B2ASP

A Clingo SAT backend for ProB

Michael Leuschel, Maxime Zielinger
Heinrich-Heine University of Düsseldorf

14.11.2025, HSD Düsseldorf
Scryer Prolog Meetup

STUPS Group Core Expertise

Software Engineering and Programming Languages Group

- **Formal Methods**:
  - ProB validation toolset: animation/simulation, constraint solving, test-case generation, model checking, data validation, EN51028 T2 certified, visualisation, validation obligation manager
  - Modelling & verification of safety critical systems (hybrid train detection, moving block train control, ...)
  - Certification of AI, combining formal models and AI, KI-LOK project
  - B, Event-B, TLA+, Alloy, CSP, Z, ...
- **Programming Languages**
  - symbolic AI, constraint logic programming
  - tool validation, static program analysis, just-in-time compilation

Before starting: Other Work

of potential interest to Scryer Prolog audience

- Partial Evaluation and Program Analysis for Prolog
  - Ecce
- Jupyter Prolog Notebooks for SWI+SICStus; adaptable for Scryer?
- Testing and certification: Prolog fuzzer, Mathematical Laws checking via constraint solving/model checking
- ProB can also animate Prolog rules
  - Inference rules of Event-B, natural deduction visualisation (cf Mark Thom's first talk), see later
- Teaching: Introduction to Logic Programming (A*, Minimax, MCTS, ... in Prolog), Advanced Topics in Logic Programming (writing interpreters, parsers, semantic analysis in Prolog, big-step / small-step semantics, ...)

Jupyter Kernel for Prolog

Well-Formed Formulas (WFF)

All atomic propositions are WFF) If a und b are WFF then so are:

- (¬ a)
- (a ∧ b)
- (a ∨ b)
- (a → b)
- (a ⟺ b) No other formulas are WFF.

Note in the slides we use the single arrow `→` instead of the double arrow `⇒` for implication which is maybe more standard.

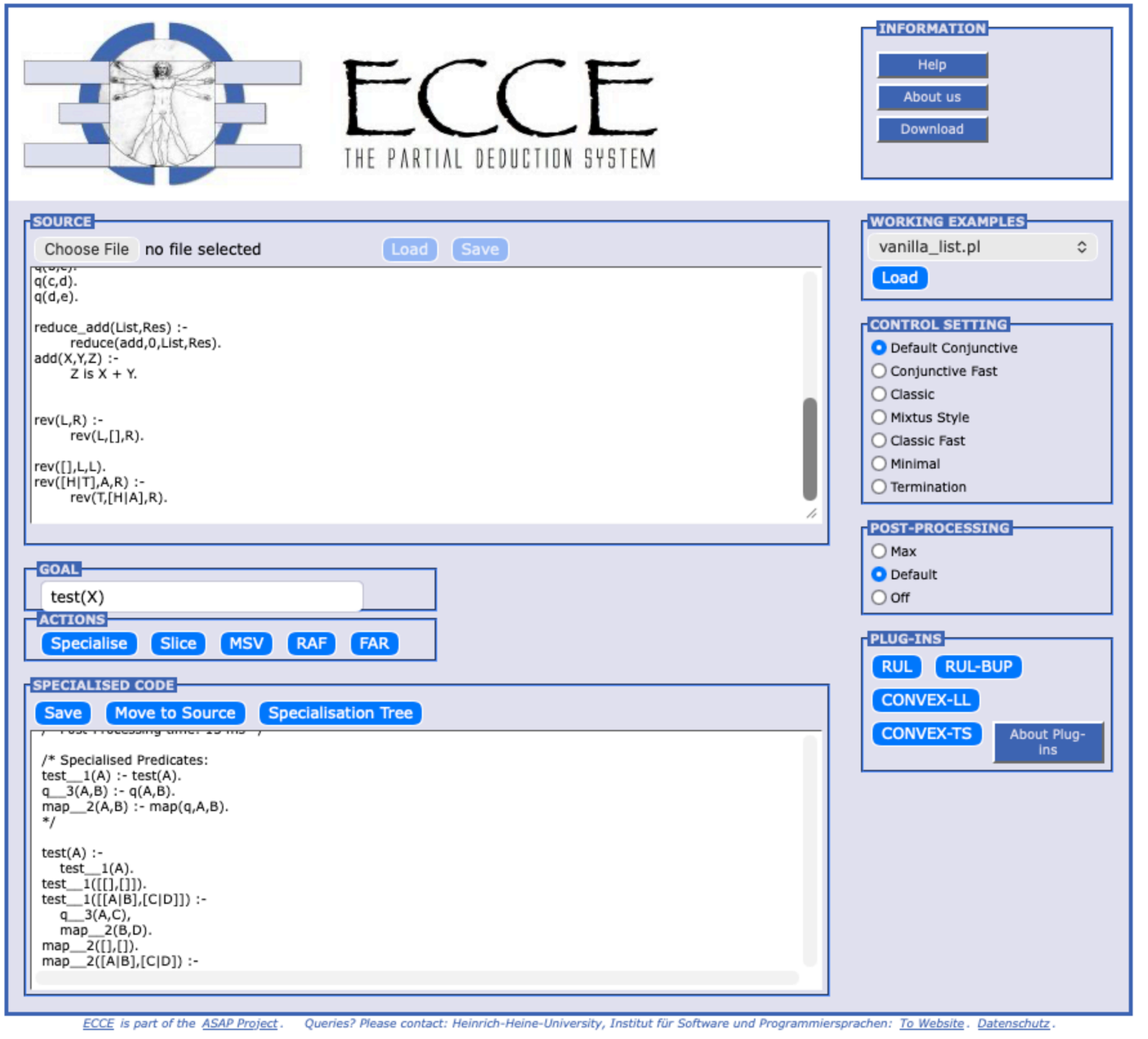
Comment: a, b are metavariablens outside the syntax of propositional logic.

Maybe this reminds you of formal grammars from a theoretical computer science lecture. And indeed, the above can be written as a grammar using a non-terminal symbol wff.

In Prolog, grammars can actually be written using DCG notation, which we will see and understand much later in the course. Here we simply write the grammar in Prolog style and can then use it to check if a formula is a WFF:

```
:- set_prolog_flag(double_quotes, codes).
wff --> "p". % atomic proposition
wff --> "q". % atomic proposition
wff --> "(¬",wff,")".
wff --> "(", wff, "∧", wff, ")".
wff --> "(", wff, "∨", wff, ")".
wff --> "(", wff, "→", wff, ")".
wff --> "(", wff, "⟺", wff, ")".
```

https://github.com/hhu-stups/prolog-jupyter-kernel



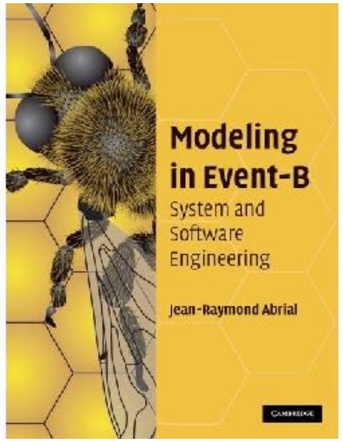
https://ecce.stups.hhu.de/ecce/index.php

```
>>> :print @FUZZ
@FUZZ = #id1.(id1 : POW(STRING) & {id0|id0 : REAL & ({} = id1\/id1 & id1/\id1 = {})} : FIN({id0|id0 : REAL & ({}
= id1\/id1 & id1/\id1 = {})}))
∃id1.(
  {id0|(∅ = id1\/id1 ∧ id1/\id1 = {})} ∈ FIN({id0|(∅ = id1\/id1 ∧ id1/\id1 = {})})
)

>>> :print @FUZZ
@FUZZ = #id0.(id0 : POW(INTEGER * POW(INTEGER * POW(STRING))) & last(last(id0)) /<<: inter({RANGE_LAMBDA__|
RANGE_LAMBDA__ : POW(STRING) & RANGE_LAMBDA__ = STRING}))
∃id0.(
  last(last(id0)) ⊄ inter({ρ|ρ = STRING})
)


>>> :print @FUZZ-ARITH
@FUZZ = btrue
⊤

>>> :print @FUZZ-ARITH
@FUZZ = id0 : POW(INTEGER * (INTEGER * INTEGER) * INTEGER) & id1 : POW(INTEGER * (INTEGER * INTEGER) * INTEGER) &
(MAXINT /: {} & MAXINT > 75 => prj1(NATURAL1,pred) /<<: id0 \/ id1)
(
  (
    MAXINT ∉ ∅
    ∧
    MAXINT > 75
  )
  ⇒
  prj1(ℕ₁,pred) ⊄ id0 ∪ id1
)
```

```
MACHINE ArithmeticLaws
DEFINITIONS
 MAX_x == 50;
 MAX_y == 9;
 MAX_z == 4;
 MIN_x == -3;
 MIN_y == -3;
 MIN_z == -3
VARIABLES x,y,z
INVARIANT
 x:INTEGER & y:INTEGER & z:INTEGER &
 x*y = y*x &
 x*(y+z) = x*y + x*z &
 x+y = y+x &
 x*1 = x &
 1*x = x &
 x*0 = 0 &
 0*x = 0 &
 (x+y)+z = x + (y+z) &
 (x*y)*z = x*(y*z) &
 2*x = x+x &
 x**2 = x*x &
 (x>=0 => 1**x = 1) &
 (x>=0 => (((x / 2)*2 = x) <=> (x mod 2 = 0))) &
 (x>0 => 2**x = 2*(2**(x-1))) &
 (x>0 => 2**(10*x) = 2*(2**(10*x-1))) &
 (x>0 => 3**(10*x) = 3*(3**(10*x-1))) &

 (x>y or x<=y) &
 (x>y or x=y or x<y) &
&…
```

