SETS_RODIN18

June 4, 2018

1 Solving (Set) Constraints in B and Event-B

1.1 A quick Introduction to B

1.1.1 Basic Datavalues in B

B provides the booleans, strings and integers as built-in datatypes. (Strings are not available in Event-B.)

```
In [1]: BOOL
```

```
Out[1]: {FALSE,TRUE}
```

```
In [2]: "this is a string"
```

```
Out[2]: "this is a string"
```

```
In [3]: 1024
```

```
Out[3]: 1024
```

Users can define their own datatype in a B machine. One distinguishes between explicitly specified enumerated sets and deferred sets.

```
In [24]: ::load
         MACHINE MyBasicSets
         SETS Trains = {thomas, gordon}; Points
         END
```

```
[2018-06-02 14:41:51,919, T+56859881] "Shell-0" de.prob.cli.PrologProcessProvider.makeProcess(Pr
[2018-06-02 14:41:53,113, T+56861075] "Shell-0" de.prob.cli.PortPattern.setValue(PortPattern.jav
[2018-06-02 14:41:53,114, T+56861076] "Shell-0" de.prob.cli.InterruptRefPattern.setValue(Interru
[2018-06-02 14:41:53,119, T+56861081] "ProB Output Logger for instance 4d84f477" de.prob.cli.Pro
[2018-06-02 14:41:53,132, T+56861094] "ProB Output Logger for instance 4d84f477" de.prob.cli.Pro
```

```
Out[24]: Loaded machine: MyBasicSets : []
```

```
In [86]: Trains
```

```
Out[86]: {thomas,gordon}
```

For animation and constraint solving purposes, ProB will instantiate deferred sets to some finite set (the size of which can be controlled and is partially inferred).

```
In [5]: Points
```

```
Out[5]: {Points1,Points2}
```

1.1.2 Pairs

B also has pairs of values, which can be written in two ways:

```
In [6]: (thomas,10)
```

```
Out[6]: (thomas↦10)
```

```
In [7]: thomas |-> 10
```

```
Out[7]: (thomas↦10)
```

Tuples simply correspond to nested pairs:

```
In [8]: (thomas |-> gordon |-> 20)
```

```
Out[8]: ((thomas↦gordon)↦20)
```

1.1.3 Sets

Sets in B can be specified in multiple ways. For example, using explicit enumeration:

```
In [9]: {1,3,2,3}
```

```
Out[9]: {1,2,3}
```

or via a predicate by using a set comprehension:

```
In [10]: {x|x>0 & x<4}
```

```
Out[10]: {1,2,3}
```

One can use on of the *base sets* :

```
In [88]: (BOOL,INTEGER,STRING,Trains,Points)
```

```
Out[88]: (((({FALSE,TRUE}↦INTEGER)↦STRING)↦{thomas,gordon})↦{Points1,Points2})
```

For integers there are a variety of other sets, such as intervals:

```
In [11]: 1..3
```

```
Out[11]: {1,2,3}
```

or the set of implementable integers INT = MININT..MAXINT or the set of implementable natural numbers NAT = 0..MAXINT.

Sets can be higher-order and contain other sets:

```
In [12]: { 1..3,  {1,2,3,2}, 0..1, {x|x>0 & x<4} }

Out[12]: {{0,1},{1,2,3}}
```

Relations are modelled as sets of pairs:

```
In [13]: { thomas|->gordon, gordon|->gordon, thomas|->thomas}

Out[13]: {(thomas↦thomas),(thomas↦gordon),(gordon↦gordon)}
```

Note: a pair is an element of a Cartesian product, and a relation is just a subset of a Cartesian product. The above relation is a subset of:

```
In [87]: Trains * Trains

Out[87]: {(thomas↦thomas),(thomas↦gordon),(gordon↦thomas),(gordon↦gordon)}
```

Functions are relations which map every domain element to at most one value:

```
In [14]: { thomas|->1, gordon|->2}

Out[14]: {(thomas↦1),(gordon↦2)}
```

1.2 Expressions vs Predicates vs Substitutions

1.2.1 Expressions

Expressions in B have a value. With ProB and with ProB's Jupyter backend, you can evaluate expresssions such as:

```
In [15]: 2**1000

Out[15]: 1071508607186267320948425049060001810561404811705533607443750388370351051124936122493198
```

B provides many operators which return values, such as the usual arithmetic operators but also many operators for sets, relations and functions.

```
In [12]: (1..3 \/ 5..10) \ (2..6)

Out[12]: {1,7,8,9,10}

In [11]: (1..3 ∪ 5..10) \ (2..6)

Out[11]: {1,7,8,9,10}

In [17]: ran({(thomas↦1),(gordon↦2)})

Out[17]: {1,2}

In [18]: {(thomas↦1),(gordon↦2)} (thomas)

Out[18]: 1

In [19]: {(thomas↦1),(gordon↦2)}~[2..3]

Out[19]: {gordon}
```

1.3 Predicates

ProB can also be used to evaluate predicates (B distinguishes between expressions which have a value and predicates which are either true or false).

```
In [20]: 2>3
```

```
Out[20]: FALSE
```

```
In [21]: 3>2
```

```
Out[21]: TRUE
```

Within predicates you can use **open** variables, which are implicitly existentially quantified. ProB will display the solution for the open variables, if possible.

```
In [22]: x*x=100
```

```
Out[22]: TRUE

         Solution:
                 x = −10
```

We can find all solutions to a predicate by using the set comprehension notation. Note that by this we turn a predicate into an expression.

```
In [23]: {x|x*x=100}
```

```
Out[23]: {−10,10}
```

1.3.1 Substitutions

B also has a rich syntax for substitutions, aka statements. For example `x := x+1` increments the value of x by 1. We will not talk about substitutions in the rest of this presentation. The major differences between classical B and Event-B lie in the area of substitutions, machine composition and refinement.

1.4 Definition of Constraint Solving

Constraint solving is determine whether a predicate with open/existentially quantified variables is satisfiable and providing values for the open variables in case it is. We have already solved the predicate `x*x=100` above, yielding the solution `x=-10`. The following is an unsatisfiable predicate:

```
In [24]: x*x=1000
```

```
Out[24]: FALSE
```

The difference to **proof** is that in constraint solving one has to produce a solution (aka a model). The difference to **execution** is that not all variables are known.

1.5 Constraint Solving Applications

Constraint solving has many applications in formal methods in general and B in particular.

Animation It is required to animate implicit specifications. Take for example an event

```
train_catches_up = any t1,t2,x where t1:dom(train_position) & t2:dom(train_position) &
                                     train_position(t1) < train_position(t2) &
                                     x:1..(train_position(t2)-train_position(t1)-1) then
                        train_position(t1) := train_position(t1)+x end
```

To determine whether the event is enabled, and to obtain values for the parameters of the event in a given state of the model, we have to solve the following constraint:

```
In [25]: train_position = {thomas|->100, gordon|->2020} &
         t1:dom(train_position) & t2:dom(train_position) & train_position(t1) < train_position(t
         x:1..(train_position(t2)-train_position(t1)-1)
```

```
Out[25]: TRUE

         Solution:
                 x = 1
                 train_position = {(thomas↦100),(gordon↦2020)}
                 t1 = thomas
                 t2 = gordon
```

```
In [26]: train_position = {thomas|->2019, gordon|->2020} &
         t1:dom(train_position) & t2:dom(train_position) & train_position(t1) < train_position(t
         x:1..(train_position(t2)-train_position(t1)-1)
```

```
Out[26]: FALSE
```

Feasibility Analysis Suppose that we have the invariant, `train_position:TRAINS-->1..10000` we can check whether the event is **feasible** in at least one valid state by solving:

```
In [27]: train_position:Trains-->1..10000 &
         t1:dom(train_position) & t2:dom(train_position) & train_position(t1) < train_position(t
         x:1..(train_position(t2)-train_position(t1)-1)
```

```
Out[27]: TRUE

         Solution:
                 x = 1
                 train_position = {(thomas↦1),(gordon↦3)}
                 t1 = thomas
                 t2 = gordon
```

