Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm*

Ilkka Niemelä
Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science
P.O. Box 5400, FIN-02015 HUT, Finland
`Ilkka.Niemela@hut.fi`
www.tcs.hut.fi/~ini

Abstract

Logic programming with the stable model semantics is put forward as a novel constraint programming paradigm. This paradigm is interesting because it bring advantages of logic programming based knowledge representation techniques to constraint programming and because implementation methods for the stable model semantics for ground (variable-free) programs have advanced significantly in recent years. For a program with variables these methods need a grounding procedure for generating a variable-free program. As a practical approach to handling the grounding problem a subclass of logic programs, domain restricted programs, is proposed. This subclass enables efficient grounding procedures and serves as a basis for integrating built-in predicates and functions often needed in applications. It is shown that the novel paradigm embeds classical logical satisfiability and standard (finite domain) constraint satisfaction problems but seems to provide a more expressive framework from a knowledge representation point of view. The first steps towards a programming methodology for the new paradigm are taken by presenting solutions to standard constraint satisfaction problems, combinatorial graph problems and planning problems. An efficient implementation of the paradigm based on domain restricted programs has been developed. This is an extension of a previous implementation of the stable model semantics, the **Smodels** system, and is publicly available. It contains, e.g., built-in integer arithmetic integrated to stable model computation. The implementation is described briefly and some test results illustrating the current level of performance are reported.

*This is an extended version of a paper presented at the Workshop on Computational Aspects of Nonmonotonic Reasoning, Trento, Italy, May 30-June 1, 1998. The work has been supported by the Academy of Finland through Project 43963.

Contents

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

$$A \leftarrow B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n \tag{1}$$

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. A very natural definition for the solution sets is provided by *stable models* [17] which form one of the leading declarative semantics of logic programs. However, a logic programming system supporting the constraint interpretation of rules is very different from typical logic programming systems, such as Prolog implementations. Given a program the main task of such a system is to compute solution sets, i.e. stable models, for the program. This differs substantially from the usual logic programming paradigm which builds on goal-directed backward chaining query evaluation 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 previously mainly from the point of view of extending Prolog style goal-directed implementation techniques by allowing, e.g., arithmetic or finite domain constraints in the rules and by integrating the necessary constraint solvers into a logic programming system. This constraint logic programming paradigm [21] has been extended to include nonmonotonic reasoning capabilities such as abduction [22]. 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. Recently, similar ideas on employing rules as a methodology for expressing constraints capturing many kinds of problems such as combinatorial problems, graph problems and diagnosis, have been presented [10, 11, 15, 5, 6]. Especially close to our approach is the proposal put forward independently by Marek and Truszczyński [26] to use stable models as an alternative basis for logic programming where rules are interpreted as constraints in the same way as in our approach.

The novel constraint programming paradigm based on stable models is becoming increasingly interesting for practical purposes as implementations of the stable model semantics have advanced significantly in recent years. A number of new methods for computing stable models have been developed, e.g. [2, 7, 11, 29, 34], and the performance of the implementations of these methods has progressed rapidly. For example, the **Smodels** system [29] has provided quite encouraging results in many application areas. Reasonably large combinatorial problems (e.g., graph colorings and Hamiltonian circuits) have been solved using the system [29, 30]. It is able to handle computationally hard propositional satisfiability problems, e.g., random 3-SAT problems in the phase transition region [12] having around 250 variables. See, [33], for a comparison of **Smodels** and Crawford's tableau system [12] which is an efficient implementation of the Davis-Putnam method for deciding propositional satisfiability. The **Smodels** system has been applied to the planning domain [13] where it provides comparable and occasionally significantly better performance than efficient general purpose planners such as Graphplan [3].

There is also interesting new work on applying **Smodels** in verification of distributed systems. For example, it has been used with encouraging experimental results as a "fixed-point engine" for implementing efficient model checking algorithms for distributed systems [24] and as the basic inference engine in a partial-order verification method for Petri nets [20].

In this paper we aim to bring the novel paradigm yet another step closer to practical applications (i) by proposing a subclass of *domain restricted programs* as a basis for handling rules with variables and built-in predicates and functions and (ii) by developing programming methodology for the novel paradigm.

Rules with variables provide a compact and easily maintainable knowledge representation mechanism which is useful in many applications. However, the most competitive available methods for computing stable models, e.g. [2, 7, 11, 29, 34], are based on the idea of working with ground rules. This means that for a program with variables a *grounding procedure* is needed for computing a grounded program, i.e. a set of ground instances of the program, which is sound and complete in the sense that the grounded program has exactly the same stable models as the original program with variables. Clearly, implementing the grounding procedure by generating the whole ground instantiation of the program leads to unacceptable performance as the size of the ground instantiation can grow very fast.

Some research has already been done on efficient grounding procedures. Cholewiński [9] studies efficient grounding in the context of default logic. The SLG system developed by Chen and Warren [7] handles query-answering in the stable model semantics for non-ground programs in the following way: a query is first evaluated with respect to the well-founded semantics using SLG resolution (see [8, 32]) which produces a residual program for the subgoals that are relevant for the query. The residual program can be used for answering the query with respect to the well-founded semantics. Given some restrictions on the program, e.g. range restrictedness, the residual program is ground and it can be used for stable model computations by employing one of the methods working on ground programs. Combining SLG resolution and methods for ground programs offers an interesting approach to query-answering in the stable model semantics. Efficient WAM-based implementations for SLG resolution such as the XSB system [32] make this approach even more attractive.

For constraint programming the use of SLG resolution is not unproblematic for two reasons. First, in real applications the resulting ground programs might be large, e.g., containing more than 100 000 rules. This leads to efficiency problems as current implementation techniques for SLG resolution are not tuned for handling residual programs of this size. Second, the use of the residual program can lead to unsound results: the stable models of the residual program do not necessarily correspond to stable models of the original program [7]. The problem originates from the fact that SLG resolution is query-oriented, i.e., it works in a backward chaining way starting from an initial query, and the residual program contains only rules that are relevant for the query in this backward chaining sense. However, in constraint programming we are looking for a solution that satisfies *all* the constraints. In order to guarantee soundness we must choose the initial query for SLG resolution carefully so that all constraints are covered. Of course, a safe choice is to consider all predicates in the program but this could lead to performance problems. Determining an initial query that ensures soundness but is not unnecessarily large might not be trivial in all cases.

Example 1 Consider a situation where we have a choice between a and b and b leads to a chain of reasoning that can under some conditions, e.g., depending on other choices, lead to a conflict. So in a simplified setting we could have a set of rules of the form

$$\begin{array}{ll} a \leftarrow \text{not } b & c_1 \leftarrow b, \text{not } c_0 \\ b \leftarrow \text{not } a & c_2 \leftarrow c_1 \\ & \vdots \\ & c_n \leftarrow c_{n-1} \\ & c_0 \leftarrow c_n, \text{not } a \end{array}$$

When considering queries involving a and b in a backward chaining manner, the rules on the right hand side are not relevant. So given that we are interested in a and b, we might conclude that there are two solution sets, one containing a and one b. However, the solution set containing b but not a does not satisfy the constraints given by the rules on the right hand side. Hence, in order to guarantee soundness for SLG resolution we should include in the initial query some c_i as well. Sometimes it is quite straightforward to decide what needs to be included in the initial query but one should be careful not to overlook any possibility. ■

The soundness problem is avoided in the **Smodels** system [29] where the grounding procedure works bottom-up and handles range restricted programs in a sound and complete way. The grounding procedure seems to generate relatively small ground programs but does not appear to scale very well when the size of the generated ground program grows. The dlv system [15] implementing disjunctive stable model semantics has also an intelligent grounding procedure which is sound. The test results in [15] indicate that this grounder scales better than the original grounder in the **Smodels** system.

In this paper we propose a practical approach to solving the grounding problem where the subclass of programs to be handled is restricted. We put forward a subclass of programs, *domain restricted programs*, for which it is possible to devise efficient grounding procedures capable of handling efficiently cases where the resulting ground programs contain hundreds of thousands of rules and for which the modeling capabilities are still satisfactory for practical purposes. This subclass provides also a framework where external and built-in predicates and functions can be straightforwardly integrated into the novel paradigm.

