Propositional satisfiability in answer-set programming

Deborah East and Mirosław Truszczyński

Department of Computer Science, University of Kentucky
Lexington KY 40506-0046, USA
{deast, mirek}@cs.engr.uky.edu

Abstract. We show that propositional logic and its extensions can support answer-set programming in the same way stable logic programming and disjunctive logic programming do. To this end, we introduce a logic based on the logic of propositional schemata and on a version of the Closed World Assumption. We call it the *extended logic of propositional schemata with CWA* (PS^+, in symbols). An important feature of the logic PS^+ is that it supports explicit modeling of constraints on cardinalities of sets. In the paper, we characterize the class of problems that can be solved by finite PS^+ theories. We implement a programming system based on the logic PS^+ and design and implement a solver for processing theories in PS^+. We present encouraging performance results for our approach — we show it to be competitive with *smodels*, a state-of-the-art answer-set programming system based on stable logic programming.

1 Introduction

Logic is most commonly used in declarative programming and knowledge representation as follows. To solve a problem we represent its constraints and the relevant background knowledge as a theory in the language of some logic. We formulate the goal (the statement of the problem) as a formula of the logic. We then use proof techniques to decide whether this formula follows from the theory. A proof of the formula, variable substitutions or both determine a solution.

Recently, an alternative way in which logic can be used in computational knowledge representation has emerged from studies of nonstandard variants of logic programming such as logic programming with negation and disjunctive logic programming [MT99,Nie99]. This alternative approach is rooted in semantic notions and is based on methods to compute models. To represent a problem, we design a finite theory so that its *models* (and not proofs or variable substitutions) determine problem solutions (answers). To solve the problem, we compute models of the corresponding theory[1]. This model-based approach is now often referred to as *answer-set programming* (or ASP).

---

[1] We commonly restrict the language by disallowing function symbols to guarantee finiteness of models of finite theories. In the present paper, we also adopt this assumption.

Logic programming with stable model semantics [GL88] (*stable logic programming* or *SLP*, in short) is an example of an ASP formalism [MT99]. In SLP, we represent problem constraints by a fixed program (independent of problem instances). We represent a specific instance of the problem (input data) by a collection of ground atoms. To solve the problem, we find stable models of the program formed jointly by the two components. To this end, we first *ground* it (compute its equivalent propositional representation) and, then, compute stable models of this grounded propositional program. Thanks to the emergence of fast systems to compute stable models of propositional logic programs, such as *smodels* [NS00], SLP is quickly becoming a viable declarative programming environment for computational knowledge representation. Disjunctive logic programming with the semantics of answer sets [GL91] is another logic programming formalism that fits well into the answer-set programming paradigm. An effective solver for computing answer sets of disjunctive programs, *dlv*, is available [ELM$^+$98] and its performance is comparable with that of *smodels*.

Our goal in this paper is to propose answer-set programming formalisms based on propositional logic and its extensions. Our approach is motivated by recent improvements in the performance of satisfiability checkers. Researchers developed several new and fast implementations of the basic Davis-Putnam method such as *satz* [LA97] and *relsat* [BS97]. A renewed interest in local-search techniques resulted in highly effective (albeit incomplete) satisfiability checkers such as *WALKSAT* [SKC94], capable of handling large CNF theories, consisting of millions of clauses. Improvements in the performance resulted in an expanding range of applications of satisfiability checkers, with planning being one of the most spectacular examples [KMS96,KS99].

The way in which propositional satisfiability solvers are used in planning [KMS96] clearly fits the ASP paradigm. Planning problems are encoded as propositional theories so that models correspond to plans. In our paper, we extend ideas proposed in [KMS96] in the domain of planning and show that propositional satisfiability can be used as the foundation of a general purpose ASP system. To this end, we propose a logic to serve as a modeling language. This logic is a modification of the logic of propositional schemata [KMS96]; we explicitly separate theories into data and program, and use a version of Closed World Assumption (CWA) to define the semantics. This logic is nonmonotonic. We call it the *logic of propositional schemata with CWA* (or, PS^{cwa}).

The logic PS^{cwa} offers only basic logical connectives to help model problem constraints. We extend logic PS^{cwa} to support direct representation of constraints involving cardinalities. Examples of such constraints are: "at least k elements from the list must be in the model" or "exactly k elements from the list must be in the model". They appear commonly in statements of constraint satisfaction problems. We refer to this new logic as *extended logic of propositional schemata with Closed World Assumption* and denote it by PS^+.

In the paper we characterize the class of problems that can be solved by *finite* PS^+ theories. In other words, we determine the expressive power of the logic

PS^+. Specifically, we show that it is equal to the expressive power of function-free logic programming with the stable-model semantics.