Many other applications exist: generating **testcases**, finding counter examples using **bounded model checking** or other algorithms like IC3. Other applications are analysing **proof obligations**. Take the proof obligation for an event theorem t1 $/=$ t2:

```
train_position:Trains-->1..10000 & train_position(t1) < train_position(t2) |- t1 /= t2
```

We can find counter examples to it by negating the proof goal:

```
In [28]: train_position:Trains-->1..10000 & train_position(t1) < train_position(t2) & not( t1 /=
```

```
Out[28]: FALSE
```

Modelling and Solving Problems in B Obviously, we can also use constraint solving to solve puzzles or real-life problems. #### Send More Money Puzzle #### We now try and solve the SEND+MORE=MONEY arithmetic puzzle in B, involving 8 distinct digits:

```
In [7]: :prettyprint {S,E,N,D, M,O,R, Y} <: 0..9 & S >0 & M >0 & card({S,E,N,D, M,O,R, Y}) = 8 &
          S*1000 + E*100 + N*10 + D + M*1000 + O*100 + R*10 + E = M*10000 + O*1000 + N*100 + E*
```

Out[7]:

$\{S, E, N, D, M, O, R, Y\} \subseteq 0..9 \wedge S > 0 \wedge M > 0 \wedge card(\{S, E, N, D, M, O, R, Y\}) = 8 \wedge S * 1000 + E * 100 + N * 10 + D + M * 1000 + O * 100 + R * 10 + E = M * 10000 + O * 1000 + N * 100 + E * 10 + Y$

```
In [29]: {S,E,N,D, M,O,R, Y} <: 0..9 & S >0 & M >0 &
            card({S,E,N,D, M,O,R, Y}) = 8 &
            S*1000 + E*100 + N*10 + D +
            M*1000 + O*100 + R*10 + E =
          M*10000 + O*1000 + N*100 + E*10 + Y
```

```
Out[29]: TRUE

         Solution:
                R = 8
                S = 9
                D = 7
                E = 5
                Y = 2
                M = 1
                N = 6
                O = 0
```

We can find all solutions (to the unmodified puzzle) using a set comprehension and make sure that there is just a single soltuion:

```
In [30]:   {S,E,N,D, M,O,R, Y |
            {S,E,N,D, M,O,R, Y} <: 0..9 &  S >0 & M >0 &
            card({S,E,N,D, M,O,R, Y}) = 8 &
            S*1000 + E*100 + N*10 + D +
            M*1000 + O*100 + R*10 + E =
            M*10000 + O*1000 + N*100 + E*10 + Y }
```

```
Out[30]: {(((((((9↦5)↦6)↦7)↦1)↦0)↦8)↦2)}
```

KISS PASSION Puzzle A slightly more complicated puzzle (involving multiplication) is the KISS * KISS = PASSION problem.

```
In [31]:    {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
```

```
Out[31]: TRUE

         Solution:
                 P = 4
                 A = 1
                 S = 3
                 I = 0
                 K = 2
                 N = 9
                 O = 8
```

Finally, a simple puzzle involving sets is to find a subset of numbers from 1..5 whose sum is 14:

```
In [32]: x <: 1..5 & SIGMA(y).(y:x|y)=14
```

```
Out[32]: TRUE

         Solution:
                 x = {2,3,4,5}
```

1.6 How to solve (set) constraints in B

We will now examine how one can perform constraint solving for B.

1.6.1 Booleans

If we have only booleans, constraint solving is equivalent to SAT solving. Internally, ProB has an interpreter which does **not** translate to CNF (conjunctive normal form), but is otherwise similar to DPLL: deterministic propagations are carried out first (unit propagation) and there are heuristics to choose the next boolean variable to enumerate. (We do not translate to CNF also because of well-definedness issues.)

Knights and Knave Puzzle Here is a puzzle from Smullyan involving an island with only knights and knaves. We know that: - Knights: always tell the truth - Knaves: always lie

We are given the following information about three persons A,B,C on the island: 1. A says: "B is a knave or C is a knave" 2. B says "A is a knight"

What are A, B and C? Note: we model A,B,C as boolean variables which are equal to TRUE if they are a knight and FALSE if they are a knave.

```
In [33]:  (A=TRUE <=> (B=FALSE or C=FALSE)) & // Sentence 1
           (B=TRUE <=> A=TRUE) // Sentence 2
```

```
Out[33]: TRUE

         Solution:
                 A = TRUE
                 B = TRUE
                 C = FALSE
```

1.6.2 Integers

Let us take the integer constraint `x*x=100` which we saw earlier. This constraint is actually more complicated than might appear at first sight: the constraint is not linear and the domain of x is not bounded. Indeed, B supports mathematical integers without any bit size restriction. So, let us first look at some simpler constraints, where the domains of the variables are all bounded. Let us consider a small arithmetic puzzle

```
  X Y
+ X Y
-----
  Y 0
```

```
In [34]: X:1..9 & Y:0..9 & X*10 + Y + X*10 + Y = Y*10
```

```
Out[34]: TRUE

         Solution:
                 X = 2
                 Y = 5
```

Given that we know the domain of X and Y one can represent the integers by binary numbers and convert the constraint to a SAT problem.

Let us look at an even simpler constraint X:0..3 & Y:0..3 & X+Y=2. As you can see there are three solutions for this constraint:

```
In [35]: {X,Y|X:0..3 & Y:0..3 & X+Y=2}
```

```
Out[35]: {(0↦2),(1↦1),(2↦0)}
```

We will now study how such constraints can be solved.

Solving constraints by translating to SAT Given that we know the domain of X and Y in the constraint `X:0..3 & Y:0..3 & X+Y=2` one can represent the integers by binary numbers and convert the constraint to a SAT problem. The number 2 is 10 in binary and we can represent X and Y each by two bits X0,X1 and Y0,Y1. We can translate the addition to a propositional logic formula:

Bit1	Bit0
X1	X0

Bit1	Bit0
+ Y1	Y0
Z1	Z0

Let us find one solution to this constraint, by encoding addition using an additional carry bit:

```
In [36]:  ((X0=TRUE <=> Y0=TRUE) <=> Z0=FALSE) &
          ((X0=TRUE & Y0=TRUE) <=> CARRY0=TRUE) &
          (CARRY0=FALSE => ((X1=TRUE <=> Y1=TRUE) <=> Z1=FALSE)) &
          (CARRY0=TRUE => ((X1=TRUE <=> Y1=TRUE) <=> Z1=TRUE)) &
          Z0=FALSE & Z1=TRUE
```

```
Out[36]: TRUE

         Solution:
                 Z0 = FALSE
                 Y0 = TRUE
                 Z1 = TRUE
                 X0 = TRUE
                 Y1 = TRUE
                 X1 = TRUE
                 CARRY0 = TRUE
```

```
In [37]: {X0,X1,Y0,Y1,Z0,Z1,CARRY0 | ((X0=TRUE <=> Y0=TRUE) <=> Z0=FALSE) &
            ((X0=TRUE & Y0=TRUE) <=> CARRY0=TRUE) &
            (CARRY0=FALSE => ((X1=TRUE <=> Y1=TRUE) <=> Z1=FALSE)) &
            (CARRY0=TRUE => ((X1=TRUE <=> Y1=TRUE) <=> Z1=TRUE)) &
             Z0=FALSE & Z1=TRUE}
```

```
Out[37]: {((((((FALSE↦FALSE)↦FALSE)↦TRUE)↦FALSE)↦TRUE)↦FALSE),((((((FALSE↦TRUE)↦FALSE)↦
```

As you can see, we have found four solutions and not three! One solution is 3+3=2. This is a typical issue when translating arithmetic to binary numbers: we have to prevent overflows, which we do below:

```
In [38]: {X0,X1,Y0,Y1,Z0,Z1,CARRY0 | ((X0=TRUE <=> Y0=TRUE) <=> Z0=FALSE) &
            ((X0=TRUE & Y0=TRUE) <=> CARRY0=TRUE) &
            (CARRY0=FALSE => ((X1=TRUE <=> Y1=TRUE) <=> Z1=FALSE)) &
            (CARRY0=TRUE => ((X1=TRUE <=> Y1=TRUE) <=> Z1=TRUE)) &
            (CARRY0=TRUE => (X1=FALSE & Y1=FALSE)) & // no overflow
            (CARRY0=FALSE => (X1=FALSE or Y1=FALSE)) & // no overflow
             Z0=FALSE & Z1=TRUE}
```

```
Out[38]: {((((((FALSE↦FALSE)↦FALSE)↦TRUE)↦FALSE)↦TRUE)↦FALSE),((((((FALSE↦TRUE)↦FALSE)↦
```

In ProB, we can use **Kodkod** backend to achieve such a translation to SAT. - Kodkod (https://github.com/emina/kodkod) is the API to the **Alloy** (http://alloytools.org) constraint analyzer and takes relational logic predicates and translates them to SAT. - The SAT problem can be solved by any SAT solver (Sat4J, minisat, glucose,...). - ProB translates parts of B to the Kodkod API and translates the results back to B values. - Prior to the translation, ProB performs an interval analysis to determine possible ranges for the integer decision variables.

