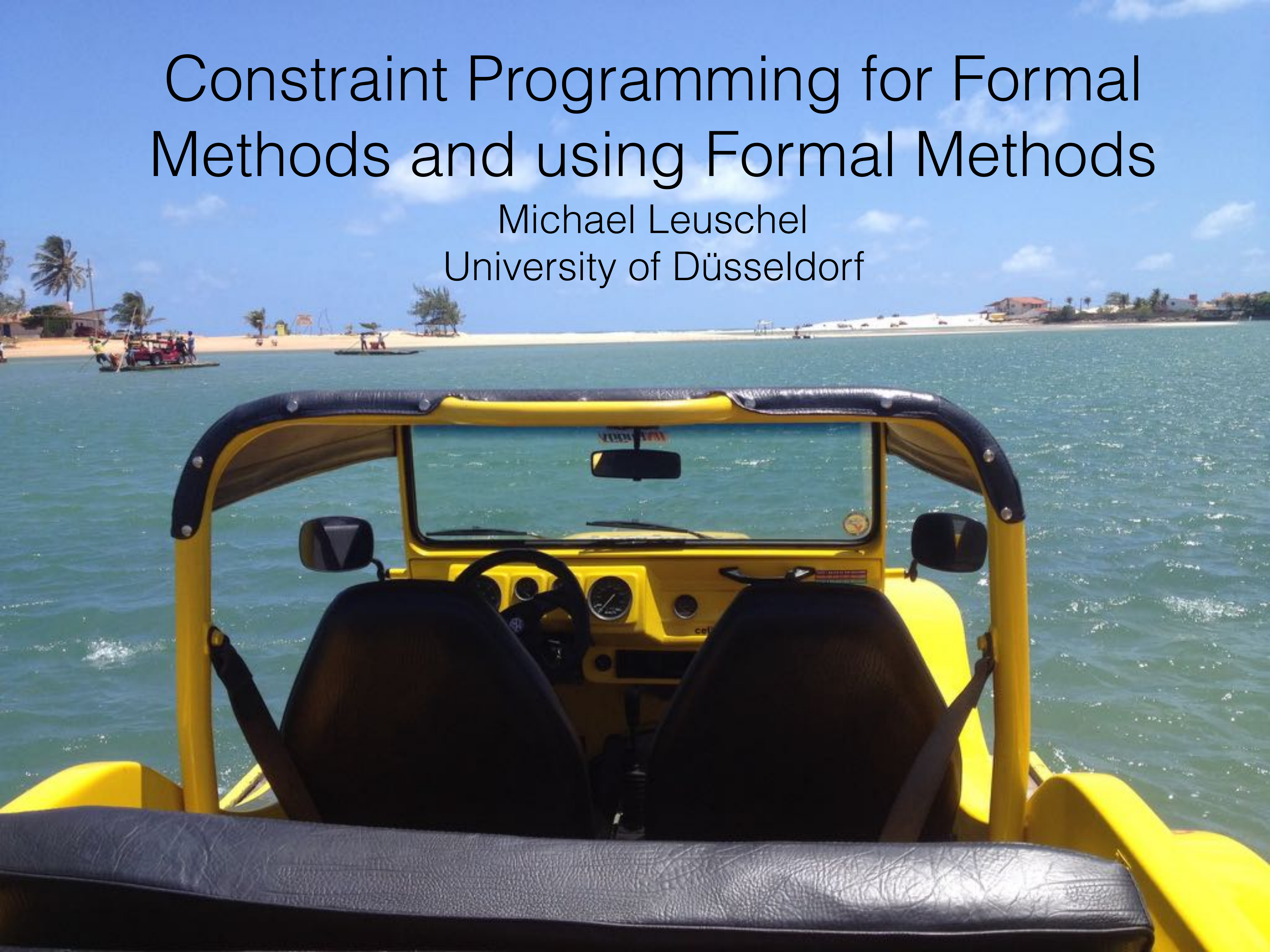
Constraint Programming for Formal Methods and using Formal Methods

Michael Leuschel
University of Düsseldorf



B-Method

Specification

Tool Support

Refinement

Logical Foundations

- Typed first-order **predicate logic**
  - Well-Definedness Conditions to stay in two-valued logic
- **Arithmetic** over mathematical integers and implementable integers
- **Set theory**
  - Sets, Relations, Functions, Sequences
  - including higher-order functions

$$p \in \mathrm{dom}(a) \rightarrowtail \mathrm{dom}(a) \land \forall i \cdot (i \in 1..(\mathrm{size}(a)-1) \Rightarrow p(a(i)) < p(a(i+1)))$$

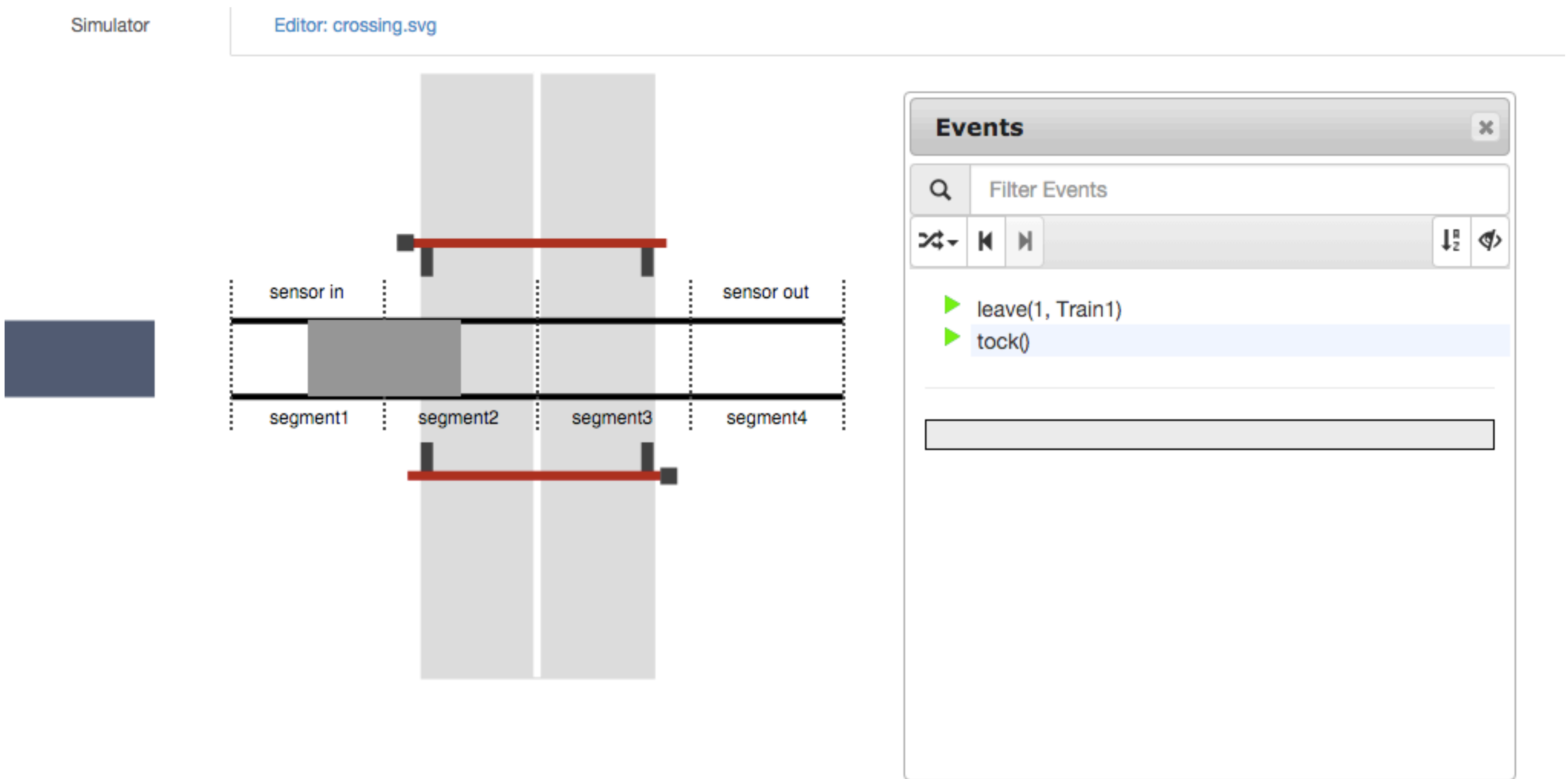
Constraint Programming for B

ProB

- **Validation** Tool for **High-level** Specifications
  - Multiple Languages: B, Z, Event-B,TLA, CSP, B||CSP
  - Multiple Validation Technologies: Animation, Model Checking, Refinement Checking, Disproving, …
  - Most rely on ProB **Constraint Solving** Kernel for B Datatypes and Operators

http://www3.hhu.de/stups/prob/

Side Note: ProB can do CSP



Constraint Solving for B

- **Execution**:
  - $\{2,3,5\} \cap (4..6) \quad \leadsto \quad \{4,5\}$
- **Proof**:
  - $x \geq 0 \wedge n > 0 \vdash x+n > 0 \quad \leadsto$ YES
- **Constraint Solving**:
  - $x \geq 0 \wedge n > 0 \wedge (x+n) \in \{2,3\} \quad \leadsto \quad x=0, n=2$

Overview of Talk

1. Applications of constraint solving **for** typical validation tasks for B (or other formal methods)

2. Ways to solve constraints involving B (or related formal methods)

3. Expressing typical constraint solving problems in B (and highlighting new Latex package)

4. Applications of expressing constraint problems **in** B

5. New integrated technique illustrated on an example

Applications of Constraint Programming for B

- Animate implicit / high-level specifications
  B4MSecure http://b4msecure.forge.imag.fr [Ledru et al. CAiSe'11]
  iUML [Snook et al.], Alstom Interlocking Simulator [Mejia et al.]
  CODA [Butler, Colley, Edmunds, Snook, Evans, Gran, Marshall]

- Constraint-based invariant, deadlock, refinement checks (ICLP'11)

- Model-Based Testing [SEFM'15], Beta [de Matos, Moreira, Neto 2012], [Moreira et al. TAP'15], BTestBox (https://github.com/ValerioMedeiros/BTestBox), [Dinca,Ipate,Stefanescu ABZ'12]

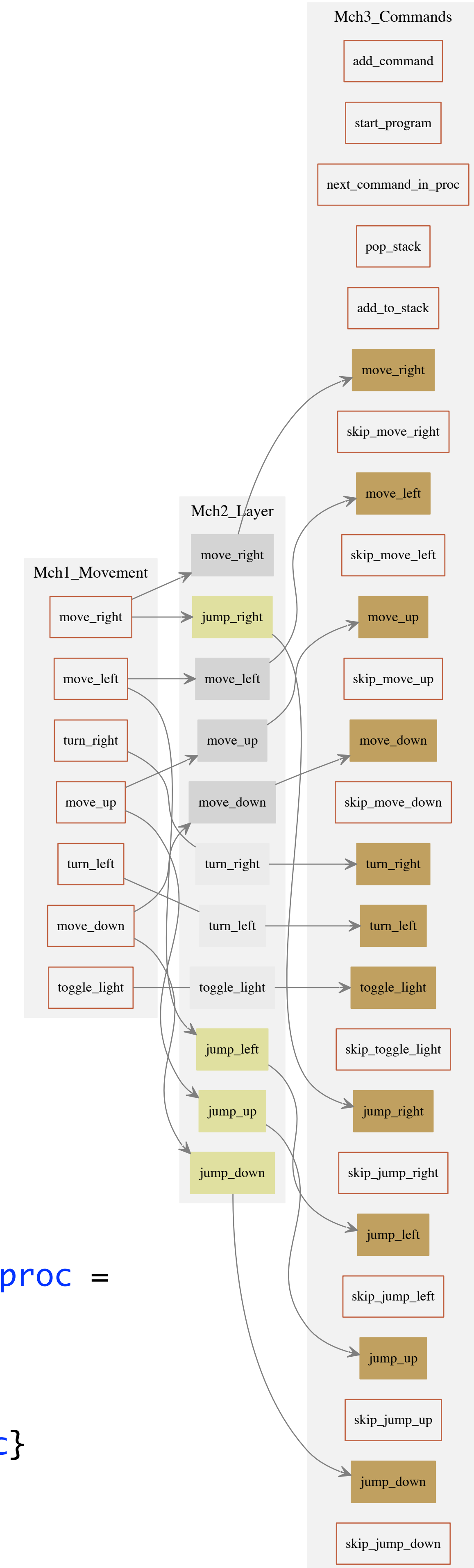
- Disprover/Prover for B and Event-B (SEFM'15), BEval [Medeiros,Deharbe]

- Enabling Analysis (ABZ'16)

- Symbolic Model Checking (BMC,k-Induction, IC3) (ABZ'16),
  Genesis [Radhouani, Idani, Ledru, Rajeb 15] http://genisis.forge.imag.fr