B and Event-B Formal Methods: Industrial Uses

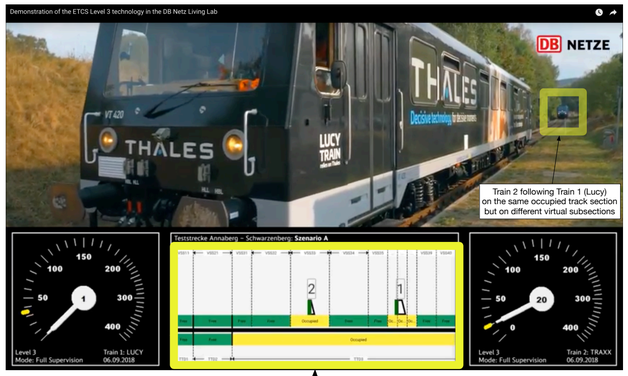
>100 metro lines worldwide



Software

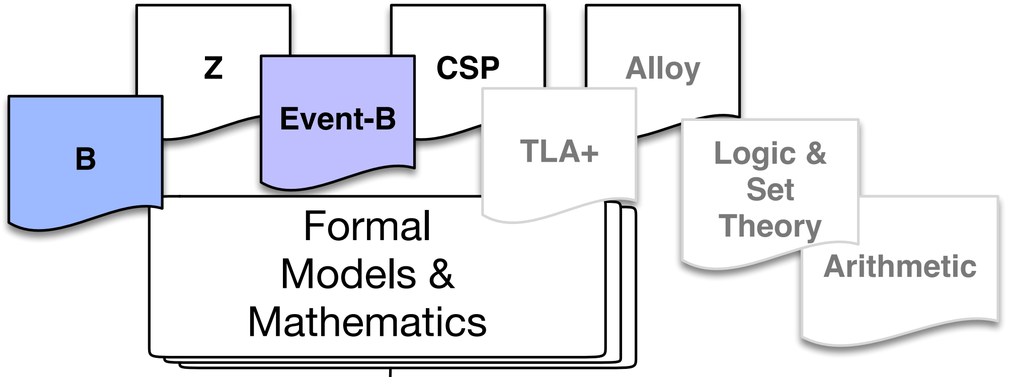
Data & Config. Validation

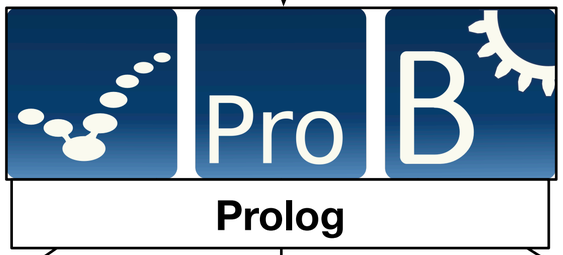
System Specification & Executable Model



System Analysis & Safety Case

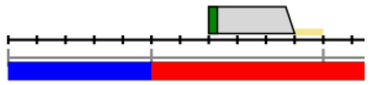
Validation Tool for Formal Models



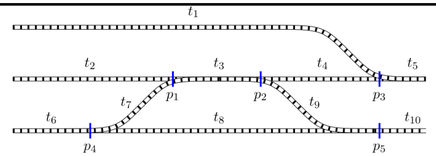


T2 Certification

Animation
Domain-Specific
Visualisation



Data
Validation



Model Checking and
Constraint-Based
Verification

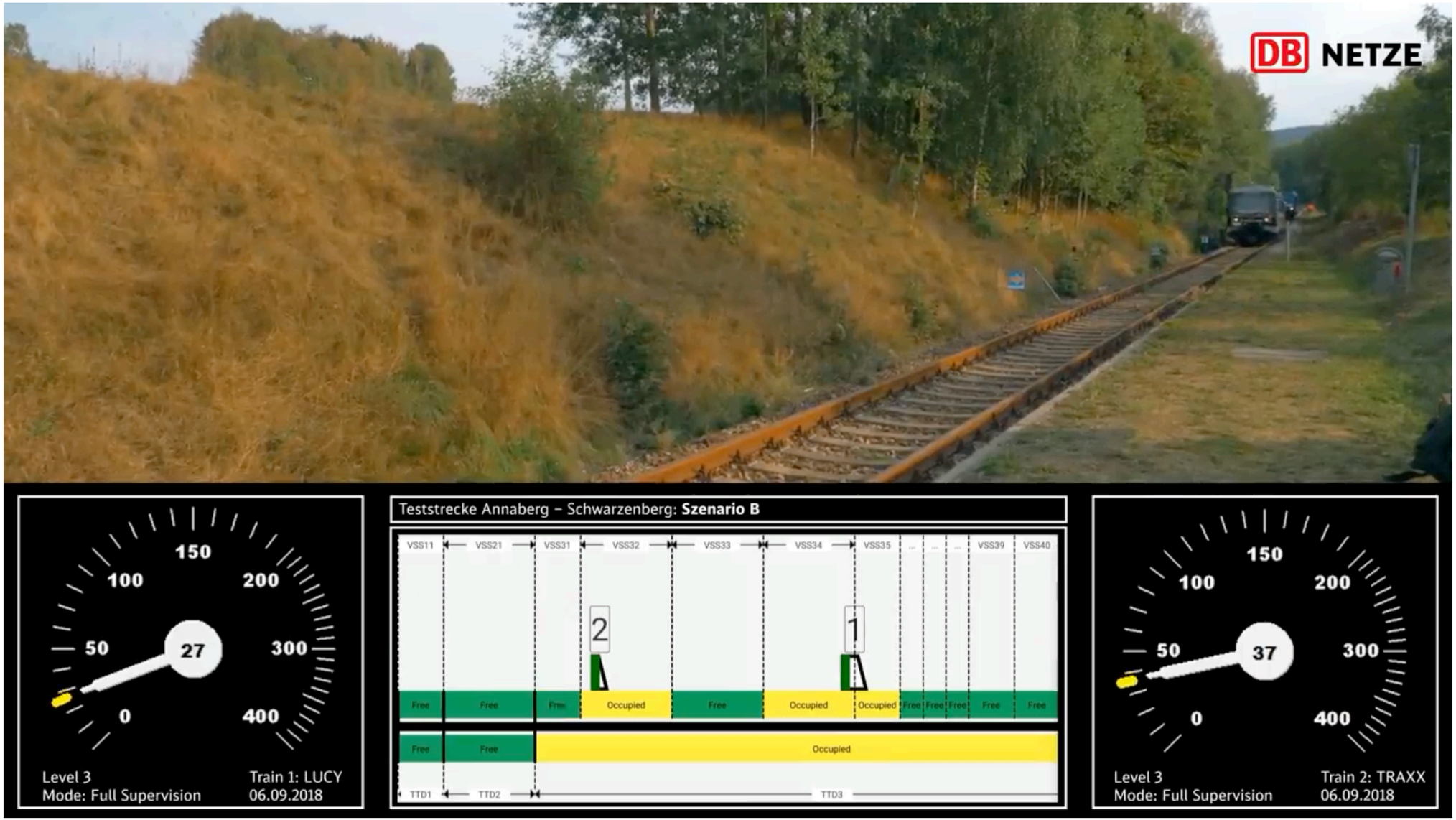
✓

First article
FM'2003

50 1972 2022

received **Colmerauer Prize**
(50 Years of Prolog)

ProB running in SICStus **Prolog** in real-time executing a **formal** B **model** of the Hybrid-Level 3 principles (now ETCS Level 2 HTD)



Train 2 following Train 1 (Lucy) on the same occupied track section, but on different virtual subsections

Source: https://www.youtube.com/watch?v=FjKnugbmrP4

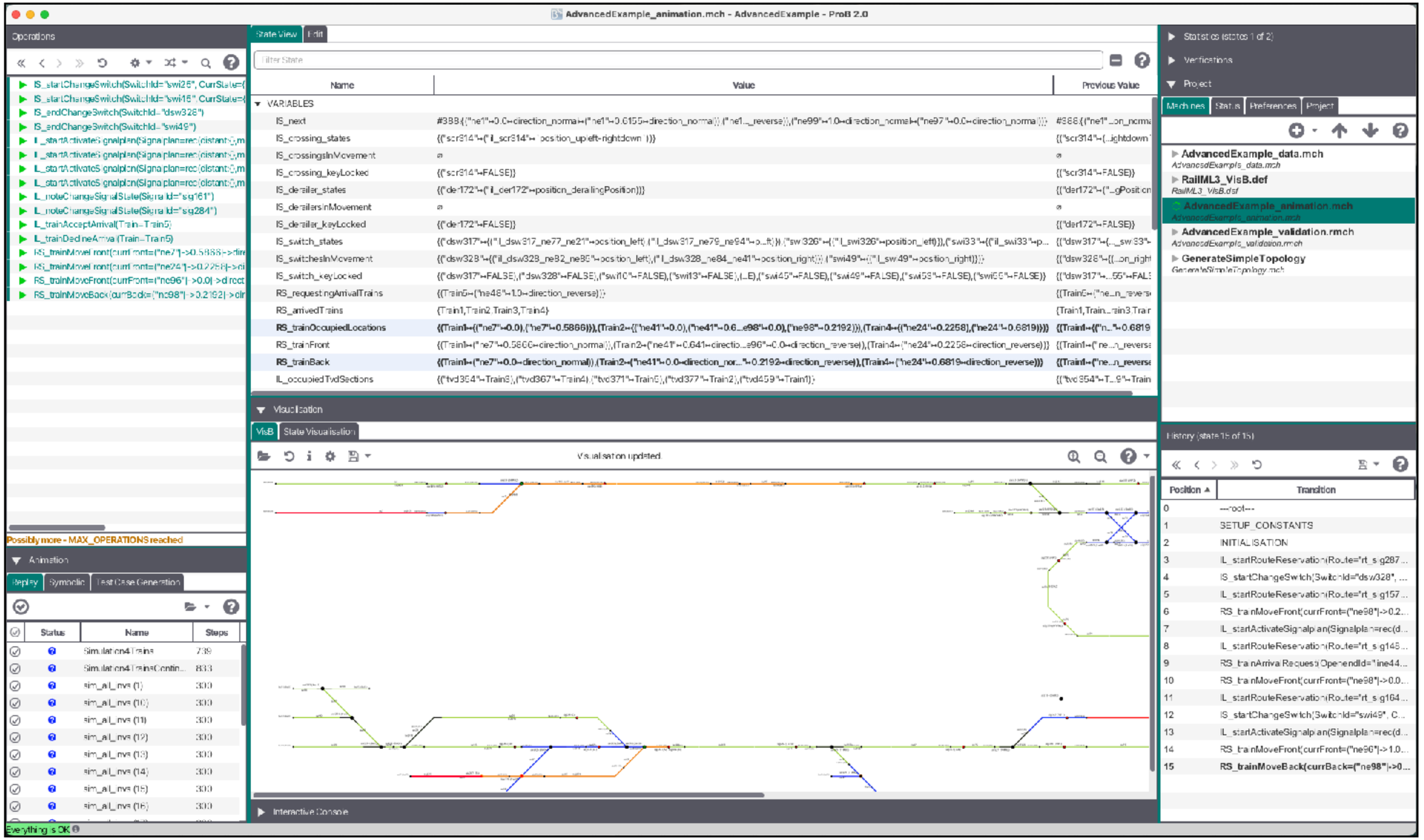
ProB2-UI - Demo

Demo:

- sequent_simple

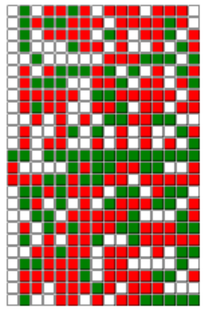
- Sequent

- Lamp



B2ASP

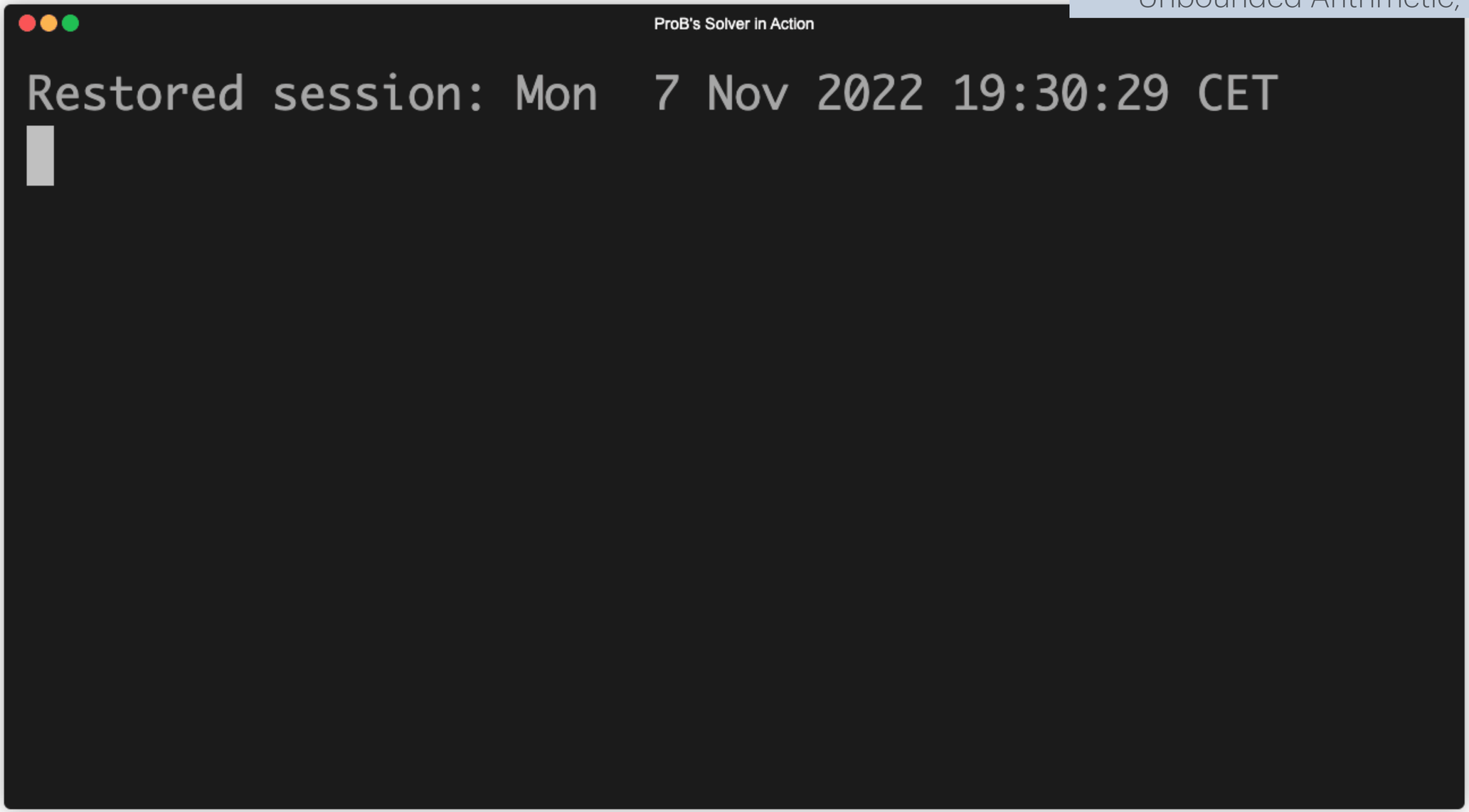
Essentials

{B} 

- New translation of **B** (Predicate Logic, Higher-Order Set Theory, Arithmetic) to **SAT** and back
- Available in **ProB** and applicable to other state-based formalisms (**TLA+**, **Z**, ...)
- Technique fully rooted in Prolog and Logic Programming: CLP(FD) Bounds Analysis, Prolog Translation from B to **ASP** (Answer Set Programming) and back (probably could be run in Scryer Prolog)
- Not a general purpose solver, but very useful for dedicated **constraint satisfaction** and **optimisation** problems and symbolic verification

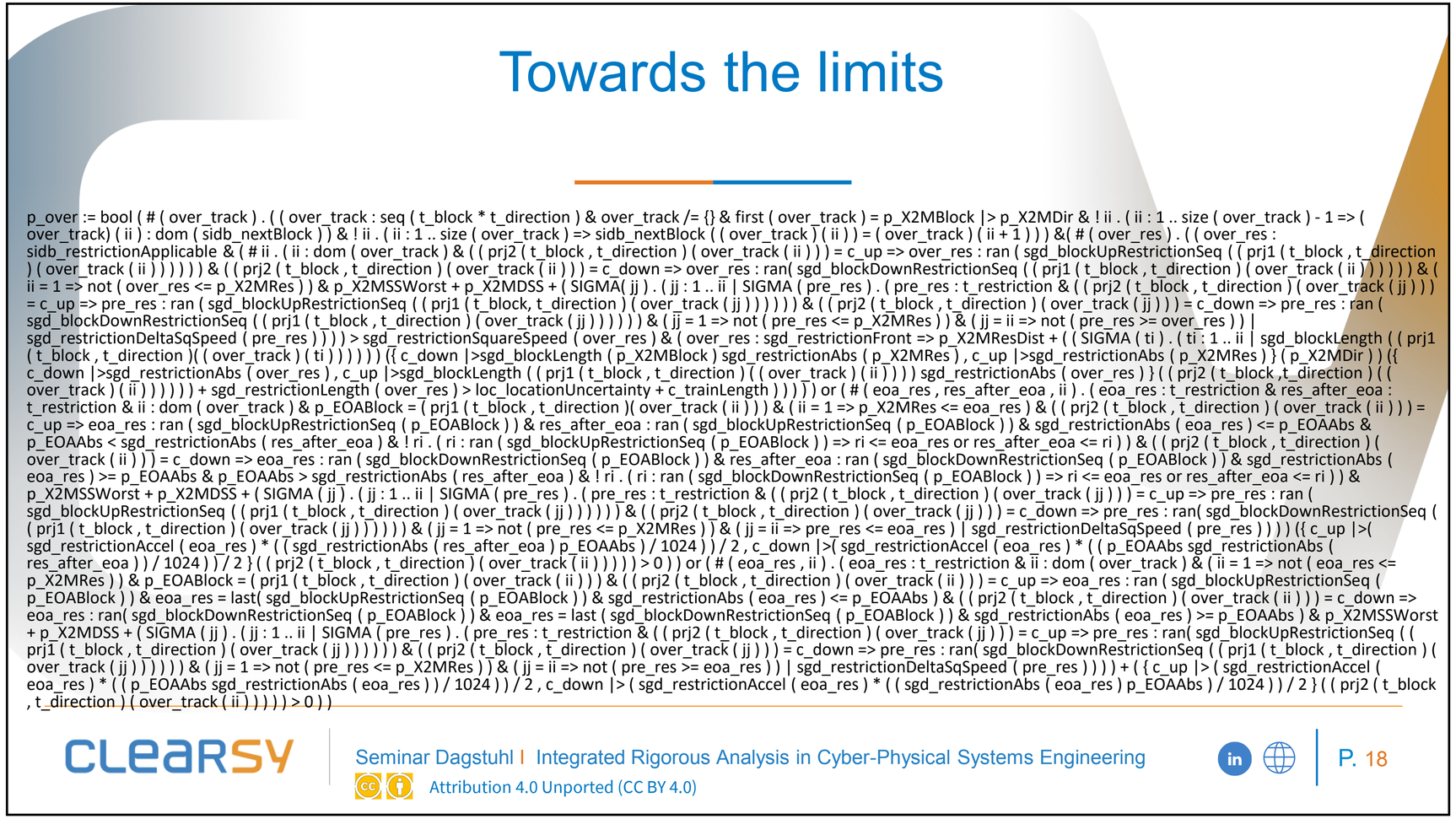
ProB's Solver in Action

Demo: Set Unification, Set Comprehension with open variable, Unicode, Sets of Sets,, Unbounded Arithmetic, ...



Data Validation

ProB's default solver is good for animation and data validation



from Talk of Thierry Lecomte

models with up to
10 million lines of B

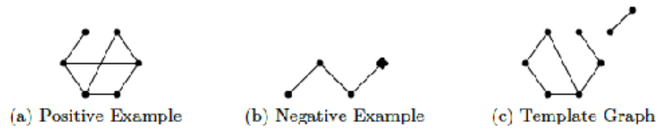
EN50128
Certification as T2 tool

cf. CLEARSY Smart Solver,
Thales Rubin, Siemens,...

But other applications

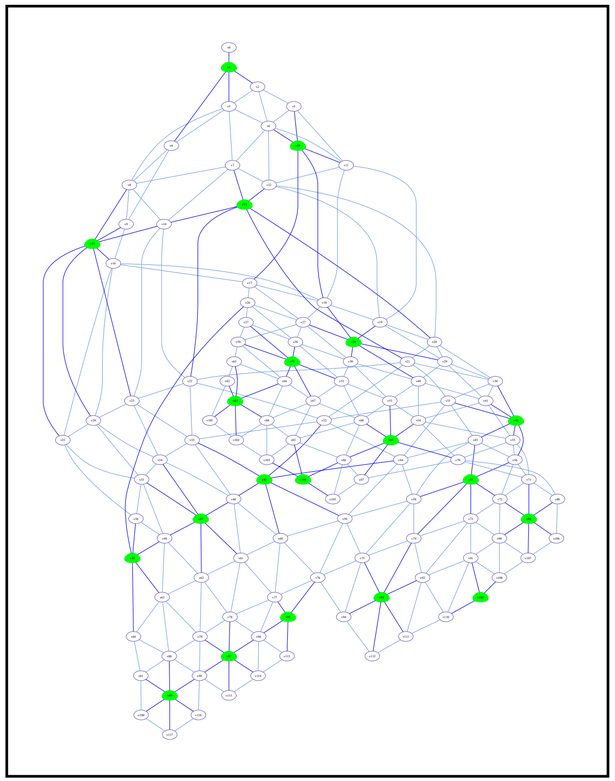
require different solvers

Machine Learning: Graph Mining



(a) Positive Example (b) Negative Example (c) Template Graph

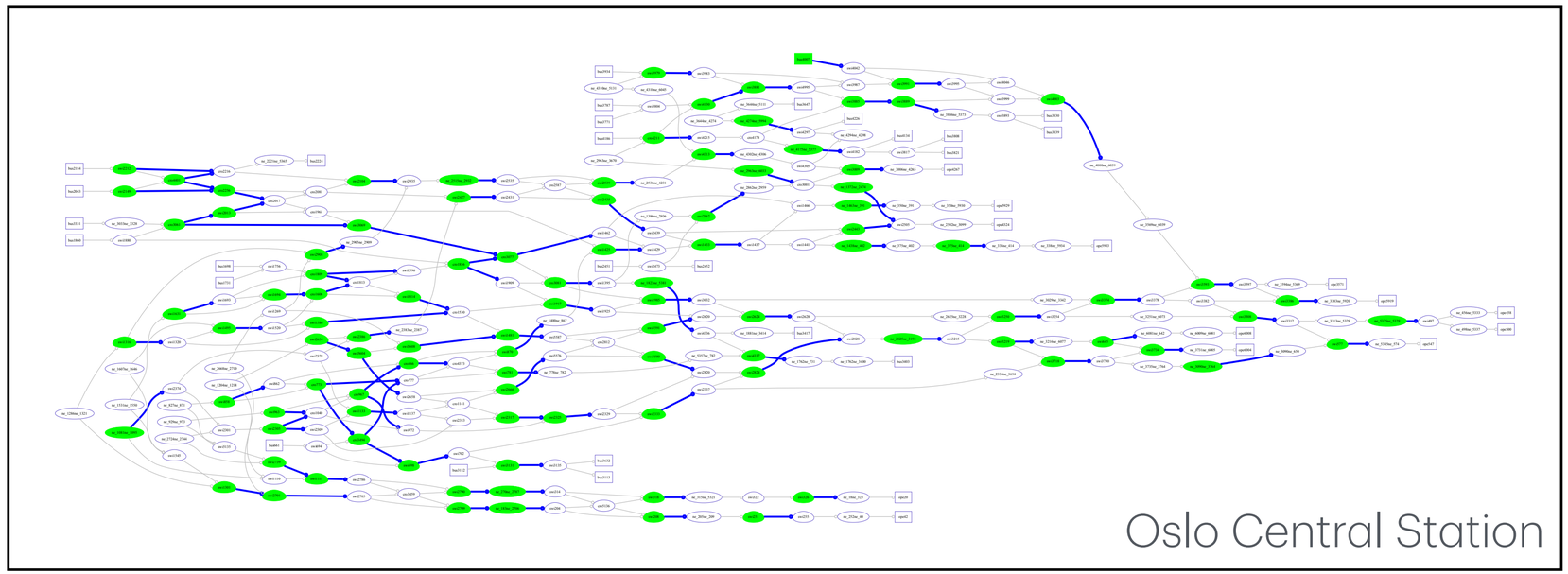
Biological Application:
Minimal Dominating Sets



