

Knowledge Representation

Chapter 3. Relational Representation and Reasoning

Pedro Cabalar

Dept. Computer Science
University of Corunna, SPAIN

March 10, 2021

1 Deductive Databases

2 Answer Set Programming

3 Actions and change

4 Diagnosis

Relational Representation

- **Atoms** = instead of propositions, we have now **predicates**. They represent **relations among entities**:

```
neighbour (france, spain) .  
exports (germany, france, cars) .
```

- **Herbrand Domain** = set of individuals, each one uniquely identified by a (lowercase) constant name. E.g.
 $D = \{\text{germany, france, spain, cars, ...}\}.$
- **Unique Names Assumption (UNA)** =
different terms represent different individuals.
 $\text{spain} \neq \text{france}, \text{spain} \neq \text{cars}, \text{spain} \neq \text{españa}$
- We can use unary predicates to represent **types**:

```
country(spain). country(france). country(germany).  
tradegood(cars). tradegood(food).
```

```
country(spain; france; germany).  
tradegood(cars; food).
```

Relational Representation

- A set of facts becomes the **extensional database (EDB)**!

```
neighbour(spain, france) .  
neighbour(france, germany) .  
exports(spain, germany, food) .  
exports(spain, france, food) .  
exports(germany, france, cars) .  
exports(france, spain, cars) .
```

Table neighbour

C1	C2
spain	france
france	germany

Table exports

FROM	TO	GOOD
spain	germany	food
spain	france	food
germany	france	cars
france	spain	cars

Relational Representation

- A query to the EDB becomes a rule with **variables**.
Variable = name with upcase initial (**X**, **Y**, **Country**, ...) universally quantified and denoting arbitrary individuals.
'_' = **anonymous** variable (different each time it occurs)

```
exgood(G) :- exports(_,_,G). exgood(G) :- exports(X1,X2,G).  
 $\forall X1, X2, G (exports(X1, X2, G) \rightarrow exgood(G))$ 
```

- Ex.: “neighbours of France and goods she imports from them”

```
answer(N,G) :- neighbour(france,N), exports(N,france,G).
```

SQL equivalent is more verbose

```
SELECT neighbour.C2, exports.GOOD FROM neighbour  
INNER JOIN exports ON neighbour.C2=exports.FROM  
WHERE neighbour.C1=france AND exports.TO=france;
```

Problem: we get no goods from Spain using our previous data!
We had `neighbour(spain, france)` but not the opposite!

Deductive Databases

- Predicate `neighbour` should be symmetric! We add a rule

```
neighbour(X, Y) :- neighbour(Y, X) .
```

- **Deductive database**: some predicates are **intensional** or **(partially) deduced from rules**, rather than extensional (list of facts).
- **Ground atom** = predicate + constants, **no variables**.
Grounding = replacing variables by **all** their possible instances.
(although it is actually more intelligent than that)

Example: the grounding of program

```
neighbour(spain,france) . neighbour(france,germany) .  
neighbour(X,Y) :- neighbour(Y,X) .
```

would **potentially** yield the rules

```
neighbour(spain,france) . neighbour(france,germany) .  
neighbour(spain,france)      :- neighbour(france,spain) .  
neighbour(spain,germany)     :- neighbour(germany,spain) .  
neighbour(france,spain)      :- neighbour(spain,france) .  
neighbour(france,germany)    :- neighbour(germany,france) .  
neighbour(germany,spain)     :- neighbour(spain,germany) .  
neighbour(germany,france)    :- neighbour(france,germany) .
```

Example: the grounding of program

```
neighbour(spain,france) . neighbour(france,germany) .  
neighbour(X,Y) :- neighbour(Y,X) .
```

would **potentially** yield the rules, but **in practice** ...

```
neighbour(spain,france) . neighbour(france,germany) .  
neighbour(spain,france)      :- neighbour(france,spain) .  
neighbour(spain,germany)     :- neighbour(germany,spain) .  
neighbour(france,spain)      :- neighbour(spain,france) .  
neighbour(france,germany)    :- neighbour(germany,france) .  
neighbour(germany,spain)     :- neighbour(spain,germany) .  
neighbour(germany,france)    :- neighbour(france,germany) .
```


