Answer-Set Programming: a Declarative Knowledge Representation Paradigm

ESSLLI'2001 — Logic and Computing: an introductory course

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

Mirosław Truszczyński
University of Kentucky
Department of Computer Science
Lexington, KY 40506–0046, USA
`http://www.cs.engr.uky.edu/~mirek/`
e-mail: `mirek@cs.uky.edu`

1 Overview

The area of nonmonotonic reasoning originated about twenty years ago [Rei78, McC80, Rei80, MD80] in response to challenges of knowledge representation and, more general, artificial intelligence. Commonsense reasoning, defeasible inferences, belief revision and formalizations of the frame problem seemed to require a different framework than that offered by first-order logic. As a result, nonmonotonic logics were proposed: default logic by Reiter [Rei80], modal nonmonotonic logics by McDermott and Doyle [MD80, McD82] and Moore [Moo85], and circumscription by McCarthy [McC80].

In about the same time, Horn logic programming was proposed by Kowalski as an alternative, declarative as opposed to procedural, programming paradigm [Kow74, Kow79]. Horn logic programming offered a promise of an easy program development and verification. But with negation in the syntax, it was no longer clear what the semantics is. The problem was successfully addressed in the late 80s with the emergence of the stable model semantics by Gelfond and Lifschitz [GL88], well-founded semantics by Van Gelder, Ross and Schlipf [VRS91], and some other related semantics capturing aspects of the intuition of the negation as failure.

There are strong connections between logic programming with negation and nonmonotonic reasoning. For one, stable model semantics is nonmonotonic itself. Secondly, logic programming with stable model semantics can be viewed as a fragment of default logic [MT89, BF91]. In fact, logic programming with stable model semantics for a number of years now has been regarded as a computational embodiment of nonmonotonic reasoning and a candidate for an effective knowledge representation tool. How to actually process logic programs under stable model semantics and how to interpret them has been, however, less clear. The stable model semantics seemed to be at odds with the traditional logic programming paradigm of a single intended model.

In the late 90s several implementations of algorithms for computing stable models of logic programs and disjunctive logic programs were proposed [NS96, Sim00, NS00, ELM$^+$97, ELM$^+$98]. And, as they were studied, it became clear that they are instantiations of a different programming paradigm than that of standard logic programming. This new paradigm has been formally identified in the late 90s and has since become known as Answer-Set Programming (or ASP, for short) [MT99, Nie99, Lif99]. Answer-set programming is a paradigm in which programs are built as theories in some formal system $\mathcal{F}$ with a well-defined syntax, and with a semantics that assigns to a theory P in the system a *collection* of subsets of some domain. These subsets are referred to as *answer sets* of P and specify the results of computation based on P. To solve a problem Π in an ASP formalism, we find a program P so that the solutions to Π can be reconstructed in polynomial time from the answer sets to P.

What are the benefits of ASP over traditional logic programming approaches? We believe that the major advantage of ASP is its simplicity and connection to constraint programming. There are no function symbols, and there is no need for resolution or unification (at least in the present implementations; this situation may change as the field develops). The role of recursion is dramatically restricted. Essentially, the only recursion that is needed is that required by transitive closure specification. Clauses and programs are viewed as *constraints* on sets. Programs are developed by representing problem constraints in a direct and straightforward fashion.

Further, most ASP formalisms have their roots in default logic or other similar nonmonotonic logics. Consequently, modeling incomplete information, defeasible reasoning and such phenomena as the frame problem become quite easy. Finally, there is some formal and experimental evidence that domain or problem representations using ASP formalisms based on nonmonotonic logics lead to more concise programs. Conciseness of representations is important as it might lead to significant speed up in processing times. Ease of modeling and program development is important as it may determine whether the approach will become widely accepted.

In recent years it became evident that answer-set programming formalisms can also be built on top of propositional satisfiability checkers and their extensions [ET00]. The logic PS+ and its implementation, system *aspps* [ET01], is an illustration of this approach.

Answer-set programming is a new development. It is too early to say definitely whether it will make any significant impact on the practice of declarative programming and practical knowledge representation. There are, however, strong indications that it will. Its theoretical foundations are well understood, including complexity and expressive power. It is applicable to a wide range of search and decision problems. It is closely related to constraint programming. Last but not least, there are already promising implementations.

2 Applications

We briefly review interesting areas where ASP systems have already been used.

- Planning

  Planning was one of the main areas that motivated the development of nonmonotonic reasoning systems and the use of ASP in this area has been studied by a number of researchers, see e.g. [DNK97, Lif99, Nie99, Par01, TB01, SBM01]. An interesting example of an advanced project is research at Texas Tech University on developing a decision support system for the flight controllers of space shuttles using the ASP approach [BBG+01].

- Product configuration

  General methodology for product configuration using ASP techniques has been developed [SN99, SNTS00, SNTS01] and interesting application projects have been started, see e.g. the WeCoTin project (`http://www.soberit.hut.fi/WeCoTin/`). Research on software configuration using ASP [Syr00a, Syr00b] has led to a prototype configurator for the complete Debian Linux system distribution [Syr99] (`http://www.tcs.hut.fi/~tssyrjan/configuration/`).