We have implemented a new grounding procedure for the **Smodels** system based on domain restricted programs with non-recursive domain definitions. It works much more efficiently than the original grounder in **Smodels** because for non-recursive domains ground instances can be generated using efficient database techniques. The original grounding procedure in **Smodels** supports a wider class of programs, range restricted ones, but the difference does not seem to be significant and, in practice, range restricted rules can be extended to domain restricted ones with little effort. We start the development of a programming methodology for the novel constraint programming paradigm by working through standard examples from constraint satisfaction, combinatorial graph problems and planning.

The rest of the paper is organized as follows. First we explain the formal underpinnings of the new paradigm, the stable model semantics. Then we discuss rules with variables and introduce the class of domain restricted programs which serves as a basis for implementing efficient grounding procedures and for integrating built-in predicates and functions. We show that traditional Boolean constraints, i.e. propositional satisfiability, can be embedded in a simple way into logic programs with the stable model semantics ($\mathbf{LP_{SM}}$) and argue that from a knowledge representation point of view $\mathbf{LP_{SM}}$ is more expressive than propositional logic. We illustrate the use of $\mathbf{LP_{SM}}$ by discussing examples from constraint satisfaction problems, combinatorial graph problems and planning. Finally we report on our work on implementing the paradigm and give some experimental results that illustrate the current level of efficiency of our implementation. We finish with some concluding remarks.

2 Stable Model Semantics

In this section we formalize the constraint interpretation of rules by giving a declarative semantics for the solution sets in terms of *stable models* [17]. We study first ground (variable-free) rules, i.e., rules where the atoms are variable-free atomic formulae. In the next section we show how the semantics generalizes to rules with variables.

The starting point of the declarative semantics for solution sets is the intuitive reading of a rule of the form (1) as a constraint stating that if all the *positive body literals* B_i are included in the solution set and for each *negative body literal* not C_j the atom C_j is not included, then the *head* of the rule A must be included in the solution set. However, this reading does not capture all the important properties of solutions sets. In particular, it is desired that solution sets are minimal, i.e., a subset of a solution is no longer a solution. This means that, e.g., the program consisting of the rule $p \leftarrow p$ has the empty set as its unique solution set and that the set $\{p\}$ is not a valid solution. Nonetheless, minimality is not enough to capture the intended semantics. This can be seen by considering a program

$$p \leftarrow p$$
$$q \leftarrow \text{not } p$$

where $\{p\}$ and $\{q\}$ are minimal sets of atoms satisfying the intuitive reading of the rules. However, the set $\{p\}$ is somehow circularly justified: p is in the solution because p is in it! In applications such circularly grounded solutions are often unacceptable and the idea is to devise a semantics which guarantees both minimality and strong groundedness of the solutions. This can be achieved by defining solution sets as the stable models of the program.

The stable model semantics [17] generalizes in an elegant way the minimal model semantics of definite programs [36] 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* P^S 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 C in its body with $C \in S$ and

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

The reduct P^S can be seen as the set of potentially applicable rules given the stable model S, i.e., as the rules where the negative body literals are satisfied by the model. Note that in the reduct the negative body literals of the potentially applicable rules are removed and, hence, the rules are definite. The idea is to capture minimality and groundedness of a stable model by requiring that every atom in the model is a consequence of the potentially applicable rules given the model and every consequence of the potentially applicable rules is included in the model.

For a definite program P we can define a unique set of consequences $\text{Cl}(P)$ in various equivalent ways. We can see the program as a set of inference rules and then $\text{Cl}(P)$ is the deductive closure of the rules. On the other hand, a rule can be taken as a (definite) clause where the head is the positive literal and body literals are the negative literals of the clause. Then $\text{Cl}(P)$ is the unique minimal model of the clauses which coincides with the atomic logical consequences from the clauses.

Example 2 Consider the definite program P

$$p \leftarrow$$

$$
\begin{aligned}
q &\leftarrow p \\
r &\leftarrow p, q \\
t &\leftarrow r, s \\
s &\leftarrow s
\end{aligned}
$$

Now $\mathrm{Cl}(P) = \{p, q, r\}$. In order to verify this we can consider the program as a set of inference rules

$$\{\frac{}{p}, \frac{p}{q}, \frac{p,q}{r}, \frac{r,s}{t}, \frac{s}{s}\}$$

and then $\mathrm{Cl}(P) = \{p, q, r\}$ is the deductive closure of the rules, i.e., the least set of atoms closed under the rules. The closure can be constructed in a forward chaining manner by starting from the empty set and including a consequence of a rule to the set if the premises of the rule are already contained in the set. For example, for the rules corresponding to the program P, p is included first and then q followed by r. On the other hand, we can see the program as a set of definite clauses

$$\{p, q \vee \neg p, r \vee \neg p \vee \neg q, t \vee \neg r \vee \neg s, s \vee \neg s, \}$$

and $\mathrm{Cl}(P) = \{p, q, r\}$ is the unique (subset) minimal model of the clauses. Note that a set of atoms is the minimal model of a set of definite clauses iff (if and only if) it is the set of atomic consequences from the clauses. ■

Definition 2.1 *Let P be a ground (variable-free) program. Then a set of ground atoms S is a* stable model *of P iff $S = \mathrm{Cl}(P^S)$.*

Hence, the stability of a model means that it can reproduce itself in the sense that it is the fixed point of the operator $\Gamma_P(S) = \mathrm{Cl}(P^S)$.

Example 3 Program P

$$
\begin{aligned}
p &\leftarrow \text{not } q, r \\
q &\leftarrow \text{not } p \\
r &\leftarrow \text{not } s \\
s &\leftarrow \text{not } p
\end{aligned}
$$

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

$$
\begin{aligned}
p &\leftarrow r \\
r &\leftarrow
\end{aligned}
$$

and $S = \mathrm{Cl}(P^S)$. For instance, $S' = \{p, s\}$ is not a stable model of P because the reduct $P^{S'}$ is $\{p \leftarrow r\}$ and its deductive closure is $\{\} \neq S'$. In fact, P has another stable model $\{s, q\}$.

On the other hand, the program P'

$$
\begin{aligned}
f' &\leftarrow \text{not } f', f \\
f &\leftarrow
\end{aligned}
$$

has no stable models. To see this, assume that P' has a stable model S. Then $f \in S$. If $f' \in S$, then the reduct is $\{f \leftarrow\}$ and its closure does not include f'. Hence, $f' \notin S$ but then the reduct is

$$
\begin{aligned}
f' &\leftarrow f \\
f &\leftarrow
\end{aligned}
$$

whose closure contains f'. However, if we remove the second rule, then the resulting program has a stable model $\{\}$. ■

The definition of stable models captures the two key properties of solution sets.

- 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 has a justification in terms of the program, i.e., it is derivable from the reduct of the program with respect to the model.

As we saw above, a program can possess multiple stable models or none at all. The definition of a stable model is non-constructive in the sense that it does not provide any direct method for constructing a stable model from a program. However, given a candidate set of atoms it can be checked in linear time whether it is a stable model of a program as the unique minimal model of a set of definite clauses can be computed in linear time [14]. It has turned out that the problem of deciding whether a ground program has a stable model is NP-complete [25]. The techniques for computing stable models for ground programs have advanced rapidly in recent years and now there are systems capable of computing stable models for programs with tens of thousands of non-stratified rules.

It should be noticed that unlike models for classical logic stable models have a *nonmonotonic* behavior, i.e., adding new rules can lead to new models. Consider, e.g., the program P' in Example 3 which has no stable model. If we add a fact $f' \leftarrow$, then the resulting program has a unique stable model $\{f, f'\}$. In fact, stable models are closely related to other formalizations of nonmonotonic reasoning. Their origins are in Moore's autoepistemic logic [27] and logic programs can be seen as a special case of autoepistemic theories with the not operator treated as disbelief $\neg L$ where L is the belief operator of autoepistemic logic [17]. Stable models have been shown to correspond to other formalizations of nonmonotonic reasoning. For example, they coincide with the in and out sets computed by a justification-based truth maintenance system (TMS) when logic programming rules are treated as justifications in a TMS [16]. Furthermore, logic program rules can be seen as default rules in Reiter's default logic [31] and then stable models correspond to default extensions [18].