The details were presented at FM'2012 (Plagge, L.).

```
In [39]: :solve kodkod x:0..2 & y:0..2 & x+y=2
```

```
[2018-05-30 15:21:22,569, T+18381] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:22,570, T+18382] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:22,570, T+18382] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
```

```
Out[39]: TRUE

         Solution:
                 x = 2
                 y = 0
```

We can find all solutions and check that we find exactly the three expected solutions:

```
In [40]: :solve kodkod {x,y|x:0..2 & y:0..2 & x+y=2}=res
```

```
[2018-05-30 15:21:22,674, T+18486] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:22,675, T+18487] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
```

```
Out[40]: TRUE

         Solution:
                 res = {(0↦2),(1↦1),(2↦0)}
```

Translation to SMTLib At iFM'2016 we (Krings, L.) presented a translation to SMTLib format to use either the Z3 or CVC4 SMT solver. Compared to the Kodkod backend, no translation to SAT is performed, SMTLib supports integer predicates and operators in the language. Here is ProB's SMTLib/Z3 API calls for the constraint: - mk_var(integer,x) $\rightarrow$ 2 - mk_var(integer,y) $\rightarrow$ 3 - mk_int_const(0) $\rightarrow$ 4 - mk_op(greater_equal,2,4) $\rightarrow$ 5 - mk_int_const(2) $\rightarrow$ 6 - mk_op(greater_equal,6,2) $\rightarrow$ 7 - mk_int_const(0) $\rightarrow$ 8 - mk_op(greater_equal,3,8) $\rightarrow$ 9 - mk_int_const(2) $\rightarrow$ 10 - mk_op(greater_equal,10,3) $\rightarrow$ 11 - mk_op(add,2,3) $\rightarrow$ 12 - mk_int_const(2) $\rightarrow$ 13 - mk_op(equal,12,13) $\rightarrow$ 14 - mk_op_arglist(conjunct,[5,7,9,11,14]) $\rightarrow$ 15

```
In [41]: :solve z3 x:0..2 & y:0..2 & x+y=2
```

```
Out[41]: TRUE

         Solution:
                 x = 0
                 y = 2
```

ProB's CLP(FD) Solver ProB's default solver makes use of constraint logic programming. For arithmetic, it builts on top of CLP(FD), the finite domain library of SICStus Prolog. In CLP(FD): - every integer variable is associated with a domain of possible values, typically an interval - when adding a new constraints, the domains of the involved variables are updated, or more precisely narrowed down. - at some point we need to chose variables for enumeration; typically ProB chooses the value with the smallest domain.

Let us use a slightly adapted constraint `x:0..9 & y:0..9 & x+y=2` to illustrate how constraint processing works:

- x:0..9 $\rightsquigarrow$ x:0..9, y:$-\infty$..∞
- y:0..9 $\rightsquigarrow$ x:0..9, y:0..9
- x+y=2 $\rightsquigarrow$ x:0..2, y:0..2
- Enumerate (label) variable x
- x=0 $\rightsquigarrow$ x:0..0, y:2..2
- x=1 $\rightsquigarrow$ x:1..1, y:1..1
- x=2 $\rightsquigarrow$ x:2..2, y:0..0

```
In [42]: :solve prob {K,P} <: 1..9 & {I,S,A,O,N} <: 0..9 & (1000*K+100*I+10*S+S) * (1000*K+100*I

Out[42]: TRUE

         Solution:
                 P = 4
                 A = 1
                 S = 3
                 I = 0
                 K = 2
                 N = 9
                 O = 8

In [43]: :solve z3 {K,P} <: 1..9 & {I,S,A,O,N} <: 0..9 & (1000*K+100*I+10*S+S) * (1000*K+100*I+1


        :solve: Computation not completed: no solution found (but one might exist)


In [44]: :solve kodkod {K,P} <: 1..9 & {I,S,A,O,N} <: 0..9 & (1000*K+100*I+10*S+S) * (1000*K+100

[2018-05-30 15:21:26,850, T+22662] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:26,851, T+22663] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn


Out[44]: TRUE

         Solution:
                 P = 4
                 A = 1
                 S = 3
                 I = 0
        K = 2
        N = 9
        O = 8
```

Result for KISS*KISS=PASSION puzzle:

Solver	Runtime
ProB Default	0.01 sec
Kodkod Backend	1 sec
Z3 Backend	? > 100 sec

1.6.3 Unbounded integers

The SAT translation via Kodkod/Alloy requires to determine the bid width. It cannot be applied to unbounded integers. Even for bounded integers it is quite tricky to get the bid widths correct: one needs also to take care of intermediate results. Alloy can detect incorrect models where an overflow occured, but to our understanding not where an overflow prevented a model (e.g., use inside negation or equivalence, see `#(V.SS->V.SS)=0 iff no V.SS` in paper at ABZ conference).

SMTLib is more tailored towards proof than towards model finding; as such it has typically no/less issues with unbounded values. The ProB default solver can also deal with unbounded integers: it tries to narrow down domains to finite ones. If this fails, an unbounded variable is enumerated (partially) and an **enumeration warning** is generated. In case a solution is found, this warning is ignored, otherwise the result of ProB's analysis is **UNKNOWN**. Some inconsistencies cannot be detected by interval/domain propagation; here it helps to activate ProB's CHR module which performs some additional inferences.

Let us perform some experiments. Both ProB and Z3 can solve the following:

```
In [45]: :solve z3 x*x=100

Out[45]: TRUE

         Solution:
                 x = −10
```

Here is an example where ProB generates an enumeration warning, but finds a solution:

```
In [46]: x>100 & x mod 2000 = 1 & x mod 3000 = 1

[2018-05-30 15:21:26,997, T+22809] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn


Out[46]: TRUE

         Solution:
                 x = 6001

In [47]: :solve z3 x>100 & x mod 2000 = 1 & x mod 3000 = 1
```

```
Out[47]: TRUE

         Solution:
                 x = 6001
```

Here ProB generates an enumeration warning and does not find a solution, hence the result is **UNKNOWN**. Here Z3 finds a solution.

```
In [48]: :solve prob x>100 & x mod 2000 = 1 & x mod 3000 = 1 & (x+x) mod 4501 = 0

[2018-05-30 15:21:27,139, T+22951] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn



        :solve: Computation not completed: no solution found (but one might exist)


In [49]: :solve z3 x>100 & x mod 2000 = 1 & x mod 3000 = 1 & (x+x) mod 4501 = 0

Out[49]: TRUE

         Solution:
                 x = 6756001
```

Here is an inconsistency which cannot be detected by CLP(FD)'s interval propagation. ProB can detect it with CHR enabled, but without the module the result is **UNKNOWN**.

```
In [50]: :solve z3 x>y & y>x

Out[50]: FALSE

In [51]: :solve prob x>y &y>x

[2018-05-30 15:21:29,975, T+25787] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:29,975, T+25787] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:29,976, T+25788] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:29,976, T+25788] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:29,976, T+25788] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn



        :solve: Computation not completed: no solution found (but one might exist)
```

1.6.4 Summary for Integer Arithmetic

Solver	Unbounded	Model Finding	Inconsistency Detection (Unbounded)
ProB CLP(FD)	yes	very good	limited with CHR
ProB Z3	yes	reasonable	very good
ProB Kodkod	no	good	-

1.6.5 Deferred and Enumerated Sets

Given an enumerated set `Trains = {thomas, gordon}` we associate a variable `x:Trains` with an integer decision variable in the range 1..2. Similarly, deferred sets are given a finite cardinality *n*, and a decision variables are in the range 1..n.

```
In [52]: x:Trains & y:Trains & x/=y

Out[52]: TRUE

         Solution:
                 x = thomas
                 y = gordon
```

1.7 Set Constraints

After booleans, integers and enumerated set elements, let us now move to constraint solving involving set variables.

