Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm

Ilkka Niemelä
Helsinki University of Technology
Dept. of Computer Science and Eng., Digital Systems Laboratory
P.O. Box 1100, FIN-02015 HUT, Finland
`Ilkka.Niemela@hut.fi` `http://saturn.hut.fi/~ini`

Abstract

Logic programming with the stable model semantics is put forward as an interesting constraint programming paradigm. It is shown that the paradigm embeds classical logical satisfiability but seems to provide a more expressive framework from a knowledge representation point of view. The applicability of the paradigm is demonstrated using combinatorial graph problems and planning problems as examples. An efficient implementation of the paradigm has been developed and is publicly available. The implementation is described briefly, pointers to encouraging experimental results are provided and some test results on blocks world planning problems are reported.

1 Introduction

We put forward logic programs with the stable model semantics ($\mathbf{LP_{SM}}$) as an interesting constraint programming paradigm. The goal is to bring advantages of logic programming based knowledge representation techniques to constraint programming. These techniques seem particularly useful in dynamic domains (such as planning) where, e.g., the frame problem and the qualification problem emerge. The underlying idea in this paradigm is to interpret the rules of a program as constraints on a solution set for the program. A solution set is a set of atoms and a logic program rule of the form

$$A \leftarrow B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n \qquad (1)$$

is seen as a constraint on this set stating that if $B_1, \ldots, B_m$ are in the solution set and none of $C_1, \ldots, C_n$ are included, then A must be included in the set. These solution sets are the key objects of interest in this paradigm and, hence, the task of a system implementing the paradigm is to compute solution sets for programs. Such a system differs substantially from typical logic programming systems, such as Prolog implementations, which are goal-directed backward chaining query evaluation systems where the task of the system is to compute for a given query a yes/no answer or more generally an answer substitution.

The integration of constraints and logic programming has been studied mainly from the point of view of extending Prolog style goal-directed implementation techniques by allowing, e.g., arithmetic constraints in the rules and by integrating the necessary constraint solvers into a logic programming system. This constraint logic programming paradigm [JL87] has been extended to include nonmonotonic reasoning capabilities such as abduction [KM97]. However, the constraint logic programming paradigm differs significantly from our approach where the rules have a declarative semantics and can be understood themselves as constraints. Hence in our paradigm, rules can be used directly for expressing constraints without extending the language to allow constraint expressions in the rules.

A very natural definition for the solution sets is provided by *the stable model semantics* [GL88] which is one of the leading declarative semantics of logic programs. The stable model semantics generalizes in a simple way the minimal model semantics of definite programs [vEK76] to the case where negative body literals are allowed in the program rules. For a ground (variable-free) program P, the stable models are defined as follows. The *reduct* of a program P with respect to a set of atoms S is the (definite) program obtained from P by deleting

(i) each rule that has a negative literal not a in its body with $a \in S$ and

(ii) all negative literals in the bodies of the remaining rules.

Now a set of ground atoms S is a *stable model* of P if and only if S is the unique minimal model of the reduct of P with respect to S. For example, program P

$$p \leftarrow \text{not } q, r$$
$$q \leftarrow \text{not } p$$
$$r \leftarrow \text{not } s$$
$$s \leftarrow \text{not } p$$

has a stable model $S_1 = \{r, p\}$ because the reduct of P with respect to S_1 is

$$p \leftarrow r$$
$$r$$

and S_1 is its unique minimal model. For instance, $S_2 = \{p, s\}$ is not a stable model of P because the reduct is $p \leftarrow r$ and its unique minimal model $\{\}$. In fact, P has another stable model $\{s, q\}$. Hence, a program can possess multiple stable models but sometimes none at all as is the case for the program $p \leftarrow$ not p. The problem of deciding whether a ground program has a stable model is NP-complete [MT91]. For a non-ground program P, the stable models are those of the ground instantiation of the program with respect to its Herbrand universe.

The constraint interpretation of the rules provided by the stable model semantics is quite natural and, in fact, stable models have been shown to correspond exactly to the models computed by a justification-based truth maintenance system (TMS) when logic programming rules are seen as justifications in a TMS [Elk90] and to default extensions when rules are treated as default rules in Reiter's default logic [GL90].

