable model semantics is computable. In our approach a construction closely related to the well-founded semantics plays an important role and has led us to devise also an implementation of the well-founded semantics.

In this paper we describe our implementation of the well-founded and stable model semantics for range-restricted function-free normal programs. The 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 that is sufficient to ensure that no stable models are lost.

The implementation is able to solve a range of computational problems related to normal programs.

- It can compute the well-founded model of a program.
- It is able to decide whether a program has a stable model.
- It can generate all or a given number of stable models of a program.
- It is also able to handle two basic query-answering tasks, i.e., to decide whether a given literal is satisfied in some or all of the stable models of a program.

The implementation of the stable model semantics for ground programs is based on a novel technique where bottom-up backtracking search with a powerful pruning method is employed [11, 12]. One of the advantages of this technique is that it can be implemented to work in linear space. This 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. The linear space complexity ensures that these kinds of hard instances can be solved 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 [5] which is a system for generating test cases for nonmonotonic reasoning.

There are several approaches to computing stable models (see, e.g., [1]). Recently, some more advanced implementations have emerged [2, 4, 14, 6]. The methods described in [2, 14] cannot 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 DeReS system [6] implements Reiter's default logic and can thus be used also for computing stable models. The SLG system [3, 4] appears to be able to handle larger examples. We have performed an experimental comparison between our system and SLG. In the tests our system clearly outperformed SLG. For further details of the tests and a comparison of the techniques used in the two systems, see [11, 12].

The rest of the paper is organized as follows. In the next section we describe some of the underlying ideas of our implementation techniques. Section 3 illustrates the use of the system. Section 4 contains some results from the experimental evaluation of the system and Section 5 explains how to obtain the system.

## 2 Theoretical Background and Implementation Techniques

Our implementation of the stable model semantics for ground programs is based on an approach introduced for implementing default logic [10]. This approach provides a framework for developing bottom-up methods for computing extensions of default theories. We have used the framework to devise an efficient implementation for the stable model semantics. One of the underlying ideas in the approach is that stable models are characterized in terms of their so-called *full sets*, i.e., their complements with respect to the negative atoms in the program (negative atoms in the program for which the corresponding positive atoms are not included in the stable model) [10, 11]. This characterization leads to a novel bottom-up backtracking algorithm for searching for stable models. The algorithm exploits a powerful pruning method which is based on approximating stable models. The approximation technique is closely related to the well-founded semantics. The algorithm is also able to handle *focused model search*, i.e., to concentrate the search on models with given characteristics, e.g., not containing a given set of atoms. The algorithm has been implemented in C++ and the implementation possesses some interesting properties.

- It includes an efficient (quadratic time) algorithm for computing the well-founded model of a ground program. The algorithm employs a Fitting operator to speed up the computation. In practice the well-founded model can often be computed in linear time.
- It runs in linear space.
- It employs linear time algorithms for computing the deductive closures that are needed in the algorithm.
- It exploits a dynamic search heuristic.

Our algorithm for computing the grounded version of a range-restricted function-free normal program is based on the idea of generating only those instances of range-restricted rules that are at least potentially applicable in the stable models of the program. Descriptions of both algorithms and details of the implementation techniques can be found in [11, 12].

## 3 Using Smodels

In order to compute stable models, one uses the program *smodels* that computes the models, and the parser *parse* that translates logic programs into a format *smodels* accepts, see Figure 1. Both programs work as filters, i.e., they read from the standard input and write to the standard output. We begin by describing

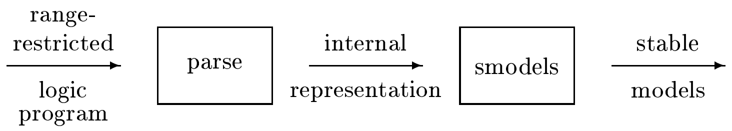


**Fig. 1.** Overall Architecture

the input format to *parse* and continue with the options and some examples of their use.