For processing, theories in PS^+ could be compiled into propositional theories and "off-the-shelf" satisfiability checkers could be used for processing. However, propositional representations of constraints involving cardinalities are usually very large and the sizes of the compiled theories limit the effectiveness of satisfiability checkers, even the most advanced ones, as processing engines. Thus, we argue against the compilation of the cardinality constraints. Instead, we propose an alternative approach. We design a "target" propositional logic for the logic PS^+ (propositional logic PS^+). In this logic, cardinality constraints have explicit representations and, therefore, do not need to be compiled any further. We develop a satisfiability checker for the propositional logic PS^+ and use it as the processing back-end for the logic PS^+. Our solver is designed along the same lines as most satisfiability solvers implementing the Davis-Putnam algorithm but it takes a direct advantage of the cardinality constraints explicitly present in the language.

Experimental results on the performance of the overall system are highly encouraging. We obtain concise encodings of constraint problems and the performance of our solver is competitive with the performance of *smodels* and of state-of-the-art complete satisfiability checkers. Our work demonstrates that building propositional solvers capable of processing of high-level constraints is a promising research direction for the area of propositional satisfiability.

Our paper is organized as follows. In the next section we introduce the logic PS^{cwa} — a fragment of the logic PS^+ without cardinality constraints. We determine the expressive power of the logic PS^{cwa} in Section 3. We discuss the full logic PS^+ in Section 4. In the subsequent section we discuss implementation details and experimental results. The last section of the paper contains conclusions and comments on the future work.

2 Basic logic PS^{cwa}

Our approach is based on the logic of *propositional schemata*. The syntax of this logic is that of first-order logic without function symbols. The semantics is that of *Herbrand interpretations* and *models*, which we identify with subsets of the *Herbrand base*. In the paper we consider only those theories in which at least one constant symbol appears. Among all formulas in the language, of main interest to us are *clauses*, that is, expressions of the form

$$a_1 \wedge \ldots \wedge a_m \Rightarrow B_1 \vee \ldots \vee B_n, \tag{1}$$

where each a_i is an atom and each B_j is an atom or an expression of the form $\exists Y\, b(s)$, where $b(s)$ is an atom and Y is a tuple of (not necessarily all) variables appearing in $b(s)$. Each of m and n (or both) may equal 0. If $m = 0$, we replace the conjunct in the antecedent of the clause with a special symbol $\mathbf{T}$ (*truth*). If $n = 0$, we replace the empty disjunct in the consequent of the clause with a special symbol $\mathbf{F}$ (*contradiction*). We assume that each clause is universally quantified and drop the universal quantifiers from the notation. We further simplify the notation by replacing each expression $\exists Y\, b(s)$ in the antecedent by $b(s')$, where in s' we write a special symbol '_' for each variable from Y in s.

Let T be a *finite* theory consisting of clauses. For a formula $B = \exists Y\, b(s)$ appearing in the consequent of a clause in T, we define B^e to be the disjunction $B^e = \quad b(s^1) \vee \ldots \vee b(s^k)$, where s^i, $1 \leq i \leq k$, range over all term tuples that can be obtained from s by replacing variables in Y with constants appearing in T. Since T is finite, the disjunction is well defined (it has only finitely many disjuncts).

For a clause $C \in T$ of the form (1), we define a clause C^e by

$$C^e = \quad a_1 \wedge \ldots \wedge a_m \Rightarrow B_1^e \vee \ldots \vee B_n^e, \tag{2}$$

A *ground instance* of C is any formula obtained from C^e by replacing every variable in C^e by a constant appearing in T (different occurrences of the same variable must be replaced by the same constant). We define the *grounding* of T, $gr(T)$ as the collection of all ground instances of clauses in T, except for tautologies; they are not included in $gr(T)$. We have the following well-known result.

Proposition 1. *Let T be a finite clausal theory. Then a set of ground atoms M is a Herbrand model of T if and only if M is a (propositional) model of $gr(T)$.*

The language may contain several *predefined* predicates and function symbols such as the equality operator and arithmetic comparators and operations. We assign to these symbols their standard interpretation. However, we emphasize that the domains are restricted only to those constants that appear in a theory.

We evaluate all expressions involving predefined function symbols and all atoms involving predefined relation symbols in the grounding process. If any argument of a predefined relation is not of the appropriate type, we interpret the corresponding atom as false. If a function yields as a result a constant that does not appear in the theory or if one of its arguments is not of the required type, we also interpret the corresponding atom as false. We then eliminate tautologies and simplify the remaining clauses by removing true "predefined" atoms from the antecedents and false "predefined" atoms from the consequents.