1.7.1 Translation to SAT

The Kodkod/Alloy backend translates sets bit vectors. The size of the vector is the number of possible elements.

Take for example the following constraint:

```
In [53]: x <: 1..2 & y <: 1..2 & x \/ y = 1..2 & 1:x & x <<: y

Out[53]: TRUE

         Solution:
                 x = {1}
                 y = {1,2}

In [54]: {x1,x2,y1,y2} <: BOOL &
          x1=TRUE or y1=TRUE & x2=TRUE or y2=TRUE &   // x \/ y = 1..2
          x1=TRUE &   // 1:x
          (x1=TRUE => y1=TRUE) & (x2=TRUE => y2=TRUE) & (x1/=y1 or x2/=y2)  // x <<: y

Out[54]: TRUE

         Solution:
                 y1 = TRUE
                 x1 = TRUE
                 y2 = TRUE
                 x2 = FALSE
```

This translation to SAT is exactly what the Kodkod backend does:

```
In [55]: :solve kodkod x <: 1..2 & y<: 1..2 & x \/ y = 1..2 & 1:x & x <<: y
```

```
[2018-05-30 15:21:30,279, T+26091] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:30,280, T+26092] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
```

```
Out[55]: TRUE

         Solution:
                 x = {1}
                 y = {1,2}
```

Limitations of translating set constraints to SAT: - this cannot deal with **unbounded** sets: we need to know a finite type for each set, so that we can generate a finite bit vector - this approach cannot usually deal with **higher order** sets (sets of sets), as the size of the bit vector would be prohibitively large

Given that:

```
In [56]: card(POW(1..100))

Out[56]: 1267650600228229401496703205376
```

translating the following constraint to SAT would require a bit vector of length 1267650600228229401496703205376.

```
In [57]: x <: POW(1..100) & {100}:x & !y.(y:x => {card(y)}:x)

Out[57]: TRUE

         Solution:
                 x = {{100},{1}}
```

Also, in the following constraint, the set x is unbounded and no translation to SAT is feasible (without a very clever analysis of the universal implications).

```
In [58]: {100}:x & !y.(y:x => (!z.(z:y => y \/ {z / 2}:x)))

Out[58]: TRUE

         Solution:
                 x = {{100},{50,100},{25,50,100},{12,25,50,100},{6,12,25,50,100},{3,6,12,25,50,1
```

1.7.2 Translation to SMTLib

This can in principle deal with higher-order sets and unbounded sets, but makes heavy use of quantifiers. The practical usefulness is currently very limited.

```
In [22]: :solve z3 x ⊆ 1..2 & y ⊆ 1..2 & x ∪ y = 1..2

Out[22]: TRUE

         Solution:
                 x = ∅
                 y = {1,2}
```

Internally, the constraint is rewritten to support operators which do not exist in SMTLib:

```
In [18]: ∀ smt_tmp28.(smt_tmp28 ∈ x ⇒ smt_tmp28 ≥ 1 & 2 ≥ smt_tmp28) &
         ∀ smt_tmp29.(smt_tmp29 ∈ y ⇒ smt_tmp29 ≥ 1 & 2 ≥ smt_tmp29) &
         x ∪ y = {1,2}
```

```
Out[18]: TRUE

         Solution:
                 x = ∅
                 y = {1,2}
```

This in turn gets translated to SMTLib (calls to the Z3 API): - mk_var(set(integer),x) → 2 - mk_var(set(integer),y) → 3 - mk_bounded_var(integer,_smt_tmp28) → 4 - mk_op(member,4,2) → 5 - mk_int_const(1) → 6 - mk_op(greater_equal,4,6) → 7 - mk_int_const(2) → 8 - mk_op(greater_equal,8,4) → 9 - mk_op_arglist(conjunct,[7,9]) → 10 - mk_op(implication,5,10) → 11 - mk_quantifier(forall,[4],11) → 12 - mk_bounded_var(integer,_smt_tmp29) → 13 - mk_op(member,13,3) → 14 - mk_int_const(1) → 15 - mk_op(greater_equal) → 13,15,16 - mk_int_const(2) → 17 - mk_op(greater_equal,17,13) → 18 - mk_op_arglist(conjunct,[16,18]) → 19 - mk_op(implication,14,19) → 20 - mk_quantifier(forall,[13],20) → 21 - mk_op(union,2,3) → 22 - mk_int_const(1) → 23 - mk_int_const(2) → 24 - mk_set([23,24]) → 25 - mk_op(equal,22,25) → 26 - mk_op_arglist(conjunct,[12,21,26]) → 27

This can be solved by Z3 but not by CVC4. Already the slightly more complicated example from above (or the other examples) cannot be solved:

```
In [23]: :solve z3 x ⊆ 1..2 & y ⊆ 1..2 & x ∪ y = 1..2 & 1∈x & x ⊂ y
```

```
        :solve: Computation not completed: time out
```

Let us look at another relatively simple example which poses problems:

```
In [74]: :solve z3 f = {1|->3, 2|->6} & r = f~[{6}]
```

```
        :solve: Computation not completed: no solution found (but one might exist)
```

To understand why this simple constraint cannot be solved, we have to know how the translation works: The relational inverse gets translated into two universal quantifications for SMTLib:

```
 x = y~
<=>
 !(st11,st12).(st11 |-> st12 : x => st12 |-> st11 : y) &
 !(st11,st12).(st12 |-> st11 : y => st11 |-> st12 : x))
```

Similarly, r = f[s] is translated as follows:

```
 r = f[s]
<=>
 !st27.(st27 : r => #st26.(st26 |-> st27 : f & st26 : s) &
 !st27.(#st26.(st26 |-> st27 : f & st26 : s) => st27 : r)
```

The resulting predicate (without the inverse and image operators) is the following, which Z3 cannot solve (but ProB can).

```
In [15]: :prettyprint f = {(1|->3),(2|->6)} &
         #st13.(r = st13 & (
             !st15.(st15 : st13 => #st14.(#st16.(st14 |-> st15 : st16 &
             (!(st17,st18).(st17 |-> st18 : st16 => st18 |-> st17 : f) &
              !(st17,st18).(st18 |-> st17 : f => st17 |-> st18 : st16))) & st14 : {6})) &
              !st15.(#st14.(#st19.(st14 |-> st15 : st19 & (!(st20,st21).(st20 |-> st21 : st19 =>
              !(st20,st21).(st21 |-> st20 : f => st20 |-> st21 : st19))) & st14 : {6}) => st15 :
```

Out[15]:

$f = \{(1 \mapsto 3), (2 \mapsto 6)\} \wedge (\exists / * LET * / (st13).((st13) = r \wedge \forall st15 \cdot (st15 \in st13 \Rightarrow \exists st16 \cdot (6 \mapsto st15 \in st16 \wedge (\forall (st17, st18) \cdot (st17 \mapsto st18 \in st16 \Rightarrow st18 \mapsto st17 \in f) \wedge \forall (st17, st18) \cdot (st18 \mapsto st17 \in f \Rightarrow st17 \mapsto st18 \in st16)))) \wedge \forall st15 \cdot (\exists st19 \cdot (6 \mapsto st15 \in st19 \wedge (\forall (st20, st21) \cdot (st20 \mapsto st21 \in st19 \Rightarrow st21 \mapsto st20 \in f) \wedge \forall (st20, st21) \cdot (st21 \mapsto st20 \in f \Rightarrow st20 \mapsto st21 \in st19))) \Rightarrow st15 \in st13)))$

```
In [82]: :time :solve prob f = {(1|->3),(2|->6)} &
         #st13.(r = st13 & (
             !st15.(st15 : st13 => #st14.(#st16.(st14 |-> st15 : st16 &
             (!(st17,st18).(st17 |-> st18 : st16 => st18 |-> st17 : f) &
              !(st17,st18).(st18 |-> st17 : f => st17 |-> st18 : st16))) & st14 : {6})) &
              !st15.(#st14.(#st19.(st14 |-> st15 : st19 & (!(st20,st21).(st20 |-> st21 : st19 =>
              !(st20,st21).(st21 |-> st20 : f => st20 |-> st21 : st19))) & st14 : {6}) => st15 :

Execution time: 0.009604207 seconds


Out[82]: TRUE


         Solution:
                r = {2}
                f = {(1↦3),(2↦6)}

In [3]: :time :solve cvc4 f = {(1|->3),(2|->6)} &
        #st13.(r = st13 & (
            !st15.(st15 : st13 => #st14.(#st16.(st14 |-> st15 : st16 &
            (!(st17,st18).(st17 |-> st18 : st16 => st18 |-> st17 : f) &
             !(st17,st18).(st18 |-> st17 : f => st17 |-> st18 : st16))) & st14 : {6})) &
             !st15.(#st14.(#st19.(st14 |-> st15 : st19 & (!(st20,st21).(st20 |-> st21 : st19 =>
             !(st20,st21).(st21 |-> st20 : f => st20 |-> st21 : st19))) & st14 : {6}) => st15 :


        :time: :solve: Computation not completed: no solution found (but one might exist)
```

The SMTLib translation is still of limited value for finding models. However, for finding inconsistencies it is much better and can detect certain inconsistencies which ProB's solver cannot. While both ProB and Z3 can solve the following:

```
In [85]: :solve prob x:s1 & x:s2 & x /: (s1 /\ s2) & s1 <: INTEGER

Out[85]: FALSE
```

only the Z3 backend can solve this one:

```
In [90]: :solve z3 x:s1 & x/:s2 & x /: (s1 \/s2) & s1 <: INTEGER

Out[90]: FALSE
```

1.7.3 ProB's Set Solver

ProB has actually three set representations: - Prolog lists of elements - AVL trees for fully known sets - symbolic closures for large or infinite sets

For finite sets, the AVL tree representation is the most efficient and allows for efficient lookups. It, however, requires all elements to be fully known.

The symbolic closure can be used for large or infinite sets. ProB will automatically use it for sets it knows to be infinite, or when an enumeration warning occurs during an attempt at expanding a set.

The list representation is used for sets where some of the members are known or partially known.

AVL tree representation The following generates the AVL tree representation:

```
In [48]: {x|x∈0..2**10 & x mod 100 = 0}

Out[48]: {0,100,200,300,400,500,600,700,800,900,1000}
```

A lot of operators and predicates have optimised versions for the AVL tree represenation, e.g.,

```
In [49]: s = {x|x∈0..2**10 & x mod 100 = 0} &
         mx = max(s) &
         mn = min(s)

Out[49]: TRUE

         Solution:
                 mn = 0
                 s = {0,100,200,300,400,500,600,700,800,900,1000}
                 mx = 1000
```

Symbolic closure representation In the following case, ProB knows that the set is infinite and is kept symbolic:

```
In [64]: {x|x>1000}
```

```
Out[64]: {x|x > 1000}
```

Symbolic sets can be used in various ways:

```
In [65]: inf = {x|x>1000} & 1024 : inf & not(1000:inf) & res  = (900..1100) ∩ inf
```

```
Out[65]: TRUE

         Solution:
                 inf = {x|x > 1000}
                 res = (1001 [U+2025] 1100)
```

For the following set, ProB tries to expand it and then an enumeration warning occurs (also called a **virtual timeout**, ProB realises that no matter what time budget it would be given a timeout would occur). The set is then kept symbolic and can again be used in various ways.

```
In [66]: inf = {x|x>1000 & x mod 25 = 0} & 1025 ∈ inf & not(1000∈inf) & res  = (900..1100) ∩ in
```

```
[2018-05-30 15:21:33,608, T+29420] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
```

```
Out[66]: TRUE

         Solution:
                 inf = {x|x > 1000 ∧ x mod 25 = 0}
                 res = ((900 [U+2025] 1100) ∩ {x|x > 1000 ∧ x mod 25 = 0})
```

The virtual timeout message can be removed (and performance improved) by adding the symbolic pragma:

```
In [50]: inf = /*@symbolic*/ {x|x>1000 & x mod 25 = 0} & 1025 ∈ inf & not(1000∈inf) & res  = (9
```

```
Out[50]: TRUE

         Solution:
                 inf = {x|x > 1000 ∧ x mod 25 = 0}
                 res = ((900 [U+2025] 1100) ∩ {x|x > 1000 ∧ x mod 25 = 0})
```

Internally, a symbolic representation is a **closure** in functional programming terms: all dependent variables are *compiled* into the closure: the closure can be passed as a value and evaluated without needing access to an environment. In Prolog this is represented as a tuple: - closure(Parameters,Types,CompiledPredicate) For example, a set {x|x>v} where v has the value 17 is compiled to: - closure([x],[integer],x>17)

List representation The list representation is used when a finite set is partially known and constraint solving has to determine the set.

```
In [68]: vec: 1..10 --> 0..9999 &
         vec(1) : {1,10} &
         !x.(x:2..10 => vec(x) = vec(x-1)*2)

Out[68]: TRUE

         Solution:
                 vec = {(1↦1),(2↦2),(3↦4),(4↦8),(5↦16),(6↦32),(7↦64),(8↦128),(9↦256),(
```

Note that Kodkod translation and SMT translation not very effective for the above. The Kodkod translation can deal with a simpler version of the above:

```
In [69]: :time :solve kodkod vec: 1..8 --> 0..199 & vec(1) : {1,10} & !x.(x:2..8 => vec(x) = vec

[2018-05-30 15:21:36,367, T+32179] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:36,368, T+32180] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
Execution time: 2.572491877 seconds


Out[69]: TRUE

         Solution:
                 vec = {(3↦4),(5↦16),(6↦32),(7↦64),(1↦1),(2↦2),(4↦8),(8↦128)}

In [70]: :time :solve prob vec: 1..8 --> 0..199 & vec(1) : {1,10} & !x.(x:2..8 => vec(x) = vec(x

Execution time: 0.018834117 seconds


Out[70]: TRUE

         Solution:
                 vec = {(1↦1),(2↦2),(3↦4),(4↦8),(5↦16),(6↦32),(7↦64),(8↦128)}

In [71]: :time :solve z3 vec: 1..8 --> 0..199 & vec(1) : {1,10} & !x.(x:2..8 => vec(x) = vec(x-1

[2018-05-30 15:21:39,027, T+34839] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn



        :time: :solve: Computation not completed: time out
```

What about explicit computations? How well does the SMTLib translation fare here?

```
In [53]: :solve z3 x = 1..1000 /\ (200..300)
```

```
        :solve: Computation not completed: no solution found (but one might exist)

In [58]: :time :solve z3 x = 1..40 /\ (6..15)

Execution time: 0.183256698 seconds


Out[58]: TRUE

         Solution:
                 x = {6,7,8,9,10,11,12,13,14,15}

In [59]: :time :solve z3 x = 1..60 /\ (6..15)

Execution time: 0.739173891 seconds


Out[59]: TRUE

         Solution:
                 x = {6,7,8,9,10,11,12,13,14,15}

In [60]: :time :solve z3 x = 1..80 /\ (6..15)

Execution time: 2.059420909 seconds


Out[60]: TRUE

         Solution:
                 x = {6,7,8,9,10,11,12,13,14,15}

In [61]: :time :solve prob x = 1..80 /\ (6..15)

Execution time: 0.005402824 seconds


Out[61]: TRUE

         Solution:
                 x = (6 [U+2025] 15)
```

In the following the inverse operator seems to pose problems to Z3:

```
In [51]: :solve z3 s1 = {2,3,5,7,11} & s2 = {4,8,16,32} & c = s1*s2 & r=c~[{8}]


        :solve: Computation not completed: no solution found (but one might exist)
```

```
In [52]: :solve prob s1 = {2,3,5,7,11} & s2 = {4,8,16,32} & c = s1*s2 & r=c~[{8}]

Out[52]: TRUE

         Solution:
                 r = {2,3,5,7,11}
                 c = ({2,3,5,7,11} ∗ {4,8,16,32})
                 s1 = {2,3,5,7,11}
                 s2 = {4,8,16,32}
```