When using the rules as constraints, often *integrity constraints*, i.e., rules of the form

$$\leftarrow B_1, \ldots, B_m, \mathrm{not}\ C_1, \ldots, \mathrm{not}\ C_n \tag{2}$$

are needed. These kinds of rules are straightforward to encode using ordinary rules. This can be done, for example, by introducing two new atoms f and f' and a new rule $f' \leftarrow \mathrm{not}\ f', f$ and finally replacing every rule of the form (2) with one having f as its head. Another approach could be to incorporate the integrity constraints to the semantics by defining that a set of atoms S is a stable model for a program P (possibly with integrity constraints) iff S is a stable model for the ordinary rules in P and satisfies the integrity constraints in P. A set of atoms S satisfies a constraint of the form (2) iff it is not the case that $\{B_1, \ldots, B_m\} \subseteq S$ and $\{C_1, \ldots, C_n\} \cap S = \emptyset$.

Example 4 Consider the program P in Example 3 extended by two integrity constraints

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

This program has only one stable model $\{r, p\}$ as the other stable model of P, $\{s, q\}$, does not satisfy the first integrity constraint above. ■

Integrity constraints provide a powerful and simple to use constraint programming technique for pruning unwanted stable models as they cannot introduce new stable models but only can eliminate them.

Proposition 2.2 *Let P be a program and IC a set of integrity constraints. Then if S is a stable model of $P \cup IC$, then S is a stable model of P.*

3 Logic Programs with Variables

In this section we generalize the definition of solution sets (stable models) to programs with variables. Then we introduce a subclass of programs, *domain restricted programs*, as a basis for developing efficient grounding procedures needed by current methods for computing stable models which work on ground rules. Finally we indicate how this subclass provides a framework for incorporating external and built-in predicates and functions to our constraint programming paradigm.

The stable model semantics for ground programs presented in the previous section can be extended straightforwardly to programs with variables by employing the notion of *Herbrand models*. For a program, its *Herbrand universe* is the set of ground terms constructed from the constants and functions in the program and its *Herbrand base* is the set of atomic ground formulae built from the Herbrand universe and the predicate symbols of the program. A Herbrand model is a subset of the Herbrand base. Notice that the Herbrand base of a finite program is finite iff the program contains no function symbols.

Definition 3.1 *For a non-ground program P, the stable models of P are those of the ground instantiation P_H of P with respect to its Herbrand universe.*

Because P_H is the set of ground rules obtained from the rules in P by replacing variables in the rules by ground terms given in the Herbrand universe of P, stable models are Herbrand models of the program, i.e., subsets of its Herbrand base.

Example 5 In this paper we use a logic programming convention that the terms beginning with a capital letter are variables. Consider the program P

$$\begin{aligned}
&d_1(a) \leftarrow \\
&d_1(b) \leftarrow \\
&d_1(c) \leftarrow \\
&s(X,Y,Z) \leftarrow d_1(X), d_2(Y), d_3(Z), \text{not } d_3(X) \\
&s(X,X,X) \leftarrow d_1(X)
\end{aligned}$$

Its Herbrand universe is $\{a, b, c\}$ and Herbrand base is

$$\{d_1(a), d_1(b), d_1(c), d_2(a), d_2(b), \ldots, s(b,c,c), s(c,c,c)\}.$$

The ground instantiation of the program with respect to its Herbrand universe has 33 rules

$$\begin{aligned}
&d_1(a) \leftarrow \\
&d_1(b) \leftarrow \\
&d_1(c) \leftarrow \\
&s(a,a,a) \leftarrow d_1(a), d_2(a), d_3(a), \text{not } d_3(a)
\end{aligned}$$

$$s(a,a,b) \leftarrow d_1(a), d_2(a), d_3(b), \text{not } d_3(a)$$

$$\vdots$$

$$s(c,c,c) \leftarrow d_1(c)$$

and it has a unique stable model $\{d_1(a), d_1(b), d_1(c), s(a,a,a), s(b,b,b), s(c,c,c)\}$. ■

The role of variables in the new constraint programming approach is different from that in the usual Prolog style logic programming paradigm where variables stand for arbitrary terms providing recursive data structures built using function symbols. Here the idea is to keep the basic decision problems decidable and, thus, to avoid function symbols. The role of more complicated data structures is played by the stable models.

Hence, we are considering function-free programs whose Herbrand universe is always finite and thus the ground instantiation of the program is finite, too. Notice, however, that the ground instantiation can be very large compared to the original program. An upper bound on the size of the ground instantiation is rc^v where r is the number of rules and c the number of constants in the program and v is the upper bound on the number of distinct variables in any rule of the program. When implementing stable model computation for programs with variables the size of the ground instantiation may become problematic because the most competitive available methods for computing stable models, e.g. [2, 7, 11, 29, 34], are based on the idea of working with ground rules. Clearly, the approach where the whole ground instantiation of the program is computed first and stable models are computed from the ground instantiation can lead to very big overheads and unacceptable performance.

Example 6 Consider the program P in Example 5. The ground instantiation of P has 33 rules of which clearly the 27 ground rules related to the fourth rule do not contribute to the stable models of P. ■

Next we present a technique for handling the grounding problem in a way which works fairly efficiently in practice. Our idea is to restrict the class of programs so that a sound and complete subset of the ground instantiation can be computed efficiently. Soundness and completeness means that the subset has exactly the same stable models as the whole ground instantiation (and thus the original program).

We restrict the programs by requiring that every variable in a rule appears in some *domain predicate* for which it is easy to determine which of its ground instances are in the stable models of the original program. If the relevant ground instances of the domain predicates in a rule are easy to determine, then the relevant ground instances of the rule can be computed efficiently one rule at a time. In the following we make these ideas more precise. First we define the class of *domain restricted logic programs.*

Definition 3.2 *A logic program P is domain restricted for a set of predicates D if for each rule in P it holds that every variable in the rule appears also in a positive body literal of the rule for which the predicate is from D.*

Notice that a domain restricted program is also a range restricted one, i.e., if a variable appears in a rule it also appears in some positive body literal in the same rule. The extra condition here is that the positive literal must be a domain predicate from a given set. For example, the program P in Example 5 is domain restricted for $\{d_1, d_2, d_3\}$ but not for $\{d_1, d_2\}$.

Definition 3.3 *Let P be a logic program that is domain restricted with respect to D and $\hat{D}$ a set of ground instances of predicates in D. We define the program $P_{\hat{D}}$ as the set of ground* *instances of the rules in P such that each positive body literal with a predicate from D belongs to $\hat{D}$.*

Example 7 Consider the program P in Example 5 and set $D = \{d_1, d_2, d_3\}$. For $\hat{D} = \{d_1(a), d_2(a), d_3(a), d_3(b)\}$, $P_{\hat{D}}$ is

$$\begin{array}{l}
d_1(a) \leftarrow \\
d_1(b) \leftarrow \\
d_1(c) \leftarrow \\
s(a,a,a) \leftarrow d_1(a), d_2(a), d_3(a), \text{not } d_3(a) \\
s(a,a,b) \leftarrow d_1(a), d_2(a), d_3(b), \text{not } d_3(a) \\
s(a,a,a) \leftarrow d_1(a)
\end{array}$$

■

The next problem is to determine when a set of ground instances $\hat{D}$ of the domain predicates is sufficient in the sense that P and $P_{\hat{D}}$ have exactly the same stable models. For that we define a notion of completeness.

Definition 3.4 *Let P be a logic program that is domain restricted with respect to D and $\hat{D}$ a set of ground instances of predicates in D. Then $\hat{D}$ is complete for P iff for each ground instance $\hat{d}$ of a predicate $d \in D$ it holds that (i) if $\hat{d}$ is in some stable model of P, then $\hat{d} \in \hat{D}$ and (ii) if $\hat{d}$ is in some stable model of $P_{\hat{D}}$, then $\hat{d} \in \hat{D}$.*

A complete set of ground instances is sufficient as shown by the following theorem.

