

Answer Set Programming

Ilkka Niemelä

Department of Information and Computer Science
Aalto University, Finland
Ilkka.Niemela@tkk.fi
<http://users.ics.tkk.fi/ini/>

Part I

Introduction to ASP

Content

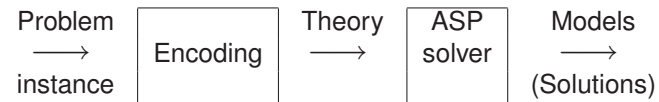
- ▶ Introduction to Answer Set Programming (ASP)
- ▶ Stable Model Semantics
- ▶ Solving Problems with ASP
- ▶ ASP Solver Technology
- ▶ Further Information: Systems, Applications, Literature

Answer Set Programming

- ▶ Term coined by Vladimir Lifschitz.
- ▶ Roots: KR, logic programming, nonmonotonic reasoning.
- ▶ Based on some formal system with semantics that assigns a theory a collection of answer sets (models).
- ▶ An **ASP solver**: computes answer sets for a theory.
- ▶ Solving a problem in ASP:
Encode the problem as a theory such that **solutions** to the problem are given by **answer sets** of the theory.

ASP—cont'd

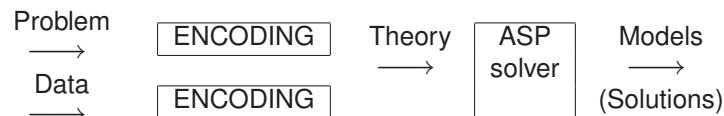
- ▶ Solving a problem using ASP



Possible formal system	Models
Propositional logic	Truth assignments
CSP	Variable assignments
Logic programs	Stable models
Model expansion	First-order structures

ASP Using Logic Programs

- ▶ Uniform encoding:
separate problem specification and data
- ▶ Compact, easily maintainable representation
- ▶ Integrating KR, DB, and search techniques
- ▶ Handling dynamic, knowledge intensive applications:
data, frame axioms, exceptions, defaults, closures



Example. k -coloring problem

- ▶ Given a graph (V, E) find an assignment of one of k colors to each vertex such that no two adjacent vertices share a color.
- ▶ Encoding 3-coloring using propositional logic
 - ▶ For each vertex $v \in V$ include the clauses:
 $v_1 \vee v_2 \vee v_3$
 $\neg v_1 \vee \neg v_2$
 $\neg v_1 \vee \neg v_3$
 $\neg v_2 \vee \neg v_3$
 - ▶ and for each edge $(v, u) \in E$ the clauses:
 $\neg v_1 \vee \neg u_1$
 $\neg v_2 \vee \neg u_2$
 $\neg v_3 \vee \neg u_3$
- ▶ 3-colorings of a graph (V, E) and models of the encoding correspond: vertex v colored with color i iff v_i true in a model.

Coloring Problem (Uniform Encoding)

```
% Problem encoding
1 { colored(V,C):color(C) } 1 :- vtx(V).
:- edge(V,U), color(C), colored(V,C), colored(U,C).

% Data
vtx(a). ...
edge(a,b). ...
color(r). color(g). ...
```

👉 Legal colorings of the graph given as data and stable models of the problem encoding and data correspond:
a vertex v colored with a color c iff `colored(v, c)` holds in a stable model.

What is ASP Good for?

Search problems:

- ▶ Constraint satisfaction
- ▶ Planning, routing
- ▶ Computer-aided verification
- ▶ Security analysis
- ▶ Linguistics
- ▶ Network management
- ▶ Product configuration
- ▶ Combinatorics
- ▶ Diagnosis

 Declarative problem solving

ASP Using Logic Programs

- ▶ Logic programming: framework for merging KR, DB, and search
- ▶ PROLOG style logic programming systems not directly suitable for ASP:
 - ▶ search for proofs (not models) and produce answer substitutions
 - ▶ not entirely declarative
- ▶ In late 80s new semantical basis for “negation-as-failure” in LPs based on nonmonotonic logics: **Stable model semantics**
- ▶ Implementations of stable model semantics led to ASP

