Stable Model Semantics of Weight Constraint Rules

Ilkka Niemelä[1], Patrik Simons[1], and Timo Soininen[2]

[1] Helsinki University of Technology, Dept. of Computer Science and Eng.,
Laboratory for Theoretical Computer Science,
P.O.Box 5400, FIN-02015 HUT, Finland
{Patrik.Simons,Ilkka.Niemela}@hut.fi

[2] Helsinki University of Technology, TAI Research Center and Lab. of Information Processing Science, P.O.Box 9555, FIN-02015 HUT, Finland
Timo.Soininen@hut.fi

Abstract. A generalization of logic program rules is proposed where rules are built from weight constraints with type information for each predicate instead of simple literals. These kinds of constraints are useful for concisely representing different kinds of choices as well as cardinality, cost and resource constraints in combinatorial problems such as product configuration. A declarative semantics for the rules is presented which generalizes the stable model semantics of normal logic programs. It is shown that for ground rules the complexity of the relevant decision problems stays in **NP**. The first implementation of the language handles a decidable subset where function symbols are not allowed. It is based on a new procedure for computing stable models for ground rules extending normal programs with choice and weight constructs and a compilation technique where a weight rule with variables is transformed to a set of such simpler ground rules.

1 Introduction

The implementation techniques for normal logic programs with the stable model semantics have advanced considerably during the last years. The performance of their state of the art implementations, e.g. the **smodels** system [12, 13], is approaching the level needed in realistic applications. Recently, logic program rules with the stable model semantics have also been proposed as a methodology for expressing constraints capturing for example combinatorial, graph and planning problems, see, e.g., [9, 11]. This indicates that interesting applications can be handled using normal programs and stable models. However, there are important aspects of combinatorial problems which do not seem to have a compact representation using normal rules. We explain these difficulties by first introducing the basic ideas behind the methodology of using rules for problem solving [9, 11]. Then we examine a number of examples involving cardinality, cost and resource constraints which are difficult to express using normal programs, i.e., programs consisting of rules without disjunction but with default negation in the body. On the basis of the examples we present an extension of normal rules where a generalized notion of cardinality constraints is used and which is suitable for handling choices with cardinality, cost and resource constraints in the examples.

When solving, e.g., a combinatorial problem using the stable model semantics the idea is to write a program such that the stable models of the program correspond to the solutions to the problem [9, 11]. As an example consider the 3-coloring problem where given a graph, we can build a program where for each vertex v in the graph we take the three rules on the left and for each edge (v, u) the three rules on the right

$$\begin{array}{ll} v(1) \leftarrow \mathit{not}\ v(2), \mathit{not}\ v(3) & \qquad \leftarrow v(1), u(1) \\ v(2) \leftarrow \mathit{not}\ v(1), \mathit{not}\ v(3) & \qquad \leftarrow v(2), u(2) \\ v(3) \leftarrow \mathit{not}\ v(1), \mathit{not}\ v(2) & \qquad \leftarrow v(3), u(3) \end{array}$$

Now a stable model of the program, which is a set of atoms of the form $v(n)$, gives a legal coloring of the graph where a node v is colored with the color n iff $v(n)$ is included in the stable model. These kinds of logic programming codings of different kinds of combinatorial, constraint satisfaction and planning problems can be found, e.g., in [9, 11]. The encodings demonstrate nicely the expressivity of normal programs. However, there are a number of conditions which are hard to capture using normal programs. For example, in the product configuration domain [14] choices with cardinality, cost and resource constraints need to be handled. Next we consider some motivating examples demonstrating the difficulties and show that extending normal rules by a suitable notion of cardinality constraints is an interesting approach to handling the problems. By a *cardinality constraint* we mean an expression written in the form

$$L \leq \{a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m\} \leq U\ . \tag{1}$$

The intuitive idea is that such a constraint is satisfied by any model (a set of atoms) where the cardinality of the subset of the literals satisfied by the model is between the integers L and U. For example, the cardinality constraint $1 \leq \{a, \mathit{not}\ b, \mathit{not}\ c\} \leq 2$ is satisfied by the model $\{a, b\}$ but not by $\{a\}$. These kinds of cardinality constraints are useful in a number of settings and rules extended with such constraints can be used to express different kinds of choices and cardinality restrictions. For example, vertex covers of size less than K could be captured in the following way. For a given graph, we build a program by including for each edge (v, u) a rule

$$1 \leq \{v, u\} \leftarrow$$

and then adding an integrity constraint

$$\leftarrow K \leq \{v_1, \ldots, v_n\}$$

where $\{v_1, \ldots, v_n\}$ is the set of vertices in the graph. The first rule expresses a choice saying that at least one end point for each edge should be selected and the second rule states a cardinality restriction saying that the cover must have size less than K. Now stable models of the program directly represent vertex covers of the graph. It seems that the choice rule cannot be expressed by normal rules without introducing additional atoms in the program and that there are no compact encodings of the cardinality restriction using normal rules.

