# Stable Model Semantics: From Theory to Implementations and Applications

**Ilkka Niemelä**

**Helsinki University of Technology**

**Dept. of Computer Science and Engineering**

**Finland**

`Ilkka.Niemela@hut.fi`

`http://www.tcs.hut.fi/~ini/`



## Overview

- ➤ Answer set programming
- ➤ Stable model semantics
- ➤ Implementing stable models
- ➤ Using ASP
- ➤ Applications
- ➤ Experimental results

Young's law:

All great discoveries are made by mistake.



## Introduction

- ➤ WANTED: an approach to modeling and solving AI problems (planning, route finding, puzzles, combinatorics, . . . )
  - Diverse domains
  - Constraints
  - Incomplete information
  - Frame problem
  - Spatial and temporal reasoning

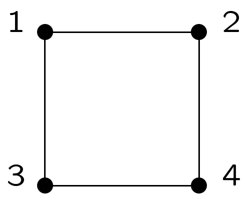


## A Solution

- ➤ Answer Set Programming (ASP) paradigm
- ➤ Stable model semantics
- ➤ Basic idea:
  - Encode problem (specification + instance) as logic program rules
  - The solutions given by the stable model of the rules

## Example — 3-colorability

- Instance: a graph $(V, E)$
- Problem specification: assign each vertex one of three colors (`r`,`b`,`g`) such that no two vertices in $V$ connected by an edge in $E$ have the same color.
- Encoding $\mathbf{E}(V, E)$ of an instance $(V, E)$:




$$vtx(1) \leftarrow \qquad edge(1,2) \leftarrow$$
$$vtx(2) \leftarrow \qquad edge(2,4) \leftarrow$$
$$vtx(3) \leftarrow \qquad edge(1,3) \leftarrow$$
$$vtx(4) \leftarrow \qquad edge(3,4) \leftarrow$$



## 3COL — cont'd

- Encoding $\mathbf{E}(3COL)$ of problem specification:

$$clr(r) \leftarrow$$
$$clr(b) \leftarrow$$
$$clr(g) \leftarrow$$
$$clrd(V,r) \leftarrow vtx(V), \mathbf{not}\ clrd(V,b), \mathbf{not}\ clrd(V,g)$$
$$clrd(V,g) \leftarrow vtx(V), \mathbf{not}\ clrd(V,b), \mathbf{not}\ clrd(V,r)$$
$$clrd(V,b) \leftarrow vtx(V), \mathbf{not}\ clrd(V,r), \mathbf{not}\ clrd(V,g)$$
$$\leftarrow clr(C), edge(V,U), clrd(V,C), clrd(U,C)$$



## 3COL — cont'd

- Stable models of $\mathbf{E}(V, E) \cup \mathbf{E}(3COL)$ and 3-colorings of $(V, E)$ correspond.
- A stable model of $\mathbf{E}(V, E) \cup \mathbf{E}(3COL)$:
  - input graph $\mathbf{E}(V, E) \cup \{clr(g), clr(b), clr(r)\}$ +
  - coloring $\{clrd(1,b), clrd(2,g), clrd(3,r), clrd(4,b)\}$
- Solving 3COL reduced to finding a stable model of the program $\mathbf{E}(V, E) \cup \mathbf{E}(3COL)$



## ASP — roots

- Knowledge Representation
  - Commonsense reasoning; defeasible reasoning
  - Nonmonotonic logics (default logic, autoepistemic logic, circumscription)
  - Elaboration tolerance
  - Representing the frame problem
- Declarative programming
  - Logic programming

## Answer set programming

- Coined by Vladimir Lifschitz
- Based on a formal system that assigns for each theory a collection of (answer) sets.
- To solve a problem $P$ for an instance $I$:
  - encode $I$ as facts $\mathbf{E}(I)$
  - encode the specification of $P$ as a theory $\mathbf{E}(P)$
  - give a (pol. time) function that extracts a solution to $P$ for $I$ from an answer set of $\mathbf{E}(I) \cup \mathbf{E}(P)$.