The rest of the paper is organized as follows. First we show that traditional Boolean constraints, i.e. propositional satisfiability, can be embedded in a simple way to $\mathbf{LP_{SM}}$ and argue that from a knowledge representation point of view $\mathbf{LP_{SM}}$ is more expressive than propositional logic. Then we illustrate the use $\mathbf{LP_{SM}}$ by some examples. Finally we report on our work on implementation and applications of the paradigm and finish with some concluding remarks.

2 Relation to Propositional Satisfiability

Stable models are sets of atoms similar to propositional models. However, there are two significant differences.

- Stable models are *minimal*: a proper subset of a stable model is not a stable model.
- Stable models are *grounded*: each atom in a stable model is derivable by the reduct of the program with respect to the model.

We show that despite the differences propositional satisfiability (**SAT**) can be easily reduced to $\mathbf{LP_{SM}}$ employing a simple local mapping. Then we argue that a similar local mapping is not possible in the reverse direction implying that $\mathbf{LP_{SM}}$ is more expressive than propositional logic from a knowledge representation point of view.

SAT can be mapped to $\mathbf{LP_{SM}}$ by constructing a ground logic program $\mathrm{Tr}_{\mathrm{SAT}}(S)$ for a set of clauses S, for example, in the following straightforward way. (i) We introduce for each atom a appearing in S two atoms a and $\hat{a}$ and include two rules

$$\hat{a} \leftarrow \text{not } a$$
$$a \leftarrow \text{not } \hat{a}$$

(ii) For each clause in S, we introduce a new atom c and include one rule for each literal l in the clause as follows: if l is a positive atom a, take the rule $c \leftarrow a$ and if l is the negation of an atom a, add $c \leftarrow \hat{a}$ and (iii) finally we include the rules

$$\mathit{satisfied} \leftarrow c_1, \ldots, c_n$$
$$f \leftarrow \text{not } f, \text{not } \mathit{satisfied}$$

where f is a new atom and $c_1, \ldots, c_n$ are all the atoms introduced for the clauses in S. You et al. [YCL96] present a reduction from propositional satisfiability to logic programs which is based on similar ideas as the mapping above but they use as the target language extended logic programs (with classical negation) and study special semantics developed for extended programs instead of the stable model semantics.

Proposition 2.1 *A set of clauses S has a model iff $\mathrm{Tr}_{\mathrm{SAT}}(S)$ has a stable model.*

Note that a stable model of $\mathrm{Tr}_{\mathrm{SAT}}(S)$ provides directly a propositional model for the clauses S where atoms in the stable model are assigned true and the rest of the atoms false (atoms a for which $\hat{a}$ is in the stable model).

The proposition shows that propositional satisfiability can be reduced to the problem of finding a stable model. The mapping can be modified easily to provide a reduction to a query evaluation problem, i.e., to the problem of finding a stable model satisfying a given query. It is sufficient to remove the last rule $f \leftarrow \text{not } f, \text{not } \mathit{satisfied}$. Then the clauses S have a model iff the corresponding set of rules has a stable model containing *satisfied*.

Although it is clear from the complexity results that the problem of finding a stable model can be reduced in polynomial time to a propositional satisfiability problem (as both are NP-complete problems), it is not obvious whether the two approaches are equally expressive from a knowledge representation point of view. In fact, there seems to be no way of mapping $\mathbf{LP_{SM}}$ to **SAT** in a similar local modular fashion as we embedded **SAT** to $\mathbf{LP_{SM}}$ above where small local changes in the input clauses lead to small local changes in the corresponding logic program. We can show this formally under mild assumptions on the notion of modularity. Consider, e.g., a notion of modularity where a mapping $\mathrm{T}(\cdot)$ from logic programs to propositional clauses is said to be *modular* if for any program P, for each set of atomic facts F, there is a set of propositional formulae $\mathrm{T}'(F)$ such that $P \cup F$ has a stable model iff $\mathrm{T}'(F) \cup \mathrm{T}(P)$ is satisfiable. The intuition here is that for a modular mapping, adding an atom to the program should lead to a local change not evolving the translation of the rest of the program.