Hardware: Clearsy Safety Platform

Symbolic **Railway** Interlocking Verification

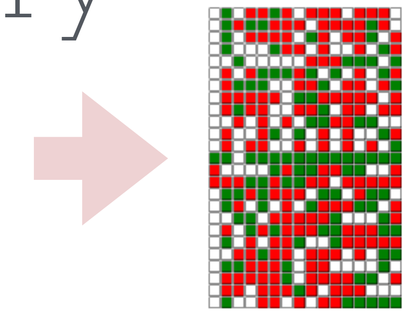
Data Generation: Balise Placement



Oslo Central Station

Alternatives to Solving B / Set Theory

ProB default: CLP(FD) + co-routines + attributed vars

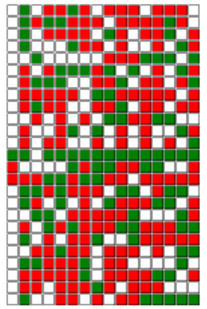
{B} 

- **CHR** (Constraint Handling Rules)
  - already used (optionally) for a few constraints
  - tricky to make a robust & fast solver
- **SMT** (Z3, CVC4) [SchmidtLeuschel: Int. J. Softw. Tools Technol. Transf. 24(6): 1043-1077 (2022)]
  - ok for unsatisfiable constraints,
  - not yet good for model finding

Alternatives to Solving B / Set Theory

ProB default: CLP(FD) + co-routines + attributed vars

{B}



- **SAT**
  - via **Kodkod** library (works but unsoundness issue for card constraints) [FM'12]
  - **B2SAT** [FM'24]: encoding SAT translation rule in Prolog, interleave with ProB default solver
    - but we need to encode everything: aggregates (cardinality, sum,...), arithmetic, custom data types, set encodings, ... ; currently only a few specific card constraints supported
    - Idea of this paper: use another **logic programming** technology: **ASP** (Answer Set Programming) and make use of their translations to SAT (e.g., **clingo** tool)

ASP (Answer Set Programming)

Basics

- Normal logic program (i.e., with negation)
- Compute stable models: typically by “grounding” and SAT solving
- **Stable** model of a logic program (called answer set): is a model in classical logic, where all atoms in the model are justified by some rule
  - Example: three classical logic models: {q,p}, {q,r}, {p,q,r}. Only {q,p} is stable.

```
q.
p :- q, not r
```

ASP (Answer Set Programming)

Language Constructs

- Horn clauses, with variables (aka Datalog)
- Disjunctions in head
- Integrity Constraints
- Choice
- Aggregates

```
mortal(X) :- human(X).

p(X) ; q(X) :- r(X).

:- edge(X,Y), col(X,C), col(Y,C).

1 { p(X) } 2.

... #sum{.....}
```

Clingo & ASP (Answer Set Programming)

https://potassco.org/clingo/

- **clingo**: Datalog → **gringo** → Ground Datalog → **clasp** → Model via SAT solving

```
p(1).
p(2).

q(2).
q(3).

pandq(X) :- p(X),q(X).

porq(X) :- p(X).
porq(X) :- q(X).
```

Datalog/ASP

clingo

```
q(2) q(3) p(1) p(2) pandq(2) porq(2) porq(3) porq(1)
```

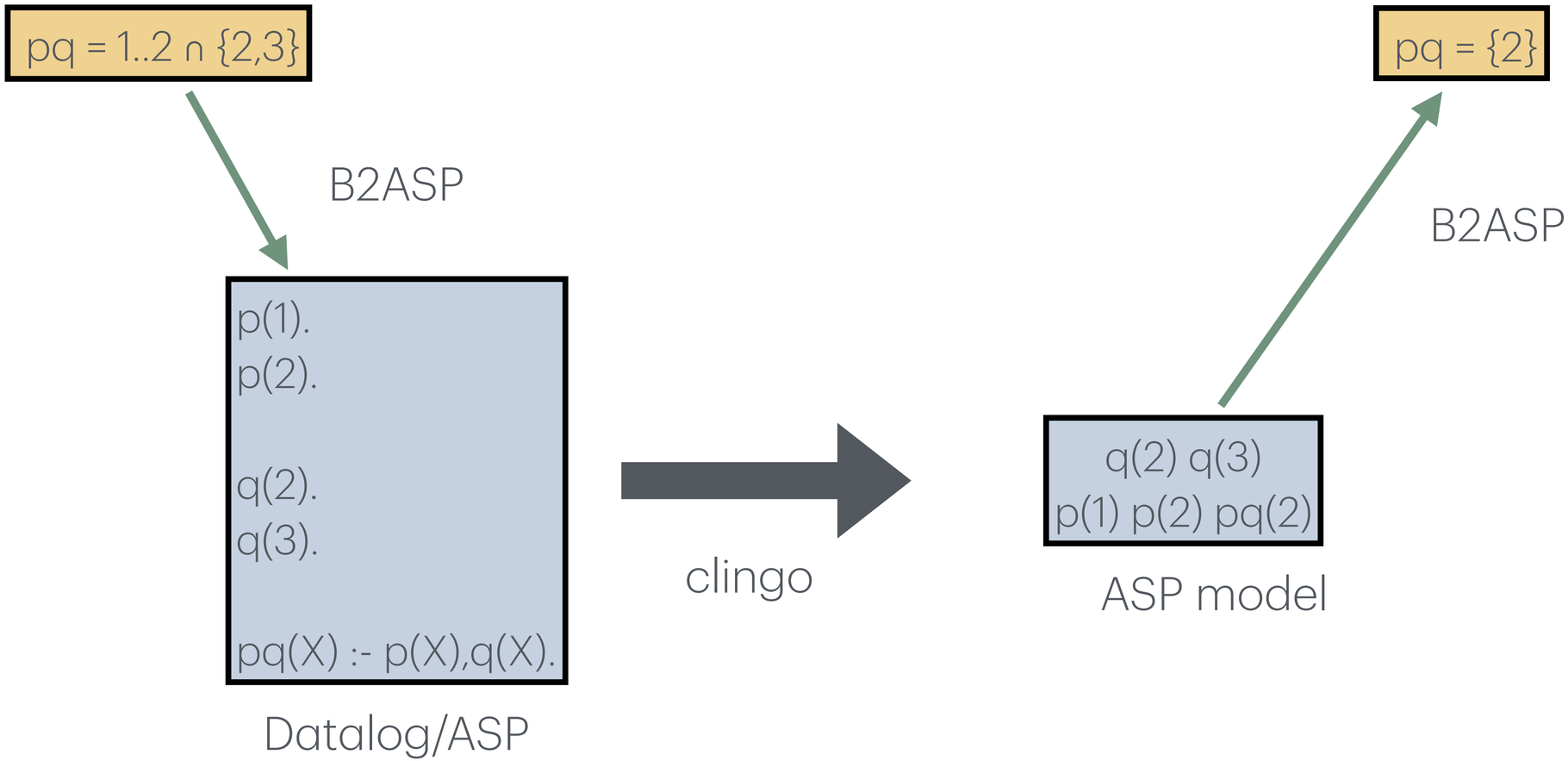
(stable) model

```
$ clingo simple.lp
clingo version 5.8.0
Reading from simple.lp
Solving...
Answer: 1 (Time: 0.002s)
q(2) q(3) p(1) p(2) pandq(2) porq(2) porq(3) porq(1)
SATISFIABLE

Models       : 1
Calls        : 1
Time         : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.001s
```

B2ASP

Basic Principle (simplified)

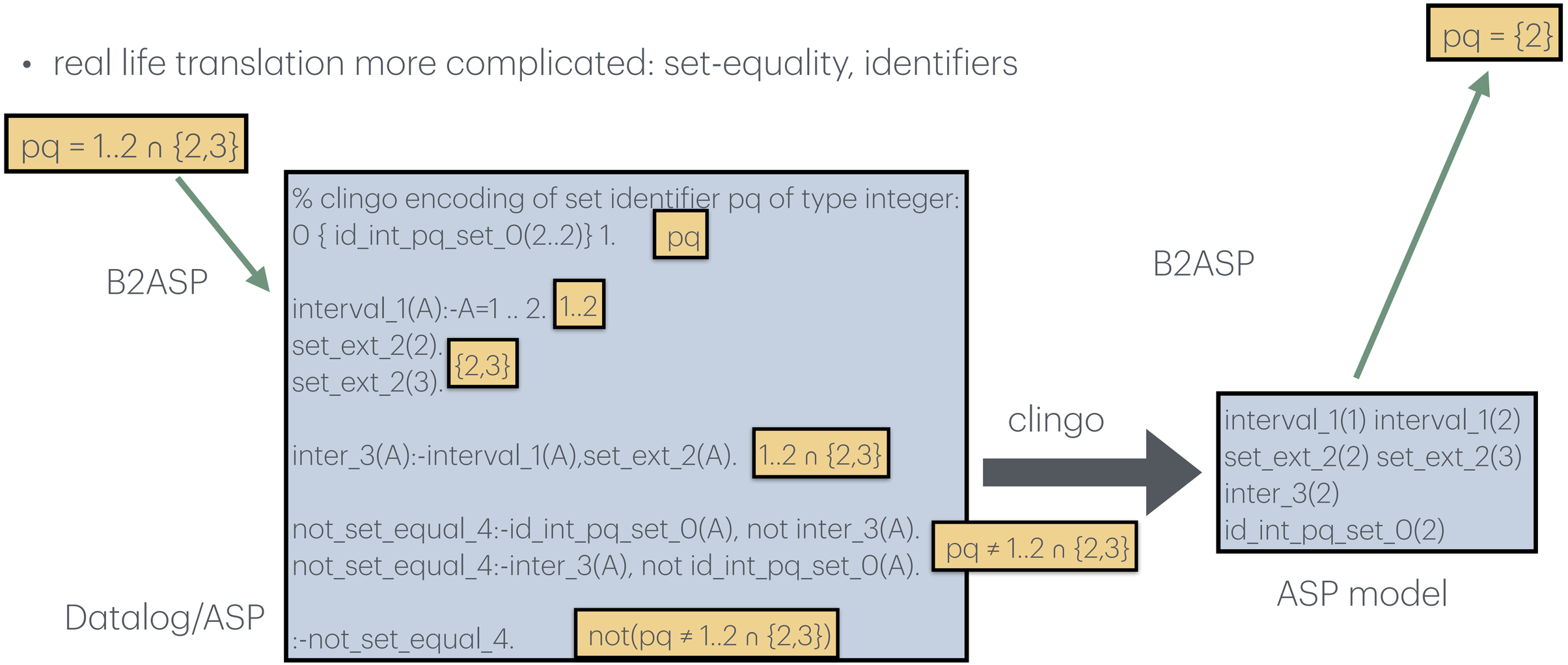
- B Formula ⇢ ASP ⇢ clingo ⇢ Model ⇢ B solution