- Computer aided verification

  ASP has been used for a variety of key inference tasks in computer aided verification. The `Smodels` system has been employed as an inference engine in a model checker [LRS98]. An analysis method for Petri nets based on finite complete prefixes has been built on top of `Smodels` [Hel99b, Hel99a, EH01], see the `mcsmodels` and `unfsmodels` tools (`http://www.tcs.hut.fi/~kepa/tools/`). Similarly, it has been used in a stubborn set reduction method for Petri net reachability analysis [Var00], see the `prod` tool (`http://www.tcs.hut.fi/Software/prod/`). Recently, a symbolic model checking method for asynchronous systems based on bounded model checking techniques and ASP has been developed [HN01a, HN01b].

There are a number other interesting areas where ASP methodology has been employed including dynamic constraint satisfaction [SGN99], wire routing [ELW00, ET01], network management [SL01], feature interaction [AAdR99, AABdR00], logical cryptanalysis [HMN00], security protocol analysis [AM00], network inhibition analysis [ABS00], diagnosis [GG01], and computation of (total and partial) stable models for disjunctive programs [JNSY00].

3 Where to get the software?

This tutorial discusses knowledge representation tools based on a new computational paradigm of Answer Set Programming (ASP). ASP has its roots in nonmonotonic reasoning and, especially, in logic programming with negation under the stable model semantics. There are several implementations of this paradigm. The two discussed in the tutorial are *Smodels* and *aspps*. The former has been developed in Helsinki University of Technology in the group led by Ilkka Niemelä. The latter has been developed in the University of Kentucky by Deborah East and Mirosław Truszczyński.

Two other well advanced ASP systems are *dlv* and *ccalc*. The *dlv* system has been developed at the Technical University of Vienna by Nicola Leone, Thomas Eiter and their collaborators. The *ccalc* system has been developed at the University of Texas at Austin by Norm McCain and is now being maintained by Texas Action Group. Yet another system of interest is *ldl++* developed by Natraj Ami and Kayliang Ong at MCC (with technical guidance from Carlo Zaniolo). The urls for all these systems are given below.

smodels: `http://www.tcs.hut.fi/Software/smodels/`
aspps: `http://www.cs.engr.uky.edu/ai/aspps/`
dlv: `http://www.dbai.tuwien.ac.at/dlv/`
ccalc: `http://www.cs.utexas.edu/users/tag/cc/`
ldl++: `http://www.cs.ucla.edu/ldl/`

4 Example programs

In this section we present programs for solving several logic puzzles, combinatorial problems and planning problems. We start by providing problem statements. Then, we present *Smodels* and, later, *aspps* programs for these problems.

4.1 Problems

Missionaries and cannibals. There are three cannibals and three missionaries on the left bank of a river. They have a boat. The boat has capacity 2. If at any time there are more cannibals than missionaries on either bank, the cannibals eat the missionaries. The goal is to arrange the way of moving all 6 of them to the right bank. The problem of missionaries and cannibals is a classic AI puzzle that received quite a lot of attention [Lif00].

15-puzzle. A 4×4 grid is filled with 15 movable tiles labeled $1, \ldots, 15$. One square is left blank. Tiles can be rearranged by moving a tile into a blank square (assuming the two locations share an edge). The goal is to rearrange the tiles so that they are arranged from 1 to 15 in the *row-major* order, with the blank square occupying the left top corner. The 15-puzzle problem is a hard search problem [Kor00].

Blocks-world planning. Blocks are arranged in stacks on the floor. Each block is located on exactly one block or on the floor. No two blocks are placed on the same block. A block that is on top of its stack can be moved and placed on the top of another stack or on the floor. Find a plan (a sequence of actions) to rearrange the blocks into a desired final configuration. Both sequential (only one block moved in a step) and concurrent (allows for several blocks to be moved in the same step) versions are of interest. This is a classic planning problem in AI.

n-queens problem. Given an $n \times n$ chessboard, the goal is to place on it n queens so that no two queens attack each other (that is, no two queens are in the same row, column or in the same diagonal). The n-queens problem is often used in testing constraint solvers.

Hamilton cycle problem. Given a directed graph G, find a cycle in G that goes through all vertices of the graph exactly once (if such a cycle exists). The Hamilton-cycle problem is a classic NP-complete graph problem.

Vertex-cover problem. A set of vertices of an undirected graph is called a *vertex cover* if every edge in the graph has at least one of its ends in the set. The problem is, given an undirected graph G and an integer k, to compute a vertex cover of G with no more than k vertices (if such a cover exists). The vertex-cover problem is another classic NP-complete problem. It is also a difficult test problem for search algorithms.

Schur numbers. A set A of integers is sum-free if for every $x, y \in A$ (not necessarily distinct), $x + y \notin A$. The Schur number $S(k)$ is defined as the smallest integer n such that the set $\{1, 2, \ldots, n\}$ **cannot** be partitioned into k parts so that each part is sum-free.

4.2 *Smodels* programs

A typical command line for the `Smodels` system is of the form

```
lparse [options] file1 file2 | smodels m
```

Typical options are '-c const=n' to set a constant appearing in the program and '-dn' for eliminating all domain predicates while doing grounding in order to reduce the size of the grounded program. The integer m gives the number of models that need to be found (default is one and '0' means all models).

