Predicate-calculus based logics for modeling and solving search problems

DEBORAH EAST
Texas State University - San Marcos
and
MIROSŁAW TRUSZCZYŃSKI
University of Kentucky

The answer-set programming (ASP) paradigm is a way of using logic to solve search problems. Given a search problem, to solve it one designs a logic theory so that *models* of this theory represent problem solutions. To compute a solution to the problem one computes a model of the theory. Several answer-set programming formalisms have been developed on the basis of logic programming with the semantics of answer sets. In this paper we show that predicate logic also gives rise to effective implementations of the ASP paradigm, similar in spirit to logic programming with the answer-set semantics and with a similar scope of applicability. Specifically, we propose two logics based on predicate calculus as formalisms for encoding search problems. We show that the expressive power of these logics is given by the class NPMV. We demonstrate their use in programming and discuss computational approaches to model finding. To address this latter issue, we follow a two-pronged approach. On one hand, we show that the problem can be reduced to that of computing models of propositional theories and more generally, of collections of *pseudo-boolean* constraints. Consequently, programs (solvers) developed in the areas of propositional and pseudo-boolean satisfiability can be used to compute models of theories in our logics. On the other hand, we develop native solvers designed specifically to exploit features of our formalisms. We present experimental results demonstrating computational effectiveness of the overall approach.

Categories and Subject Descriptors: I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods —*predicate logic*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Logic and constraint programming*

General Terms: Predicate logic, Search problems, Constraints

Additional Key Words and Phrases: Satisfiability, pseudo-boolean constraints

1. INTRODUCTION

In this paper we show that predicate calculus gives rise to a declarative language for modeling search problems and enables a uniform way of solving them by pro-

Author's address: Deborah East, Department of Computer Science, Texas State University - San Marcos, San Marcos, TX 78666, USA. Mirosław Truszczyński, Department of Computer Science, University of Kentucky, Lexington, KY 40506-0046.
Parts of this paper appeared in Proceedings of AAAI-2000 [East and Truszczyński 2000] and in Proceedings of KI-2001 [East and Truszczyński 2001].

viding a programming front-end for methods to compute models of propositional formulas and sets of pseudo-boolean constraints. In the way we design our language, and interpret and process programs, we adhere to general principles of the answer-set programming (ASP, for short) [Marek and Truszczyński 1999; Niemelä 1999]. However, unlike typical implementations of ASP that are based on logic programming with the stable-model semantics and its more general variant, disjunctive logic programming with the answer-set semantics, our formalism uses the syntax of predicate calculus and the semantics of Herbrand models.

Logic is most commonly used in declarative programming as follows. To solve a problem, we represent its general constraints and relevant background knowledge as a theory. We express a specific instance of the problem as a formula. We then use proof techniques to decide whether this formula follows from the theory. A proof of the formula (more precisely, a variable substitution constructed by the proof) determines a solution, which in most cases is represented by a ground term. This use of logic in programming and computing stems from the pioneering work by Robinson [1965], Green [1969] and Kowalski [1974]. It led to the establishment of *logic programming* as, arguably, the most prominent and most broadly accepted logic-based declarative programming formalism, and to the development of Prolog as its implementation by Colmerauer and his group [Colmerauer et al. 1973].

Recently, researchers proposed an alternative way to use logic in declarative problem solving, commonly referred to as *answer-set programming* (or ASP) paradigm [Marek and Truszczyński 1999; Niemelä 1999]. In ASP, one represents a computational problem as a theory in some logic so that *models* of this theory, and not proofs or variable substitutions, represent problem solutions. In ASP, finding models rather than proofs is a primary computational task and serves as a uniform processing mechanism.

The concept of the ASP paradigm emerged from the area of *stable logic programming* (SLP, for short), that is, logic programming with the stable-model semantics [Gelfond and Lifschitz 1988][1]. Over the years researchers demonstrated that problems such as planning, reasoning about action, diagnosis and abduction can be described by logic programs so that stable models of these programs represented problem solutions. [Baral 2003] provides an in-depth discussion of these applications and is a good source of references. Soon it became clear, however, that SLP can also be used to encode constraint satisfaction problems [Marek and Truszczyński 1999; Niemelä 1999] and, more generally, a broad class of search problems [Saccà 1997; Marek and Remmel 2003]. In all cases, the approach was the same. Programs encoding problems were constructed so that their stable models encoded problem solutions. An important development, significantly simplifying modeling tasks, was the extension of the basic language with *aggregates* — language constructs to represent constraints on *sets* of ground atoms [Simons et al. 2002].

In general, answer sets of (disjunctive) logic programs are infinite and, unless one devises for them some finitary representation schema, they cannot be computed. To

[1]There is an extension of SLP that allows a richer language (disjunctions in the heads of program rules and two types of negation, default and strong) and whose semantics is given by *answer sets* [Gelfond and Lifschitz 1991]. It is this formalism and, more precisely, its semantics, that gave rise to the term *answer-set programming*.

overcome this difficulty, it is common in ASP to restrict attention to programs that are finite and do not contain function symbols. Answer sets of such programs are finite sets of ground literals and there are algorithms to compute them. Two most advanced implementations of such algorithms are *smodels* [Niemelä and Simons 2000] (in the case without disjunctions) and *dlv* [Eiter et al. 1998; Leone et al. 2003]. These implementations compute answer sets in two steps. First, an input program is *grounded*, that is, replaced by a program that consists of ground rules only and has the same answer sets as the original one. Second, answer sets of the *ground* program are computed by means of search algorithms.

The two systems we mentioned, *smodels* and *dlv*, while structured similarly as implementations of the Davis-Putnam method for propositional satisfiability (or SAT) testing, develop and exploit search techniques specific to the case of logic programming with the answer-set semantics and to the types of aggregate constraints they support. Two other systems, *cmodels* [Babovich and Lifschitz 2002] and *assat* [Lin and Zhao 2002], reduce the problem of answer-set computation directly to that of computing models of propositional theories and then use off-the-shelf SAT solvers such as *zchaff* [Lin and Zhao 2002]. *Cmodels* and *assat* are noteworthy as they point to connections with SAT and open a possibility to take advantage in ASP of recent dramatic advances in the SAT area. Both programs have, however, some limitations. The translations to SAT instances, which *cmodels* and *assat* use, for some programs lead to very large SAT instances (exponential in the size of the original program). Moreover in order to use SAT solvers, both programs have first to compile away aggregate constraints, that is, replace them with equivalent propositional theories. Consequently, theories get bigger and performance often degrades.

The notion of the answer-set programming paradigm first appeared explicitly in the context of logic programming with the semantics of answer sets (originally, as we mentioned, under the restriction to normal programs and the semantics of stable models). However, it is clear that its general principle of *models representing solutions* applies to any logic system where the concept of a model is well defined. Our goal in this paper is to provide arguments for this more general view of the ASP paradigm. To this end, we show that predicate logic with the semantics given by Herbrand models, together with SAT solvers as processing engines, leads to an effective implementations of the (broadened) ASP paradigm. A specific logic we propose to this end is a modification of the logic of propositional schemata, which was developed as a language to encode planning problems [Kautz et al. 1996]. The key concept of our approach is that of a *data-program* pair (D, P), which represents a search problem Π by the *program* (collection of rules) P, and a concrete instance of Π by the *data* (collection of ground atoms) D. To define the semantics of data-program pairs, we restrict the class of Herbrand models of the theory $D \cup P$ to those Herbrand models that satisfy a version of Reiter's Closed-World Assumption. We refer to our logic as the logic of propositional schemata with Closed-World Assumption or, simply, as the logic of propositional schemata. We denote this logic by PS.

Through grounding, the semantics of theories in the logic PS and, in particular, of data-program pairs reduces to the standard propositional semantics. That makes it straightforward to extend both the language and the semantics of the logic PS to provide direct means for modeling constraints that do not have simple propositional encodings. In particular, we extend the logic *PS* with constructs 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". In that we are similar to *smodels* and *dlv* that also support concise encodings of cardinality constraints (as well as some other aggregates) [Simons et al. 2002; Dell'Armi et al. 2003].

We also consider an extension of the logic *PS* by *definite Horn* rules to support concise ways to model closures of ground facts under inference rules (in particular, to represent the transitive closure of binary relations). This task is simple in logic programming formalisms but not in predicate calculus. Consequently, the semantics of this extension of the logic *PS* gets more complex. Models have to satisfy an additional *postcondition*. However, it is still simpler than the fixpoint condition that appears in the definition of stable models and answer-sets. We refer to these extensions of our logic as *extended logic of propositional schemata* (with Closed-World Assumption) and denote it by *PS+*. An important point to make here is that the semantics of the logic *PS+* is a direct generalization of the semantics of the logic *PS*.

In the paper we study basic properties of the logic *PS* and observe that they extend to the logic *PS+*, as well. We show that the logic *PS* is nonmonotonic, identify sources of nonmonotonicity and its implications. We demonstrate the use of the logic *PS* as a representation language by developing programs for several search problems. We characterize the class of problems that can be solved by programs in the logic *PS*. To this end, we define a formal setting for the study of the expressive power of ASP formalisms. We establish that the expressive power of the logic *PS* is equal to the class NPMV [Selman 1994]. In particular, it is the same as the expressive power of SLP.

As we pointed out, when using the logics *PS* and *PS+* to solve a problem for a particular instance, we represent the problem and the instance by a data-program pair so that Herbrand models of the data-program pair correspond to problem solutions. Consequently, the basic computational task is that of computing Herbrand models. Since these Herbrand models are precisely the models of propositional theories obtained by grounding data-program pairs (computing their equivalent propositional representation), the task can be accomplished in a similar two-step process to that used to compute answer sets. Given a finite data-program pair, we first ground it and then find models of the ground theory obtained.

For grounding (the first step), we implemented a program, *psgrnd* that, given a data-program pair, produces an equivalent theory in the propositional fragment of the logic *PS+*, in which cardinality constraints (and Horn rules, if they are allowed in the language) have explicit representations. To compute models of theories produced by *psgrnd*, we developed a satisfiability solver program, *aspps*. It is designed along the same lines as most satisfiability solvers implementing the Davis-Putnam algorithm but it takes advantage of the cardinality and closure constraints present in the language.

For data-program pairs that do not involve Horn rules there are alternatives to *aspps*. If the input data-program pair is in the language of the basic logic *PS* (no cardinality constraints and no Horn rules), the result of the grounding step is simply a (standard) propositional theory. In fact, one of the options of *psgrnd* outputs the result as a CNF theory in the DIMACS format. If the input data-program pair contains cardinality constraints (but still no Horn rules), the result of grounding is a theory in a logic that is closely related to the propositional logic with pseudo-boolean constraints, which received much attention in the satisfiability and constraint satisfaction communities [Benhamou et al. 1994; Barth 1995; Walser 1997; Aloul et al. 2002; Dixon and Ginsberg 2002; Prestwich 2002], Pseudo-boolean constraints are essentially linear inequalities with integer coefficients and with domains of all variables restricted to 0 and 1. They generalize constraints specified by propositional clauses but can also express concisely more complex constraints involving aggregates such as cardinality of a set or sum of (integer) weights of elements in the set.

There are simple translations of theories in the propositional logic *PS+* into the syntax of pseudo-boolean propositional logic that in most cases do not lead to any significant growth in the theory size. It follows that in the second phase of the process to compute models of *PS+* theories without Horn rules, one can use "off-the-shelf" SAT solvers and solvers for sets of pseudo-boolean constraints (SAT(PB) solvers, for short). In this way, our logic *PS+* and our program *psgrnd* provide a uniform programming front-end for SAT and SAT(PB) solvers and facilitate their use.

Experimental results on the performance of the overall approach are encouraging. They show that the logic *PS+* can be an effective tool to model search problems and can serve as a programming front-end for a broad range of processing back-ends, including SAT and SAT(PB) solvers, as well as native *PS+* solvers such as *aspps*. They demonstrate that our solver *aspps* is competitive with current ASP solvers such as *smodels*, with complete SAT solvers such as *zchaff* and *satz* and with a SAT(PB) solver *PBS*. In fact, in several instances we considered, *aspps* was faster. Our experiments also provide further evidence that building propositional solvers capable of processing high-level constraints is a promising research direction for both propositional satisfiability and constraint satisfaction communities.

Several interrelated factors motivate us in this work. Our first goal was to broaden the scope of the ASP paradigm. Its roots are in the formalism of logic programming and in knowledge representation as the intended application area. However, the basic tenet of the ASP paradigm applies to any logic with well-defined notion of a model. Its scope of applicability goes beyond knowledge representation and is best described in terms of classes of search problems or, equivalently, partial multivalued functions [Selman 1994; Marek and Remmel 2003]. Our logics *PS* and *PS+*, the corresponding programs *psgrnd* and *aspps*, and their performance substantiate this more general view of ASP.

Second, we aimed at the development of an effective programming front-end that would capitalize on dramatic improvements in the performance of SAT and SAT(PB) solvers and would facilitate their use as computational tools. In recent years, researchers have developed several fast implementations of the basic Davis-Putnam method such as *satz* [Li and Anbulagan 1997], *relsat* [Bayardo, Jr and Schrag 1997] and, most recently, *zchaff* [Moskewicz et al. 2001a; 2001b]. A renewed interest in local-search techniques resulted in highly effective (albeit incomplete) satisfiability checkers such as *WALKSAT* [Selman et al. 1994], capable of handling large CNF theories, consisting of millions of clauses. Similarly, the research in pseudo-boolean satisfiability resulted in several effective complete and local-search solvers building both on artificial intelligence search and on operations research techniques [Barth 1995; Walser 1997; Dixon and Ginsberg 2002; Prestwich 2002; Liu and Truszczyński 2003]. With the focus on the design of fast solvers, the issue of modeling tools has received relatively little attention. Our logic *PS*+, providing a programming front-end to programs computing models of propositional theories and collections of pseudo-boolean constraints, is a way to address the matter.

Third, the semantics of our logics, given by a class of Herbrand models, is essentially the semantics of propositional logic and, consequently, results in a system that is, we believe, simpler than those based on logic programming with the stable-model or answer-set semantics. While this claim is by its very nature subjective, there is an argument to support it. In all formalisms we discussed, be it our logics or systems based on logic programming, the semantics of programs with variables is lifted, through grounding, from the semantics in the corresponding propositional case. In logic-programming based approaches, the semantics of propositional programs is already non-standard (in particular, non-monotonic) and requires concepts such as the reduct, least model computation (or minimality with respect to closure under the reduct rules) and, ultimately, some fixpoint condition. In our case, the semantics in the propositional case that we lift up through grounding is *exactly* the propositional semantics. The only difficulty that remains for the programmer is to understand the process of grounding. In contrast, in logic-programming approaches the programmer needs to understand intricacies of the semantics at the ground level in addition to understanding how the grounding works.

Our paper is organized as follows. In the next section we introduce the syntax and the semantics of the logic *PS*. We describe there also the process of grounding. Section 3 presents an extension of our basic logic by the equality and arithmetic operations and relations. We define the key concept of data-program pairs in Section 4. The following section presents several examples illustrating the use of the logic *PS* and of data-program pairs in representing some well-known constraint satisfaction problems. We study the expressive power of the logic *PS* in Section 6. In Section 7, we discuss extensions of the logic *PS* by syntactic constructs to represent cardinality and closure constraints. Section 8 provides a brief overview of the program *psgrnd* that we designed as a tool to ground *PS*+ data-program pairs. It also describes the program *aspps*, a tool we designed to compute models of ground *PS*+ theories. We present and discuss experimental results on the performance of of the overall approach in Section 9. They include comparisons of *aspps* with SAT and SAT(PB) solvers, as well as with *smodels*, an ASP solver based on the syntax of logic programming. The last section discusses the results of the paper, as well as some related work. It also outlines several directions for future research.

2. BASIC LOGIC *PS*

In this section, we introduce the logic *PS* that provides a theoretical basis for a declarative programming front-end to satisfiability solvers and facilitates their use.