Theorem 3.5 *Let P be a domain restricted logic program with respect to D and $\hat{D}$ a subset of ground instances of predicates in D with respect to the Herbrand universe of P such that $\hat{D}$ is complete for P. Then P and $P_{\hat{D}}$ have the same stable models.*

Proof. Let P_H be the ground instantiation of P with respect to its Herbrand universe and let S be a stable model of P, i.e., $S = \text{Cl}(P_H^S)$. We show that S is a stable model of $P_{\hat{D}}$, i.e., that $S = \text{Cl}(P_{\hat{D}}^S)$ holds by establishing

$$\text{Cl}(P_H^S) = \text{Cl}(P_{\hat{D}}^S). \tag{3}$$

($\supseteq$) Clearly, $P_{\hat{D}} \subseteq P_H$, hence $P_{\hat{D}}^S \subseteq P_H^S$ and thus $\text{Cl}(P_{\hat{D}}^S) \subseteq \text{Cl}(P_H^S)$ by the monotonicity of Cl. ($\subseteq$) Let $a \in \text{Cl}(P_H^S)$. Taking the rules in P_H^S as inference rules, this means that there is a proof $a_0, a_1, \ldots, a_n = a$ where for each a_i, there is a rule $a_i \leftarrow b_1, ..., b_m \in P_H^S$ such that each b_l is some a_j with $j < i$. We show that for all $i = 0, 1, \ldots, n$, $a_i \in \text{Cl}(P_{\hat{D}}^S)$ by induction on i. Clearly, a_0 is a fact in the program P which cannot contain variables as the program is domain restricted. Hence, $a_0 \leftarrow \in P_{\hat{D}}$ and $a_0 \in \text{Cl}(P_{\hat{D}}^S)$. Assume that there is a proof $a_0, a_1, \ldots, a_i$. Hence, there is a rule r of the form $a_i \leftarrow b_1, ..., b_m \in P_H^S$ such that each b_l is some a_j with $j < i$. By the induction hypothesis, each $b_l \in \text{Cl}(P_{\hat{D}}^S)$. We show that $r \in P_{\hat{D}}^S$ holds which implies $a_i \in \text{Cl}(P_{\hat{D}}^S)$. As $r \in P_H^S$, there is rule $r' \in P$ such that r is obtained from r' by replacing the variables of r' by ground terms and then removing the negative body literals. For each domain predicate d in r', there is a ground instance b_j in r. But as each such ground instance b_j belongs to the stable model S of P, then by the property (i) for a complete set of ground instances $\hat{D}$,

$b_j \in \hat{D}$ and, hence, $r \in P^S_{\hat{D}}$. Thus, we have shown that $a \in \mathrm{Cl}(P^S_H)$ implies $a \in \mathrm{Cl}(P^S_{\hat{D}})$. Hence, (3) holds.

Similarly, we can show that a stable model of $P_{\hat{D}}$ is a stable model of P by using the property (ii) of a complete set of ground instances. □

Example 8 It is not very hard to come up with complete sets of ground instances of the domain predicates under some restrictions on the program. Below we provide a few examples:

- If a program P is range restricted, then we can take D to be the set consisting of every predicate appearing in a positive body literal and $\hat{D}$ to be the set including all their ground instances. Now $P_{\hat{D}}$ is the same as the ground instantiation of P and is clearly complete.

- Let a program P be domain restricted with respect to a set D of predicates which appear in the head of a rule in P only if the rule is a fact. Now we can take as the set of ground instances $\hat{D}$ the set of facts for the predicates from D in P. This set is complete for P.

- Let a program P be domain restricted with respect to a set of predicates D which have stratified [1] definitions in P depending only on predicates in D. Then a complete set of ground instances $\hat{D}$ is given by the unique stable model of the rules relevant to the predicates from D in P.

■

The idea is to employ the notion of domain restricted programs and Theorem 3.5 to find a subclass of programs with satisfactory expressive power for applications such that the grounding problem can still be solved efficiently. Notice that when a complete set of ground instances $\hat{D}$ has been computed, then a sufficient set of ground instances of the program, $P_{\hat{D}}$, can be produced very efficiently by considering one rule of the original program at a time. Hence, the key question is to find a subclass of programs for which a complete set of ground instances can be computed efficiently and the resulting set of ground rules is still of manageable size.

The last alternative in the example above, i.e., domain restricted programs with stratified domains, is a promising basis for handling the grounding problem. It allows an expressive language for defining domain predicates, i.e., that of stratified programs, and it facilitates the computation of the ground program as the domain predicates can be evaluated separate from the rest of the program one stratum at a time starting from the lowest strata. Furthermore, with stratified domain predicates we can tighten the definition of the resulting ground program $P_{\hat{D}}$ and exploit also the domain predicates in the negative body literals. This means that without loosing completeness we can limit the ground instances in $P_{\hat{D}}$ to those where each positive body literal with a predicate from D belongs to $\hat{D}$ *and where for each negative body literal* not C *with a predicate from* D*, the atom* C *is not included in* $\hat{D}$. These properties make this subclass of programs more attractive than, e.g., range restricted programs. This is because range restricted programs do not seem to share such computational advantages but do not offer much more modeling power to compensate the added complexity of the grounding problem. Note that it can be efficiently checked whether a program is domain restricted with stratified domains. This can be done by using the dependency graph of the program [1] to determine the set of all predicates D defined in a stratified way and then checking whether the program is domain restricted for this set D.

Stratified domain definitions allow also recursive definitions which means that determining the complete set of ground instances for the domain predicates involves recursive query evaluation techniques. It is not straightforward to obtain good performance when evaluating recursive rules and further complications arise as we aim to incorporate built-in predicates and functions in the domain definitions.

In order to circumvent these difficulties we are putting forward a subclass that we call *domain restricted programs with non-recursive domains*. This is a subclass of programs with stratified domains where the domain predicates have stratified but non-recursive definitions in the program depending only on other domain predicates. This means that, e.g., the transitive closure tc of a domain predicate rel given by the rules

$$\begin{aligned} tc(X,Y) &\leftarrow rel(X,Y) \\ tc(X,Y) &\leftarrow rel(X,Z), tc(Z,Y) \end{aligned}$$

cannot serve as a further domain predicate in this restricted subclass although it can when general stratified domains are allowed. Notice that the second rule is not domain restricted as tc cannot act as a domain predicate. In order to transform the rule to a domain restricted one, a new domain predicate for the variable Y needs to be added. A straightforward way to define such a domain predicate d is to use a rule $d(Y) \leftarrow rel(X,Y)$.

This subclass is an interesting compromise: (i) Non-recursive domain predicates allow a fair amount of modeling power corresponding to view definitions in relational databases. Hence, new domain predicates can be defined from the basic ones using, e.g., unions, intersections, differences, projections and joins. (ii) A complete set of ground instances for such a set of domain predicates can be computed efficiently *using database techniques* by evaluating the predicates one stratum at a time using database operations starting from the lowest strata. We demonstrate the expressivity of domain restricted programs with non-recursive domains by using this subclass in all the examples for the rest of the paper.

We end this section by pointing out that the notion of domain restricted programs with non-recursive domains provides a simple framework where different kinds of external and built-in predicates can be easily incorporated to rule bodies. This is because difficult semantical issues related to floundering are avoided and the evaluation of built-in predicates and functions is straightforward to integrate into the database techniques for computing the complete set of ground instances for the domain predicates. The idea is that we allow in the rule bodies predicates and functions that have their definitions given externally, e.g., in a relational database or as a built-in procedure implemented in some other programming language and use them to define further domain predicates.

Example 9 We illustrate the possibilities by an example of using built-in predicates and functions for integers. For example, we could have a built-in predicate $minus(X,Y,Z)$ corresponding to the set of facts $minus(x,y,z)$ where x,y,z are integers such that $z = x - y$. There is no need for this set of facts to be represented explicitly in the program and it could be implemented as an external procedure in some other programming language. For keeping the semantics clear it is enough to avoid floundering, i.e., a call to an external procedure where the parameters have uninitialized values. This can happen when an externally defined predicate has a variable as an argument and the range of ground terms over which the variable can vary is not clear. Domain restriction eliminates such a possibility.

Conceptually there is no problem in incorporating also built-in *functions* in the body literals of rules as long as we avoid floundering. Hence, we could have a built-in function $minus(X,Y)$ which we could use in the rule body. In what follows we use the usual infix notation for arithmetic functions.

The following shorthand is often handy for representing basic domains. We assume that a rule $d(n..m) \leftarrow$ stands for a set of facts $d(n) \leftarrow; d(n+1) \leftarrow; \ldots; d(m) \leftarrow$ if n, m are integers such that $n \leq m$.

Using these kinds of constructions it is then fairly easy to represent many interesting problems. We illustrate the ideas by a program that defines a (domain) predicate *grid* modeling grid graphs where nodes are pairs of integers and $((i,j),(i',j'))$ is an edge iff $|i-i'|+|j-j'|=1$.