## ASP — cont'd

- Examples of ASP systems:

| Formal system | Answer sets |
|---|---|
| Normal logic programs | Stable models |
| Disjunctive logic programs | Stable models |
| Extended programs | Answer sets |
| Default logic | Extensions |
| Classical logic | Standard models |
| Constraint satisfaction | Variable assignments |



## ASP based on stable models

- Extends CSP/SAT and logic programming
  - CSP/SAT + structured representation + default negation
  - Logic programming + choice constructs
- Particularly suitable for applications with grounded sequences of choices:

  planning, product configuration, verification of distributed systems, ...



## Stable models vs. SAT

- Rules with exceptions:

  $fly(X) \leftarrow bird(X), \mathbf{not}\ exc\_bird(X)$
  $exc\_bird(X) \leftarrow pinguin(X)$
  $exc\_bird(X) \leftarrow no\_wings(X)$
  $exc\_bird(X) \leftarrow bird(X), footinconcrete(X)$
  ...

- Frame axioms:

  $on(X, Y, T_2) \leftarrow$
  $\quad nextstate(T_2, T_1), on(X, Y, T_1), \mathbf{not}\ moving(X, T_1)$
  $moving(X, T) \leftarrow moveop(X, Y, T)$
  ...

## ASP vs. logic programming

- Logic programming
  - Single intended model
  - Extensive use of recursion

```
append([],X,X).
append([X|Y],Z,[X|T]) :- append(Y,Z,T).
```

- ASP based on stable models
  - Several intended models
  - Rules as constraints
  - Limited use of recursion

$clrd(V,r) \leftarrow vtx(V), \mathbf{not}\ clrd(V,b), \mathbf{not}\ clrd(V,g)$



## Overview

- Answer set programming
- **Stable model semantics**
- Implementing stable models
- Using ASP
- Applications
- Experimental results

Peer's law:
The solution to a problem changes the nature of the problem.



## Stable model semantics

- Consider normal logic program rules

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

- Seen as a constraint 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



## Stable models — cont'd

- Program:

  $b \leftarrow$

  $f \leftarrow b, \mathbf{not}\ eb$

  $eb \leftarrow p$

  $eb \leftarrow nw$

  Answer set:

  $\{b, f\}$

- An answer set should be *justified* by the rules
- $\{b, eb\}$ is not a proper answer set:
  $eb$ cannot be justified

# Stable models

- The propositional (variable free) case first
- $P$ — ground program
- $S$ — set of ground atoms
- Justifiability captured by reduct
- Reduct $P^S$ (Gelfond-Lifschitz)
  - delete each rule having a body literal $\mathbf{not}\ C$ with $C \in S$
  - remove all negative body literals from the remaining rules



# Reduct

| Model | Program | $\mapsto$ | Reduct |
|---|---|---|---|
| $\{b, f\}$ | $b \leftarrow$ | | $b \leftarrow$ |
| | $f \leftarrow b, \mathbf{not}\ eb$ | | $f \leftarrow b$ |
| | $eb \leftarrow p$ | | $eb \leftarrow p$ |
| | $eb \leftarrow nw$ | | $eb \leftarrow nw$ |
| $\{b, eb\}$ | $b \leftarrow$ | | $b \leftarrow$ |
| | $f \leftarrow b, \mathbf{not}\ eb$ | | |
| | $eb \leftarrow p$ | | $eb \leftarrow p$ |
| | $eb \leftarrow nw$ | | $eb \leftarrow nw$ |



# Reduct — cont'd

- Reduct $P^S$: potentially applicable rules given $S$
- $P^S$ is a definite program with unique least model $LM(P^S)$
- Least model: atoms deducible by applicable rules
- Rules: $p \leftarrow$ Least model: $p$ (by $p \leftarrow$)

  $q \leftarrow p$ $q$ (by $q \leftarrow p$)

  $r \leftarrow p, q$ $r$ (by $r \leftarrow p, q$)