Let us consider an example. Let T be a theory consisting of the following two clauses:

$$C_1 = \quad q(b,c) \Rightarrow p(a)$$
$$C_2 = \quad p(X) \Rightarrow (\exists Y q(X,Y)) \vee (X = a).$$

There are three constants, a, b and c, and two predicate symbols, p and q, in the language. Symbols X and Y denote variables. The clause C_2 can also be written (using the simplified notation) as

$$C_2 = \quad p(X) \Rightarrow q(X, \_) \vee (X = a).$$

To compute $gr(T)$ we need to compute all ground instances of C_2 (C_1 is itself its only ground instance). First, we compute the formula C_2^e:

$$C_2^e = \quad p(X) \Rightarrow q(X,a) \vee q(X,b) \vee q(X,c) \vee (X = a).$$

To obtain all ground instances of C_2 (or C_2^e), we replace X with a, b and c. The first substitution results in a tautology (due to occurrence of '$a = a$' in the consequent of the clause). Two other substitutions yield the following two ground instances of C (we drop atoms '$b = a$' and '$c = a$' from the consequents; they are false by the standard interpretation of equality):

$$p(b) \Rightarrow q(b,a) \vee q(b,b) \vee q(b,c)$$
$$p(c) \Rightarrow q(c,a) \vee q(c,b) \vee q(c,c).$$

These two clauses together with C_1 form $gr(T)$. The sets of ground atoms $\{p(a), q(b,c)\}$ and $\{p(b), p(c), q(b,a), q(c,c)\}$ are two examples of models of T (or $gr(T)$).

In order for the logic of propositional schemata to be useful as a programming tool, we modify it to separate input data from the program encoding the problem to be solved. We distinguish in the set of predicates Pr of the language a subset, Pr'. We call its elements *data* predicates. We assume that predefined predicates are *not* data predicates. All predicates other than data predicates and predefined predicates are called *program* predicates. A *theory* of our logic is a pair (D, P), where D is a *finite* collection of ground atoms whose predicate symbols are data predicates (*data*), and P is a *finite* collection of clauses (a *program*).

To define the semantics for the logic, we use grounding and a form of CWA. We say that a set of ground atoms (built of data and program predicates) is a *model* of a theory (D, P) if

M1: M is a model of $gr(D \cup P)$ (or, equivalently, M is an Herbrand model of $D \cup P$), and

M2: for every ground atom $p(t)$ such that $p \in Pr'$ (p is a data predicate), $p(t) \in M$ if and only if $p(t) \in D$.

We call the logic described above the *logic of propositional schemata with CWA* and denote it by PS^{cwa}. Due to (M2), not every model of $gr(D, P)$ is a model of (D, P). Consequently, one can show that our logic is nonmonotonic. This difference between the logic of propositional schemata and the logic PS^{cwa}, while seemingly small, has significant consequences for the expressive power of the logic and its applicability as a programming tool.

Before addressing these two issues, let us consider an example. Let A and B be two disjoint and finite sets. We define $D = \{p_1(a) : a \in A\} \cup \{p_2(b) : b \in B\}$. We define P to consist of two clauses:

Ex1: $q_1(X) \Rightarrow p_1(X)$ Ex2: $q_2(X) \Rightarrow p_2(X)$.

The constants are elements of $A \cup B$; X is a variable. The predicates are p_1, p_2, q_1 and q_2. The first two are data predicates.

By (M2), each model of a PS^{cwa} theory (D, P) contains D. However, it does not contain any ground atom $p_1(b)$, where $b \in B$, nor any ground atom $p_2(a)$, where $a \in A$. Each ground instance of the clause (Ex1) is of the form $q_1(c) \Rightarrow p_1(c)$, where c is a constant ($c \in A \cup B$). Since $p_1(c) \in M$ if and only if $c \in A$, it follows that if $q_1(c) \in M$, then $c \in A$. Similarly, we obtain that if $q_1(c) \in M$, then $c \in A$. Thus, M is a model of (D, P) if and only if $M = D \cup \{q_1(a) : a \in A'\} \cup \{q_2(b) : b \in B'\}$, for some $A' \subseteq A$ and $B' \subseteq B$.

Let us choose an element from A, say a_0, and an element from B, say b_0. Let us then add to P the clause

Ex3: $p_1(a_0) \Rightarrow p_1(b_0)$

We denote the new program by P'. The PS^{cwa} theory (D, P) has no models even though $gr(D, P)$ is propositionally consistent. The reason is that all propositional models satisfying $gr(D, P)$ contain $p_1(b_0)$. Thus, none of these models satisfies condition (M2). This example illustrates that our semantics is different from circumscription as circumscription preserves consistency. Circumscription applied to p_1 would result in models in which the extension of p_1 in D would be minimally extended by one more constant b_0. Our (strong) minimization principle does not allow for any additions to the extension of data predicates. Intuitively, it is exactly as it should be. **Data predicates are meant to represent input data. The program should not be able to extend it.**

Logic PS^{cwa} is a tool to model problems. To illustrate this use of the logic, we show how to encode the *vertex-cover* problem for graphs. Let $G = (V, E)$ be a graph. A set $W \subseteq V$ is a *vertex cover* of G if for every edge $\{x, y\} \in E$, x or y (or both) are in W. The vertex-cover problem is defined as follows: given a graph $G = (V, E)$ and an integer k, decide whether G has a vertex cover with no more than k vertices.

For the vertex-cover problem the input data is described by the following set of ground atoms:

$D_{vc} = \{vtx(v) \colon v \in V\} \cup \{edge(v, w) \colon \{v, w\} \in E\} \cup \{size(k)\} \cup \{pos(i) \colon 1, \ldots, n\}$.