For applications it is important to be able to work with first-order rules having variables. Hence, this kind of a cardinality constraint needs to be generalized to the first-order case where the set on which the constraint is imposed could be given compactly using expressions with variables. Consider, e.g., the problem of capturing cliques in a graph which is given by two relations *vertex* and *edge*, i.e., two sets of ground facts $vertex(v)$ and $edge(v,u)$ specifying the vertices and edges of the graph, respectively. The idea is to define the set of ground atoms in the constraint by attaching conditions to non-ground literals which are local to each constraint, i.e., using *conditional literals*, for example, in the following way:

$$0 \leq \{clique(X) : vertex(X)\} \leftarrow \tag{2}$$

where the set of atoms in the constraint consists of those instances of $clique(v)$ for which $vertex(v)$ holds. Such a rule chooses a subset of vertices and cliques. Cliques, i.e., subsets of vertices where each pair of vertices is connected by an edge, can be captured by including the rule

$$\leftarrow clique(X), clique(Y), not\ (X = Y), not\ edge(X,Y)\ .$$

It is also useful to allow both *local* and *global variables* in a rule. The scope of a local variable is one constraint, as for the variable X in (2), but the scope of a *global variable* is the whole rule. The first of the following rules capturing the colorings of a graph demonstrates the usefulness of this distinction.

$$1 \leq \{colored(V,C) : color(C)\} \leq 1 \leftarrow vertex(V) \tag{3}$$

$$\leftarrow edge(V,U), colored(V,C), colored(U,C) \tag{4}$$

Here V is a global variable in the first rule stating the requirement that for each vertex v exactly one instance of $colored(v,c)$ should be chosen such that $color(c)$ holds for the term c. The set of facts $color(c)$ provides the available colors.

As the examples show cardinality constraints are quite expressive and useful in practice. However, in for instance product configuration [14] applications there are conditions which are hard to capture even using cardinality constraints. One important class is resource or cost constraints. A typical example of these is the knapsack problem where the task is to choose a set of items i_j each having a weight w_j and value v_j such that the sum of the weights of the chosen items does not exceed a given limit W but the sum of the values exceeds a given limit V.

It turns out that these kinds of constraints can be captured by generalizing cardinality constraints in a suitable way which becomes obvious by noticing that a cardinality constraint of the form (1) can be seen as a linear inequality

$$L \leq a_1 + \cdots + a_n + \overline{b_1} + \cdots + \overline{b_m} \leq U$$

where $a_i, \overline{b_j}$ are variables with values 0 or 1 such that $x+\overline{x} = 1$ for all variables x. We can generalize this by allowing a real-valued coefficient for each variable, i.e., a weight for each atom in the cardinality constraint. Hence we are considering constraints of the form

$$L \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \mathit{not}\ b_1 = w_{b_1}, \ldots, \mathit{not}\ b_m = w_{b_m}\} \leq U \qquad (5)$$

where, e.g., w_{a_1} is a real-valued weight for the atom a_1. The idea is that a stable model satisfies the constraint if the sum of the weights of the literals satisfied by the model is between L and U. For example, $1.02 \leq \{a = 1.0, b = 0.02, \mathit{not}\ c = 0.04\} \leq 1.03$ is satisfied by $\{a, b, c\}$ but not by $\{a\}$. Hence, a weight constraint of the form (5) corresponds to a linear inequality

$$L \leq w_{a_1} \times a_1 + \cdots + w_{a_n} \times a_n + w_{b_1} \times \overline{b_1} + \cdots + w_{b_m} \times \overline{b_m} \leq U \qquad (6)$$

Using weight constraints the knapsack problem can be captured using the following rules:

$$\begin{aligned} 0 \leq \{i_1 = w_1, \ldots, i_n = w_n\} \leq W \leftarrow& \\ &\leftarrow \{i_1 = v_1, \ldots, i_n = v_n\} \leq V \end{aligned}$$

In the light of the examples it seems that weight constraints provide an expressive and uniform framework for handling large classes of combinatorial problems. In this paper we present a novel rule language which extends normal rules by taking weight constraints as the basic building blocks of the rules. Hence, the extended rules which we call *weight rules* are of the form

$$C_0 \leftarrow C_1, \ldots, C_n\ . \qquad (7)$$

Here each C_i is a weight constraint

$$\begin{aligned} L \leq \{&a_1 : c_1 = w_1, \ldots, a_n : c_n = w_n, \\ &\mathit{not}\ a_{n+1} : c_{n+1} = w_{n+1}, \ldots, \mathit{not}\ a_m : c_m = w_m\} \leq U \end{aligned} \qquad (8)$$

where a_i, c_i are atomic formulae possibly containing variables. These kinds of constraints are a first-order generalization of weight constraints of the form (5).

