In-source Testing in Answer Set Programming

declarative 2024, Amsterdam

Erik J. Groeneveld

Contents

- Who am I?
- What is Answer Set Programming
- What is In-source Testing
- How to test ASP programs
- Hands-on
- Questions

I love programming

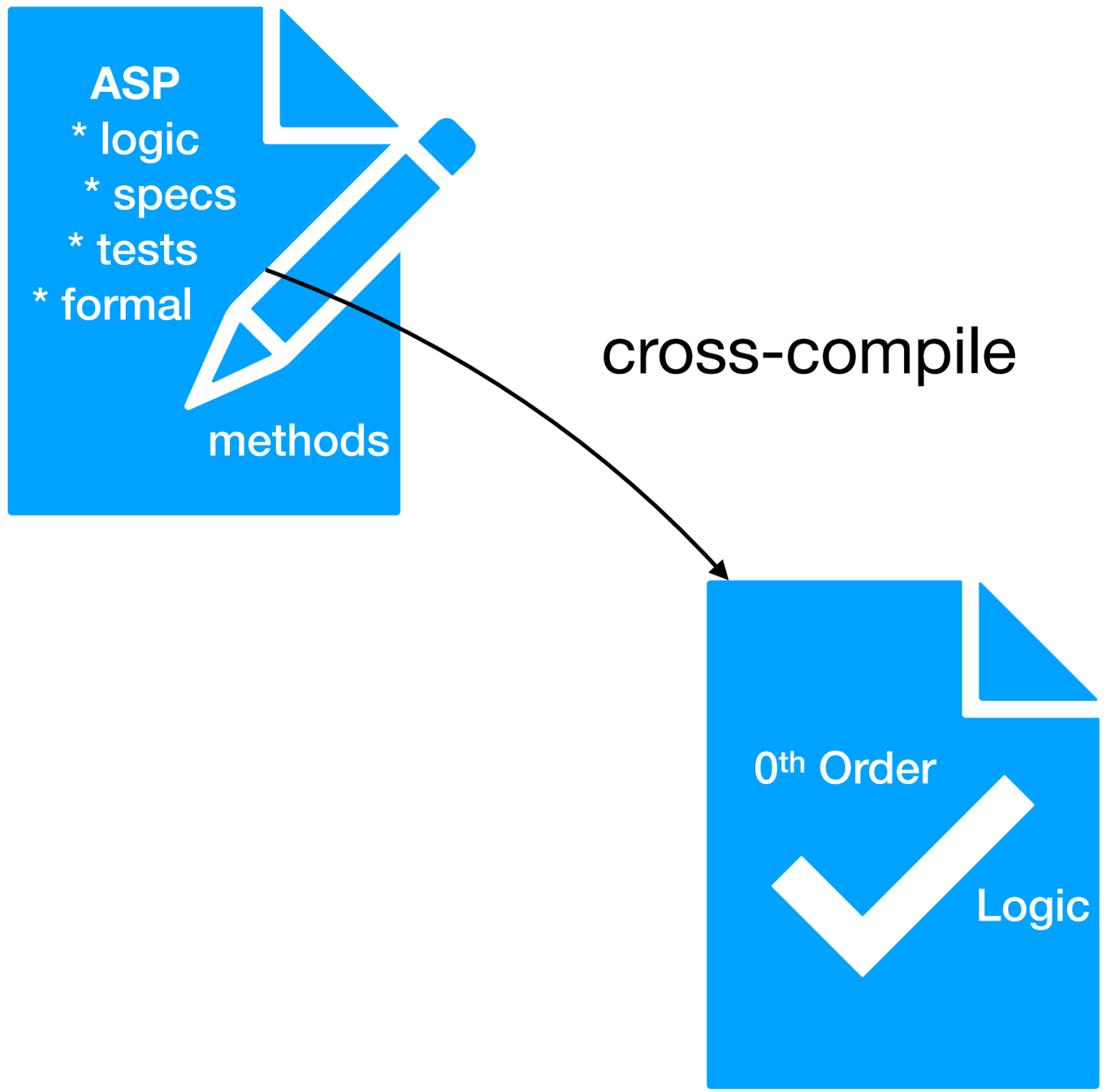
- 1996: Research engineer @Baan
- 1999: Software Engineering Research Centre (SERC)
- 2001: Owner of Seecr - search with Lucena
- 2024: Independent (finally)