This set specifies the set of vertices and the set of edges of an input graph. It provides the limit on the size of a vertex cover sought. Lastly, it uses a predicate *pos* to specify a range of *integers* that will be used to label vertices. The problem itself is described by the program P_{vc}:

VC1: $vpos(I, X) \Rightarrow vtx(X)$
VC2: $vpos(I, X) \Rightarrow pos(I)$
VC3: $vtx(X) \Rightarrow vpos(\_, X)$
VC4: $vpos(I, X) \wedge vpos(J, X) \Rightarrow I = J$
VC5: $vpos(I, X) \wedge vpos(I, Y) \Rightarrow X = Y$
VC6: $edge(X, Y) \wedge vpos(I, X) \wedge vpos(J, Y) \wedge size(K) \Rightarrow (I \leq K) \vee (J \leq K)$

(VC1) and (VC2) ensure that $vpos(i, x)$ is false if i is not an integer from the set $\{1, \ldots, n\}$ or if x is not a vertex. (VC3)-(VC5) together enforce that the atoms $vpos(i, x)$ that are true in a model of the PS^{cwa} theory (D_{vc}, P_{vc}) define a permutation of the vertices in V. Finally, (VC6) ensures that each edge has at least one vertex assigned by *vpos* to positions $1, \ldots, k$ (in other words, that vertices labeled $1, \ldots, k$ form a vertex cover). The correctness of this encoding is formally established in the following result.

Proposition 2. *Let $G = (V, E)$ be an undirected graph and let k be a positive integer. A set of vertices $\{w_1, \ldots, w_k\} \subseteq V$ is a vertex cover of G if and only if $M = D_{vc} \cup \{vpos(i, w_i) \colon i = 1, \ldots, k\}$ is a model of the theory (D_{vc}, P_{vc}).*

For another example, we will consider the n-queens problem, that is, the problem of placing n queens on a $n \times n$ chess board so that no queen attacks another.

In this case, the representation of input data describes the set of row and column indices:
$D_{nq} = \{pos(i): 1, \ldots, n\}$.
The problem itself is described by the program P_{nq}. The predicate q describes a distribution of queens on the board: $q(x, y)$ is true precisely when there is a queen in the position (x, y).

nQ1: $q(R, C) \Rightarrow pos(R)$
nQ2: $q(R, C) \Rightarrow pos(C)$
nQ3: $q(R, C1) \wedge q(R, C2) \Rightarrow C1 = C2$
nQ4: $q(R1, C) \wedge q(R2, C) \Rightarrow R1 = R2$
nQ5: $q(R, C), q(R + I, C + I) \Rightarrow \mathbf{F}$
nQ6: $q(R, C), q(R + I, C - I) \Rightarrow \mathbf{F}$

The first two clauses ensure that if $q(r, c)$ is true in a model of (D_{nq}, P_{nq}) then r and c are integers from the set $\{1, \ldots, n\}$. The following two clauses enforce the constraint that no two queens are placed in the same row or the same column. Finally, the last two clauses guarantee that no two queens are placed on the same diagonal. As in the case of the vertex cover problem, also in this case we can formally show the correctness of this encoding.

These examples demonstrate that PS^{cwa} programs can serve as representations of computational problems. Two key questions arise: (1) what is the expressive power of the logic PS^{cwa}, and (2) how to use the logic PS^{cwa} as a practical computational tool. We address both questions in the remainder of the paper.

3 Expressive power of PS^{cwa}

A *search* problem, Π, is given by a set of finite *instances*, D_Π, such that for each instance $I \in D_\Pi$, there is a finite set $S_\Pi(I)$ of all *solutions* to Π for the instance I [GJ79]. The graph-coloring, vertex-cover and n-queens problems considered in the previous section are search problems. More generally, all constraint satisfaction problems including basic AI problems such as planning, scheduling and product configuration can be cast as search problems.

We say that a PS^{cwa} program P *solves* a search problem Π if there exist:

1. A mapping d that can be computed in polynomial time and that encodes instances to Π as sets of ground atoms built of data predicates
2. A partial mapping sol, computable in polynomial time, that assigns to (some) sets of ground atoms solutions to Π (elements of $\bigcup_{I \in D_\Pi} S_\Pi(I)$)

such that for every instance $I \in D_\Pi$, $s \in S_\Pi(I)$ if and only if there exists a model M of the PS^{cwa} theory $(d(I), P)$ such that M is in the domain of the mapping sol and $sol(M) = s$.

A search problem Π is in the class *NP-search* if there is a nondeterministic Turing Machine TM such that (1) TM runs in polynomial time; (2) for every instance $I \in D_\Pi$, the set of strings left on the tape when accepting computations for I terminate is precisely the set of solutions $S_\Pi(I)$.

We now have the following theorem that determines the expressive power of the logic PS^{cwa}. Its proof is provided in the appendix.