Example: the grounding of program

```
neighbour(spain,france) . neighbour(france,germany) .  
neighbour(X,Y) :- neighbour(Y,X) .
```

would **potentially** yield the rules, but **in practice** ...

```
neighbour(spain,france) . neighbour(france,germany) .
```

```
neighbour(france,spain) .
```

```
neighbour(germany,france) .
```

Deductive Databases

- **Datalog**: deductive database paradigm using normal logic programs (under **stratified negation**) with predicates and variables.

👍 Remember: stratified implies a **unique stable model**.

- Datalog is **more expressive than SQL**, but less expressive than logic programs without the stratification limitation.
- It allows, for instance, defining **recursive relations**, such as:

```
connected(X,Y) :- neighbour(X,Y).  
connected(X,Z) :- neighbour(X,Y), connected(Y,Z).
```

so that we would get `connected(spain, germany)` even though they are not neighbours.

- Bodies can add conditions on variables $X \neq Z$, $X > Z * (Y + 1)$, etc.

```
connected(X,Z) :- neighbour(X,Y), connected(Y,Z), X != Z.
```

Deductive Databases

- **Domain independence**: answers shouldn't change if we just augment the Herbrand Domain

```
switch(1..3).  
p(X,Y) :- X<Y.    % ordered pairs of different switches
```

returns $p(1,2)$, $p(1,3)$, $p(2,3)$ if $D = \{1,2,3\}$
but for $D = \{1,2,3,4\}$ we miss $p(1,4)$, $p(2,4)$, $p(3,4)$.
The set of possible pairs of integers is **infinite**!

```
p(X) :- not switch(X).    % anything that is not a switch
```

The potential D with non-switches is **even worse**!

- All variable occurrences in a rule must be **safe**

Definition (Safety: guarantees domain independence)

A variable is **safe** if it occurs in a non-negated predicate in the body.

```
p(X,Y) :- X<Y, switch(X), switch(Y).  
q(X) :- object(X), not switch(X).    % define valid objects!
```

1 Deductive Databases

2 Answer Set Programming

3 Actions and change

4 Diagnosis

Answer Set Programming

- **Answer Set Programming** (ASP) = we allow normal logic programs (unstratified negation) with predicates and variables.
- In ASP, the stable models are called **answer sets**.
- **Example:**

```
pacifist(X) :- quaker(X), not bellicious(X).  
bellicious(X) :- republican(X), not pacifist(X).  
quaker(nixon). republican(nixon).  
republican(reagan).
```

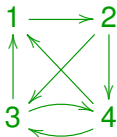
Two answer sets:

```
Answer: 1  
... bellicious(reagan) bellicious(nixon)  
Answer: 2  
... bellicious(reagan) pacifist(nixon)
```

An example: Hamiltonian circuits

Definition (*HAMILT*)

The **Hamiltonian Cycle** problem, *HAMILT*, consists in deciding whether a graph contains a **cyclic path** in a graph that visits each vertex **exactly once**. *HAMILT* is an **NP**-complete problem.



- extensional database `mygraph.gph` with the graph

```
vtx(1). vtx(2). vtx(3). vtx(4).  
edge(1,2). edge(2,3). edge(2,4).  
edge(3,1). edge(3,4). edge(4,3). edge(4,1).
```

- Examples of medium sized graphs (200 nodes, 1250 edges):

[http://www.cs.uky.edu/ai/benchmark-suite/
hamiltonian-cycle.html](http://www.cs.uky.edu/ai/benchmark-suite/hamiltonian-cycle.html)

An example: Hamiltonian circuits

- Predicate `in(X, Y)` points out that an edge $X \rightarrow Y$ is in the cycle. We generate arbitrary choices

```
{in(X, Y)} :- edge(X, Y).
```