- Programming Languages
  - (GW-)Basic
  - Pascal
  - C/C++
  - Python
  - Clojure
  - Answer Set Programming

- Metaprogramming
  - Efficiency
  - Size reduction
  - Patterns & idioms
  - Metaclasses
  - Integration
  - Extreme Programming
  - Push boundaries

Relevant Project

- Railway Interlocking
- Now 0th Order Logic
- Formal Specification
- Unit Testing
- Formal Methods
- => Higher Order Logic



What is In-Source Testing?

- Put your test right between your code.
    - Same language
    - Same file/compilation unit
        - In a class
        - In a function
        - At the end, beginning, interspersed
        - assert++

Why In-source?

- Reduce test code base maintenance
- Automatic and deterministic collection of tests (import)
- Automatic subset selection
- Easier refactoring (move code)
- Intuitive test shifting from unit/integration/system
- Test different environments (tests part of program)
- Less framework’ish in general (more control, less magic)

Answer Set Programming

- Answer Set Programming
  - https://potassco.org/
  - API's
    - C++/Python/Lua
- Callouts
  - @function
- Intercept
  - Observer
  - Propagator
- Main control

- What we need today
  - facts
  - rules & variables
  - constraints
  - conditional literals
  - aggregates
  - optimisation
  - explained when we meet them

```
% facts (atoms)
beer(valdieu, 8).
beer(radler, 0).
pils(gulpener, 6).
ale(kilkenny, 4).

% rule: head :- body
alcoholic(B)  :-  beer(B, _).

% disjunction
beer(B, A)  :-  pils(B, A).
beer(B, A)  :-  ale(B, A).

% conjunction
special(B)  :-  beer(B, A),  A > 5.

% choice
{ drink(radler, 0) }.
{ drink(B, A) }  :-  beer(B, A).

% constraint, it cannot be that...
:-  drink(B, A),  A = 0.

% conditional literals ('such that')
specials(T)  :-  T = { beer(B, A)  :  special(B) }.

% output control
#show drink/2.
#show specials/1.
```