Theorem 1. *A search problem Π can be solved by a PS^{cwa} program if and only if $\Pi \in$ NP-search.*

Decision problems can be viewed as special search problems. For the class of decision problems, Theorem 1 implies the following corollary (a counterpart to the result on the expressive power of DATALOG$^\neg$ [Sch95]).

Corollary 1. *A decision problem Π can be solved by a PS^{cwa} program if and only if Π is in NP.*

4 Extending PS^{cwa} — the logic PS^+

We will now discuss ways to enhance effectiveness of logic PS^{cwa} as a modeling formalism and propose ways to improve computational performance. When considering the PS^{cwa} theories developed for the n-queens and vertex-cover problems one observes that these theories could be simplified if the language of the logic PS^{cwa} contained direct means to model constraints such as: "exactly one element is selected" or "at most k elements are selected".

With this motivation, we extend the language of the logic PS^{cwa}. We define a *c-atom* (cardinality atom) as an expression $m\{p(X,\_,Y)\}n$, where m and n are non-negative integers, X and Y are tuples of variables and p is a program predicate[2].

The interpretation of a c-atom is that for every ground tuples x and y that can be substituted for X and Y, at least m and at most n atoms from the set

$$\{p(x,c,y) \colon c \text{ is a constant appearing in the theory}\}$$

are true. One of m and n may be missing from the expression. If m is missing, there is no lower-bound constraint on the number of atoms that are true. If m is missing, there is no upper-bound constraint on the number of atoms that are true. It is also possible to have more "underscore" symbols in c-atoms. In such case, when forming the set of atoms on which cardinality constraints are imposed, all possible ways to replace the "underscore" symbols by constants are used.

An *extended clause* is a clause built of c-atoms. The notions of a *program* and *theory* are defined as in the case of the logic PS^{cwa}.

A theory in the extended syntax can be grounded, that is, represented as a set of propositional clauses, in a similar way as before. In particular, data and

[2] In our implementation, we support a somewhat more general form of c-atoms.

predefined predicates are treated in the same way and are subject to the same version of CWA that was used for the logic PS^{cwa}. While grounding, c-atoms are interpreted as explained earlier. Grounding allows us to lift the semantics of propositional logic to the theories in the extended syntax. We call the resulting logic the *extended logic* PS^{cwa} and denote it by PS^{+}.

In the logic PS^{+} we can encode the vertex cover problem in a more straightforward and more concise way. Namely, the problem can be represented without the need for integers to label the vertices of an input graph! This new representation (D'_{vc}, P'_{vc}) is given by:
$D'_{vc} = \{vtx(v) \colon v \in V\} \cup \{edge(v,w) \colon \{v,w\} \in E\}$,
and $P'_{vc} =$

VC′1: $invc(X) \Rightarrow vtx(X)$
VC′2: $\{invc(\_)\}k$
VC′3: $edge(X,Y) \Rightarrow invc(X) \vee invc(Y)$.

Atoms $invc(x)$ that are true in a model of the PS^{cwa} theory (D'_{vc}, P'_{vc}) define a set of vertices that is a candidate for a vertex cover. (VC′2) guarantees that no more than k vertices are included. (VC′3) enforces the vertex-cover constraint.

We close this section with an observation on the expressive power of the logic PS^{+}. Since it is a generalization of the logic PS^{cwa}, it can capture all problems that are in the class NP-search. On the other hand, the problem of computing models of a PS^{+} theory with a fixed program part is an NP-search problem, it follows that the expressive power of the logics PS^{+} does not extend beyond the class NP-search. In other words, the logic PS^{+} also captures the class NP-search.

5 Computing with PS^{+} theories

To process PS^{+} theories, one approach is to ground them into collections of propositional clauses. However, CNF representations of c-atoms may be quite large; the constraint "at most n atoms in the set $\{p_1, \ldots, p_k\}$ are true", is captured by $\theta(k^{n+1})$ clauses $p_{i_1}, \ldots, p_{i_{n+1}} \Rightarrow \mathbf{F}$, one for each $(n+1)$-element subset $\{p_{i_1}, \ldots, p_{i_{n+1}}\}$ of $\{p_1, \ldots, p_k\}$.

Thus, we propose another approach. The idea is to develop an extension of propositional logic representing c-atoms directly. Let At be a set of propositional variables. By a *propositional c-atom* we mean any expression of the form $m\{p_1, \ldots, p_k\}n$, where m and n are non-negative integers and $p_1, \ldots, p_k$ are atoms in At (one of m and n may be missing). By an *extended propositional clause* we mean an expression of the form

$$C = \quad A_1 \wedge \ldots \wedge A_s \Rightarrow B_1 \vee \ldots \vee B_t,$$

where all A_i and B_i are propositional c-atoms.

Let $M \subseteq At$ be a set of atoms. We say that M *satisfies* a generalized atom $m\{p_1, \ldots, p_k\}n$ if

$$m \leq |M \cap \{p_1, \ldots, p_k\}| \leq n.$$

Further, M *satisfies* a generalized clause C if M satisfies at least one atom B_j or does not satisfy at least one atom A_i. We call the resulting logic the *propositional logic* PS^+. Clearly, M satisfies an atom $1\{p\}1$ if and only if $p \in M$. Thus, the propositional logic PS^+ extends the (clausal) propositional logic.