So we are given some basic domain predicates

$$\begin{aligned} xdim(1..x) &\leftarrow \\ ydim(1..y) &\leftarrow \end{aligned}$$

which represent the dimensions of the grid graph where x and y are integers with $x, y \geq 1$. We define the predicate $grid(I, J, I', J')$ such that $((I,J),(I',J'))$ is an edge in the grid graph with dimension x, y as follows:

$$\begin{aligned} grid(I,J,I',J') \quad \leftarrow \quad & xdim(I), ydim(J), xdim(I'), ydim(J'), \\ & abs(I-I') + abs(J-J') = 1. \end{aligned}$$

where abs is a built-in function such that $abs(x) = |x|$. It should be noticed that external predicates and functions can be used for defining new domain predicates in a non-recursive fashion without any semantical difficulties as long as the program remains domain restricted. For example, the predicate *grid* can be taken as a new non-recursively defined domain predicate, although it depends not only on previously defined domain predicates $xdim, ydim$ but also on built-in functions and predicates '$-$', '$+$', 'abs', '$=$'. Hence, it can be used for defining further domain predicates. For example, we could code a node (I, J) as a single integer $X = (I-1)*y+J$ and define a corresponding graph for these nodes in terms of new domain predicates *edge* and *vertex*:

$$\begin{aligned} edge(X,Y) \quad &\leftarrow \quad grid(I,J,I',J'), X = (I-1)*y+J, Y = (I'-1)*y+J' \\ vertex(X) \quad &\leftarrow \quad edge(X,Y) \\ vertex(Y) \quad &\leftarrow \quad edge(X,Y) \end{aligned}$$

■

4 Relation to Propositional Satisfiability

Stable models are sets of atoms similar to propositional models. However, there are two significant differences. Stable models are *minimal* and *grounded.* We show that despite the differences propositional satisfiability (**SAT**) can be easily reduced to $\mathbf{LP_{SM}}$ by 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

$$\begin{aligned} \hat{a} &\leftarrow \text{not } a \\ a &\leftarrow \text{not } \hat{a} \end{aligned}$$

(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 rule $\leftarrow \text{not } c$.

Example 10 For a set of clauses

$$S = \{a \vee \neg b, \neg a \vee b\}$$

the translation $\mathrm{Tr}_{\mathrm{SAT}}(S)$ contains the rules

$$\begin{array}{lll}
\hat{a} \leftarrow \text{not } a & c_1 \leftarrow a & c_2 \leftarrow \hat{a} \\
a \leftarrow \text{not } \hat{a} & c_1 \leftarrow \hat{b} & c_2 \leftarrow b \\
\hat{b} \leftarrow \text{not } b & \leftarrow \text{not } c_1 & \leftarrow \text{not } c_2 \\
b \leftarrow \text{not } \hat{b} & &
\end{array}$$

■

You et al. [38] 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 4.1 *A set of clauses S has a model iff $\mathrm{Tr}_{\mathrm{SAT}}(S)$ has a stable model.*

The proposition shows that propositional satisfiability can be reduced to the problem of finding 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).

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 $\mathbf{LP}_{\mathbf{SM}}$ to **SAT** in a similar local modular fashion as we embedded **SAT** to $\mathbf{LP}_{\mathbf{SM}}$ above where small local changes in the input clauses lead to small local changes in the corresponding logic program. Notice that our translation from **SAT** to $\mathbf{LP}_{\mathbf{SM}}$ is very modular as each clause can be translated to a set of rules independently of other clauses. We can show that such a mapping in the reverse direction is not possible even 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, $P \cup F$ has a stable model iff $F \cup \mathrm{T}(P)$ is satisfiable. The intuition here is that for a modular mapping, adding an atom to the program should lead to a local change not involving the translation of the rest of the program.

Proposition 4.2 *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 $\{p\} \cup \mathrm{T}(P)$ is unsatisfiable. This implies that $P \cup \{p \leftarrow\}$ has no stable model which is clearly not the case. Hence, no modular mapping exists. □

We finish the section by discussing the implications of the results to the relative knowledge representation capabilities of propositional logic and $\mathbf{LP_{SM}}$. The modular mapping from **SAT** to $\mathbf{LP_{SM}}$ indicates that whenever there is a natural representation of (some part of) a domain using propositional logic, this can be used almost directly in the framework of $\mathbf{LP_{SM}}$ with small overhead through, e.g., the mapping above. Notice that the overhead caused by the introduction of an extra atom $\hat{a}$ for each propositional atom a is not significant because the state of the art implementations of stable model computation propagate the rules efficiently in both directions and can determine the other atom immediately whenever one of $a, \hat{a}$ becomes determined. Hence, the structure of the search space for propositional models of a set of clauses is similar to that for stable models of the corresponding set of rules. Also maintaining such a representation as rules is comparable to maintenance of the propositional representation because small changes in the clausal representation lead to small local changes in the corresponding rule set.

The last proposition implies that there could be situations having a natural representation in $\mathbf{LP_{SM}}$ but not when employing propositional logic in the sense that even simple updates like adding a new fact could lead to 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. Hence, the results in this section seem to strongly suggest that $\mathbf{LP_{SM}}$ provides a more expressive knowledge representation framework than classical propositional logic.

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 appear to provide interesting computational advantages. We return to this point in Section 8.

5 Relation to Constraint Satisfaction Problems

In the previous section we showed that Boolean constraints can be embedded into $\mathbf{LP_{SM}}$ using a simple local translation but that a similar local translation in the reverse direction is not possible. More general forms of constraints seem to have similar problems in capturing stable models but the other direction is still fairly straightforward. We demonstrate this by outlining a simple local mapping of constraint satisfaction problems (CSPs) to $\mathbf{LP_{SM}}$. Then we discuss some standard problems from the CSP literature.

A CSP consists of a set of variables with finite domains and a set of constraints. Each constraint specifies a set of allowed combinations of variables and values. A solution to the CSP is an assignment of values to variables such that each variable has exactly one value from its domain and all constraints are satisfied, i.e., for each constraint the assignment agrees with an allowed combination in the constraint. It is straightforward to represent such a problem using rules.

- For each domain value c in the CSP we adopt a constant c.
- For each domain d in the CSP we adopt a one-place predicate d and a set of facts $d(c_1) \leftarrow; \ldots; d(c_n) \leftarrow$ where $c_1, \ldots, c_n$ are the possible values of the domain d.
- For each variable v with the domain d in the CSP we adopt one-place predicates v and ov and two rules

$$\begin{aligned} v(X) &\leftarrow d(X), \text{not } ov(X) \\ ov(X) &\leftarrow d(X), d(Y), v(Y), X \neq Y \end{aligned}$$

where the predicate $ov(X)$ models the fact that the variable v has some other value than X.

- For each constraint co giving a set of allowed value combinations for a set of variables $v_1, \dots, v_j$ we take the fact $constraint(co) \leftarrow$ and for each allowed value combination $v_1 = c_1, \dots v_j = c_j$ a rule
$$sat(co) \leftarrow v_1(c_1), \dots, v_j(c_j)$$
and finally a rule
$$\leftarrow constraint(C), \text{not } sat(C).$$
stating that each constraint C must be satisfied.

Hence, a CSP can be represented in $\mathbf{LP_{SM}}$ in a very straightforward and easily maintainable way. For example, adding a new domain value c to the domain d can be done just by adding the corresponding fact $d(c) \leftarrow$. Sometimes constraints in a CSP are given in terms of *disallowed* combinations of values to variables. These kinds of constraints are also straightforward to represent with rules. For example, a disallowed combination $v_1 = c_1, \dots v_j = c_j$ can be captured with a rule
$$\leftarrow v_1(c_1), \dots, v_j(c_j).$$

5.1 Examples

Constraints often have a very natural representation directly as logic program rules. We illustrate this using a few standard examples from the CSP literature.

Pigeon: Put N pigeons into M holes so that there is at most one pigeon in a hole.