The parser accepts the following syntax. The atoms are strings of parentheses, underscores, alphabetical, and numerical characters. In the case of range-restricted programs, atoms begin with a lower-case alphabetical or a numerical character, and variables with an upper-case alphabetical character. The parentheses must always match. To express negation one uses not-atoms, which are atoms preceded by the string 'not'. A rule begins with an atom denoting its head followed by the inference symbol '$:-$', which is in turn followed by the body of the rule as a comma separated list of atoms and not-atoms. Finally, the rule ends with a period. As an example, the rule

$$a \leftarrow b, \mathrm{not}(c)$$

is written as

$$a :- \ b, \mathrm{not}\ c.$$

Comments begin with a percent sign and end with a line-break. Moreover, the parser allows rules with variables as long as the rules are range-restricted. A rule is range-restricted if the variables that appear in the head or in the negative literals in the body of the rule also appear in the positive literals in the body of the rule. Thus, the rules

$$a(X) :- \ b(X), \mathrm{not}\ c. \quad \text{and} \quad p :- \ q\big(f(X)\big), \mathrm{not}\ r(X).$$

are range-restricted, but the rules

$$a(X) :- \ b(Y). \quad \text{and} \quad p :- \ \mathrm{not}\ q(X).$$

are not. Note that functions are accepted but ignored, they are only treated as part of the predicates.

The number of stable models that are computed can be determined by the string 'compute' followed by a number. Alternatively, if the string 'compute' is followed by the string 'all', then all stable models are computed. If the 'compute' string is not specified, at most one stable model is computed.

If the previous construction is followed by a comma separated list of atoms and not-atoms enclosed in braces, then only stable models that contain the atoms and do not contain the not-atoms in the list are computed.

The idea of the 'compute' instruction is to provide the ability to perform focused model search, i.e., to concentrate the search on models with given conditions. This means that *smodels* can be used not only to generate a given number of stable models but also for *query-evaluation*. For instance, if we want to evaluate whether there is a stable model of a program containing the atom $p(a)$ but not the atom $d$, we would use an input file containing the program and a 'compute' instruction as follows

$$\begin{array}{l}
\%\ \text{input program} \\
p(X) :- r(X), \text{not } q(X). \\
q(X) :- r(X), \text{not } p(X). \\
r(b) :- \text{not } d. \\
d :- \text{not } p(c). \\
r(c) :- r(b). \\
r(a). \\
\%\ \text{query specification} \\
\text{compute } \{\, p(a), \text{not } d \,\}
\end{array}$$

For this input file, the system would return a stable model containing the atom $p(a)$ but not the atom $d$ provided that such a model exists. To evaluate whether $p(a)$ is in every stable model of the program, the 'compute' instruction is changed to

$$\text{compute } \{\, \text{not } p(a) \,\}$$

Then the system searches for a counter-model, i.e., a model not containing $p(a)$. If no such model is found, $p(a)$ belongs to every stable model of the program.

We now turn to the command line options. The parser takes two options; the option '-plain', which treats variables as ordinary atoms, and the option '-text', which produces readable output in the form of a logic program, i.e., if the program is range-restricted it is first grounded and then displayed.

The program *smodels* takes one optional argument and one option, '-w'. The argument is a number determining how many stable models are computed, a zero indicating all. The option, when present, makes *smodels* compute only the well-founded model.

In conclusion, a stable model of the set of rules in the file 'prog' would typically be computed by the command line

```
parse < prog | smodels
```

producing the output

```
smodels version 1.5. Reading...done
Answer: 1
Stable Model: p(a) p(c) r(c) p(b) r(b) r(a)
Full set: d q(c) q(b) q(a)
True
Duration: 0.013
Number of extension calls: 2
```

Number of wrong choices: 0
Number of atoms: 10
Number of rules: 10

The line 'Answer: 1' indicates that the following stable model is the first one, which together with the corresponding full set is printed on the next two lines. The word 'True' tells us that there might be more stable models, whereas the word 'False' would have told us that there are no more stable models. The duration is expressed in seconds and includes the time it takes to read the input and print the output. The number of extension calls indicates how much of the search space has been explored, and the number of wrong choices indicates how many times backtracking has taken place. The number of atoms and rules are self-explanatory.

## 4 Evaluation

We have tested our implementation quite extensively using test cases from

- the logic programming literature,
- combinatorial graph problems,
- circuit diagnosis, and
- propositional satisfiability.

Here we briefly describe some tests involving n-colorings and Hamiltonian circuits in planar graphs, and propositional satisfiability. For more details, further results, and a comparison against the SLG system we refer to [11, 12].

The graphs in the tests are created with the Stanford GraphBase [9], a highly portable collection of programs that serves as a platform for combinatorial algorithms. The propositional formulas, in turn, are randomly generated formulas in conjunctive normal form, whose clauses contain exactly three atoms, and whose clause to atom ratio is 4.3. This ratio was chosen, as it determines a region of hard satisfiability problems [7]. The formulas are generated by a program developed by Bart Selman.