1.7.4 ProB's Solving Algorithm

ProB tries to accomplish several conflicting goals: - being able to deal with concrete data, i.e., sets and relations containing thousands or hundreds of thousands of elementas - being able to deal with symbolic, infinite sets, relations and functions. - being able to perform efficient computation over large data as well as constraint solving

For example, efficient computation over large concrete data is the following:

```
In [72]: :time :solve prob s1 = {x|x:1..10**n & x mod n = 0} & s2 = {y|y:1..10**n & y mod (n+1)

Execution time: 0.508970360 seconds


Out[72]: TRUE

         Solution:
                 s3 = ∃500∈{20,40,...,9980,10000}
                 n = 4
                 s1 = ∃2500∈{4,8,...,9996,10000}
                 s2 = ∃2000∈{5,10,...,9995,10000}
```

Here is a simple verison of the above

```
In [73]: :time :solve prob x = 1..n & y = 2*n..3*n & n = 100 & xy = x \/ y

Execution time: 0.020065212 seconds


Out[73]: TRUE

         Solution:
                 xy = ∃201∈{1,2,...,299,300}
                 x = (1 [U+2025] 100)
                 y = (200 [U+2025] 300)
                 n = 100

In [74]: :solve z3 x = 1..n & y = 2*n..3*n & n = 100 & xy = x \/ y

[2018-05-30 15:21:42,299, T+38111] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
```

```
:solve: Computation not completed: time out
```

```
In [75]: :time :solve kodkod x = 1..n & y = 2*n..3*n & n = 100 & xy = x \/ y
```

```
[2018-05-30 15:21:42,809, T+38621] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:42,810, T+38622] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
Execution time: 0.472061287 seconds
```

```
Out[75]: TRUE

         Solution:
                 xy = {3,5,6,7,9,10,11,12,13,14,15,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
                 x = {3,5,6,7,9,10,11,12,13,14,15,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,3
                 y = {200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,21
                 n = 100
```

ProB employs the *Andorra* principle: deterministic computations are done first. As there are multiple set representations, there are actually two kinds of deterministic computations: - deterministic computations that generate an efficient representation, e.g., an AVL set representation - and other deterministic computations

The ProB solver has a **WAITFLAG store** where choice points and enumerations are registered with a given priority. - Priority 0 means that an efficient representation can be generated - Priority 1 is a deterministic computation not guaranteed to produce an efficient representation - Priority k is a choice point/enumeration which may generate k possible values

At each solving step one waitflag is activated, the one with the lowest priority. CLP(FD) variables are also registered in the WAITFLAG store and are enumerated before a waitflag of the same priority is activated. For tie breaking one typically uses the **most attached constraints first** (ffc) heuristic.

Example Let us examine how `x = 1..n & y = 2*n..3*n & n = 100 & xy = x \/ y` is solved. - all constraints are registered - in phase 0 `n=100` is run - this means that `1..n` can be efficiently computed - this means that `x = 1..n` triggers in phase 0 - then `2*n` and `3*n` can be computed, followed by `2*n..3*n` - this means that `y = 2*n..3*n` triggers in phase 0 - again, this means that `x \/ y` can be efficiently computed - finally `xy = x \/ y` can be executed in phase 0

No enumeration was required. In this case ProB's constraint solver works similar to a topological sorting algorithm.

Dealing with unbounded enumeration Note: if an unbounded enumeration is encountered, the solver registers an **enumeration warning** in the current scope (every quantification / comprehension set results in a new inner scope). Depending on the kind of scope (existential/universal) and on whether a solution is found, the warning gets translated into an **UNKNOWN** result.

1.7.5 Functional Programming

Some functions are automatically detected as infinite by ProB, are kept symbolic but can be applied in several ways:

```
In [76]: f = %x.(x:INTEGER|x*x) &
         r1 = f(100000) &
         r2 = f[1..10] &
         r3 = ([2,3,5,7,11] ; f) &
         r4 = iterate(f,3)(2) &
         f(sqrt) = 100

Out[76]: TRUE

         Solution:
                 r2 = {1,4,9,16,25,36,49,64,81,100}
                 r3 = [4,9,25,49,121]
                 r4 = 256
                 sqrt = 10
                 f = λx·(x ∈ INTEGER|x ∗ x)
                 r1 = 10000000000

In [77]: f = {x,y|x:NATURAL & y**2 >= x & (y-1)**2 <x } & // integer square root function
         r1 = f(100000) &
         r2 = f[1..10] &
         r3 = ([2,3,5,7,11] ; f) &
         r4 = iterate(f,3)(2) &
         f(sqr) = 100 &
         r5 = closure1(f)[{10000}]

[2018-05-30 15:21:43,022, T+38834] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:43,023, T+38835] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:43,024, T+38836] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:43,056, T+38868] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn


Out[77]: TRUE

         Solution:
                 r2 = {1,2,3,4}
                 r3 = [2,2,3,3,4]
                 r4 = 2
                 r5 = {2,4,10,100}
                 sqr = 9802
                 f = {x,y|x ∈ NATURAL ∧ y × 2 ≥ x ∧ (y − 1) × 2 < x}
                 r1 = 317
```

1.7.6 Reification

Reification is linking the truth value of a constraint with a boolean variable. ProB's kernel provides support for reifying a considerable number of constraints (but not yet all!). Reification is important for efficiency, to avoid choice points and is important to link various solvers of ProB (set, arithmetic, boolean,...).

```
In [78]: (x>100 <=> (ReifVar=TRUE)) & (x<125 <=> (ReifVar=FALSE)) & x<200

Out[78]: TRUE

         Solution:
                 x = 125
                 ReifVar = TRUE
```

1.7.7 Relation to SETLOG

Setlog (http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html) is based on non-deterministic set unification Setlog has additional inference rules

The former causes problems with larger sets, in our experience. The latter could be added to ProB via CHR, but currently not done.

Let us look at a simpler Setlog example from the article "{log} as a Test Case Generator for the Test Template Framework" by Cristia, Rossi and Frydman:

```
1 in R & 1 nin S & inters(R,S,T) & T = {X}
```

This can be encoded in B as follows:

```
In [79]: 1:R & 1/:S & R/\S=T & T={X}

Out[79]: TRUE

         Solution:
                 R = {1,0}
                 S = {0}
                 T = {0}
                 X = 0
```

Another example from that paper is

```
X in int(1,5) & Y in int(4,10) & inters({X},{Y},R) & X >= Y
```

which has three solutions

```
X=4,Y=4,R={4}; X=5,Y=5,R={5}; X=5,Y=4,R={}.
```

Let us check this with ProB:

```
In [80]: {X,Y,R|X: 1..5 & Y: 4..10 & {X}/\{Y}=R & X>=Y}

Out[80]: {((4↦4)↦{4}),((5↦4)↦∅),((5↦5)↦{5})}
```

However, in particular with unbounded sets Setlog can solve some constraints that ProB, Z3 and Kodkod cannot:

```
un(A,B,D) & disj(A,C) & D=C & ris(X in A,[],true,X) neq {}
```

In B this corresponds to:

```
In [81]: A \/ B = D & A /\ C = {} & D=C & A /= {} & A:POW(STRING)

[2018-05-30 15:21:43,336, T+39148] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:43,337, T+39149] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn



        :eval: UNKNOWN (FALSE with enumeration warning)
```

From A/B=D we and D=C we could infer A<:C and hence A/=C and hence A={} which is in conflict with A/={}. ProB (as well as Kodkod and Z3 backends) can only infer the conflict for finite domains:

```
In [82]: A \/ B = D & A /\ C = {} & D=C & A /= {} & A:POW(BOOL)

Out[82]: FALSE

In [83]: :solve kodkod A \/ B = D & A /\ C = {} & D=C & A /= {} & A:POW(BOOL)

[2018-05-30 15:21:43,486, T+39298] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn
[2018-05-30 15:21:43,487, T+39299] "ProB Output Logger for instance 762b9e9a" de.prob.cli.ProBIn


Out[83]: FALSE

In [84]: :solve z3 A \/ B = D & A /\ C = {} & D=C & A /= {} & A:POW(BOOL)

Out[84]: FALSE
```