- Only one outgoing vertex, only one incoming vertex:

```
:- in(X, Y), in(X, Z), Y!=Z.  
:- in(X, Z), in(Y, Z), X!=Y.
```

- Disregard disconnected cycles. We use `reached(X)` meaning that X can be reached from an arbitrary fixed vertex, say 1.

```
reached(X) :- in(1, X).  
reached(Y) :- reached(X), in(X, Y).
```

and we forbid unreachable vertices:

```
:- vtx(X), not reached(X).
```

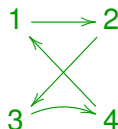
An example: Hamiltonian circuits

- Making the call:

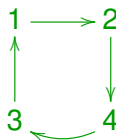
```
clingo 0 hamilt.lp
```

We obtain two answers:

```
Answer: 1  
in(4,3) in(3,1) in(2,4) in(1,2)  
Answer: 2  
in(4,1) in(3,4) in(2,3) in(1,2)  
SATISFIABLE
```



Answer 1



Answer 2

An example: Hamiltonian circuits

- We can split `clingo` in two steps:
`grinder` `gringo` + `propositional solver` `clasp`.
- Download `gringo` from potassco.org and make the call

```
$ gringo hamilt.txt | clasp 0
```

- To display the ground program, try the following

```
$ gringo -t hamilt.txt
...
:-in(1,2),in(1,3).
:-in(1,3),in(1,2).
:-in(2,1),in(2,3).
...
reached(2):-in(1,2).
reached(3):-in(2,3),reached(2).
reached(3):-in(1,3),reached(1).
...
```

"Real world"
(combinatorial)
problem



solutions



ENCODING

DECODING

**Problem
instance
(EDB)**

```
vtx(1). vtx(2). vtx(3). vtx(4).  
edge(1,2). edge(2,3). edge(2,4).  
edge(3,1). edge(3,4). edge(4,3).
```

**Problem
specif.
(KB)**

```
{in(X,Y)} 1:- edge(X,Y).  
:- in(X,Y), in(X,Z), Y!=Z.  
:- in(X,Z), in(Y,Z), X!=Y.  
reached(X) :- in(1,X).  
reached(Y) :- reached(X), in(X,Y).  
:- vtx(X), not reached(X).
```

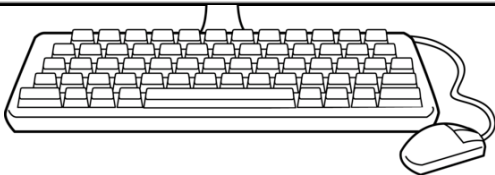
```
$ clingo 0  
mygraph.gph  
hamilt.txt
```

```
% Answer 1  
in(4,3).  
in(3,1).  
in(2,4).  
in(1,2).
```

```
% Answer 2  
in(4,1).  
in(3,4).  
in(2,3).  
in(1,2).
```

answer
sets

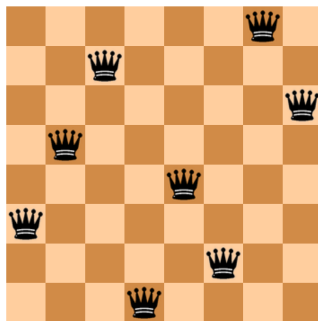
ASP as a problem solving paradigm



ASP vs Prolog

	ASP	Prolog
semantics	several $n \geq 0$ answer sets	unique (canonical) model
problem solving	1 answer set = 1 solution	1 var. instantiation = 1 solution <code>?- graph(G), hamilt(G,X).</code> <code>X=[(4,3),(3,1),(2,4),(1,2)];</code> <code>X=[(4,1),(3,4),(2,3),(1,2)]</code>
computational power	NP -complete	Turing -complete
language type	specification (execution)	programming (flow control: ordering, cut,...)

8 Queens revisited



Example (8-queens problem)

- Arrange 8 queens in a 8×8 chessboard so they do not attack one each other.
- Exercise: [encode](#) the problem in ASP. (Use cardinality atoms).