Hence, a command line for the missionaries-and-cannibals encoding presented next could be the following

```
lparse -dn -c n=11 mc.lp | smodels 0
```

where the constant n is set to '11' and *smodels* is instructed to find all stable models.

For more information on the language accepted by the `Smodels` system, the options and examples, see the `lparse` user's manual available in the distribution.

Missionaries and cannibals. The two key predicates are *state* and *move*. Atom $state(CC, MM, B, T)$ is read as follows: at time T there are CC cannibals and MM missionaries on the left bank. $B = 1$ means that the boat is on the left bank, $B = 0$ means that the boat is on the right bank. Atom $move(C, M, T)$ means that at time T, C cannibals and M missionaries are moving to the opposite bank of the river (the direction depends on the location of the boat).

```
% Domain predicates
time(0..n).           %  specifies time range (number of moves)
                      %  in which the solution is to be found
number(0..3).         %  specifies allowed number of cannibals
                      %  (or missionaries) on a bank
boatcap(0..2).        %  specifies allowed numbers of
                      %  people in the boat

% Initial state:
% 3 cannibals and 3 missionaries on the left (1)
% bank of the river at time 0.

state(3,3,1,0).

%  Define safe states
safestate(CC,MM) :- number(CC;MM), MM==0.
safestate(CC,MM) :- number(CC;MM), MM>=CC.

%  Select a valid move at time T, given the current state

good1(CC,MM,C,M) :- boatcap(C;M), number(CC;MM), time(T),
        1 <= C+M, C+M <= 2, C <= CC, M <= MM,
        safestate(CC-C,MM-M),
        safestate(3-CC+C,3-MM+M).

1 { move(C,M,T): boatcap(C): boatcap(M):  good1(CC,MM,C,M)} 1 :-
        time(T), number(CC;MM),
```

```
        state(CC,MM,1,T),
        not goal(T).

good0(CC,MM,C,M) :- boatcap(C;M), number(CC;MM), time(T),
        1 <= C+M, C+M <= 2, C <= 3-CC, M <= 3-MM,
        safestate(CC+C,MM+M),
        safestate(3-CC-C,3-MM-M).

1 { move(C,M,T): boatcap(C): boatcap(M): good0(CC,MM,C,M)} 1 :-
        time(T), number(CC;MM),
        state(CC,MM,0,T),
        not goal(T).

%  Compute the next state
state(CC-C,MM-M,0,T+1) :- number(CC;MM), boatcap(C;M), time(T),
        state(CC,MM,1,T), move(C,M,T).

state(CC+C,MM+M,1,T+1) :- number(CC;MM), boatcap(C;M), time(T),
        state(CC,MM,0,T), move(C,M,T).

%  Determine whether the goal has been reached at time T
goal(T) :- time(T), state(0,0,0,T).

%  Determine that the goal was reached
ok :- time(T), goal(T).

%  Eliminate all plans which do not lead to the goal
:- not ok.
```

15-puzzle. The key predicates are *move* and *in*. Atom *move(T,X,Y)* has the meaning: at time T tile 0 is moved to location *(X,Y)* and *in(T,X,Y,A)* is read as: at time T tile A is in location *(X,Y)*. The initial configuration is given as facts *in0(x,y,a)*.

```
% An example initial configuration
% 12 moves required
%   1  5  2  7
%   4  0  3 11
%   8  9  6 15
%  12 13 10 14
```

```
in0(1,1,1).   in0(1,2,5).  in0(1,3,2).  in0(1,4,7).
in0(2,1,4).   in0(2,2,0).  in0(2,3,3).  in0(2,4,11).
in0(3,1,8).   in0(3,2,9).  in0(3,3,6).  in0(3,4,15).
in0(4,1,12).  in0(4,2,13). in0(4,3,10). in0(4,4,14).

% Domain predicates

time(0..t).  pos(1..4).  entry(0..15).

% Select where to move tile 0
{ move(T,X,Y) } :- time(T), T < t, pos(X;Y;X1;Y1),
        in(T,X1,Y1,0), abs(X-X1) + abs(Y-Y1) == 1.

% Only one move at a time allowed
:- 2 { move(T,X,Y):pos(X):pos(Y) }, time(T), T< t.

% For every time there must be a move
:- { move(T,X,Y):pos(X):pos(Y) } 0, time(T), T< t.


% Effects of a move
% Define the moving tile for each time T
moving(T,A) :- time(T), T < t, pos(X;Y), entry(A),
        move(T,X,Y), in(T,X,Y,A).

% The moving tile A is moved to the location of tile 0
in(T+1,X,Y,A) :- time(T), T < t, pos(X;Y), entry(A),
        moving(T,A), in(T,X,Y,0).

% Tile 0 is moved to the location (X,Y) when doing move(T,X,Y)
in(T+1,X,Y,0) :- time(T), T < t, pos(X;Y), move(T,X,Y).

% Frame axiom
in(T+1,X,Y,A) :- time(T), T < t, entry(A), pos(X;Y), A != 0,
        in(T,X,Y,A), not moving(T,A).

% An example of a pruning rule
% Do not undo the last move
:- time(T), T < t-2, pos(X;Y), in(T,X,Y,0), in(T+2,X,Y,0).

% Goal configuration
```

```
%  0  1  2  3
%  4  5  6  7
%  8  9 10 11
% 12 13 14 15

in_t(1,1,0).   in_t(1,2,1).   in_t(1,3,2).   in_t(1,4,3).
in_t(2,1,4).   in_t(2,2,5).   in_t(2,3,6).   in_t(2,4,7).
in_t(3,1,8).   in_t(3,2,9).   in_t(3,3,10).  in_t(3,4,11).
in_t(4,1,12).  in_t(4,2,13).  in_t(4,3,14).  in_t(4,4,15).

% Goal configuration must be satisfied
:- entry(A), pos(X), pos(Y), in(t,X,Y,A), not in_t(X,Y,A).
:- entry(A), pos(X), pos(Y), not in(t,X,Y,A), in_t(X,Y,A).

% Initial configuration
in(0,X,Y,A) :- in0(X,Y,A).
```

