aspps — an implementation of answer-set programming with propositional schemata

Deborah East and Mirosław Truszczyński

Department of Computer Science
University of Kentucky
Lexington KY 40506-0046, USA

Abstract. We present an implementation of an answer-set programming paradigm, called *aspps* (short for answer-set programming with propositional schemata). The system *aspps* is designed to process PS^+-theories. It consists of two basic modules. The first module, *psgrnd*, grounds an PS^+-theory. The second module, referred to as *aspps*, is a solver. It computes models of ground PS^+-theories.

1 Introduction

The most advanced answer-set programming systems are, at present, *smodels* [NS00] and *dlv* [ELM+98]. They are based on the formalisms of logic programming with stable-model semantics and disjunctive logic programming with answer-set semantics, respectively. We present an implementation of an answer-set programming system, *aspps* (short for answer-set programming with propositional schemata). It is based on the *extended logic of propositional schemata with closed world assumption* that we denote by PS^+. We introduced this logic in [ET01].

A theory in the logic PS^+ is a pair (D, P), where D is a collection of ground atoms representing a *problem instance* (input data), and P is a *program* — a collection of PS^+-clauses (encoding of a problem to solve). The meaning of a PS^+-theory $T = (D, P)$ is given by a *family* of PS^+-models [ET01]. Each model in this family represents a solution to a problem encoded by P for data instance D.

The system *aspps* is designed to process PS^+-theories. It consists of two basic programs. The first of them, *psgrnd*, grounds a PS^+-theory. That is, it produces a ground (propositional) theory extended by a number of special constructs. These constructs help model cardinality constraints on sets. The second program, referred to as *aspps*, is a solver. It computes models of grounded PS^+-theories. It is designed along the lines of a standard Davis-Putnam algorithm for satisfiability checking. Both *psgrnd* and *aspps*, examples of PS^+-programs and the corresponding performance results are available at `http://www.cs.uky.edu/ai/aspps/`.

2 PS^+-theories

A PS^+*-theory* is a pair (D, P), where D is a collection of ground atoms and P is a collection of PS^+-clauses. Atoms in D represent input data (an instance of a problem). In our implementation these atoms may be stored in one or more *data* files. The set of PS^+-clauses models the constraints (specification) of the problem. In our implementation, all the PS^+-clauses in P are stored in a single *rule* file.

All statements in data and rule files must end with a period (.). Clauses may be split across several lines. Blank lines can be used in data and rule files to improve readability. Comments may be used too. They begin with '%' and continue to the end of the line.

Data files. Each ground atom in a data file must be given on a single line. Constant symbols may be used as arguments of ground atoms. In such cases, these constant symbols must be specified at the command line (see Section 3). Examples of ground atoms are given below:

$vtx(2)$.
$vtx(3)$.
$size(k)$.

A set of ground atoms of the form $\{p(m), p(m+1), \dots, p(n)\}$, where m and n are non-negative integers or integer constants specified at the command line, can be represented in a data file as '$p[m..n]$.'. Thus, the two ground atoms $vtx(2)$ and $vtx(3)$ can be specified as '$vtx[1..3]$.'.

Predicates used by ground atoms in data files are called *data predicates*.

Rule files. The rule file of a PS^+-theory consists of two parts. In the first one, the *preamble*, we declare all *program* predicates, that is, predicates that are not used in data files. We also declare types of all variables that will be used in the rule files. Typing of variables simplifies the implementation of the grounding program *psgrnd* and facilitates error checking.

Arguments of each program predicate are typed by unary *data* predicates (the idea is that when grounding, each argument can only be replaced by an element of an extension of the corresponding unary data predicate as specified by the data files). A program predicate q with n arguments of types $dp_1, \dots, dp_n$, where all dp_i are data predicates, is declared in one of the following two ways:

$pred\ q(dp_1, \dots, dp_n)$.
$pred\ q(dp_1, \dots, dp_n) : dp_m$.

In the second statement, the n-ary data predicate dp_m further restricts the extension of q — it must be a subset of the extension of dp_m (as specified by the data files).

Variable declarations begin with the keyword *var*. It is followed by the *unary* data predicate name and a list of alpha-numeric strings serving as variable names (they must start with a letter). Thus, to declare two variables X and Y of type dp, where dp is a unary data predicate we write:

$var\ dp\ X, Y$.