Part II

Stable Model Semantics

LPs with Stable Models Semantics

- ▶ Consider first normal logic program rules

$$A \leftarrow B_1, \dots, B_m, \text{not } C_1, \dots, \text{not } C_n$$

- ▶ Seen as constraints on an answer set (stable model):
 - ▶ if $B_1, \dots, B_m$ are in the set and
 - ▶ none of $C_1, \dots, C_n$ is included,then A must be included in the set
- ▶ A stable model is a set of atoms
 - (i) which satisfies the rules and
 - (ii) where each atom is **justified** by the rules (negation by default)

Stable Models — cont'd

- Program:

$$\begin{aligned} b &\leftarrow \\ f &\leftarrow b, \text{not } eb \\ eb &\leftarrow p \end{aligned}$$
- Stable model:

$$\{b, f\}$$
- Another candidate model: $\{b, eb\}$
satisfies the rules but is not a proper stable model:
 eb is included for no reason.
- Justifiability of stable models is captured by the notion of a **reduct** of a program.
-  The stable model semantics [Gelfond/Lifschitz, 1988].

Definite Programs

- For the reduct we need to consider first definite programs, i.e. normal programs without negation (not $\neg$).
- Such a program P has a unique least model $LM(P)$ satisfying the rules.
- $LM(P)$ can be constructed, e.g., by forward chaining.

Examples.

$P_1 :$	$P_2 :$	$P_3 :$
$p \leftarrow$	$p \leftarrow q$	$p \leftarrow q$
$q \leftarrow p$	$q \leftarrow p$	$q \leftarrow p$
$LM(P_1) = \{p, q\}$	$LM(P_2) = \{\}$	$LM(P_3) = \{p, q\}$

Stable Models — cont'd

- Consider the propositional (variable free) case:
 P — ground program
 S — set of ground atoms
- Reduct P^S (Gelfond-Lifschitz)
 - delete each rule having a body literal **not** C with $C \in S$
 - remove all negative body literals from the remaining rules
- P^S is a definite program (and has a unique least model $LM(P^S)$)
- S is a stable model of P iff $S = LM(P^S)$.**

Example. Stable models

S	P	P^S	$LM(P^S)$
$\{b, f\}$	$b \leftarrow$ $f \leftarrow b, \text{not } eb$ $eb \leftarrow p$	$b \leftarrow$ $f \leftarrow b$ $eb \leftarrow p$	$\{b, f\}$
$\{b, eb\}$	$b \leftarrow$ $f \leftarrow b, \text{not } eb$ $eb \leftarrow p$	$b \leftarrow$ $f \leftarrow b$ $eb \leftarrow p$	$\{b\}$

- The set $\{b, eb\}$ is not a stable model of P but $\{b, f\}$ is the (unique) stable model of P

Example. Stable models

- ▶ A program can have **none**, one, or **multiple** stable models.
- ▶ Program: $p \leftarrow \text{not } q$
 $q \leftarrow \text{not } p$ Two stable models:
 $\{p\}$
 $\{q\}$
- ▶ Program: $p \leftarrow \text{not } p$ No stable models

Programs with variables

- ▶ Hence, the rule $\text{path}(X,Y) :- \text{edge}(X,Y).$ in P represents:
 $\text{path}(1,1) :- \text{edge}(1,1).$
 $\text{path}(1,2) :- \text{edge}(1,2).$
 $\text{path}(2,1) :- \text{edge}(2,1).$
 $\text{path}(2,2) :- \text{edge}(2,2).$
 $\text{path}(1,3) :- \text{edge}(1,3).$
 $\dots$
- ▶ The Herbrand base of a program is the set ground atoms built from the predicates and the Herbrand universe of the program.
- ▶ For P the Herbrand base is
 $\{ \text{path}(1,1), \text{edge}(1,1), \text{path}(1,2), \dots \}$
- ▶ A Herbrand model is a subset of the Herbrand base.

Programs with variables

- ▶ Variables are needed for uniform encodings
- ▶ Semantics: **Herbrand models**
- ▶ A rule is seen as a shorthand for the set of its ground instantiations over the Herbrand universe of the program
- ▶ The **Herbrand universe** is the set of terms built from the constants and functions in the program

Example. For the program P :

```
edge(1,2).  
edge(1,3).  
edge(2,4).  
path(X,Y) :- edge(X,Y).  
path(X,Y) :- edge(X,Z), path(Z,Y).
```

The Herbrand universe is $\{ 1, 2, 3, 4 \}$.