Proposition 2.2 *There is no modular mapping from logic programs to clauses.*

Proof. Consider a program $P = \{p \leftarrow \text{not } p\}$. Assume that $\mathrm{T}(\cdot)$ is a modular mapping. Then as P has no stable models, $\mathrm{T}(P)$ is unsatisfiable. But then $\mathrm{T}'(\{p\}) \cup \mathrm{T}(P)$ is unsatisfiable no matter how the set of formulae $\mathrm{T}'(\{p\})$ is chosen. This implies that $P \cup \{p\}$ has no stable model which is clearly not the case. Hence, no modular mapping exists. □

The proposition implies that there could be situations having a natural representation using $\mathbf{LP_{SM}}$ but not using propositional logic in the sense that even simple updates like adding a new fact could result into non-local changes in the propositional representation of the situation. In particular, this seems to hold in dynamic situations where, e.g., the frame problem and the qualification problem have to be addressed. Another significant difference between **SAT** and $\mathbf{LP_{SM}}$ is in the structure of the search spaces where the minimality and groundedness properties of stable models seem to provide interesting computational advantages. We return to this point in the concluding remarks.

3 Applying the Paradigm

In the previous section we showed that Boolean constraints can be embedded to $\mathbf{LP_{SM}}$ using a linear time translation. This implies that $\mathbf{LP_{SM}}$ can be employed for constraint programming whenever Boolean constraints are sufficient. However, constraints have often a very natural representation directly as logic program rules. In the previous section we even argued that rules under the stable model semantics are more expressible than Boolean constraints. In particular, rules have advantages in cases where the built-in properties of stable models, i.e. minimality and groundedness, play a role. We demonstrate the applicability of $\mathbf{LP_{SM}}$ using examples from combinatorial graph problems and planning.

Combinatorial graph problems

Logic programs with the stable model semantics provide a framework where typical constraint programming tasks can be expressed in a straightforward way. As the first example, we consider the 3-colorability problem, i.e., the problem of finding an assignment of one of three colors to each vertex of a graph such that vertices connected with an arc do not have the same color. This problem can be mapped to a stable model finding problem as follows. Assume that we have a database giving a graph in terms of atomic facts of the form $\mathit{vertex}(v)$ and $\mathit{arc}(v, u)$. Then take the program consisting of these facts and the rules in Figure 1.

Now the program has a stable model if and only if there is a 3-coloring of the graph. Note that the mapping from colorability to $\mathbf{LP_{SM}}$ is constructive in the sense that a 3-coloring of the graph is directly obtained from a stable model by taking the facts of the form $\mathit{color}(v, c)$ true in the model. A reduction from 3-colorability to a query evaluation problem can be obtain by removing the last rule in Figure 1. Then there is a 3-coloring of the graph iff the program has a stable model not containing *noncoloring*.

As an example which is not straightforward to map to a **SAT** problem but which has quite a natural coding in $\mathbf{LP_{SM}}$ we consider the Hamiltonian circuit problem, i.e., the problem of finding a path from a graph that visit each vertex of the graph exactly once and returns to the starting vertex. Again assume that we have a database giving a graph in terms of atomic facts of the form $\mathit{vertex}(v)$ and $\mathit{arc}(v, u)$. We add to the facts the rules in Figure 2 and take one of the vertices v as the starting vertex ($\mathit{initialnode}(v)$ is added). The resulting program has a stable model if and only if the graph

$$
\begin{array}{l}
color(V, blue) \leftarrow vertex(V), \text{not } color(V, green), \text{not } color(V, red) \\
color(V, green) \leftarrow vertex(V), \text{not } color(V, blue), \text{not } color(V, red) \\
color(V, red) \leftarrow vertex(V), \text{not } color(V, green), \text{not } color(V, blue) \\
noncoloring \leftarrow arc(V, U), color(V, C), color(U, C) \\
f \leftarrow \text{not } f, noncoloring
\end{array}
$$

Figure 1: A program for 3-colorability

has a Hamiltonian circuit. Note that the mapping from Hamiltonian circuits to $\mathbf{LP}_{\mathbf{SM}}$ is constructive in the sense that a circuit is directly obtained from a stable model by taking the facts of the form $hcircuit(v, u)$ true in the model. A reduction to a query evaluation problem can be obtain by removing the last rule in Figure 2. Then the graph has a Hamiltonian circuit if and only if the resulting program has a stable model not containing $noncircuit$.

The mapping exploits the groundedness and minimality properties of stable models by representing Hamiltonian circuits using rules with exceptions typically employed in logic programming based knowledge representation. For example, the first rule says that an arc belongs to the circuit if there is no exception, i.e., no other route between the two nodes. This leads to a compact and easily maintainable representation where, e.g., the graph can be changed without changing the rules describing the conditions on the circuit.