  $s \leftarrow r, s$



# Stable models

- $LM(P^S)$: atoms justifiable given the $S$-applicable rules $P^S$
- $S$ is a stable model of $P$ if $S$ is
  - sound: everything in $S$ is justifiable by the $S$-applicable rules ($S \subseteq LM(P^S)$)
  - complete: everything justifiable by the $S$-applicable rules is in $S$ ($LM(P^S) \subseteq S$)
- $S$ is a stable model of $P$ iff $S = LM(P^S)$

## Stable models — cont'd

| Model | Program | $\mapsto$ | Reduct | $\mapsto$ | LM |
|---|---|---|---|---|---|
| $\{b, f\}$ | $b \leftarrow$ | | $b \leftarrow$ | | $\{b, f\}$ |
| | $f \leftarrow b, \textbf{not}\ eb$ | | $f \leftarrow b$ | | |
| | $eb \leftarrow p$ | | $eb \leftarrow p$ | | |
| | $eb \leftarrow nw$ | | $eb \leftarrow nw$ | | |
| $\{b, eb\}$ | $b \leftarrow$ | | $b \leftarrow$ | | $\{b\}$ |
| | $f \leftarrow b, \textbf{not}\ eb$ | | | | |
| | $eb \leftarrow p$ | | $eb \leftarrow p$ | | |
| | $eb \leftarrow nw$ | | $eb \leftarrow nw$ | | |



## Programs with variables

- Herbrand models only
- Rule with variables — shorthand for all its ground instances
- Rule
$$bd(X) \leftarrow eng(X) \wedge \textbf{not}\ wd(X)$$
represents
$$bd(k) \leftarrow eng(k) \wedge \textbf{not}\ wd(k)$$
$$bd(l) \leftarrow eng(l) \wedge \textbf{not}\ wd(l)$$
$$\ldots$$



## Programs with variables — cont'd

| Program $P$: | $ground(P)$: |
|---|---|
| $bd(X) \leftarrow eng(X) \wedge \textbf{not}\ wd(X)$ | $bd(k) \leftarrow eng(k) \wedge \textbf{not}\ wd(k)$ |
| | $bd(l) \leftarrow eng(l) \wedge \textbf{not}\ wd(l)$ |
| $wd(X) \leftarrow art(X) \wedge \textbf{not}\ bd(X)$ | $wd(k) \leftarrow art(k) \wedge \textbf{not}\ bd(k)$ |
| | $wd(l) \leftarrow art(l) \wedge \textbf{not}\ bd(l)$ |
| $eng(k) \leftarrow;\ eng(l) \leftarrow;\ art(l) \leftarrow$ | $eng(k) \leftarrow;\ eng(l) \leftarrow;\ art(l) \leftarrow$ |
| | Stable models: |
| | $\{eng(k), eng(l), art(l), bd(k), wd(l)\}$ |
| | $\{eng(k), eng(l), art(l), bd(k), bd(l)\}$ |



## Stable models — properties

- Program can have none, one, or multiple stable models.
- Stable models are closely related to JTMS, default logic and autoepistemic logic.
- It is NP-complete to decide whether a ground program has a stable model.
- Normal programs capture all problems in NP.

## Generalizations

- Classical negation
- Can be handled by normal programs (renaming):

$p \leftarrow \textbf{not}\ \neg p$ corresponds to $p \leftarrow \textbf{not}\ p'$

$\leftarrow p, p'$

- Disjunctions in the head
- Higher complexity ($\Sigma_2^p/\Pi_2^p$-complete) and expressivity (all problems in $\Sigma_2^p$)



## Overview

- Answer set programming
- Stable model semantics
- **Implementing stable models**
- Using ASP
- Applications
- Experimental results

Fifth law of unreliability:
To err is human, but to really foul things up requires a computer.



## Implementing stable models

- Typically a two phase technique employed
  - **Grounding**:
    Given program $P$ with variables, generate a set of ground instances of the rules (a subset of $ground(P)$) which preserves the stable models.
  - **Stable models** for ground programs:
    Davis-Putnam type backtracking search
- Used in: Smodels, `dlv`, dcs, ...