Blocks-world planning. Key predicates are *moveop* and *on*. Atom *moveop(X, Y, T)* has the meaning: at time T block X is moved on top of object Y and *on(X, Y, T)* is read as: at time T block X is on object Y. Initial conditions include facts about the blocks and *on* predicate and a goal rule.

```
% An example of initial and goal conditions
% Blocks
block(a). block(b). block(c).           %             b
                                        %    a        c
% Initial configuration                 %    b  c     a
on(a,b,0). on(b,floor,0). on(c,floor,0). %    -------
                                        %    Initial  Goal
% Goal conditions
goal(T) :- time(T), on(b,c,T), on(c,a,T).

% Domain predicates
time(0..n).

nextstate(Y,X) :- time(X), time(Y),
        Y = X + 1.

% Auxiliary predicates
```

```
object(floor).
object(X) :- block(X).

covered(X,T) :- block(Z), block(X), time(T),
        on(Z,X,T).

on_something(X,T) :- block(X), object(Z), time(T),
        on(X,Z,T).

available(floor,T) :- time(T).
available(X,T) :- block(X), time(T),
        on_something(X,T).

% Select a moveop provided that preconditions hold
{ moveop(X,Y,T) } :-
        time(T), block(X), object(Y),
        X != Y, on_something(X,T),
        available(Y,T),
        not covered(X,T),
        not covered(Y,T).

% Operator effects:
on(X,Y,T2) :- block(X), object(Y),
        nextstate(T2,T1),
        moveop(X,Y,T1).

% Frame axioms
on(X,Y,T2) :- block(X), object(Y),
        nextstate(T2,T1),
        on(X,Y,T1),
        not moving(X,T1).

moving(X,T) :-  time(T), block(X), object(Y),
        moveop(X,Y,T).

% Concurrent moveop instances are allowed
% A block cannot be moved to two destination
:- 2 { moveop(X,Y,T):object(Y) },
       block(X), time(T).
% The destination cannot be moving
:- block(X), object(Y), time(T),
```

```
        moveop(X,Y,T),
        moving(Y,T).
% No two blocks moved onto the same block
:- 2 { moveop(X,Y,T):block(X) },
       block(Y), time(T).


% Exclude models where the goal has not been reached.
:- not goal.
goal :- time(T), goal(T).
goal(T2) :- nextstate(T2,T1), goal(T1).

% Optimizations
% Stop when the goal has been reached
:- block(X), object(Y), time(T),
        moveop(X,Y,T),
        goal(T).

% No move from floor to floor
:- block(X), time(T),
       moveop(X,floor,T), on(X,floor,T).

% No move on something and then to floor
:- nextstate(T2,T1), block(X), object(Y),
        moveop(X,Y,T1), moveop(X,floor,T2).
```

n-queens problem. Domain predicate d specifies the number of queens and the dimensions of the board. The main predicate is the predicate q. Atom $q(X,Y)$ represents the fact that there is a queen in the position (X,Y).

```
% Domain predicate: the dimension of the board.
d(1..n).

% Exactly one queen in each row
1 { q(X,Y):d(Y) } 1 :- d(X).

% Exactly one queen in each column
1 { q(X,Y):d(X) } 1 :- d(Y).

% No two queens on a diagonal
```

```
:- d(X;Y;X1;Y1), q(X,Y), q(X1,Y1),
        X < X1,  Y != Y1, abs(X - X1) == abs(Y - Y1).
```

Hamilton cycle problem. The graph is given as facts $vtx(X)$ and $edge(X, Y)$ and one vertex v is given as initial vertex $initialvtx(v)$. The key predicate is hc. Atom $hc(X, Y)$ represents the fact that edge (X, Y) is in the Hamiltonian cycle to be constructed.

```
% Select edges for the cycle
{ hc(X,Y) } :- edge(X,Y).

% Each vertex has at most one incoming edge in a cycle
:- 2 { hc(X,Y):edge(X,Y) }, vtx(Y).

% Each vertex has at most one outgoing edge in a cycle
:- 2 { hc(X,Y):edge(X,Y) }, vtx(X).

% Every vertex must be reachable from the initial vertex through
% chosen hc(v,u) edges
:- vtx(X), not r(X).

r(Y) :- hc(X,Y), edge(X,Y), initialvtx(X).
r(Y) :- hc(X,Y), edge(X,Y), r(X), not initialvtx(X).
```