Planning

Planning provides a particularly interesting application area for nonmonotonic reasoning systems because this is a domain from which some of the main motivation for developing nonmonotonic formalisms originates. In planning difficult issues related to reasoning about action and change such as the frame problem have to be addressed and the expressivity of the nonmonotonic formalisms can be utilized to overcome some of the difficulties. We illustrate with a blocks world example how planning problems can be mapped to nonmonotonic reasoning problems. For more detailed accounts, we refer the reader, e.g., to [GL93, DNK97].

In the blocks world we are given initial conditions concerning blocks on a table stating how they are stacked on top of each other, similar goal conditions and a robot that is able to move blocks. The aim is to generate a plan for the robot to move blocks from the initial configuration so that the goal conditions are satisfied.

Consider the following example. In the initial configuration we have three blocks a, b, c such that b and c are on the table and a is on top of b. The goal conditions are that c is on a and b is on c. A possible solution for this planning problem is a sequence of moves where a is moved on the table, c is moved on a and finally b is moved on c.

The idea is to map a planning problem to a logic program such that stable models correspond to valid plans. For formalizing blocks world planning we use situations where facts hold. We assume that we have situations $t_0, \ldots, t_n$ where t_0 is the initial situation and the available situations are given using facts of the form $time(t_i)$. A predicate $nextstate$ specifies the order of the situations, i.e., for each $i = 0, \ldots, n-1$, $nextstate(t_{i+1}, t_i)$ holds. We employ predicates $on(X, Y, T)$ (X is on Y in the situation T) and $moveop(X, Y, T)$ (X is moved on Y in the situation T) and assume that the available blocks are specified using facts of the form $block(b)$.

The initial conditions are straightforward to formalize. For instance, for the example above it is sufficient to include the facts

$$
\begin{array}{l}
on(a, b, t_0) \\
on(b, table, t_0) \\
on(c, table, t_0)
\end{array}
$$

In order to capture the goal conditions we employ a predicate $goal$ which holds in any situation where the goal conditions have been reached and in all subsequent situations. For the example above, the resulting rules are

$$
\begin{array}{l}
goal(T) \leftarrow time(T), \\
\qquad\qquad on(b, c, T), \\
\qquad\qquad on(c, a, T) \\
goal(T_2) \leftarrow time(T_1), nextstate(T_2, T_1), goal(T_1) \\
goal \leftarrow time(T), goal(T) \\
f \leftarrow \text{not } f, \text{not } goal
\end{array}
$$

In order to formalize the preconditions and effects of

$$hcircuit(V_1, V_2) \leftarrow arc(V_1, V_2), \text{not } otherroute(V_1, V_2)$$
$$otherroute(V_1, V_2) \leftarrow arc(V_1, V_2), hcircuit(V_1, V_3), \text{not } (V_2 = V_3)$$
$$otherroute(V_1, V_2) \leftarrow arc(V_1, V_2), hcircuit(V_3, V_2), \text{not } (V_1 = V_3)$$
$$reached(V_2) \leftarrow hcircuit(V_1, V_2), reached(V_1)$$
$$reached(V_2) \leftarrow hcircuit(V_1, V_2), initialnode(V_1)$$
$$noncircuit \leftarrow vertex(V), \text{not } reached(V)$$
$$f \leftarrow \text{not } f, noncircuit$$

Figure 2: A program for Hamiltonian Circuits

the move operator we use the following rules.

$$\begin{aligned} moveop(X, Y, T) \leftarrow\ & time(T), \\ & moveable(X), \\ & block(Y), \\ & \text{not } (X = Y), \\ & on\_something(X, T), \\ & \text{not } covered(X, T), \\ & \text{not } covered(Y, T), \\ & \text{not } blocked\_move(X, Y, T) \end{aligned}$$
$$\begin{aligned} on(X, Y, T_2) \leftarrow\ & time(T_1), \\ & nextstate(T_2, T_1), \\ & moveop(X, Y, T_1) \end{aligned}$$
$$on\_something(X, T) \leftarrow on(X, Z, T)$$
$$\begin{aligned} covered(X, T) \leftarrow\ & moveable(X), \\ & on(Z, X, T) \end{aligned}$$
$$moveable(X) \leftarrow block(X), \text{not } (X = table)$$

It is enough to provide *frame axioms* only for the predicate *on*.