Theories of the logic PS^+ can be grounded in the extended propositional logic by generalizing the approach described in Section 2. We represent c-atoms as propositional c-atoms and avoid a blow-up in the size of the representation. The problem is that SAT checkers cannot now be used to resolve the satisfiability of the extended propositional logic as they are not designed to work with the extended syntax.

It is clear, however, that the techniques developed in the area of SAT checkers can be extended to the propositional logic PS^+. We have developed a Davis-Putnam like procedure, *aspps*, that finds models of propositional PS^+. We also developed a program *psgrnd* that accepts theories in the syntax of the logic PS^+ and grounds them into propositional PS^+ theories. Thus, the two programs together can be used as a processing mechanism for an answer-set programming system based on the logic PS^+. The programs *psgrnd* and *aspps* are available at `http://www.cs.uky.edu/ai/aspps/`.

In our experiments we considered the vertex-cover problem and several combinatorial problems including n-queens problem, pigeonhole problem and the problem to compute Schur numbers. All our experiments were performed on a Pentium III 500MHz machine running linux.

We were mostly interested in comparing the performance of our system *psgrnd/aspps* with that of *smodels*. The reason is that both programs accept similar syntax and allow for very similar modeling of constraints. We also experimented with a satisfiability checker *satz*.

In the case of vertex cover, for each $n = 50, 60, 70$ and 80, we randomly generated 100 graphs with n vertices and $2n$ edges. For each graph G, we computed the minimum size k_G for which the vertex cover can be found. We then tested *aspps*, *smodels* and *satz* on all the instances (G, k_G). The results represent the average execution times Encodings we used for testing *aspps* and *smodels* where based on the clauses (VC$'$1) - (VC$'$3). For *satz* we used encodings based on the clauses (VC1) - (VC6) (cardinality constraints cannot be handled by *satz*).

A propositional CNF theory obtained by grounding the program (VC1) - (VC6), has $\Theta(n^2)$ atoms, $\Theta(mn^2)$ clauses and its total size is also $\Theta(mn^2)$. For input instances we used in our experiments, these theories were of such large sizes (over one million rules in the case of graphs with 80 vertices) that *satz* did not terminate in the time we allocated (5 minutes). Thus, no times for *satz* are reported. On the other hand, since the propositional PS^+ theory obtained by grounding the PS^+ program (VC$'$1) - (VC$'$3) has only $\Theta(m + n)$ clauses (a few hundred clauses for graphs with 80 vertices) and its total size has the same asymptotic estimate. This is dramatically less than in the case of theories *satz* had to process. Both *aspps* and *smodels* performed very well, with *aspps* being about three times faster than *smodels*. The timing results are summarized in Table 1.

n	50	60	70	80
aspps	0.04	0.22	1.26	6.45
smodels	0.12	0.76	4.14	22.35

Table 1. Timing results (in seconds) for the vertex-cover problem.

For the n-queens problem, our solver performed exceptionally well. It scaled up much better than *smodels* both in the case when we were looking for one solution and when we wanted to compute all solutions. In particular, our program found a solution to the 36 queens problem in 0.97 sec. It also outperformed *satz*.

# of queens	18	19	20	21	22	23
aspps	0.02	0.02	0.07	0.07	0.11	0.12
smodels	2.35	1.28	13.25	19.31	167.1	380.35
satz	1.16	0.61	4.35	0.95	28.64	1.42

Table 2. Timing results (in seconds) for the n-queen problem.

The pigeonhole problem consists of showing that it is not possible to place p pigeons in h holes if $p > h$. For this problem *aspps* showed the best performance — about three times faster than the other two solvers (all programs showed a similar rate of growth in the execution time).

(p, h)	(9,8)	(10,9)	(11,10)	(12,11)
aspps	0.59	5.63	60.08	702.02
smodels	2.7	21.56	219.99	2469.97
satz	1.87	17.28	178.20	2044.42

Table 3. Timing results (in seconds) for the pigeonhole problem

The Schur problem consists of placing n numbers $1, 2, \ldots, n$ in k bins so that the set of numbers assigned to a bin is not closed under sums. That is, for all numbers x, y, z, $1 \leq x, y, z \leq n$, if x and y are in a bin b, then z is not in b (x and y need not be distinct). The Schur number $S(k)$ is the maximum number n for which such a placement is still possible.

We considered the problem of the existence of the placement for $k = 4$ and values of n ranging from 40 to 45. For $n \leq 44$ all programs found a "Schur" placement. However, no "Schur" placement exists for $n = 45$ (and higher values of n). All programs were able to establish the non-existence of solutions for $n = 45$ (but the times grew significantly). Our results summarizing the performance of our system and *smodels* on the theories encoding the constraints of the problem are shown in Table 4. *aspps* and *satz* seem to performed better than *smodels*, with *satz* being slightly faster for values of n closer to the Schur number.

n	40	41	42	43	44	45
aspps	0.03	0.03	0.03	0.03	1.83	54.5
smodels	0.3	0.38	0.32	0.36	35.8	>1500
satz	0.21	0.23	0.24	0.25	0.96	20.4

Table 4. Timing results (in seconds) for the Schur-number problem.