Lightbot

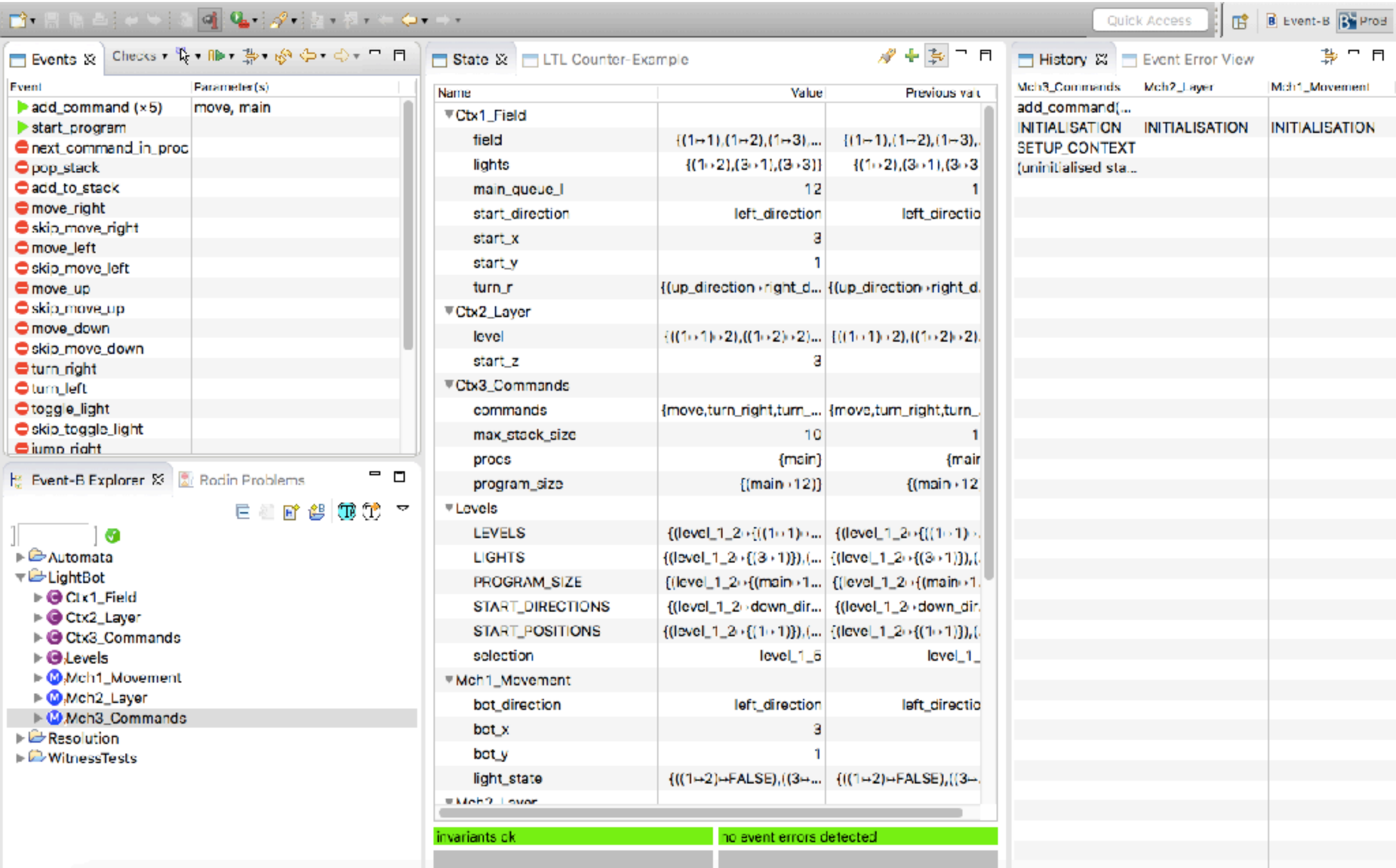
Event-B model(s)
used in teaching



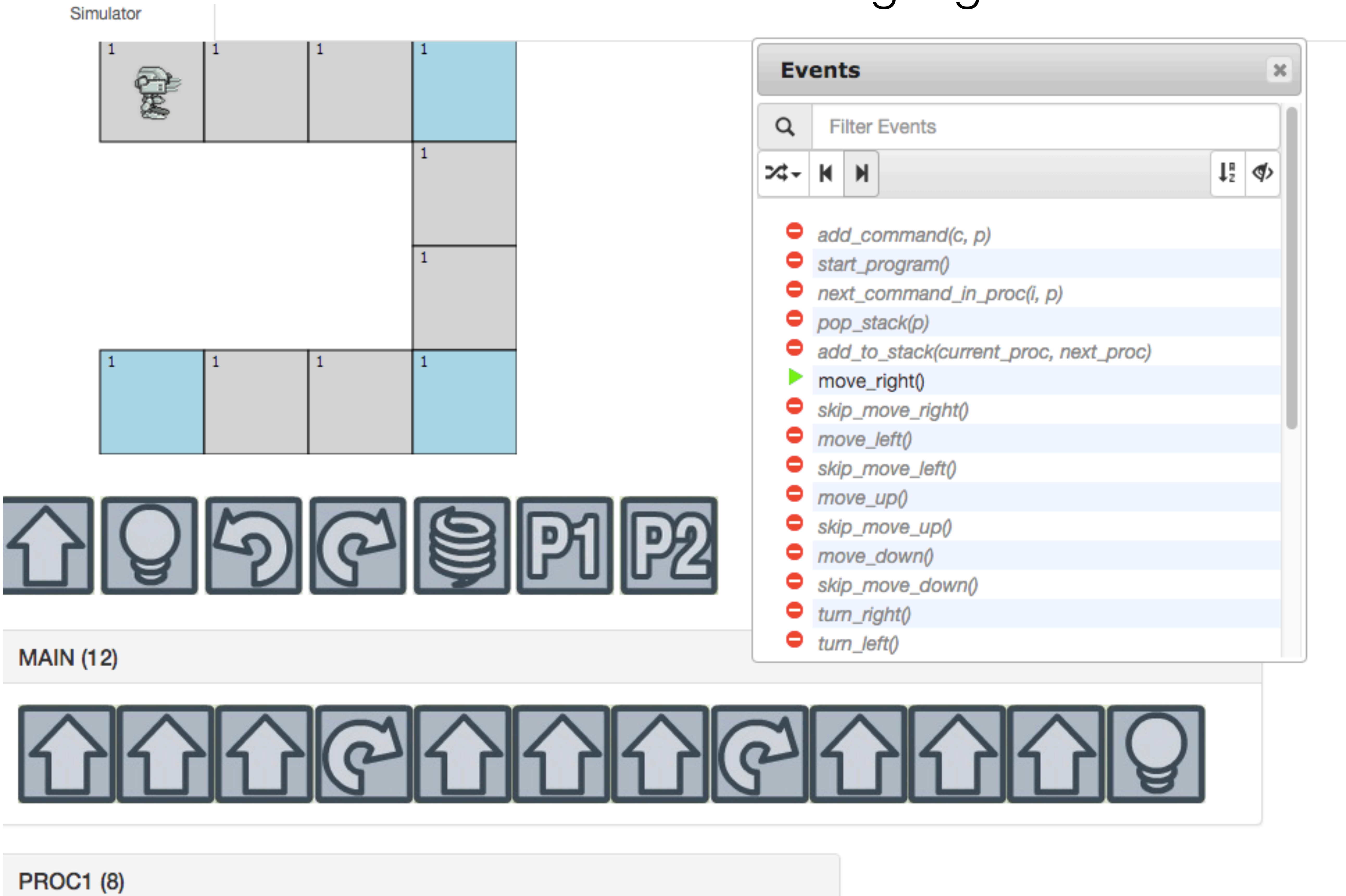
one event of third refinement shown:

```
  …
event add_to_stack
    any current_proc next_proc
    where
      @next_proc1 next_proc ∈ procs
      @next_proc2 current_command = next_proc
      @next_proc3 programs(next_proc) ≠ ∅
      @stack_size card(program_stack) < max_stack_size
      @current_proc current_proc ∈ procs ∧ current_proc ∈ ran(program_stack) ∧ current_proc =
program_stack(card(program_stack))
    then
      @act2 current_command ≔ null
      @current_proc2 program_stack ≔ program_stack ∪  {card(program_stack)+1↦ next_proc}
      @act3 index(next_proc) ≔ 0
  end
..
```