Explicit negation

- We can sometimes be interested in a second negation, **strong** or **explicit** negation (originally called “classical”). Example:

```
fill :- empty, not fire.
```

risky! we fill when **no information** on fire, but no guarantee.

- We could use auxiliary atom `no_fire` (“I’m sure there is no fire”)

```
fill :- empty, no_fire.  
:- fire, no_fire.  
no_fire :- wet.
```

- **Explicit negation** ‘-’ makes this same effect.

```
fill :- empty, -fire.  
-fire :- wet.
```

and the constraint `:- fire, -fire` is implicit.

Einstein's 5 houses riddle: who keeps fishes as pets?

- 1 The Brit lives in the red house.
- 2 The Swede keeps dogs as pets.
- 3 The Dane drinks tea.
- 4 The green house is on the immediate left of the white house.
- 5 The green house's owner drinks coffee.
- 6 The owner who smokes Pall Mall rears birds.
- 7 The owner of the yellow house smokes Dunhill.
- 8 The owner living in the center house drinks milk.
- 9 The Norwegian lives in the first house.
- 10 The Blends smoker is neighbor of the one who keeps cats.
- 11 The horse keeper is neighbor of the one who smokes Dunhill.
- 12 The owner who smokes Bluemasters drinks beer.
- 13 The German smokes Prince.
- 14 The Norwegian lives next to the blue house.
- 15 The Blends smoker lives next to the one who drinks water.

New features

- **Pooling**: abbreviate several facts in a same atom

```
house(1..5).  
color(red;green;blue;white;yellow).
```

is the same than

```
house(1). house(2). house(3). house(4).house(5).  
color(red). color(green). color(blue).  
color(white). color(yellow).
```

- **Constants**: can be defined in the file

```
#const numhouses=5.  
house(1..numhouses).
```

or passed as arguments in command line

```
$ clingo -c numhouses=5 einstein.txt
```

- **Function symbols as constructors.**

```
owner( person(bill,gates), microsoft ).  
owner( person(jeff,bezos), amazon ).  
owner( company(inditex), zara).  
family(Y) :- owner( person(X,Y), Z).
```


New features

- **Aggregate** = function on **sets** of values.
- We may have #sum, #max, #min, #avg, #count. Example:

```
income(jan,5). income(feb,3).  
income(mar,-2). income(apr,10).  
total(S) :- #sum{X: income(M,X)} = S.
```

- Problem: if we have repeated values, they count once

```
income(may,10). income(jun,10).
```

the set is still {5, 3, -2, 10} and S=16.

- We use tuples (the sum applies to the **first component**):

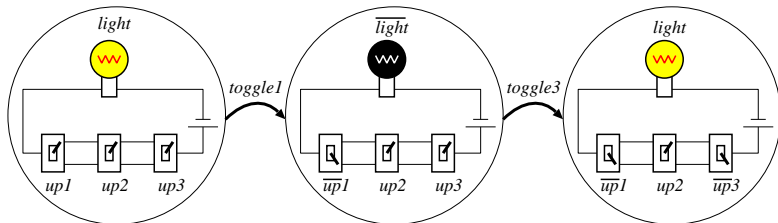
```
total(S) :- #sum{X,M: income(M,X)} = S.
```

```
{X,M: income(M,X)} = {(5,jan), (3,feb), (-2,mar), (10,apr),  
(10,may), (10,jun)}
```

- 1 Deductive Databases
- 2 Answer Set Programming
- 3 Actions and change**
- 4 Diagnosis

Back to our simple example

- Lamp and switches revisited
- **Fluents:** $up1, up2, up3, light$ (Boolean).
- **Actions:** $toggle1, toggle2, toggle3$.
- **State:** a possible configuration of fluent values. Example: $\{\overline{up1}, up2, \overline{up3}, light\}$.
- **Situation:** a moment in time. We can just use $0, 1, 2, \dots$



Reasoning about actions with ASP

- Download system `telingo` (temporal `clingo`)
- We can make groups of rules

```
#program initial. % At timepoint t=0
...
#program dynamic. % Transition from t-1 to t
...
#program always. % Any timepoint t=0..n-1
...
#program final. % Last timepoint t=n-1
...
```