$$\begin{aligned} on(X, Y, T_2) \leftarrow\ & time(T_1), \\ & nextstate(T_2, T_1), \\ & on(X, Y, T_1), \\ & \text{not } moving(X, T_1) \end{aligned}$$
$$moving(X, T) \leftarrow moveop(X, Y, T)$$

What remains to be stated are the blocking conditions for the moves. The first set of conditions covers the cases where the goal has been reached or the instance of the move operator has not been chosen.

$$\begin{aligned} blocked\_move(X, Y, T) \leftarrow\ & moveable(X), \\ & block(Y), \\ & goal(T) \end{aligned}$$
$$\begin{aligned} blocked\_move(X, Y, T) \leftarrow\ & time(T), \\ & moveable(X), \\ & block(Y), \\ & \text{not } moveop(X, Y, T) \end{aligned}$$

The second set depends on whether concurrency is allowed, i.e., whether more than one operator can be applied in a situation. We allow this and block only the operator instances whose effects are in conflict.

$$\begin{aligned} blocked\_move(X, Y, T) \leftarrow\ & moveable(X), \\ & block(Y), \\ & moveop(X, Z, T), \\ & \text{not } (Y = Z) \end{aligned}$$
$$\begin{aligned} blocked\_move(X, Y, T) \leftarrow\ & moveable(X), \\ & moving(Y, T) \end{aligned}$$
$$\begin{aligned} blocked\_move(X, Y, T) \leftarrow\ & moveable(X), \\ & block(Y), \\ & \text{not } (Y = table), \\ & moveop(Z, Y, T), \\ & \text{not } (X = Z) \end{aligned}$$

This kind of constraint formulation of planning allows flexible integration of different kinds of pruning rules. For example, we can exclude a move from the table back to the table or a move on top of something and then immediately to the table:

$$\begin{aligned} f \leftarrow\ & moveop(X, table, T), \\ & on(X, table, T) \end{aligned}$$
$$\begin{aligned} f \leftarrow\ & nextstate(T_2, T_1), \\ & moveop(X, Y, T_1), \\ & moveop(X, table, T_2) \end{aligned}$$

For a program constructed like this it holds that the program has a stable model if and only if there is a sequence of moves from the initial configuration to a situation satisfying the goal conditions that can be executed concurrently in at most n steps. Note that a stable model provides directly a valid plan with the facts of the form $moveop(x, y, t)$ true in the model. A plan can be built just by arranging the facts in the order given by the situation argument t. A valid sequential plan is obtained from this by arranging facts with the same situation argument in any linear order.

A query-oriented formulation of planning can be obtained by removing the rule $f \leftarrow \text{not } f, \text{not } goal$. Then there is a plan iff the resulting program has a stable model containing *goal*. If pruning rules are employed, then a stable model containing *goal* but not f should be searched for.

The expressivity of logic program rules is exploited, e.g., in representing frame axioms and blocking conditions. A very compact representation is obtained with nice modularity properties, e.g., updating the representation with new blocks or operators is fairly straightforward.

4 Implementation

We have developed a C++ implementation of $\mathbf{LP_{SM}}$ [NS96a, NS96b, NS97]. Our system called **Smodels** implements the stable model semantics for range-restricted function-free normal programs. A program is range-restricted if in every rule each variable that appears in the head or in a negative literal in the body of the rule also appears in a positive literal in the body of the rule. For example, the rules in the previous section are range-restricted.

The system includes two modules: (i) `smodels` which implements $\mathbf{LP_{SM}}$ 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 the stable models are preserved.

The implementation of the stable model semantics for ground programs is based on bottom-up backtracking search where one of the underlying ideas 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) [Nie95, NS96a]. The search space is pruned efficiently by exploiting an approximation technique for stable models which is closely related to the well-founded semantics [VGRS91]. The same approximation technique is employed in a powerful dynamic search heuristics.

One of the advantages of our implementation method is that it has linear space requirements. 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. See [NS96a, NS96b, NS97], for more detailed information on the implementation techniques. Our system has turned out to be significantly more efficient than other recent implementations of the stable model semantics (e.g. [BNNS94, CW96, SNV95, CMT96]) and it is the first system that can handle highly non-stratified programs with tens of thousands of ground rules.