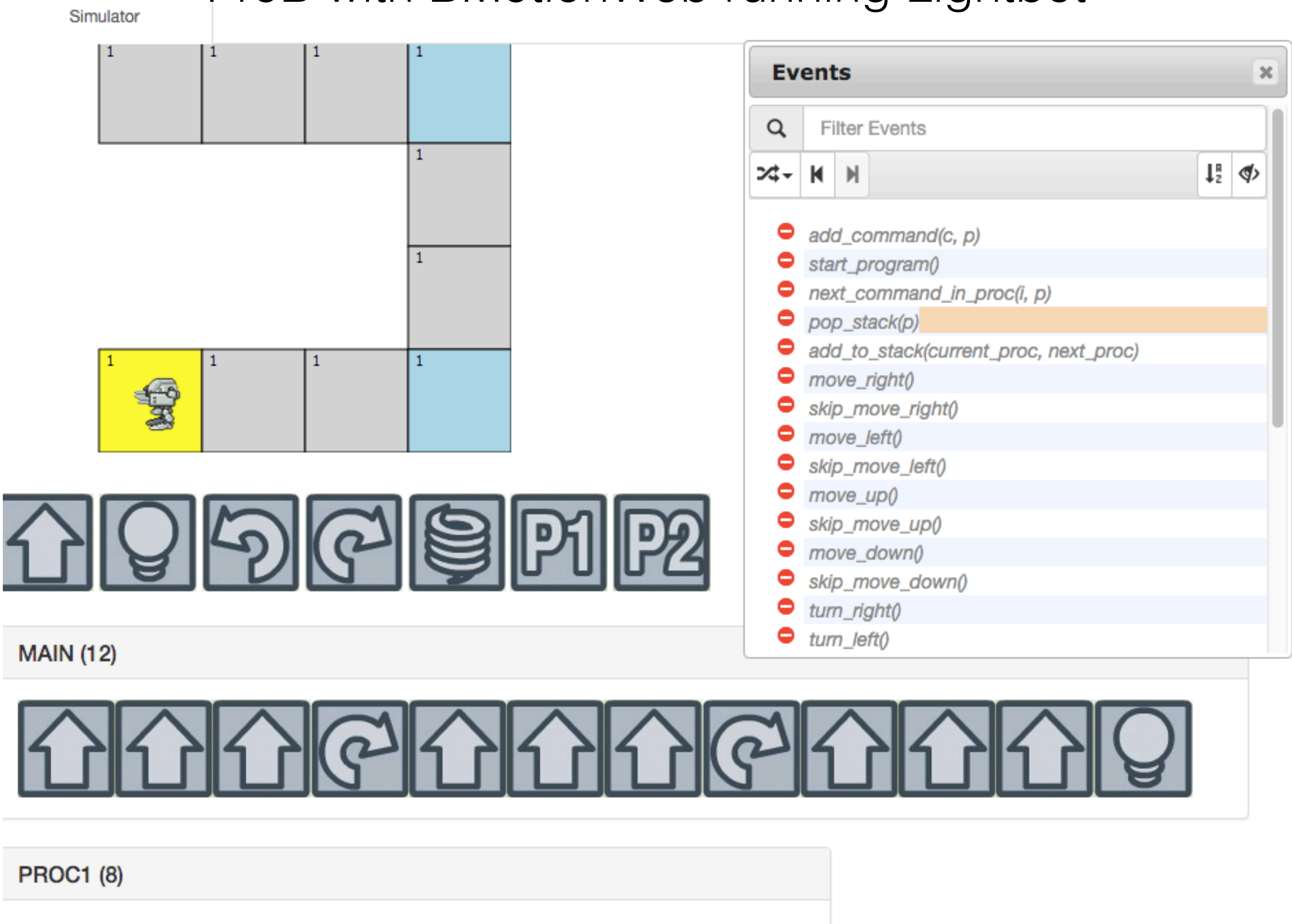
ProB Animator in Rodin for Lightbot



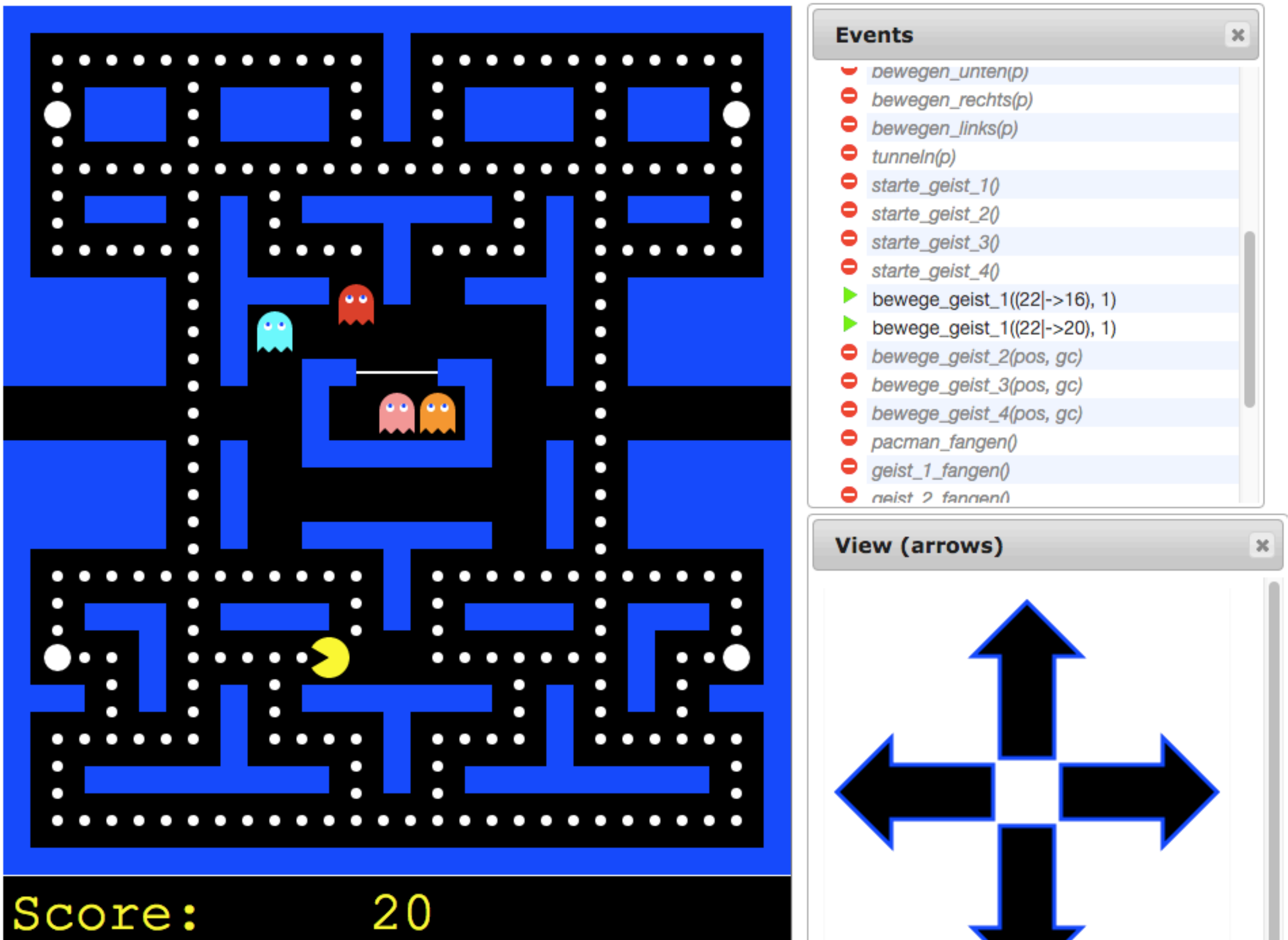
ProB with BMotionWeb running Lightbot



ProB with BMotionWeb running Lightbot



ProB with BMotionWeb for Pac Man



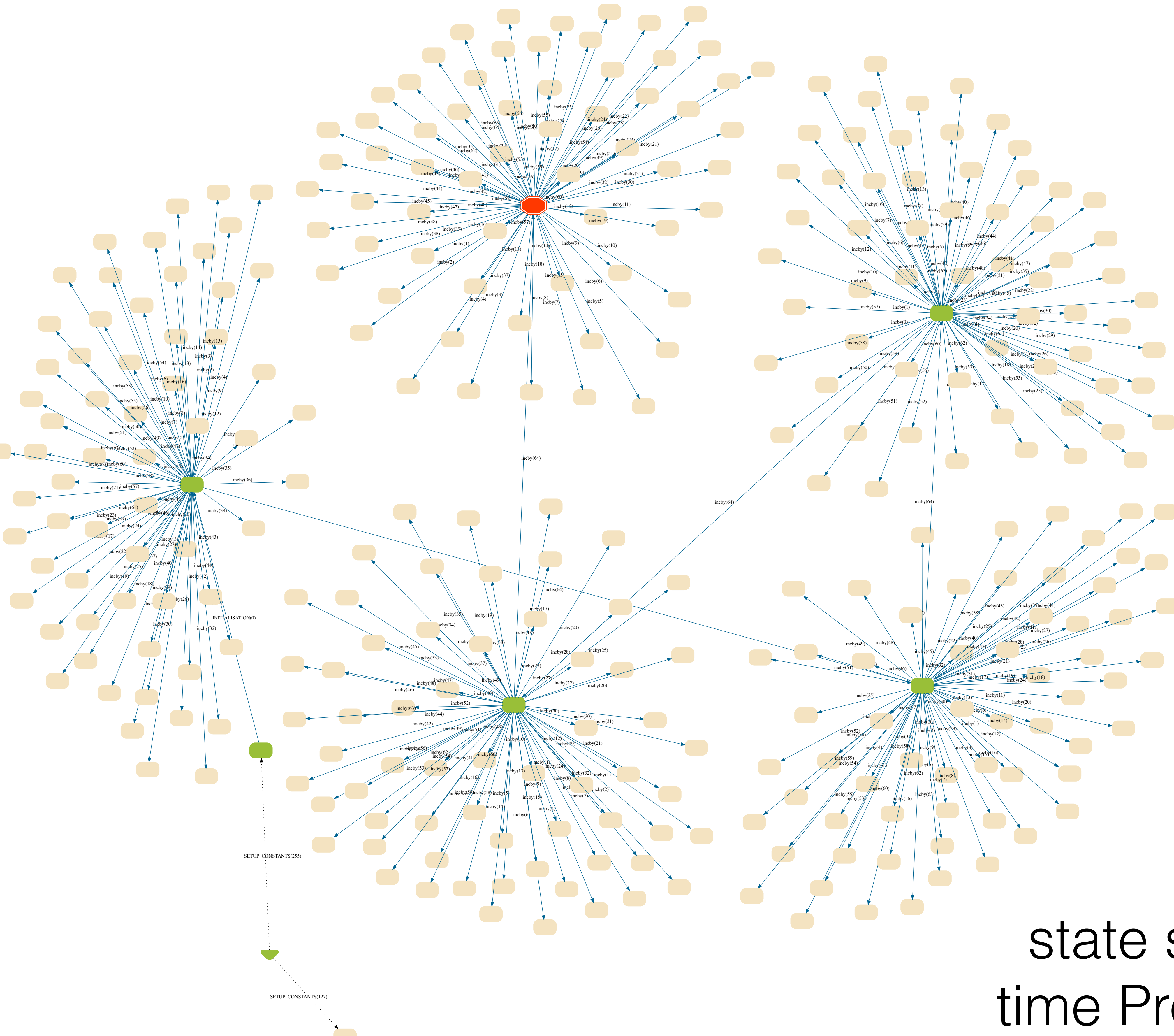
Animation: Need for Constraint Solving

```
IXL_DC_cmd_signal_permissive_aspect =
ANY
    p_signal1
,   p_signal2
,   p_blocks_path
WHERE
    p_signal1 : t_signal
&   p_signal2 : t_signal
&   p_blocks_path : POW (t_oriented_block)
&   vi_IXL_DC_cmd_signals (p_signal1) = c_green
&   v1_IXL_DC_cmd_signals (p_signal1) = c_red

/* we search a path with : sig1 -> sig2 */
&   p_blocks_path = closure (v1_IXL_DC_next_block |>> {c_downstream_block_of_signal (p_signal2)}) [{c_downstream_block_of_signal (p_signal1)}]
&   c_upstream_block_of_signal (p_signal2) : p_blocks_path

/* not in this case :  sig1 -> sig -> sig2 */
&   ! (sig). (sig : t_signal - {p_signal2} => c_upstream_block_of_signal (sig) /: p_blocks_path)

/* conjugated points of the path's points are positionned */
&       c_next_block_with_point
        [   UNION(p_pt).
            (   p_pt : c_next_block_with_point~ [v1_IXL_DC_next_block |> (p_blocks_path- {c_downstream_block_of_signal (p_signal1)})]
            |   c_conjugated_points (p_pt)
            )
        ]
    <:  v1_IXL_DC_next_block

/* protection points of the path's points are positionned */
&       c_next_block_with_point
        [   UNION(p_pt).
            (   p_pt : c_next_block_with_point~ [v1_IXL_DC_next_block |> (p_blocks_path- {c_downstream_block_of_signal (p_signal1)})]
            |   closure1 (c_protection_point) [{p_pt}]
            )
        ]
    <:  v1_IXL_DC_next_block

/* path's blocks are booked */
&   p_blocks_path    <: v1_IXL_DC_booked_blocks

/* upstream block of signal is booked */
&   c_upstream_block_of_signal (p_signal1) : v1_IXL_DC_booked_blocks

/* all trains in the turnaround area of the considered signal are safely stopped in the exit direction */
&   (   c_turnaround_blocks_of_signal (p_signal1) /\ v1_IXL_DC_reversal_blocks /= {}
    =>
        c_turnaround_blocks_of_signal (p_signal1) /\ c_underlying_sdd_of_oriented_block~ [vi_IXL_DC_ctl_sdds] <: v1_IXL_DC_reversal_blocks
    &   c_opposite_blocks [c_turnaround_blocks_of_signal (p_signal1) /\ c_underlying_sdd_of_oriented_block~ [vi_IXL_DC_ctl_sdds]] <: v1_IXL_DC_traffic_direction
    &   c_underlying_block [c_turnaround_blocks_of_signal (p_signal1) /\ c_underlying_sdd_of_oriented_block~ [vi_IXL_DC_ctl_sdds]] <: vi_IXL_DC_msg_tios
    )
THEN
    v1_IXL_DC_cmd_signals (p_signal1) := c_green
||  v1_IXL_DC_reversal_blocks := v1_IXL_DC_reversal_blocks - (c_turnaround_blocks_of_signal (p_signal1) /\ v1_IXL_DC_reversal_blocks)
END
```

Test-Case Generation and Bounded Model Checking

- model checking and test-case generation can be done on explicit state space
- problem when high-branching factor:
  - constant valuation,
  - operation parameters, …
- simple example:

```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN c := c+i END
END
```

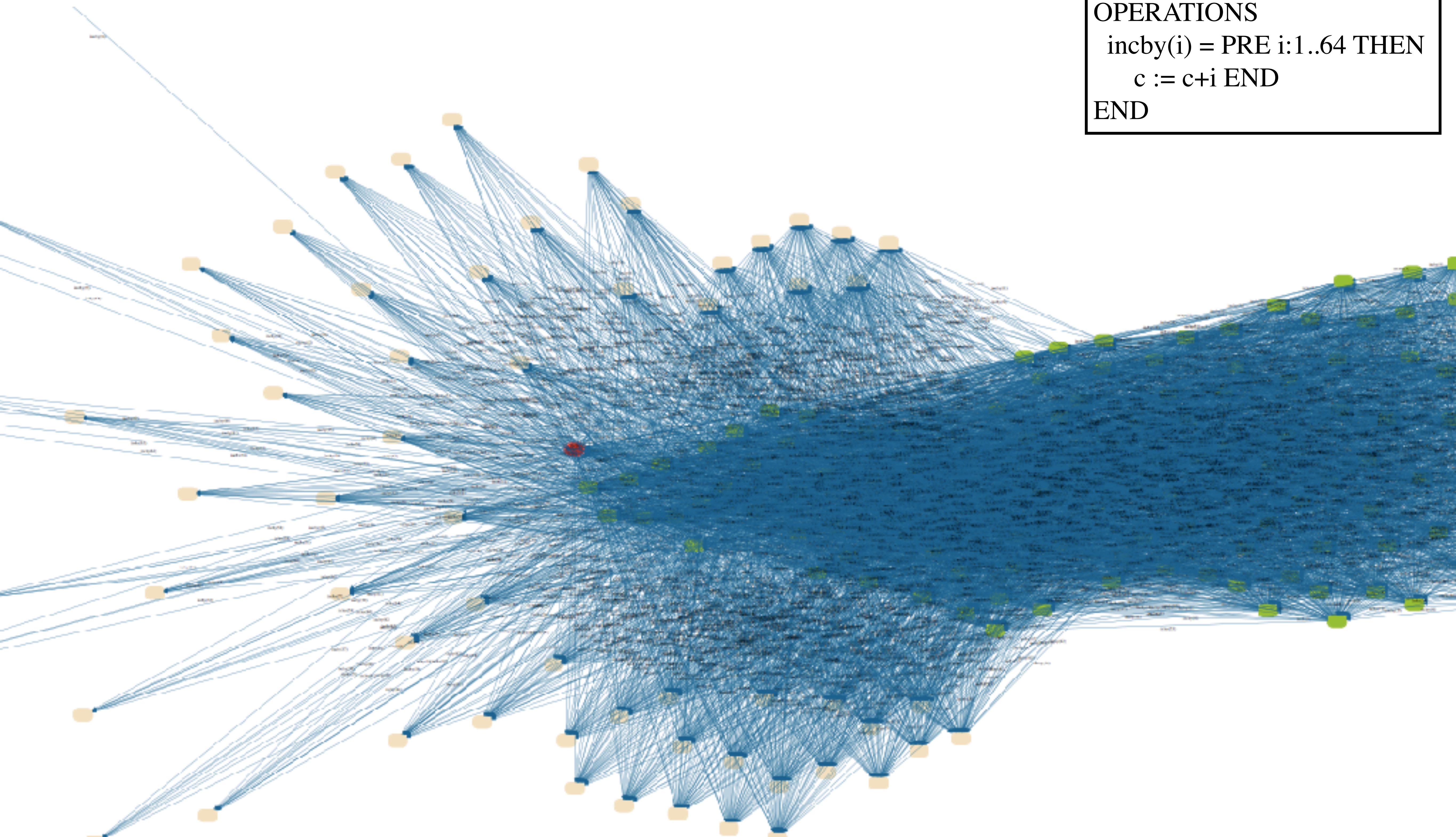



```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN
      c := c+i END
END
```

state space at the time ProB finds error (default mixed bf/df)

with breadth-first even worse
part of explored state space:

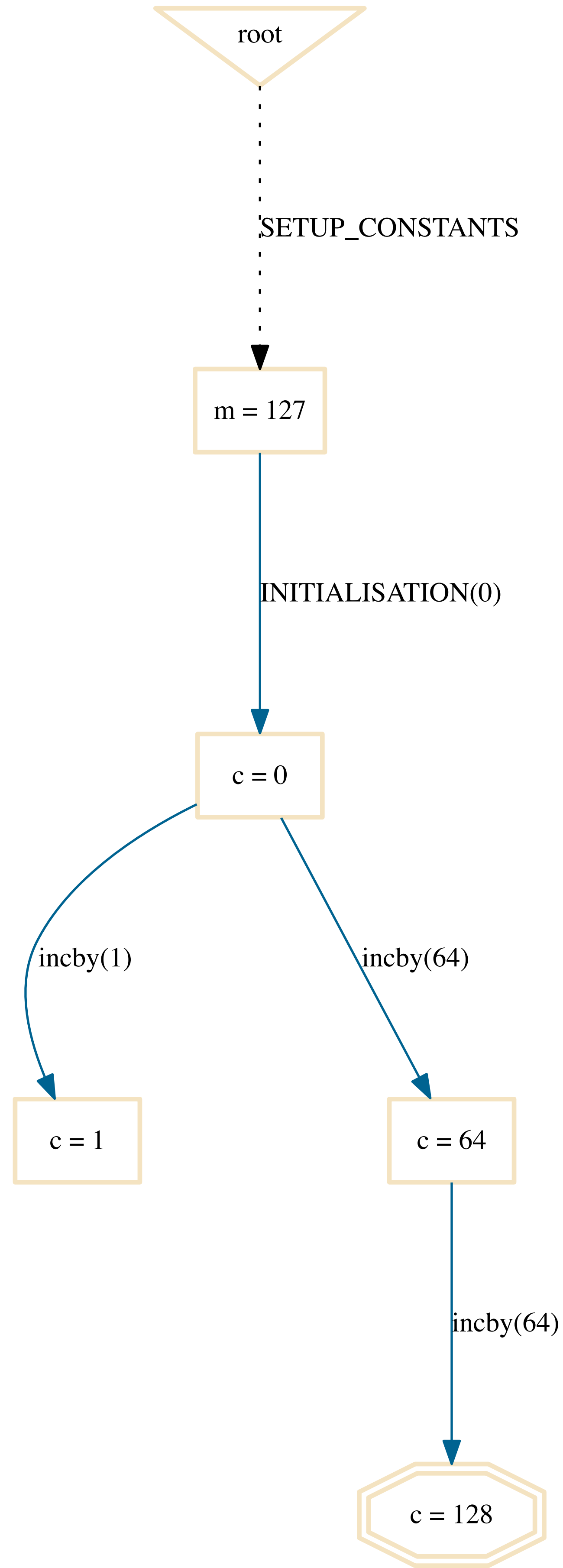
```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN
      c := c+i END
END
```



Idea

- do not enumerate all possible constant values (m) and all parameters (i)
- let constraint solver instantiate them depending on target:
    - test-case generation: execute all/one event leading to a target predicate (valid end state of a test)
    - bounded-model checking (BMC): execute a sequence of events leading to invariant violation

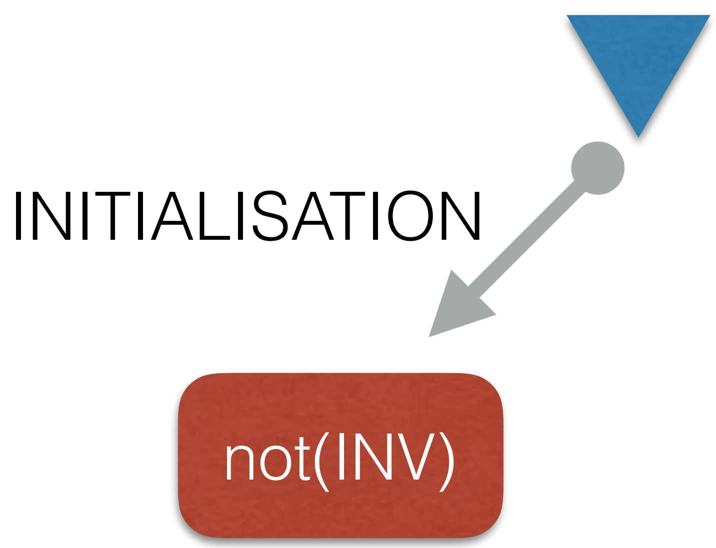
probcli Counter.mch -bmc 10 -spdot out.dot




```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN
     c := c+i END
END
```

BMC Algorithm: Steps

```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN
      c := c+i END
END
```

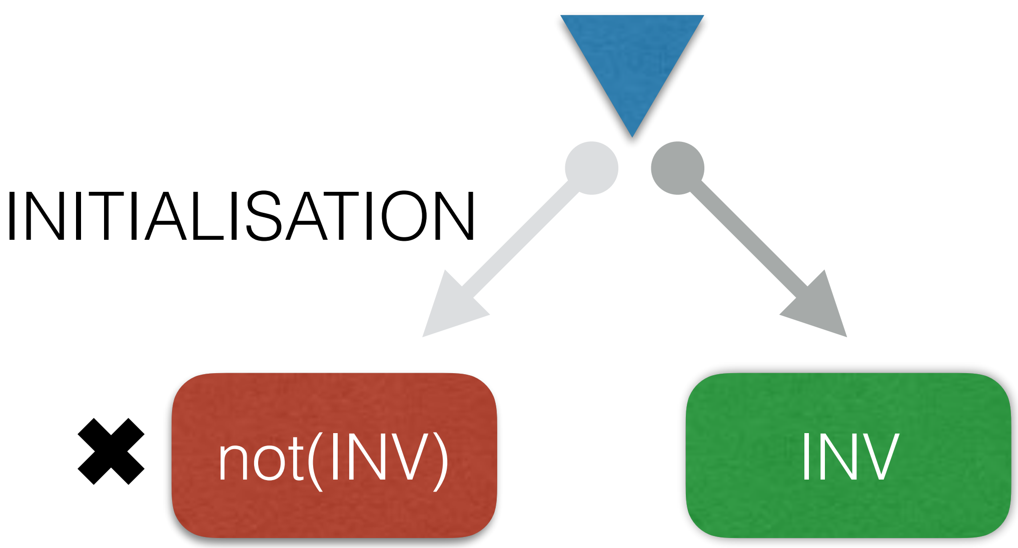


Call to solver:

$$m \in \{127,255\} \wedge c=0 \wedge \neg(c \geq 0 \wedge c \leq m)$$

BMC Algorithm: Steps

```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN
      c := c+i END
END
```