- Predicate names preceded by `'` refer to timepoint $t-1$
- Predicate names preceded by `_` refer to timepoint $t=0$

Reasoning about actions with ASP

```
% File: switches.lp (domain description)
switch(1..3).
action(tog(X)) :- switch(X).

#program dynamic.
% Effect axioms
h(sw(X),up)    :- 'h(sw(X),down), o(tog(X)).
h(sw(X),down)  :- 'h(sw(X),up),    o(tog(X)).
h(light,off)   :- 'h(light,on),    o(tog(_)).
h(light,on)    :- 'h(light,off),   o(tog(_)).
% Executability constraints: none in this case
% Inertia: c(F)= fluent F has changed
h(F,V) :- 'h(F,V), not c(F).
c(F)    :- 'h(F,V), h(F,W), V!=W.

% Action generation
1 { o(A): _action(A) } 1.
```

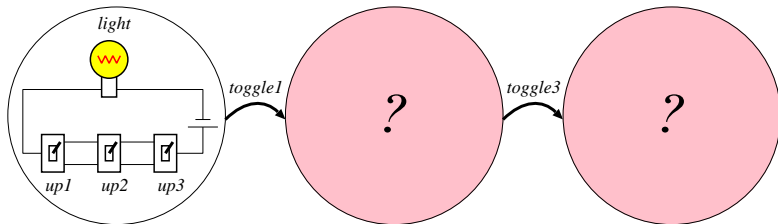
We want to solve some **typical reasoning problems**.

The most usual ones:

- **Simulation** (aka **prediction**, aka **temporal projection**):
run a sequence of actions on an initial state
- **Temporal explanation** (aka **postdiction**):
fill gaps from partial observations
- **Planning**: obtain sequence of actions to reach some goal
- **Diagnosis**: explain unexpected observed results
- **Verification**: check system properties

Prediction (simulation, or temporal projection)

- **Knowing:** initial state + sequence of actions
- **Find out:** final state (alternatively sequence of intermediate states)



Reasoning about actions with ASP

Prediction example

```
% File: switches-predict.lp (instance of prediction problem)
#program initial.
h(light,off).
h(sw(X),up) :- switch(X).
```

We assert a sequence of facts using:

```
% Sequence of performed actions
&tel{
    &true
    ;> o(tog(3))
    ;> o(tog(1))
    ;> o(tog(2))
    ;> o(tog(2))
}.
#show h/2.
#show o/1.
```

where `;>` is a sequence operator

Reasoning about actions with ASP

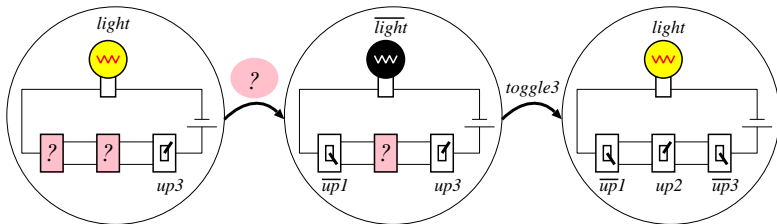
Prediction example

Calling `telingo switches.txt switches-predict.txt`

```
Answer: 1
State 0:
  h(light,off) h(sw(1),up) h(sw(2),up) h(sw(3),up)
State 1:
  o(tog(3))
  h(light,on) h(sw(1),up) h(sw(2),up) h(sw(3),down)
State 2:
  o(tog(1))
  h(light,off) h(sw(1),down) h(sw(2),up) h(sw(3),down)
State 3:
  o(tog(2))
  h(light,on) h(sw(1),down) h(sw(2),down) h(sw(3),down)
State 4:
  o(tog(2))
  h(light,off) h(sw(1),down) h(sw(2),up) h(sw(3),down)
```

Postdiction (or temporal explanation)

- **Knowing:** partial observations of states and performed actions
- **Find out:** **complete information** on states and performed actions