The weight rules are given a declarative nonmonotonic semantics that extends the stable model semantics of normal logic programs [4] and generalizes the propositional choice rules presented in [16] to the first-order case where type information and weight constraints can be used. Unlike the approaches based on associating priorities, preferences, costs, probabilities or certainty factors to rules (see e.g. [1, 8, 10, 6] and the references there), our aim is to provide a relatively simple way of associating weights or costs to atoms and representing constraints using the weights. Approaches such as NP-SPEC [3], constraint logic programs (CLP) and constraint satisfaction problems are not based on stable model semantics like ours and thus do not include default negation. In addition, our semantics treats the constraints, rules and choices uniformly unlike the CLP and NP-SPEC approaches. There is also some related work based on stable models. For example, in [2] priorities are added to integrity constraints. However, this is done to express *weak constraints*, as many of which as possible should be satisfied, and not weight constraints which must all be satisfied. In [5] several types of aggregates are integrated to Datalog in a framework based on stable models in order to express dynamic programming optimization problems. This contrasts with our approach which is not primarily intended to capture optimization. In addition, their approach covers only the subclass of programs with stratified negation and choice constructs. Our approach also differs from the main semantics of disjunctive logic programs in that they are based on subset minimal choices through disjunction while we support a general notion of cardinality constraints.

The computational complexity of the decision problem for the language is analyzed and found to remain in **NP** for ground rules. The first implementation of the language handles a decidable subset of weight rules where function symbols are not allowed. Although the semantics of the language is based on real-valued weights, the implementation handles only integer weights in order to avoid problems arising from finite precision of real number arithmetic. The implementation is based on the **smodels-2** procedure [15] which is a new extended version of the **smodels** procedure [12, 13]. It computes stable models for ground logic programs but supports several types of rules extending normal logic programs. Our language extends that handled by **smodels-2** further: it is first-order with conditional literals, variables, and built-in functions; both upper and lower bounds of a constraint can be given and a weight constraint is allowed also in the head of a rule. However, we show that it is possible to translate a set of weight rules containing variables to a set of simple ground rules supported by **smodels-2**. This provides the basis for our implementation.

2 Weight Constraint Rules

We extend logic program rules by allowing weight constraints of the type (8) with conditional literals that have real-valued weights. First we develop a semantics for ground rules and then we show how to generalize this to rules with variables.

2.1 Ground Rules

The basic building block of a weight constraint is a *conditional atom* which is an expression of the form $p : q$ where the *proper part* p and *conditional part* q are atomic formulae. In ground rules formulae p and q are variable-free (ground) atoms. If q is $\top$, i.e., always valid, it is typically omitted. A *conditional literal* is a conditional atom or its negation, an expression of the form *not* $p : q$. Note that the *not* is intended as a nonmonotonic, default negation.

A *weight constraint* C is an expression of the form:

$$\mathrm{l}(C) \leq \mathrm{lit}(C) \leq \mathrm{u}(C)$$

where $\mathrm{lit}(C)$ is a set of conditional literals and $\mathrm{l}(C), \mathrm{u}(C)$ two real numbers denoting the *lower* and *upper bounds*, respectively. The bounds $\mathrm{l}(C), \mathrm{u}(C)$ can also be missing in which case we denote them by $\mathrm{l}(C) = -\infty, \mathrm{u}(C) = \infty$, respectively. To each constraint C we associate a *local weight function* $\mathrm{w}(C)$ from the set of literals in C to the real numbers, typically specified directly as in the constraint for C below:

$$2.1 \leq \{p : d_1 = 1.1, not\ q : d_2 = 1.0001\}$$

where, e.g., $\mathrm{w}(C)(not\ q) = 1.0001$ and $\mathrm{u}(C) = \infty$. The extension to allow $<$ in the constraints is straightforward but for brevity we discuss only $\leq$. Finally, a *weight program* is a set of *weight rules*, i.e., expressions of the form (7) where each C_i is a weight constraint and where the head C_0 contains no negative literals.

Our semantics for weight rules generalizes the stable model semantics for normal logic programs and is given in terms of models that are sets of atoms. First we define when a model satisfies a rule and then using this concept the notion of stable models.

Definition 1. *A set of atoms S* satisfies *a weight constraint C ($S \models C$) iff for the weight* $\mathrm{W}(C,S)$ *of C in S, $\mathrm{l}(C) \leq \mathrm{W}(C,S) \leq \mathrm{u}(C)$ holds where*

$$\mathrm{W}(C,S) = \sum_{p \in \mathrm{plit}(C,S)} \mathrm{w}(C)(p) + \sum_{not\ p \in \mathrm{nlit}(C,S)} \mathrm{w}(C)(not\ p)$$

with $\mathrm{plit}(C,S) = \{p \mid p : q \in \mathrm{lit}(C), \{p,q\} \subseteq S\}$ *and* $\mathrm{nlit}(C,S) = \{not\ p \mid not\ p : q \in \mathrm{lit}(C), p \notin S, q \in S\}$ *which are the positive and negative literals satisfied by S, respectively. A rule r of the form (7) is satisfied by S ($S \models r$) iff S satisfies C_0 whenever it satisfies $C_1, \ldots, C_n$.*

We also allow *integrity constraints*, i.e., rules without the *head* constraint C_0, which are satisfied if at least one of the *body* constraints $C_1, \ldots, C_n$ is not.