Vertex-cover problem. The graph is given as facts $vtx(X)$ and $edge(X, Y)$ and k is the upper bound for the number of vertices in a cover. The main predicate is $incover$. Atom $incover(X)$ represents the fact that vertex X is in a vertex cover.

```
% choose at least one of the ends of an edge
% to be included in the cover
1 { incover(X), incover(Y) } :- edge(X,Y).

% Eliminate models where k+1 or more vertices have been chosen
:- k+1 {incover(X):vtx(X) }.
```

Schur numbers. The range of integers to be partitioned is n and the number of parts is k. The main predicate is $inpart$. Atom $inpart(X, P)$ represents the fact that number X is included in part P. The idea is to find a partition of integers $\{1, 2, ..., n\}$

into k parts such that each part is sum free, i.e. for any X and Y, X and $X+X$ are in different parts and if X and Y are in the same part, then $X+Y$ is in a different part.

```
% Domain predicates
number(1..n). part(1..p).

% Assign each integer to exactly one part.
1 { inpart(X,P):part(P) } 1 :- number(X).

% X and X+X cannot be in the same part
:- number(X),  part(P), inpart(X,P), inpart(X+X,P).

% X, Y, and X+Y cannot be in the same part
:- number(X;Y), X != Y, part(P),
        inpart(X,P), inpart(Y,P), inpart((X+Y),P).


% Remove some symmetries by taking always the lowest free part
:- number(X), part(P), inpart(X,P),
        part(P1), P1 < P, not occupied(X,P1).

% occupied(X,P) iff there is a number Y < X in P
occupied(X,P) :- number(X;Y), part(P),  Y < X, inpart(Y,P).
```

4.3 *aspps* programs

Each program consists of two files: the *data* file (`data`) and the `rule` file (`rules`). In each case, we start by producing a ground version of the program. To this end we run the `grnd` program as follows:

```
grnd -c t1=k1 t2=k2 -d data -r rules
```

Here $t1$ and $t2$ are two constants appearing in the program whose values (say, $k1$ and $k2$) are entered at the command line when invoking `grnd`. If necessary, more constants can be specified in this way.

The program `grnd` produces a ground version of the program that is stored in the file `k1_k2_data_rules.tdc`. To find answer sets we run the program `aspps`. Its input is the file `k1_k2_data_rules.tdc` produced by `grnd`.

```
aspps -f k1_k2_data_rules.tdc
```

Option -S followed by a list of predicates, prints out all the ground atoms in the solution that are built of predicates on the list. For instance,

```
aspps -f k1_k2_data_rules.tdc -S p1 p2
```

lists all ground atoms built of predicates $p1$ and $p2$ that belong to a solution found by `aspps`. No output is produced if no solutions were found. A more detailed description of the programs `grnd` and `aspps` is available at the website given in Section 3.

Missionaries and cannibals. Data file.

```
time[0..t].          %  specifies time range (number of moves) in
                     %  which the solution is to be found
num[0..3].           %  specifies allowed number of cannibals (or
                     %  missionaries on a bank
boat[0..1].          %  0 - boat on the left bank
                     %  1 - boat on the right bank
instate(3,3,0).      %  initial state
```

Rules file. The two key predicates are *state* and *move*. Atom $state(T, CC, MM, B)$ has the following meaning: at time T there are CC cannibals and MM missionaries on the left bank. $B = 0$ means that the boat is on the left bank, $B = 1$ means that the boat is on the right bank. Atom $move(T, C, M)$ means that at time T, C cannibals and M missionaries are moving to the opposite bank of the river (the direction depends on the location of the boat).

```
% Declare predicates
   pred state( time , num , num , boat ).
   pred move( time , num , num ).

% Declare variables
   var time T.
   var num M,C,MM,CC.
   var boat B.

% At time 0, the state coincides with the initial state given
% by the input
```

```
   state(0,CC,MM,B) -> instate(CC,MM,B).

% Select a configuration at time T
   1{state(T,CC,MM,B):num(CC):num(MM):boat(B)}1.

% Select a move for time T
   T<t -> 1{move(T,C,M):num(C):num(M)}1.

% No moves at time t (the plan ends at time t)
   move(t,C,M) -> .

% Enforce the boat capacity constraint
   move(T,C,M) -> M+C<=2.

% At least one person must be moving to the other bank
   move(T,C,M) -> 0<C+M.

% If the boat is on the left bank no more than CC cannibals
% and no more than MM missionaries may be chosen to move to
% the other side
   state(T,CC,MM,0),move(T,C,M) -> C<=CC.
   state(T,CC,MM,0),move(T,C,M) -> M<=MM.

% If the boat is on the right bank no more than 3-CC cannibals
% (this is how many of them are on the right bank) and no more
% than 3-MM missionaries may be chosen to move to the other side
   state(T,CC,MM,1),move(T,C,M) -> C<=3-CC.
   state(T,CC,MM,1),move(T,C,M) -> M<=3-MM.

% Ensure that a move is not immediately "undone"
   0<T, T<t, move(T,C,M), move(T-1,C,M) -> .

% Relate the next state to the previous one and to the move
% chosen
   T<t, move(T,C,M), state(T,CC,MM,0) -> state(T+1,CC-C,MM-M,1).
   T<t, move(T,C,M), state(T,CC,MM,1) -> state(T+1,CC+C,MM+M,0).

% Eliminate states not safe for the missionaries
   state(T,CC,MM,B), MM!=0, CC > MM -> .
   state(T,CC,MM,B), MM!=3, CC < MM -> .
```

```
% At time t the goal state must be reached
   state(t,0,0,1).
```