Reasoning about actions with ASP

Postdiction example:

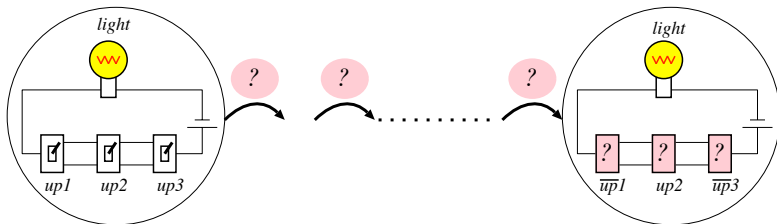
```
% switches-postdict.lp
#program initial.
% Completing unknown facts
1 {h(sw(X),up); h(sw(X),down)} 1 :- switch(X).
1 {h(light,on); h(light,off)} 1.

% Observations: we use a constraint!
:- not &tel{
    h(sw(3),up) & h(light,on)
    ;> h(light,off) & h(sw(1),down) & h(sw(3),up)
    ;> o(tog(3))
}.
```

Calling `telingo 0 switches.txt switches-postdict.txt` we get 4 possible explanations

Planning

- **Knowing:** initial state + **goal** (partial description of final state)
- **Find out:** **plan** (sequence of actions) that guarantees reaching the goal



Reasoning about actions with ASP

Planning example

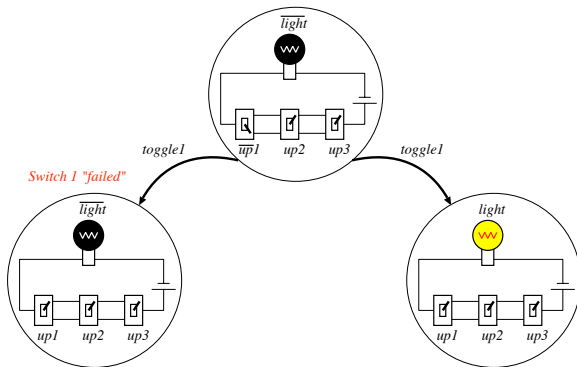
```
% File: switches-plan.lp
#program initial.
h(light,on).
h(sw(X),up) :- switch(X).

#program final.
goal :- h(light,on),h(sw(1),down),
        h(sw(2),up),h(sw(3),down).
:- not goal.
```

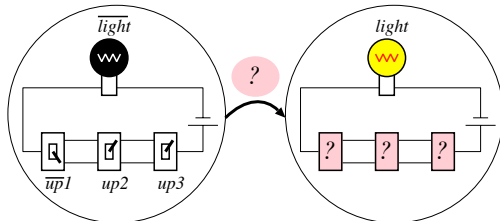
Calling `telingo 0 switches.txt switches-plan.txt` we get two minimal plans of length 2 toggling 1 and 3 or vice versa.

Planning vs Postdiction

- Note that **planning** seems a **type of postdiction**. For deterministic systems, this is true, but ...
- **Nondeterministic** transition system: fixing **current state** + **performed action** \longrightarrow several possible **successor states**.
- For instance, switch 1 up **may fail** to turn the light on...



Planning vs Postdiction



- For **postdiction**, one **valid explanation** is: we performed ***toggle1***, and it succeeded to turn the light on.
- For **planning**, ***toggle1*** is **not a valid plan**: it **does not guarantee** reaching the goal ***light***. Possible plans are ***toggle2*** or ***toggle3***.

- **Knowing:** a model distinguishing between **normal** and **abnormal** transitions + a partial set of observations (usually implying abnormal behavior).
- **Find out:** the **minimal** set of **abnormal transitions** that explains the observations.
- We will see an ASP example later on.
- Similar to postdiction, but we are additionally interested in minimality of explanations.

Exercise

“Elaborating Missionaries and Cannibals Problem” [J. McCarthy]

3 missionaries and 3 cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How shall they cross?



We will use the following **fluents**:

- 1 $n(G, B)$ = is the number of persons of group G at bank B .