Programs with variables

- ▶ The grounding of a program P yields:
 - ▶ a propositional logic program
 - ▶ built of atoms from the Herbrand base of P , $HB(P)$
 - ▶ denoted $\text{grnd}(P)$.
- ▶ $M \subseteq HB(P)$ is a stable model of P if M is a stable model of $\text{grnd}(P)$.

Stable Models — cont'd

- ▶ A stratified program (no recursion through negation) has a unique stable model (canonical model).
- ▶ It is **linear time to check** whether a set of atoms is a stable model of a ground program.
- ▶ It is **NP-complete to decide** whether a ground program has a stable model.
- ▶ Normal programs (without function symbols) give a **uniform encoding** to every NP search problem.

Extensions to Normal Programs

▶ Encoding of choices

- ▶ A key point in ASP
- ▶ Choices can be encoded using normal rules with unstratified negation

$$\begin{aligned}a &\leftarrow \text{not } a', b, \text{not } c \\ a' &\leftarrow \text{not } a\end{aligned}$$

- ▶ **Choice rules**, however, provide a much more intuitive encoding:

$$\{a\} \leftarrow b, \text{not } c$$

- ▶ Disjunctive rules: $a \vee a' \leftarrow b, \text{not } c$
 - ▶ Higher expressivity and complexity (Σ_2^P)
 - ▶ Special purpose implementations (dlv, claspD)
 - ▶ Can be implemented also using an ASP solver for normal programs as the **core engine** (GnT)

Extensions to Normal Programs

- ▶ An **integrity constraint** is a rule without a head:

$$\leftarrow B_1, \dots, B_m, \text{not } C_1, \dots, \text{not } C_n$$

- ▶ It can be seen as a shorthand for

$$F \leftarrow \text{not } F, B_1, \dots, B_m, \text{not } C_1, \dots, \text{not } C_n$$

- ▶ and it eliminates stable models where the body $B_1, \dots, B_m, \text{not } C_1, \dots, \text{not } C_n$ is satisfied.

- ▶ **Classical negation** can be handled by normal programs (renaming):

$$p \leftarrow \text{not } \neg p \quad \text{corresponds to} \quad \begin{aligned} p &\leftarrow \text{not } p' \\ &\leftarrow p, p' \end{aligned}$$

Extensions — cont'd

- ▶ Many extensions implemented using an ASP solver as the **core engine**:

- ▶ preferences
- ▶ nested logic programs
- ▶ circumscription, planning, diagnosis, ...

▶ Aggregates

- ▶ count
Example: choose 2–4 hard disks
- ▶ sum
Example: the total capacity of the chosen hard disks must be at least 200 GB.
- ▶ Built-in support for aggregates in the search procedures

Extensions — cont'd

- ▶ Optimization
Example: prefer the cheapest set of hard disks
- ▶ Weak constraints with weight and priority levels

$:\sim B_1, \dots, B_m, \text{not } C_1, \dots, \text{not } C_n[w : l]$

(built-in support in `d1v`)

- ▶ Function symbols
 - ▶ Stable model semantics is highly undecidable if arbitrary function symbols are allowed.
 - ▶ (Safety) restrictions needed to guaranteeing decidability:

$d_edge(t(V), t(U)) \leftarrow edge(V, U), \text{not } edge(U, V)$

- ▶ Built-in predicates and functions:

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

Example. Rules in `lparse`

- ▶ Cardinality constraints
 $2 \{ \text{hd_1}, \dots, \text{hd_n} \} 4$
- ▶ Weight constraints
 $200 [\text{hd_1} = 60, \dots, \text{hd_n} = 130]$

A.k.a. **pseudo-Boolean constraints**:

$$60hd_1 + \dots + 130hd_n \geq 200$$

- ▶ Optimization
`minimize [ hd_1 = 100, ..., hd_n = 180 ].`
- ▶ Conditional literals:
expressing sets in cardinality and weight constraints
 $1 \{ \text{colored}(V,C) : \text{color}(C) \} 1 :- \text{vtx}(V).$

Part III

Solving Problems using ASP

- ▶ Uniform encodings: separate data and problem encoding
- ▶ Basic methodology: **generate and test**
 - ▶ **Generator rules**: provide candidate answer sets (typically encoded using choice constructs)
 - ▶ **Tester rules**: eliminate non-valid candidates (typically encoded using integrity constraints)
 - ▶ **Optimization statements**: Criteria for preferred answer sets (typically using cost functions)