15-puzzle. See the slides (Section 5).

Blocks-world planning. Data predicate *time* specifies the time horizon (constant t is entered from the command line). Data predicate block defines the set of blocks (constant k is entered from the command line). Data predicate *on0* defines the initial configuration. Atom $on0(A, B)$ means that block A is (initially) on block B. Similarly, data predicate *onF* provides a specification for the goal configuration. Atom $onF(A, B)$ means that in the goal configuration block A must be on block B. The specification of the goal configuration may be incomplete. The data file given here is the example large.c, proposed by Kautz and Selman (`http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html`).

Key program predicates are *on*, *move* and *top*. Atom $on(T, A, B)$ means that at time T block A is on block B. Atom $move(T, A)$ represents the fact that at time T, block A is one of the blocks to be moved. Finally, atom $top(T, A)$ means that at time T block A is at the top of its stack of blocks.

Data file.

```
time[0..t].

block[0..15].    % Defines the set of blocks; 0 represents the floor

% Initial configuration
   on0(13,0).  on0(12,13).  on0(1,12).  on0(2,1).  on0(3,2).
   on0(15,0).  on0(14,15).  on0(4,14).  on0(5,4).  on0(10,5).
   on0(11,10).  on0(6,0).  on0(7,6).  on0(8,7).  on0(9,8).

% Partial description of the goal configuration
   onF(5,10).  onF(1,5).  onF(14,1).  onF(9,4).  onF(8,9).
   onF(13,8).  onF(15,13).  onF(11,7).  onF(3,11).  onF(2,3).
   onF(12,2).
```

Rule file.

```
%% Declare predicates
```

```
       pred on( time , block , block ).
       pred move( time , block ).
       pred top( time , block ).

%% Declare variables
       var time T.
       var block A,B,C.

%% Specify an arrangement of blocks at time T
   % Each block other than the floor is on top of exactly one
   % block
       0<A -> 1 { on(T,A,B): block(B) } 1.
   % Each block other than the floor is either a top block or
   % is under exactly one block
       0<A -> 1{top(T,A), on(T,B,A): block(B)}1.
   % floor is never on any other block
       on(T,0,A) -> .
   % floor is never a top
       top(T,0) -> .


%% The state at time 0 must coincide with the initial configuration
       on(0,A,B) -> on0(A,B).
       on0(A,B) -> on(0,A,B).

%% The state at time t must be consistent with the specification
%% of the goal
       onF(A,B) -> on(t,A,B).

%% Constraints on moves
   % Floor never moves
       move(T,0) -> .
   % Moved blocks are restricted to top blocks.
       on(T,B,A), move(T,A) -> .

   % No move is made at time t.
       move(t,A) -> .

%% A block cannot be moved on top of a block that is being moved
%% in the same move
        move(T,A),move(T,B),on(T+1,A,B) -> .
```

```
%% No block is directly moved back where it was
       move(T,A), on(T,A,B), on(T+2,A,B) -> .

%% No block moves on the same location it was in
       move(T,A), on(T,A,B), on(T+1,A,B) -> .

%% Compute the new state
   % Both of these constraints together are not necessary. But
   % when put together they seem to work best.
       T<t, on(T,A,B) -> 1{on(T+1,A,B), move(T,A)}1.
       on(T+1,A,B) -> 1{move(T,A), on(T,A,B)}1.
```

n-queens problem. Data predicate *number* specifies the number of queens and the dimensions of the board. The main predicate is the predicate *queen*. Atom $queen(C, R)$ represents the fact that there is a queen in the position (C, R).

Data file.

```
number[1..n].
```

Rule file.

```
% Predicate declarations
   pred queen(number, number).

% Variable declarations
   var number C, R, I.

% Exactly one queen in each row
   1 { queen(C,R) : number(C) } 1.

% Exactly one queen in each column
   1 { queen(C,R) : number(R) } 1.

% No two queens on an "ascending" diagonal
   queen(C,R), queen(C+I,R+I) -> .

% No two queens on a "descending" diagonal
   queen(C,R), queen(C+I,R-I) -> .
```

Hamilton cycle problem. We specify the rule file only. The data file must define the vertex set of the graph (for instance, the statement `vtx[1..n].` will define the vertex set to be $\{1, \ldots, n\}$, where n is a parameter entered from the command line). The data file must also list all (directed) edges of the graph as facts $edge(a, b)$, where *edge* is another data predicate.