Ex.: $h(n(mis, l), 3)$ = “*there are 3 missionaries in the left bank*”

- 2 `boat` points out the boat bank. Ex. $h(boat, l)$ = “*the boat is at left bank*”

Exercise: missionaries and cannibals

We will use **action**:

- $\text{move}(M, C)$ = move M missionaries and C cannibals.
- For simplicity, we include two **action attributes** $\text{moved}(\text{mis}, N)$ and $\text{moved}(\text{can}, N)$ that point out **separately** how many persons of each group are moved.

Exercise: missionaries and cannibals

We begin with types and initial state

```
#program initial.
% Some types
group(mis;can).
bank(l;r).
opposite(l,r). opposite(r,l).
action(move(M,C)) :- M=0..2, C=0..2, M+C<3, M+C>0.

% Initial state
h(n(G,l),3) :- group(G).
h(n(G,r),0) :- group(G).
h(boat,l).
```

Exercise: missionaries and cannibals

Rules for transitions

```
#program dynamic.
% Action generation
1 {o(A) : _action(A) } 1.

% Auxiliary (action attributes)
moved(mis,M) :- o(move(M,C)) .
moved(can,C) :- o(move(M,C)) .

% Executability axioms
:- moved(G,N) , 'h(boat,B) , 'h(n(G,B),M) , N>M.

% Effect axioms (no inertia needed)
h(n(G,B),M+N) :- 'h(n(G,B),M) , h(boat,B) , moved(G,N) .
h(n(G,B),M-N) :- 'h(n(G,B),M) , 'h(boat,B) , moved(G,N) .
h(boat,B1)      :- 'h(boat,B) , _opposite(B,B1) .
```

Inertia not needed because **all fluents are changed**

Exercise: missionaries and cannibals

Rules for transitions

```
#program always.  
% Missionaries not outnumbered by cannibals  
:- h(n(mis,B),M), h(n(can,B),C), C>M, M>0.  
  
#program final.  
:- not goal.  
goal :- h(n(mis,r),3), h(n(can,r),3).  
  
#show o/1. % We only show performed actions
```

- We execute `telingo 0 mc.txt` and it will try length $t = 1, 2, \dots$ until a solution is found.
- Four solutions of length $t = 11$ are eventually found.

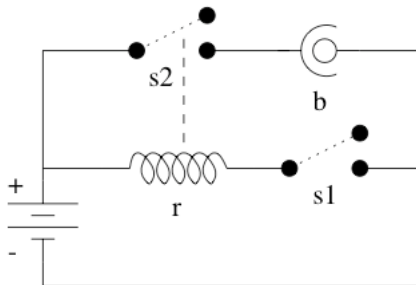
- 1 Deductive Databases
- 2 Answer Set Programming
- 3 Actions and change
- 4 **Diagnosis**

- An agent acts in a dynamic environment and observes the results of her actions.
- Sometimes she gets **discrepancies**: observations \neq expected result

Diagnosis

- Example [Balduccini & Gelfond 03]

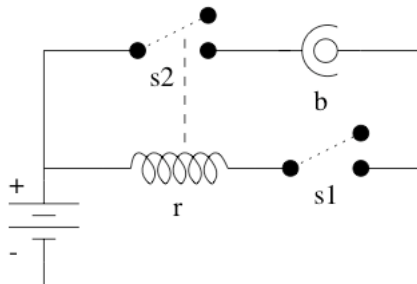
*We have a circuit with lightbulb **b** and a relay **r**. The agent can close **s1** causing **s2** to close (if **r** is not damaged). The bulb emits light if **s2** is closed and **b** is not damaged.*



Diagnosis example

- Example [Balduccini & Gelfond 03]

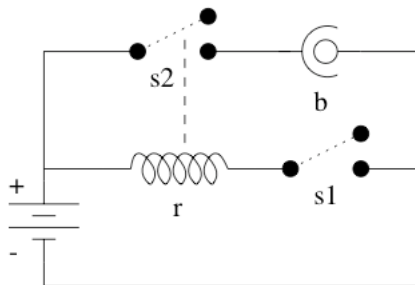
*Exogenous action **break** damages the relay. Action power-**surge** damages *r*, and *b* too, if the latter is not protected (**prot**).*



Diagnosis example

- Example [Balduccini & Gelfond 03]

We close s1 but b does not emit light: what has happened?



