# The Implementation of Answer Set Solvers

Ilkka Niemelä

Ilkka.Niemela@hut.fi, http://www.tcs.hut.fi/~ini/

Laboratory for Theoretical Computer Science
Helsinki University of Technology



# Contents





# 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 the **answer sets** of the theory.



# ASP—cont'd

- Solving a problem using ASP

  Problem instance $\longrightarrow$ $\boxed{\text{Encoding}}$ $\xrightarrow{\text{Theory}}$ $\boxed{\text{ASP solver}}$ $\xrightarrow{\text{Models}}$ (Solutions)

- 

| Possible formal system | Models |
|---|---|
| Propositional logic | Truth assignments |
| CSP | Variable assignments |
| Logic programs | Stable models |

# 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 $v \in V$: | For each edge $(v,u) \in E$: |
|---|---|
| $v(1) \vee v(2) \vee v(3)$ | $\neg v(1) \vee \neg u(1)$ |
| $\neg v(1) \vee \neg v(2)$ | $\neg v(2) \vee \neg u(2)$ |
| $\neg v(1) \vee \neg v(3)$ | $\neg v(3) \vee \neg u(3)$ |
| $\neg v(2) \vee \neg v(3)$ | |

- 3-colorings of a graph $(V,E)$ and models of the encoding correspond:
  vertex $v$ colored with color $i$ iff $v(i)$ true in the model.



# What is ASP Good for?

Search problems (with substantial amount of data):

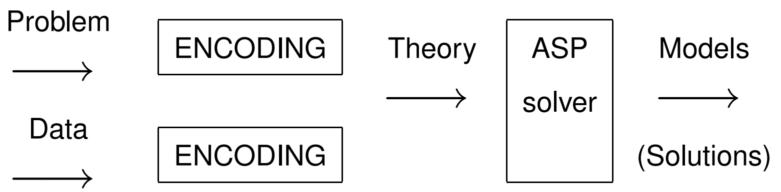
- Constraint satisfaction
- Planning, routing
- Computer-aided verification
- Security analysis
- Product configuration
- Combinatorics
- Diagnosis

☞ Declarative problem solving



# Towards ASP in Practice

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






# ASP Using Logic Programs

- Logic programming: framework for merging KR, DB, and search
- PROLOG style logic programming systems not directly suitable for ASP:
  - they search proofs (not models) and produce answer substitutions
  - they are 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

# Example. $3$-coloring

**Problem**: $clrd(V,1) \leftarrow \text{not } clrd(V,2), \text{not } clrd(V,3), vtx(V)$
$clrd(V,2) \leftarrow \text{not } clrd(V,1), \text{not } clrd(V,3), vtx(V)$
$clrd(V,3) \leftarrow \text{not } clrd(V,1), \text{not } clrd(V,2), vtx(V)$
$\leftarrow edge(V,U), clrd(V,C), clrd(U,C)$

**Data**: $vtx(v) \quad vtx(u) \quad \ldots$
$edge(v,u) \quad edge(u,w) \quad \ldots$

☞ 3-colorings and stable models of the encoding correspond: $v$ colored $i$ iff $clrd(v,i)$ in the model.



# LPs with Stable Models Semantics

- Consider normal logic program rules

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

- Seen as constraints on an answer set (stable model):
  - if $B_1, \ldots, B_m$ are in the set and
  - none of $C_1, \ldots, C_n$ 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.



# Stable Models — cont'd

- Program:
  $b \leftarrow$
  $f \leftarrow b, \text{not } eb$
  $eb \leftarrow p$

  Stable model:
  $\{b,f\}$
- For example, $\{b, eb\}$ 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].



# 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 with unique least model $LM(P^S)$
- **$S$ is a stable model of $P$ iff $S = LM(P^S)$.**

## Example. Stable models

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

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



## Stable Models — cont'd

- A program can have **none**, one, or **multiple** stable models. For example, program $P$

$$\begin{array}{lcl} p_1 \leftarrow \text{not } q_1 & \dots & p_n \leftarrow \text{not } q_n \\ q_1 \leftarrow \text{not } p_1 & \dots & q_n \leftarrow \text{not } p_n \end{array}$$

has $2^n$ stable models:
$\{p_1, p_2, \dots, p_n\}, \{q_1, p_2, \dots, p_n\}, \dots, \{q_1, q_2, \dots, q_n\}$.