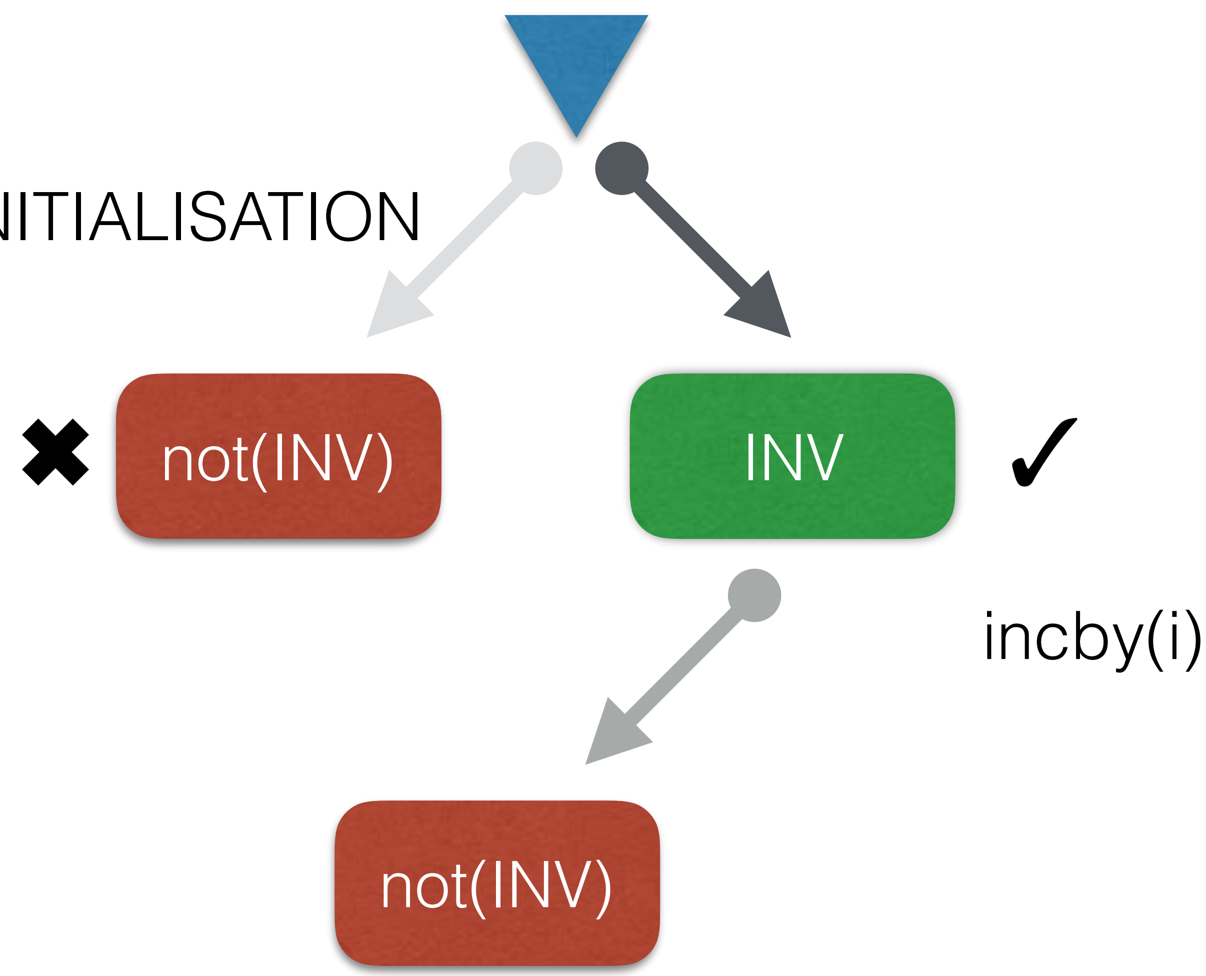


Call to solver:

$m \in \{127,255\} \wedge c=0$

BMC Algorithm: Steps

```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN
      c := c+i END
END
```

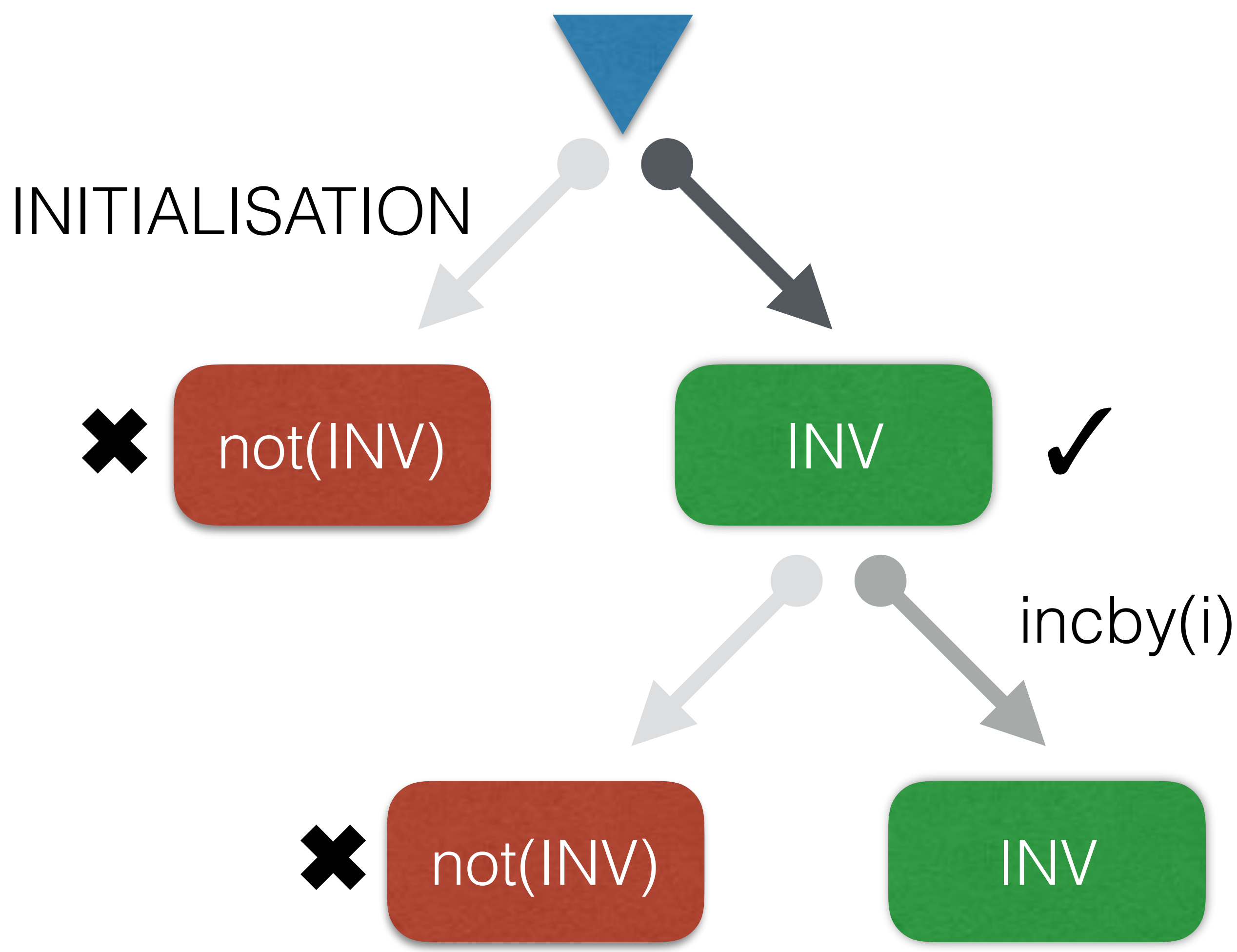


Call to solver:

$m \in \{127,255\} \wedge c=0 \wedge (c \geq 0 \wedge c \leq m) \wedge$
$i \in 1..64 \wedge c'=c+i \wedge \neg(c' \geq 0 \wedge c' \leq m)$

BMC Algorithm: Steps

```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN
      c := c+i END
END
```

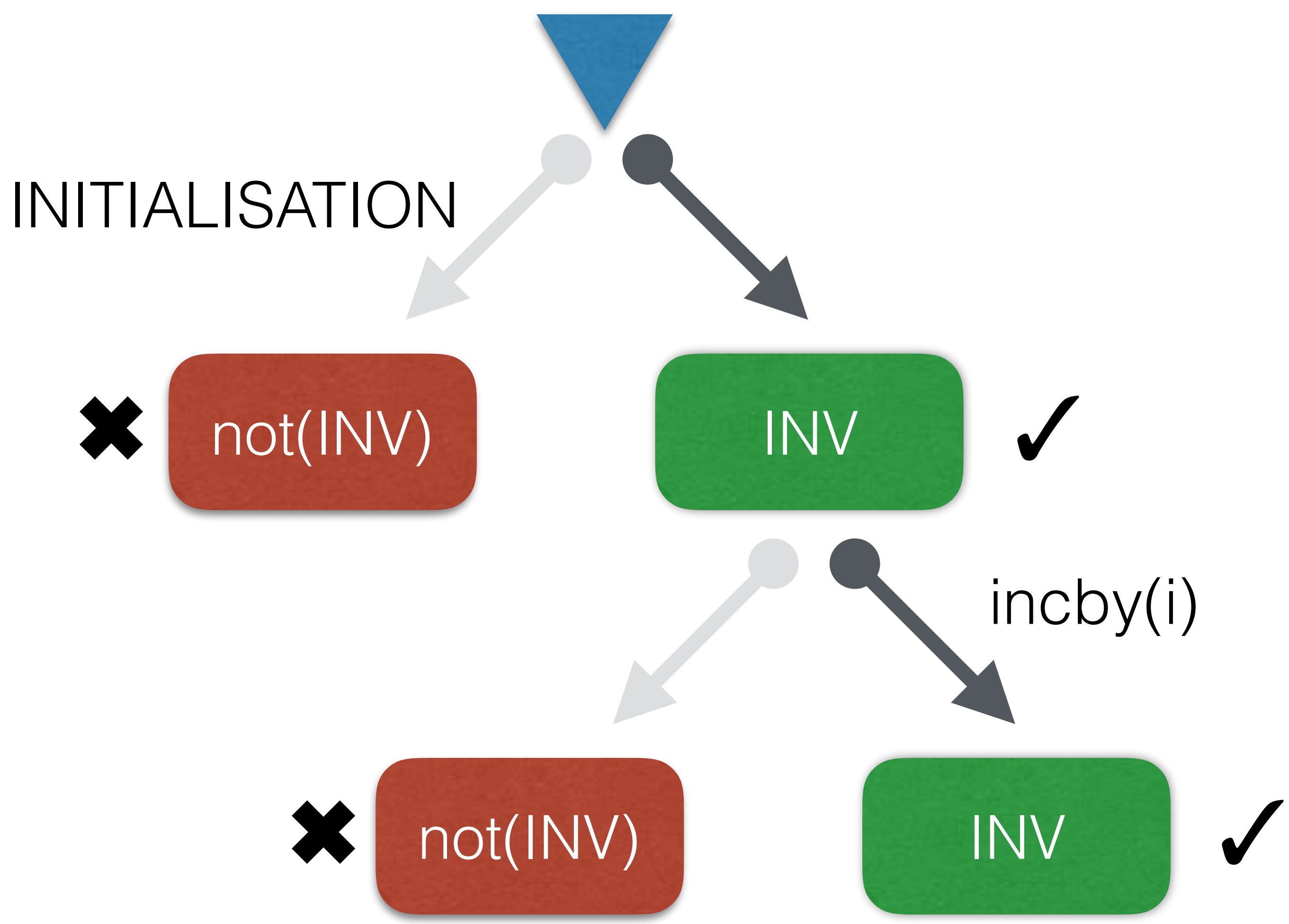


Call to solver:

$m \in \{127,255\} \wedge c=0 \wedge (c \geq 0 \wedge c \leq m) \wedge$
$i \in 1..64 \wedge c'=c+i$

BMC Algorithm: Steps

```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN
      c := c+i END
END
```



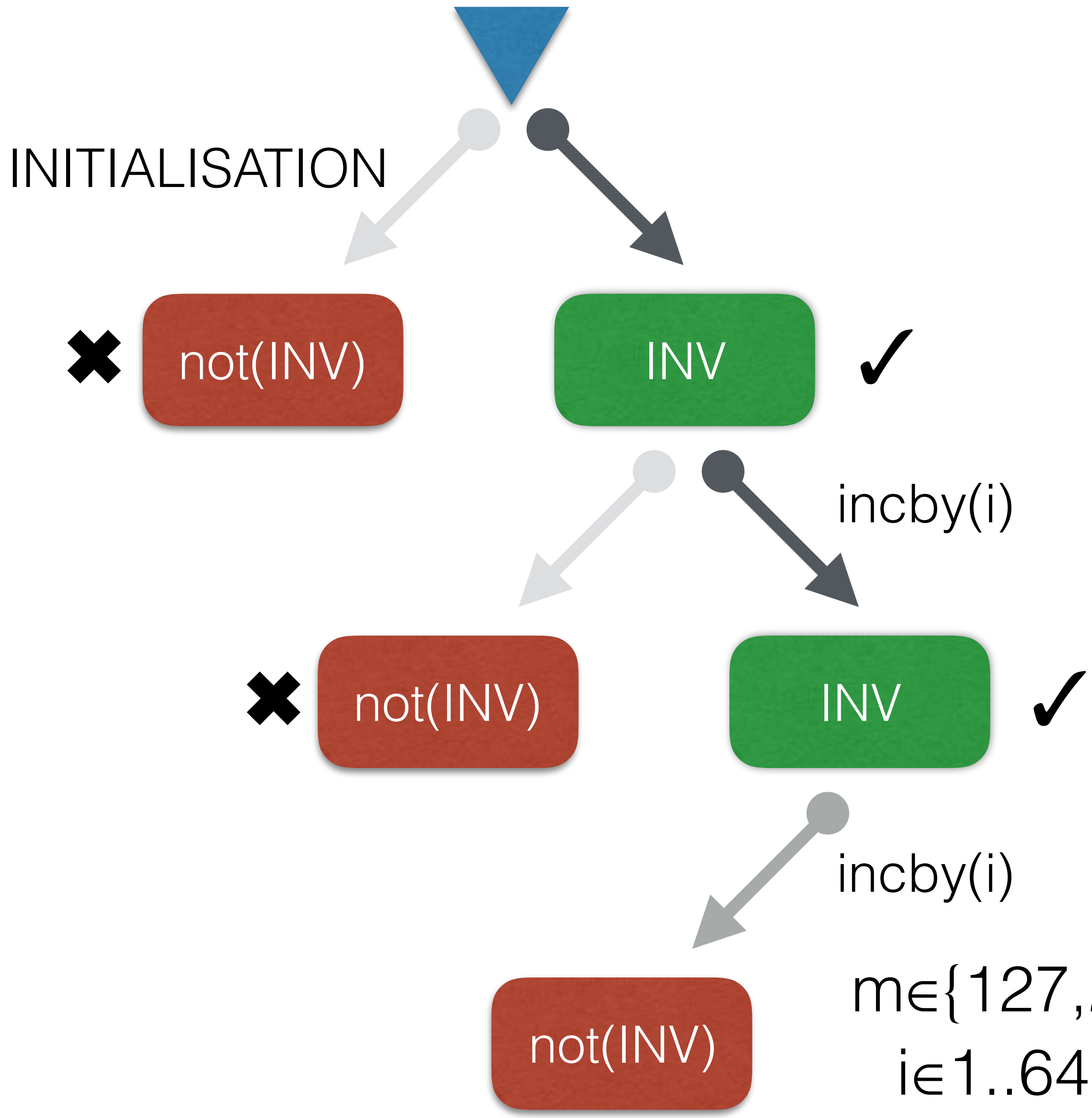
INVARIANT
$\rightsquigarrow$ incby
$\rightsquigarrow$ guard of incby ?