2.1 Syntax

Syntactically, the logic *PS* is a fragment of first-order logic without function symbols (or, in other words, *predicate logic*). Specifically, the language of the logic *PS* consists of:

(1) infinite denumerable sets R, C and V of *relation*, *constant* and *variable* symbols
(2) symbols $\bot$ and $\top$ (later interpreted always as falsity and truth)
(3) boolean connectives $\wedge$, $\vee$ and $\rightarrow$, the quantifiers $\exists$ and $\forall$, and punctuation symbols '(', ')' and ','.

In the paper, following the example of logic programming, we adopt the convention that upper-case letters denote variables and lower-case letters stand for constants.

Constant and variable symbols are the only *terms* of the language. Constants are the only *ground* terms and they form the *Herbrand universe* of the language. *Atoms* are expressions of the form $p(t_1, \ldots, t_n)$, where p is an n-ary relation symbol from R and t_i, $1 \leq i \leq n$, are terms. An atom $p(t_1, \ldots, t_n)$ is *ground* if all its terms are ground. The set of all ground atoms forms the *Herbrand base* of the language.

In the logic *PS*, we restrict the use of existential quantifiers. Let us consider a tuple of terms $(t_1, \ldots, t_n)$ and let $X_1, \ldots, X_k$ be pairwise distinct variables such that each X_i, $1 \leq i \leq k$, appears in the tuple $(t_1, \ldots, t_n)$ exactly once. An expression of the form

$$\exists X_1, \ldots, X_k \;\; p(t_1, \ldots, t_n)$$

is an *e-atom*. For instance, the expression $\exists X, Z \;\; p(X, Y, Z, c)$ is an e-atom while the expression $\exists X, Z \;\; p(X, X, Z, c)$ is not. In the logic *PS*, existential quantifiers appear *exclusively* in e-atoms.

The requirement that each X_i, $1 \leq i \leq k$, appears in the tuple $(t_1, \ldots, t_n)$ exactly once is not essential and can be lifted (we show one way how to do it in Section 7). We adopt it here as it allows us to simplify the notation for e-atoms. Namely, we write an e-atom

$$\exists X_1, \ldots, X_k \;\; p(t_1, \ldots, t_n)$$

as

$$p(t'_1, \ldots, t'_n),$$

where $t'_i = t_i$, if t_i is not one of the variables $X_1, \ldots, X_k$, and $t_i =$ '_' (underscore), otherwise. For instance, we write an e-atom $\exists X, Z \;\; p(X, Y, Z, c)$ as $p(\_, Y, \_, c)$. We emphasize that e-atoms are not atoms in the standard sense. They are *formulas* of a particular syntactic structure. We chose the term e-atom to reflect our simplified notation in which we commonly write them.

The *only* formulas we allow in the logic *PS* are *rules*, that is, formulas

$$\forall X_1, \ldots, X_k (A_1 \wedge \ldots \wedge A_m \rightarrow B_1 \vee \ldots \vee B_n),$$

where all A_i, $1 \leq i \leq m$, and B_j, $1 \leq j \leq n$, are either atoms or e-atoms, none of A_i's is an e-atom (in other words, e-atoms do not appear in the antecedents of rules) and $X_1, \ldots, X_k$ are the free variables appearing in $A_1 \ldots, A_m$ and $B_1, \ldots, B_n$. If $m = 0$, we replace the conjunct in the antecedent of the rule with the symbol $\top$. If

$n = 0$, we replace the empty disjunct in the consequent of the rule with the symbol $\perp$.

As usual, we drop the universal quantifiers from the rule notation. For instance, to denote the rule

$$\forall X, Y, Z(p(X,Y) \wedge p(Y,Z) \rightarrow q(\_,X) \vee q(Y,\_) \vee r(Z)),$$

which is already a shorthand for

$$\forall X, Y, Z(p(X,Y) \wedge p(Y,Z) \rightarrow \exists W \;\; q(W,X) \vee \exists W \;\; q(Y,W) \vee r(Z)),$$

we write

$$p(X,Y) \wedge p(Y,Z) \rightarrow q(\_,X) \vee q(Y,\_) \vee r(Z).$$

This notation is reminiscent of that commonly used for *clauses* in predicate logic. There is a key difference, though. Some of the atoms in the consequent of a rule may be e-atoms (as it is the case in the example just given). Thus, unlike in the case of clauses, a rule of the logic *PS* may contain the existential quantifier in the consequent.

The last syntactic notion we need is that of a theory. A *theory* in the logic *PS* (or, a *PS* theory) is any *finite* collection of rules that contains at least one occurrence of a constant symbol.

To recap the discussion of the syntax of the logic *PS*, it is essentially a fragment of first-order logic with the following restrictions and caveats: (1) function symbols are not allowed, (2) rules are the only formulas, (3) theories are finite and contain at least one constant symbol, and (4) through the use of notational conventions, the quantifiers are only implicitly present in the language.

2.2 Semantics

The difference between the logic *PS* and the corresponding fragment of the first-order logic is in the way we interpret theories. Namely, we view a theory T as a representation of a *certain class of models of* T and not as a representation of logical consequences of T. In fact, due to the way we use the logic *PS*, the concept of provability plays virtually no role in logic *PS*. This is an essential departure from the classical first-order logic perspective.

Specifically, we assign to a *PS* theory T a collection of its *Herbrand models*. The concepts of an Herbrand interpretation, of truth in an interpretation and of an Herbrand model, which we use in the paper, are standard (for details, we refer to any text in logic, for instance, [Nerode and Shore 1993]). Here we will only introduce some necessary notation. Let T be a *PS* theory. We denote by $HU(T)$ the *Herbrand universe* of T, that is, in our case, the set of all constants that *appear* in T. By the definition of a *PS* theory, this set is finite and non-empty. We denote by $HB(T)$ the *Herbrand base* of T, that is, the collection of all ground atoms $p(c_1, \ldots, c_n)$, where p is an n-ary relation symbol appearing in T and $c_i \in HU(T)$, $1 \leq i \leq n$. Following a standard practice, we identify Herbrand interpretations of T with subsets of $HB(T)$.

The restriction to Herbrand interpretations is important. In particular, it implies that the logic *PS* is *nonmonotonic*. Indeed, if $T_1 \subseteq T_2$ are two *PS* theories, it is not necessary that every Herbrand model of T_2 is a Herbrand model of T_1. For example, let $T_1 = \{\top \rightarrow p(\_),\ \ p(a) \rightarrow \bot\}$, and let $T_2 = T_1 \cup \{\top \rightarrow p(b)\}$. It is easy to see that $M = \{p(b)\}$ is a Herbrand model of T_2 and that T_1 has no Herbrand models. Indeed, $HU(T_1) = \{a\}$ and the only Herbrand model satisfying the first rule is $M' = \{p(a)\}$. This model, however, does not satisfy the second rule. In contrast, classical first-order logic is monotone: for every two collections of sentences $T_1 \subseteq T_2$, if M is a model of T_2 then it is a model of T_1, as well. In the example discussed above, the Herbrand model specified by the subset $\{p(b)\}$ of $HB(T_2)$ is a model of T_1 but *not* a Herbrand model of T_1.

The nonmonotonicity of the logic PS is not surprising and properties such as the one discussed above are not limited to the logic PS only. Restricting the semantics of first-order logic to special classes of models is a well-known general mechanism to specify nonmonotonic logics. For instance, considering only *minimal* models yields *circumscription* [McCarthy 1980], one of the most influential nonmonotonic logics. Similarly, in logic programming, building the semantics on the class of *stable models* rather than on the class of all models, leads to stable logic programming, an extensively studied nonmonotonic system [Gelfond and Lifschitz 1988; Marek and Truszczyński 1993; Baral 2003] with wide applications in knowledge representation. It is somewhat surprising that, despite its simplicity, the logic PS was overlooked by the nonmonotonic-logic community and, apparently, has not received any attention so far.

2.3 Models of PS theories, grounding and propositional satisfiability

The restriction to Herbrand models is not only responsible for the nonmonotonicity of the logic PS. It also allows us to develop algorithms to compute models of PS theories. Namely, as in the case of stable logic programming, models of a PS theory T (these models are, by definition, Herbrand models) can be computed in two steps. First, we *ground* T to a propositional theory that has the same Herbrand models as T. Next, we compute models of T by computing models of the ground theory. This latter task can be accomplished by off-the-shelf propositional satisfiability solvers.

The concept of grounding is similar to that used in the context of universal theories in first-order logic or programs in logic programming. The only difference comes from the fact that rules in PS theories may include e-atoms in the consequents. We will now discuss the task of grounding in detail. Let t be a term tuple (some of its components may be the underscore symbols '_') and let ϑ be a ground variable substitution that contains in its domain all variables appearing in t. By $t\vartheta$ we denote a term tuple t' obtained from t by replacing each variable X appearing in t with the constant assigned to X by ϑ. Next, we define $p(t)\vartheta$ as the disjunction of all ground atoms of the form $p(t')$, where t' is obtained from t by replacing all occurrences of the underscore symbol '_' in t with constants from $HU(T)$, that is, constants that appear in T. We note that $p(t)\vartheta$ depends on T. We also note that if t contains no occurrences of the underscore symbol, then $p(t)\vartheta = p(t\vartheta)$ (no disjunction).

Further, for a rule $r \in T$, where

$$r = \quad A_1 \wedge \ldots A_m \rightarrow B_1 \vee \ldots \vee B_n,$$

we define

$$r\vartheta = \quad A_1\vartheta \wedge \ldots A_m\vartheta \rightarrow B_1\vartheta \vee \ldots \vee B_n\vartheta.$$

We note that atoms A_i are not e-atoms. Thus, $A_i\vartheta$, $1 \leq i \leq m$, is a single ground atom (grounding does not introduce disjunctions in the antecedents of the rules). We also note that B_is may be e-atoms and $B_i\vartheta$ may, in fact, be a disjunction of ground atoms.

Finally, we define $gr(T)$ to consist of all rules $r\vartheta$, where $r \in T$ and ϑ is a ground substitution containing all variables in r in its domain.

We will illustrate the concepts we introduced with an example. Let T be a *PS* theory that consists of the following two rules:

$$C_1 = \quad q(b,c) \rightarrow p(a)$$
$$C_2 = \quad p(X) \rightarrow q(X, \_).$$

To compute $gr(T)$ we need to compute all ground instances of C_2 (C_1 is already in the ground form). Since there are only three constants in the theory (a, b and c) and only one variable X in C_2, there are three ground variable substitutions to consider: $\vartheta_1 = \{X/a\}$, $\vartheta_2 = \{X/b\}$ and $\vartheta_3 = \{X/c\}$. Clearly, $(X, \_)\vartheta_1 = (a, \_)$ and $q(X, \_)\vartheta_1 = q(a,a) \vee q(a,b) \vee q(a,c)$. Thus,

$$C_2\vartheta_1 = \quad p(a) \rightarrow q(a,a) \vee q(a,b) \vee q(a,c).$$

The effects of applying ϑ_2 and ϑ_3 are similar. It follows that $gr(T)$ consists of C_1 together with the following ground rules:

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

The following proposition establishes the adequacy of the concept of grounding in the study of models of *PS* theories. The proof of the proposition is simple and reflects closely the corresponding argument in the first-order case. Thus, we omit it.

PROPOSITION 2.1. *Let T be a PS theory. A set $M \subseteq HB(T)$ is a Herbrand model of T if and only if M is a propositional model of $gr(T)$.*

This property is the basis for algorithms to compute models of *PS* theories that we develop later in the paper. The idea is that given a *PS* theory T, we first compute the propositional theory $gr(T)$ and, then, compute models of $gr(T)$ (for instance, by means of satisfiability provers). From that perspective, it is important to understand how the size (the total number of atom occurrences) of $gr(T)$ depends on the size of T. Let us assume that T is a *PS* theory of size $s(T)$. Let us also assume that T contains n constant symbols, that the arity of each predicate symbol appearing in T is at most k, and that each rule in T contains at most l different variables. Let r be a rule in T consisting of p atoms and let ϑ be a ground substitution containing in its domain all variables of r. Clearly, $r\vartheta$ consists of at most pn^k ground atoms (grounding may expand each atom in the head of r into a disjunction of n^k atoms). Since there are n^l different ways to ground variables appearing in r, the total size of all ground instantiations $r\vartheta$ of r is pn^{k+l}. It follows that the size of $gr(T)$ is bounded by $s(T)n^{k+l}$. In particular, if we fix k and l (theories we encounter when using our logic in programming are always of this type), the size of $gr(T)$ is polynomial in the size of T.

3. EQUALITY AND ARITHMETIC IN THE LOGIC PS

In the following sections, we will often consider a version of the logic PS, in which some relation symbols are given a prespecified interpretation. They are equality, inequality and basic arithmetic relations such as $\leq$, $<$, $\geq$, $>$, $+$, $*$, $-$, $/$, etc. Inclusion of these relation symbols in the language is important as they facilitate the task of programming (modeling knowledge, representing constraints as rules). We will use standard symbols to represent them, as well as the standard infix notation. In particular, we will write $t_1 = t_2$ rather than $=(t_1, t_2)$, and $t = t_1 + t_k$ (or $t_1 + t_2 = t$) rather than $+(t_1, t_2, t)$. We will denote the set of these relation symbols by EA.

In this section, we will define the semantics for this variant of the logic PS. The idea is to interpret all symbols in the set EA according to their intended meaning. Specifically, let C be the set of constant symbols. We define a theory $=_C$ to consist of all rules of the form

(1) $\top \rightarrow = (t,t)$ (we will write them as $\top \rightarrow (t = t)$), for every $t \in C$, and
(2) $= (s,t) \rightarrow \bot$ (we will write them as $(s = t) \rightarrow \bot$), for every $s, t \in C$ such that $s \neq t$.

Next, we define a theory $+_C$ to consist of all rules of the form

(1) $\top \rightarrow +(t,u,s)$ (we will write them as $\top \rightarrow (s = t + u)$), for every integers $s, t, u \in C$ such that $s = t + u$, and
(2) $+(t,u,s) \rightarrow \bot$ (we will write them as $(s = t+u) \rightarrow \bot$), for every $s, t, u \in C$ such that at least one of s, t, u is not an integer, or s, t, u are integers and $s \neq t + u$.

In the same way we define theories p_C for other relation symbols in EA such as $\leq$, $-$, $*$, etc. All these theories provide explicit intended definitions of the corresponding relation symbols. We will often refer to the relation symbols in EA as *predefined* since their interpretation is fixed.

Let T be a PS theory in the language containing distinguished relation symbols from the set EA. Let C be the set of constants appearing in T (that is, $C = HU(T)$). A set M of ground atoms in the language is a *model* of T if M is a model of the theory $T \cup \bigcup\{p_C : p \in EA\}$ as defined above.

It is clear that, with the help of additional variables, we can express in the logic PS arbitrary arithmetic expressions. For instance, we will write

$$q((X + Y) * Z, a) \wedge B \rightarrow H$$

and interpret this expression as

$$(T_1 = X + Y) \wedge (T_2 = T_1 * Z) \wedge q(T_2, a) \wedge B \rightarrow H$$

Similarly, we will interpret

$$B \rightarrow q((X + Y) * Z, a) \vee H$$

as the rule

$$B \wedge (T_1 = X + Y) \wedge (T_2 = T_1 * Z) \rightarrow q(T_2, a) \vee H.$$

In each case, variables T_1 and T_2 are different from all other variables appearing in the rules. In order to obtain uniqueness of the interpretation, when decomposing arithmetic expressions, we follow the standard order in which they are evaluated. Let us emphasize that arithmetic expressions are simply notational shortcuts and not elements of the language.

In the remainder of the paper, we will always assume that the language contains predefined relation symbols. Since their extensions are already fully specified, we will omit the corresponding ground atoms when describing models of theories.

To illustrate these concepts, let us consider the following *PS* theory T:

$$T = \{\top \rightarrow p(1),\ \ \top \rightarrow p(2),\ \ p(X) \rightarrow q(X, X+1)\}.$$

This theory represents the following *PS* theory T':

$$T' = \{\top \rightarrow p(1),\ \ \top \rightarrow p(2),\ \ p(X) \wedge (Y = X + 1) \rightarrow q(X, Y)\}.$$

The theory $gr(T')$ consists of the first two rules (they are already ground) and the following four instantiations of the third rule:

$$\begin{array}{l} p(1) \wedge (1 = 1 + 1) \rightarrow q(1, 1) \\ p(1) \wedge (2 = 1 + 1) \rightarrow q(1, 2) \\ p(2) \wedge (1 = 2 + 1) \rightarrow q(2, 1) \\ p(2) \wedge (2 = 2 + 2) \rightarrow q(2, 2). \end{array}$$

Models of this theory are (by the definition) models of the theory T. For instance, $\{p(1), p(2), q(1, 2)\}$ is a model of T. We point out that, according to our convention, we omitted from the model description the atom $2 = 1 + 1$ (or, more formally, the atom $=(1, 1, 2)$).