The test cases are generated by translating a given graph to a ground logic program in such a way that every stable model of the program corresponds to a solution to the problem in question.

We translate the n-coloring problem 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 $i \in \{0, 1, \ldots, n-1\}$, we include the rule

$$\begin{aligned} \mathit{color}(a,i) \leftarrow\ & \mathrm{not}\big(\mathit{color}(p_1,i)\big), \ldots, \mathrm{not}\big(\mathit{color}(p_j,i)\big), \\ & \mathrm{not}\big(\mathit{color}(a, i+1 \bmod n)\big), \ldots, \\ & \mathrm{not}\big(\mathit{color}(a, i+n-1 \bmod n)\big), \end{aligned}$$

and for each vertex $a$ we include the rule

$$h \leftarrow \mathrm{not}\big(\mathit{color}(a,0)\big), \mathrm{not}\big(\mathit{color}(a,1)\big), \ldots, \mathrm{not}\big(\mathit{color}(a,n-1)\big).$$

Finally, we consider only the stable models that do not contain the atom $h$.

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 $a_1, \ldots, a_j$ are the neighbors of $b$ excluding $a$, we include the rule

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

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 $b \neq d$ we include the rule

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

and if $b = d$ we include the rule

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

Finally, we add the rule $d \leftarrow$, and consider only the stable models that do not contain the vertex $a$, for $a \neq d$, nor the atom $d'$.

In contrast, the translation of the 3-SAT problem is entirely straightforward. For every atom $a$ we add the rules

$$a \leftarrow \mathrm{not}(\overline{a}) \quad \text{and} \quad \overline{a} \leftarrow \mathrm{not}(a),$$

and for every clause $c$ we add $c \leftarrow a$ if $a$ is in the clause, and $c \leftarrow \overline{a}$ if $\neg a$ is in the clause. The satisfying assignments are then given by the stable models that contain all clauses.

Some test results are tabulated in Figure 2. They have been calculated as the average time of 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.

The tests were run on a Pentium 75MHz, with 32 MB of memory and the Linux 2.0.29 operating system. The C++ program *smodels* was compiled using gcc version 2.7.2.1.

When evaluating our system we have used test cases generated from random graphs and random propositional formulas. There are a number of reasons for using these kinds of randomly generated test cases. First, in order to determine the scalability and limits of the implementation it is important to have classes of test cases of increasing size and complexity instead of single isolated examples. Our experience indicates that test cases based on combinatorial problems quite naturally provide such classes of examples.

Second, our aim is to develop a framework for *declarative* logic programming where the performance of the system is not too sensitive to the way in which the program is represented. Finding a stable model is a typical combinatorial problem where tuning the test cases even in a very modest way, e.g., by changing the order of some rules in the program, can lead to huge improvements in the running times. This might then give a totally unrealistic picture of the actual performance and *stability* of the algorithm. By using random test cases and by shuffling the input program, we can evaluate the stability of the implementation in a declarative setting.

Third, there are a lot of results on solving combinatorial problems involving randomly generated instances. Hence, we know how to find hard instances of these problems and we know the performance of the best special purpose algorithms. It is indeed very important to have points of reference independent of logic programming techniques for measuring the efficiency and overhead of logic programming implementations.

| Planar graphs, 3-coloring | | | | |
|---|---|---|---|---|
| Vertices | Rules | Min | Average | Max |
| 100 | 400 | 0.06 | 0.06 | 0.06 |
| 200 | 800 | 0.11 | 0.15 | 0.28 |
| 300 | 1200 | 0.24 | 0.25 | 0.25 |
| 400 | 1600 | 0.30 | 0.32 | 0.35 |
| 500 | 2000 | 0.32 | 0.37 | 0.39 |
| 600 | 2400 | 0.36 | 0.43 | 0.45 |
| 700 | 2800 | 0.45 | 0.51 | 0.59 |
| 800 | 3200 | 0.52 | 0.57 | 0.60 |
| 900 | 3600 | 0.63 | 0.65 | 0.67 |