- Constraints often used:

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

(shorthand for $f \leftarrow B_1, \dots, \text{not } C_n, \text{not } f$)



## Variables and Functions

- Variables needed for uniform encodings

Program:

$$\begin{array}{l} clrd(V,1) \leftarrow \text{not } clrd(V,2), \text{not } clrd(V,3), vtx(V) \\ clrd(V,2) \leftarrow \text{not } clrd(V,1), \text{not } clrd(V,3), vtx(V) \\ clrd(V,3) \leftarrow \text{not } clrd(V,1), \text{not } clrd(V,2), vtx(V) \\ \leftarrow edge(V,U), clrd(V,C), clrd(U,C) \end{array}$$

Data:

$$\begin{array}{lll} vtx(v) & vtx(u) & \dots \\ edge(v,u) & edge(u,w) & \dots \end{array}$$



## Variables and Functions — cont'd

- Semantics: Herbrand models
- A rule is seen as a shorthand for the set of its ground instantiations.

**Example.**

$$clrd(V,1) \leftarrow \text{not } clrd(V,2), \text{not } clrd(V,3), vtx(V)$$

is a shorthand for

$$\begin{array}{l} clrd(v,1) \leftarrow \text{not } clrd(v,2), \text{not } clrd(v,3), vtx(v) \\ clrd(u,1) \leftarrow \text{not } clrd(u,2), \text{not } clrd(u,3), vtx(u) \\ clrd(1,1) \leftarrow \text{not } clrd(1,2), \text{not } clrd(1,3), vtx(1) \\ \dots \end{array}$$

## Stable Models — cont'd

- A stratified program has a unique stable model (coincides with the canonical model given by e.g. the well-founded semantics).
- 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 solution to every NP search problem.



# Implementing ASP Solvers for Normal Programs



## 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.
  `SMODELS` (`lparse`), `dlv`, `XASP`, `aspps`
- **Model search** for ground programs:
  - Special-purpose search procedures
    `SMODELS`, `dlv`, `NoMoRe`, `aspps`
  - Translation to SAT
    `CMODELS`, `ASSAT`

# Grounding

- Stable model semantics is highly undecidable if arbitrary function symbols are allowed.
- Approaches to guaranteeing decidability:
  - Disallow function symbols and use **safe** rules where every variable in a rule appears also in a positive body literal in the rule (`dlv`)
  - Domain-restricted programs (`SMODELS/lparse`)
  - Finitary programs [Bonatti, AIJ 2004]
- LP and DDB techniques used [ICLP04: Cumbo/Faber/Greco/Leone]



# Domain-Restricted Programs

- A domain-restricted program has two parts:
  - domain predicate definitions (stratified set of rules)
  - other rules
- **Domain-restricted** rules: each variable in a rule appears also in a positive domain predicate in the body of the rule
- Also the stratified rules defining domain predicates are required to be domain-restricted: each variable in a rule must appear also in a positive domain predicate defined on an earlier stratum ($\omega$-stratification [Syrjänen, LPNMR'2001])



# Domain-Restriction — cont'd

- Function symbols can be allowed; built-ins easy to add.
- Rules defining domain predicates have a unique stable model $M_{\text{dom}}$ which can be computed by deductive database techniques.
- $M_{\text{dom}}$ is a subset of every stable model of the program.
- Ground instances of other rules can be generated using database techniques (join evaluation): Only those ground instances need to be taken where the domain predicates in the body hold (are in $M_{\text{dom}}$).