4. DATA-PROGRAM PAIRS AND THEIR SEMANTICS

The logic *PS*, described in the previous section, can be used as a basis for a declarative programming formalism based on the paradigm of answer-set programming [Marek and Truszczyński 1999]. To fully develop it, we need to introduce the concepts of input data and a program. We follow the approach proposed and studied in the area of relational databases [Ullman 1988]. A relational database can be viewed as a collection of ground atoms of some logic language. We often use (for instance, in the context of DATALOG and its variants) the term *extensional* database to refer to a collection of ground atoms specifying a relational database. Queries are finite theories, often of special form, in this logic language (for instance, definite Horn theories without function symbols serve as queries in the case of DATALOG). Queries define new properties (relations) in terms of those relations that are explicitly specified by the underlying (extensional) database.

Guided by these intuitions, we define a *data-program pair* to be a pair (D, P), where D is a finite set of ground atoms in a language of the logic *PS* and P is a finite collection of *PS* rules. We use data-program pairs to represent specific computational problem instances. We view D as an encoding of relevant input data and P as a declarative specification of the computational task in question.

Accordingly, given a data-program pair (D, P), we refer to D as a *data set* and to P as a *program*. We use the term *data predicate* for all relation symbols appearing in atoms in D. We use the term *program predicate* to refer to all relation symbols that appear in P and are neither data predicates nor predefined predicates from EA. Intuitively, D is a counterpart of an extensional database and P is a counterpart of a database query.

We will now introduce a semantics for data-program pairs. To this end, we will encode a data-program pair (D, P) as a theory in the logic PS. Since D is a set of ground atoms representing the problem instance (input data), we assume that D provides a *complete* specification of the input. That is, we assume that no other ground atoms built of predicates appearing in D are true. Since the only formulas in the logic PS are rules, we encode the information specified by D as a set of rules $cl(D)$ defined as follows. For every relation symbol p appearing in D and for every ground tuple t (with constants from the Herbrand universe of $D \cup P$) of appropriate arity, if $p(t) \in D$, we include in $cl(D)$ the rule

$$\top \to p(t).$$

Otherwise, if $p(t) \notin D$, we include in $cl(D)$ the rule

$$p(t) \to \bot.$$

It is clear that the set $cl(D)$ can be regarded as the result of applying Reiter's Closed-World Assumption to D.

We represent a data-program pair (D, P) by a PS theory $cl(D) \cup P$. We say that a set M of ground atoms, $M \subseteq HB(cl(D) \cup P)$, is a *model* of a data-program pair (D, P) if it is a model of $cl(D) \cup P$. We denote the set of all models of a data-program pair (D, P) by $Mod(D, P)$.

In separating data and program predicates and in adopting the closed-world assumption for the treatment of data atoms we are guided by the intuition that data predicates are intended to represent input data. Their extensions should not be affected by the computation. The effects of the computation should be reflected in the extensions of program predicates only.

5. PROGRAMMING WITH LOGIC PS

We designed the logic PS and introduced the concept of a data-program pair to model computational problems. To illustrate this use of our formalism, we show how to encode several well-known search problems by means of data-program pairs. We assume that the language contains predefined relation symbols to represent equality and arithmetic relations.

We start with the **graph k-colorability problem**: given an undirected graph and a set of k colors, the objective is to find an assignment of colors to vertices so that no two identically colored vertices are joined with an edge (or to determine that no such coloring exists).

We set

$$D_{gcl}(G, k) = \{vtx(v) \colon v \in V\} \cup \{edge(v, w) \colon \{v, w\} \in E\} \cup \{color(i) \colon 1 \le i \le k\}.$$

The set of atoms D_{gcl} represents an instance of the coloring problem. The predicates vtx, $edge$ and $color$ are data predicates. Their extensions define vertices and edges of an input graph, and the set of available colors.

Next, we construct a program, P_{gcl}, encoding the constraints of the problem. It involves predicates *vtx*, *edge* and *color*, specified in the data part, and defines a new relation *clrd* that models assignments of colors to vertices.

C1: $clrd(X,C) \rightarrow vtx(X)$
C2: $clrd(X,C) \rightarrow color(C)$
C3: $vtx(X) \rightarrow clrd(X, \_)$
C4: $clrd(X,C) \wedge clrd(X,D) \rightarrow (C = D)$
C5: $edge(X,Y) \wedge clrd(X,C) \wedge clrd(Y,C) \rightarrow \perp$.

The condition (C1) states that the only objects that get colored are vertices. Indeed, by the definition, a model of the theory $(D_{gcl}(G,k), P_{gcl})$ contains an atom $vtx(x)$ if and only if x is a vertex of an input graph. Thus, if $clrd(v,c)$ belongs to a model of $(D_{gcl}(G,k), P_{gcl})$, then $vtx(v)$ belongs to the model and, so, v is a vertex. Similarly, (C2) states that the only objects assigned by the predicate *clrd* to a vertex are colors. (C3) states that each vertex X gets assigned at least one color. (C4) enforces that each vertex is assigned at most one color. (C5) ensures that two vertices connected by an edge are assigned different colors. These rules correctly capture the constraints of the coloring problem.

Proposition 5.1. *Let $G = (V,E)$ be an undirected graph and let k be a positive integer. If an assignment $f\colon V \rightarrow \{1,\ldots,k\}$ is a k-coloring of G then $M = D_{gcl}(G,k) \cup \{clrd(v,f(v))\colon v \in V\}$ is a model of the data-program pair $(D_{gcl}(G,k), P_{gcl})$. Conversely, if M is a model of the data-program pair $(D_{gcl}(G,k), P_{gcl})$ then $M = D_{gcl}(G,k) \cup \{clrd(v,f(v))\colon v \in V\}$, for some k-coloring $f\colon V \rightarrow \{1,\ldots,k\}$ of G.*

Proof: ($\Rightarrow$) Let us assume that $f\colon V \rightarrow \{1,\ldots,k\}$ is a k-coloring of G. We will show that $M = D_{gcl}(G,k) \cup \{clrd(v,f(v))\colon v \in V\}$ is a model of (D_{gcl}, P_{gcl}), that is, it is a model of $gr(cl(D_{gcl}(G,k)) \cup P_{gcl})$. From the definition of M it follows that M satisfies all rules in $cl(D_{gcl}(G,k))$. We will now show that M satisfies all rules in $gr(cl(D_{gcl}(G,k)) \cup P_{gcl})$ that are obtained by grounding rules in P_{gcl}.

First, we consider an arbitrary ground instance of rule (C1), say, $clrd(x,c) \rightarrow vtx(x)$, where x and c are two constants of the language. It is clear from the definition of M that if $clrd(x,c) \in M$, then $x \in V$ and, consequently, $vtx(x) \in M$. Thus all ground instances of (C1) are satisfied by M.

Next, we consider a ground instance r of rule (C3), say,

$$r = \quad vtx(x) \rightarrow \bigvee\{clrd(x,c)\colon c \in V \cup \{1,\ldots,k\}\},$$

where $x \in V \cup \{1,\ldots,k\}$. If $vtx(x) \in M$, then $x \in V$. Since $f(x) \in \{1,\ldots,k\}$, and $clrd(x,f(x)) \in M$, it follows that r is satisfied by M. All other rules can be dealt with in a similar way.

($\Leftarrow$) We will now assume that M is a model of (D_{gcl}, P_{gcl}). By the definition of a model, we have (1) $vtx(x) \in M$ if and only if $x \in V$, (2) $edge(x,y) \in M$ if and only if $\{x,y\} \in E$, and (3) $color(i) \in M$ if and only if $i \in \{1,\ldots,k\}$.

Now, we observe that since M satisfies all ground instances of (C1), if $clrd(x,c) \in M$, then $x \in V$. Similarly, since M satisfies all ground instances of (C3), for every $x \in V$ there is at least one constant c such that $clrd(x,c) \in M$. On the other hand, since M satisfies all ground instances of (C2), for each such constant c, $c \in \{1, \ldots, k\}$. Next, we have that M satisfies all ground instances of (C4). Consequently, for every $x \in V$ there is exactly one $c \in \{1, \ldots, k\}$ such that $clrd(x, c) \in M$. Let us denote by f the function that assigns to each $x \in V$ the unique $c \in \{1, \ldots, k\}$ such that $clrd(x, c) \in M$. It follows that $M = D \cup \{clr(v, f(v)) \colon v \in V\}$. Moreover, as M satisfies all ground instances of (C5), f is a k-coloring of G. □

Let us note that the proof of Proposition 5.1 implies, in fact, that the correspondence between models and colorings is a bijection.

Next, we will describe a data-program pair encoding an instance of 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, $k \leq |V|$, 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:

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

This set of atoms specifies the set of vertices and the set of edges of an input graph. It also provides a set of k *indices* which we will use to select a subset of no more than k vertices in the graph, a candidate for a vertex cover of cardinality at most k.

The vertex cover problem itself is described by the program P_{vc}. It introduces a new relation symbol vc. Intuitively, we use vc to represent the fact that a vertex has been selected to a candidate set.

VC1: $vc(I, X) \rightarrow vtx(X)$
VC2: $vc(I, X) \rightarrow index(I)$
VC3: $index(I) \rightarrow vc(I, \_)$
VC4: $vc(I, X) \wedge vc(I, Y) \rightarrow X = Y$
VC5: $edge(X, Y) \rightarrow vc(\_, X) \vee vc(\_, Y)$.

(VC1) and (VC2) ensure that $vc(i, x)$ is false if i is not an integer from the set $\{1, \ldots, k\}$ or if x is not a vertex (that is, if $vc(i, x)$ is true, $i \in \{1, \ldots, k\}$ and $x \in V$). The rules (VC3) and (VC4) together impose the requirement that every index i has exactly one vertex assigned to it. It follows that the set of ground atoms $vc(i, x)$ that are true in a model of the data-program pair $(D_{vc}(G, k), P_{vc})$ defines a subset of V with cardinality at most k. Finally, (VC5) ensures that each edge has at least one end vertex assigned by vc to an index from $\{1, \ldots, k\}$ (in other words, that vertices assigned to indices $1, \ldots, k$ form a vertex cover). The correctness of this encoding is formally established in the following result. Its proof is similar to that of Proposition 5.1 and we omit it.

PROPOSITION 5.2. *Let $G = (V, E)$ be an undirected graph and let k be a positive integer. If $W \subseteq V$ is a vertex cover of G and $|W| \leq k$, then for every sequence $w_1, \ldots, w_k$ that enumerates all elements in W (possibly with repetitions), $M = D_{vc}(G, k) \cup \{vc(i, w_i) \colon i = 1, \ldots, k\}$ is a model of the data-program pair $(D_{vc}(G, k), P_{vc})$. Conversely, if M is a model of $(D_{vc}(G, k), P_{vc})$ then the set $W = \{w \in V \colon vc(i, w) \in M,$ for some $i = 1, \ldots, k\}$ is a vertex cover of G with $|W| \leq k$.*

In this case, we do not have a one-to-one correspondence between models and vertex covers of cardinality at most k. This is because we represent sets by means of sequences.

Next, we will consider the **Hamiltonian-cycle problem** in a directed graph. To represent an input graph $G = (V, E)$ we use the following set of ground atoms:

$$D_{hc}(G) = \{vtx(x) \colon x \in V\} \cup \{edge(x, y) \colon (x, y) \in E\} \cup \{index(i) \colon i = 1, \ldots, |V|\}.$$

The set of indices is introduced as part of input because we will represent a Hamiltonian cycle by a bijective sequence of vertices such that every two consecutive vertices in the sequence. as well as the last and the first, are connected with an edge. To represent such sequences we use a relations symbol *hc_perm*. The program, P_{hc}, defining "Hamiltonian" sequences $hc\_perm(i, x)$ looks as follows. In this example we assume that $\oplus$ denotes a predefined relation of addition modulo n defined on the set of integers $\{1, \ldots, n\}$ (thus, in particular, $n \oplus 1 = 1$).

HC1: $hc\_perm(I, X) \rightarrow index(I)$
HC2: $hc\_perm(I, X) \rightarrow vtx(X)$
HC3: $index(I) \rightarrow hc\_perm(I, \_)$
HC4: $vtx(X) \rightarrow hc\_perm(\_, X)$
HC5: $hc\_perm(I, X) \wedge hc\_perm(I, Y) \rightarrow X = Y$
HC6: $hc\_perm(I, X) \wedge hc\_perm(J, X) \rightarrow I = J$
HC7: $hc\_perm(I, X) \wedge hc\_perm(I \oplus 1, Y) \rightarrow edge(X, Y)$
HC8: $hc\_perm(1, 1)$.

The first two rules ensure that if $hc\_perm(i, x)$ is true in a model of (D_{hc}, P_{hc}) then i is an integer from the set $\{1, \ldots, |V|\}$ and $x \in V$. The rules (HC3) - (HC6) together enforce the constraint that *hc_perm* defines a *permutation* of vertices. We note that each of the rules (HC3) and (HC6) is redundant as it is implied by the three others. However, our experiments show that having all four rules represented *explicitly* in the program results in better computational properties. The rule (HC7) imposes the Hamiltonicity constraint that from every vertex in the sequence to the next one (and from the last one to the first one, too) there is an edge in the graph. Finally, the last rule functions as a symmetry breaker. Without loss of generality, we may assume that the cycle represented by the predicate *hc_perm* starts with the vertex 1 and that fact is enforced by the rule (HC8). Clearly, that rule is not necessary for the correctness of the encoding but it significantly reduces the size of the search space. Formally, we have the following result.

Proposition 5.3. *Let $G = (V, E)$ be a directed graph with n vertices. If a permutation $v_1, \ldots, v_n$ of V is a Hamiltonian cycle of G and $v_1 = 1$, then $M = D_{hc}(G) \cup \{hc\_perm(i, v_i) \colon i = 1, \ldots, n\}$ is a model of the data-program pair $(D_{hc}(G), P_{hc})$. Conversely, if M is a model of the data-program pair $(D_{hc}(G), P_{hc})$, then $M = D_{hc}(G) \cup \{hc\_perm(i, v_i) \colon i = 1, \ldots, n\}$, for some permutation $v_1, \ldots, v_n$ of V forming a Hamiltonian cycle of G and such that $v_1 = 1$.*

We will next 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. The representation of input data specifies the set of row and column indices:

$$D_{nq}(n) = \{index(i) \colon i = 1, \ldots, n\}.$$

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: $q(R, C) \rightarrow \mathit{index}(R)$
nQ2: $q(R, C) \rightarrow \mathit{index}(C)$
nQ3: $\mathit{index}(R) \rightarrow q(R, \_)$
nQ4: $q(R, C1) \wedge q(R, C2) \rightarrow (C1 = C2)$
nQ5: $q(R1, C) \wedge q(R2, C) \rightarrow (R1 = R2)$
nQ6: $q(R, C) \wedge q(R + I, C + I) \rightarrow \bot$
nQ7: $q(R, C) \wedge q(R + I, C - I) \rightarrow \bot$

The first two rules 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 rules (nQ3) - (nQ5) together enforce the constraint that each row and each column contains exactly one queen. Finally, the last two rules guarantee that no two queens are placed on the same diagonal. We note that, as in the case of the hamiltonian cycle problem, there are symmetries in the n-queens problem that could be exploited to reduce the size of the search space. The structure of these symmetries is slightly more complex and, for simplicity of the presentation, we decided not to model them. As in the other cases, we can formally state and prove the correctness of our encoding. The proof is again quite similar to that of Proposition 5.1 and so we we omit it.