Example: Coloring

```
% Problem encoding

% Generator rule
1 {colored(V,C):color(C)} 1 :- vtx(V).

% Tester rule
:- edge(V,U), color(C), colored(V,C), colored(U,C).

% Optimization statement
minimize {colored(V,4):vtx(V)}.

% Data
vtx(a). ...
edge(a,b). ...
color(r). color(g). ...
```

Example: Review assignment

```
% Data
reviewer(r1),...
paper(p1), ...
classA(r1,p1), ... % Preferred papers
classB(r1,p2), ... % Doable papers
coi(r1,p3), ...    % Conflicts of interest

% Problem encoding

% Generator rule
% Each paper is assigned 3 reviewers
3 { assigned(P,R):reviewer(R) } 3 :- paper(P).
```

Tester Rules

- ▶ Integrity constraints
- ▶ $\text{:- } a_1, \dots, a_n, \text{ not } b_1, \dots, \text{ not } b_m.$
- ▶ eliminate stable models but cannot introduce new ones:
 - ▶ Let P be a program and IC a set of integrity constraints
 - ▶ Then S is a stable model of $P \cup IC$ iff:
 - ▶ S is a stable model of P , and
 - ▶ S satisfies all ICs

Review Assignment — cont'd

```
% Tester rules

% No paper assigned to a reviewer with coi
:- assigned(P,R), coi(R,P).
% No reviewer has an unwanted paper.
:- paper(P), reviewer(R),
   assigned(P,R), not classA(R,P), not classB(R,P).
% No reviewer has more than 8 papers
:- 9 { assigned(P,R): paper(P) }, reviewer(R).
% Each reviewer has at least 7 papers
:- { assigned(P,R): paper(P) } 6, reviewer(R).
% No reviewer has more than 2 classB papers
:- 3 { assignedB(P1,R): paper(P1) }, reviewer(R).
assignedB(P,R) :- classB(R,P), assigned(P,R).
% Minimize the number of classB papers
minimize [ assignedB(P,R):paper(P):reviewer(R) ].
```

Example: Satisfiability

- ▶ Given a formula, solutions to the satisfiability problem are propositional models, i.e., sets of atoms.

☞ Candidate answer sets.

- ▶ **Generator**

- ▶ For each atom a_i in the formula, introduce a **choice rule** $\{ a_i \}$.
- ▶ For the program: 2^n stable models:
 $\{ a_1 \}$ $\{ \}$
 $\dots$ $\dots$
 $\{ a_n \}$ $\{ a_1, \dots, a_n \}$

Satisfiability — cont'd

- ▶ For more involved testers consider general formulas.
For example, $(a \vee \neg b) \wedge (\neg a \leftrightarrow b)$.

- ▶ Generator: for each atom x , rule $\{ x \}$.
 $\{ a \}$.
 $\{ b \}$.

Satisfiability — cont'd

- ▶ Satisfiability **testers** for formulas illustrate how to encode complicated logical conditions using ASP.
- ▶ For a clause $a_1 \vee \dots \vee a_n \vee \neg b_1 \vee \dots \vee \neg b_m$ a satisfiability tester can be given as an integrity constraint:

$:- \text{not } a_1, \dots, \text{not } a_n, b_1, \dots, b_m.$

- ▶ **Example.**

Clauses T	Program P_T	Stable model
$a \vee \neg b$	$:- \text{not } a, b.$	$\{ a \}$
$\neg b \vee \neg a$	$:- a, b.$	
$b \vee a$	$:- \text{not } a, \text{not } b.$	$\{ a \}, \{ b \}.$

- ▶ Models of T and stable models of P_T correspond

Satisfiability — cont'd

- ▶ Tester — evaluates a formula q recursively
- ▶ For each subformula:
 - ▶ the conditions under which it is true are given
 - ▶ false cases by default: it is false unless otherwise stated
- ▶ A satisfying truth assignment: a stable model satisfying

$:- \text{not } q.$

Satisfiability — cont'd

- ▶ Tester encoding

Subformula p	Rules
$l_1 \wedge \dots \wedge l_n$	$p \leftarrow p_{l_1}, \dots, p_{l_n}$
$l_1 \vee \dots \vee l_n$	$p \leftarrow p_{l_1}$...
	$p \leftarrow p_{l_n}$
$\neg l$	$p \leftarrow \text{not } p_l$
$l_1 \leftrightarrow l_2$	$p \leftarrow p_{l_1}, p_{l_2}$ $p \leftarrow \text{not } p_{l_1}, \text{not } p_{l_2}$

Satisfiability — cont'd

- ▶ For the formula $p_1: \underbrace{(a \vee \neg b)}_{p_2} \wedge \underbrace{(\neg a \leftrightarrow b)}_{p_3}$

▶ Program:

```

:- not p1.
p1:- p2, p3.
p2:- a.
p2:- not b.
p3:- a, not b.
p3:- not a, b.
{ a }. { b }.