## Smodels

- Helsinki University of Technology
- Developed since 1995 by Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen
- Available under GPL:
  `http://www.tcs.hut.fi/Software/smodels/`
- Implemented in C++

## Smodels — cont'd

- Handles domain-restricted function free normal programs with built-ins and extensions
- Design goals
  - Linear space complexity for ground programs
  - Coping with large programs (100k+ rules)
  - Complete method
- Two modules:
  - `lparse` (grounding)
  - `smodels` (stable models for ground programs)



## Rules with variables

- Function symbols are not allowed (decidability)
- Built-in (e.g. arithmetic) functions supported
- Rules domain-restricted:

  For each variable a positive domain predicate in the body

  `even(X+1) :- d(X), not even(X).`



## Grounding

- Evaluation of domain predicates
- Generation of ground instances compatible with domain predicates
- Suppose that `clr(C)` and `edge(X,Y)` are domain predicates that have been evaluated
- For

  `:- clr(C), edge(X,Y), clrd(X,C), clrd(Y,C).`

  grounding produces all ground rules

  `:- clr(r), edge(a,b), clrd(a,r), clrd(b,r).`

  for which `clr(r)` and `edge(a,b)` hold

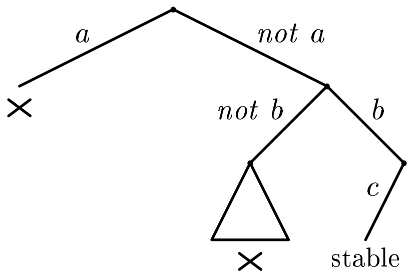


## Grounding — cont'd

- Domain predicates:
  - in Smodels: non-recursive (database views)
  - In `dlv`: range-restricted programs evaluated using deductive database techniques

## Search in `smodels`

➤ Stable model computation:
- Bottom-up backtracking search building a binary search tree
- Choice points on atoms






## Search in `smodels`

➤ To prune the search space:
- choices are made one at a time (heuristic)
- before making a new choice, the set of remaining choice points is pruned and conflicts are detected (propagation rules)

➤ Low level implementation using Dowling-Gallier type data structures



## Propagation rules

➤ Similar to unit propagation:
Given $q \leftarrow p_1, \mathbf{not}\ p_2$ and $\{p_1, \mathbf{not}\ p_2\}$ derive $q$
Given $q \leftarrow p_1, \mathbf{not}\ p_2$ and $\{\mathbf{not}\ q, p_1\}$ derive $p_2$

➤ Stronger propagation also used:
- Well-founded semantics
- Lookahead



## 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)

## Smodels extensions

- Choices over subsets
- `{ a1,..., an } :- b, not c.`
  - If the body holds, then any subset of `{a1,...,an}` (including empty) can be derived
- Choice rules



## Choice rules — an example

- Rule `{ a } :- b.`
  - Corresponds to
    ```
    a :- not na, b.
    na :- not a.
    ```
  - One stable model: `{}`
- Program:
  ```
  { a } :- b.
  b.
  ```
  - Two stable models: `{b}`, `{b,a}`



## Further extensions — cont'd

- Cardinality constraints
- $l \; \{ \; p_1, \ldots, p_n \; \} \; u$
  - satisfied in a model $S$ if the number of the literals from $\{p_1, \ldots, p_n\}$ satisfied in $S$ is between integers $l$ and $u$ (inclusive)
- Example: vertex cover of size less than $12$:

```
1 { vi,vj } 2 .   % for all edges (vi,vj)
:- 12 { v1,...,vn } .
```



## Further extensions — cont'd

- Weight constraints: `l [ p1=w1,...,pn=wn ] u`
- Optimization: `minimize [ p1=w1,...,pn=wn ] .`
- Type definitions (conditions):

  `1 { clrd(V,C):clr(C) } 1 :- vtx(V).`
- Two ways of using Smodels:

  (i) Stand-alone

  (ii) Embedded through API (as a C++ library)

## Overview

➤ Answer set programming

➤ Stable model semantics

➤ Implementing stable models

➤ **Using ASP**

➤ Applications

➤ Experimental results

Tenenbaum's law of replicability:

The most interesting things happen only once.