# Example. Domain-restriction

$$
\begin{array}{l}
col(1);\ col(2);\ col(3);\ r(u,v);\ r(u,w)\ldots \\
d(U) \leftarrow r(V,U) \\
tc_r(V,U) \leftarrow r(V,U) \\
tc_r(V,U) \leftarrow r(V,Z), tc_r(Z,U), d(U) \\
edge(t(V),t(U)) \leftarrow tc_r(V,U), \text{not } tc_r(U,U), \text{not } tc_r(V,V) \\
vtx(V) \leftarrow edge(V,U);\ \ vtx(U) \leftarrow edge(V,U) \\
clrd(V,1) \leftarrow \text{not } clrd(V,2), \text{not } clrd(V,3), vtx(V) \\
clrd(V,2) \leftarrow \text{not } clrd(V,1), \text{not } clrd(V,3), vtx(V) \\
clrd(V,3) \leftarrow \text{not } clrd(V,1), \text{not } clrd(V,2), vtx(V) \\
\leftarrow edge(V,U), col(C), clrd(V,C), clrd(U,C)
\end{array}
$$

- All predicates except $clrd/2$ are domain predicates.

# Example. Grounding

- Suppose that $vtx(V)$ is a domain predicate and the unique stable model $M_{\mathrm{dom}}$ for the rules defining domain predicates contains $vtx(v_1), \ldots, vtx(v_n)$
- Then for a rule

  $$clrd(V,1) \leftarrow \mathrm{not}\ clrd(V,2), \mathrm{not}\ clrd(V,3), vtx(V)$$

  grounding produces

  $$clrd(v_1,1) \leftarrow \mathrm{not}\ clrd(v_1,2), \mathrm{not}\ clrd(v_1,3), vtx(v_1)$$
  $$\ldots$$
  $$clrd(v_n,1) \leftarrow \mathrm{not}\ clrd(v_n,2), \mathrm{not}\ clrd(v_n,3), vtx(v_n)$$



# Model Search

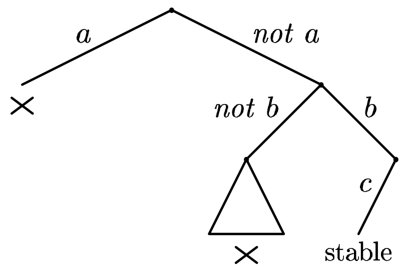
Two promising approaches to model computing for ground programs

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



# Special Purpose Search Procedures

- A typical approach (`SMODELS`, `dlv`):
  - Backtracking search building a binary search tree
  - choice points on atoms
- Similar to Davis-Putman-Logemann-Loveland (DPLL) procedure for SAT






# Model Search — cont'd

- To prune the search space:
  - choices are made one at a time (**heuristics**)
  - before making a new choice, the set of remaining choice points is pruned and conflicts are detected (**propagation rules**)
  - Each node in the search tree corresponds to a model candidate, i.e., a set of literals derivable by the propagation rules from current choices

# Propagation Rules

- Given a model candidate $A$ (say with a new choice added), propagation rules extend $A$.
- Typical idea is to use (a slight generalization of) the well-founded semantics (WFS)
- For efficiency reasons this is implemented using propagation rules which are based on local information (as far as possible).
- Propagation rules do not eliminate stable models: If literal $l$ is derivable by a propagation rule from $A$, then $l$ holds in every stable model compatible with $A$.



# Propagation Rules — cont'd

- To reach WFS the **Upper Bound** rule can be used: **Given** a model candidate $A$, if an atom $p \notin LM(P')$ **derive** not $p$ where $P'$ is a program obtained by deleting negative body literals from the rules in $P$ whose body is not false under $A$.

**Example.** Consider a model candidate $A = \{q_2\}$

$$
\begin{array}{llll}
P: & p_2 \leftarrow p_1, \text{not } q_1 & P': & p_2 \leftarrow p_1 \\
 & p_1 \leftarrow p_2, \text{not } q_1 & & p_1 \leftarrow p_2 \\
 & p_2 \leftarrow \text{not } q_2 & &
\end{array}
$$

As $LM(P') = \{\}$, we get not $p_1$, not $p_2$, not $q_1$, not $q_2$.