PROPOSITION 5.4. *Let n be a positive integer. If a set $\{(r_i, c_i): i = 1, 2, \ldots, n\}$ of n points on an $n \times n$ board is a solution to the n-queens problem then the set $M = D_{nq}(n) \cup \{q(r_i, c_i): i = 1, 2, \ldots, n\}$ is a model of the data-program pair $(D_{nq}(n), P_{nq})$. Conversely, if M is a model of the data-program pair $(D_{nq}(n), P_{nq})$ then $M = D_{nq}(n) \cup \{q(r_i, c_i): i = 1, 2, \ldots, n\}$, for some solution $\{(r_i, c_i): i = 1, 2, \ldots, n\}$ to the n-queens problem.*

As in the case of the graph-coloring problem, the correspondence between models and valid arrangements of queens on the board is a bijection.

For the last example in this section, we consider computing the **transitive closure** of a finite directed graph $G = (V, E)$, where V is a set of vertices and E is a set of directed edges (we will assume that G has no loops). We recall that the transitive closure of the graph $G = (V, E)$ is the directed graph (V, E') such that an edge (x, y) belongs to E' if and only if there is in G a directed path from x to y of length at least 1.

We will now describe the representation of data instances and give a *PS* program solving the transitive closure problem. The data instance consists of a specification of an input graph (V, E) and of a collection of integers, $\{1, 2, \ldots, |V|\}$ that will allow us to count edges in the paths. Thus, we set

$$D_{tc}(G) = \{\mathit{vtx}(v): v \in V\} \cup \{\mathit{edge}(v, w): \{v, w\} \in E\} \cup \{\mathit{index}(i): 1 \leq i \leq n\},$$

where $n = |V| - 1$.

Next, we construct a program, P_{tc}, encoding the constraints of the problem. Our encoding uses an auxiliary 4-ary relation symbol *path*. The intended meaning of $\mathit{path}(X, Y, Z, I)$ is that it is true precisely when there is a directed path from X to Y such that Z is the immediate predecessor of Y on the path and the path length is at most I. In P_{tc} we define the relation *path* and use it to specify the relation *tc* that represents the transitive closure of the input graph.

TC1: $path(X,Y,Z,I) \rightarrow vtx(X)$
TC2: $path(X,Y,Z,I) \rightarrow vtx(Y)$
TC3: $path(X,Y,Z,I) \rightarrow vtx(Z)$
TC4: $path(X,Y,Z,I) \rightarrow index(I)$
TC5: $tc(X,Y) \rightarrow vtx(X)$
TC6: $tc(X,Y) \rightarrow vtx(Y)$
TC7: $path(X,Y,X,1) \rightarrow edge(X,Y)$
TC8: $edge(X,Y) \rightarrow path(X,Y,X,1)$
TC9: $path(X,Y,Z,1) \rightarrow X = Z$
TC10: $path(X,Y,Z,I+1) \rightarrow path(X,Z,\_,I)$
TC11: $path(X,Y,Z,I+1) \rightarrow edge(Z,Y)$
TC12: $index(I+1) \wedge path(X,Z,W,I) \wedge edge(Z,Y) \rightarrow path(X,Y,Z,I+1)|X = Y$
TC13: $tc(X,Y) \rightarrow path(X,Y,\_,\_)$
TC14: $path(X,Y,Z,I) \rightarrow tc(X,Y)$
TC15: $path(X,X,Z,I) \rightarrow \perp$.

The first four rules enforce that if an atom $path(x,y,z,i)$ is in a model of the data-program pair $(D_{tc}(G), P_{tc})$ then x, y and z are vertices ($vtx(x)$, $vtx(y)$ and $vtx(z)$ hold) and i is an index ($index(i)$ holds). The effect of the next two rules is similar but they concern the relation symbol tc. The rules (TC7) - (TC9) enforce conditions that atoms $path(x,y,z,1)$ must satisfy to be in a model. The rules (TC10) - (TC12) enforce recursive conditions that atoms $path(x,y,z,i)$, $i \geq 2$, must satisfy in order to be in the model. The rules (TC13) - (TC14) define the relation symbol tc in terms of the relation *path*. Finally, the rule (TC15) guarantees that the relation tc has no loops by disallowing paths that start and end in the same vertex.

The following result can now be proved by an easy induction.

PROPOSITION 5.5. *Let G be a directed graph. Then, the data-program pair $(D_{tc}(G), P_{tc})$ has a unique model that consists of (1) all atoms in $D_{tc}(G)$, (2) all atoms $path(x,y,z,i)$, where $x \neq y$, such that there is a path in G from x to y of length i and with z being the last but one vertex on this path, and (3) of all atoms $tc(x,y)$, where $x \neq y$ such that there is a directed path of positive length from x to y in G.*

We have chosen to discuss in detail the question of the transitive closure since it is well known that this property is not definable in first-order logic [Abiteboul and Vianu 1991]. We can define it in our logic *PS* because our notion of definability is different: data-program pairs define concepts as special Herbrand models. A more detailed discussion of these issues follows in the next section.

6. EXPRESSIVE POWER OF THE LOGIC *PS*

Our discussion in the previous sections demonstrated the use of the logic *PS* as a tool to represent computational problems. In this section, we will study the expressive power of the logic *PS*. It is common to measure the expressive power of formal systems in terms of complexity classes, that is classes of decision problems. We develop the logic PS as a tool to specify search problems or, equivalently, partial multivalued functions. Thus, following [Selman 1994], we will measure the expressive power of the logic PS in terms of classes of search problems. Specifically, we will show that the class of computational problems that can be represented by means of finite PS programs is the class NPMV, that is, the class of all search problems recognized by polynomial-time nondeterministic Turing transducers [Selman 1994].

We first recall some database terminology [Ullman 1988]. Let Dom be a fixed infinite set (for instance, the set of all natural numbers). A *relational schema* over a domain Dom is a nonempty sequence $R = (r_1, \ldots, r_k)$ of relation symbols. Each relation symbol r_i comes with integer arity $a_i > 0$. An *instance* of a relation schema R is a *nonempty* and *finite* set of ground atoms, each of the form $r_i(u_1, \ldots, u_{a_i})$, where $1 \leq i \leq k$ and $u_1, \ldots, u_{a_i} \in Dom$. By $\mathcal{I}(R)$ we denote the set of all instances of a relational schema R. Since Dom is fixed, from now on we will not explicitly mention it. We also emphasize that, unlike in standard presentations, we require that instances of a relation schema be nonempty.

Relational schemas provide a framework for a precise definition to a class of computational problems known as *search problems*. Let R and S be two disjoint relational schemas. A *search problem* (over relational schemas R and S) is a recursive relation $\Pi \subseteq \mathcal{I}(R) \times \mathcal{I}(S)$. The set $\mathcal{I}(R)$ is the set of *instances* of Π. Given an instance $I \in \mathcal{I}(R)$, the set $\{J \in \mathcal{I}(S) : (I, J) \in \Pi\}$ is the set of *solutions* to Π for the instance I. In [Selman 1994], search problems are referred to as partial multivalued functions. It is also worth noting that decision problems can be cast as (special) search problems.

It is clear that the graph problems and the n-queens problem considered earlier in the paper are examples of search problems. More generally, all constraint satisfaction problems over discrete domains, including such basic AI problems as planning, scheduling and product configuration, can be cast as search problems.

A *search language* or *language*, for short, is a set L of expressions and a function μ such that for every expression $e \in L$, $\mu(e)$ is a search problem. We call μ the *interpretation function* for L. By the *expressive power* of a language L we mean the class of search problems defined by expressions from L: $\{\mu(e) \colon e \in L\}$.

We note that the concept of a search problem extends that of a database *query* [Vardi 1982], which is defined as a partial recursive *function* from $\mathcal{I}(R)$ to $\mathcal{I}(S)$. Consequently, fragments of search languages consisting of those expressions that define partial functions are, in particular, database query languages. In fact, one can regard a search problem as a *second-order query* — a mapping from the set of instances of some relational schema R into the *power set* of the set of instances of another (disjoint) relational schema S. Pushing the analogy further, a search language can be viewed as a second-order database query language. An expression in such a language defines, given an instance of a relational schema R, a *collection* of instances of a relation schema S, rather than a single instance.

Our goal in this section is to show that the logic PS also gives rise to a search language and to establish the expressive power of that language. An *expression* is a triple (P, R, S), where P is a PS program, and R and S are disjoint nonempty sets of relation symbols in P. We will show that (P, R, S) can be viewed as a specification of a search problem over relational schemas R and S. Namely, let

$D \in \mathcal{I}(R)$. For every set $M \subseteq HB(D \cup P)$, by $M[S]$ we denote the set of all those atoms in M that are built by means of relation symbols from S. We define the interpretation function μ as follows:

$$\mu(P, R, S) = \{(D, F) \colon D \in \mathcal{I}(R) \text{ and } F = M[S], \quad \text{where } M \in \mathit{Mod}(D, P)\}.$$

It is clear that $\mu(P, R, S) \subseteq \mathcal{I}(R) \times \mathcal{I}(S)$. Consequently, the set of PS expressions together with the function μ is a search language.

In a similar way we can view as a search language the language of DATALOG$^{\neg}$ (logic programming without function symbols) with the semantics of Herbrand models, supported models [Clark 1978; Apt 1990] or stable models [Gelfond and Lifschitz 1988]. Since the expressive power of DATALOG$^{\neg}$ with the supported-model semantics will play a role in our considerations, we will recall relevant notions and results[2].

Let $\mathcal{L}$ be a language of predicate logic. A DATALOG$^{\neg}$ *clause* is an expression r of the form

$$r = \quad p(X) \leftarrow q_1(X_1), \ldots, q_m(X_m), \neg q_{m+1}(X_{m+1}), \ldots, \neg q_{m+n}(X_{m+n}),$$

where $p, q_1, \ldots, q_{m+n}$ are relation symbols and $X, X_1, \ldots, X_{m+n}$ are tuples of constant and variable symbols with arities matching the arities of the corresponding relation symbols. We call the atom $p(X)$ the *head* of the clause r and denote it by $h(r)$. If a clause has empty body, we represent it by its head (thus, atoms can be regarded as clauses). For a clause r we also set

$$B(r) = q_1(X_1) \wedge \ldots \wedge q_m(X_m) \wedge \neg q_{m+1}(X_{m+1}) \wedge \ldots \wedge \neg q_n(X_n).$$

A DATALOG$^{\neg}$ *program* is a collection of DATALOG$^{\neg}$ clauses. Let P be a DATALOG$^{\neg}$ program. As usual, we call relation symbols that appear in the heads of clauses in P *intentional*. We refer to all other relation symbols in P as *extensional*. We denote the sets of intentional and extensional relation symbols of a DATALOG$^{\neg}$ program P by $I(P)$ and $E(P)$, respectively. Next, for a relation symbol p that appears in P, we denote by $\mathit{Def}(p)$ the set of all clauses in P whose head is of the form $p(t)$, for some tuple t of constant and variable symbols. In other words, $\mathit{Def}(p)$ consists of all clauses that *define* p.

In the paper we restrict our attention to DATALOG$^{\neg}$ programs of special form, called *I/O* programs, providing a clear separation of data facts (ground atoms representing data) from clauses (definitions of intentional relation symbols). To this end, we define first a class of *pure* programs. We say that a DATALOG$^{\neg}$ program P is *pure* if

(1) for every relation symbol $p \in I(P)$, all clauses in $\mathit{Def}(p)$ have the same head of the form $p(X)$, where X is a tuple of distinct variables

(2) P contains no occurrences of constant symbols

(3) $E(P) \neq \emptyset$.

[2]All concepts related to DATALOG$^{\neg}$ that we mention here can be defined in a more general setting of logic programming languages that include function symbols. For an in-depth discussion of logic programming, we refer the reader to [Apt 1990].

Pure programs are, in particular, in the so-called normal form as they satisfy condition (1) [Apt 1990]. An *I/O program* is a DATALOG$^{\neg}$ program of the form $D \cup P$, where P is a pure program and $D \in \mathcal{I}(E(P))$ (that is, D is a nonempty and finite set of ground atoms built of relation symbols in $E(P)$). To simplify the discussion, we define supported models for I/O programs only. It does not cause any loss of generality. Indeed, one can show that for every logic program containing at least one constant symbol there is an I/O program with the same intentional relation symbols and such that supported models of both programs, when restricted to intentional ground atoms, coincide (that is, under the semantics of supported models, both programs define the same relations).

Let P be a pure program and let $D \in \mathcal{I}(E(P))$. For a predicate p from $I(P)$, we define its *(Clark's) completion* $cc(p)$ as

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

where X is a tuple of variables and Y_r is the tuple of distinct variables occurring in the body of r but not in the head of r (we exploit the normal form of P here) [Clark 1978]. We define the (Clark's) completion of P, $CC(P)$, by setting

$$CC(P) = \{cc(p)\colon p \in Pr\}.$$

Finally, we define a set of ground atoms $M \subseteq HB(D \cup P)$ to be a *supported model* of an I/O program $D \cup P$ if it is a Herbrand model of $cl(D) \cup CC(P)$, where $cl(D)$ is defined as in Section 2. We denote by $Supp(D \cup P)$ the collection of all supported models of $D \cup P$.

Let P be a pure program and let $S \subseteq I(P)$. We define

$$\begin{aligned}\nu(P,S) = \{(D,F)\colon\ & D \in \mathcal{I}(E(P)),\ \text{and}\ F = M[S],\\ & \text{where}\ D \neq \emptyset\ \text{and}\ M \in Supp(D \cup P)\}.\end{aligned}$$

Since $E(P)$, $I(P)$ and S can be regarded as relational schemas, $\nu(P,S)$ is a search problem. Thus, the set of expressions (P,S), where P is a pure program and S is a subset of $I(P)$, together with the function ν form a search language.

The expressive power of this language is known. A search problem Π over relational schemas R and S is in the class *NPMV* if there is a nondeterministic polynomial-time *transducer* (a Turing machine with the designated read-only input tape and the write-only output tape) computing Π [Selman 1994]. That is, for every instance I of the schema R (input instance of Π) placed on the input tape, the set of strings left on the output tape by accepting computations for I is precisely the set $\{J \in \mathcal{I}(S)\colon (I,J) \in \Pi\}$, that is, the set of solutions to Π for the input I.

The class NPMV is precisely the class of search problems captured by finite DATALOG$^{\neg}$ programs with the supported-model semantics.

Theorem 6.1 [Marek and Remmel 2003]. *For every finite pure program P and every $S \subseteq I(P)$, $\nu(P,S)$ is a search problem in the class NPMV. Conversely, for every problem Π in the class NPMV there is a pure program P and a set $S \subseteq I(P)$ such that $\nu(P,S) = \Pi$.*

We will now show that the expressive powers of *PS* and of DATALOG$^{\neg}$ with supported model semantics are the same. Namely, we will prove the following result.

THEOREM 6.2. *For every finite pure program P and every set $S \subseteq I(P)$, there is a finite PS program P' such that $E(P) \cup I(P)$ are among the relation symbols appearing in P' and $\nu(P,S) = \mu(P', E(P), S)$. Conversely, for every finite PS program P' and every nonempty and disjoint sets R and S of relation symbols appearing in P', there is a finite pure program P such that $R = E(P)$, $S \subseteq I(P)$ and $\mu(P', R, S) = \nu(P, S)$.*

Proof: Let P be a pure program. We will consider the completion $CC(P)$ of P and construct its equivalent representation in terms of PS rules (we recall that PS rules are just special formulas from the language of predicate logic).

We build this representation of $CC(P)$ as follows. Let p be a predicate symbol in $I(P)$. Let us assume that $p(X)$, where X is a tuple of distinct variables, is the common head of all clauses in $\mathit{Def}(p)$. Let us consider a clause $r \in \mathit{Def}(p)$, say

$$r = \quad p(X) \leftarrow q_1(X_1), \ldots, q_m(X_m), \neg q_{m+1}(X_{m+1}), \ldots, \neg q_n(X_n),$$

and let Y_r be a tuple of distinct variables that appear in the body of r but not in its head. We introduce a new predicate symbol d_r, of the arity $|X|+|Y_r|$ and define the following PS rules

$$\begin{aligned}
\psi_i(r) = &\quad d_r(X, Y_r) \rightarrow q_i(X_i), \;\; i = 1, \ldots, m\\
\psi_i(r) = &\quad d_r(X, Y_r) \wedge q_i(X_i) \rightarrow \bot, \;\; i = m+1, \ldots, n\\
\psi_0(r) = &\quad q_1(X_1) \wedge \ldots \wedge q_m(X_m) \rightarrow d_r(X, Y_r) \vee q_{m+1}(X_{m+1}) \vee \ldots \vee q_n(X_n).
\end{aligned}$$

We define $\Psi(r) = \{\psi_0(r), \psi_1(r), \ldots, \psi_n(r)\}$. It is clear that $\Psi(r)$ entails (in the first-order logic) the universal sentence $d_r(X, Y_r) \leftrightarrow B(r)$ (intuitively, $\Psi(r)$ specifies $d_r(X, Y_r)$ so that it can be regarded as an abbreviation for $B(r)$).