Experimental Results

Some quite encouraging results have been obtained when applying the system. Reasonably large combinatorial problems (e.g., graph colorings and Hamiltonian circuits) have been solved using the system [NS96a, NS96b, NS97]. In **SAT** problems [NS97] the system shows similar performance as the best Davis-Putnam implementations (e.g., Crawford's tableau [CA96]) even for hard instances such as the random 3-SAT problems in the phase transition region [CA96]. The system has been applied to the planning domain [DNK97] where it provides comparable and occasionally significantly better performance than the best general purpose planners such as Graphplan.

In order to provide a flavor of the performance of the system we report on experiments involving challenging blocks world examples. We consider three test cases:

- large.c is a 15 block problem which is already difficult for the best domain independent planners such as Graphplan,
- large.d is a 17 block problem and
- large.e a 19 block problem.

We translate the examples to logic programs as described in the previous section. Table 1 contains two entries for each problem: one reporting the time needed to find a valid plan with the "optimal" number of situations given as input and one reporting the time needed to show optimality, i.e., that no plan (no stable model) exists when the number of situation is decreased by one. For example, for large.c, the available situations $time(t_0), \ldots, time(t_8)$ are given as a part of the program whereas for large.c', the available situations are $time(t_0), \ldots, time(t_7)$.

The times are execution times of the `smodels` module when given a parsed and grounded version of the program as input. The tests were performed using `smodels` version 1.10 and `parse` version 1.3 on a Pentium II 266MHz with 128MB of memory and the Linux 2.0.33 operating system. The test cases are available at http://saturn.hut.fi/pub/smodels/tests/bw-tests.tar.gz.

Availability

Our system is freely available at http://saturn.hut.fi/pub/smodels/. Documentation and an extensive set of

problem	number of situations	number of ground rules	running time (s)
large.c	9	90817	25
large.c'	8	80686	2
large.d	10	140848	50
large.d'	9	126715	10
large.e	11	209075	100
large.e'	10	190012	21

Table 1: Running times for blocks world examples

test cases can be obtained from 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.

5 Conclusions

We put forward logic programs with the stable model semantics as an interesting constraint programming paradigm. The aim is to bring advantages of knowledge representation techniques provided by logic programs to constraint programming in dynamic domains such as planning. There is already an implementation of the paradigm and some very encouraging experimental results have been obtained. The success seems to originate from the following points. (i) The built-in properties of the stable models (minimality and groundedness) enable compact representation of constraint satisfaction problems even when, e.g., the frame problem needs to be addressed. (ii) The search space for stable models does not grow with the number of atoms in the program but only the negative body literals contribute to the search space. This is different from, e.g., propositional satisfiability where each new atom can potentially increase the search space. This property is exploited in our implementation [NS96a]. (iii) Stable models can be approximated using well-founded models [VGRS91]. We have generalized this approximation scheme so that stable models can be dynamically approximated in each point of the search space [NS96a]. This approximation scheme provides a powerful pruning technique that is the key element behind the encouraging performance of our implementation.

Acknowledgements

This work has been supported by the Academy of Finland.

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.

[CA96] J.M. Crawford and L.D. Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81(1):31–57, 1996.

[CMT96] 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.

[CW96] 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.

[DNK97] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*, Toulouse, France, September 1997. Springer-Verlag. To appear.

[Elk90] C. Elkan. A rational reconstruction of nonmonotonic truth maintenance systems. *Artificial Intelligence*, 43:219–234, 1990.

[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.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the 7th International Conference on Logic Programming*, pages 579–597, Jerusalem, Israel, June 1990. The MIT Press.

[GL93] Michael Gelfond and Vladimir Lifschitz. Representing Actions and Change by Logic Programs. *Journal of Logic Programming*, 17:301–322, 1993.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In Michael J. O'Donnell, editor, *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, FRG, January 1987. ACM Press.

[KM97] A.C. Kakas and C. Mourlas. ACLP: Flexible solutions to complex problems. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 387–398, Dagstuhl, Germany, July 1997. Springer-Verlag.

[MT91] W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.

[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.

[NS96a] I. Niemelä and P. 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.

[NS96b] 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/.

[NS97] Ilkka Niemelä and Patrik Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 420–429, Dagstuhl, Germany, July 1997. Springer-Verlag.

[SNV95] 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.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–742, 1976.

[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.

[YCL96] J.-H. You, R. Cartwright, and M. Li. Iterative belief revision in extended logic programming. *Theoretical Computer Science*, 170:383–406, 1996.