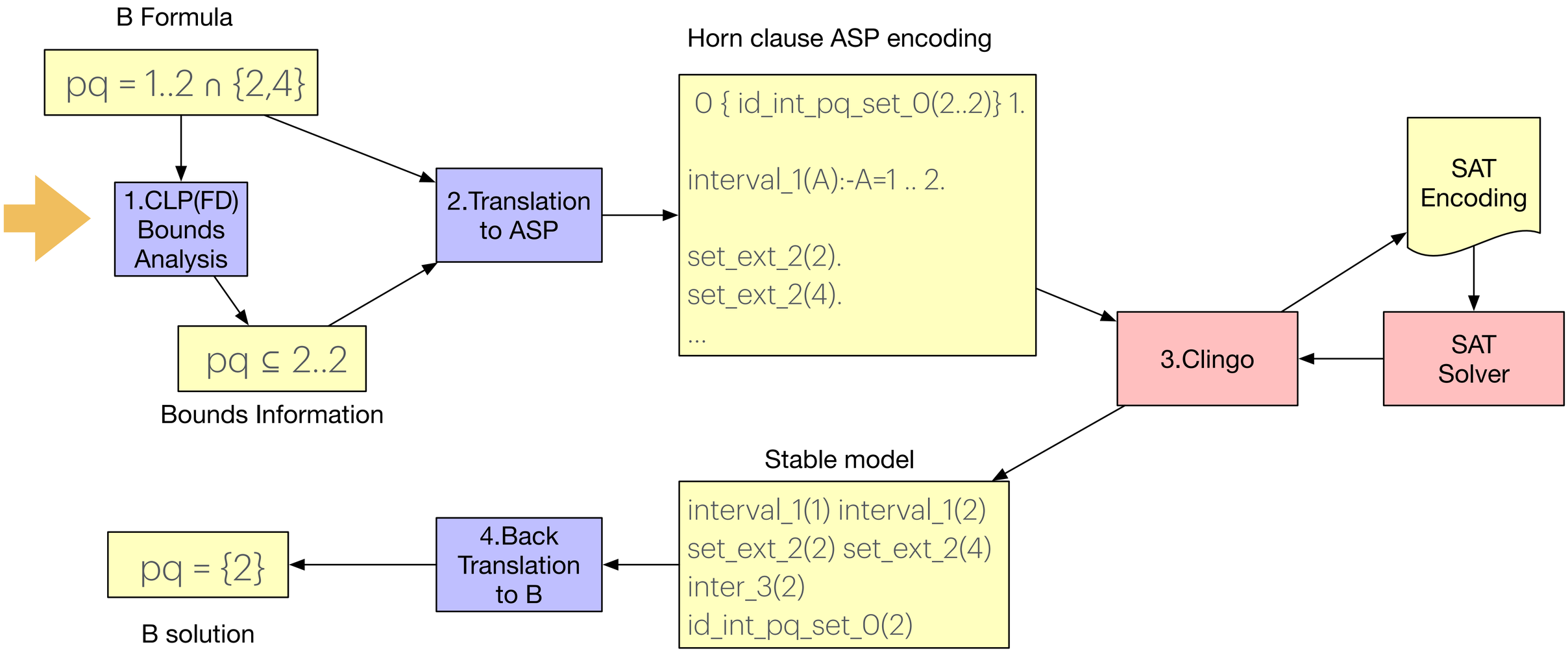
B2ASP

Basic principle with real translation example

- real life translation more complicated: set-equality, identifiers

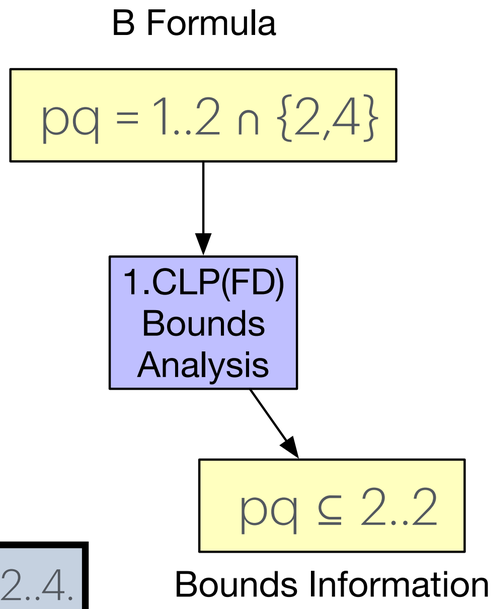


Overview of the full Pipeline



B2ASP

1.Bounds Analysis



- Idea: represent possible values of scalars and sets by CLP(FD) variables
- **CLP(FD)** keeps track of (union of) intervals for possible values
- We do not perform labelling, we are interested in **bounds**
- Tricky bit: we do not want the analysis to fail for 0..1 ∩ {2,3}
  (but return the empty set)
  - Solution: separate upper and lower bound FD value:
    empty set can be represented by Low > Up
- Tricky bit: multiple representation of empty set,
  set equality is not equality of bounds!

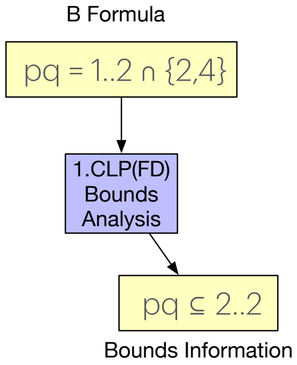
```
?- X in 1..2, Y in 2..4.
X in 1..2,
Y in 2..4 ?
yes
```

```
?- X in 1..2, Y in 2..4, X=Y.
X = 2,
Y = 2 ?
yes
```

```
| ?- X in 0..1, Y in 2..4, X = Y.
no
```

B2ASP

Bounds Analysis



- for sets we have three FD variables: binterval(**Low**,**Up**,**NonEmpty**) with NonEmpty #<=> (Low #=< Up)

```
intersect_bounds(binterval(Low1,Up1,NE1),
                 binterval(Low2,Up2,NE2),Bounds) :-
  init_binterval(Low,Up,Bounds),
  NE #=< NE1, % if set1 empty intersection empty
  NE #=< NE2, % ditto for set 2
  Low #= max(Low1,Low2), Up #= min(Up1,Up2).
```

pq = 1..2 ∩ {2,3}

B2ASP

Phase 1

pq = 1..2 ∩ {2,3}

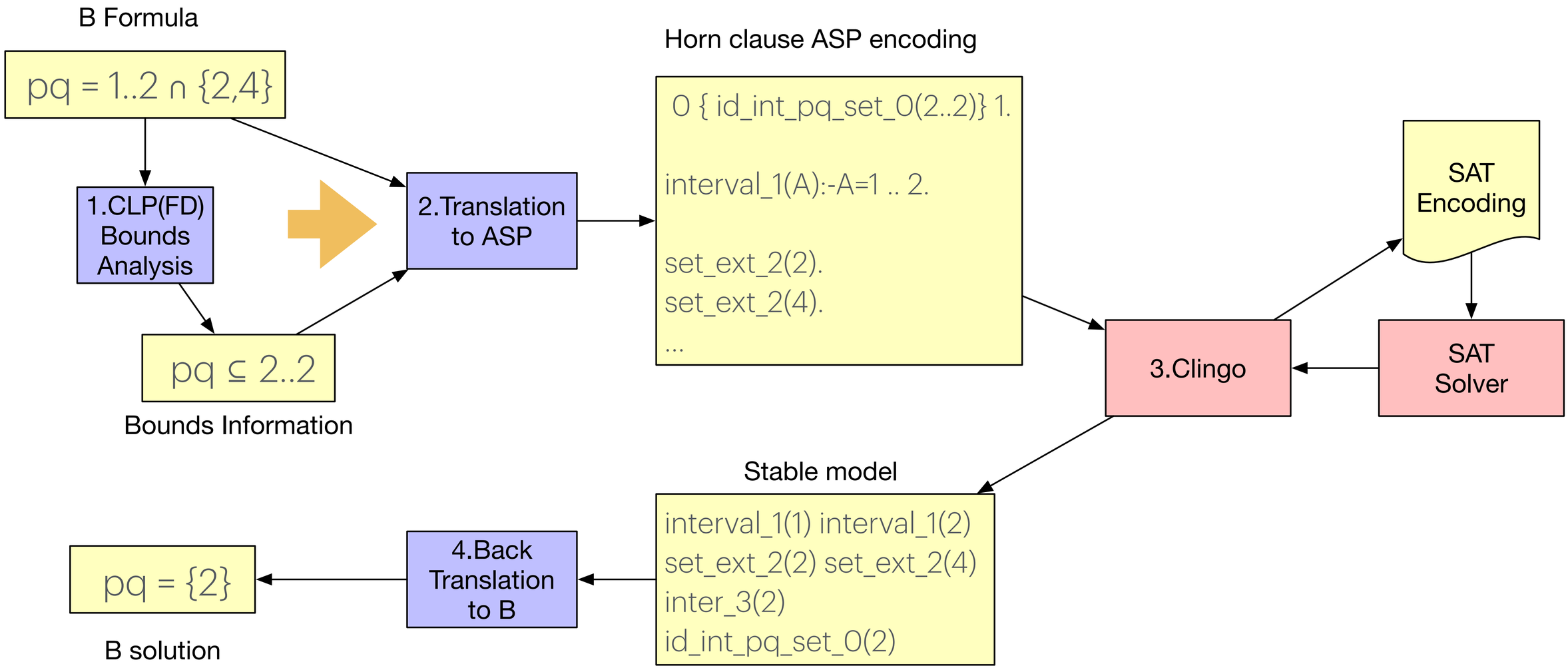
binterval(X,Y,NonEmpty) ⇔ binterval(2,2,1)

1..2 ∩ {2,3}

binterval(1,2,1) ∩ binterval(2,3,1)