Call to solver:

$$m \in \{127,255\} \wedge (c \geq 0 \wedge c \leq m) \wedge$$
$$i \in 1..64 \wedge c' = c+i \wedge$$
$$i' \in 1..64$$

BMC Algorithm: Steps

```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN
      c := c+i END
END
```



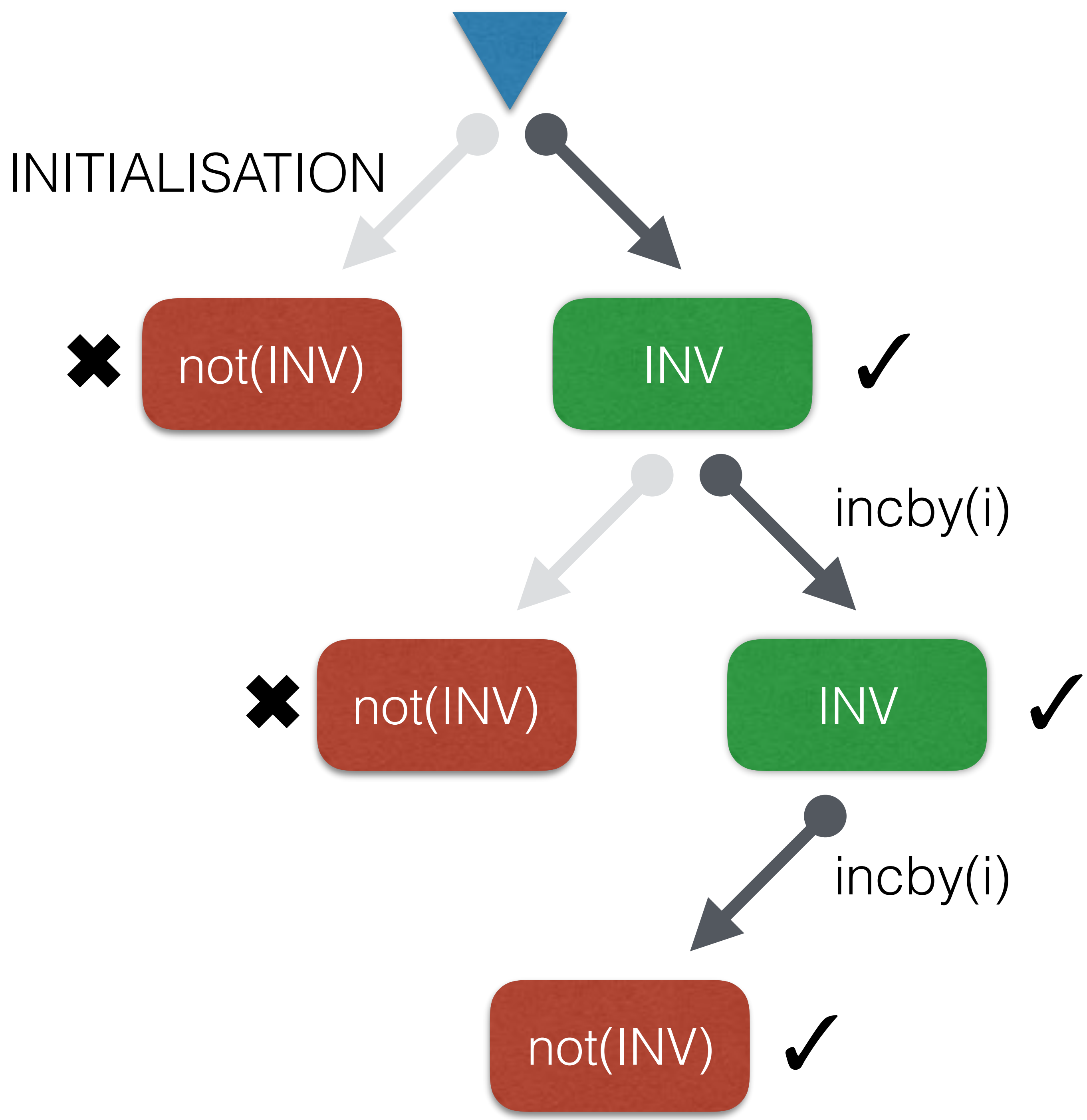
incby $\rightsquigarrow$ incby ✓

Call to solver:

$$m \in \{127,255\} \wedge c=0 \wedge (c \geq 0 \wedge c \leq m) \wedge$$
$$i \in 1..64 \wedge c' = c+i \wedge (c' \geq 0 \wedge c' \leq m) \wedge$$
$$i' \in 1..64 \wedge c'' = c'+i' \wedge \neg(c'' \geq 0 \wedge c'' \leq m)$$

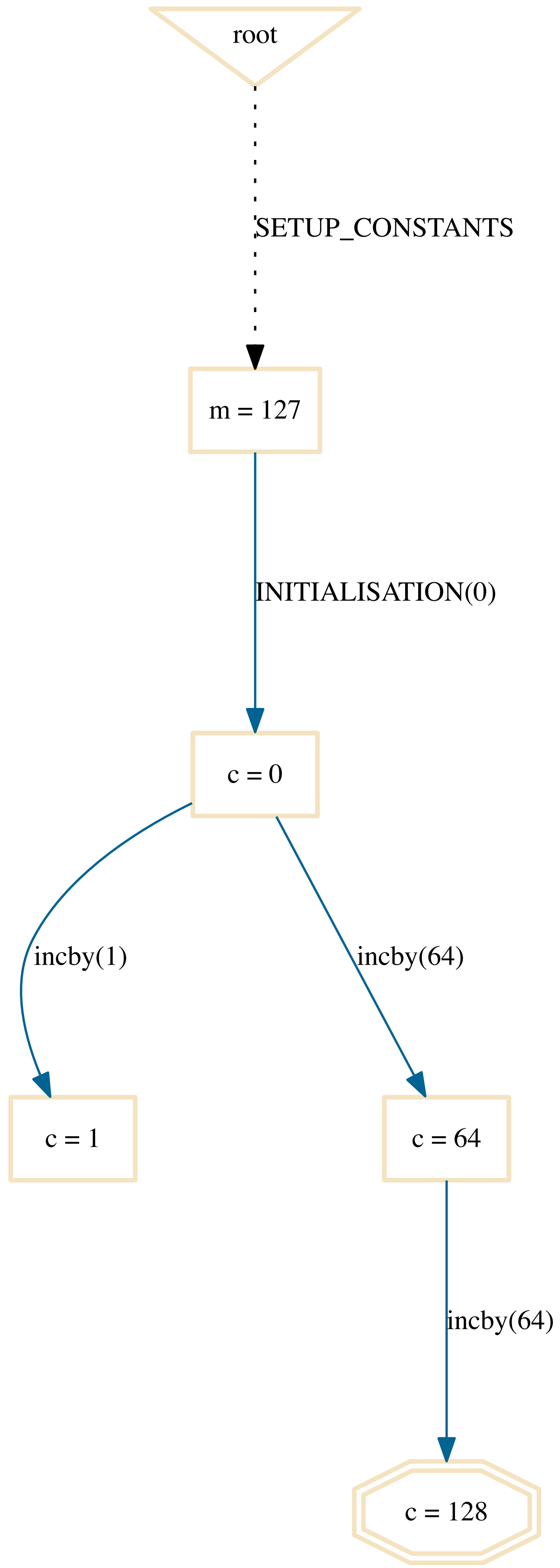
BMC Algorithm: Steps

```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN
      c := c+i END
END
```



incby ⇝ incby ✓

Counterexample found !



Counterexample

```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN
      c := c+i END
END
```

Call to solver:

$m \in \{127,255\} \wedge c=0 \wedge (c \geq 0 \wedge c \leq m) \wedge$
$i \in 1..64 \wedge c' = c+i \wedge (c' \geq 0 \wedge c' \leq m) \wedge$
$i' \in 1..64 \wedge c'' = c'+i' \wedge \neg(c'' \geq 0 \wedge c'' \leq m)$

Solution:

$m=127$

$c=0$

$i=64,\ c'=64$

$i'=64,\ c''=128$

BMC/Test-Case Algorithm

1. **Input:** set of events $target \subseteq Events$ and a set of events $final \subseteq Events$
2. Initially we set $paths := \{[\,]\}$ where $[\,]$ is the path of length 0
3. $depth := 0$
4. $final$ is the set of events which have to be final; in our case study: $target$
5. $target' := target$
6. **for** every $p \in paths$ of length $depth$ **do**:
7.   **for** every $t \in target \cap enable(p)$ **do**:
8.   **if** solve constraints of path $p \leftarrow t$ **then**
9. $target := target \setminus \{t\}$, store solution as test for t
10. $path := path \cup \{p \leftarrow t\}$
11.   **fi**
12.   **od**
13. **od**
14. **if** $target = \varnothing$ or maximum depth reached **then** return **fi**
15. **for** every $p \in paths$ of length $depth$ **do**:
16.   **for** every $e \in feasibleAfter(p) \setminus target' \setminus final$ **do**:
17.   **if** solve constraints of path $p \leftarrow e$ **then** $path := path \cup \{p \leftarrow e\}$ **fi**
18.   **od**
19. **od**
20. $depth := depth + 1$; **goto** 5

Application

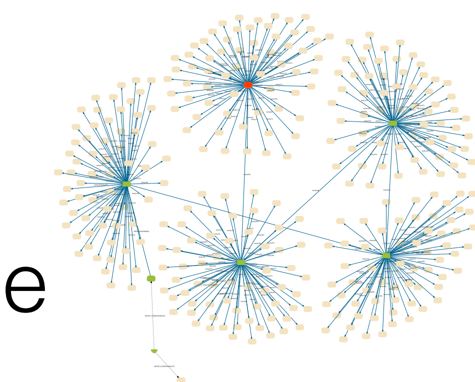
- More details about an application for Java Card security in [SEFM’15] paper
  - runtime reduced from estimated 2 years to 45 minutes for creating all relevant mutants

Demo:
Example from yesterday

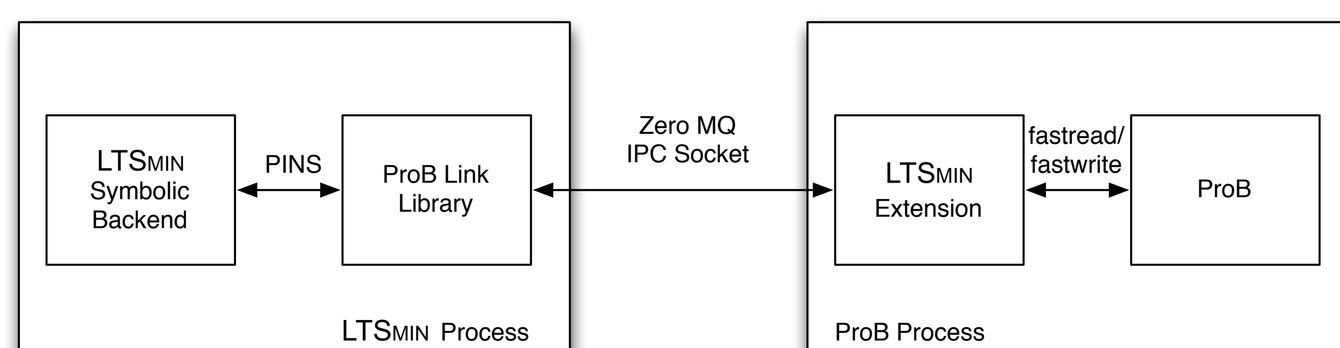
```
MACHINE EmailTestCaseGenExample
SETS User = {u1,u2,u3,u4,u5}
VARIABLES Wifi, Bluetooth, Airplane, EmailLoggedIn, to_deliver
INVARIANT
 Wifi:BOOL &
 Bluetooth:BOOL &
 Airplane:BOOL & (Airplane=TRUE => (Wifi=FALSE & Bluetooth=FALSE)) &
 EmailLoggedIn <: User &
 (EmailLoggedIn /= {} => Wifi=TRUE) &
 to_deliver: User <-> User
INITIALISATION
  Wifi,Bluetooth,Airplane := FALSE,FALSE,FALSE || EmailLoggedIn := {} || to_deliver := {}
OPERATIONS
  TurnWifiOn = PRE Wifi=FALSE THEN Wifi:=TRUE || Airplane :=FALSE END;
  TunrAirplaneModeOn = PRE Airplane=FALSE THEN
     Airplane := TRUE || Wifi,Bluetooth := FALSE,FALSE
     || EmailLoggedIn := {}
  END;
  LogIntoEmail(u) = PRE u:User & Wifi=TRUE & u /: EmailLoggedIn THEN
     EmailLoggedIn := EmailLoggedIn \/ {u}
  END;
  LogOutOfEmail(u) = PRE u:EmailLoggedIn THEN
     EmailLoggedIn := EmailLoggedIn \ {u}
  END;
  SendEmail(u,to) = PRE u:EmailLoggedIn & to:User THEN to_deliver := to_deliver \/ {u|->to} END;
  ReceiveEmail(u,from) = PRE u:EmailLoggedIn & from|->u:to_deliver & from/=u THEN
     to_deliver := to_deliver \ {from|->u} END;
  SynContacts(u) = PRE u:EmailLoggedIn THEN skip END
END
```

Recent ProB developments for explicit state model checking

- parB: uses ZeroMQ for parallel model checking



- TLC Backend: uses TLC model checker as alternate backend; very fast for low-level B models, can be parallelised (AWS), can deal with very large state spaces (on disk)
- LTSMin Bridge: links ProB with LTSMin model checking engine:
  - BDD-style symbolic model checking
  - Partial Order Reduction



Conclusion Part 1

- Constraint programming for B enables many important validation and verification activities
  - enables animation (and thus model checking) of high level specifications
  - many validation tasks can be encoded as finding the solution of a B predicate: bounded model checking, test-case generation
  - complements proof

Part 2:
Overview of Constraint Solvers for B

Challenges of Constraint Programming for B

- **undecidable** formal method, with nested **quantifiers** and negation: $\forall n.(n > 2 \Rightarrow \neg \exists a, b, c\,.\, a^n + b^n = c^n)$
- **higher-order** functions and relations, **unbounded** variables
- we want **reliable** solving, e.g., satisfy CENELEC EN50128 T2 or T3 tool classification
- we want to find **all** solutions to predicates
- we want to be able to deal with real data: **large** relations, large integers

Default ProB Solver

- written in SICStus Prolog with CLP(FD) finite domain library
    - ?- X in 1..3, Y #= X+X. ⤳ Y in 2..6
- ProB uses CLP(FD) for integers and deferred set / enumerated set elements
    - >>> x:1..3 & y=x+x ⤳ x = ?:1..3 & y = ?:2..6
- Booleans are encoded separately using pred_false,pred_true values
- Set have three representations: Prolog lists, AVL-trees, **symbolic** closures and custom solver using Prolog co-routines which trigger upon set-instantiations
- Main Idea: **determinism** first, efficient set representation first