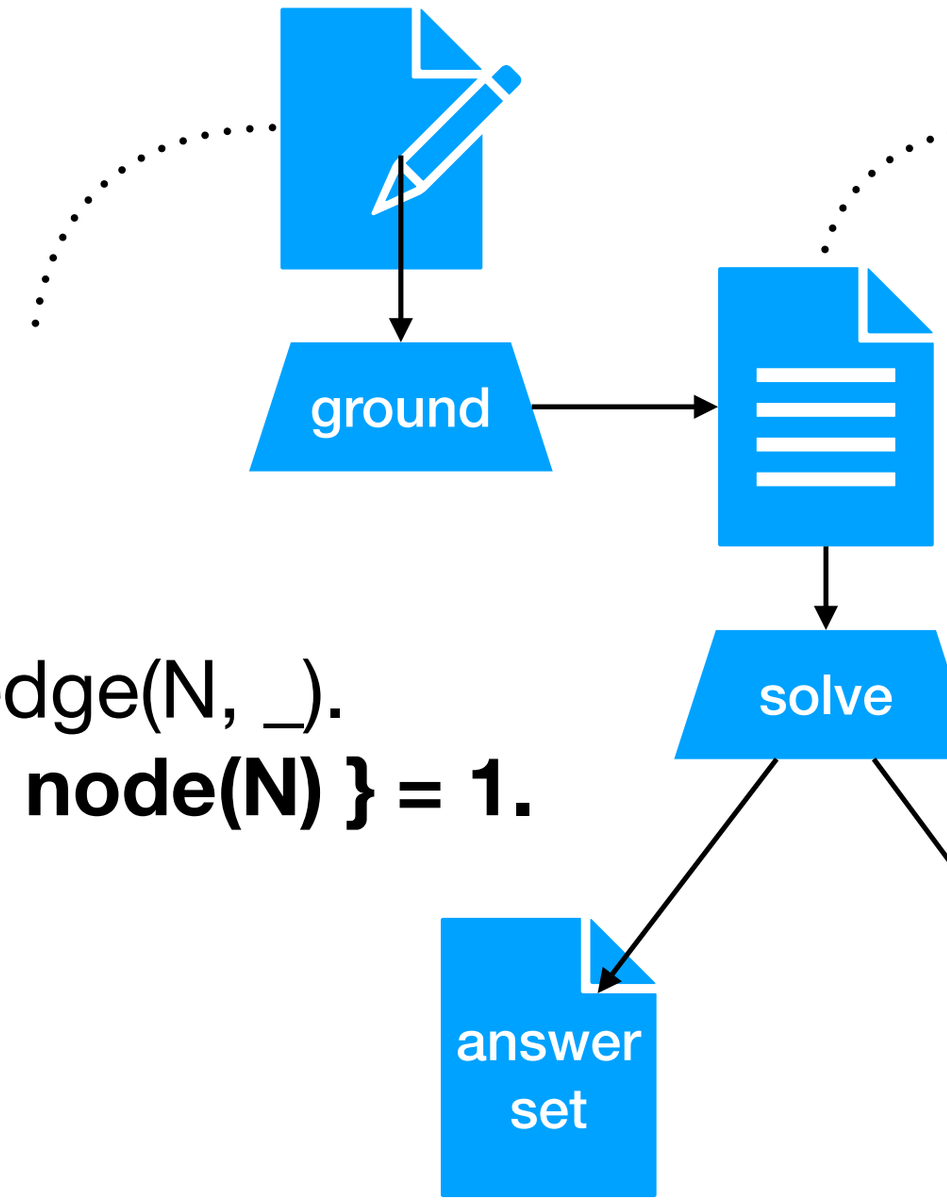
crash course ASP

see Codespace

we'll repeat it when we meet them again

How to test ASP?

```
% source
edge(0, 1).
edge(1, 0).
node(N) :- edge(N, _).
{ start(N)  :  node(N) } = 1.
```




```
% ground
edge(0, 1).
edge(1, 0).
node(0)  :-  edge(0, 1).
node(1)  :-  edge(1, 0).
{ start(0);  start(1) } = 1.
```