Overview of the full Pipeline

2. Translation to ASP

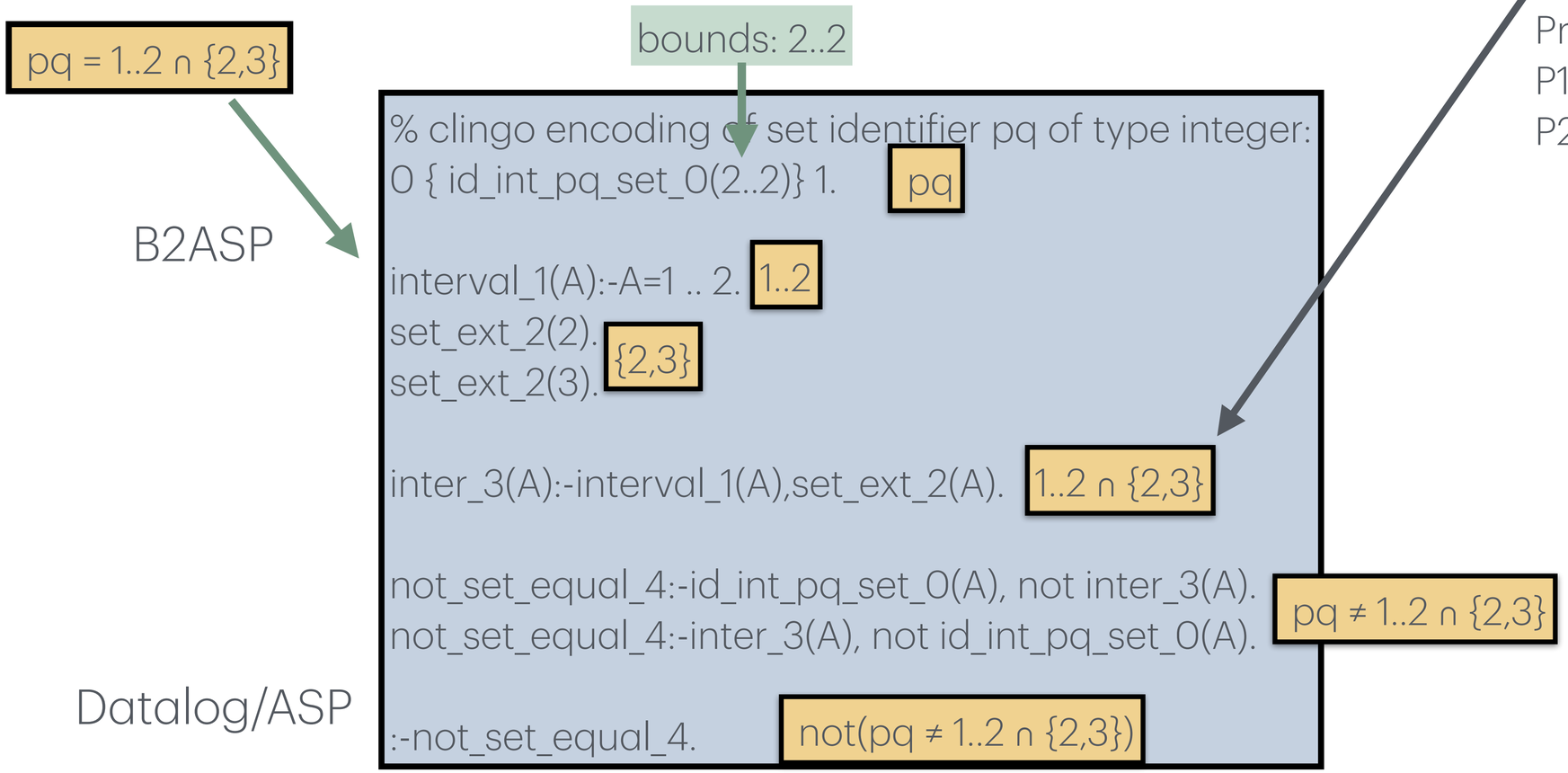


B2ASP

2.Prolog Rules

```
:- mode trans_set(+BAST,+LocalID_Env,-ClingoPred).

trans_set(intersection(A,B),Env,Pred) :- !,
   trans_set(A,Env,P1),
   trans_set(B,Env,P2),
   gen_clingo_pred_clause(inter,Pred,[X],Env, (ecall(P1,[X],Env),ecall(P2,[X],Env)) ).
```

- Example: how to translate intersection operator



Example Translation Rules

Set Expressions

Set Expression ϕ	Clauses for P_ϕ	Side Condition (comment)
$\varnothing$	$P_\phi(X)$ `:- 1=2.`	
x	`0{`$P_\phi(Low..Up)$`}`n`.`	$x \subseteq$ Low..Up (bounds analysis) $n = Up + 1 - Low$
$A \ldots B$	P_ϕ`(Z) :-` P_A`(X),` P_B`(Y), Z = X..Y.`	
$A \cup B$	P_ϕ`(X) :-` P_A`(X).` P_ϕ`(X) :-` P_B`(X).`	
$A \cap B$	P_ϕ`(X) :-` P_A`(X),` P_B`(X).`	
$A \setminus B$	P_ϕ`(X) :-` P_A`(X), not` P_B`(X).`	
$dom(A)$	P_ϕ`(X) :-` P_A`((X,_)).`	
$ran(A)$	P_ϕ`(X) :-` P_A`((_,X)).`	
A^{-1}	P_ϕ`((X,Y)) :-` P_A`((Y,X)).`	(inverse)
$A; B$	P_ϕ`((X,Z)) :-` P_A`((X,Y)),` P_B`((Y,Z)).`	(composition)
$closure1(A)$	P_ϕ`((X,Y)) :-` P_A`((X,Y)).` P_ϕ`((X,Z)) :-` P_A`((X,Y)),` P_ϕ`((Y,Z)).`	(transitive closure)

Example Translation Rules

Math vs Prolog

Set Expression ϕ	Clauses for P_ϕ	Side Condition (comment)
$\varnothing$	$P_\phi(X)$:- 1=2.	
x	0{$P_\phi(Low..Up)$}n.	$x \subseteq$ Low..Up (bounds analysis) $n = Up + 1 - Low$
$A...B$	P_ϕ(Z) :- P_A(X), P_B(Y), Z = X..Y.	
$A \cup B$	P_ϕ(X) :- P_A(X). P_ϕ(X) :- P_B(X).	
$A \cap B$	P_ϕ(X) :- P_A(X), P_B(X).	

```
:- mode trans_set(+BAST,+LocalID_Env,-ClingoPred).

trans_set(intersection(A,B),Env,Pred) :- !,
  trans_set(A,Env,P1),
  trans_set(B,Env,P2),
  gen_clingo_pred_clause(inter,Pred,[X],Env, (ecall(P1,[X],Env),ecall(P2,[X],Env)) ).
```

compute **$P_A(X)$**

compute **$P_B(X)$**

$P_\phi(X)$ **$P_A(X)$** **$P_B(X)$**

Example Translation Rules

Scalar Expressions

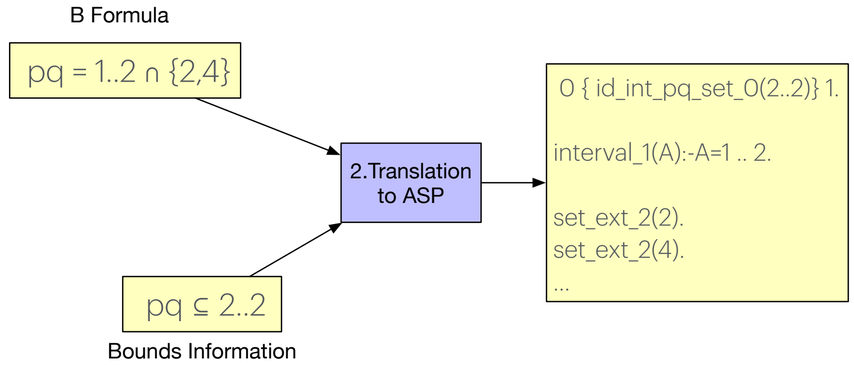
Scalar Expression ϕ	Clauses for P_ϕ	Side Condition (comment)
x	`1{`P_ϕ`(`$Low..Up$`)}1.`	$x \in$ Low..Up (bounds analysis)
n	$P_\phi(n)$.	n is an integer literal
$A + B$	P_ϕ`(Z) :-` P_A`(X),` P_B`(Y), Z=X+Y.`	
$A * B$	P_ϕ`(Z) :-` P_A`(X),` P_B`(Y), Z=X*Y.`	(similar for -,...)
$f(A)$	P_ϕ`(Y) :-` P_A`(X),` P_f`((X,Y)).`	A is scalar
$prj1(A)$	P_ϕ`(X) :-` P_A`((X,_)).`	(projection)
$prj2(A)$	P_ϕ`(X) :-` P_A`((_,X)).`	(projection)
IF A THEN B ELSE C END	P_ϕ`(X) :-` P_A`,` P_B`(X).` P_ϕ`(X) :-` P_C`(X), not` P_A.	
card(A)	P_ϕ`(X) :- X=#count{Y :` P_A`(Y)}.`	
$\Sigma(x).(x \in A \mid x)$	P_ϕ`(X) :- X=#sum{Y :` P_A`(Y)}.`	x identifier

Example Translation Rules

Predicates (as negations; aka integrity constraints)

Predicate ϕ	Clauses for P_ϕ	Side Condition
$\top$ (truth)	P_ϕ :- 1=2.	
$\bot$ (falsity)	P_ϕ.	
$A \wedge B$	P_ϕ :- P_A. P_ϕ :- P_B.	
$A \vee B$	P_ϕ :- P_A, P_B.	
$A \subseteq B$	P_ϕ :- P_A(X), not P_B(X).	
$A \in B$	P_ϕ :- P_A(X), not P_B(X).	
$A = B$	P_ϕ :- P_A(X), not P_B(X). P_ϕ :- P_B(X), not P_A(X).	A,B are sets
$A = B$	P_ϕ :- P_A(X), P_B(Y), X$\neq$Y.	A,B are scalars
$A > B$	P_ϕ :- P_A(X), P_B(Y), X$\leq$Y.	

Translation: tricky bits



- identifiers and associated cardinality constraints
- negation, need to generate integrity constraints
- Datalog rules need to be safe (finitely many ground instances on its own)
- Universal quantification: need to store in environment and pass as additional arguments to generated Clingo predicates
- Comprehension sets: new identifier generated + two universal quantifications
- Easier than initially thought: relations, as we can use pair constructor (comma) in Datalog rules