We will now use atoms $d_r(X, Y_r)$ to define PS rules that form an equivalent representation to the formula $cc(p)$. Let us recall that

$$cc(p) = \quad p(X) \Leftrightarrow \bigvee \{\exists Y_r \;\; B(r) \colon r \in \mathit{Def}(p)\}.$$

Thus, we define the following PS rules:

$$\begin{aligned}
cc'_r(p) = &\quad d_r(X, Y_r) \rightarrow p(X), \;\; r \in \mathit{Def}(p)\\
cc'(p) = &\quad p(X) \rightarrow \bigvee \{\exists Y_r d_r(X, Y_r) \colon r \in \mathit{Def}(p)\}.
\end{aligned}$$

It is clear (by first-order logic tautologies) that

$$\Phi(p) = \{\Psi(r) \colon r \in \mathit{Def}(p)\} \cup \{cc'_r(p) \colon r \in \mathit{Def}(p)\} \cup \{cc'(p)\}$$

and $cc(p)$ have the same first-order models (modulo new relation symbols d_r).

Let us define $P' = \bigcup \{\Phi(p) \colon p \in I(P)\}$. Clearly, P' is a PS program and every relation symbol in $E(P) \cup I(P)$ is in P'. Moreover, by the comment made above, for every instance D of the schema $E(P)$, $cl(D) \cup P$ and $cl(D) \cup P'$ have the same models and, in particular, the same Herbrand models (again modulo new relation symbols). Thus, $\nu(P,S) = \mu(P, E(P), S)$.

We will now prove the second part of the assertion. Let P' be a PS program and let R and S be nonempty and disjoint sets of relation symbols appearing in P'. By Theorem 6.1, it is enough to show that the search problem $\mu(P', R, S)$ belongs to the class NPMV. This is, however, straightforward. A nondeterministic Turing transducer M for solving $\mu(P', R, S)$ can be described as follows:

(1) An instance $D \in \mathcal{I}(R)$ forms the initial content of the input tape.
(2) M grounds (in a deterministic way) the data-program pair (D, P') (the design of M depends on P' in a way that supports that task). Since the data complexity of grounding is in the class P, the task can be accomplished in polynomial time with respect to the size of D (measured as the total number of symbols in D).
(3) M generates in a nondeterministic fashion (using its guessing module) a subset of the Herbrand base. This task involves a number of guesses that is not greater than $|HB(D \cup P')|$, again a polynomial in the size of D.
(4) Next, M checks (deterministically) that the subset that was guessed is a model of the ground theory. This task can be accomplished in time that is polynomial in the size of the grounding of the data-program pair (D, P) which, as we already pointed out, is polynomial in the size of D.
(5) If the subset that was guessed is not a model, M moves to halting state NO (non-accepting). Otherwise, M scans the model and writes onto the the output tape all atoms in the model that are built of relation symbols in S. Upon completion of the task, M moves to halting state YES (accepting).

It is clear that tape contents for accepting computations are precisely projections of models of the data-program pair (D, P) onto S. That is, M computes the search problem Π nondeterministically in polynomial time. It follows that $\mu(P', R, S)$ is in the class NPMV. □

Corollary 6.3. *A search problem* Π *is in the class NPMV if and only if there is a finite PS program* P *and nonempty disjoint sets* R *and* S *of relation symbols appearing in* P *such that* $\Pi = \mu(P, R, S)$.

Decision problems can be viewed as special search problems. Thus, in particular, every decision problem in the class NPMV (these problems form the class NP) can be expressed by means of a finite *PS* program (and two nonempty disjoint sets of relation symbols appearing in it). This observation is a counterpart to a result by Schlipf concerning DATALOG$^{\neg}$ [Schlipf 1995].

Corollary 6.4. *A decision problem* Π *is in the class NP if and only if there is a finite PS program* P *and nonempty disjoint sets* R *and* S *of relation symbols appearing in* P *such that* $\Pi = \mu(P, R, S)$.

We conclude this section by noting that the existential fragment of the second-order logic (the logic ESO) [Fagin 1974; Gottlob et al. 2000] is also a search language and it is closely related to our approach. The only difference is that of a perspective — we work within the first-order framework, while the logic ESO is concerned with theories in the second-order logic. [Fagin 1974] established the expressive power of the logic ESO in terms of classes of *decision* problems and showed it to be that of the class NP. One could generalize that result to the setting, where the expressive power is measured by classes of search problems, and derive the main result of this section through the connection to the logic ESO. We provided a different proof to exhibit an explicit *uniform* translation of logic programs with the *supported-model* semantics to our formalism, which generalizes similar translations at the propositional level that are behind the system *cmodels* [Babovich and Lifschitz 2002]). That encoding is important, as it shows that at least in some cases (for instance, when supported and stable models coincide), the negation as failure available in logic programming can be uniformly compiled away (and exhibits a way to do it).

7. EXTENSIONS OF THE LOGIC *PS*

From the programming point of view, the logic *PS* provides a limited repertoire of modeling means: constraints must be represented as rules (essentially, standard clauses of predicate logic). We will now present ways to enhance the effectiveness of logic *PS* as a programming formalism. Namely, we will introduce extensions to the basic formalism of the logic *PS* to provide direct support representations of some common "higher-level" constraints. We denote this extended logic *PS* by *PS*+.

7.1 Adding cardinality atoms

When considering the *PS* 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* contained direct means to capture constraints such as: "exactly one element is selected" or "at most k elements are selected".

We already noted earlier that extensions of the language of DATALOG$^\neg$ with explicit constructs to model such constraints and the corresponding modifications in the algorithms to compute stable models resulted in significant performance improvements. These gains can be attributed to the fact that programs in the extended language are usually much more concise, their ground versions often use fewer ground atoms and have smaller sizes. Thus, the search space of candidate models typically is also smaller.

It is natural to expect that similar gains are also possible in the case of our formalism. With this motivation in mind, we extend the language of the logic *PS* by cardinality atoms. We we first consider a propositional language specified by a set of atoms At. By a *propositional cardinality atom* (propositional c-atom, for short), we mean any expression of the form $m\{p_1,\ldots,p_k\}n$ (one of m and n, but not both, may be missing), where m and n are non-negative integers and $p_1,\ldots,p_k$ are atoms from At. The notion of a rule generalizes in an obvious way to the case when propositional c-atoms are present in the language. Namely, a *c-rule* is 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) atoms or c-atoms.

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

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

If m is missing, we only require that $|M \cap \{p_1,\ldots,p_k\}| \leq n$. Similarly, when n is missing, we only require that $m \leq |M \cap \{p_1,\ldots,p_k\}|$. A set of atoms M *satisfies* a c-rule C if M satisfies at least one atom B_j or does not satisfy at least one atom A_i.

For example, if $At = \{a,b,c,d\}$, then the expression

$$a \rightarrow 2\{a,c,d\} \vee d$$

is a rule. The set $M = \{a,c\}$ is its model while $M' = \{a,b\}$ is not.

To generalize the idea of a cardinality atom to the language of predicate calculus, we need syntax that will facilitate concise representations of sets. We define a cardinality atom or *c-atom*, for short, to be any expression

$$l\{S_1; S_2; \ldots; S_k\}u,$$

where l and u are terms (constants or variables) and $S_1, S_2, \ldots, S_k$ are *set definitions*. Intuitively, the meaning of a c-atom $l\{S_1; S_2; \ldots; S_k\}u$ is that there are at least l and no more than u atoms that are true in the union of the sets specified by the set definitions $S_1, \ldots, S_k$. We will now make this intuition precise. Our definitions are similar to those proposed in [Simons et al. 2002] in the context of SLP.

A *set definition* is an expression of the form $p(t)[L] : d_1(s_1) \land \ldots \land d_m(s_m)$, where p is a program relation symbol, L is a list of variables, d_i, $1 \leq i \leq m$, are data or predefined relation symbols, and t, s_i, $1 \leq i \leq m$, are tuples of terms. We require that the underscore symbol does not occur in the term tuples s_i, $1 \leq i \leq m$, but it may appear in the term tuple t. We call the conjunction $d_1(s_1) \land \ldots \land d_m(s_m)$ the *condition* of the set definition. We note that it is possible that L is empty (in such case we omit it from the notation) and that $m = 0$ (in such case we may omit the symbol ':'). We also note that the restriction to data and built-in predicates only in the condition of a set definition could be relaxed. We adopt it as it simplifies the definition of the semantics of c-atoms.

Variables that appear in the list L of a set definition S are *bound* (in S). Variables appearing in S that are not bound are *free*.

We will now specify the meaning of a set definition. We will start with an example. The expression $S = p(X,Y)[Y] : d(X,Y)$ is a set definition (assuming that d is a data predicate). The variable X is free in it and Y is bound. Intuitively, designating X as free and Y as bound means that for every constant x, S stands for the set of all atoms $p(x,y)$, where y is a constant such that $d(x,y)$ holds.

We will now make this intuition precise. Let $S = p(t)[L] : d_1(s_1) \land \ldots \land d_m(s_m)$ be a set definition appearing in a c-rule (we give a definition below) of a data-program pair $T = (D,P)$. If there are underscore symbols in t, we replace each of them by a new distinct variable and include all these new variables in the list L. From now on, we assume that the underscore symbols have been removed from t. Let ϑ be a ground substitution whose domain contains all free variables in S and does not contain any variables that are bound in S (we note that the sets of free and bound variables are disjoint). By $S\vartheta$ we denote the set of atoms of the form $p(t\vartheta\vartheta')$, where ϑ' is a ground substitution with the domain consisting of all variables that are bound in S such that for every i, $1 \leq i \leq m$, $d_i(s_i\vartheta\vartheta')$ holds (we recall that data and predefined relation symbols are fully specified by a data-program pair and this latter condition can be verified efficiently). We note that if L is empty, $S\vartheta$ consists of a single ground atom $p(t\vartheta)$ (none of the variables appearing in $p(t)$ is bound).

We can now make the definition of a c-atom precise. It is an expression of the form $l\{S_1; S_2; \ldots; S_k\}u$, where l and u are terms (constants or variables), $S_1, S_2, \ldots, S_k$ are set definitions, and l and u are not local in S_i, $1 \leq i \leq k$.

A c-rule is an expression of the form

$$C = \quad A_1 \land \ldots \land A_s \rightarrow B_1 \lor \ldots \lor B_t,$$

where all A_i and B_i are atoms, e-atoms or c-atoms. As before we do not allow e-atoms to appear in the antecedent of the rule. To specify the meaning of a c-rule C, we ground it and replace it with a set of propositional rules (with propositional cardinality atoms). Let us now consider a c-atom $A = l\{S_1; \ldots ; S_k\}u$ appearing in C. We start by renaming all bound variables by new unique names different from any other variable name in the rule (the renaming does not change the meaning of any of the set definitions in A). In this way, the sets of bound and free variables in C are disjoint. Let ϑ be a ground substitution whose domain contains all free variables and none of the bound ones. A. We define $A\vartheta$ as follows:

(1) $A\vartheta = \perp$, if $l\vartheta$ or $u\vartheta$ are not integers appearing as constants in T

(2) $A\vartheta = l\vartheta\{S_1\vartheta \cup \ldots \cup S_k\vartheta\}u\vartheta$, otherwise. In this case, $l\vartheta$ and $u\vartheta$ are integer constants appearing in T, and $S_1\vartheta \cup \ldots \cup S_k\vartheta$ is the set of ground atoms.

It is clear $A\vartheta$ is a propositional c-atom. Applying ϑ to all atoms in C produces a c-rule $C\vartheta$. This rule is, clearly, a propositional c-rule. Let $T = (D, P)$ be a data-program pair. We define $gr(T)$ to consist of all propositional c-rules of the form $C\vartheta$, where C is a c-rule in P and ϑ is a ground substitution that contain in its domain all free variables in C and none of C's bound variables. We define a set M of ground atoms to be a *model* of T if it is a model of $gr(T)$.

We will now illustrate these definitions with an example. Let (D, P) be a data-program pair (in the language extended with c-atoms). Let us assume that

$$D = \{d_1(1), d_1(2), d_1(3), d_2(a), d_2(b)\}$$

and that

$$C = \quad d_1(X) \rightarrow X\{p(X,Y)[Y] \colon d_1(Y) \wedge Y \geq X; \quad q(X)[X] \colon d_2(X)\}$$

is a rule in P. We start by rewriting the rule as

$$C' = \quad d_1(X) \rightarrow X\{p(X,Y)[Y] \colon d_1(Y) \wedge Y \geq X; \quad q(Z)[Z] \colon d_2(Z)\}$$

s to ensure that the sets of bound and free variables are different. As a result, X is free and Y and Z are bound.

There are five different ground substitutions ϑ for X (that do not have Y nor Z in their domain). If ϑ replaces X with a or b, the head of the rule grounds to $\perp$ (by the first rule of the definition of $A\vartheta$). If ϑ replaces X with, say, 2, we have

$$(p(X,Y)[Y] \colon d_1(Y) \wedge Y \geq X)\vartheta = \{p(2,2), p(2,3)\},$$

as 2 and 3 are the only two substitutions for Y that satisfy the conditions $d_1(Y) \wedge Y \geq 2$ (we recall that ϑ assigns 2 to X). The situation is similar if X is replaced with 1 and 3.

Clearly, we also have

$$(q(Z)[Z] \colon d_2(Z))\vartheta = \{q(a), q(b)\}.$$

(In fact, that identity holds no matter what constant X is replaced with). Thus, $gr(D, P)$ consists of the following five rules:

$$d_1(1) \rightarrow 1\{p(1,1), p(1,2), p(1,3), q(a), q(b)\}$$
$$d_1(2) \rightarrow 2\{p(2,2), p(2,3), q(a), q(b)\}$$

$d_1(3) \rightarrow 3\{p(3,3), q(a), q(b)\}$
$d_1(a) \rightarrow \bot$
$d_1(b) \rightarrow \bot.$

The last two of these rules are immaterial ($d_1(a)$ and $d_1(b)$ are both interpreted as false). The third rule implies that the atoms $p(3,3)$, $q(a)$ and $q(b)$ must be true in every model of (D, P).

In the extended logic one can express formulas $\exists X_1, \ldots, X_k p(t)$, in which variables X_i are not necessarily distinct. We recall that in Section 2, when introducing the logic *PS*, we imposed the requirement that variables that are existentially quantified be pairwise distinct. However, we stated there that it can be lifted. We illustrate one way to do it in the logic *PS*+ with an example. Let us consider a formula $\exists X p(X,X)$. We express this formula in the logic *plcp* with the c-atom $1\{p(X,X)[X]\}$.

In the extended logic *PS*+ we can encode the vertex cover problem in a more straightforward and more concise way. Namely, there is no the need for integers to represent indices as sets are represented directly and not in terms of sequences! In this new representation $(D'_{vc}(G,k), P'_{vc})$, $D'_{vc}(G,k)$ is given by

$$D'_{vc}(G,k) = \{vtx(v)\colon v \in V\} \cup \{edge(v,w)\colon \{v,w\} \in E\} \cup \{size(k)\},$$

and P'_{vc} consists of the rules:

VC′1: $invc(X) \rightarrow vtx(X)$
VC′2: $size(K) \rightarrow \{invc(X)[X]\colon vtx(X)\}K$
VC′3: $edge(X,Y) \rightarrow invc(X) \vee invc(Y)$.

Atoms $invc(x)$ that are true in a model of the *PS* 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.