# Local Propagation Rules

| Rule | $P$ | $A$ | Derived |
|---|---|---|---|
| Fitting-1 | $q \leftarrow p_1, \text{not } p_2$ | $p_1, \text{not } p_2$ | $q$ |
| Fitting-2 | $q \leftarrow p_1, \text{not } p_2$<br>$q \leftarrow p_3, \text{not } p_4$ | $p_2, \text{not } p_3$ | not $q$ |
| BP-1 | $q \leftarrow p_1, \text{not } p_2$ | $q$ | $p_1, \text{not } p_2$ |
| BP-2 | $q \leftarrow p_1, \text{not } p_2$ | not $q, p_1$ | $p_2$ |

☞ These rules can be implemented efficiently (linear time, local steps) but do not quite capture WFS.



# Computing Models

- Each branch of the search tree where propagation rules do not derive new literals and where there are no conflicts (both $a$ and not $a$ derived) provides a stable model (the positive atoms in the branch).

Program:

$p \leftarrow \text{not } q, \text{not } u$

$q \leftarrow \text{not } p, \text{not } u$

$p \leftarrow q$

Search tree:

not $u$ (F-2)

↙ ↘

| | | | |
|---|---|---|---|
| not $p$ | (Choice) | $p$ | (Choice) |
| $q$ | (F-1/BP-2) | not $q$ | (F-2) |
| $p$ | (F-1) | | |
| $\times$ | | stable model : $\{p\}$ | |

# Propagation Rules — cont'd

- BP-1 and Upper bound have no counter part in SAT checkers
- Denote by $\mathbf{expand}(P,A)$ the closure of $A$ under all the rules (Fitting-1 — Upper Bound)
- $\mathbf{expand}(P,A)$ generalizes the well-founded model
- In fact $\mathbf{expand}(P,\emptyset)$ is the well-founded model of $P$

For further details see e.g.:
J. Dix, U. Furbach, and I. Niemelä. Nonmonotonic reasoning: Towards efficient calculi and implementations. In *Handbook of Automated Reasoning, Volume II*, chapter 19, pages 1241–1354. Elsevier Science, Amsterdam, 2001.



# Propagation Rules — cont'd

- $\mathbf{expand}(P,A)$ can often be computed in linear time by postponing the application of Upper Bound and then doing it as locally as possible (Simons, 2000).
- But in the worst case quadratic time:
  - closure of (Fitting-1 – BP-2) in linear time
  - Upper Bound in linear time
- $\mathbf{expand}$ is the key procedure in `SMODELS`



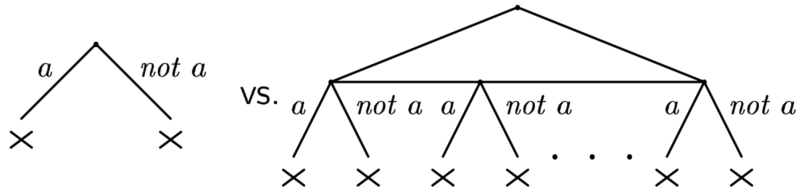
# Lookahead

- `SMODELS` uses an even stronger propagation rule
- **Lookahead** rule
  - Given program $P$ and literals $A$:
    - if for a literal $l$, $\mathbf{expand}(P,A\cup\{l\})$ contains a conflict ($p$ and $\mathrm{not}\ p$ for some atom $p$),
    - then derive the complement of $l$
- Lookahead goes beyond WFS



# Lookahead — cont'd

- Lookahead can yield a smaller search space:




- Full lookahead can be computationally expensive.
  See [ICLP04: You/Huo]
  However, substantial optimization possible (Simons, 2000)

# Search Heuristics

- Typical approaches to selecting choice points:
  - Maximum Occurrences in rules of Minimum size (MoM's) heuristics
  - Unit propagation counting
  - Preferring choice points with the smallest combined remaining search space (`SMODELS`)
- Conflict clause learning (Ward, Schlipf, 2004)
  - Heuristics based on conflict counting (Chaff) or activity counting (BerkMin).
  - Backjumping
  - Restarts



# Heuristics in `SMODELS`

- Computed while doing lookahead
- For each undetermined atom $p$, two lookaheads:
  $A_p = \textbf{expand}(P, A \cup \{p\})$
  $A_{\text{not } p} = \textbf{expand}(P, A \cup \{\text{not } p\})$
- If $p$ is the next choice point, an upper bound on the remaining search space is $E(p) = 2^u + 2^{u'}$ where $u$ ($u'$) is the number of undetermined atoms in $A_p$ ($A_{\text{not } p}$)
- Heuristics: choose an atom $p$ with the smallest estimated remaining search space $E(p)$