The key program predicate is *cycle*. Atom $cycle(X, Y)$ represents the fact that vertex Y is in position X. In other words, the predicate *cycle* represents a permutation of the vertices of the graph. The permutation represents a Hamilton cycle if for every position X, there is an edge between the vertex in position X and the vertex in position $X + 1$ (the next position after n is 1).

Rule file.

```
% Declare program predicates
   pred cycle( vtx , vtx ).

% Declare variables
   var vtx X, Y, Z.

% Choose an ordering of nodes on the cycle. For each
% vertex Y choose exactly one location X (the type vtx
% is used to represent locations)
   1 { cycle(X,Y): vtx(X) } 1.

% Choose an ordering of nodes on the cycle, continued.
% For each location X choose exactly one vertex Y
   1 { cycle(X,Y): vtx(Y) } 1.

% Enforce the hamiltonicity restriction
   X<n, cycle(X,Y), cycle(X+1,Z) -> edge(Y,Z).
   cycle(n,Y), cycle(1,Z) -> edge(Y,Z).
```

Vertex-cover problem. We specify the rule file only. The data file must define the vertex set of the graph (for instance, the statement `vtx[1..n].` will define the vertex set to be $\{1, \ldots, n\}$, where n is a parameter entered from the command line). The data file must also list all the edges of the graph as facts $edge(a, b)$, where *edge* is another data predicate. The main (and only) program predicate is *incover*. Atom $incover(X)$ represents the fact that the vertex X is in a vertex cover.

Rule file.

```
% Declare program predicates
   pred incover( vtx ) .

% Declare variables
   var vtx X, Y.

% Select a set of vertices for a vertex cover
   { incover(X) : vtx(X) } k.

% Enforce the vertex-cover constraint:at least one end
% of each edge must be in the cover
   edge(X,Y) -> incover(X) | incover(Y).
```

Schur numbers. The data file specifies the range of integers to be partitioned, n, and the number of parts, k. This can be accomplished by two statements (*number* and *part* are data predicates): `number[1..n].` and `part[1..k].`

The main program predicate is *inpart*. Atom *inpart*(X, P) represents the fact that number X is included in part P.

Rule file.

```
% Declare program predicates
   pred inpart(number,part).

% Declare variables
   var number X,Y.
   var part P.

% Assign each integer to exactly one part
   1 { inpart(X,P) : part(P) } 1.

% Enforce the sum-free constraint
   inpart(X,P), inpart(Y,P), inpart(X+Y,P) -> .
```

References

[AABdR00] R. Accorsi, C. Areces, W. Bouma, and M. de Rijke. Features as constraints. In *Proceedings of the Sixth Feature Interaction Workshop*, Glasgow, Scotland, May 2000.

[AAdR99] R. Accorsi, C. Areces, and M. de Rijke. Towards feature interaction via stable models. In *Proceedings of the Second Workshop on Formal Methods*, Florianpolis, Brasil, October 1999.

[ABS00] T. Aura, M. Bishop, and D. Sniegowski. Analyzing single-server network inhibition. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 108–117, Cambridge, UK, July 2000. IEEE Computer Society Press.

[AM00] L.C. Aiello and F. Massacci. An executable specification language for planning attacks to security protocols. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 88 –102, Cambridge, UK, July 2000. IEEE Computer Society Press.

[BBG+01] M. Balduccini, M. Barry, M. Gelfond, M. Nogueira, and R. Watson. An a-prolog decision support system for the space shuttle. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, Las Vegas, Nevada, 2001. Springer-Verlag.

[BF91] N. Bidoit and C. Froidevaux. Negation by default and unstratifiable logic programs. *Theoretical Computer Science*, 78(1, (Part B)):85–112, 1991.

[DNK97] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*, pages 169–181, Toulouse, France, September 1997. Springer-Verlag.

[ET00] D. East and M. Truszczyński. Datalog with constraints. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 163–168, 2000.

[ET01] D. East and M. Truszczyński. Answer-set programming with propositional schemata. Submitted, 2001.

[ET01] D. East and M. Truszczyński. More on wire routing with asp. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 39–44, Stanford, USA, March 2001. AAAI Press.

[ELM+97] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A deductive system for non-monotonic reasoning. In *Logic programming and nonmonotonic reasoning (Dagstuhl, Germany, 1997)*, volume 1265 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 1997.

[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.

[ELW00] E. Erdem, V. Lifschitz, and M.D.F. Wong. Wire routing and satisfiability planning. In *Proceedings of the First International Conference on Computational Logic, Automated Deduction: Putting Theory into Practice*, pages 822–836, London, U.K., July 2000. Springer-Verlag.

[EH01] J. Esparza and K. Heljanko. Implementing LTL model checking with net unfoldings. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'2001)*, Toronto, Canada, May 2001. Accepted for publication.

[GG01] M. Gelfond and J. Galloway. Diagnosing dynamic systems in A-Prolog. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 160–166, Stanford, USA, March 2001. AAAI Press.