```

Stable models:
{a,p1,p2,p3}

- ▶ Satisfying truth assignments for p_1 and the stable models of the program correspond

Fixed Points

- ▶ The stable model semantics captures inherently **minimal fixed points** enabling compact encodings of **closures**
- ▶ **Example.** Reachability from node s .

```

r(s).
r(V) :- edge(U,V), r(U).
edge(a,b). ...

```

- ▶ The program captures reachability:
it has a unique stable model S s.t. v is reachable from s iff $r(v) \in S$.
- ▶ **Example.** Transitive closure of a relation $q(X, Y)$

```

t(X,Y) :- q(X,Y).
t(X,Y) :- q(X,Z), t(Z,Y).

```

Example — Hamiltonian cycles

- ▶ A Hamiltonian cycle: a *closed* path that visits all vertices of the graph exactly once.
- ▶ Input: a graph
 - ▶ $\text{vtx}(a), \dots$
 - ▶ $\text{edge}(a,b), \dots$
 - ▶ $\text{initialvtx}(a_0)$, for some vertex a_0

Hamiltonian cycles — cont'd

- ▶ Candidate answer sets: subsets of edges.
- ▶ Generator:
 $\{ \text{hc}(X,Y) \} \text{ :- edge}(X,Y).$
- ▶ Stable models of the generator given a graph:
 - ▶ input graph +
 - ▶ a subset of the ground facts $\text{hc}(a,b)$ for which there is an input fact $\text{edge}(a,b).$

Hamiltonian cycles — cont'd

- ▶ Tester (ii):
Every vertex is reachable from a given initial vertex through chosen $\text{hc}(v,u)$ edges:
 $\text{:- vtx}(X), \text{ not } r(X).$
 $r(Y) \text{ :- hc}(X,Y), \text{ edge}(X,Y), \text{ initialvtx}(X).$
 $r(Y) \text{ :- hc}(X,Y), \text{ edge}(X,Y), r(X).$
- ▶ Only Hamiltonian cycles pass the tests (i–ii).

Hamiltonian cycles — cont'd

- ▶ Tester (i):
Each vertex has at most one chosen incoming edge and one outgoing edge.
 $\text{:-hc}(X,Y), \text{ hc}(X,Z), \text{ edge}(X,Y), \text{ edge}(X,Z), Y \neq Z.$
 $\text{:-hc}(Y,X), \text{ hc}(Z,X), \text{ edge}(Y,X), \text{ edge}(Z,X), Y \neq Z.$
- ▶ Only subsets of chosen edges $\text{hc}(v,u)$ forming paths (possibly closed) pass the test.

Hamiltonian cycles — cont'd

- ▶ Given:
 - ▶ the graph, the generator rule, and the tester rules (i–ii)Hamiltonian cycles and stable models correspond.
- ▶ A Hamiltonian cycle: atoms $\text{hc}(v,u)$ in a stable model.

Hamiltonian cycles — cont'd

- ▶ Cardinality constraints enable an even more compact encoding.

- ▶ Tester (i) using 2 variables:

```
: - 2 { hc(X,Y) : edge(X,Y) }, vtx(X).
```

```
: - 2 { hc(X,Y) : edge(X,Y) }, vtx(Y).
```

Planning — cont'd

- ▶ Planning is PSPACE-complete.
- ▶ Planning with:
 - ▶ deterministic operators
 - ▶ complete knowledge about the initial situation, and with
 - ▶ an upper bound on the length of the planis NP-complete.

Example: planning

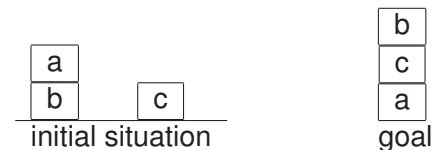
- ▶ Given:

- ▶ a set of operators
- ▶ initial situation and goal

- ▶ find a sequence of operator instances leading from initial to goal situation.