# Translations to SAT

- The goal is to exploit the efficiency of SAT solvers for answer set computation

$$\begin{array}{ccc}
\boxed{\text{Normal program } P} & & \text{answer set} \\
\downarrow & & \\
\boxed{\text{Set of clauses } S} & & \uparrow \\
\downarrow & & \\
\boxed{\text{SAT solver}} & \longrightarrow & \text{model}
\end{array}$$



# Translations to SAT — cont'd

- Translating LPs to SAT is non-trivial
  - Modular translations cannot be done (Niemelä)
  - Without new atoms exponential blow-up (Lifschitz and Razborov)
  - One-to-one correspondence between propositional models and answer sets challenging
- Approaches
  - Using program completion for handling a substantial fragment (Fages; Erdem, Lifschitz)
  - Extend completion with loop formulae dynamically (Lin, Zhao; ...)
  - Compiling LPs to SAT (Ben-Eliyahu; Janhunen)

# Program Completion

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

  **Example.**

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

- Supported models of a logic program and propositional models of its completion coincide.
- For tight programs supported and stable models coincide (Fages).



# Program Completion — cont'd

- Stable models for tight programs can be computed using a SAT solver (CMODELS):
  - Form the completion and transform that to CNF (typically with new atoms).
  - Run a SAT solver on the CNF and translate results back.
- Does not handle programs with positive loops

$$\begin{array}{lcl} a \leftarrow b & \text{vs.} & a \leftrightarrow b \\ b \leftarrow a & & b \leftrightarrow a \end{array}$$



# Loop Formulae

- Positive dependency graph $G(P)$ for a program $P$:
  - vertices are the atoms in $P$;
  - there is an edge $(p,q)$ if there is a rule with $p$ as the head and $q$ as a positive body literal.
- Loop is a strongly connected subgraph of $G(P)$.
- For a loop $L$ define loop formula

$$LF(L,P) = \bigvee_{p \in L} p \rightarrow \bigvee_{r \in R^-(L)} B_r$$

where $B_r$ is the body of a rule $r$ and $R^-(L)$ is the set of rules with the head in $L$ but the positive body literals not.



# Loop Formulae — cont'd

- Intuition: if at least one atom in a loop $L$ is in a stable model, then there must be an atom $q$ in $L$ with a rule whose positive body literals are not in the loop.
- Theorem: $S$ is a stable model of $P$ iff it is a model of $\mathrm{comp}(P) \cup LF$ where $LF$ is the set of loop formulae for the loops is $G(P)$.
- There can be exponentially many loops
- An incremental approach is possible (ASSAT)

## ASSAT

$T := \text{comp}(P)$;
repeat
  Find a model $M$ of $T$;
  if there is no then return false
  else if $M$ is a stable model of $P$ then return $M$
  else
    compute a loop $L$ such that $M \not\models LF(L,P)$;
    $T := T \cup \{LF(L,P)\}$
  endif

- Loops $L$ above can be found efficiently
- In the worst case exp. many loop formulae added
- If a stable model exists, often good performance; if not, a potential problem



## Compilation to SAT

- Goal: Translating a logic program to a set of clauses such that there is a one-to-one correspondence between stable models and propositional models
- The basic idea: the program completion is augmented with clauses capturing the justifiability property
- Using a level numbering technique a $O(\|P\| \times \log|At(P)|)$ translation available (Janhunen, ECAI'2004)



# Extensions to Normal Programs



## Extensions

- **Classical negation**
  Can be handled by normal programs (renaming):
  $p \leftarrow \text{not } \neg p$ corresponds to $p \leftarrow \text{not } p'$
  $\leftarrow p, p'$
- **Disjunctions** in the head
  - Higher complexity ($\Sigma_2^p/\Pi_2^p$-complete) and expressivity (all problems in $\Sigma_2^p$)
  - Special purpose implementation available (`dlv`)
  - Can be implemented also using an ASP solver for normal programs as the **core engine** (`GnT`)