## Using answer set programming

➤ A basic methodology: **generate and test**

- Generator: provides candidate answer sets (odd loops/disjunctions/choice constructs)
- Tester: eliminates non-valid candidates (integrity constraints + LP techniques)



## Example: satisfiability

➤ Generator

- Candidate answer sets: propositional models, i.e., sets of atoms
- For each atom `a`, rule `{ a }.`
- For program / $2^n$ stable models:

| For program | $2^n$ stable models: |
|---|---|
| `{ a1 }.` | `{ }` |
| ... | ... |
| `{ an }.` | `{ a1,...,an }` |



## Satisfiability — cont'd

➤ Consider general formulae (not in CNF): $(a \vee \neg b) \wedge (\neg a \leftrightarrow b)$

➤ Tester — evaluates a formula $q$ recursively

➤ For each subformula of $q$:

- 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

```
:- not q.
```

# Satisfiability — cont'd

- Tester encoding

| Subformula $p$ | Rules |
|---|---|
| $l_1 \land \cdots \land l_n$ | $p \leftarrow p_{l_1}, \ldots, p_{l_n}$ |
| $l_1 \lor \cdots \lor l_n$ | $p \leftarrow p_{l_1}$<br>$\ldots$<br>$p \leftarrow p_{l_n}$ |
| $\neg l$ | $p \leftarrow \textbf{not } p_l$ |
| $l_1 \leftrightarrow l_2$ | $p \leftarrow p_{l_1}, p_{l_2}$<br>$p \leftarrow \textbf{not } p_{l_1}, \textbf{not } p_{l_2}$ |



# Satisfiability — cont'd

- Consider formula $p_1$: $\underbrace{(a \lor \neg b)}_{p_2} \land \underbrace{(\neg a \leftrightarrow b)}_{p_3}$
- Program:

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

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



# Overview

- Answer set programming
- Stable model semantics
- Implementing stable models
- Using ASP
- **Applications**
- Experimental results

William and Holland's law:
If enough data is collected, anything may be proven by statistical methods.



# Applications

- Planning

  Dimopoulos, Nebel, Koehler: Encoding Planning Problems in Nonmonotonic Logic Programs

  Lifschitz: Answer Set Planning
- Model checking

  Liu, Ramakrishnan, and Smolka: Fully Local and Efficient Evaluation of Alternating Fixed Points

## Applications — cont'd

- Reachability analysis

  Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets

  See, the `mcsmodels` tool:
  `http://www.tcs.hut.fi/~kepa/tools/`

  Varpaaniemi: Stable Models for Stubborn Sets



## Applications — cont'd

- Product configuration

  Soininen and Niemelä: Developing a declarative rule language for applications in product configuration

  Syrjänen: A Linux configurator
  (Thu 5.30; 340 Huxley)

- Dynamic Constraint Satisfaction

  Soininen and Niemelä and Gelle: A Fixpoint Definition of Dynamic Constraint Satisfaction



## Applications — cont'd

- Feature interaction

  Accorsi, Areces and Maarten: Towards Feature Interaction via Stable Models

- Logical cryptanalysis

  Hietalahti, Massacci, and Niemelä: DES: a Challenge Problem for Nonmonotonic Reasoning Systems



## Applications — cont'd

- Network inhibition analysis

  Aura, Bishop, Sniegowski: Analyzing single-server network inhibition

- Computing (total and partial) stable models for disjunctive programs

  Janhunen, Niemelä, Simons, and You: Unfolding Partiality and Disjunctions in Stable Model Semantics

## Overview

- ➤ Answer set programming
- ➤ Stable model semantics
- ➤ Implementing stable models
- ➤ Using ASP
- ➤ Applications
- ➤ **Experimental results**

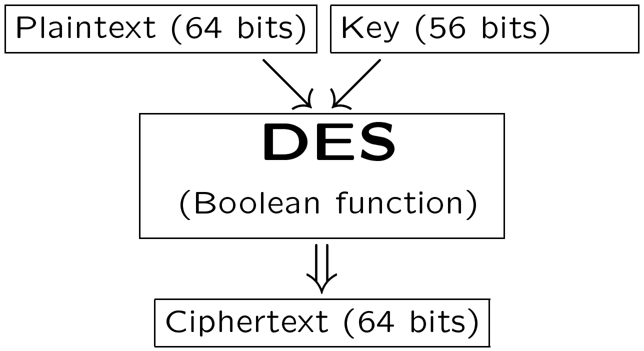
Fett's law of the lab:

Never replicate a successful experiment.