Demo with -repl

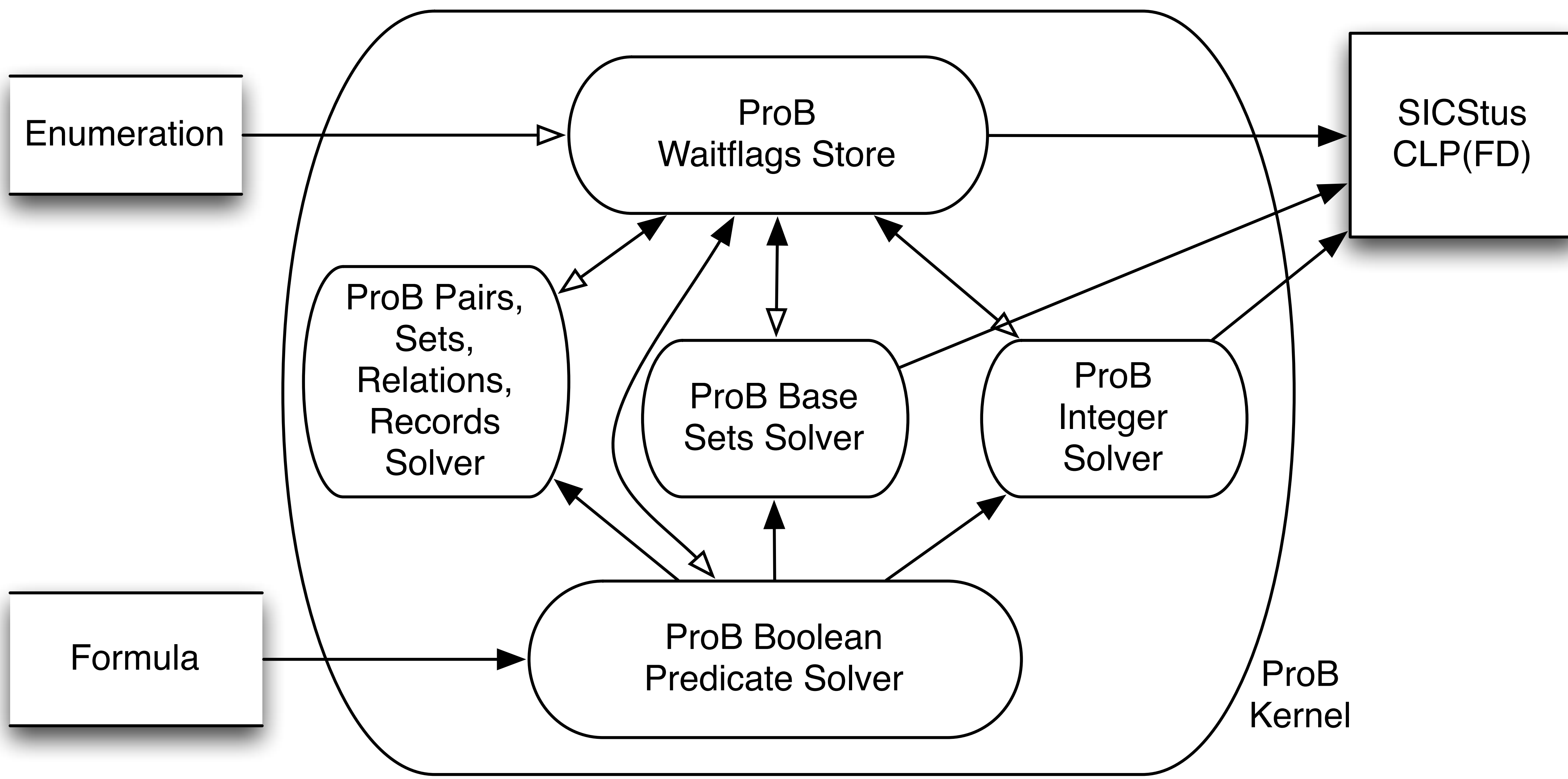
- ! deterministic mode

>>> x:1..3 & y=x+x
$\leadsto$ x = ?:1..3 & y = ?:2..6

>>> r = {1|->11, 2|->22, 3|->33, 4|->44} & x|->y:r & y>22
$\leadsto$ x = ?:3..4 & y = ?:{33}\/{44}

>>> cube = {x,y|y=x*x*x} & cube(v)=512

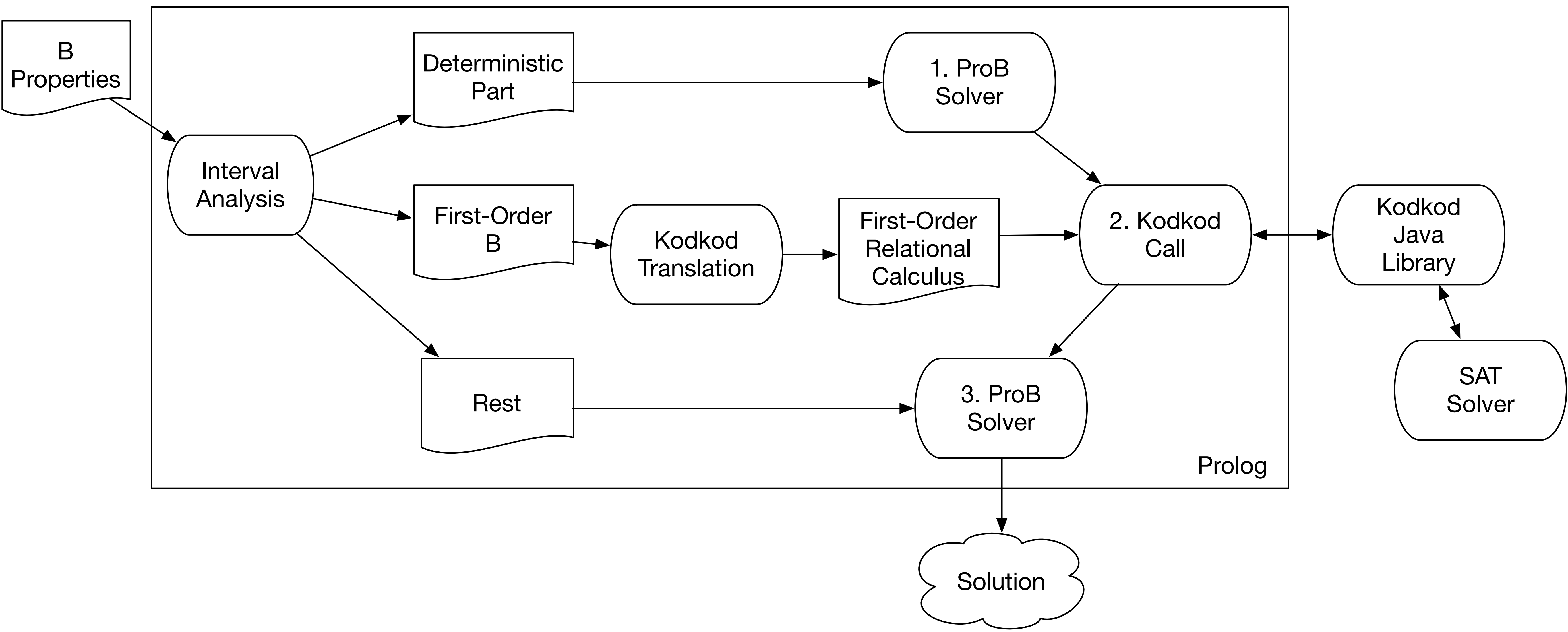
ProB Architecture



Kodkod Library

- Java library to translate first-order relational calculus expressions into propositional logic for SAT solving
  - can make use of various SAT solvers
  - core of the Alloy tool

Overview of ProB Kodkod Backend



Demo with -repl

```
>>> :kodkod r: 1..5 <-> 1..5 & (r;r) = r & r /= {} & dom(r)=1..5 & r[2..3]=4..5

kodkod ok:   r : 1 .. 5 <-> 1 .. 5 & (r ; r) = r & r /= {} & do...  ints: irange(0,5), intatoms: irange(0,5)

Kodkod module started up successfully (SAT solver SAT4J with timeout of 1500 ms).

Times for computing solutions: [4,3,4,2,1,5,1,2,2,1,2,1,1,1,1,1,2,1,1,1,1,1]

Existentially Quantified Predicate over r is TRUE

Solution:

       r = {(1|->4),(2|->4),(3|->4),(3|->5),(4|->4),(5|->4),(5|->5)}

>>> $

% Type: pred

% Eval Time: 10 ms (400 ms walltime)
```

ProB Z3 Backend

- Prior work: Deharbe et al.: SMT solvers for proof in Rodin
- Here use Z3’s native set support
- Translation more tuned for solving than for proof
- Two possible integrations:
  - 1) translate entire predicate to Z3
  - 2) interleave calls to Z3 in ProB interpreter

Unsatisfiability

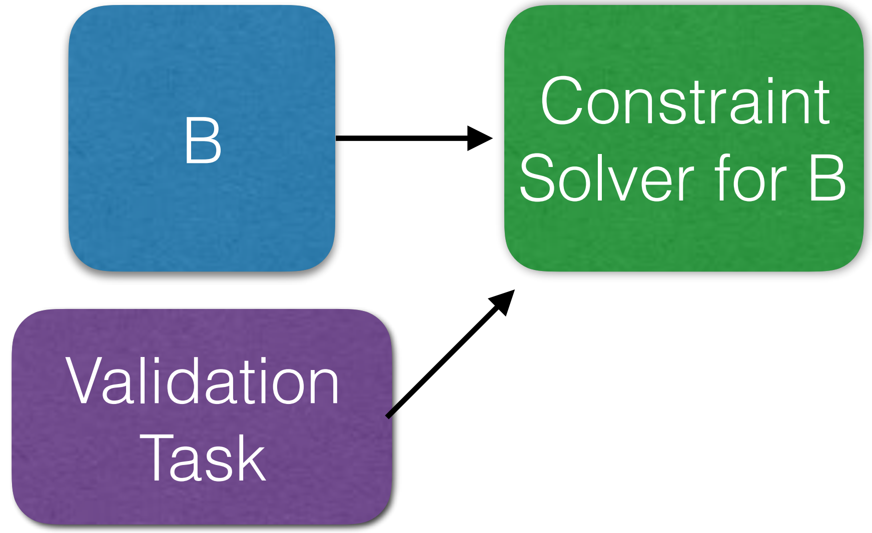
- $x<y$ & $y<x$
  - Time-out with CLP(FD)
  - Kodkod backend not applicable (no finite bound)
  - Z3: quickly finds inconsistency
- Z3 often very good at detecting inconsistencies; useful in ProB disprover or in interleaved interpreter to prune branches

Satsifiable (Model Finding)

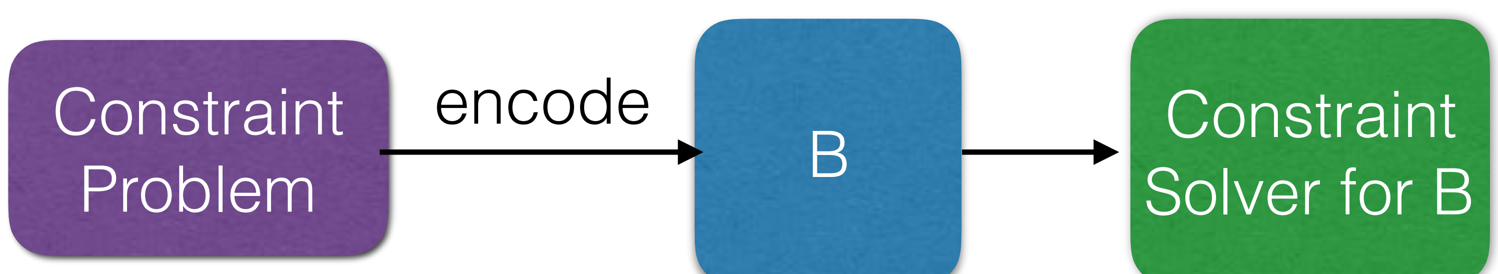
- KISS*KISS=PASSION puzzle
- {K,P} <: 1..9 & {I,S,A,O,N} <: 0..9 & (1000*K+100*I+10*S+S) * (1000*K+100*I+10*S+S) = 1000000*P+100000*A+10000*S+1000*S+100*I+10*O +N & card({K, I, S, P, A, O, N}) = 7
  - 0 ms with ProB
  - 800 ms with Kodkod backend
  - 4570 ms with Z3 backend

Anecdotal Evidence

- ProB standard **Prolog** solver: good for integers, disequalities, finding models, can deal with symbolic set representations
  - not good for unbounded inconsistent predicates, complicated relational constraints with image, closure,…
- **Kodkod/SAT**: good for enumerated sets, relational operators (composition, closure, image)
  - not good for large integers, large sets; cannot deal with sets of sets, unbounded values
- **Z3/SMT**: good for detecting inconsistencies, particular unbounded domains
  - unreliable for finding models, finite problems often get translated into unbounded SMTLib problems



Part 3: Constraint Programming in B



Why do CP in B ?

- Teaching B, Mathematics, Theoretical Computer Science
- More expressive language (will be shown on examples)
- Proof Support (from B)
- Stress test our solver for B
- Experiment with an alternate approach of using B:
  - formal models as artefacts at runtime
  - no code generation, or much more limited code generation

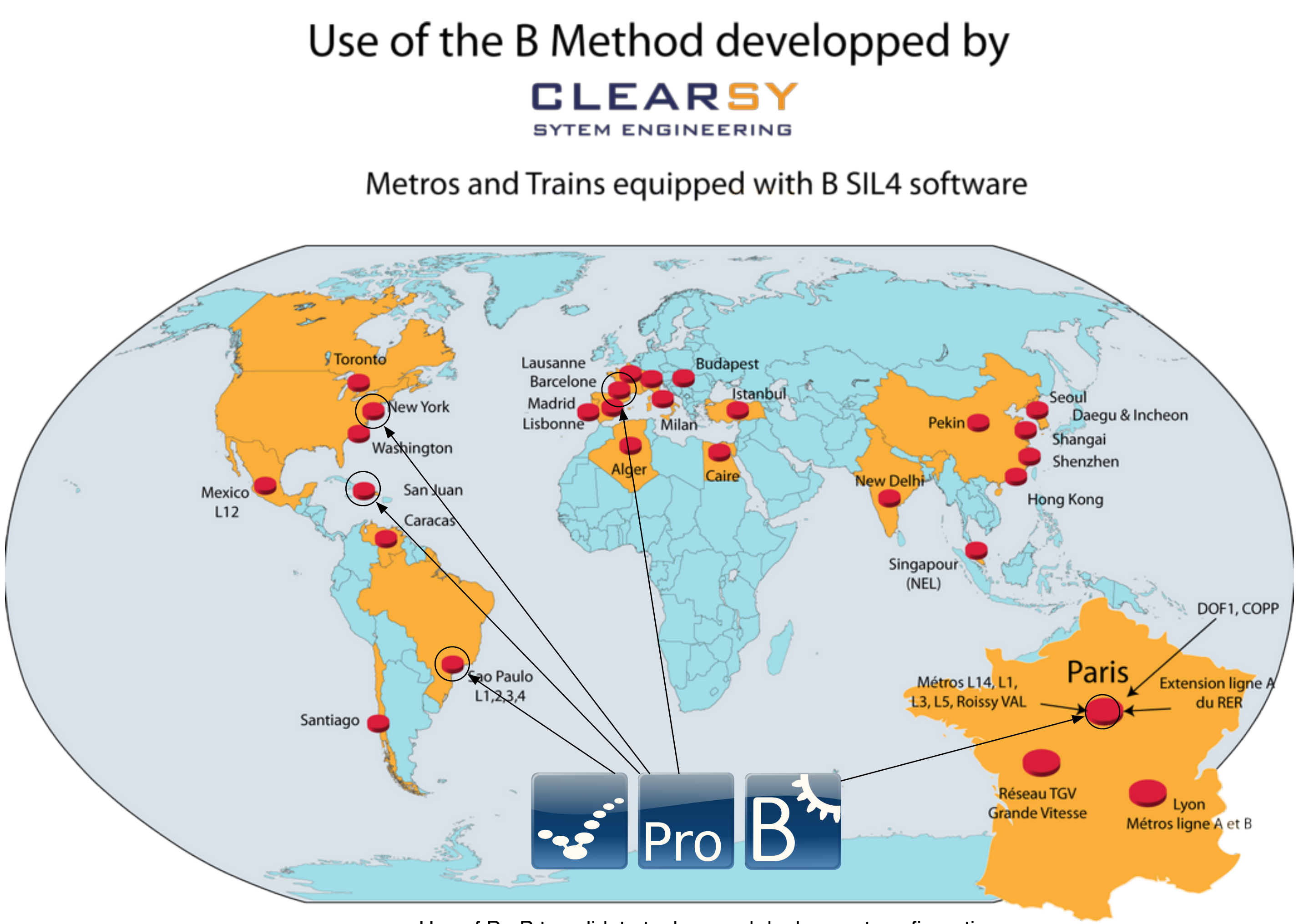
Latex Slides generated with B and ProB

Part 4: Applications of Constraint Programming within B

Practical Use ?

- Solving Puzzles with B is nice for teaching
- Produces test cases for ProB
- But does it have any practical significance or application ?