The implementation allows for *predefined* predicates and function symbols such as the equality operator ==, arithmetic comparators <=, >=, < and >, and arithmetic operations +, $-$, $*$,/, *abs*() (absolute value), $mod(N, b)$, $max(X, Y)$ and $min(X, Y)$. We assign to these symbols their standard interpretation. However, we emphasize that the domains are restricted only to those constants that appear in a theory.

The second part of the rule file contains the program itself, that is, a collection of clauses describing constraints of the problem to be solved.

By a *term tuple* we mean a tuple whose each component is a variable or a constant symbol, or an arithmetic expression. An atom is an expression of one of the following four forms.

1. $p(t)$, where p is a predicate (possibly a predefined predicate) and t is a tuple of variables, constants and arithmetic expressions.
2. $p(t, Y) : dp(Y)$, where p is a program predicate, t is a term tuple, and dp is a unary data predicate
3. $m\{p(t) : d_1(t_1) : \ldots : d_k(t_k)\}n$, where p is a program predicate, each d_i is a data or a predefined predicate, and t and all t_i are term tuples
4. $m\{p_1(t), \ldots, p_k(t)\}n$, where all p_i are program predicates and t is a term tuple

Atoms of the second type are called *e-atoms* and atoms of types 3 and 4 are called *c-atoms*. Intuitively, an e-atom '$p(t, Y) : dp(Y)$' stands for 'there exists Y in the extension of the data predicate dp such that $p(t, Y)$ is true'. An intuitive meaning of a c-atom '$m\{p(t) : d_1(t_1) : \ldots : d_k(t_k)\}n$' is: from the set of all atoms $p(t)$ such that for every i, $1 \leq i \leq k$, $d_i(t^{p.i})$ is true ($t^{p.i}$ is a projection of t onto attributes of d_i), at least m and no more than n are true. The meaning of a c-atom '$m\{p_1(t), \ldots, p_k(t)\}n$' is similar: at least m and no more than n atoms in the set $\{p_1(t), \ldots, p_k(t)\}$ are true.

We are now ready to define clauses. They are expressions of the form

$$A_1, \ldots, A_m \rightarrow B_1 | \ldots | B_n.$$

where A_i's and B_j's are atoms, ',' stands for the conjunction operator and '|' stands for the disjunction operator.

3 Processing PS^+-theories

To compute models of a PS^+-theory (D, P) we first ground it. To this end, we use the program *psgrnd*. Next, we compute models of the ground theory produced by *psgrnd*. To accomplish this task, we use the program *aspps*. For the detailed description of the grounding process and, especially, for the treatment of e-atoms and c-atoms, and for a discussion of the design of the *aspps* program, we refer the reader to [ET01].

The required input to execute *psgrnd* is a single program file, one or more data files and optional constants. If no errors are found while reading the files

and during grounding, an output file is constructed. The output file is a machine readable file whose name is a catenation of the constants and file names with the extension **.tdc**.
psgrnd **-r rfile -d dfile1 dfile2** ... **[-c c1=v1 c2=v2** ...]

Required arguments

-r **rfile** is the file describing the problem (rule file). There must be exactly one rule file.
-d **datafilelist** is one or more files containing data that will be used to instantiate the theory.

Optional arguments

-c **name=value** This option allows the use of constants in both the data and rule files. When **name** is found while reading input files it is replaced by **value**; **value** can be any string that is valid for the data type. If **name** is to be used in a range specification, then **value** must be an integer.

The program *aspps* is used to solve the grounded theory constructed by *psgrnd*. The name of the file containing the theory is input on the command line. After executing the *aspps* program, a file named aspps.stat is created or appended with statistics concerning this run of *aspps*.
aspps **-f filename [-A] [-P] [-C [x]] [-S name]**

Required arguments

-f **filename** is the name of the file containing a theory produced by *psgrnd*.

Optional arguments

-A Prints the positive atoms for solved theories in readable form.
-P Prints the input theory and then exits.
-C [**x**] Counts the number of solutions. This information is recorded in the statistics file. If **x** is specified it must be a positive integer; *aspps* stops after finding **x** solutions or exhausting the whole search space, whichever comes first.
-S **name** Show positive atoms with predicate name.

References

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

ET01. D. East and M. Truszczyński. Propositional satisfiability in answer-set programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI'2001.* Lecture Notes in Artificial Intelligence, Springer Verlag, 2001.

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