## DES

- ➤ DES (Boolean function) encoded directly as rules






## DES — cont'd

- ➤ Experimenting with key search problems
- ➤ Known plaintext attack:
  given plaintext and ciphertext find a possible key for a limited version of DES (1–3 rounds)
- ➤ Smodels handles 3 rounds



## DES — cont'd

(Massacci, 1999):

- ➤ State-of-the art SAT-checkers (SATO/ntab/walksat) using an optimized encoding: 2 rounds
- ➤ A tuned SAT-checker (rel_sat based) using an optimized encoding: 3 rounds

# Hamiltonian cycles

- A Hamiltonian cycle: a closed path that visits all vertices of the graph exactly once
- Hamiltonian cycles are challenging for SAT solvers as no compact SAT encodings (with linear number of atoms) are known



# Hamiltonian cycles — cont'd

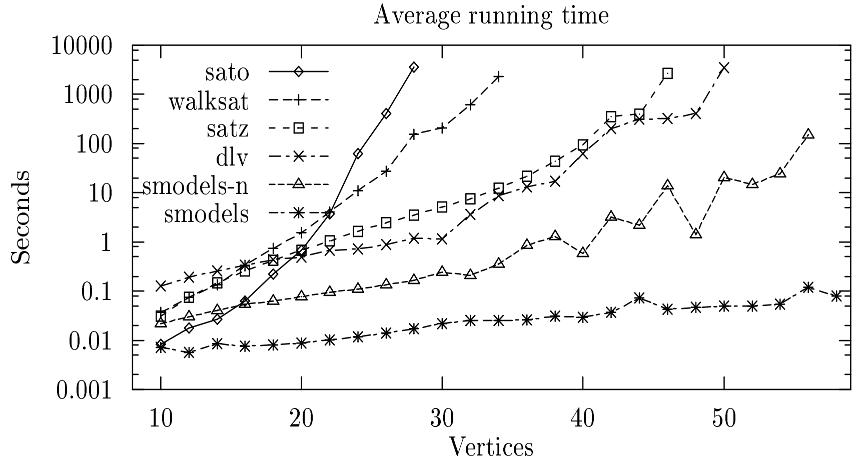
- Instance $(V, E)$:

```
vtx(v).
edge(v,u).        % one fact for each {v,u} in E
initialvtx(v0). %for some vertex v0
```

- 
```
% Generator - for each vertex two edges chosen
2 { hc(V,U):edge(V,U),
    hc(W,V):edge(W,V) } 2 :- vtx(V).
% Tester - every vertex is visited
:- vertex(V), not r(V).
r(V) :- initialvtx(V).
r(V) :- hc(V,U), edge(V,U), r(U).
r(V) :- hc(U,V), edge(U,V), r(U).
```

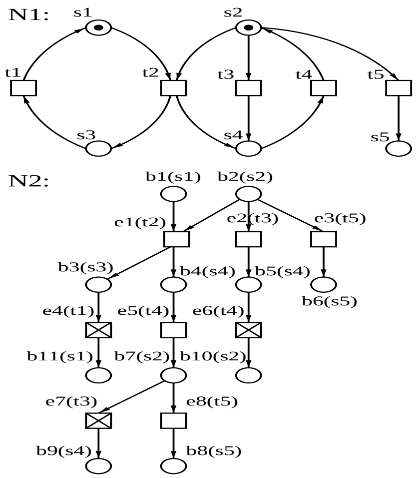


# Hamiltonian cycles — cont'd






# Reachability analysis




$P_D(N2)$ :