In the case of the last three problems, it was possible to eliminate cardinality constraints without significant increase in the size of grounded theories. As a result, *satz* performed well.

6 Conclusions

Our work demonstrates that propositional logic and its extensions can support answer-set programming systems in a way in which stable logic programming and disjunctive logic programming do[3]. In the paper we described logic PS^+ that can be used to this end. We presented an effective implementation of a grounder, *psgrnd*, and a solver, *aspps*, for processing theories in the logic PS^+. Our experimental results are encouraging. Our system is competitive with *smodels*, and in many cases outperforms it. It is also competitive with satisfiability solvers such as *satz*.

The results of the paper show that programming front-ends for constraint satisfaction problems that support explicit coding of complex constraints facilitate modeling and result in concise representations. They also show that solvers such as *aspps* that take advantage of those concise encodings and process high-level constraints directly, without compiling them to simpler representations, exhibit very good computational performance. These two aspects are important. Satisfiability checkers often cannot effectively solve problems simply due to the fact that encodings they have to work with are large. For instance, for the vertex-cover problem for graphs with 80 vertices and 160 edges, *aspps* has to deal with theories that consist of a few hundred of rules only. In the same time pure propositional encodings of the same problem contain over one million clauses — a factor that undoubtedly is behind much poorer performance of *satz* on this problem.

Our work raises new questions. Further extensions of logic PS^+ are possible. For instance, constraints that impose other conditions on set cardinalities than those considered here (such as, the *parity* constraint) might be included. We will pursue this direction. Similarly, there is much room for improvement in the area of solvers for the propositional logic PS^+. In particular, we will study local search algorithms as possible satisfiability solvers for propositional PS^+ theories.

Finally, we note that the experimental results presented here are meant to show that *aspps* is competitive with other solvers and, we think, they demonstrate this. However, these results are still too fragmentary to provide basis for any conclusive comparison between the three solvers tested. Such a comparison is further complicated by the fact that the same problem may have several different encodings with different computational properties. Developing the methodology for comparing solvers designed to work with different formal systems is a chal-

[3] We point out, though, that stable logic programming and disjunctive logic programming directly support negation-as-failure and, consequently, yield more direct solutions to some knowledge representation problems such as, for example, the frame problem.

lenging problem for builders of constraint solvers and declarative programming systems.

Acknowledgments

This work was partially supported by the NSF grants CDA-9502645, IRI-9619233 and EPS-9874764.

References

[Apt90] K. Apt. Logic programming. In J. van Leeuven, editor, *Handbook of theoretical computer science*, pages 493–574. Elsevier, Amsterdam, 1990.

[BS97] R.J. Bayardo, Jr and R.C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*. MIT Press, 1997.

[Cla78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and data bases*, pages 293–322. Plenum Press, New York-London, 1978.

[ELM+98] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A KR system `dlv`: Progress report, comparisons and benchmarks. In *Proceeding of the Sixth International Conference on Knowledge Representation and Reasoning (KR '98)*, pages 406–417. Morgan Kaufmann, 1998.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness.* W.H. Freeman and Co., San Francisco, Calif., 1979.

[GL88] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[GL91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[KMS96] H.A. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proceedings of KR-96*, pages 374–384. Morgan Kaufmann, 1996.

[KS99] H.A. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *Proceedings of IJCAI-99*, San Mateo, CA, 1999. Morgan Kaufmann.

[LA97] C.M. Li and M. Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, 1997.

[Llo84] J. W. Lloyd. *Foundations of logic programming.* Symbolic Computation. Artificial Intelligence. Springer-Verlag, Berlin-New York, 1984.

[MR01] W. Marek and J.B. Remmel. On the foundations of answer-set programming. In *Answer-Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning.* AAAI Press, 2001. Papers from the 2001 AAAI Spring Symposium, Technical Report SS-01-01.

[MT99] V.W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt, W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[Nie99] I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[NS00] I. Niemelä and P. Simons. Extending the smodels system with cardinality and weight constraints. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer Academic Publishers, 2000.

[Sch95] J. Schlipf. The expressive powers of the logic programming semantics. *Journal of the Computer Systems and Science*, 51(1):64–86, 1995.

[SKC94] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle, USA, 1994.

Appendix

We will present here a sketch of a proof of our main result concerning the expressive power of the logic PS^{cwa}. The proof relies on some basic notions from logic programming (we refer the reader to [Apt90,Llo84] for details).