# Extensions — cont'd

- Many extensions can be handled by translation to standard logic programs and then using a standard solver as the core inference engine. See, for example
  - nested programs
    [ICLP04: Linke/Tompits/Woltran]
  - preferences
    [ICLP04: Nieuwenborgh/Heymans/Vermeir]
  - circumscription
    [ICLP04: Wakaki/Inoue]



# Extensions — cont'd

- Aggregates
  - Cardinality constraints
    Example: choose 2–4 hard disks
  - Weight constraints
    Example: the total capacity of the chosen hard disks must be at least 20 GB.
  - Built-in support for aggregates in the search procedures (`SMODELS`, `dlv`)
  - External implementation integrated
    [ICLP04: Elkabani/Pontelli/Son]



# Extensions — cont'd

- Optimization
  Example: prefer the cheapest set of hard disks
  (Built-in support in `SMODELS`)
- Weak constraints with weight and priority levels

$$:\sim B_1, \ldots, B_m, \mathrm{not}\; C_1, \ldots, \mathrm{not}\; C_n [w : l]$$

  (Built-in support in `dlv`)



# Example. Rules in SMODELS

- Cardinality constraints:
  $2\ \{hd_1, \ldots, hd_n\}\ 4$
- Weight constraints
  (sum of capacities at least 20GB):
  $20\ [hd_1 = 6, \ldots, hd_n = 13]$
- Optimization
  (a model with the cheapest set of hard disks):
  $\mathit{minimize}\,[hd_1 = 100, \ldots, hd_n = 600]$

# 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
% Each paper is assigned 3 reviewers
3 { assigned(P,R):reviewer(R) } 3 :- paper(P).
% No paper assigned to a reviewer with coi
:- assigned(P,R), coi(R,P).
```



# Review Assignment — cont'd

```
% 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) ].
```



# SMODELS system

(`http://www.tcs.hut.fi/Software/smodels`)

program (variables) → [`lparse` front-end] → program in normal form → [SMODELS search] → stable models

- Front-end: (deductive) DB techniques for stratified programs
- Special purpose search engine:
  - array data structures (Dowling-Gallier type)
  - local computations for large rule sets
  - linear space requirements
  - optimization built-in



# Other ASP Implementations

| | |
|---|---|
| `dlv` | `http://www.dbai.tuwien.ac.at/proj/dlv/` |
| `GnT` | `http://www.tcs.hut.fi/Software/gnt/` |
| `CMODELS` | `http://www.cs.utexas.edu/users/tag/cmodels.html` |
| `ASSAT` | `http://assat.cs.ust.hk/` |
| `NoMoRe` | `http://www.cs.uni-potsdam.de/~linke/nomore/` |
| `XASP` | distributed with XSB v2.6 |
| | `http://xsb.sourceforge.net` |
| `aspps` | `http://www.cs.engr.uky.edu/ai/aspps/` |
| `ccalc` | `http://www.cs.utexas.edu/users/tag/cc/` |

## Applications

- Planning
  USAdvisor project at Texas Tech:
  A decision support system for the flight controllers of space shuttles
  [ICLP04: Gelfond]
- Product configuration
  –Intelligent software configurator for Debian/Linux
  –WeCoTin project (Web Configuration Technology)
- Computer-aided verification
  –Partial order methods
  –Bounded model checking



## Applications—cont'd

- VLSI routing
  [ICLP04: Erdem/Wong]
- Planning
  [ICLP04: Jia/You/Yuan]
- Combinatorial problems, network management, network security, security protocol analysis, linguistics, . . .
- C. Baral. Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, 2003.



## Conclusions

**ASP = KR + DB + search**

- ASP emerges as a viable KR tool
- Efficient implementations under development
  (`Smodels`, `aspps`, `dlv`, `XASP`, `CMODELS`, `ASSAT`, . . .)
- Expanding functionality and ease of use
- Growing range of applications



## Topics for further research

- Intelligent grounding
- Model computation without full grounding
- Program transformations, optimizations
  [ICLP04: Pearce]
- Learning, restarting, backjumping
- Heuristics
- Local search techniques
- Language extensions
- Programming methodology
- Tool support