Cardinality atoms also yield alternative encodings to the graph-coloring and n-queens problems. In both cases, we use the same representation of input data and modify the program component only. In the case of the graph-coloring problem, a single rule, (C′3), directly stating that every vertex is assigned exactly one color, replaces two old rules (C3) and (C4).

C′1: $clrd(X,C) \rightarrow vtx(X)$
C′2: $clrd(X,C) \rightarrow color(C)$
C′3: $1\{clrd(X,C)[C]\colon color(C)\}1$
C′4: $edge(X,Y) \wedge clrd(X,C) \wedge clrd(Y,C) \rightarrow \bot$.

In the case of the n-queens problem the change is similar. The rules (nQ3) and (nQ4) are replaced with a single rule (nQ′3) and the rules (nQ5) and (nQ6) with a single rule (nQ′4).

nQ′1: $q(R,C) \rightarrow index(R)$
nQ′2: $q(R,C) \rightarrow index(C)$
nQ′3: $1\{q(R,C)[C]\colon index(C)\}1$
nQ′4: $1\{q(R,C)[R]\colon index(R)\}1$
nQ′5: $\{q(R+I-1,I)[I] : index(I)\}1$
nQ′6: $\{q(I,C+I-1)[I] : index(I)\}1$
nQ′7: $\{q(R-I+1,I)[I] : index(I)\}1$

nQ′8: $\{q(n-I+1, C+I-1)[I] : index(I)\}1$.

The rule (nQ′5) enforces the condition that the main ascending diagonal and all ascending diagonals above it contain at most one queen. The rule (nQ′6) enforces the same condition for the ascending main diagonal and all ascending diagonals below it. Finally, the rules (nQ′7) and (nQ′8) enforce the same condition for descending diagonals. In the original encoding we used only two rules to represent these conditions. We could use them here again. However, the four rules that we propose here, and that are possible thanks to the availability of c-atoms, result in significantly smaller ground theories. We address this issue in detail in Section 8.

7.2 Adding closure computation to logic $PS+$

In Section 5, we presented programs capturing the concepts of reachability in graphs and of transitive closure of binary relations. These representations are less elegant and, more importantly, less concise than representations possible in SLP. For instance, the transitive closure of a binary relation r can be computed by the following DATALOG program:

TC′1: $tc(X,Y) \leftarrow r(X,Y)$
TC′2: $tc(X,Y) \leftarrow r(X,Z), tc(Z,Y)$.

This encoding capitalizes on the minimality that is inherent in the stable-model semantics (in this case, the program, being a definite Horn program, has a unique *least* model). Moreover, the grounding of this program has size linear in the cardinality of the relation r.

Constraints involving reachability, transitive closure and other related concepts are quite common. In the problem of existence of a Hamiltonian cycle in a directed graph, we may first constrain candidate sets of edges to those that span collections of disjoint cycles covering all vertices in the graph (for instance, by imposing the restriction that in each vertex exactly one edge from the candidate set starts and exactly one edge from the candidate set ends). Clearly, such a candidate set is a Hamiltonian cycle if and only if it is connected. This requirement can be enforced by the constraint that all graph vertices be *reachable*, by edges in the candidate set, from some (arbitrary) vertex in the graph.

With this motivation in mind, we will now introduce yet another extension of the basic logic providing, in particular, means to express constraints involving reachability, connectivity, transitive closure and similar related concepts in a way they are used in SLP. To this end, we extend both the syntax and the semantics of the logic $PS+$.

As it is standard, by a *definite Horn* rule we mean a PS rule (a rule without cardinality atoms) whose consequent is a single *regular* atom (that is, not an e-atom as, we recall, e-atoms are formulas representing disjunctions of atoms). Definite Horn rules play a key role in this extension of the logic. The idea is to split the program component in a data-program pair into three parts. Intuitively, the first of them will describe initial constraints on the space of candidate solutions. The second, consisting of definite Horn rules, will "close" each candidate generated by the first part. The third component will provide additional constraints that have to be satisfied by the closure.

Formally, by an *extended program* we mean a triple (G, H, V) such that

(1) G and V are collections of (arbitrary) $PS+$ rules, called *generating* and *verifying* rules, respectively, and H is a collection of definite Horn rules

(2) No relation symbol appearing in the consequent of a rule in H appears in rules from G.

An extended data-program pair is a pair (D, P), where D is a set of ground atoms (data) and P is an extended program. When listing an extended program, we use the following convention. We write Horn rules as in logic programming, starting at the left with the head, followed by the (reversed) arrow $\leftarrow$ as the implication connective and, finally, followed by the conjunction of the atoms of the body. There is no need to explicitly distinguish between rules in G and V as the partition is implicitly defined by H. Namely, non-Horn rules involving relation symbols appearing in the consequents of Horn rules form the set V. All other non-Horn rules form the set G.

Let (D, P) be an extended data-program pair, where $P = (G, H, V)$. A set of ground atoms from the Herbrand base of (D, G) is a *model* of (D, P) if

(1) M is a model of (D, G)

(2) the *closure* of M under H, that is, the least Herbrand model of the definite Horn theory $M \cup H$, satisfies all ground instances of rules in V.

The first condition enforces that models of (D, P) satisfy all constraints specified by G. Thus, G can be regarded as a *generator* of the search space, as there are still additional constraints to be satisfied. The second condition eliminates all these models generated by G whose closure under H violates some of the constraints given by V. In other words, H computes the closure and V *verifies* whether the closure has all of the desired properties.

As an illustration of the way this extension of logic $PS+$ can be used we will provide a formal representation of the Hamiltonian-cycle problem, capturing intuitions described above. Let $G = (V, E)$ be a directed graph and let v_0 be an arbitrary vertex in V. To represent this data we set

$$D'_{hc}(G, v_0) = \{vtx(v) \colon v \in V\} \cup \{edge(v, w) \colon \{v, w\} \in E\} \cup \{start(v_0)\}.$$

Formally speaking, for the Hamiltonian-cycle problem, there is no need to include v_0 in the data set. We do it, as our encoding involves the notion of reachability, for which some arbitrary "starting" point is needed. The (extended) program part, P'_{hc}, consists of the following six rules.

HC$'$1: $hc\_edge(X, Y) \rightarrow edge(X, Y)$.
HC$'$2: $1\{hc\_edge(Y, X)[Y] \colon vtx(Y)\}1$.
HC$'$3: $1\{hc\_edge(X, Y)[Y] \colon vtx(Y)\}1$.
HC$'$4: $visit(Y) \leftarrow visit(X) \wedge hc\_edge(X, Y)$.
HC$'$5: $visit(Y) \leftarrow start(X)$.
HC$'$6: $visit(X)$.

We note the use of our notational convention. Clearly, the rules (HC$'$4) and (HC$'$5) form the Horn part (it is indicated by the way they are written). It follows that the rules (HC$'$1)-(HC$'$3) are generating and the rule (HC$'$6) is verifying. Intuitively, the rule (HC$'$1) guarantees that if an atom $hc\_edge(x, y)$ is true then, (x, y) is an edge (in other words, only edges of the graph can be chosen to form a

Hamiltonian cycle). Rule (HC$'$2) captures the constraint that for every vertex x there is exactly one selected edge that ends in x. Similarly, the rule (HC$'$3) captures the constraint that for every vertex x there is exactly one selected edge that starts in x. Thus, every model of the data-program pair consisting of $D_{hc}(G, v_0)$ and the rules (HC$'$1)-(HC$'$3) contains $D_{hc}(G, v_0)$ and a set of atoms $hc\_edge(x, y)$ that describe a particular selection of edges and that span in G disjoint cycles covering all its vertices. Rules (HC$'$4) and (HC$'$5) define the relation *visit* that describes all vertices in G reachable from v_0 by means of selected edges. Finally, the last rule verifies that all vertices are reached, that is, that selected edges form, in fact, a Hamiltonian cycle.

7.3 Expressive power of extended logics

We close this section with an observation on the expressive power of the logic $PS+$. Since it is a generalization of the logic PS, it can express all problems that are in the class NPMV. On the other hand, the search problem of computing models of a data-program pair (D, P), where P is a fixed $PS+$ program, is an NPMV problem (a simple modification of the proof of the second assertion of Theorem 6.2 demonstrates that). Thus, it follows that the expressive power of the logics $PS+$ does not extend beyond the class NPMV. In other words, the logic $PS+$ also captures the class NPMV.

8. COMPUTING WITH $PS+$ THEORIES

In the preceding sections, we focused on the use of the logic $PS+$ as a language for encoding (programming) search problems and established its expressive power. In order to use the logic $PS+$ as a computational problem solving tool we need algorithmic methods for processing data-program pairs and finding their models.

Let us recall that a set M of ground atoms is a model of a data-program pair (D, P) if and only if M is a model of the theory $gr(cl(D) \cup P))$. Thus, to compute models one could proceed in two steps: (1) compute $gr(cl(D) \cup P))$, and (2) find models of the ground theory. We refer to these steps as *grounding* and *solving*, respectively. This two-step approach is used successfully by all current implementations of SLP, including *smodels* and *dlv*. We will adhere to it, as well.

8.1 Grounding $PS+$ theories

It is easy to see that the data complexity of grounding is in the class P. That is, there is an algorithm that for every data-program pair (D, P) computes $gr(cl(D) \cup P))$ and, assuming that P is fixed, works in time that is polynomial in the size of D. For instance, a straightforward enumeration of all substitutions of appropriate arities (determined by the numbers of free variables in program rules) can be adapted to yield a polynomial-time algorithm for grounding (we provided additional relevant comments at the end of Subsection 2.3).

This straightforward approach can be improved. The size of grounding (although polynomial in the size of the data part) is often too large to be computed in practice. To address this potential problem, we note that to compute the models it is not necessary to use $gr(cl(D) \cup P))$. Any propositional theory, which has the same models as $gr(cl(D) \cup P))$ can be used instead. In this context, let us note that the truth values of all ground atoms appearing in $gr(cl(D) \cup P))$, which are built of data relation symbols can be computed efficiently by testing whether they are present in D. Similarly, we can effectively evaluate truth values of all ground atoms built of predefined relation symbols by, depending on the relation, checking whether two constants are identical, different or, in the case of integer constants, whether one is the sum, product, etc. of two other integer constants.

Thus, the theory $gr(cl(D) \cup P))$ can be simplified by taking into account the truth values of ground atoms built of data and predefined relation symbols. Let A be such a ground atom.

(1) If A appears in the consequent of the clause and is true, we eliminate this clause
(2) If A appears in the consequent of the clause and is false, we eliminate A from the consequent of the clause
(3) If A appears in the body of a clause and is true, we eliminate A from the body
(4) If A appears in the body of the clause and is false, we eliminate this clause.

These simplifications may reveal other atoms with forced truth values and the process continues, much in the spirit of the unit propagation used in satisfiability solvers. For instance, if we obtain a rule consisting of a single (regular) atom and this atom appears in the consequent of the rule, the atom must be true. If, on the other hand, this single atom appears in the body of the rule, it must be false. Furthermore, if a cardinality atom of the form $m\{p_1, \ldots, p_k\}n$, is forced to be true and the number of atoms p_i that have been already assigned value true is n, then all the unassigned atoms p_i must be false. In addition, if the number of atoms p_i that have been already assigned value false is $k - m$, then all unassigned atoms must be true. Similar propagation rules exist for the case when a c-atom is forced to be false.

We continue the process of simplifying the theory as long as new atoms with forced truth values are discovered. We call the theory that results when no more simplifications are possible the *ground core* of a data-program pair (D, P). We denote it by $core(D, P)$.

We have the following straightforward result (as in the other cases before, we do not explicitly mention ground predefined atoms when specifying models).

PROPOSITION 8.1. *Let (D, P) be a data-program pair. A set M of ground atoms is a model of (D, P) if and only if $M = D \cup T \cup M'$, where T is the set of atoms that are forced to be true and M' is a model of $core(D, P)$.*

Proposition 8.1 suggests that for the grounding step, it is enough to compute $core(D, P)$ rather than $gr(cl(D) \cup P))$. It is an important observation. The size of the theory $core(D, P)$, measured as the total number of symbol occurrences, is usually much smaller when compared to that of $gr(cl(D) \cup P))$. Following this general idea, we designed and implemented a program, *psgrnd* that, given a data-program pair (D, P), computes its ground equivalent $core(D, P)$.

8.2 Solving propositional $PS+$ theories

We will now focus on the second step — searching for models of a propositional $PS+$ theory. First, we will consider the class of theories that are obtained by grounding data-program pairs whose program component does not contain c-atoms. In this case, the ground core of a data-program pair is a collection of standard propositional clauses (written as implications). The program *psgrnd* provides an option that, in such case, produces the ground core of the input data-program pair in the DIMACS format. Consequently, most of the current implementations of propositional satisfiability (SAT) solvers can be used in the solving step to compute models. Thus, we can view the logic *PS* as a programming tool for modeling problems in terms of propositional constraints and regard *psgrnd* as a front-end facilitating the use of SAT solvers.

If c-atoms and Horn rules are present in a program, the theory after grounding and simplification is a propositional *PS*+ theory that contains, in general, (propositional) c-atoms and propositional definite Horn rules. Thus, SAT solvers are not directly applicable. One approach in such case is to represent c-atoms and closure rules by means of equivalent (standard) propositional theories. It is possible since, as we noted earlier, logics *PS* and *PS*+ have the same expressive power.

We argue, however, that a more promising approach to compute models of data-program pairs is to design solvers for propositional *PS*+ theories that are direct outcomes of the grounding process and, in general, may contain c-atoms. The reason is that using high-level constraints results in programs whose ground representations are often more concise then those obtained by corresponding programs that do not involve such constraints. We will illustrate this point using programs developed earlier in the paper.

We start with the vertex-cover problem. Let G be an input graph with n vertices and m edges, and let k be an integer specifying the cardinality of a vertex cover. In the case of the program consisting of rules (VC1) - (VC5), our grounding algorithm results in a propositional theory with kn atoms of the form $vc(i, x)$ and with $\Theta(kn^2)$ rules of total size (measured by the number of atom occurrences) also $\Theta(kn^2)$. On the other hand, grounding of the program consisting of rules (VC$'$1) - (VC$'$3) yields a theory with n atoms of the form $invc(x)$ and with $\Theta(m)$ rules of total size $\Theta(n + m)$. Thus, this latter encoding involves fewer atoms (if $k \geq 2$) and is asymptotically smaller in size, especially in view of the fact that k is often as large as a positive fraction of the number of vertices in the graph.

Next, we will consider the Hamiltonian-cycle problem. Our first encoding (rules (HC1) - (HC8)) grounds to a theory with n^2 atoms and with total size $\Theta(n^2 + n(n^2 - m))$. Our second encoding, involving definite Horn rules (rules (HC$'$1) - (HC$'$6)), grounds to a theory with $n^2 + n$ atoms and the total size of $\Theta(n^2)$. Thus, even though this theory uses slightly more atoms, it has significantly smaller total size. In fact, the total size is one order of magnitude smaller, unless a graph is "almost complete", that is, the number of "missing" edges is $o(n^2)$.

For the original encoding of the n-queens program, *psgrnd* produces a propositional theory of size $\Theta(n^3)$. On the other hand, it is easy to see that grounding of the the second encoding (the one involving cardinality atoms), has size $\Theta(n^2)$ — a gain of an order of magnitude.

In the case of the encodings for the graph-coloring problems, we also obtain more concise theories by grounding programs designed with the use of c-atoms. Indeed, the rule (C$'$3) grounds to a smaller theory than rules (C3) and (C4). The improvement is, in general, by a constant factor and so, it is not asymptotically better.

Since encodings involving c-atoms are usually smaller and define smaller search spaces, it is important to design solvers that can take direct advantage of these small representations. We developed a solver, *aspps* (short for "answer-set programming with propositional schemata"), that can directly handle c-atoms and closure rules. The *aspps* solver is an adaptation of the Davis-Putnam algorithm for computing models of propositional CNF theories. That is, it is a backtracking search algorithm whose two key components are *unit propagation* and *branching*. Unit propagation "propagates" through the theory truth values established so far. If there is a rule with all atoms in the antecedent assigned value true and all but one atom in the consequent assigned value false, then the remaining "unassigned" atom in the consequent must be true for the rule to hold. Similarly, if all atoms in the consequent of a rule are false and if all but one atom in the antecedent are true, the only "unassigned" atom in the antecedent must be false. In this way any partial assignment of truth values to atoms *forces* truth assignments on some additional atoms. When a contradiction is derived in this process (two contradictory truth assignments to an atom are forced), the program backtracks. When no more atoms can be forced and no contradiction has been discovered, the second component, *branching*, selects an *unassigned* atom to split the search space and continue.