Example 1. Consider the weight constraints

$$C_1 : \ 2 \leq \{p : d_1 = 1, not\ q : d_1 = 2, r : d_2 = 1.5\} \leq 5$$
$$C_2 : \ 2 \leq \{p : d_2 = 1, not\ q : d_2 = 2, r : d_1 = 1.5\} \leq 5$$

and a set of atoms $S = \{p, d_1, r\}$. Now $\mathrm{plit}(C_1, S) = \{p\}$ and $\mathrm{nlit}(C_1, S) = \{not\ q\}$ and, hence, $\mathrm{W}(C_1, S) = 1 + 2 = 3$. Similarly, $\mathrm{W}(C_2, S) = 1.5$. Thus, $S \models C_1$ but $S \not\models C_2$ and $S \models C_1 \leftarrow C_2$ but $S \not\models C_2 \leftarrow C_1$. Moreover, $S \models \leftarrow C_1, C_2$ but $S \not\models \leftarrow C_1$.

We define stable models first for weight programs with non-negative weights. We then show how the general case, i.e., programs with negative weights reduce to this case. In the definition we need the notion of a *deductive closure* of rules in a special form

$$P \leftarrow C_1, \ldots, C_n$$

where P is a ground atom and each weight constraint C_i contains only positive literals and non-negative weights, and has only a lower bound condition. We call such rules *Horn weight rules.* A set of atoms is *closed* under a set of rules if each rule is satisfied by the atom set. A set of Horn weight rules P has a unique smallest set of atoms closed under P. We call it the *deductive closure* and denote it by $\mathrm{cl}(P)$. The uniqueness is implied by the fact that Horn weight rules are monotonic, i.e., if the body of a rule is satisfied by a model S, then it is satisfied by any superset of S. Note that the closure can be constructed iteratively by starting from the empty set of atoms and iterating over the set of rules and updating the set of atoms with the head of a rule not yet satisfied until no unsatisfied rules are left.

Example 2. Consider a set of Horn weight rules P

$$
\begin{aligned}
&a \leftarrow 1 \leq \{a = 1\} \\
&b \leftarrow 0 \leq \{b = 100\} \\
&c \leftarrow 6 \leq \{b = 5, d = 1\}, 2 \leq \{b = 2, a = 2\}
\end{aligned}
$$

The deductive closure of P is the set of atoms $\{b\}$ which can be constructed iteratively by starting from the empty set and realizing that the body of the second rule is satisfied by the empty set and, hence, b should be added to the closure. This set is already closed under the rules. If a rule

$$d \leftarrow 1 \leq \{a = 1, b = 1, c = 1\}$$

is added, then the closure is $\{b, d, c\}$.

Stable models for programs with non-negative weights are defined in the following way using the concept of a reduct. The idea is to define a stable model of a program P as an atom set S that *satisfies* all rules of P and that is the *deductive closure* of a *reduct* of P w.r.t. S. The role of the reduct is to provide the possible justifications for the atoms in S. Each atom in a stable model is justified by the program P in the sense that it is derivable from the reduct.

We introduce the reduct in two steps. First we define the reduct of a constraint and then generalize this to rules. The reduct C^S of a constraint C w.r.t. to a set of atoms S is the constraint

$$L' \leq \{p : q = w \mid p : q = w \in \mathrm{lit}(C)\}$$

where $L' = \mathrm{l}(C) - \sum_{not\ p \in \mathrm{nlit}(C,S)} \mathrm{w}(C)(not\ p)$. Hence, in the reduct all negative literals and the upper bound are removed and the lower bound is decreased by w for each $not\ p : q = w$ satisfied by S. The idea here is that for negative literals satisfied by S, their weights contribute to satisfying the lower bound. However, this does not yet capture the condition part of the negative literals satisfied by S. In order to guarantee that the conditions are justified by the program a set $\mathrm{j}(C, S)$ of *justification constraints* is used:

$$\mathrm{j}(C, S) = \{1 \leq \{q = 1\} \mid not\ p : q = w \in \mathrm{lit}(C), p \notin S, q \in S\}$$

For example, for a constraint C: $3 \leq \{not\ p : q = 2, not\ r : p = 3, p : q = 1\} \leq 4$ and a set $S = \{q\}$ we get the reduct and justification constraint

$$C^S = 1 \leq \{p : q = 1\} \qquad \mathrm{j}(C, S) = \{1 \leq \{q = 1\}\}$$

The reduct P^S for a program P w.r.t. a set of atoms S is a set of Horn weight rules which contains a rule r' with an atom p as the head if $p \in S$ and there is a rule $r \in P$ such that $p : q = w$ appears in the head with $q \in S$, and the upper bounds of the constraints in the body of r are satisfied by S. The condition q is moved to the body as q is the justification condition for p and the body of r' is obtained by taking the reduct of the constraints in the body of r and adding the corresponding justification constraints. Formally the reduct is defined as follows.