| Planar graphs, 4-coloring | | | | |
|---|---|---|---|---|
| Vertices | Rules | Min | Average | Max |
| 10 | 50 | 0.01 | 0.02 | 0.08 |
| 20 | 100 | 0.02 | 0.02 | 0.08 |
| 30 | 150 | 0.03 | 0.05 | 0.10 |
| 40 | 200 | 0.04 | 0.08 | 0.11 |
| 50 | 250 | 0.05 | 0.09 | 0.17 |
| 60 | 300 | 0.06 | 0.15 | 0.65 |
| 70 | 350 | 0.08 | 45.81 | 456.08 |
| 80 | 400 | 0.09 | 0.14 | 0.22 |
| 90 | 450 | 0.11 | 14.39 | 138.46 |

| Planar graphs, Hamiltonian circuit | | | | |
|---|---|---|---|---|
| Vertices | Rules | Min | Average | Max |
| 14 | 133 | 0.03 | 0.04 | 0.05 |
| 16 | 157 | 0.05 | 0.51 | 1.28 |
| 18 | 181 | 0.07 | 0.85 | 3.17 |
| 20 | 197 | 0.07 | 0.13 | 0.29 |
| 22 | 221 | 0.09 | 0.10 | 0.13 |
| 24 | 245 | 0.10 | 19.04 | 91.38 |
| 26 | 269 | 0.12 | 55.68 | 272.67 |
| 28 | 293 | 0.14 | 47.56 | 441.27 |
| 30 | 317 | 0.16 | 673.70 | 3417.52 |

| Random 3-SAT | | | | |
|---|---|---|---|---|
| Atoms | Rules | Min | Average | Max |
| 20 | 298 | 0.02 | 0.03 | 0.09 |
| 30 | 447 | 0.03 | 0.04 | 0.06 |
| 40 | 596 | 0.04 | 0.15 | 0.26 |
| 50 | 745 | 0.24 | 0.40 | 0.87 |
| 60 | 894 | 0.38 | 1.08 | 2.43 |
| 70 | 1043 | 0.25 | 0.93 | 2.22 |
| 80 | 1192 | 0.67 | 5.27 | 15.62 |
| 90 | 1341 | 0.31 | 6.41 | 39.51 |
| 100 | 1490 | 17.68 | 101.83 | 278.05 |

**Fig. 2.** Test Results

## 5 Availability

The Smodels system is freely available at http://saturn.hut.fi/pub/smodels/. Documentation and an extensive set of test cases, including those described in the previous section, are available at the same location.

In order to make use of the system you will need a C++ compiler and other standard tools such as *make* and *tar*. The system has been developed under Linux and should work as is on any platform having the appropriate GNU tools installed.

## 6 Conclusions

We have developed a C++ implementation of the well-founded and stable model semantics for range-restricted function-free normal programs. The main emphasis has been in developing an implementation of the stable model semantics that could be used in realistic applications. The novel implementation technique, leading to linear space complexity, has turned out to be very competitive when compared to other available implementation methods. Our implementation appears to be the only available system capable of handling non-stratified ground programs with several hundreds or even thousands of rules quite efficiently. This indicates that our implementation has significantly advanced the state of the art in computing the stable model semantics and has brought the semantics considerably closer to applications.

## References


1. C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19&20(73–148), 1994.
2. 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.
3. W. Chen and D.S. Warren. The SLG system, 1993. Available at ftp://seas.smu.edu/pub/ or ftp://sbcs.sunysv.edu/pub/XSB/.
4. W. Chen and D.S. Warren. Computation of stable models and its integration with logical query processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.
5. P. Cholewiński, V.W. Marek, A. Mikitiuk, and M. Truszczyński. Experimenting with nonmonotonic reasoning. In *Proceedings of the 12th International Conference on Logic Programming*, pages 267–281, Tokyo, June 1995.
6. P. Cholewiński, V.W. Marek, and M. Truszczyński. Default reasoning system DeReS. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 518–528, Cambridge, MA, USA, November 1996. Morgan Kaufmann Publishers.
7. J.M. Crawford and L.D. Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81(1):31–57, 1996.
8. 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.
9. D.E. Knuth. The Stanford GraphBase, 1993. Available at ftp://labrea.stanford.edu/.

10. 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.
11. Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, September 1996. The MIT Press.
12. Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. Fachbericht Informatik 7–96, Universität Koblenz-Landau, 1996. Available at http://www.uni-koblenz.de/universitaet/fb4/publications/GelbeReihe/.
13. K. Sagonas, T. Swift, and D.S. Warren. An abstract machine for computing the well-founded semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 274–288, Bonn, Germany, September 1996. The MIT Press.
14. V.S. Subrahmanian, D. Nau, and C. Vago. WFS + branch bound = stable models. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):362–377, 1995.
15. 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.