A key difference between *aspps* and satisfiability algorithms is in how *branching* is implemented. In satisfiability solvers, in order to branch, we pick an atom, say a, and split the search space into *two* parts. In one of them we assume that the atom a is true. In the other one we assume that a is false. In either case, once the assignment of a truth value to a is made, it is propagated in the way we described above.

Propositional $PS+$ theories may, in general, contain c-atoms and *aspps* considers them too when selecting a way to branch. To explain the method that *aspps* uses, let us observe that the unit propagation may, in particular, force a truth value onto a c-atom appearing in the theory. That constrains possible truth assignments to unassigned atoms that appear in the c-atom.

For example, let us consider a propositional c-atom $C = 1\ \{a, b, c, d\}\ 1$ that has been forced to be true. Let us assume that d has already been assigned value false and that a, b and c are still unassigned. There are exactly three ways in which atoms a, b and c can be assigned truth values consistent with C being true:

$$\begin{array}{l} a = \mathbf{t}, b = \mathbf{f}, c = \mathbf{f} \\ a = \mathbf{f}, b = \mathbf{t}, c = \mathbf{f} \\ a = \mathbf{f}, b = \mathbf{f}, c = \mathbf{t}. \end{array}$$

It follows that if there are c-atoms whose truth values have been forced, we have additional ways to split the search. Namely, we can consider in turn each truth assignment to unassigned atoms appearing in C, which is consistent with the truth value of C. In our example, if C is true, we could split the search space into *three* subspaces by setting (1) $a = \mathbf{t}$, $b = \mathbf{f}$ and $c = \mathbf{f}$, (2) $a = \mathbf{f}$, $b = \mathbf{t}$ and $c = \mathbf{f}$, and (3) $a = \mathbf{f}$, $b = \mathbf{f}$ and $c = \mathbf{t}$, respectively.

The heuristics used in the branch selection are of vital importance for the overall performance of a solver. We will now describe the method that we implemented in *aspps*. The idea is to approximate the degree to which an atom (or c-atom) is constrained and to select one that is constrained the most. To this end, for each rule r we define its *weight* $W(r)$ by

$$W(r) = k^{m-l}.$$

In the formula, k is a constant, determined empirically, and in the current implementation set to 13, $m = \min(L, 10)$, where L is the maximum length of a rule in the (ground) theory, and l is the length of the rule r. It follows that shorter the rule, the greater its weight. We also note that the value of m for all theories with the maximum length of a rule at least 10 is the same (the choice of 10 as the threshold value was also determined through experiments).

The weight of a propositional atom is defined as the sum of the weights of all rules in which it appears (whether within a c-atom or not). The weight of a cardinality atom is the sum of the weights of unassigned propositional atoms which appear in it.

When looking for a way to branch, the *aspps* program considers all propositional atoms that have not been assigned a truth value yet. It also considers some of the c-atoms that have been forced true by earlier truth-value assignments. Let C be such a c-atom and let A be the set of unassigned atoms appearing in C. The atom C is considered as a basis for branching if the number of truth assignments to atoms in A consistent with the fact that C is true (that is, the number of branches that C defines) is less than or equal to $|A|$.

If there are c-atoms satisfying these conditions, *aspps* will select one with the maximum weight. It will then generate all truth assignments to unassigned atoms in C that are consistent with C being true, and will use these truth assignments to split the search space. Otherwise, *aspps* will branch on a propositional atom with the maximum weight and will split the search space into two parts. If a propositional *PS*+ theory contains Horn clauses, they play no role in the process of selection of the next atom for branching. They do, however, participate in the propagation step.

The source codes, information on the implementation details for programs *psgrnd* and *aspps* and on their use is available at `http://www.cs.uky.edu/ai/aspps/`.

9. EXPERIMENTAL RESULTS

Several data-program pairs that we presented in the paper (both with and without c-atoms and Horn rules) show that the logics *PS* and *PS*+ are effective as formalisms for modeling search problems. In this section we will provide evidence for the computational effectiveness of our programs *psgrnd* and *aspps*.

On one hand, we demonstrate that *psgrnd* facilitates the use of SAT and SAT(PB) solvers as processing engines for computing solutions to search problems. Indeed, whenever we use these solvers in our tests, they are run on theories generated by *psgrnd* from data-program pairs in the logic *PS* encoding search problems and their specific instances.

On the other hand, we demonstrate results showing good performance of our solver program *aspps*. In our tests we focus on problems that we use as examples throughout the paper: the vertex-cover, n-queens, graph-coloring, and Hamiltonian cycle problems. The first three of these problems have encodings that involve c-atoms but not Horn rules. Ground theories produced by *psgrnd* from the corresponding data-program pairs are (after some trivial syntactic modifications) in the format of pseudo-boolean constraints. For these problems, we compare *aspps* with *PBS*, a recently developed program for computing models of collections of pseudo-boolean constraints [Aloul et al. 2002].

For all four problems that we considered, the encodings as *PS*+ data-program pairs, which we presented in the paper, can be rewritten in a straightforward way into logic programs with cardinality constraints. That direct correspondence between *PS*+ data-program pairs we gave and logic programs allowed us to compare the performance of *aspps* with *smodels*, at present one of the most advanced implementations of answer-set programming paradigm based on logic programming with cardinality constraints.

Finally, all four problems have also encodings as data-program pairs in the logic *PS*. When grounded with *psgrnd*, these data-program pairs result in propositional theories. Consequently, we also compared the performance of *aspps* directly with SAT solvers.

The results show that *aspps* performs very well. However, we stress that our experiments did not aim at demonstrating superiority of one solver over another. That would require a much more comprehensive and careful experimental study. Our objective was rather simply to demonstrate the feasibility of the overall approach and to point to the effectiveness of *aspps* on several diverse problems.

In the experiments we used the following versions of these programs:

(1) *zchaff.2003.7.23* [Moskewicz et al. 2001b]
(2) *satz215.2* [Li 1997]
(3) *PBSv2.1_linux* [Aloul et al. 2003]
(4) *lparse-1.0.13* and *smodels-2.27* [Niemelä et al. 1997]
(5) *aspps.2003.06.04* and *psgrnd.2003.06.04*, for both *aspps* and as a front-end for satisfiability solvers [East and Truszczyński 2001].

All our experiments were performed on a Pentium IV 3.2 GHz machine with 1GB of memory and running GNU Linux version 2.4.22.

In the case of vertex cover, for each $n = 50, 60, 70, 80$ 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*, *PBS* and *smodels* on all the instances (G, k_G). encoded by data-program pairs (D'_{vc}, P'_{vc}) (we recall that P'_{vc} consists of rules (VC$'$1) - (VC$'$3)). In the case of *smodels*, we rewrote the rules in P'_{vc} into the syntax of logic program rules with cardinality constraints. We then grounded the data-program pairs using *psgrnd* (*lparse*, in the case of *smodels*) and used *aspps*, *PBS* and *smodels* on the resulting propositional theories and programs as solvers.

We also tested *satz*, *zchaff* and *aspps* on the instances (G, k_G), this time encoded by data-program pairs (D_{vc}, P_{vc}) (we recall that P_{vc} consists of rules (VC1) - (VC5), which do not contain c-atoms). We grounded these data-program pairs using *psgrnd* and used *zchaff*, *satz* and *aspps* as solvers on the resulting propositional theories.

In all cases, if a solver timed out for a set of instances of a given size, we did not test it on the set of instances of larger sizes. The results of the experiments are gathered in Figure 1. They show the average execution times. Overall, solvers that take advantage of c-atoms (first three lines in the table) perform much better than those that have to work with pure (no c-atoms) propositional theories. As we observed, the size of the encoding (VC′1) - (VC′3) is, in general, asymptotically smaller than that of (VC1) - (VC5). Thus, satisfiability solvers had to deal with much larger theories (hundreds of thousands of clauses for graphs with 80 vertices as opposed to a few hundred when c-atoms are used). Consequently, neither the SAT solvers nor *aspps* could handle *regular propositional* theories encoding instances involving even the smallest graphs we tried ($n = 50$). Among the solvers for theories (programs) with c-atoms, *aspps* performed several times faster than *smodels* (except for $n = 60$) and the difference seemed to grow with n. *PBS* performed worse. It could handle instances for $n = 50$ but timed out in other cases.

n	50	60	70	80
aspps	0.007	0.040	0.246	0.938
PBS	61.846	**	**	**
smodels	0.035	0.016	0.901	4.289
aspps (CNF encoding)	**	**	**	**
satz	**	**	**	**
zchaff	**	**	**	**

Fig. 1. Timing results (in seconds) for the vertex-cover problem. Average time for each set of 100 random generated graphs. ** timed out or halted after 10 minutes for a single instance.

For the n-queens problem, we ran a similar collection of tests. We tried *aspps*, *PBS* and *smodels* on theories and programs derived from the encoding (nQ′1) - (nQ′8), and we tested *satz*, *zchaff* and *aspps* on theories derived from the encoding (nQ1) - (nQ7). *Aspps*, performed exceptionally well and found solutions in all cases (in each case in a fraction of a second). *PBS* successfully dealt with the case of $n = 40$ but timed out in all other cases. *Smodels* timed out in all cases. Working with theories without c-atoms, *aspps* was also the best. *Satz* was able to find solutions for $n = 40$ and 50 but it timed out for $n = 60$, 70 and 80. *Zchaff* timed out in all the cases we tried. *Aspps*, when run on theories with c-atoms was several times faster than when run on theories without c-atoms. Thus, cardinality constraints again seem to play a crucial role. Grounding the program (nQ′1) - (nQ′8) for $n = 70$ leads to a theory with 4900 atoms and 416 rules. In contrast, theories obtained by grounding the program (nQ1) - (nQ7) are much larger. The theory in the case of $n = 70$ consists of 4900 atoms and 562030 rules. Figure 2 summarizes these results.

In the case of the graph colorability problem, as we observed in the previous section, c-atoms do not give rise to significant gains in the size of the ground theory. Given the amount of research devoted to satisfiability solvers and still relatively few efforts to develop fast solvers for logics involving cardinality constraints, it is not surprising that satisfiability solvers outperform here *aspps PBS* and *smodels*. Our results also show *satz* outperforming *zchaff*, which may be attributed to the fact that our test graphs were randomly generated and did not have any significant

# of queens	40	50	60	70	80
aspps	0.03	0.03	0.33	0.04	0.00
PBS	7.11	**	**	**	**
smodels	**	**	**	**	**
aspps (CNF encoding)	0.17	0.42	0.97	1.63	2.89
satz	3.63	26.60	**	**	**
zchaff	**	**	**	**	**

Fig. 2. Timing results (in seconds) for the n-queen problem. ** timed out or halted after 10 minutes for a single instance.

internal structure that could be capitalized on by *zchaff*. On theories without c-atoms, *aspps* is outperformed both by *satz* and by *zchaff*. This is an indication that search heuristics for *aspps* can be further improved.

As concerns theories with c-atoms, *aspps* and *smodels* show essentially the same performance and outperform *PBS*, which times out for $n = 200$ and $n = 300$. We summarize the relevant results in Figure 3. The graphs for the 3-colorability problem were generated randomly with vertex/edge ratios such that approximately 1/2 of the graphs were 3-colorable. For each value $n = 100$, 200 and 300, we generated a set of 1000 graphs. The values that we report are the average execution times.

n	100	200	300
aspps	0.000	0.116	4.558
PBS	0.059	**	**
smodels	0.037	0.263	5.761
aspps (CNF encoding)	0.020	2.530	**
satz	0.005	0.036	0.642
zchaff	0.002	0.064	4.646

Fig. 3. Timing results (in seconds) for the graph 3-coloring problem. ** timed out or halted after 10 minutes for a single instance.

Our last experiment concerned the problem of computing Hamiltonian cycles. In the tests, we considered graphs with $n = 20$, 40, 60, 80 and 100 vertices with the number m of edges chosen so that the likelihood of the existence of a Hamiltonian cycle be close to 0.5. For each set of parameters, we generated a family of 1000 instances and the corresponding collection of theories (programs) based on the encoding (HC$'$1) - (HC$'$6). The results show *aspps* performing much faster than *smodels*. We also considered CNF theories based on the encoding (HC1) - (HC8) and tested the performance of *aspps*, *PBS*, and the two SAT solvers *satz* and *zchaff* on these CNF encodings. All these programs could only handle the family of smallest instances ($n = 20$, $m = 75$) and timed out in all other cases. In the case 20/75, *zchaff* outperformed *satz*. As in other cases additional constraints (c-atoms and transitive closure computation, in this case) result in much smaller theories, which seems to make all the difference. The average execution times collected in the experiments are given in Figure 4.

V/E	20/75	40/180	60/300	80/425	100/550
aspps	0.000	0.000	0.001	0.001	0.002
smodels	0.023	0.083	0.204	0.401	0.665
aspps (CNF encoding)	10.569	**	**	**	**
PBS (CNF encoding)	2.879	**	**	**	**
satz	0.130	**	**	**	**
zchaff	0.071	**	**	**	**

Fig. 4. Timing results (in seconds) for the determining presence of a Hamilton cycle in a graph. ** timed out or halted after 10 minutes for a single instance.

These experiments validate the overall approach proposed in the paper. They show that *psgrnd* can be used effectively as a grounder not only with *aspps* but also with off-the-shelf SAT and SAT(PB) solvers. Further, the test results show that solvers taking advantage of complex constraints such as cardinality and closure constraints solve instances of search problems faster than SAT solvers can. To a large degree that is related to the fact that encodings of search problems involving cardinality and closure constraints are typically much smaller in size than standard propositional logic encodings. Importantly, the test results also show that among the three solvers capable of handling complex constraints, when run on theories considered in the paper, *aspps* outperformed both *smodels* and *PBS*. Direct comparisons of *aspps* with SAT solvers on CNF encodings are certainly inconclusive as in many cases all solvers timed out. However, for the n-queen problem, *aspps* outperformed *satz* and *zchaff*, while for the graph coloring problem and the smallest instances of the Hamiltonian cycle problem, SAT solvers outperformed *assps*. The results suggest that further improvements in the implementation of *aspps*, exploiting recent advances in SAT solver design, are possible.

10. DISCUSSION

The main contributions of our work are as follows. First, we developed two logics, *PS* and *PS*+, based on predicate calculus, that can serve as a high-level language to specify search problems. The task of computing models for theories in this logic reduces to the task of propositional or pseudo-boolean satisfiability and we developed a program, *psgrnd*, to support appropriate reductions. Consequently, our logics can serve as programming front-ends for SAT and SAT(PB) solvers.

Second, as we pointed out in Section 8, in some cases the results of *psgrnd* need to be further processed before SAT(PB) solvers can be used. Modifications that are required introduce new propositional variables, which may cause the performance of these solvers to degrade. Moreover, SAT(PB) solvers can be used only for programs without Horn rules. Therefore, we developed our own solver, called *aspps*, for computing models of theories produced by *psgrnd*. *Aspps* is better attuned to the syntax of the logic *PS*+ and, as our experiments demonstrate, it often outperforms other solvers that we considered.

Third, our results show that the answer-set programming paradigm extends beyond formalisms based on logic programming with answer-set semantics. The logic *PS*+, through, the notion of a data-program pair, is capable of modeling search problems in the class NPMV. Programs such as *aspps*, as well as SAT and SAT(PB) solvers, are competitive with *smodels* — a state-of-the art ASP system based on the syntax of logic programming and the semantics of stable models.