This problem can be solved with the following program

$$\begin{aligned}
&pos(P,H) \leftarrow pigeon(P), hole(H), \text{not } negpos(P,H)\\
&negpos(P,H) \leftarrow pigeon(P), hole(H), \text{not } pos(P,H)\\
&\leftarrow pigeon(P), hole(H), hole(H'), pos(P,H), pos(P,H'), H \neq H'\\
&\leftarrow pigeon(P), \text{not } hashole(P)\\
&hashole(P) \leftarrow pigeon(P), hole(H), pos(P,H)\\
&\leftarrow pigeon(P), pigeon(P'), hole(H), pos(P,H), pos(P',H), P \neq P'
\end{aligned}$$

where the domain predicates *hole* and *pigeon* give the available holes and pigeons. The idea is that $pos(p,h)$ gives a legal position of pigeon p in hole h. For each hole h and pigeon p, $pos(p,h)$ is modeled as a 'two-valued' atom, i.e., every stable model contains either it or its 'complement' $negpos(p,h)$. For representing the necessary constraints we use the technique based on integrity constraints for eliminating stable models not corresponding to valid assignments of pigeons to holes as follows. The first two rules establish the two-valued character of *pos* and provide the candidate stable models. The rest of the rules prune this set of models. The third rule is an integrity constraint stating that a pigeon cannot be in two holes and the following two rules that a pigeon must be in at least one hole. Note that we employ a new "defined" predicate *hashole* for representing the constraint. Such defined predicates appearing in integrity constraints do not introduce new stable models. The last rule says that there cannot be two pigeons in the same hole. The resulting program has a stable model iff the pigeon problem has a solution and a solution can be read from the stable model S as follows: $pos(p,h) \in S$ iff pigeon p is in hole h in the solution.

Queens: Place n queens on an $n \times n$ board so that no queen checks against any other queen.

This problem can be handled using the following program

$$
\begin{aligned}
&q(X,Y) \leftarrow d(X), d(Y), \text{not } negq(X,Y)\\
&negq(X,Y) \leftarrow d(X), d(Y), \text{not } q(X,Y)\\
&\leftarrow d(X), d(Y), d(X'), q(X,Y), q(X',Y), X' \neq X\\
&\leftarrow d(X), d(Y), d(Y'), q(X,Y), q(X,Y'), Y' \neq Y\\
&\leftarrow d(X), d(Y), d(X'), d(Y'), q(X,Y), q(X',Y'), X \neq X', Y \neq Y',\\
&\qquad abs(X - X') = abs(Y - Y')\\
&\leftarrow d(X), \text{not } hasq(X)\\
&hasq(X) \leftarrow d(X), d(Y), q(X,Y)
\end{aligned}
$$

where the domain predicate d provides the dimension of the board, i.e., $d(1..n) \leftarrow$ is included in the program. The idea is that $q(x,y)$ gives a legal position of a queen and it is again modeled as a 'two-valued' atom using the first two rules. The integrity constraint based technique is used for eliminating non-valid solutions. The third rule says that there cannot be two queens in the same row, the fourth eliminates two queens in the same column and the fifth two queens in the same diagonal. The last two rules say that there must be a queen in each column. Notice that in a problem like this (integer) arithmetic enables very compact representation of constraints as exemplified by the fifth rule. The resulting program has a stable model iff the queens problem has a solution and a solution can be read from the stable model S as follows: $q(x,y) \in S$ iff (x,y) is a legal position for a queen on the board.

Schur: Partition the integers $N = \{1, 2, ..., n\}$ into b boxes such that for any $x, y \in N$, (i) x and $2x$ are in different boxes and (ii) if x and y are in the same box, then $x + y$ is in a different box.

This problem can be solved using the following program

$$
\begin{aligned}
&pos(X,B) \leftarrow n(X), b(B), \text{not } negpos(X,B)\\
&negpos(X,B) \leftarrow n(X), b(B), \text{not } pos(X,B)\\
&\leftarrow n(X), b(B), b(B'), pos(X,B), pos(X,B'), B \neq B'\\
&\leftarrow n(X), \text{not } hasbox(X)\\
&hasbox(X) \leftarrow n(X), b(B), pos(X,B)\\
&\leftarrow n(X), b(B), pos(X,B), pos(2 * X, B)\\
&\leftarrow n(X), n(Y), b(B), pos(X,B), pos(Y,B), pos(X + Y, B)
\end{aligned}
$$

where the set of integers is given by the domain predicate n, the boxes by the predicate b and $pos(x,y)$ means that the integer x can be put in a box y. Again we use a combination of the 'two-valued' modeling technique and integrity constraints where the first five rules specify the two-valued character of pos and state that each number can be in exactly one box. The last two rules correspond directly to the conditions (i) and (ii) above.

Often combinatorial problems and constraint satisfaction problems have a large amount of symmetric solutions. By eliminating symmetries the search space of such a problem can be pruned considerably. In $\mathbf{LP}_{\mathbf{SM}}$ it is possible to do this declaratively without modifying the underlying search procedure for stable models by adding new rules. We illustrate this with the program above that allows symmetric solutions where the boxes are permuted. These can be eliminated by assuming a linear order for the boxes (naming them by integers) and by using the integrity constraint based technique leading to the following rules saying that for each integer x we should use the smallest available box, i.e., a box for which no smaller box is free of integers smaller than x.

$$\leftarrow n(X), b(B), pos(X,B), b(B'), B' < B, \text{not } occupied(X,B')$$
$$occupied(X,B) \leftarrow n(X), b(B), n(Y), Y < X, pos(Y,B)$$

Here $occupied(X,B)$ models the fact that there is some integer $Y < X$ occupying the box B.

6 Combinatorial Graph Problems

In this section we demonstrate the applicability of $\mathbf{LP_{SM}}$ to solving combinatorial graph problems by considering two typical problems: colorability and Hamiltonian circuits. The idea is to illustrate the knowledge representation capabilities of rules and show that $\mathbf{LP_{SM}}$ provides a compact and easily maintainable approach to describing such problems. Maintainability means that the rules specifying the correct solutions are independent of the graph under consideration and, thus, the graph can be changed without changing other parts of the program and similarly for important parameters for the problems, e.g., the number of available colors, which can be altered without modifying any other part of the program.

K-colorability

First consider the k-colorability problem, i.e., the problem of finding an assignment of one of k 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 $vertex(v) \leftarrow$ and $arc(v,u) \leftarrow$ and the available colors as facts $col(c) \leftarrow$. Then take the program with the rules below.

$$color(V,C) \leftarrow vertex(V), col(C), \text{not } othercolor(V,C)$$
$$othercolor(V,C) \leftarrow vertex(V), col(C), col(D), C \neq D, color(V,D)$$
$$\leftarrow arc(V,U), col(C), color(V,C), color(U,C)$$

The first two rules demonstrate a knowledge representation technique based on *rules with exceptions*. The first rule says that vertex V has color C unless there is some exception ($othercolor$) and the second rule specifies the exceptions. This provides the candidate solutions and the third rule eliminates those not corresponding to legal colorings. The program has a stable model iff there is a k-coloring of the graph. Note that the mapping from colorability to $\mathbf{LP_{SM}}$ is constructive in the sense that a k-coloring of the graph is directly obtained from a stable model by taking the facts of the form $color(v,c)$ that are true in the model.

Hamiltonian circuits

As an example of a problem which is not straightforward to map to a constraint satisfaction problem but which has a natural coding in $\mathbf{LP_{SM}}$ we consider the Hamiltonian circuit problem, i.e., the problem of finding a path in a graph that visits each vertex of the graph exactly once

$$
\begin{array}{l}
hc(V,U) \leftarrow arc(V,U), \text{not } otherroute(V,U) \\
otherroute(V,U) \leftarrow arc(V,U), arc(V,W), hc(V,W), U \neq W \\
otherroute(V,U) \leftarrow arc(V,U), arc(W,U), hc(W,U), V \neq W \\
reached(U) \leftarrow arc(V,U), hc(V,U), reached(V), \text{not } initialnode(V) \\
reached(U) \leftarrow arc(V,U), hc(V,U), initialnode(V) \\
\leftarrow vertex(V), \text{not } reached(V)
\end{array}
$$

Figure 1: A program for Hamiltonian circuits.

and returns to the starting vertex. Again assume that we have a database giving a graph in terms of atomic facts of the form $vertex(v) \leftarrow$ and $arc(v,u) \leftarrow$. We add to the facts the rules in Figure 1 and take one of the vertices v as the starting vertex ($initialnode(v) \leftarrow$ is added). The idea is that a fact $hc(v,u)$ holds if the arc (v,u) belongs to the Hamiltonian circuit. The first three rules ensure that for each node exactly one incoming and outgoing arc belong to the path. Here we employ rules with exceptions again. 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. The last three rules state that the path forms a cycle which visits all nodes and returns to the initial node. By exploiting the groundedness property of stable models the notion of a path forming a cycle can be captured in a compact way using a defined predicate *reached* and an integrity constraint. This leads to an easily maintainable representation where, e.g., the graph can be changed without changing the rules describing the conditions on the circuit. The resulting program has a stable model iff the graph has a Hamiltonian circuit. Note that the mapping from Hamiltonian circuits to $\mathbf{LP_{SM}}$ is constructive in the sense that a circuit is directly obtained from a stable model by taking the facts of the form $hc(v,u)$ in the model.