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

[Hel99a] K. Heljanko. Minimizing finite complete prefixes. In H.-D. Burkhard, L. Czaja, H.S. Nguyen, and P.H. Starke, editors, *Concurrency, Specification and Programming (Proceedings of the CS&P'99 Workshop)*, pages 83–95, Warsaw, Poland, 1999.

[Hel99b] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fundamenta Informaticae*, 37(3):247–268, 1999.

[HMN00] M. Hietalahti, F. Massacci, and I. Niemelä. DES: a challenge problem for nonmonotonic reasoning systems. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (cs.AI/0003073)*, Breckenridge, Colorado, USA, April 2000. cs.AI/0003039.

[HN01a] K. Heljanko and I. Niemelä. Answer set programming and bounded model checking. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 90–96, Stanford, USA, March 2001. AAAI Press.

[HN01b] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Vienna, Austria, September 2001. Springer-Verlag.

[JNSY00] T. Janhunen, I. Niemelä, P. Simons, and J. You. Unfolding partiality and disjunctions in stable model semantics. In A.G. Cohn, F. Guinchiglia, and B. Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning*, pages 411–419, Breckenridge, Colorado, USA, April 2000. Morgan Kaufmann Publishers.

[Kor00] Richard E. Korf. Recent progress in the design and analysis of admissible heuristic functions. In *Proccedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 1165–1170, 2000.

[Kow74] R. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP 74*, pages 569–574, Amsterdam, 1974. North Holland.

[Kow79] R. Kowalski. *Logic for Problem Solving.* North Holland, Amsterdam, 1979.

[Lif99] V. Lifschitz. Answer set planning. In *Logic programming, Proceedings of the 1999 International Conference on Logic Programming*, pages 23–37. MIT Press, 1999.

[Lif00] V. Lifschitz. Missionaries and cannibals in the causal calculator. In *Principles of Knowledge Representation and Reasoning, Proceedings of the Seventh International Conference (KR2000)*, pages 85 – 96. Morgan Kaufmann Publishers, 2000.

[LRS98] X. Liu, C.R. Ramakrishnan, and S.A. Smolka. Fully local and efficient evaluation of alternating fixed points. In Bernhard Steffen, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 5–19, Lisbon, Portugal, March/April 1998. Springer-Verlag.

[McC80] J. McCarthy. Circumscription — a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1-2):27–39, 1980.

[McD82] D. McDermott. Nonmonotonic logic II: nonmonotonic modal theories. *Journal of the ACM*, 29(1):33–57, 1982.

[MD80] D. McDermott and J. Doyle. Nonmonotonic logic I. *Artificial Intelligence*, 13(1-2):41–72, 1980.

[Moo85] R.C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[MT89] W. Marek and M. Truszczyński. Stable semantics for logic programs and default theories. In E.Lusk and R. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 243–256. MIT Press, 1989.

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

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

[NS96] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of JICSLP-96*. MIT Press, 1996.

[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.

[Par01] A. Parmar. A declarative implementation of planning with control. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 160–166, Stanford, USA, March 2001. AAAI Press.

[Rei78] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and data bases*, pages 55–76. Plenum Press, 1978.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

[SBM01] T.C. Son, C. Baral, and S. McIlraith. Extending answer set planning with sequence, conditional, loop, non-deterministic choice, and procedure constructs. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 202–209, Stanford, USA, March 2001. AAAI Press.

[SGN99] T. Soininen, E. Gelle, and I. Niemelä. A fixpoint definition of dynamic constraint satisfaction. In Joxan Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, pages 419–433, Alexandria, Virginia, USA, October 1999. Springer-Verlag.

[Sim00] P. Simons. Extending and implementing the stable model semantics. Doctoral Dissertation. Research report A58, Helsinki University of Technology, Helsinki, Finland, April 2000.

[SL01] T.C. Son and J. Lobo. Reasoning about policies using logic programs. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 210–216, Stanford, USA, March 2001. AAAI Press.

[SN99] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In Gopal Gupta, editor, *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 305–319, San Antonio, Texas, January 1999. Springer-Verlag.

[SNTS00] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Unified configuration knowledge representation using weight constraint rules. In *Workshop Notes of the ECAI'2000 Configuration Workshop*, pages 79–84, Berlin, Germany, August 2000.

[SNTS01] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 195–201, Stanford, USA, March 2001. AAAI Press.

[Syr99] T. Syrjänen. A rule-based formal model for software configuration. Research Report A55, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Helsinki, Finland, December 1999.

[Syr00a] T. Syrjänen. Including diagnostic information in configuration models. In *Proceedings of the First International Conference on Computational Logic, Automated Deduction: Putting Theory into Practice*, pages 837–851, London, U.K., July 2000. Springer-Verlag.

[Syr00b] T. Syrjänen. Optimizing configurations. In *Workshop Notes of the ECAI'2000 Configuration Workshop*, pages 85–90, Berlin, Germany, August 2000.

[TB01] L. Tuan and C. Baral. Effect of knowledge representation on model based planning: Experiments using logic programming encodings. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 110–115, Stanford, USA, March 2001. AAAI Press.

[VRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[Var00] K. Varpaaniemi. Stable models for stubborn sets. *Fundamenta Informaticae*, 43(1–4):355–375, 2000.

5 Slides