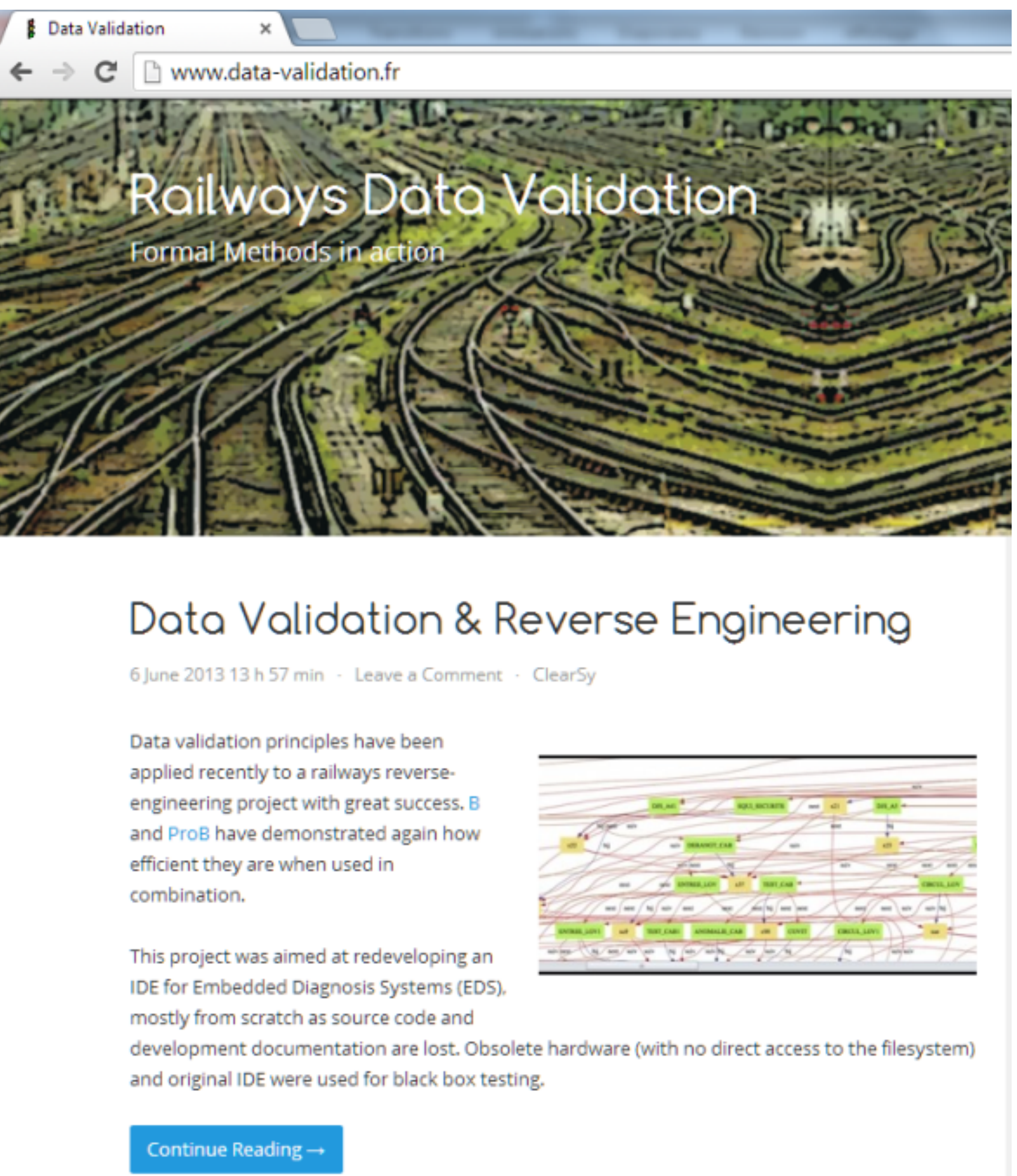
Railway Applications

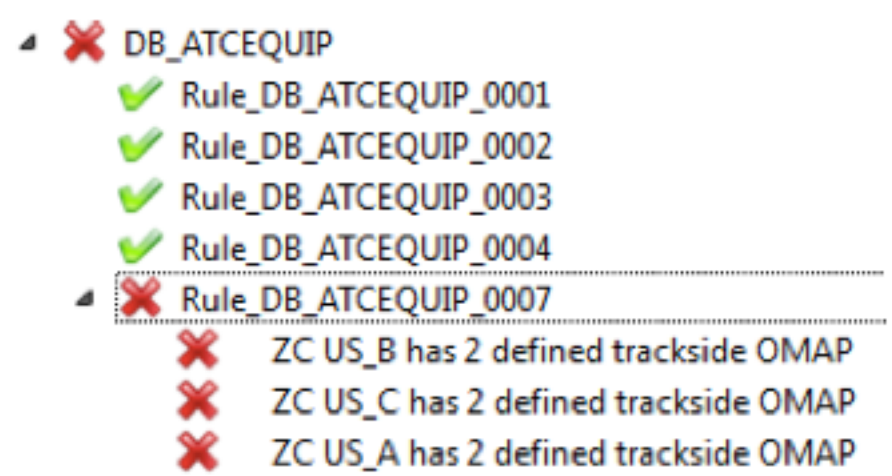


Use of ProB to validate toplogy and deployment configuration

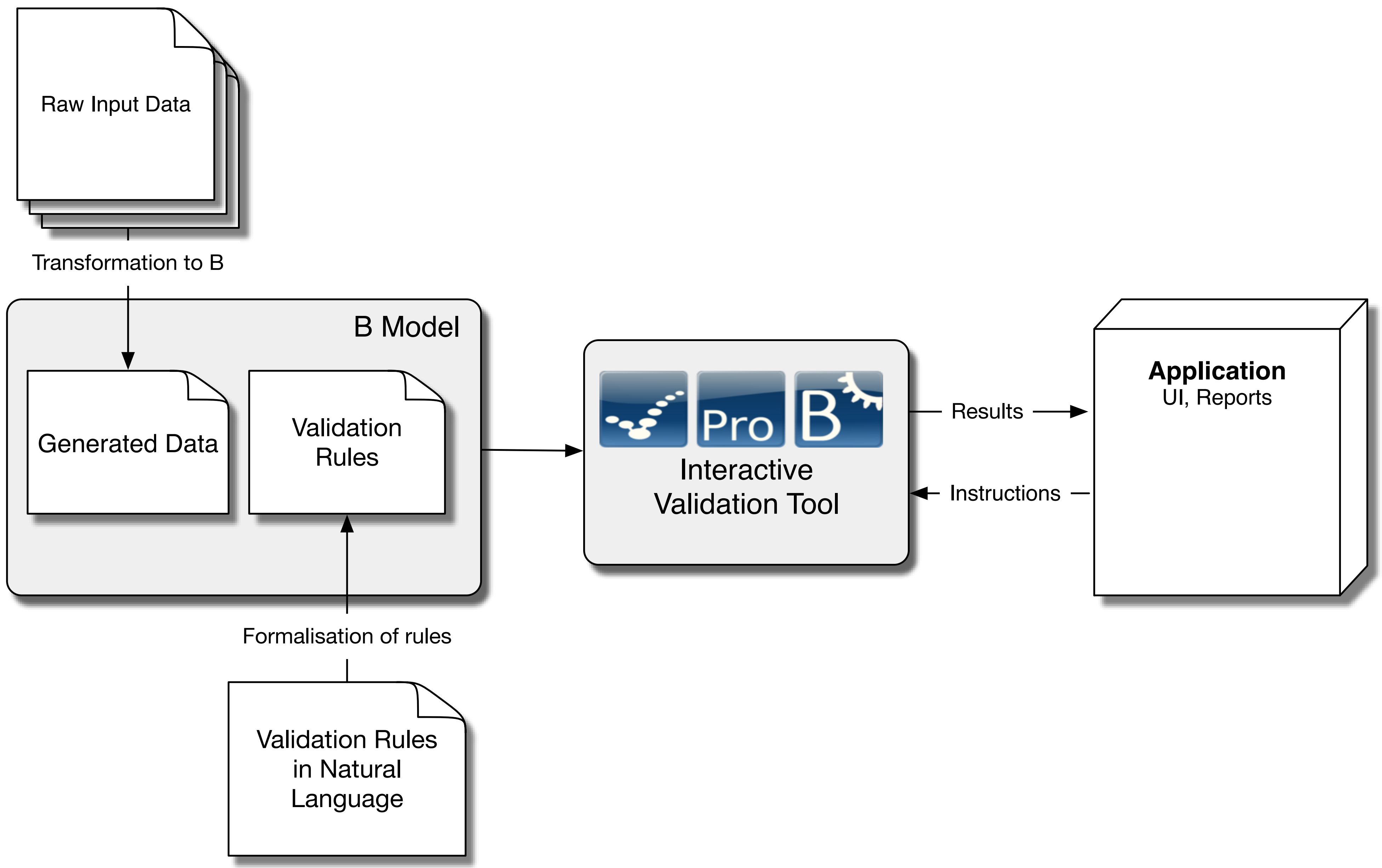
Data Validation

http://www.data-validation.fr/





Data Validation for Thales RBC



Example B Rule

"Two signals valid for the same direction
should not be placed at the same location"

$$
\begin{aligned}
&\forall signal1, signal2 \cdot (\ signal1 \in Signals \land signal2 \in Signals \land signal1 \neq signal2 \\
&\quad \land Signal\_Direction(signal1) = Signal\_Direction(signal2) \\
&\qquad \Rightarrow \neg(Signal\_TrackSegment(signal1) = Signal\_TrackSegment(signal2) \\
&\qquad\qquad \land Signal\_OperationalKM(signal1) = Signal\_OperationalKM(signal2)))
\end{aligned}
$$

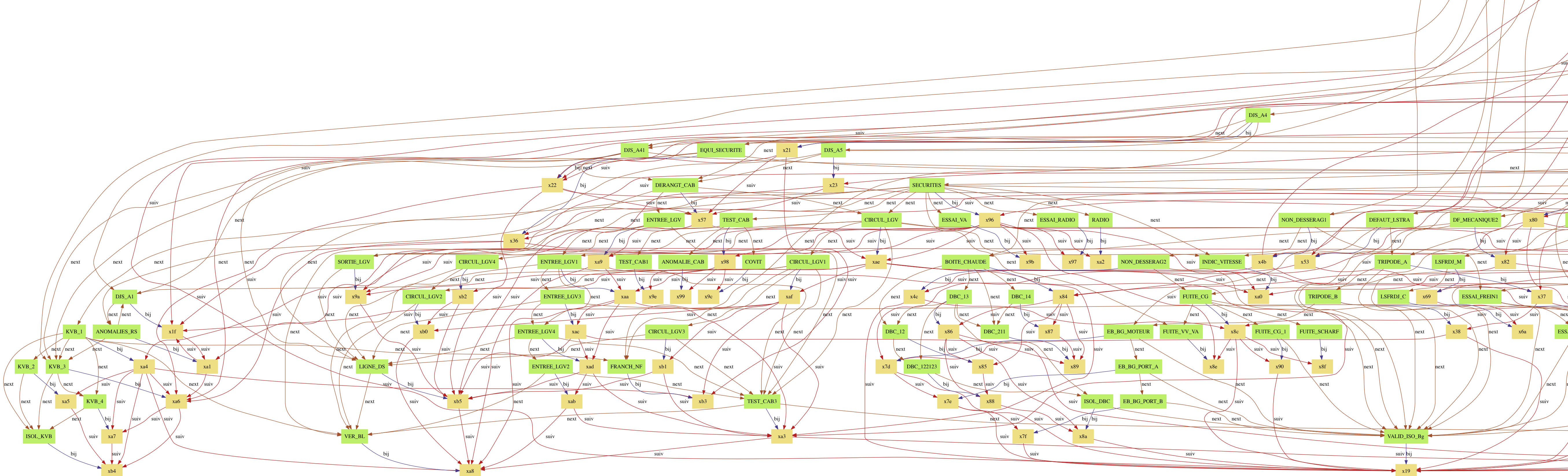
Benefits

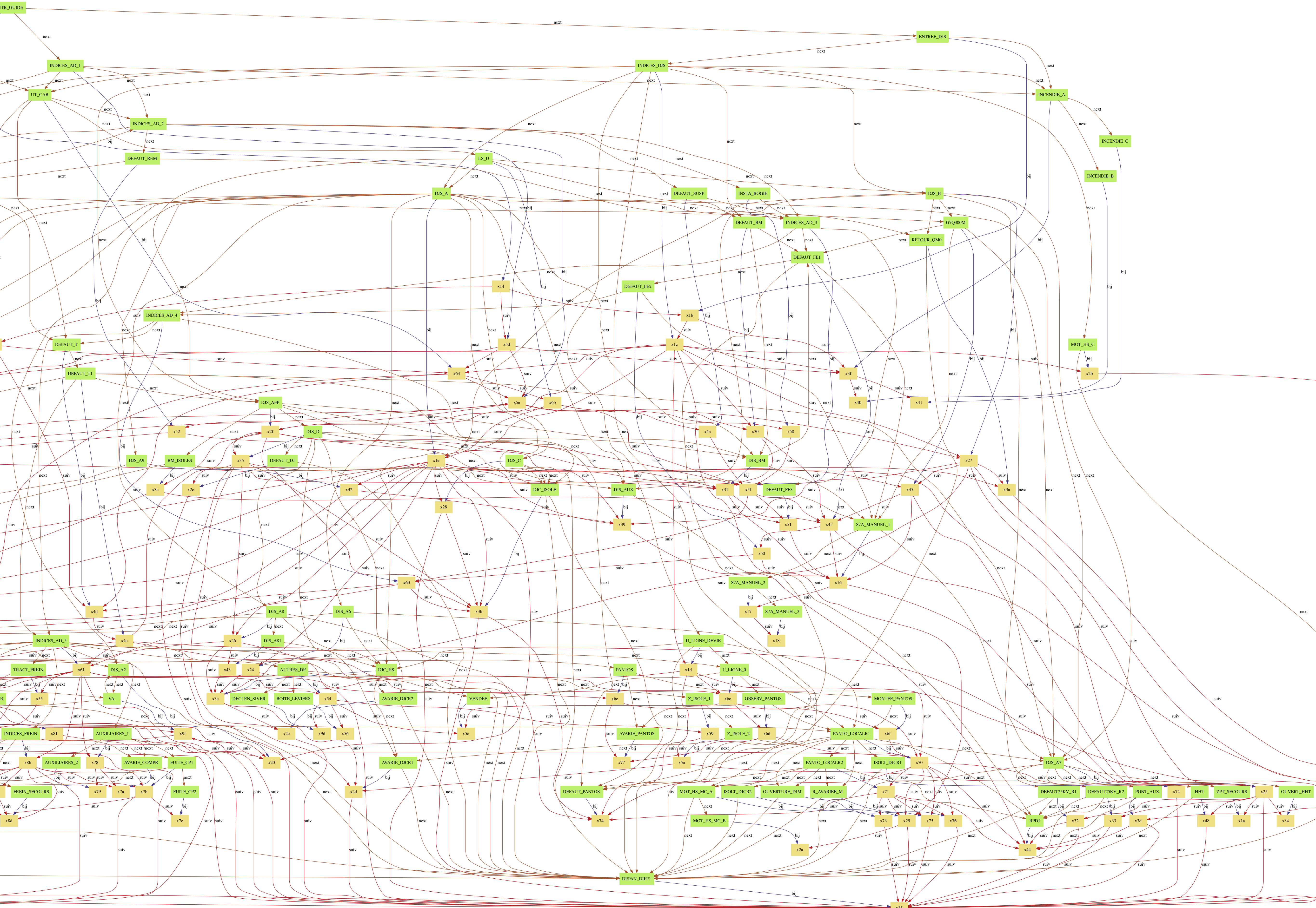
- declarative notation of the rules is well suited for reviews and eases the certification process
- correctness of complex computations can be ensured by formal assertions
- elimination of communication problems and quick turn-around
- product line management reflected in formal model

Reverse Engineering of Application Binary with ProB

bij: G7 >->> ADR &
!xx.(xx: G7 => bij[next[{xx}]] = suiv[bij[{xx}]])