7 Planning

Planning provides a particularly interesting application area for nonmonotonic reasoning systems such as implementations of $\mathbf{LP_{SM}}$ 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 logic programming rules. For more detailed accounts, we refer the reader, e.g., to [19, 13].

In the blocks world we are given initial conditions concerning blocks on a table stating how they are stacked on top of each other and similar goal conditions. The aim is to generate a plan, i.e., a sequence of move operations starting from the initial configuration and leading to a configuration where 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 onto the table, c is moved onto a and finally b is moved onto 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. Planning is PSPACE-complete [4] and one way of restricting the problem to an NP-complete one is to bound the length of the plan. Hence, we assume that we have a limited number of situations $t_0, \ldots, t_n$ where t_0 is the initial situation and the available situations are given using facts of the form $time(t_i) \leftarrow$. A predicate *nextstate* specifies the order of the situations, i.e., for each $i = 0, \ldots, n-1$, $\mathit{nextstate}(t_{i+1}, t_i)$ holds. We employ predicates $\mathit{on}(X, Y, T)$ (X is on Y in the situation T) and $\mathit{moveop}(X, Y, T)$ (X is moved onto Y in the situation T) and assume that the available blocks are specified using facts of the form $\mathit{block}(b) \leftarrow$.

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

$$\begin{aligned}
&\mathit{on}(a, b, t_0) \leftarrow \\
&\mathit{on}(b, \mathit{table}, t_0) \leftarrow \\
&\mathit{on}(c, \mathit{table}, t_0) \leftarrow
\end{aligned}$$

In order to capture the goal conditions we employ a predicate $\mathit{goal}(T)$ which holds in any situation T where the goal conditions have been reached. For the example above, the resulting rule is

$$\begin{aligned}
\mathit{goal}(T) \leftarrow\ & \mathit{time}(T), \\
& \mathit{on}(b, c, T), \\
& \mathit{on}(c, a, T)
\end{aligned}$$

The idea is that a valid plan corresponds to a stable model where the goal condition has been achieved in some available situation. This is captured by the following two rules. The third rule ensures that if predicate $\mathit{goal}(T)$ holds in a situation, then it holds also in all subsequent situations. This is employed later in the operator descriptions.

$$\begin{aligned}
&\mathit{goal} \leftarrow \mathit{time}(T), \mathit{goal}(T) \\
&\leftarrow \text{not } \mathit{goal} \\
&\mathit{goal}(T_2) \leftarrow \mathit{nextstate}(T_2, T_1), \mathit{goal}(T_1)
\end{aligned}$$

In order to formalize the preconditions and effects of the move operator we use the following rules. The first rule specifies the preconditions and uses the technique based on exceptions. An instance of the move operator is applicable if there are no exceptions, i.e., the object to be moved and the destination are not covered and the move operator instance is not explicit blocked ($\mathit{blocked\_move}$). The exceptions are then listed below. The effect of the move operator can be stated directly as given in the second rule.

$$\begin{aligned}
\mathit{moveop}(X, Y, T) \leftarrow\ & \mathit{time}(T), \\
& \mathit{block}(X), \\
& \mathit{object}(Y), \\
& X \neq Y, \\
& \mathit{on_something}(X, T), \\
& \mathit{available}(Y, T), \\
& \text{not } \mathit{covered}(X, T), \\
& \text{not } \mathit{covered}(Y, T), \\
& \text{not } \mathit{blocked_move}(X, Y, T) \\
\mathit{on}(X, Y, T_2) \leftarrow\ & \mathit{block}(X), \\
& \mathit{object}(Y), \\
& \mathit{nextstate}(T_2, T_1), \\
& \mathit{moveop}(X, Y, T_1). \\
\mathit{on_something}(X, T) \leftarrow\ & \mathit{block}(X), \\
& \mathit{object}(Z), \\
& \mathit{time}(T), \\
& \mathit{on}(X, Z, T)
\end{aligned}$$

$$
\begin{aligned}
available(table, T) &\leftarrow time(T) \\
available(X, T) &\leftarrow block(X), \\
&\quad\ time(T), \\
&\quad\ on\_something(X, T) \\
covered(X, T) &\leftarrow block(Z), \\
&\quad\ block(X), \\
&\quad\ time(T), \\
&\quad\ on(Z, X, T) \\
object(table) &\leftarrow \\
object(X) &\leftarrow block(X)
\end{aligned}
$$

It is enough to provide a *frame axiom* only for the predicate *on* and this can be stated compactly as a rule with exceptions where the exceptional situations are captured using the predicate *moving*.

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

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.

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

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, i.e., which cannot be arbitrary interleaved. Computationally this seems advantageous as it decreases search space explosion due to interleavings of independent operators in linear planning.

$$
\begin{aligned}
blocked\_move(X, Y, T) &\leftarrow block(X), \\
&\quad\ object(Y), \\
&\quad\ object(Z), \\
&\quad\ time(T), \\
&\quad\ moveop(X, Z, T), \\
&\quad\ Y \neq Z.
\end{aligned}
$$

$$
\begin{aligned}
blocked\_move(X,Y,T) \leftarrow\ & block(X), \\
& object(Y), \\
& time(T), \\
& moving(Y,T) \\
blocked\_move(X,Y,T) \leftarrow\ & block(X), \\
& block(Y), \\
& block(Z), \\
& time(T), \\
& moveop(Z,Y,T), \\
& X \neq Z
\end{aligned}
$$

This kind of a 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:

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

For a program constructed like this it holds that the program has a stable model iff 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.

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.

8 Implementation

There is a C++ implementation of $\mathbf{LP}_{\mathbf{SM}}$ called **Smodels** [29, 30] which implements the stable model semantics for range restricted function-free normal programs. It includes two modules: (i) `smodels` which implements $\mathbf{LP}_{\mathbf{SM}}$ for ground programs and (ii) `parse` which is the grounding procedure for `smodels`. We have developed a new grounding procedure, `lparse`, for the **Smodels** system which is based on domain restricted programs with non-recursive domains. The user does not need to explicate the domain predicates but `lparse` detects them automatically by using the dependency graph for the program to find all non-recursively defined predicates. The ground instances of these predicates are then computed efficiently using database techniques. The more restricted class of programs handled by `lparse` enables it to work substantially more efficiently than `parse` because of the use of database techniques and because it is able to generate the ground instances of a rule independently of other rules.

Furthermore, it includes built-in predicates and functions for integer arithmetic. More details about the implementation techniques of `lparse` and its performance compared to `parse` and the dlv system can be found in [35].

The implementation of the stable model semantics for ground programs in the **Smodels** system is based on bottom-up backtracking search where the search space for stable models is pruned efficiently by exploiting the minimality and groundedness properties of stable models. This is based on an approximation technique for stable models which is closely related to the well-founded semantics [37]. The same approximation technique is employed in a powerful dynamic search heuristics.

One of the underlying ideas in the implementation 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) [28, 29]. This characterization, which follows from the minimality and groundedness properties of stable models, implies that only negative body literals contribute to the size of the potential search space and not all atoms in the program. Hence, it is possible to employ new defined atoms without compromising efficiency, e.g., in order to achieve a clearer or more succinct representation of a problem. By the full set characterization it is clear that atoms which appear only positively in the bodies do not increase the potential search space but this seems to hold also for other atoms, e.g., having stratified definitions. This is different from, e.g., propositional logic where each new atom potentially doubles the initial search space for models which is why the use of new defined atoms is typically avoided when applying propositional logic. Hence, it is conceivable that the possibility to use new defined atoms without computational overhead in $\mathbf{LP_{SM}}$ can lead to compact representations with attractive computational properties comparing favorable to formulations in classical logic. However, it is hard to compare representations of the same problem in different frameworks and more work is needed to determine how much can be gained in actual applications.