Block-world planning

```
(operator moveop  
  (params (<X> OBJECT) (<Y> OBJECT) )  
  (preconds (clear <X>) (clear <Y>))  
  (effects (on <X> <Y>) (clear <X>)))
```



solution:
moveop(a,table,0),
moveop(c,a,1),
moveop(b,c,2)

Mapping planning to rules

- ▶ Devise a logic program such that stable models correspond to plans:
 - ▶ of length at most n
 - ▶ that are valid
 - ▶ and that reach the goal

Planning — cont'd

- ▶ Preliminaries
 - ▶ Add to each predicate a situation argument
 - ▶ $\text{on}(X,Y,T)$: X is on Y in T
 - ▶ $\text{moveop}(X,Y,T)$: X is moved onto Y in T
 - ▶ Length bound n : $\text{time}(0..n)$.
 - ▶ $\text{nextstate}(Y,X) \text{ :- time}(X), \text{time}(Y),$
 $Y = X + 1.$

Mapping planning to rules

- ▶ Candidate answer sets: valid execution sequences (of length $\leq n$) of operator instances from the initial conditions.
- ▶ Tester: eliminates those sequences that do not reach the goal.

Planning — cont'd

- ▶ Available blocks: $\text{block}(a).$
 $\text{block}(b).$
 $\text{block}(c).$
- ▶ Initial conditions: $\text{on}(a,b,0).$
 $\text{on}(b,\text{table},0).$
 $\text{on}(c,\text{table},0).$

Planning — cont'd

- ▶ Auxiliary concepts make encoding easier.
- ▶ Rules make it straightforward to define auxiliary predicates:

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

Planning — cont'd

- ▶ Generator: execution sequences of operators.
- ▶ An operator **can** be applied if 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).
```

Planning — cont'd

- ▶ Further predicates:

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

Planning — cont'd

- ▶ Operator effects:

```
on(X,Y,T2) :- block(X), object(Y),  
    nextstate(T2,T1),  
    moveop(X,Y,T1).
```

Planning — cont'd

- ▶ Frame axioms (as rules with exceptions):

```
on(X,Y,T2) :- block(X), object(Y),
             nextstate(T2,T1),
             on(X,Y,T1),
             not moving(X,T1).
% the exceptions
moving(X,T) :- time(T), block(X), object(Y),
              moveop(X,Y,T).
```

Planning — cont'd

- ▶ Blocking conditions for moveop
(no concurrent actions):

```
:- 2 { moveop(X,Y,T):block(X):object(Y) },
   time(T).
```

Planning — cont'd

- ▶ In addition, rules for blocking conflicting operator instances are needed.
- ▶ This set depends on how much concurrency in the search of a plan is allowed.
- ▶ Computationally advantageous to allow concurrency to decrease search space explosion due to interleavings of independent operators.

Planning — cont'd

- ▶ Blocking conditions for moveop
(with concurrent actions) I-II:

```
% 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).
```

Planning — cont'd

- ▶ Blocking conditions for `moveop` (with concurrent actions) III:
% No two blocks moved onto the same block
`:- 2 { moveop(X,Y,T):block(X) },
 block(Y), time(T).`

Planning — cont'd

- ▶ Plans correspond to stable models:
 - ▶ there is a stable model iff there is a valid sequence of moves that leads to goal and can be executed concurrently in at most n steps.
- ▶ A valid plan
 - ▶ facts `moveop(x,y,t)` in a model ordered by the argument t where facts with the same t can be taken in any linear order.

Planning — cont'd

- ▶ Tester: excludes models where the goal has not been reached.