Diagnosis example

- Types and domains

```
#program initial.
switch(s1;s2).
component(relay;bulb).
fluent(relay;light;b_prot).
fluent(S):-switch(S).
fluent(ab(C)) :- component(C).

value(relay,(on;off)).
value(light,(on;off)).
value(S,(open;closed)) :- switch(S).
hasvalue(F) :- value(F,V).
% Fluents are boolean by default
domain(F,(true;false)) :- fluent(F),not hasvalue(F).
% otherwise, they take the specified values
domain(F,V) :- value(F,V).
```

- Fluents *ab(C)* point out that a component is damaged

- Actions are exogenous *exog* or agent's *agent*:

```
agent(close(s1)).  
exog(break;surge).  
action(Y):-exog(Y).  
action(Y):-agent(Y).
```

Diagnosis example

```
#program dynamic.
% Inertia
h(F,V) :- 'h(F,V), not c(F).
c(F)     :- 'h(F,V), h(F,W), V!=W.

% Direct effects
h(s1,closed) :- o(close(s1)).

#program always.
% Indirect effects
h(relay,on)   :- h(s1,closed), h(ab(relay),false).
h(relay,off)  :- h(s1,open).
h(relay,off)  :- h(ab(relay),true).

h(s2,closed)  :- h(relay,on).

h(light,on)   :- h(s2,closed), h(ab(bulb),false).
h(light,off)  :- h(s2,open).
h(light,off)  :- h(ab(bulb),true).
```

Diagnosis example

```
#program dynamic.
% Executability
:- o(close(S)), 'h(S,closed).

% Malfunctioning
h(ab(bulb),true) :- o(break).
h(ab(relay),true) :- o(surge).
h(ab(bulb),true) :- o(surge), not 'h(b_prot,true).
```

We use predicates *obs_o* and *obs_h* to denote observations

```
% Observed actions actually occur
o(A) :- obs_o(A).

#program always.
% Check that observations hold
:- obs_h(F,V), not h(F,V).

#program initial.
% Completing the initial state
1 {h(F,V):_domain(F,V)} 1 :- _fluent(F).
```

Diagnosis example

- These are the observations:

```
% A history
&tel {
    obs_h(s1,open) & obs_h(s2,open) &
    obs_h(b_prot,true) &
    obs_h(ab(bulb),false) &
    obs_h(ab(relay),false)

    ;> obs_o(close(s1)) &
        obs_h(light,off)
}.

#program dynamic.
% Generate exogenous actions
{ o(Z): _exog(Z) }.

cause(X) :- o(X), _exog(X).
#show cause/1.
```

Diagnosis example

- This will provide all possible explanations, but not **minimal** diagnoses.

```
$ telingo 0 diag.lp
Answer: 1
  State 0:
  State 1:
    cause(break)
Answer: 2
  State 0:
  State 1:
    cause(break) cause(surge)
Answer: 3
  State 0:
  State 1:
    cause(surge)
SATISFIABLE
```


Diagnosis example

- **Optimization problems:** we can use `maximize/minimize`
- One possible notation is:

```
#minimize <numerical_expr>: <condition>.
```

- **Example**

```
numcauses(N) :- #count{X:cause(X)}=N.  
#minimize {N:numcauses(N)}.
```

means “get minimal number of exogenous actions”

Diagnosis example

- To obtain all minimal solutions we use the options:

```
$ telingo --opt-mode=optN -n0 diag.lp
```

Two minimal solutions are found:

```
Answer: 1
  State 0:
  State 1:
    cause(surge)
Optimization: 1
Answer: 2
  State 0:
  State 1:
    cause(break)
Optimization: 1
OPTIMUM FOUND
```