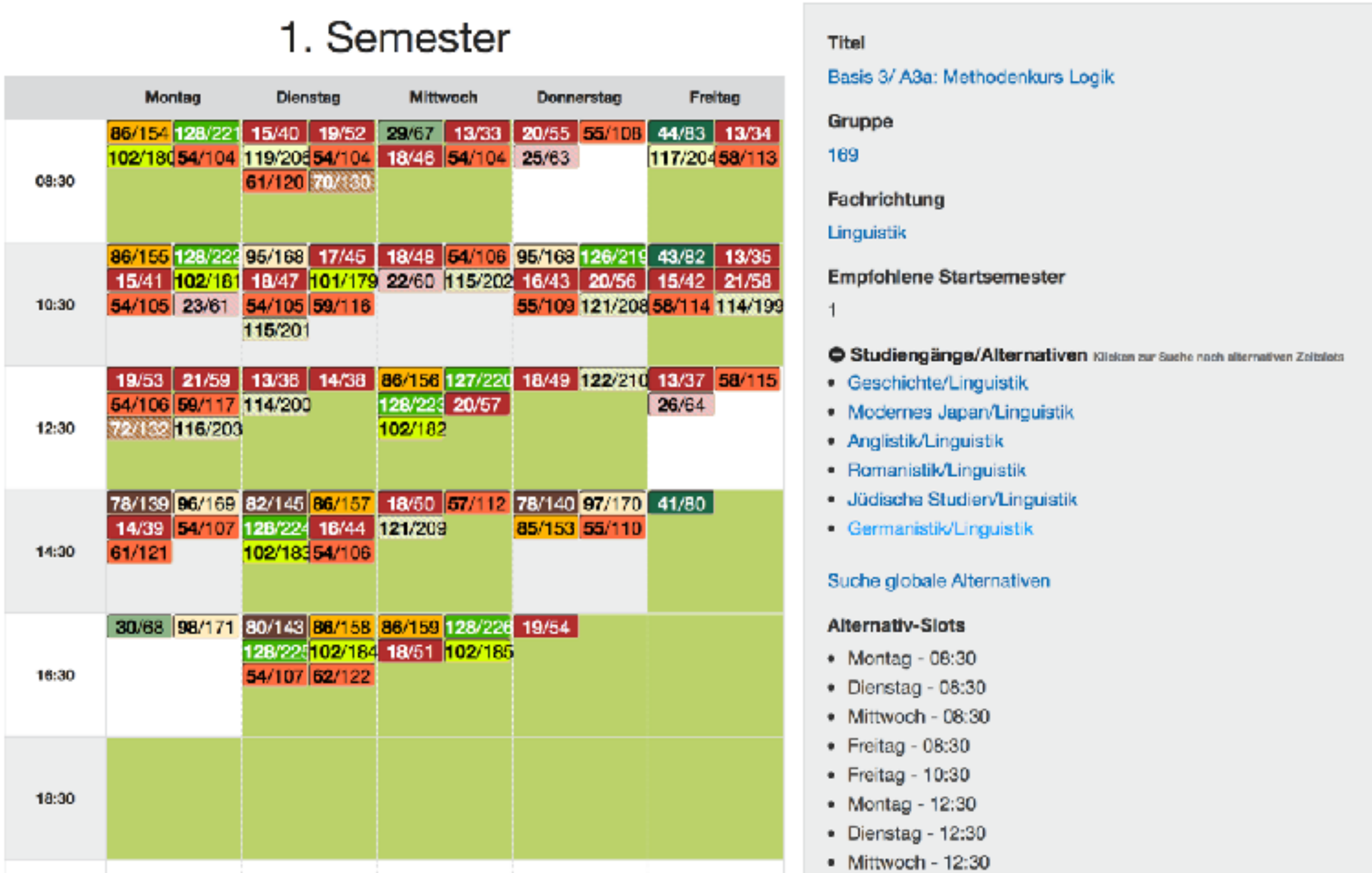
162! =
1229694218739449434110178928491750176572300599427169306620762521167814540117728965860988098467051531783599
5074429904709708273401807824365415928975695099566042246320538220924308010459938381430588227927174194100982
18920470961529319832639077341092590387200000000000000000000000000000000000000



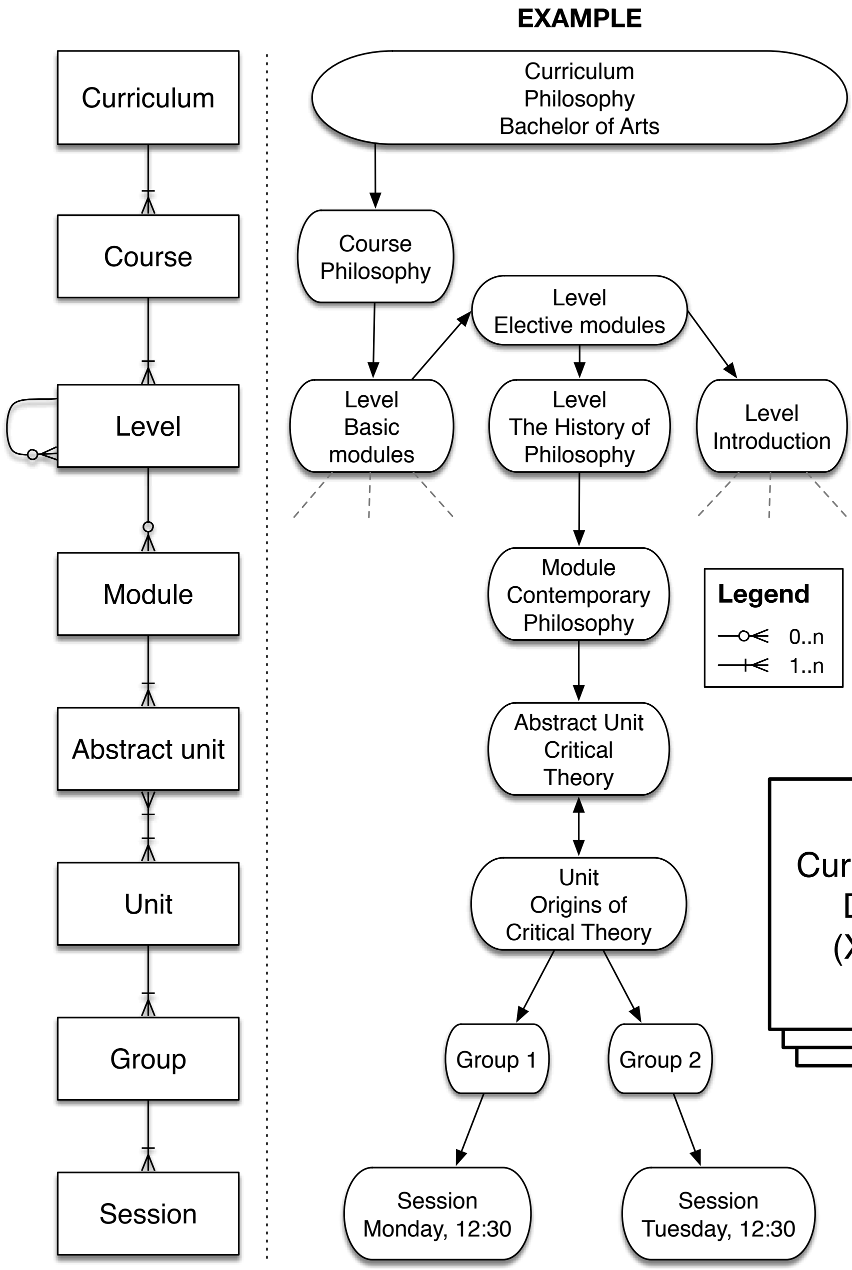


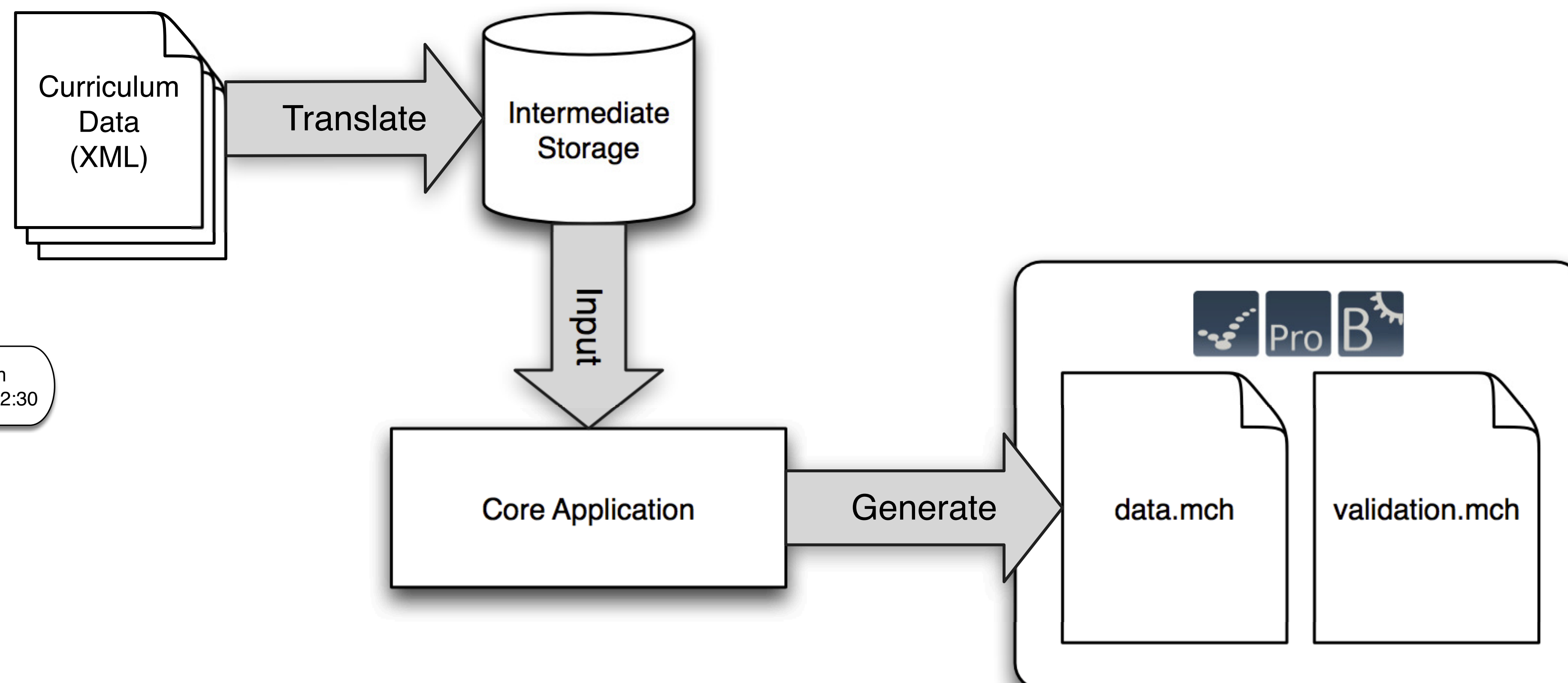
M. Leuschel : ProB Demo

ProB Constraint Solver in Action



Detect and Fix University Timetabling Conflicts [FM’15]


EXAMPLE
Curriculum
Course
Level
Module
Abstract unit
Unit
Group
Session
Curriculum
Philosophy
Bachelor of Arts
Course
Philosophy
Level
Elective modules
Level
Basic
modules
Level
The History of
Philosophy
Level
Introduction
Module
Contemporary
Philosophy
Legend
0..n
1..n
Abstract Unit
Critical
Theory
Unit
Origins of
Critical Theory
Group 1
Group 2
Session
Monday, 12:30
Session
Tuesday, 12:30


Curriculum
Data
(XML)
Translate
Intermediate
Storage
Input
Core Application
Generate
Pro
data.mch
validation.mch

Alternate Encodings (of simpler course model)

- ProB model: milliseconds to seconds to solve courses or detect impossibility

- Alloy: not successful; not able to solve timetabling for size of courses we require

- Z3: initial models very slow (hours to find solutions); latest version faster (good for inconsistency; still relatively slow for finding models: minutes vs seconds in ProB)

Part 5: A small case study: highlight expressivity of B and showcase new solving technique (Latex Slides)

Conclusions

- B/Event-B encodings often very compact and elegant
- ProB good at arithmetic & ≠ (Alphametic, Queens)
  - success for industrial case studies (data validation for Siemens, Alstom, General Electric, RATP, …)
  - extensive validation, double chain available
- Can deal with large data and (in some cases) with higher-order datatypes and unbounded domains and infinite sets symbolically
- Kodkod backend very good for relational image, closure, composition
- Z3 backend very good for detecting inconsistent, unbounded predicates
- Combined ProB CLP(FD) and Kodkod backend shows promise

One encouraging result: SMTLib Competition 2016

NIA (Main Track)

Competition results for the NIA division as of Mon Jun 27 20:09:59 GMT

Benchmarks in this division : 9

Winners:

Sequential Performances	Parallel Performances
ProB	ProB

NIA = Non-Linear Integer Arithmetic with quantifiers

Result table[1]

Solver	Sequential performance			Parallel performance				Other information
	Error Score	Correctly Solved Score	avg. CPU time	Errors	Corrects	avg. CPU time	avg. WALL time	Unsolved benchmarks
CVC4	0.000	5.000	0.019	0.000	5.000	0.019	0.020	4
ProB	0.000	8.000	0.888	0.000	8.000	0.888	0.887	1
vampire_smt_4.1	0.000	6.000	335.043	0.000	6.000	335.043	333.271	3
vampire_smt_4.1_parallel	0.000	7.000	537.677	0.000	7.000	871.029	232.010	2
z3[n]	0.000	9.000	0.038	0.000	9.000	0.038	0.037	0

The Goal

- Dealing with integers, large data like CLP (SAT/SMT not enough to automatically detect simple high-level propagation rules)
- Learning like SAT, SMT (CLP propagation not enough in presence of complicated constraints)
- Symbolic functions with good performance (partial evaluation ?)
- **(Executable) Models as Runtime Artifacts Executable Mathematics**

Thanks !

Jens Bendisposto
Carl Friedrich Bolz
Ivo Dobrikov
Nadine Elbeshausen
Fabian Fritz
Marc Fontaine
Stefan Hallerstede
Dominik Hansen
Christoph Heinzen
Michael Jastram
Philip Körner
Sebastian Krings
Lukas Ladenberger
Li Luo
Daniel Plagge
Mireille Samia
David Schneider
Corinna Spermann
Dennis Winter

Michael Butler

Thierry Massart

Edd Turner

Theory Plugin support

Highlights

automatic detection of (potentially) infinite sets

```
context Demo
constants cube isqrt sol1 sol2
axioms
 @axm0 cube=(λx·x∈ℤ|x*x*x)
 theorem @thm0 {y|cube(cube(y))=512}={2} // observe: no domain restriction on y

 @axm1 isqrt = {x↦r | x∈ℕ ∧ r∈ℕ ∧ r*r ≤ x ∧ (r+1)*(r+1)>x} // automatically detected as symbolic in new
ProB; observe: no explicit finiteness restriction on x
 theorem @thm1 isqrt(100) = 10
 theorem @thm2 isqrt(10) = 3
 theorem @thm3 isqrt[1..20] = 1..4
 theorem @thm4 ({1↦4, 2↦9, 3↦ 26} ; isqrt) = {1↦2, 2↦3, 3↦5}                map
 theorem @thm5 {x| isqrt(x)=10} = 100..120  // constraint solvi               on on x
!

 theorem @thm6 cls({1↦2, 2↦3}) = {1↦2,2↦3,
 theorem @thm7 cls(isqrt)[{16}] = {4,2,1}                     an infinite function !
 theorem @thm8 cls(isqrt)[{1024}] = {1,2,5                   e of an infinite functio
!
 @axm8 sol1 = cls(isqrt)[{   00000}]
 theorem @thm_sol1 sol1 = {1,2,4,17,316}
 @axm9 sol2 = {x|x∈1..100 ∧ cls(isqrt)[{x}] = {1,2,4}}
 theorem @thm_sol2 sol2 = 16..24

 theorem @nested {x| x∈1..100 ∧ {y|y∈x..x+10} = {z|z≥x ∧ z<x+11 ∧ ∃v·(v>x)}} = 1..100
end
```

solving over infinite domains

transitive closure for infinite relation

nested quantifiers and set comprehensions