```
:- not goal.  
goal :- time(T), goal(T).  
goal(T2) :- nextstate(T2,T1), goal(T1).  
% Actual goal conditions  
goal(T) :- time(T),  
          on(b,c,T),  
          on(c,a,T).
```

Planning — cont'd

- ▶ Easy to add optimizations:
% Stop when the goal has been reached
`:- block(X), object(Y), time(T),
 moveop(X,Y,T),
 goal(T).`

- ▶ Further optimizations (pruning rules):

```
% No move from table to table
```

```
:- block(X), time(T),  
    moveop(X,table,T), on(X,table,T).
```

```
% No move on something and then to table
```

```
:- nextstate(T2,T1), block(X), object(Y),  
    moveop(X,Y,T1), moveop(X,table,T2).
```

Part IV

ASP Solver Technology

- ▶ SAT, CSP, (M)IP

- ▶ Similarities: search for models (assignments to variables) satisfying a set of constraints.
- ▶ Differences: no logical variables, fixed points, database, DDB or KR techniques available, search space given by variable domains.

- ▶ LP, CLP:

- ▶ Similarities: database and DDB techniques.
- ▶ Differences: Search for proofs (not models), non-declarative features.

ASP Solvers

- ▶ ASP solvers need to handle two challenging tasks
 - ▶ complex data
 - ▶ search
- ▶ The approach has been to use
 - ▶ **logic programming and deductive data base techniques** for the former
 - ▶ **SAT/CSP related search techniques** for the latter
- ▶ In the current systems: separation of concerns
 - ▶  A two level architecture

Architecture of ASP Solvers

Typically a two level architecture employed

- ▶ **Grounding** step handles complex data:
 - ▶ Given program P with variables, generate a set of ground instances of the rules which preserves the models.
 - ▶ LP and DDB techniques employed.
- ▶ **Model search** for ground programs:
 - ▶ Special-purpose search procedures
 - ▶ Exploiting SAT/SMT solver technology

Model Search

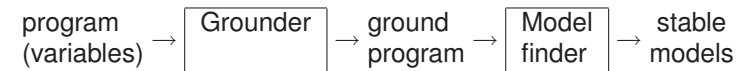
There are two successful approaches to model computing for ground programs

- ▶ Special purpose search procedures exploiting the particular properties of stable model semantics
- ▶ Translating the stable model finding problem to a propositional satisfiability problem exploiting state of the art SAT solvers



These approaches are **closely related** via (Clark's) program **completion**

Typical ASP System Tool Chain



- ▶ **Grounder:**
 - ▶ (deductive) DB techniques
 - ▶ built-in predicates/functions (e.g. arithmetic)
 - ▶ function symbols
- ▶ **Model finder:**
 - ▶ SAT technology (propagation, conflict driven clause learning)
 - ▶ Special propagation rules for recursive rules
 - ▶ Support for cardinality and weight constraints and optimization built-in

Program Completion

- ▶ Program completion $\text{comp}(P)$: a simple translation of a logic program P to a propositional formula.

Example.

$P :$	$\text{comp}(P) :$
$a \leftarrow b, \text{not } c$	$a \leftrightarrow ((b \wedge \neg c) \vee (\neg b \wedge d))$
$a \leftarrow \text{not } b, d$	$\neg b, \neg c, \neg d$
$\leftarrow a, \text{not } d$	$\neg(a \wedge \neg d)$

- ▶ **Supported models** of a logic program and **propositional models** of its completion coincide.
- ▶ For **tight programs** (no positive recursion) **supported and stable models** coincide (Fages).

Program Completion — cont'd

- ▶ Stable models for tight programs can be computed using a SAT solver:
 - ▶ Form the completion and transform that to CNF (typically with new atoms).
 - ▶ Run a SAT solver on the CNF and translate results back.
- ▶ For tight (normal) programs, unit propagation on the translated CNF and ASP propagation on the original program coincide.

Program Completion — cont'd

- ▶ For non-tight programs (with positive recursion), stable models of a program and propositional models of its completion do not coincide.

- ▶ **Example.**

$p \leftarrow q$		$p \leftrightarrow q$
$q \leftarrow p$	vs	$q \leftrightarrow p$
unique stable model: $\{\}$		2 models: $\{\}, \{p, q\}$

Translations to SAT

- ▶ Translating non-tight LPs to SAT is challenging
 - ▶ Modular translations not possible (Niemelä, 1999)
 - ▶ Without new atoms exponential blow-up (Lifschitz and Razborov, 2006)
- ▶ There are one pass translations to SAT
 - ▶ Polynomial size (Ben-Eliyahu & Dechter 1994; Lin & Zhao 2003)