Definition 2. *Let P be a weight program with non-negative weights and S a set of atoms. The reduct P^S of P w.r.t. S is defined by*

$$P^S = \{p \leftarrow 1 \leq \{q = 1\}, C_1^S, \mathrm{j}(C_1, S), \ldots, C_n^S, \mathrm{j}(C_n, S) \mid C_0 \leftarrow C_1, \ldots, C_n \in P,$$
$$p : q = w \in \mathrm{lit}(C_0), \{p, q\} \subseteq S, \text{ for all } i = 1, \ldots, n, \mathrm{W}(C_i, S) \leq \mathrm{u}(C_i)\}$$

Definition 3. *Let P be a weight program with non-negative weights. Then S is a* stable model *of P iff the following two conditions hold:*
(i) $S \models P$,
(ii) $S = \mathrm{cl}(P^S)$.

Example 3. Consider first program P_1 demonstrating the role of justification constraints.

$$0 \leq \{p : p = 2\} \leq 2 \leftarrow$$
$$2 \leq \{p = 2\} \leq 2 \leftarrow 2 \leq \{not\ q : p = 3\}$$

The empty set is a stable model of P_1 because it satisfies both rules and the reduct $P_1^{\emptyset} = \emptyset$. For $S = \{p\}$ the reduct P_1^S is

$$p \leftarrow 1 \leq \{p = 1\}$$
$$p \leftarrow -1 \leq \{\}, 1 \leq \{p = 1\},$$

Now $\mathrm{cl}(P_1^S) = \{\}$ implying that S is not a stable model although it satisfies P_1.

Consider the program P_2

$$2 \leq \{b = 2, c = 3\} \leq 4 \leftarrow 2 \leq \{not\ a = 2, b = 4\} \leq 5$$

The definition of stable models guarantees that atoms in a model must be justifiable by the program in terms of the reduct and thus, e.g., P_2 cannot have a stable model containing a. The empty set is not a stable model as $\{\} \not\models P_2$. The same holds if $S = \{b\}$ because the reduct P_2^S is empty since the upper bound in the body is exceeded. However, $S = \{c\}$ is a stable model as $S \models P_2$ and $\mathrm{cl}(P_2^S) = \{c\}$ where $P_2^S = \{c \leftarrow 0 \leq \{b = 4\}\}$. Note that as there are no conditional literals, no justification constraints are needed.

Our definition is a generalization of the stable model semantics for normal programs as a simple literal l in a normal program can be seen as a shorthand for $1 \leq \{l = 1\} \leq 1$. Thus, e.g., a normal rule $a \leftarrow b, not\ c$ is a shorthand for

$$1 \leq \{a = 1\} \leq 1 \leftarrow 1 \leq \{b = 1\} \leq 1, 1 \leq \{not\ c = 1\} \leq 1 \ .$$

The reduct of the rule w.r.t. $S = \{a, b\}$ is

$$a \leftarrow 1 \leq \{b = 1\}, 0 \leq \{\}$$

whose closure is $\{\}$ and, hence, S is not a stable model of the rule although it satisfies the rule. We use this abbreviation frequently and, furthermore, we often omit the weight of a literal if it is 1.

Definition 3 does not cover constraints with negative weights. However, it turns out that these can be transformed to constraints with non-negative weights by simple linear algebraic manipulation which translates a constraint C

$$L \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \leq U$$

to an equivalent form C' with only non-negative weights

$$\begin{aligned} &L + \sum_{w_{a_i} < 0} |w_{a_i}| + \sum_{w_{b_i} < 0} |w_{b_i}| \leq \\ &\quad \{\underbrace{a_i = w_{a_i}}_{w_{a_i} \geq 0}, \ldots, \underbrace{b_j = |w_{b_j}|}_{w_{b_j} < 0}, \ldots, \underbrace{not\ b_k = w_{b_k}}_{w_{b_k} \geq 0}, \ldots, \underbrace{not\ a_l = |w_{a_l}|}_{w_{a_l} < 0}, \ldots\} \\ &\qquad\qquad\qquad\qquad \leq U + \sum_{w_{a_i} < 0} |w_{a_i}| + \sum_{w_{b_i} < 0} |w_{b_i}| \end{aligned}$$

where negative weights are complemented together with the corresponding literal and the sum of absolute values of all negative weights is added to the bounds.

The equivalence of C and C' can be seen using the linear inequality (6) for C. We can eliminate any negative weight w_{a_i} by adding

$$|w_{a_i}| \times (a_i + \overline{a_i}) = |w_{a_i}| \qquad (a_i + \overline{a_i} = 1)$$

to the inequality. This leaves the term $|w_{a_i}| \times \overline{a_i}$ in the middle corresponding to $not\ a_i = |w_{a_i}|$. Similarly all negative weights w_{b_i} can be eliminated.

Example 4. Consider the rule

$$a \leftarrow -1 \leq \{a = -4, not\ b = -1\} \leq 0$$

where we can eliminate the negative weights in the body using the method above. Then the resulting rule is

$$a \leftarrow 4 \leq \{not\ a = 4, b = 1\} \leq 5$$

Let $S = \{a\}$. Then the reduct for the resulting rule is $\{a \leftarrow 4 \leq \{b = 1\}\}$. Hence, S is not a stable model of the rule.