Setlog has problems with larger sets. For example, the following takes 24 seconds using the latest stable release 4.9.1 of Setlog from http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html:

```
{log}=> diff(int(1,200),{50},R).

R = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,3

Another solution?  (y/n)
```

In ProB this is instantenous:

```
In [85]: :time R= (1..200) \ {50}

Execution time: 0.006102410 seconds


Out[85]: TRUE


         Solution:
                R = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28
```

1.8 Summary of Set Constraint Solving Approaches

- SAT Translation (Kodkod backend):
- needs finite and small base type, no unbounded or higher-order sets
- can be very effective for complex constraints involving image, transitive closure,...
- limited performance for large sets
- SMTLib Translation (Z3/CVC4 backend):
- can deal with unbounded and large sets
- due to heavy use of quantifiers, some finite constraints get translated into infinite ones: limited model finding capabilities
- ProB's default backend:
- can deal with unbounded and large sets
- limited constraint solving for complex constraints involving image, transitive closure,...
- works well for large sets and semi-deterministic computation
- works well for animation, data validation, disproving
- limitations trigger for symbolic model checking (IC3,...)
- Future work: improve combination with Z3, improve list representation (maybe use a bit-vector like representation)

1.9 Appendix

Some further examples for comparison of the backends

```
In [2]: :time :solve prob f: 1..n --> 1..n & !x.(x:2..n => f(x)=f(x-1)+1) & n=50

Execution time: 0.062737937 seconds


Out[2]: TRUE

        Solution:
                f = {(1↦1),(2↦2),(3↦3),(4↦4),(5↦5),(6↦6),(7↦7),(8↦8),(9↦9),(10↦10),(1
                n = 50

In [3]: :time :solve kodkod f: 1..n --> 1..n & !x.(x:2..n => f(x)=f(x-1)+1) & n=50

[2018-06-01 10:06:02,879, T+73563] "ProB Output Logger for instance 6e28bb87" de.prob.cli.ProBIn
[2018-06-01 10:06:02,880, T+73564] "ProB Output Logger for instance 6e28bb87" de.prob.cli.ProBIn
[2018-06-01 10:06:02,881, T+73565] "ProB Output Logger for instance 6e28bb87" de.prob.cli.ProBIn
Execution time: 5.019394450 seconds


Out[3]: TRUE

        Solution:
                f = {(3↦3),(5↦5),(6↦6),(7↦7),(9↦9),(10↦10),(11↦11),(12↦12),(13↦13),(14
                n = 50

In [4]: :time :solve z3 f: 1..n --> 1..n & !x.(x:2..n => f(x)=f(x-1)+1) & n=50
```

```
[2018-06-01 10:06:16,981, T+87665] "ProB Output Logger for instance 6e28bb87" de.prob.cli.ProBIn
```

```
        :time: :solve: Computation not completed: time out
```

1.9.1 Fast with Kodkod

Example with relational composition and image

```
In [13]: :time :solve prob r: 1..5 <-> 1..5 & (r;r) = r & r /= {} & dom(r)=1..5 & r[2..3]=3..4

Execution time: 2.101874422 seconds


Out[13]: TRUE

         Solution:
                 r = {(1↦1),(1↦2),(1↦3),(1↦4),(1↦5),(2↦3),(2↦4),(3↦3),(3↦4),(4↦3),(4↦

In [14]: :time :solve kodkod r: 1..5 <-> 1..5 & (r;r) = r & r /= {} & dom(r)=1..5 & r[2..3]=3..4

[2018-06-01 10:10:23,094, T+333778] "ProB Output Logger for instance 6e28bb87" de.prob.cli.ProBI
[2018-06-01 10:10:23,095, T+333779] "ProB Output Logger for instance 6e28bb87" de.prob.cli.ProBI
Execution time: 0.077583935 seconds


Out[14]: TRUE

         Solution:
                 r = {(3↦3),(3↦4),(5↦4),(1↦4),(2↦4),(4↦4)}

In [15]: :time :solve z3 r: 1..5 <-> 1..5 & (r;r) = r & r /= {} & dom(r)=1..5 & r[2..3]=3..4


        :time: :solve: Computation not completed: no solution found (but one might exist)


In [12]: :time :solve prob r: 1..5 <-> 1..5 & (r;r) = r & r /= {} & dom(r)=1..5 & r[2..3]=3..4

Execution time: 2.057316736 seconds


Out[12]: TRUE

         Solution:
                 r = {(1↦1),(1↦2),(1↦3),(1↦4),(1↦5),(2↦3),(2↦4),(3↦3),(3↦4),(4↦3),(4↦
```

Graph theorem Theorem: all undirected graphs without self-loops (and no 0-vertices) have two nodes with the same degree.

```
In [58]: ::load
         MACHINE GraphTheorem
         SETS NODES5
         PROPERTIES card(NODES5)=5
         END
```

```
[2018-06-03 08:28:52,235, T+120880197] "Shell-0" de.prob.cli.PrologProcessProvider.makeProcess(P
[2018-06-03 08:28:53,304, T+120881266] "Shell-0" de.prob.cli.PortPattern.setValue(PortPattern.ja
[2018-06-03 08:28:53,305, T+120881267] "Shell-0" de.prob.cli.InterruptRefPattern.setValue(Interr
[2018-06-03 08:28:53,307, T+120881269] "ProB Output Logger for instance 68578a" de.prob.cli.ProB
[2018-06-03 08:28:53,322, T+120881284] "ProB Output Logger for instance 68578a" de.prob.cli.ProB
```

```
Out[58]: Loaded machine: GraphTheorem : []
```

```
In [62]: :prettyprint edges : NODES5 <-> NODES5 &
                 edges~=edges &
                 not(#(n1,n2).(n1:NODES5 & n2:NODES5 & n2/=n1 &
                 card(edges[{n1}]) = card(edges[{n2}]))) &
                 dom(edges)=NODES5 & id(NODES5) /\ edges = {}
```

Out[62]:

$edges \in NODES5 \leftrightarrow NODES5 \wedge edges^{-1} = edges \wedge \neg(\exists(n1,n2) \cdot (n1 \in NODES5 \wedge n2 \in NODES5 \wedge n2 \neq n1 \wedge card(edges[\{n1\}]) = card(edges[\{n2\}]))) \wedge dom(edges) = NODES5 \wedge id(NODES5) \cap edges = \varnothing$

```
In [32]: :time :solve prob  edges : NODES5 <-> NODES5 &
                 edges~=edges &
                 not(#(n1,n2).(n1:NODES5 & n2:NODES5 & n2/=n1 &
                 card(edges[{n1}]) = card(edges[{n2}]))) &
                 dom(edges)=NODES5 & id(NODES5) /\ edges = {}


        :time: :solve: Computation not completed: time out
```

```
In [31]: :time :solve kodkod  edges : NODES5 <-> NODES5 &
                 edges~=edges &
                 not(#(n1,n2).(n1:NODES5 & n2:NODES5 & n2/=n1 &
                 card(edges[{n1}]) = card(edges[{n2}]))) &
                 dom(edges)=NODES5 & id(NODES5) /\ edges = {}
```

```
[2018-06-01 10:33:19,930, T+1710614] "ProB Output Logger for instance 1cb4e9b8" de.prob.cli.ProB
[2018-06-01 10:33:19,931, T+1710615] "ProB Output Logger for instance 1cb4e9b8" de.prob.cli.ProB
Execution time: 1.110113990 seconds
```

```
Out[31]: FALSE
```

```
In [33]: :time :solve z3  edges : NODES5 <-> NODES5 &
               edges~=edges &
               not(#(n1,n2).(n1:NODES5 & n2:NODES5 & n2/=n1 &
               card(edges[{n1}]) = card(edges[{n2}]))) &
               dom(edges)=NODES5 & id(NODES5) /\ edges = {}
```

```
        :time: :solve: Computation not completed: no solution found (but one might exist)
```