There are connections between our work and development of languages for describing and solving constraint satisfaction problems. Examples of such languages include AMPL [Fourer et al. 1993], OPL [van Hentenryck 1999] and ECL$^{\mathrm{i}}$PS^{e} [Wallace et al. 1997]. Our approach is different in several aspects. First, none of the languages we mentioned supports directly SAT or SAT(PB) solvers as processing back-ends. Intended processing engines for these languages are constraint solvers developed for problems with non-binary domains. One can specify in these languages instances of mixed integer programming problems (that generalizes the class of pseudo-boolean constraint problems), but additional processing is necessary before SAT and SAT(PB) solvers can be used. Our specifications in the logic *PS+* have explicit representations in formats required by those solvers. Second, constraint modeling languages depart in the significant way from the paradigm of declarativeness in that they mix data and problem specifications with the control of search. Consequently, problem specifications expressed in these languages are difficult to reason on. Our data-program pairs have a clear meaning in terms of *PS+* theories and their semantics. That provides a framework where one can reason about problem specification independently of problem instances or processing methods.

In the area of propositional satisfiability, the need for developing programming environments is well recognized (cf. Grand Challenge 4 in [Walsh 2003]). While there have been dramatic improvements in the performance of SAT and SAT(PB) solvers, the issue of modeling tools has received relatively little attention. As we mentioned already, one can use logic programming syntax to construct SLP representations of search problems and then use *cmodels* [Babovich and Lifschitz 2002] or *assat* [Lin and Zhao 2002] to translate them into SAT instances. Another possibility is to use the language NP-SPEC [Cadoli and Palipoli 1998], built as an extension of DATALOG, as also for NP-SPEC theories there are techniques to compile them into SAT instances [Cadoli and Schaerf 2001]. However, in each of these three cases, the syntax and the semantics of the modeling front-end is quite distinct from propositional logic. Other than the logic *PS+*, the only other language to describe SAT instances that is based on predicate calculus and that we are aware of is QPROP (quantified propositional logic) [Ginsberg and Parkes 2000; Parkes 1999]. However, QPROP is designed more as a data structure to represent large propositional theories in SAT solvers rather than as a specification language. At present, it lacks features to describe directly set definitions and c-atoms.

Recently, researchers proposed the logic ESO (the existential fragment of the second order logic) [Fagin 1974; Gottlob et al. 2000] as another possible candidate for a high-level language to specify search problems [Cadoli and Mancini 2002]. The emergence of the logic ESO in this context is not surprising as formulas of that logic specify relations (or partial multivalued functions). The approach proposed in [Cadoli and Mancini 2002] is similar to the one presented here and there is a straightforward way to compile ESO theories into SAT instances. The main difference is that we develop our logic as a first-order formalism with a simple semantics given by a class of Herbrand models, rather than as a second-order logic.

Our work points to several specific research directions related to the logic *PS*+. First, the class of aggregate constraints supported by this logic can be extended. For instance, the notion of a cardinality atom can be generalized to cover all pseudo-boolean constraints. One can also consider constructs to impose conditions on set cardinalities other than simple lower and upper bounds that we considered here. An example of such a constraint is the *parity* constraint. Similarly, there is much room for improvement in the area of solvers for the propositional logic *PS*+. We believe the notion of branching on cardinality atoms is to a large degree responsible for a good performance of *aspps* and deserves further attention. There is also a potential for developing good local search techniques for the logic *PS*+. Recent work [Liu and Truszczyński 2003] substantiates that claim. In contrast, finding successful local-search for logic programming systems turned out to be hard [Dimopoulos and Sideris 2002].

Another question is related to the problem of the expressive power and concerns auxiliary data predicates specifying ranges of integers. In our encoding of the transitive closure problem, the data set contains a collection of ground atoms $index(i)$, $1 \leq i \leq n$, where n is the number of vertices in the input graph. This set is not an inherent element of the input specification, which consists only of vertices and edges of the graph. We do not know whether there are uniform encodings of the transitive closure problem in the logic *PS*+, which do not require any auxiliary index ranges being included as data. More generally, we do not know whether disallowing such auxiliary ranges of integers has any effect on the expressive power of our formalism.

The next open research direction concerns comparisons of the logic *PS*+ to logic programming with the answer-set semantics and, specifically, the matter of suitability of the logic *PS*+ as a knowledge representation formalism. The language of our logic lacks default negation. That raises questions whether the logic *PS*+ can succinctly model frame axioms, normative statements and incomplete information, and whether one can build representations in the logic *PS*+ that are are elaboration tolerant. We believe that, to some degree, the lack of the default negation can be circumvented. First, for broad classes of logic programs (for instance, for programs encoding planning problems and many aspects of reasoning about action) default negation can be compiled away in a systematic way into propositional representations without any essential growth in the size of the theory. The approach is based on Fages lemma and its generalizations [Erdem and Lifschitz 2003], and on the concept of the Clark's completion [Clark 1978; Apt 1990], and led to the development of *cmodels* [Babovich and Lifschitz 2002]. These results suggest that, at least for some broad classes of applications, default negation can be *effectively* modeled within the logic *PS*+. Second, we showed that the expressive power of the logic *PS*+ is the same as that of the stable logic programming. That provides an "existential" argument for the fact that default negation can be simulated within the logic *PS*+ (albeit, perhaps in a non-modular way). Third, the logic *PS*+ is nonmonotonic. This nonmonotonicity seems to be weaker than that of logic programming with answer-set semantics. However, its properties and potential for capturing knowledge representation properties have not yet been studied and are not fully understood.

Finally, our work brings up a general question of the scope of the ASP paradigm.

In the paper we demonstrated that predicate logic and its extensions provide a foundation for effective declarative programming systems, in which problems are encoded as theories so that their models represent solutions. This way of problem solving is closely reminiscent to that followed by ASP systems and our work brings up an important issue of the scope of the ASP paradigm. It is most commonly associated with formalisms based on logic programming with the stable-model semantics and disjunctive logic programming with the answer-set programming. In fact, the term ASP is often considered synonymous with these logic programming systems. We argue that the scope of the ASP paradigm should be extended beyond logic programming.

There are several reasons to do so. First, the fundamental shift that led from Prolog-type logic programming to logic programming with the answer-set semantics (the original understanding of the ASP), consists of moving the focus from proofs to models and that is where it main contribution lies. However, once that is said, there is no reason to confine that computational paradigm to logic programming with the answer-set semantics. Logics where models can be represented as sets abound and, as demonstrated by our logics *PS* and *PS*+ that are based on predicate calculus with Herbrand models, they can lead to to expressive and computationally effective declarative programming systems.

Second, broadening the scope of ASP provides a direct linkage to SAT and, more generally, constraint satisfaction. Research in these areas resulted in a broad range of fast algorithms for testing satisfiability of propositional formulas and collections of pseudo-boolean constraints. These algorithms are based on advances in search techniques and constraint propagation, as well as methods developed in the area of integer programming. Arguably, they are currently more advanced than the present generation of tools to compute answer sets of logic programs. Extending the ASP paradigm to logics such as *PS* and *PS*+, where model finding reduces directly to propositional satisfiability and pseudo-boolean satisfiability, provides a direct way to tap into recent dramatic advances in solver performance achieved by the constraint satisfaction research.

We view making this emerging connection between ASP and constraint satisfaction explicit and broadly understood an important objective. SAT and SAT(PB) communities focused on the development of fast satisfiability solvers and to a large degree neglected the issue of modeling languages. In most cases, they are restricted in their syntactic constructs and geared towards specific solvers. On the other hand, ASP focused much of its effort on the development of modeling languages being driven by knowledge representation applications where, at least initially, the focus was on modeling and not on computing. Thus, bringing the two areas together will benefit both. It will provide SAT and SAT(PB) areas with effective programming front-ends for their solvers and will provide fast processing back-ends for use with ASP languages.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grants No. 9874764, 0097278 and 0325063. The authors wish to thank anonymous referees for their valuable comments and useful pointers to work related

to ours.

REFERENCES

Abiteboul, S. and Vianu, V. 1991. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci. 43,* 1, 62–124.

Aloul, F., Ramani, A., Markov, I., and Sakallah, K. 2002. PBS: a backtrack-search pseudo-boolean solver and optimizer. In *Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability.* 346 – 353.

Aloul, F., Ramani, A., Markov, I., and Sakallah, K. 2003. PBS v0.2, incremental pseudo-boolean backtrack search SAT solver and optimizer. `http://www.eecs.umich.edu/~faloul/Tools/pbs/`.

Apt, K. 1990. Logic programming. In *Handbook of theoretical computer science*, J. van Leeuven, Ed. Elsevier, Amsterdam, 493–574.

Babovich, Y. and Lifschitz, V. 2002. *Cmodels package.* `http://www.cs.utexas.edu/users/tag/cmodels.html`.

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving.* Cambridge University Press.

Barth, P. 1995. A Davis-Putnam based elimination algorithm for linear pseudo-boolean optimization. Tech. rep., Max-Planck-Institut für Informatik. MPI-I-95-2-003.

Bayardo, Jr, R. and Schrag, R. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-1997).* AAAI Press, 203–208.

Benhamou, B., Sais, L., , and Siegel, P. 1994. Two proof procedures for a cardinality based language in propositional calculus. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science (STACS-1994).* LNCS, vol. 775. Springer, 71–82.

Cadoli, M. and Mancini, T. 2002. Combining Relational Algebra, SQL, and Constraint Programming. In *Proceeding of the 4th International Workshop on Frontiers of Combining Systems (FroCoS-2002).* LNAI, vol. 2309. Springer, 147–161.

Cadoli, M. and Palipoli, L. 1998. Circumscribing datalog: expressive power and complexity. *Theor. Comput. Sci. 193*, 215–244.

Cadoli, M. and Schaerf, A. 2001. Compiling problem specifications into SAT. In *Proceedings of the European Symposium On Programming (ESOP-2001).* LNAI, vol. 2028. Springer, 387–401.

Clark, K. 1978. Negation as failure. In *Logic and data bases*, H. Gallaire and J. Minker, Eds. Plenum Press, New York-London, 293–322.

Colmerauer, A., Kanoui, H., Pasero, R., and Roussel, P. 1973. Un systeme de communication homme-machine en francais. Tech. rep., University of Marseille.

Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., and Pfeifer, G. 2003. Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003).* Morgan Kaufmann, 847–852.

Dimopoulos, Y. and Sideris, A. 2002. Towards local search for answer sets. In *Proceedings of the 18th International Conference on Logic Programming.* LNCS, vol. 2401. Springer, 363 – 367.

Dixon, H. and Ginsberg, M. 2002. Inference methods for a pseudo-boolean satisfiability solver. In *The 18th National Conference on Artificial Intelligence (AAAI-2002).* AAAI Press, 635–640.

East, D. and Truszczyński, M. 2000. Datalog with constraints. In *Proccedings of the 17th National Conference on Artificial Intelligence (AAAI-2000).* AAAI Press, 163–168.

East, D. and Truszczyński, M. 2001. ASP solver *aspps.* `http://www.cs.uky.edu/aspps/`.

East, D. and Truszczyński, M. 2001. Propositional satisfiability in answer-set programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence (KI-2001).* LNAI, vol. 2174. Springer, 138–153.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. 1998. A KR system `dlv`: Progress report, comparisons and benchmarks. In *Proceeding of the 6th International Conference on Knowledge Representation and Reasoning (KR-1998)*. Morgan Kaufmann, 406–417.

Erdem, E. and Lifschitz, V. 2003. Tight logic programs. *Theory and Practice of Logic Programming 3,* 4-5, 499–518.

Fagin, R. 1974. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Computation*, R. Karp, Ed. AMS, 43–74.

Fourer, R., Gay, D., and Kernigham, B. 1993. *AMPL: A Modeling Language for Mathematical Programming.* International Thompson Publishing.

Gelfond, M. and Lifschitz, V. 1988. The stable semantics for logic programs. In *Proceedings of the 5th International Conference on Logic Programming.* MIT Press, 1070–1080.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Gen. Comput. 9*, 365–385.

Ginsberg, M. and Parkes, A. 2000. Satisfiability algorithms and finite quantification. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning, (KR-2000).* Morgan Kaufmann, 690–701.

Gottlob, G., Kolaitis, P., and Schwentick, T. 2000. Existential second-order logic over graphs: Charting the tractability frontier. In *Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS-2000).* IEEE CS Press, 664–674.

Green, C. 1969. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI-1969).* Morgan Kaufmann, 741–747.

Kautz, H., McAllester, D., and Selman, B. 1996. Encoding plans in propositional logic. In *Proceedings of 5th International Conference on Principles of Knowledge Representation and Reasoning (KR-1996).* Morgan Kaufmann, 374–384.

Kowalski, R. 1974. Predicate logic as a programming language. In *Proceedings of the Congress of the International Federation for Information Processing (IFIP-1974).* North Holland, Amsterdam, 569–574.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. 2003. The dlv system for knowledge representation and reasoning. `http://xxx.lanl.gov/abs/cs.AI/0211004`.

Li, C. 1997. SAT solver *satz*. `http://www.laria.u-picardie.fr/~cli/EnglishPage.html`.

Li, C. and Anbulagan, M. 1997. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming.* LNCS, vol. 1330. Springer, 342–356.

Lin, F. and Zhao, Y. 2002. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proccedings of the 18th National Conference on Artificial Intelligence (AAAI-2002).* AAAI Press, 112–117.

Liu, L. and Truszczyński, M. 2003. Local-search techniques in propositional logic extended with cardinality atoms. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, CP-2003.* LNCS, vol. 2833. Springer, 495–509.

Marek, V. and Remmel, J. 2003. On the expressibility of stable logic programming. *Theory and Practice of Logic Programming 3*, 551–567.

Marek, V. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, K. Apt, W. Marek, M. Truszczyński, and D. Warren, Eds. Springer, Berlin, 375–398.

Marek, W. and Truszczyński, M. 1993. *Nonmonotonic Logic; Context-Dependent Reasoning.* Springer, Berlin.

McCarthy, J. 1980. Circumscription — a form of non-monotonic reasoning. *Artificial Intelligence 13,* 1-2, 27–39.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. 2001a. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th ACM IEEE Design Automation Conference.* ACM Press, 530–535.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. 2001b. SAT solver *chaff*. `http://www.ee.princeton.edu/~chaff/`.

Nerode, A. and Shore, R. 1993. *Logic and logic programming*. Springer, Berlin.

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

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

Niemelä, I., Simons, P., and Syrjänen, T. 1997. SLP solver *smodels*. `http://www.tcs.hut.fi/Software/smodels/`.

Parkes, A. 1999. Lifted search engines for satisfiability. Ph.D. thesis, University of Oregon, Department of Computer Science.

Prestwich, S. 2002. Randomised backtracking for linear pseudo-boolean constraint problems. In *Proceedings of the 4th International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems, (CPAIOR-2002)*. 7–20. `http://www.emn.fr/x-info/cpaior/Proceedings/CPAIOR.pdf`.

Robinson, J. 1965. A machine-oriented logic based on resolution principle. *J. ACM 12*, 23–41.

Saccà, D. 1997. The expressive powers of stable models for bound and unbound datalog queries. *J. Comput. Syst. Sci. 54,* 3, 441–464.

Schlipf, J. 1995. The expressive powers of the logic programming semantics. *J. Comput. Syst. Sci. 51,* 1, 64–86.

Selman, A. 1994. A taxonomy of complexity classes of functions. *J. Comput. Syst. Sci. 48,* 2, 357–381.

Selman, B., Kautz, H., and Cohen, B. 1994. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-1994)*. AAAI Press, Seattle, USA, 337–343.

Simons, P., Niemelä, I., and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 181–234.

Ullman, J. 1988. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, MD.

van Hentenryck, P. 1999. *The OPL Optimization Programming Language*. The MIT Press.

Vardi, M. 1982. The complexity of relational query languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC-1982)*. ACM Press, 137–146.

Wallace, M., Novello, S., and Schimpf, J. 1997. Ecl^{i}ps^{e}: A platform for constraint logic programming. `http://www.icparc.ic.ac.uk/eclipse/reports/eclipse.ps.gz`.

Walser, J. 1997. Solving linear pseudo-boolean constraints with local search. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-97)*. AAAI Press, 269–274.

Walsh, T. 2003. Challenges in SAT (and QBF). Invited talk at 6th International Conference on Theory and Applications of Satisfiability Testing. Slides available from http://4c.ucc.ie/ tw/ sat2003.ppt.

Received November 2002; revised December 2003; accepted March 2004