We have demonstrated the expressiveness of weight constraints already by a number of examples in the introduction. Here we show how they capture propositional logic and the rule-based configuration language in [16].

Example 5. (i) We can reduce *propositional satisfiability* to the problem of finding a stable model in the following way without introducing any additional atoms. Consider a set T of propositional clauses containing the atoms $a_1, \ldots, a_k$. If we construct a program with a rule $0 \leq \{a_1, \ldots, a_k\} \leftarrow$ together with a rule

$$\leftarrow \textit{not}\ a_1, \ldots, \textit{not}\ a_n, a_{n+1}, \ldots, a_m$$

for each clause $a_1 \vee \cdots \vee a_n \vee \neg a_{n+1} \vee \cdots \vee \neg a_m \in T$, then the resulting program has a stable model iff T is satisfiable. Furthermore, each stable model corresponds directly to a propositional model (the atoms in the stable model are true and the other atoms are false).

(ii) Weight rules generalize also the rule-based configuration language in [16]. For example, an inclusive choice-rule $a \mid b \mid c \leftarrow d$ can be represented as

$$1 \leq \{a, b, c\} \leftarrow d$$

and an exclusive choice-rule $a \oplus b \oplus c \leftarrow \textit{not}\ d$ is captured by

$$1 \leq \{a, b, c\} \leq 1 \leftarrow \textit{not}\ d$$

2.2 First-Order Rules

Now we consider the first-order case where rules have variables. The semantics is obtained by the use of Herbrand models. The *Herbrand universe* of the program is defined as usual, i.e., it consists of terms constructible from constants and functions appearing in the program. The *Herbrand base* is the set of ground atoms constructible from the predicate symbols and the Herbrand universe of the program. As noted in the introduction, it is useful to provide *local* variables for a constraint as well as *global*, i.e., universally quantified, variables for a rule. A constraint C with local variables $X_1, \ldots, X_n$ is written

$$\mathrm{l}(C) \leq \langle X_1, \ldots, X_n \rangle \mathrm{lit}(C) \leq \mathrm{u}(C)$$

and the variables not local to a constraint are global. With this distinction we define the *Herbrand instantiation* of a weight program which consists of all ground rules obtainable in the following way. First each global variable in a rule is substituted with a ground term from the Herbrand universe. Now the rule contains only local variables. Then for each constraint C, the set of literals in the ground instance of C is obtained by taking every substitution instance of the literals where the local variables are replaced by terms from the Herbrand universe. For example, for the rule

$$1 \leq \langle X \rangle \{p(X, Y) : d(X, Y)\} \leq 1 \leftarrow q(Y)$$

Y is a global variable and X is a local variable for the constraint in the head. If the Herbrand universe is $\{a, b\}$, the Herbrand instantiation of the rule is

$$1 \leq \{p(a,a) : d(a,a), p(b,a) : d(b,a)\} \leq 1 \leftarrow q(a)$$
$$1 \leq \{p(a,b) : d(a,b), p(b,b) : d(b,b)\} \leq 1 \leftarrow q(b)$$

With local and global variables many problems can be expressed quite succinctly. Consider, e.g., the rule (3) for assigning colors to vertices in a graph. Notice that it is not necessary to explicate the local variables for a constraint if we use a convention that all variables appearing in more than one constraint are global and all other variables are local. This convention is used in (3) and also in the rest of the paper.

The stable models of a weight program with variables are defined using the Herbrand instantiation of the program.

Definition 4. *Let P be a weight program with variables. Then a set of ground atoms S is a stable model of P iff it is a stable model of the Herbrand instantiation of P.*

Note that the definition allows a fairly dynamic notion of weights by associating a local weight function with each constraint in every ground instance of a rule.

3 Computational Aspects

Although weight programs extend e.g. normal logic programs considerably, the computational complexity remains unaffected, i.e., stays in **NP**.

Theorem 1. *The problem of deciding whether a ground weight program has a stable model is* ***NP****-complete.*

Proof. **NP**-hardness is implied by the fact that weight programs generalize normal logic programs with the stable model semantics for which **NP**-completeness has been shown [7]. Containment in **NP** follows from the property that given a set of atoms it can be checked in polynomial time whether the set is a stable model of a given program. The crucial step here is the computation of the closure of the reduct which can be done iteratively in polynomial time by starting from the empty set of atoms S and iterating over the set of rules and updating S with the heads of the rules not yet satisfied until no unsatisfied rules are left.

3.1 Implementation

The full first-order case of weight rules is clearly undecidable. We have developed an implementation for a decidable subclass in which function symbols are not allowed. This subclass offers an interesting trade-off between expressiveness and implementability which seems adequate for many practical purposes. The implementation is based on the **smodels-2** procedure [15] (available at http://www.tcs.hut.fi/pub/smodels/) and a compilation technique where a rule with variables is transformed to a set of simpler ground propagation rules. The **smodels-2** procedure, which is a new extended version of the **smodels** procedure [12, 13], computes stable models for ground logic programs but supports new types of rules that extends the normal rules. The implementation, available at http://www.tcs.hut.fi/smodels/lparse/, is a front-end that maps a general weight program to ground rules from which the stable models of the original program are computed by the **smodels-2** procedure. We first define the subclass of rules that our current implementation accepts and then explain the compilation technique.