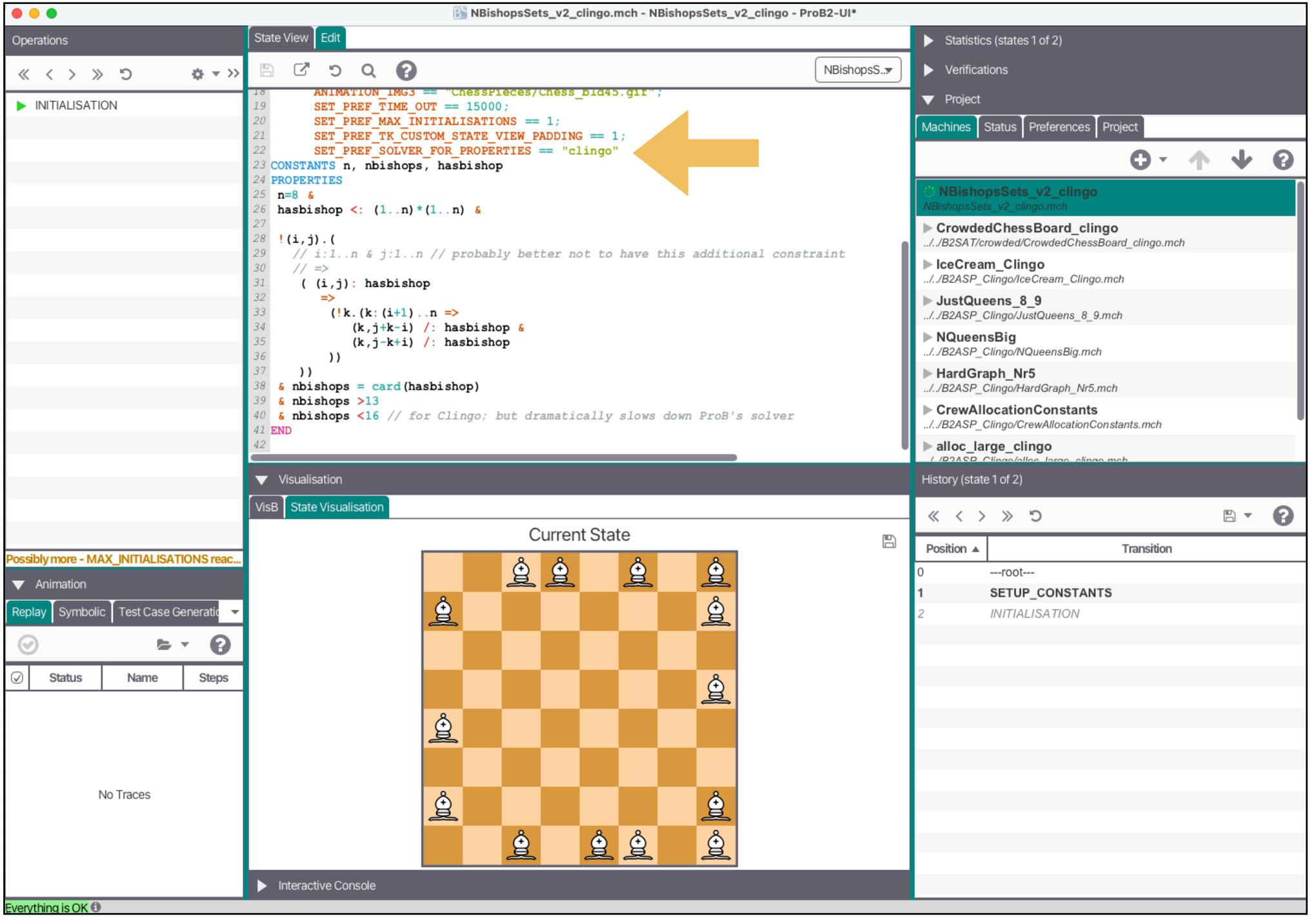
How to use B2ASP

- probcli -repl (aka console)
  - :clingo Predicate and :clingo-double-check Predicate
- Use it to solve PROPERTIES / axioms of B, TLA+, … models:
  - SET_PREF_SOLVER_FOR_PROPERTIES == “clingo”;
- As such available in ProB2-UI with various visualisations, double checking results with default solver, storing & replaying results, console for interaction, …

```
>>> :clingo  pq = 1..2 /\ {2,4}
PREDICATE is TRUE
Solution:
      pq = {2}
```

ProB2-UI

https://prob.hhu.de/



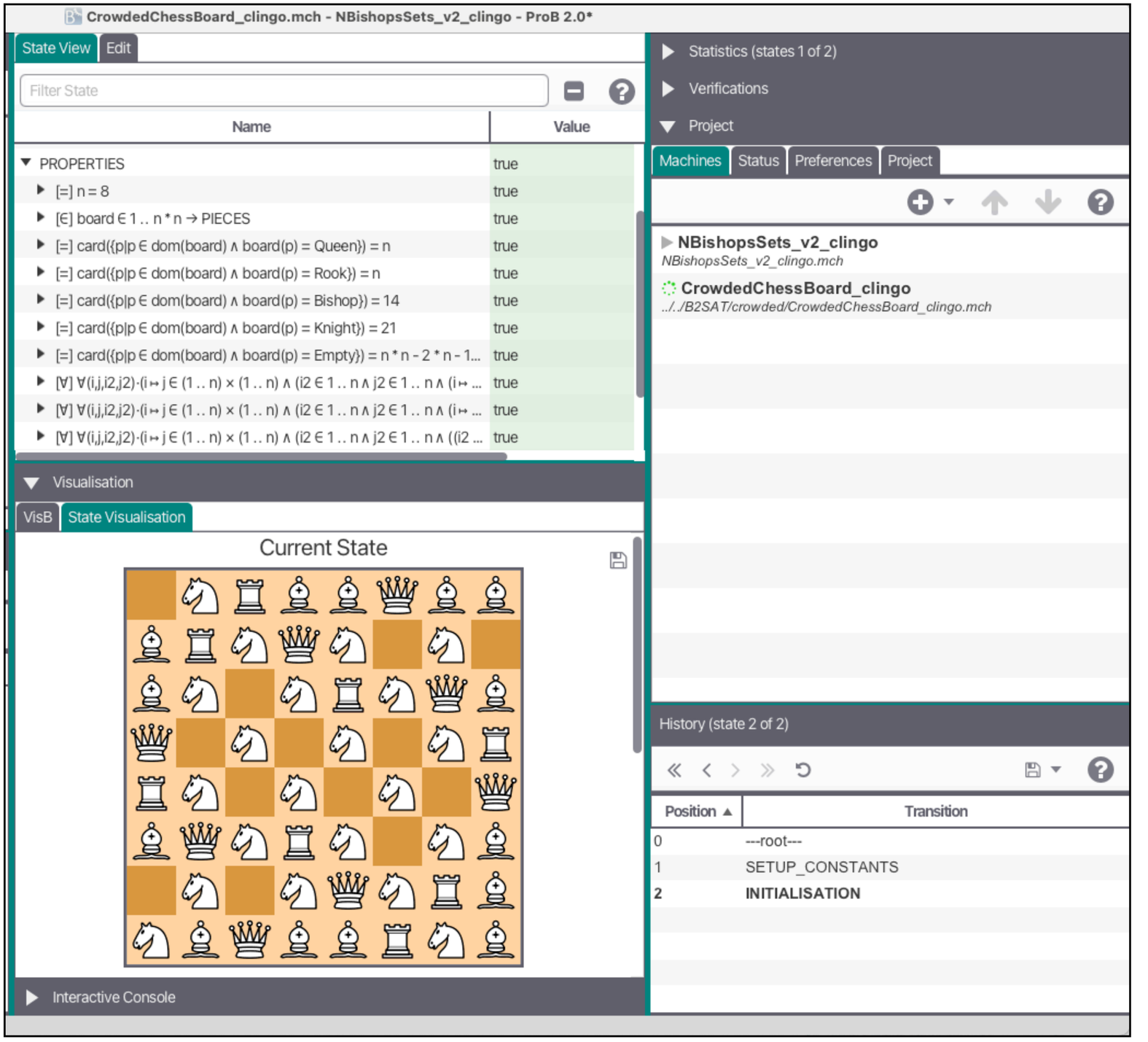
Crowded
Chessboard

Challenging Puzzle

can be solved by B2ASP

can be double checked

can be visualised



Benchmarks

Benchmark	B → ASP [1] (ms)	ASP → B [2] (ms)	Model size	Result	Total [3] (ms)
pigeon_30	3	2	91	sat	99
transitive_closure_50_bv	4	10	2707	sat	1341
blocksworld_medium	97	1	116	sat	302
blocksworld_medium_unsat	77	0	-	unsat	279
uuf-250-016	132	0	-	unsat	3702
NBishopsSet	2	1	16	sat	428
NBishopsBV	3	2	66	sat	30
WhoKilledAgatha	5	1	12	sat	26
JustQueens_8_9	8	0	38	sat	199
Loop	2	1	26	sat	31
NQueensBig	2	1	25	sat	95
IceCream_Generic	24	1	175	sat	71
DominatingSet_BV_Middle_clingo_lt13	6	2	752	sat	283
DominatingSet_BV_Middle_clingo_lt12	6	0	-	unsat	301
SendMoreMoney	3	0	-	unknown	15043

Table 1. B2ASP Backend: [1] B to ASP translation time (including bounds analysis which is at most 6 ms), [2] ASP model to B conversion time and [3] total walltime for solving formula with clingo (including [1] and [2]). Model size is expressed as the number of atoms.

solved in
172 minutes

Benchmarks

Benchmark	B2ASP		Kodkod		Default		B2SAT		Z3
pigeon_30	sat 1	✓	10.74	✓	1.80	✓	0.24	✠	149.43
transitive_closure_50_bv	sat 1	✠	0.01	✓	5.19	✓	1.12	✠	11.66
blocksworld_medium	sat 1	✓	1.15	✓	0.61	✓	0.53	✓	8.63
blocksworld_medium_unsat	unsat 1	✓	1.03	✓	0.61	✓	0.56	✓	8.28
uuf-250-016	unsat 1	✓	1.14	✠	4.16	✓	0.51	✓	1.53
NBishopsSet	sat 1	✠	0.01	✠	35.32	✠	176.42	✠	35.12
NBishopsBV	sat 1	✠	0.15	✠	457.30	✓	0.85	✠	458.39
WhoKilledAgatha	sat 1	✠	0.22	✓	0.41	✓	0.48	✠	3.89
JustQueens_8_9	sat 1	✓	346.09	✠	73.69	✓	338.36	✠	87.88
Loop	sat 1	✠	0.11	✓	0.44	✓	0.31	✠	0.78
NQueensBig	sat 1	✓	136.83	✓	0.98	✓	0.20	✠	150.91
HardGraph_Nr5	sat 1	✓	2.39	✓	0.10	✓	0.09	✓	1.85
IceCream_Generic	sat 1	✓	0.57	✓	0.48	✓	0.24	✠	298.11
DominatingSet_BV_Middle_clingo_lt13	sat 1	✓	1.97	✠	50.89	✓	0.08	✠	53.52
DominatingSet_BV_Middle_clingo_lt12	unsat 1	✓	1.69	✠	49.16	✓	0.08	✠	54.09
SendMoreMoney	✠ 1	✓	0.01	✓	0.00	✓	0.00	✓	0.00

Table 2. B2ASP compared to other backends of ProB. All runtimes are relative to B2ASP. ✠ stands for unknown, ✓ for the correct result.

Benchmarks

best

Benchmark	B2ASP	Kodkod	Default	B2SAT	Z3
pigeon_30	sat 1	✓ 10.74	✓ 1.80	✓ 0.24	✠ 149.43
transitive_closure_50_bv	sat 1	✠ 0.01	✓ 5.19	✓ 1.12	✠ 11.66
blocksworld_medium	sat 1	✓ 1.15	✓ 0.61	✓ 0.53	✓ 8.63
blocksworld_medium_unsat	unsat 1	✓ 1.03	✓ 0.61	✓ 0.56	✓ 8.28
uuf-250-016	unsat 1	✓ 1.14	✠ 4.16	✓ 0.51	✓ 1.53
NBishopsSet	sat 1	✠ 0.01	✠ 35.32	✠ 176.42	✠ 35.12
NBishopsBV	sat 1	✠ 0.15	✠ 457.30	✓ 0.85	✠ 458.39
WhoKilledAgatha	sat 1	✠ 0.22	✓ 0.41	✓ 0.48	✠ 3.89
JustQueens_8_9	sat 1	✓ 346.09	✠ 73.69	✓ 338.36	✠ 87.88
Loop	sat 1	✠ 0.11	✓ 0.44	✓ 0.31	✠ 0.78
NQueensBig	sat 1	✓ 136.83	✓ 0.98	✓ 0.20	✠ 150.91
HardGraph_Nr5	sat 1	✓ 2.39	✓ 0.10	✓ 0.09	✓ 1.85
IceCream_Generic	sat 1	✓ 0.57	✓ 0.48	✓ 0.24	✠ 298.11
DominatingSet_BV_Middle_clingo_lt13	sat 1	✓ 1.97	✠ 50.89	✓ 0.08	✠ 53.52
DominatingSet_BV_Middle_clingo_lt12	unsat 1	✓ 1.69	✠ 49.16	✓ 0.08	✠ 54.09
SendMoreMoney	✠ 1	✓ 0.01	✓ 0.00	✓ 0.00	✓ 0.00

Table 2. B2ASP compared to other backends of ProB. All runtimes are relative to B2ASP. ✠ stands for unknown, ✓ for the correct result.