One of the advantages of the 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 [29, 30, 33], for more detailed information on the implementation techniques. **Smodels** has turned out to be significantly more efficient than other recent implementations of the stable model semantics (e.g. [2, 7, 34, 11]) and it is the first system that can handle highly non-stratified programs with tens of thousands of ground rules.

Availability

The **Smodels** system is freely available at www.tcs.hut.fi/pub/smodels/. Documentation and an extensive set of 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.

9 Experimental Results

In order to provide a flavor of the performance of the system we report some results on CSPs, combinatorial graph problems and blocks world planning using the domain restricted programs described in previous sections. Table 1 contains results on standard CSPs.

Table 2 presents results on combinatorial graph problems. As test graphs we have used

Table 1: Experimental results on standard constraint satisfaction problems.

Problem	Solutions	Time (s)
Pigeon 6/6	720	0.8
Pigeon 8/7	0	4.6
Pigeon 9/8	0	42.3
Queens 8	92	2.1
Queens 10	724	39.1
Queens 16	first	13.3
Queens 18	first	103
Queens 20	first	368
Schur 3/13	3	0.15
Schur 3/14	0	0.13
Schur 4/42	first	4.9
Schur 4/43	first	5.2
Schur 4/44	first	865
Schur 4/44	273	4650
Schur 4/45	0	6110

Table 2: Experimental results on combinatorial graph problems.

Problem	Graph	Solutions	Time (s)
3-col	p1000	0	3.3
3-col	p3000	0	10.1
3-col	p6000	0	20.3
4-col	p100	first	1.5
4-col	p300	first	13.1
4-col	p600	first	51.1
hc	p20	first	0.3
hc	p25	first	0.5
hc	p29	first	16.2
hc	p30	first	0.7

random planar graphs which are constructed by Delaunay triangulation of randomly inserted points in a plane. Here the plane function found in the Stanford GraphBase [23] has been used. For example, p1000 means a random planar graph with 1000 vertices.

Table 3 contains results on experiments involving challenging blocks world examples. We consider three test cases:

- large.c is a 15 blocks problem which is already difficult for advanced domain independent planners such as Graphplan [3],
- large.d is a 17 blocks problem and
- large.e a 19 blocks problem.

We translate the examples to logic programs as described in Section 7. Table 3 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

Table 3: Results for the blocks world examples.

problem	Number of steps	Number of ground rules	Time (s)
large.c	8	81682	23.2
	7	72528	6.0
large.d	9	128000	47.1
	8	115110	11.5
large.e	10	191622	101
	9	174100	17.3

example, for large.c, the available situations are $t_0, \ldots, t_8$ and, hence, the number of steps for applying operators is 8. This means that facts $time(t_0) \leftarrow; \ldots; time(t_8) \leftarrow$ are given as a part of the program. For showing optimality the fact $time(t_8) \leftarrow$ is removed.

The time reported for each test case is the sum of the execution times of `smodels` and `lparse` given a program with variables as input. Execution time was measured using the Unix `time` command and it is the sum of user and system time. The time used by `lparse` is usually small compared to the time needed by `smodels`, except when the number of ground rules is high. For example, for the largest planning example large.e it takes 11.5 s for `lparse` to generate the corresponding ground program and for the 3-colorability problem for p6000 it takes 12.9 s to generate the corresponding ground program with 161 839 rules. The tests were performed using `smodels` version 1.12 and `lparse` version 0.9.19 (beta) on a Pentium II 266MHz with 128MB of memory and the Linux 2.0.35 operating system. The test cases are available at `http://www.tcs.hut.fi/pub/smodels/tests/lp-csp-tests.tar.gz`.

10 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. However, the paradigm differs considerably from the usual logic programming methodology which is based on goal-directed backward chaining query evaluation and where variables stand for arbitrary terms providing recursive data structures built using function symbols. In the novel paradigm function symbols are not allowed and the role of more complicated data structures is played by the stable models. The idea is that a program is seen as a set of constraints describing valid solutions to a problem and the stable models of the program correspond to the solutions satisfying the constraints.

Implementation methods for the stable model semantics have advanced significantly in recent years. However, the most competitive available methods for computing stable models are based on the idea of working with ground rules. Hence, for a program with variables a grounding procedure is needed for generating a variable-free program. As a practical solution to handling the grounding problem we introduce a subclass of programs, domain restricted programs, as a basis for developing efficient grounding procedures. It also provides a framework for extending the paradigm with built-in predicates and functions.

We have taken the first steps towards a programming methodology for the new paradigm by presenting solutions to standard constraint satisfaction problems, combinatorial graph problems and planning problems. The aim has been to devise solutions that have attractive properties from a knowledge representation point of view. We are able to provide modular programs where the part of the program describing the instance (e.g., the graph in question) is independent from the part capturing the constraints for valid solutions (e.g., colorability conditions). Furthermore, our programs provide constructive solutions in the sense that a valid solution (e.g., an assignment of colors to vertices) can be read directly from a stable model of the program.

We have developed an efficient implementation of the paradigm based on domain restricted programs. This is an extension of a previous implementation of the stable model semantics, the **Smodels** system. In particular, we have developed a new efficient grounding procedure for **Smodels** which is based on domain restricted programs with non-recursive domains and which includes built-in functions and predicates for integer arithmetic. Test results on CSPs, graph problems and planning are provided to illustrate the current level of performance of our implementation. For example, for blocks world planning the results compare well with efficient domain-independent planners such as Graphplan.

There are several interesting topics for further research. In many applications more expressive rules could be useful, e.g. for representing general disjunctive conditions. What seems to be needed are classical inclusive and exclusive disjunctions instead of disjunctions with a minimal model interpretation studied intensively in the logic programming setting. An interesting question is whether such disjunctions can be incorporated without increasing computational complexity substantially, i.e., whether key decision problems remain in NP for ground programs. Another interesting extension would be to associate numerical weights and values to atoms in order to capture, e.g., knapsack type of problems.

Acknowledgements

The author would like to thank Tommi Syrjänen for implementing the new grounding procedure `lparse`.

References

[1] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Los Altos, 1988.

[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] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[4] Tom Bylander. Complexity results for planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 274–279, Sydney, Australia, August 1991. Morgan Kaufmann Publishers.

[5] Marco Cadoli and Luigi Palopoli. Circumscribing DATALOG: Expressive power and complexity. *Theoretical Computer Science*, 1–2:215–244, 1998.

[6] Marco Cadoli, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. NP-SPEC: An executable specification language for solving all problems in NP. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 16–30, San Antonio, Texas, January 1999. Springer-Verlag.

[7] 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.

[8] W. Chen and D.S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[9] P. Cholewiński. Towards programming in default logic. In *Proceedings of the 9th International Symposium on Methodologies for Intelligent Systems*, pages 223–232, Zakopane, Poland, June 1996. Springer-Verlag.

[10] 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.

[11] 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.

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

[13] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*, pages 169–181, Toulouse, France, September 1997. Springer-Verlag.

[14] 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.

[15] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarnello. The KR system dlv: Progress report, comparisons and benchmarks. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning*, pages 406–417, Trento, Italy, June 1998. Morgan Kaufmann Publishers.

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

[17] 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.

[18] 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.

[19] Michael Gelfond and Vladimir Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[20] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 240–254, Amsterdam, the Netherlands, March 1999. Springer-Verlag.

[21] 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.

[22] 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.

[23] D.E. Knuth. The Stanford GraphBase, 1993. Available at ftp://labrea.stanford.edu/.

[24] Xinxin Liu, C.R. Ramakrishnan, and Scott A. Smolka. Fully local and efficient evaluation of alternating fixed points. In Bernhard Steffen, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 5–19, Lisbon, Portugal, March/April 1998. Springer-Verlag.

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

[26] W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999. To appear.

[27] R.C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[28] 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.

[29] 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.

[30] I. Niemelä and P. 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.

[31] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[32] 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.

[33] P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research report A47, Helsinki University of Technology, Helsinki, Finland, August 1997. Available at http://www.tcs.hut.fi/pub/reports/A47.ps.gz.

[34] V.S. Subrahmanian, D. Nau, and C. Vago. WFS + branch and bound = stable models. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):362–377, 1995.

[35] T. Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical report B18, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, October 1998. Available at http://www.tcs.hut.fi/pub/reports/B18.ps.gz.

[36] 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.

[37] 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.

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