 - ▶ $O(\|P\| \times \log |At(P)|)$ size (Janhunen 2004)
- ▶ Also incremental translations to SAT have been developed extending the completion dynamically with **loop formulas** (Lin & Zhao 2002)
 - ▶  Assat and Cmodels model finders

Translations to SMT

- ▶ Recently a compact linear size one pass translation to SMT/ **difference logic** has been devised.
 - ▶  LP2DIFF (Janhunen & Niemelä 2009).
- ▶ Difference logic = propositional logic + linear difference constraint of the form

$$x_i + k \geq x_j \text{ (or equivalently } x_j - x_i \leq k)$$

where k is an arbitrary integer constant and x_i, x_j are integer valued variables).

- ▶ Practically all major SMT solvers support difference logic
 - ▶  Most SMT solvers can be used as ASP model finders without modifications.

SAT and ASP

- ▶ ASP systems have much more expressive modelling languages than SAT: variables, built-ins, aggregates, optimization
- ▶ For model finding for ground normal programs results carry over: efficient unit propagation techniques, conflict driven learning, backjumping, restarting, ...
- ▶ ASP model finders have special (unfounded set based) propagation rules for recursive rules
- ▶ ASP model finders have **built-in support for aggregates** (cardinality and weight constraints) and optimization
- ▶ One pass compact translations to SAT and SMT available: progress in SAT and SMT solver technology can also be exploited directly in ASP model finding.

Some ASP Systems

Grounders:

dlv <http://www.dbai.tuwien.ac.at/proj/dlv/>
gringo <http://potassco.sourceforge.net/>
lparse <http://www.tcs.hut.fi/Software/smodels/>
XASP with XSB <http://xsb.sourceforge.net>

Model finders (disjunctive programs):

claspD <http://potassco.sourceforge.net/>
dlv <http://www.dbai.tuwien.ac.at/proj/dlv/>
GnT <http://www.tcs.hut.fi/Software/gnt/>

Part V

Further Information: Systems, Applications, Literature

Some ASP Systems

Model finders (non-disjunctive programs):

ASSAT <http://assat.cs.ust.hk/>
clasp <http://potassco.sourceforge.net/>
CModels <http://userweb.cs.utexas.edu/users/tag/cmodels/>
LP2DIFF <http://www.tcs.hut.fi/Software/lp2diff/>
LP2SAT <http://www.tcs.hut.fi/Software/lp2sat/>
Smodels <http://www.tcs.hut.fi/Software/smodels/>
SUP <http://userweb.cs.utexas.edu/users/tag/sup/>

- ▶ For systems, performance, benchmarks, and examples, see for instance the latest **ASP competition**:
<http://dtai.cs.kuleuven.be/events/ASP-competition/>

Applications

- ▶ Planning
For example, USAdvisor project at Texas Tech:
A decision support system for the flight controllers of space shuttles
- ▶ Product configuration
–Intelligent software configurator for Debian/Linux
–WeCoTin project (Web Configuration Technology)
–Spin-off (<http://www.variantum.com/>)
- ▶ Computer-aided verification
–Partial order methods
–Bounded model checking

Some Literature

- ▶ C. Baral. Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, 2003.
- ▶ V. Lifschitz. Foundations of Logic Programming.
<http://userweb.cs.utexas.edu/users/vl/mypapers/flp.ps>
- ▶ V. Lifschitz. Introduction to Answer Set Programming.
<http://userweb.cs.utexas.edu/users/vl/mypapers/esslli.ps>
- ▶ T. Eiter, G. Ianni, and T. Krennwallner. A Primer on Answer Set Programming. <http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-asg.pdf>

Applications—cont'd

- ▶ Data and Information Integration
- ▶ Semantic web reasoning
- ▶ VLSI routing, planning, combinatorial problems, network management, network security, security protocol analysis, linguistics . . .
- ▶ WASP Showcase Collection
<http://www.kr.tuwien.ac.at/research/projects/WASP/showcase.html>
- ▶ Applying ASP
 - ▶ as a stand alone system
 - ▶ as an embedded solver

Conclusions

ASP = KR + DB + search

- ▶ ASP emerging as a viable KR tool
- ▶ Efficient implementations under development
- ▶ Expanding functionality and ease of use
- ▶ Growing range of applications

Topics for Further Research

- ▶ Intelligent grounding
- ▶ Model computation without full grounding
- ▶ Program transformations, optimizations
- ▶ Model search
- ▶ Distributed and parallel implementation techniques
- ▶ Language extensions
- ▶ Programming methodology
- ▶ Testing techniques
- ▶ Tool support: debuggers, IDEs