The current implementation works with *domain-restricted* rules where each variable in a rule must appear in a positive domain predicate in the same rule and for each conditional literal the condition part is a domain predicate. A *domain predicate* is one that is defined non-recursively using other domain predicates with normal logic program rules. This means that a domain predicate can appear as the head of a rule only when each constraint in the rule is a simple literal.

Example 6. Consider the rules (3–4) capturing colorings of a graph. Assume that the predicates *vertex*, *edge*, and *color* are defined non-recursively, e.g., by a set of ground facts. Then they can be taken as domain predicates and the rule (3) is domain restricted but (4) is not because it contains a variable C not appearing in a domain predicate in the rule. Rule (4) can be transformed to a domain-restricted one by adding a domain predicate for C, e.g., as follows:

$$\leftarrow \mathit{edge}(V, U), \mathit{colored}(V, C), \mathit{colored}(U, C), \mathit{color}(C)$$

It is straightforward to extend domain predicates with built-in functions and predicates, e.g., for arithmetic, and this extension is supported by our implementation. This allows rules such as

$$\mathit{area}(C, W * L) \leftarrow \mathit{width}(C, W), \mathit{length}(C, L) \tag{9}$$

$$0 \leq \{\mathit{circuit}(C) : \mathit{area}(C, A) = A\} \leq 90 \leftarrow \tag{10}$$

where *area* is a domain predicate defined by the domain predicates *width* and *length* (giving the width and length of a circuit) and the second rule specifies a choice of a subset of circuits with the sum of areas at most 90. Our implementation also allows expressing weights by rules involving domain predicates as in the example. In order to avoid complications arising from finite precision of real number arithmetic our current implementation supports only integer weights.

Domain-restrictedness enables efficient compilation of a program with variables to a set of ground rules which is typically considerably smaller than the Herbrand instantiation of the program but still has exactly the same stable models. This is because for the set of domain predicates D there is a unique set D' of ground instances of predicates in D that is common to all the stable models of the program. The set D' can be computed efficiently using database techniques because domain predicates are similar to view definitions in databases. The set D' has the property that program P has the same stable models as

$P_{D'}$ where $P_{D'}$ contains those ground instances of rules in P where each ground instance of a domain predicate is in D'. Furthermore, given D', $P_{D'}$ can be computed efficiently by processing one rule in P at a time. As an example consider program P with rules (9–10) and predicates *width* and *length* given as a set of facts $F = \{width(c_1, 5), length(c_1, 10), width(c_2, 3), length(c_2, 30)\}$. Our implementation detects automatically that *width*, *length*, and *area* can be taken as domain predicates and that the rules are domain-restricted. It computes the set $D' = F \cup \{area(c_1, 50), area(c_1, 90)\}$ and from that $P_{D'}$ where, e.g., for (10) only one ground instance is included:

$$0 \leq \{circuit(c_1) : area(c_1, 50) = 50, circuit(c_1) : area(c_2, 90) = 90\} \leq 90 \leftarrow$$

The whole compilation works as follows. Given a program P the domain predicates D are determined and the set D' is computed. Then the set $P_{D'}$ is constructed and the condition parts of the literals are removed. Finally the set of ground rules obtained in this way is transformed to a set of simpler rules accepted by **smodels-2**. We finish the section by explaining this last phase.

The **smodels-2** procedure supports many types of extended rules of which we employ two: choice and weight rules. A choice rule

$$\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m$$

states that a subset of $\{h_1, \ldots, h_k\}$ is in a stable model if the body is satisfied by the model. A weight rule

$$h \leftarrow \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \geq w$$

with positive weights w_{a_i}, w_{b_i} implies the inclusion of the head h into a stable model S whenever $\sum_{a_i \in S} w_{a_i} + \sum_{b_i \notin S} w_{b_i} \geq w$.

A weight rule $C_0 \leftarrow C_1, \ldots, C_n$ is encoded as rules handled by **smodels-2** as follows. For each constraint C_i

$$\mathrm{l}(C_i) \leq \{c_i^1 = \mathrm{w}(C_i)(c_i^1), \ldots, c_i^k = \mathrm{w}(C_i)(c_i^k)\} \leq \mathrm{u}(C_i)$$

we construct two weight rules encoding whether the lower bound is satisfied (C_i^l) and the upper bound is not (C_i^u).