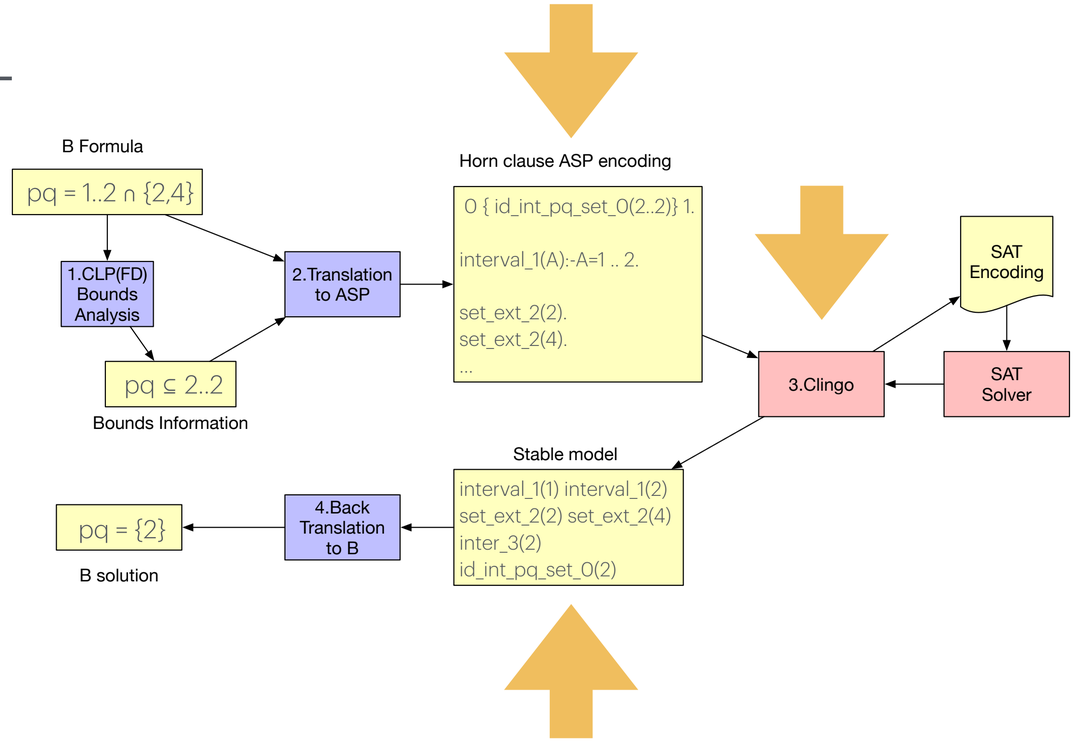
Comparison

B2ASP vs other backends

- **Z3** Backend: lots of Unknowns, often slow, not good with quantifiers (future: bounds analysis to expand quantifiers)
- **Kodkod**:
  - B2ASP faster (at least for some examples such as NQueensBig), less issues with bit-widths for integers, soundness, optimisation in principle available
- **B2SAT**:
  - B2SAT more efficient, if applicable, as it is a direct translation to SAT
  - B2ASP not interleaved with ProB solving yet
  - B2ASP has much more support for SAT translation (enumerated sets, SIGMA,...), cf. NBishopsSet which B2SAT cannot translate yet

Conclusions / Lessons 1

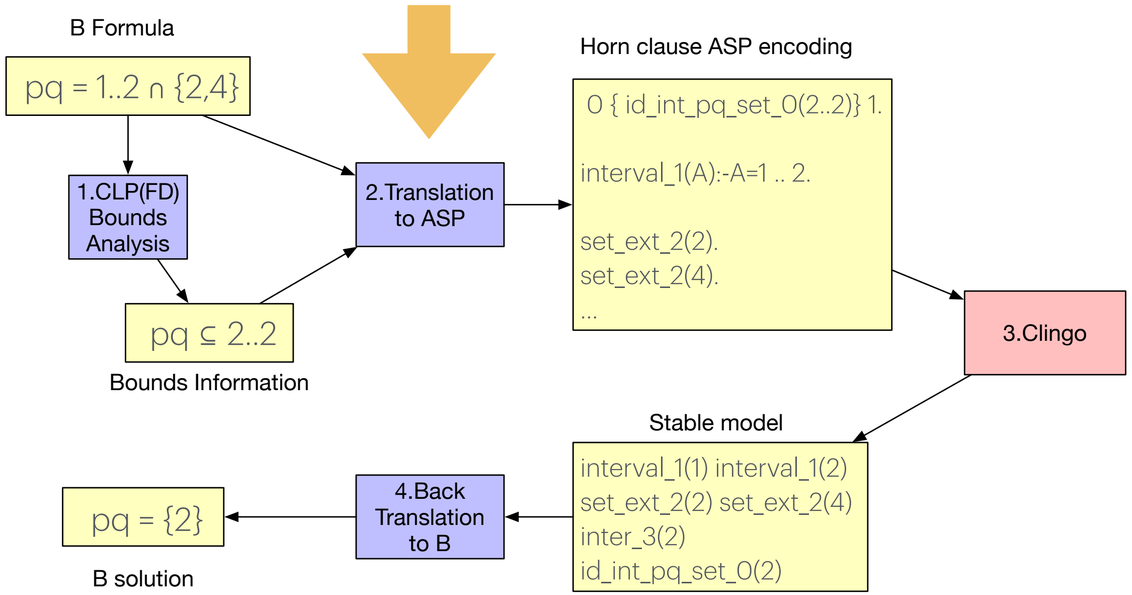
ASP as an API to SAT Solving



- ASP/clingo was a useful high-level API to SAT solving
  - (mostly) convenient to encode set theory
  - some care to create safe rules
- Pity that clingo is not directly integrated into Prolog
  - wrote DCG parser and small library to interact with clingo (via text files)
- Grounding is a bottleneck, in particular for arithmetic or relations over large base domains
  - clingcon as a remedy?

Conclusions / Lessons 2

Translation Rules in Prolog

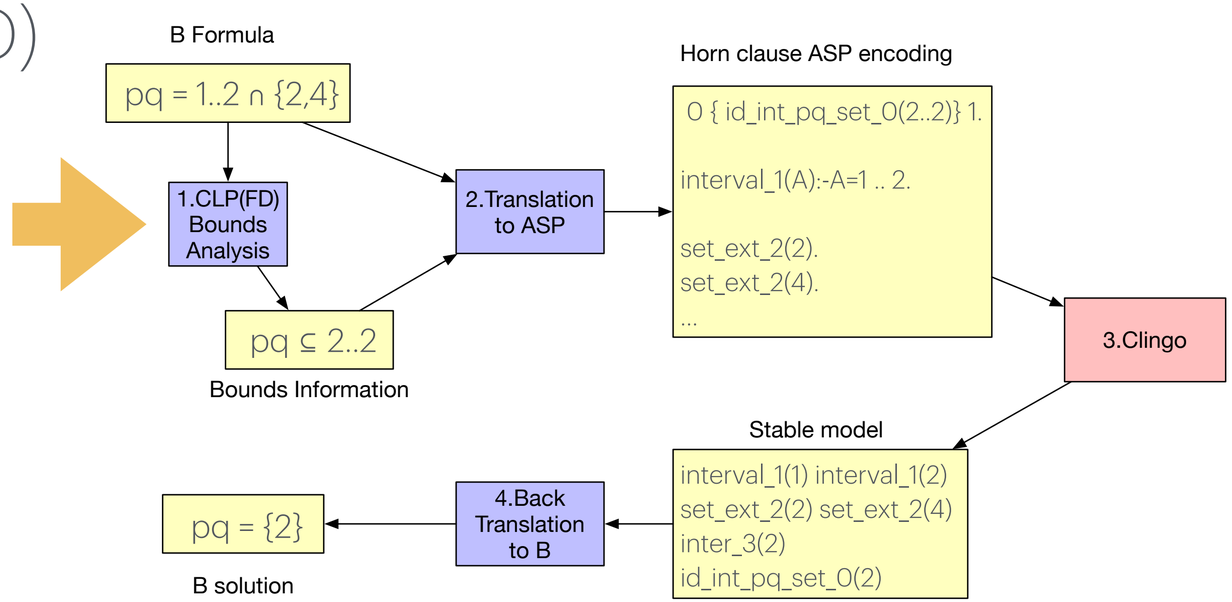


- Very convenient to express B ⇢ ASP translation rules in Prolog
  - close to mathematical formulation
  - executable, fast, easy to extend
- Pity that clingo is not directly integrated into Prolog
  - wrote DCG parser and small library to interact with clingo (via text files)
- Grounding is a bottleneck, in particular for arithmetic or relations over large base domains

Conclusions / Lessons 3

Bounds Analysis in Prolog & CLP(FD)

- Was able to re-use CLP(FD) propagation for bounds analysis
  - a bit tricky to encode empty sets, avoiding that empty intervals lead to failure of the analysis
- Fast and effective
- Can be used for other backends (Z3?) and adapted for other source formalisms

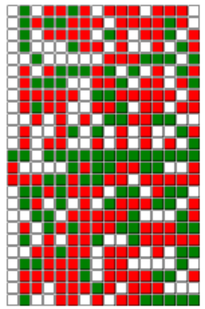


Future

- Existential quantifiers in universal quantification, more nesting allowed
- Filter out local variables as Clingo args if not needed (avoid blow up of model)
- Cache translations (avoid translating same expression multiple times)
- Scope analysis for integers (to avoid warnings and produce sound results in all cases)
- Optimisation: find optimal models for some criterion
- Interleave with ProB solving (cf B2SAT)
- No timeout yet

B2ASP

Essentials (Reprise)

{B} 

- New translation of **B** (Predicate Logic, Higher-Order Set Theory, Arithmetic) to **SAT** and back
- Available in **ProB** and applicable to other state-based formalisms (**TLA+**, **Z**, ...)
- Technique fully rooted in **Prolog** and Logic Programming: CLP(FD) Bounds Analysis, Prolog Translation from B to **ASP** (Answer Set Programming) and back
- Not a general purpose solver, but very useful for dedicated **constraint satisfaction** and **optimisation** problems and symbolic verification

Thanks for the Support

Alstom (F. Mejia,…)
ClearSy (T Lecomte, R. Lapostelle, E. Mottin,…)
Siemens
Systerel
Thales/Hitachi (N. Nayeri, G. Hemzal,…)

DFG (Gepavas I+II, IVOIRE)
EU (Rodin, Deploy, Advance)
BMBF (KILOK)

SICStus Prolog (Mats Carlsson, Per Mildner)

hhu.

STUPS Team (Düsseldorf) & Friends

Jens Bendisposto
Carl Friedrich Bolz
Michael Butler
Joy Clark
Ivo Dobrikov
Jannik Dunkelau
Nadine Elbeshausen
Fabian Fritz
Marc Fontaine
Marc Frappier
David Geleßus
Jan Gruteser
Stefan Hallerstede
Dominik Hansen
Christoph Heinzen
Yumiko Jansing
Michael Jastram
Philipp Körner
Sebastian Krings
Lukas Ladenberger
Li Luo
Thierry Massart
Daniel Plagge
Antonia Pütz
Jan Roßbach
Mireille Samia
Joshua Schmidt
David Schneider
Sherin Schneider
Corinna Spermann
Sebastian Stock
Yumiko Takahashi
Edd Turner
Miles Vella
Fabian Vu
Michelle Werth
Dennis Winter

Prolog vs B

- Horn clauses vs arbitrary quantification
- Negation as Failure vs Classical Negation
- SLD resolution vs several provers (automatic, interactive)
- User-Defined Predicates vs User-Defined Higher-Order Relations
- Terms vs Arithmetic, Sets, Relations, Functions, Sequences
- B: imperative language with proof rules, code generation
- Prolog: recursive definitions, least Herbrand model