$e_1 \leftarrow not\,(be_1)$
$be_1 \leftarrow not\,(e_1)$
$e_2 \leftarrow not\,(be_2)$
$be_2 \leftarrow not\,(e_2)$
$e_3 \leftarrow not\,(be_3)$
$be_3 \leftarrow not\,(e_3)$
$e_5 \leftarrow e_1, not\,(be_5)$
$be_5 \leftarrow not\,(e_5)$
$e_8 \leftarrow e_5, not\,(be_8)$
$be_8 \leftarrow not\,(e_8)$
$\leftarrow 2\{e_1, e_2, e_3\}$
$b_1 \leftarrow not\,(e_1)$
$b_2 \leftarrow not\,(e_1), not\,(e_2), not\,(e_3)$
$b_3 \leftarrow e_1$
$b_4 \leftarrow e_1, not\,(e_5)$
$b_5 \leftarrow e_2$
$b_7 \leftarrow e_5, not\,(e_8)$
$\leftarrow b_1, b_2$
$\leftarrow b_2$
$\leftarrow b_3$
$\leftarrow b_4$
$\leftarrow b_5$
$\leftarrow b_7$

**Reachability analysis — cont'd**

| Problem(size) | $\lvert B\rvert$ | $\lvert E\rvert$ | #c | DL | CP | Unf$^1$ | $\mathrm{DC}^2_{\mathrm{MIP}}$ | $\mathrm{DC}^1_{\mathrm{McM}}$ | $\mathrm{DC}^1_{\mathrm{smo}}$ |
|---|---|---|---|---|---|---|---|---|---|
| DME(4) | 2381 | 652 | 16 | N | 0 | 1.1 | 216.1 | 1.4 | 3.4 |
| DME(5) | 4096 | 1145 | 25 | N | 0 | 3.2 | 1968.3 | 5.5 | 11.8 |
| DME(6) | 6451 | 1830 | 36 | N | 0 | 8.5 | 13678.3 | 20.1 | 32.4 |
| DME(7) | 9542 | 2737 | 49 | N | 0 | 18.1 | - | 66.1 | 75.3 |
| DME(8) | 13465 | 3896 | 64 | N | 0 | 37.0 | - | 196.0 | 166.5 |
| DME(9) | 18316 | 5337 | 81 | N | 0 | 70.0 | - | 542.2 | 321.4 |
| DME(10) | 24191 | 7090 | 100 | N | 0 | 124.0 | - | 1268.4 | 586.6 |
| DME(11) | 31186 | 9185 | 121 | N | 0 | 207.0 | - | 3070.9 | 988.8 |
| SYNC(2) | 4007 | 2162 | 490 | N | 0 | 4.6 | 171.6 | 37.0 | 1.5 |
| SYNC(3) | 29132 | 15974 | 5381 | N | 0 | 218.6 | 11985.0 | 14073.3 | 56.4 |

Measured running times in seconds:

$^1$ = Pentium 166MHz, 64MB RAM, Linux 2.0.35.

$^2$ = Sparcstation 20/712, 96MB RAM



# Blocks world planning

| Blocks world problem | steps | Ground rules | Time (s) | |
|---|---|---|---|---|
| large.c | 8 | 19 311 | first plan | 2.3 |
| (15 blocks) | 7 | 17 087 | no plan | 0.4 |
| large.d | 9 | 27 359 | first plan | 4.4 |
| (17 blocks) | 8 | 24 533 | no plan | 0.6 |
| large.e | 10 | 37 379 | first plan | 6.7 |
| (19 blocks) | 9 | 33 879 | no plan | 0.8 |



# Other ASP systems

➤ `dlv` (disjunctive logic programs)
`http://www.dbai.tuwien.ac.at/proj/dlv/`

➤ dcs (Datalog with constraints)
`http://www.cs.engr.uky.edu/dcs/`

➤ LDL$^{++}$ (stratified programs with choice constructs and aggregates)
`http://www.cs.ucla.edu/ldl/`

➤ CCALC (tight programs)
`http://www.cs.utexas.edu/users/tag/cc/`



# Conclusions

- ASP emerges as a viable KR tool
- Efficient implementations (Smodels, `dlv`, dcs)
- Expanding functionality and ease of use
- Growing range of applications
- Use it!

Felson's law:
To steal ideas from one person is plagiarism;
to steal from many is science.