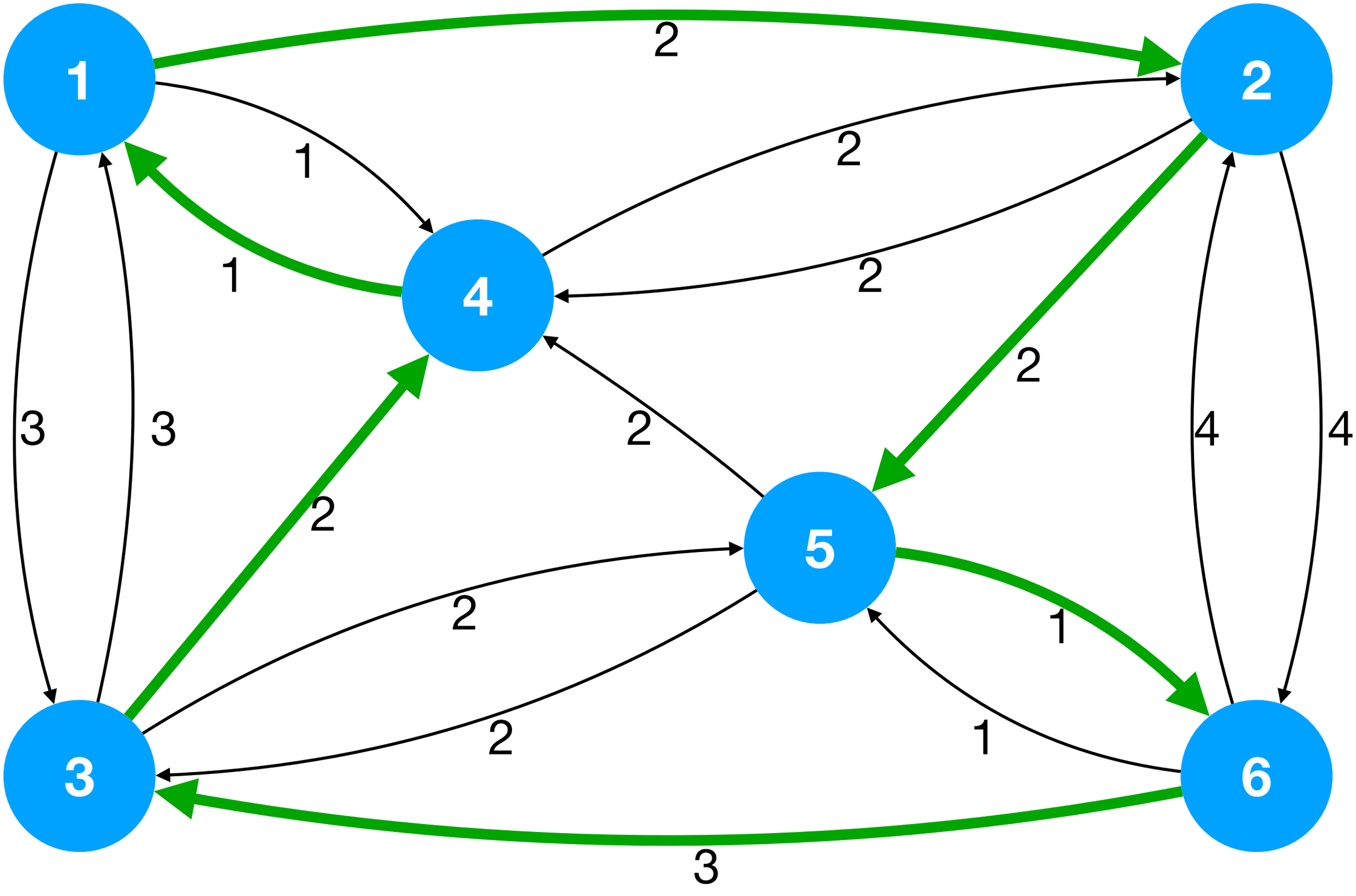
```
node(a)
node(b)
start(0)
```

```
node(a)
node(b)
start(1)
```

Design

- ASP code compatibility
  - No #script
  - No #theory
  - Only predicates.
  - Vanilla ‘clingo’ still works on code with tests.
- Drop-in replacement for ‘clingo’ (WIP)

Hamiltonian Path



Hands-On (CodeSpace)

```
%%%%%%%% Problem Instance %%%%%%%%

% edge(From, To, Cost). We use facts.
edge(1 ,2 ,2). edge(2 ,4 ,2). edge(3 ,1 ,3). edge(4 ,1 ,1). edge(5 ,3 ,2). edge(6 ,2 ,4).
edge(1 ,3 ,3). edge(2 ,5 ,2). edge(3 ,4 ,2). edge(4 ,2 ,2). edge(5 ,4 ,2). edge(6 ,3 ,3).
edge(1 ,4 ,1). edge(2 ,6 ,4). edge(3 ,5 ,2).                edge(5 ,6 ,1). edge(6 ,5 ,1).


assert("6 nodes")  :-  { node(N) } = 6.
assert("incident in", N)  :-  node(N),  edge(_, N, _).
assert("incident out", N)  :-  node(N),  edge(N, _, _).
assert("valid costs", S, E) :- edge(S, E, C),  C > 0,  C < 10.
assert("minimal cost")  :-  #sum { C, A, B  :  step(A, B),  edge(A, B, C) } < 12.


%%%%%%%% Problem Encoding %%%%%%%%

%%%% Preparation %%%%
% Infer nodes from edges. We use a simple disjunctive rule: head :- body.
node(N)  :-  edge(N, _, _).    % variable N,  wildcard _
node(N)  :-  edge(_, N, _).    % disjunction/or


%%%% Generation %%%%
% Choose an arbitrary step. We use conditional literal ("such that") + choice
step(A, B)  :  edge(A, B, _).

% if you have one step, choose a connected one, but not back.
step(B, C)  :  edge(B, C, _),  C <> A   :-   step(A, B).

% Path to given node via step's. We use a disjunctive rule with conjunction
```