We restrict our discussion to function-free languages (the case relevant to our logic PS^{cwa}). Given a predicate language $\mathcal{L}$ (as defined in Section 2), a *logic program clause* over this language is an expression r of the form

$$r = \quad p(t) \leftarrow q_1(t_1), \ldots, q_m(t_m), \mathbf{not}(q_{m+1}(t_{m+1})), \ldots, \mathbf{not}(q_{m+n}(t_{m+n}))$$

where $p, q_1, \ldots, q_{m+n} \in Pr$, (we assume that p is not a predefined predicate), and $t, t_1, \ldots, t_{m+n}$ are term tuples with the arity matching the arity of the corresponding predicate symbol. We call the atom $p(t)$ the *head* of the rule r and denote it by $h(r)$. For a rule r we also define

$$B(r) = q_1(t_1) \wedge \ldots q_m(t_m) \wedge \neg q_{m+1}(t_{m+1}) \wedge \ldots \wedge \neg q_n(t_n)$$

We will be interested in supported models of logic programs. Without loss of generality, we will restrict our attention to programs in the *normal form*. That is, we assume that (1) the head of each rule is of the form $p(t)$, where t is a tuple of variables, and (2) if p appears in the head of two rules, the heads of these two rules are exactly the same (the same tuple of variables appear in both of them) [Cla78,Apt90].

Let P be a program in the normal form. For each predicate symbol $p \in Pr(P)$, we define a formula $cc(p)$ by:

$$cc(p) = \quad p(X) \Leftrightarrow \bigvee\{\exists Y_r B(r) : r \in P', h(r) = p(X)\},$$

where X is a tuple of variables and Y_r is the tuple of variables occurring in the body of r but not in the head of r (we exploit the normal form of P here). We define the *completion* of P, $CC(P)$, by setting $CC(P) = \{cc(p) : p \in Pr\}$.

The Clark's completion is important as it allows us to characterize *supported* models of a logic program [Apt90]. Namely, we have the following result.

Theorem 2. *Let P be a logic program. A set of ground atoms M is a supported model of P if and only if it is a Herbrand model of $CC(P)$.*

We now have the following theorem.

Theorem 3. *Let P be a logic program in the normal form. Let Pr be the set of predicates appearing in P and let Pr' be the set of predicates of P that do not appear in the heads of rules in P. There is a PS^{cwa} theory $T(P)$ such that for every set of ground atoms D over predicates from Pr', a set of ground atoms M is a supported model of $D \cup P$ if and only if $M = M' \cap HB(P)$ for some model M' of the PS^{cwa} theory $(D, T(P))$.*

Proof: (Sketch) To define $T(P)$, we consider the completion $CC(P)$ of P. The idea is to take for $T(P)$ an equivalent clausal representation of $CC(P)$.

We build such representation as follows. Let p be a predicate symbol in $Pr \setminus Pr'$. The completion $CC(P)$ contains the formula

$$cc(p) = \quad p(X) \Leftrightarrow \bigvee\{\exists Y_r B(r) \colon r \in P, h(r) = p(X)\},$$

where X is a tuple of variables and Y_r is the tuple of variables occurring in the body of r but not in the head of r. For each rule $r \in P$ such that p occurs in $h(r)$, we introduce a new predicate symbol d_r, of the same arity $|X| + |Y_r|$. We define a theory $T'(P)$ to consist of the following formulas (we recall that $B(r)$ stands for the conjunction of the literals from the body of r):

$$\psi(r) = \quad d_r(X, Y_r) \Leftrightarrow B(r),$$

where $p \in Pr \setminus Pr'$, $r \in P$ and p occurs in the head of r, and

$$cc'(p) = \quad p(X) \Leftrightarrow \bigvee\{\exists Y_r d_r(X, Y_r) \colon r \in P, h(r) = p(X)\},$$

where $p \in Pr \setminus Pr'$.

It is clear that the theory $T'(P)$ is equivalent to $CC(P)$ (modulo new ground atoms). That is, $M \subseteq HB(P)$ is a model of $CC(P)$ if and only if $M = M' \cap HB(P)$, for some model M' of $T'(P)$.

One can show that $T'(P)$ can be rewritten (in polynomial time) into an equivalent clausal form, $T(P)$. Consequently, $T(P)$ is equivalent to $CC(P)$ (modulo ground atoms $d_r(t)$). It is now a routine task to verify that the theory $T(P)$ satisfies all the requirements of the statement of the theorem. $\square$

Using the terminology introduced here we will now prove Theorem 1 from Section 3.

Theorem 4. *A search problem Π can be solved by a finite PS^{cwa} program if and only if $\Pi \in$ NP-search.*

Proof: (Sketch) In [MR01] it is proved that every NP-search problem can be solved uniformly by a finite logic program under the supported-model semantics. Since the theory $T(P)$ can be constructed in polynomial time, it follows by Theorem 3 that every search problem in NP-search can be solved by a finite PS^{cwa} program. Conversely, for every fixed program P, the problem of computing models of a PS^{cwa} theory (D, P) (D is the input) is clearly in the class NP-search. Thus, only search problem in the class NP-search can be solved by finite PS^{cwa} programs. Hence, the assertion follows. $\square$