$$C_i^l \leftarrow \{c_i^1 = \mathrm{w}(C)(c_i^1), \ldots, c_i^k = \mathrm{w}(C)(c_i^k)\} \geq \mathrm{l}(C_i)$$
$$C_i^u \leftarrow \{c_i^1 = \mathrm{w}(C)(c_i^1), \ldots, c_i^k = \mathrm{w}(C)(c_i^k)\} > \mathrm{u}(C_i)$$

where C_i^l, C_i^u are new atoms. Since only integer weights are allowed in the implementation, the latter rule can be expressed using '$\geq$' instead of '$>$', by increasing $\mathrm{u}(C_i)$ by one. Then we add a choice and two normal rules

$$\{c_0^1, \ldots, c_0^k\} \leftarrow C_1^l, not\ C_1^u, \ldots, C_n^l, not\ C_n^u$$
$$\leftarrow not\ C_0^l, C_1^l, not\ C_1^u, \ldots, C_n^l, not\ C_n^u$$
$$\leftarrow C_0^u, C_1^l, not\ C_1^u, \ldots, C_n^l, not\ C_n^u$$

where the first one selects a subset of $\{c_0^1, \ldots, c_0^k\}$ and the two other rules enforce that the lower and upper bounds of the head of the rule hold if the body of the rule is satisfied. Finally negative weights are eliminated as described in Section 2.

Example 7. To give an example, the weight constraint rule

$$1 \le \{a, b\} \le 1 \leftarrow 1 \le \{a, b, not\ c\} \le 2$$

is translated into the program

$$\begin{aligned} C_0^l &\leftarrow \{a = 1, b = 1\} \ge 1 \\ C_0^u &\leftarrow \{a = 1, b = 1\} > 1 \\ C_1^l &\leftarrow \{a = 1, b = 1, not\ c = 1\} \ge 1 \\ C_1^u &\leftarrow \{a = 1, b = 1, not\ c = 1\} > 2 \end{aligned} \qquad \begin{aligned} \{a, b\} &\leftarrow C_1^l, not\ C_1^u \\ &\leftarrow not\ C_0^l, C_1^l, not\ C_1^u \\ &\leftarrow C_0^u, C_1^l, not\ C_1^u \end{aligned}$$

In order to give an idea of the performance of the implementation we provide some test results for the pigeonhole problem. In Figure 1 running times (w-rules) are shown for deciding that $n+1$ pigeons cannot be put into n holes using the program on the left for $n = 8, 9, 10$ (where pigeons and holes are given as facts $p(i)/h(j)$). The results compare favorably to those (n-rules) for solving the same problems using normal logic programs as described in [11].

$$\begin{aligned} &1 \le \{in(P, H) : h(H)\} \le 1 \leftarrow p(P) \\ &\leftarrow 2 \le \{in(P, H) : p(P)\}, h(H) \end{aligned}$$

pigeons/holes	w-rules	n-rules
9/8	2.4 s	25.1 s
10/9	22 s	258 s
11/10	225 s	2600 s

Fig. 1. The pigeonhole problem and test results for it.

4 Conclusions

We have presented a novel rule language extending normal logic programs with conditional and weighted literals and weight constraints. The declarative semantics of the language generalizes the stable model semantics of normal programs. Despite the extensions, the complexity of finding a stable model remains in **NP** for the ground case. An implementation of a computationally attractive and useful subset of the language based on the **smodels-2** procedure is described. The language seems to be particularly suitable for product configuration problems and an interesting topic for further research is to apply the language in such problems along the lines presented in [16].

Acknowledgements. The work of the first and second authors has been funded by the Academy of Finland (Project 43963), that of the second and third authors by the Helsinki Graduate School in Computer Science and Engineering, and that of the third author by the Technology Development Centre Finland. We thank Tommi Syrjänen for implementing the front-end to the **smodels-2** procedure.

References

1. G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. In *Principles of Knowledge Representation and Reasoning Proceedings of the Sixth International Conference*, pages 86–97, 1998.
2. F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive datalog. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 2–17, 1997.
3. M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. In *Practical Aspects of Declarative Languages, LNCS 1551*, pages 16–30, 1999.
4. 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, 1988.
5. S. Greco. Dynamic programming in Datalog with aggregates. *IEEE Transactions on Knowledge and Data Engineering*, 11(2):265–283, 1999.
6. J. Lu, A. Nerode, and V. Subrahmanian. Hybrid knowledge bases. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):773–785, 1996.
7. W. Marek and M. Truszczyński. Autoepistemic logic. *J. ACM*, 38:588–619, 1991.
8. W. Marek and M. Truszczyński. Logic programming with costs. Manuscript, available at http://www.cs.engr.uky.edu/˜mirek/papers.html, 1999.
9. 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.
10. R. Ng and V. Subrahmanian. Stable semantics for probabilistic deductive databases. *Information and Computation*, 110:42–83, 1994.
11. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79. Available at http://www.tcs.hut.fi/pub/reports/A52abstract.html, 1998. An extended version to be published in Annals of Mathematics and Artificial Intelligence.
12. I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, 1996.
13. 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, 1997.
14. D. Sabin and R. Weigel. Product configuration frameworks – a survey. *IEEE Intelligent Systems & Their Applications*, pages 42–49, July/August 1998.
15. P. Simons. Extending the stable model semantics with more expressive rules. In *Proceedings of the 5th International Conference on Logic Programming and Non-Monotonic Reasoning*, 1999.
16. T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Practical Aspects of Declarative Languages, LNCS 1551*, pages 305–319, 1999.