So with Kodkod we proved the theorem for 5 nodes in about a second. What if we remove the self-loops restriction:

```
In [35]: :time :solve kodkod  edges : NODES5 <-> NODES5 &
               edges~=edges &
               not(#(n1,n2).(n1:NODES5 & n2:NODES5 & n2/=n1 &
               card(edges[{n1}]) = card(edges[{n2}]))) &
               dom(edges)=NODES5
```

```
[2018-06-01 10:37:43,552, T+1974236] "ProB Output Logger for instance 1cb4e9b8" de.prob.cli.ProB
[2018-06-01 10:37:43,552, T+1974236] "ProB Output Logger for instance 1cb4e9b8" de.prob.cli.ProB
Execution time: 0.138100698 seconds
```

```
Out[35]: TRUE

         Solution:
                edges = {(NODES51↦NODES54),(NODES52↦NODES52),(NODES52↦NODES53),(NODES52↦NOI
```

```
In [38]: :time :solve kodkod  edges : NODES5 <-> NODES5 &
               edges~=edges &
               not(#(n1,n2).(n1:dom(edges) & n2:dom(edges) & n2/=n1 &
                            card(edges[{n1}]) = card(edges[{n2}]))) &
               id(NODES5) /\ edges = {} & edges /= {}
```

```
[2018-06-01 10:40:25,459, T+2136143] "ProB Output Logger for instance 1cb4e9b8" de.prob.cli.ProB
[2018-06-01 10:40:25,460, T+2136144] "ProB Output Logger for instance 1cb4e9b8" de.prob.cli.ProB
Execution time: 0.045835571 seconds
```

```
Out[38]: FALSE
```

1.9.2 Benchmark Puzzles

N-Queens

```
In [65]: :time :solve prob n=8 &
             queens : 1..n >-> 1..n &
             !(q1,q2).(q1:1..n & q2:2..n & q2>q1 => queens(q1)+(q2-q1) /= queens(q2) & queens(q1
Execution time: 0.011939124 seconds
```

```
Out[65]: TRUE

         Solution:
                 queens = {(1↦4),(2↦2),(3↦7),(4↦3),(5↦6),(6↦8),(7↦5),(8↦1)}
                 n = 8
```

```
In [66]: :time :solve z3 n=8 &
             queens : 1..n >-> 1..n &
             !(q1,q2).(q1:1..n & q2:2..n & q2>q1 => queens(q1)+(q2-q1) /= queens(q2) & queens(q1
```

```
Execution time: 1.720856952 seconds
```

```
Out[66]: TRUE

         Solution:
                 queens = {(1↦4),(2↦2),(3↦5),(4↦8),(5↦6),(6↦1),(7↦3),(8↦7)}
                 n = 8
```

```
In [67]: :time :solve kodkod n=8 &
             queens : 1..n >-> 1..n &
             !(q1,q2).(q1:1..n & q2:2..n & q2>q1 => queens(q1)+(q2-q1) /= queens(q2) & queens(q1
```

```
[2018-06-01 10:52:53,861, T+2884545] "ProB Output Logger for instance 1cb4e9b8" de.prob.cli.ProB
[2018-06-01 10:52:53,861, T+2884545] "ProB Output Logger for instance 1cb4e9b8" de.prob.cli.ProB
Execution time: 0.303716540 seconds
```

```
Out[67]: TRUE

         Solution:
                 queens = {(3↦2),(5↦5),(6↦1),(7↦8),(1↦3),(2↦6),(4↦7),(8↦4)}
                 n = 8
```

N-Bishops

```
In [70]: :time :solve prob n=3 & bshp <: (1..n)*(1..n) &
           !(i,j).({i,j}<:1..n => ( (i,j): bshp => (!k.(k:(i+1)..n => (k,j+k-i) /: bshp & (k,j-k
           card(bshp) = 3
```

```
Execution time: 0.035817988 seconds
```

```
Out[70]: TRUE

         Solution:
                 bshp = {(1↦1),(1↦2),(1↦3)}
                 n = 3
```

```
In [71]: :time :solve z3 n=3 & bshp <: (1..n)*(1..n) &
           !(i,j).({i,j}<:1..n => ( (i,j): bshp => (!k.(k:(i+1)..n => (k,j+k-i) /: bshp & (k,j-k
           card(bshp) = 3

[2018-06-01 10:56:26,064, T+3096748] "ProB Output Logger for instance 1cb4e9b8" de.prob.cli.ProB



        :time: :solve: Computation not completed: time out
```

Solving takes about 150 seconds; Kodkod translation currently does not work due to card and preventing overflows.

```
In [47]: :prettyprint n=3 & bshp <: (1..n)*(1..n)  &
           !(i,j).({i,j}<:1..n => ( (i,j): bshp => (!k.(k:(i+1)..n => (k,j+k-i) /: bshp & (k,j-k
           card(bshp) = 3
```

Out[47]:

$n = 3 \land bshp \subseteq (1..n) \times (1..n) \land \forall (i,j) \cdot (\{i,j\} \subseteq 1..n \Rightarrow (i \mapsto j \in bshp \Rightarrow \forall k \cdot (k \in i+1..n \Rightarrow (k \mapsto (j+k)-i) \notin bshp \land (k \mapsto (j-k)+i) \notin bshp))) \land card(bshp) = 3$

1.9.3 Other examples

```
In [63]: !x.(x:1..2 => !y.(y:1..3 => (x,y):r))

Out[63]: TRUE

         Solution:
                 r = {(1↦1),(1↦2),(1↦3),(2↦1),(2↦2),(2↦3)}

In [64]: :solve z3 !x.(x:1..2 => !y.(y:1..3 => (x,y):r))

[2018-06-04 08:29:46,794, T+207334756] "ProB Output Logger for instance 68578a" de.prob.cli.ProB


Out[64]: TRUE

         Solution:
                 r = {z_|z_ ∈ INTEGER ∗ INTEGER}
```

For unbounded problem above, Z3 can find a model, but for the one below not (even with mbqi and pull_nested_quantifiers).

```
In [65]: :solve z3 !x.(x:1..2 => !y.(y:1..3 => (x,y):r)) & r<: (1..10)*(1..10)

[2018-06-04 08:30:02,522, T+207350484] "ProB Output Logger for instance 68578a" de.prob.cli.ProB
```

```
        :solve: Computation not completed: no solution found (but one might exist)
```

```
In [67]: :time :solve kodkod !x.(x:1..2 => !y.(y:1..3 => (x,y):r)) & r<: (1..10)*(1..10)

[2018-06-04 08:30:14,658, T+207362620] "ProB Output Logger for instance 68578a" de.prob.cli.ProB
[2018-06-04 08:30:14,658, T+207362620] "ProB Output Logger for instance 68578a" de.prob.cli.ProB
Execution time: 0.101914781 seconds


Out[67]: TRUE

         Solution:
                 r = {(1↦3),(1↦1),(1↦2),(2↦3),(2↦1),(2↦2)}
```

```
In [68]: :time :solve prob !x.(x:1..2 => !y.(y:1..3 => (x,y):r)) & r<: (1..10)*(1..10)

Execution time: 0.065749342 seconds


Out[68]: TRUE

         Solution:
                 r = {(1↦1),(1↦2),(1↦3),(2↦1),(2↦2),(2↦3)}
```

Datavalidation example

```
In [69]: ::load
         MACHINE
             signals

         SETS
         /* Ensemble des signaux */
                    SIGNAL =
                    {    PL01
                    ,    PL02
                    ,    PL03
                    ,    PL04
                    ,    PL05
                    ,    PL06
                    ,    PL07
                    ,    PL08
                    ,    PL09
                    ,    PL10
                    ,    PL11
                    ,    PL12
                 ,         PL13
                    ,    PL14
```

```
                ,    PL15
                ,    PL16
                ,    PL17
                ,    PL18
                ,    PL19
                ,    PL20
                }
     END
```

```
[2018-06-04 09:48:47,308, T+212075270] "Shell-0" de.prob.cli.PrologProcessProvider.makeProcess(P
[2018-06-04 09:48:48,865, T+212076827] "Shell-0" de.prob.cli.PortPattern.setValue(PortPattern.ja
[2018-06-04 09:48:48,866, T+212076828] "Shell-0" de.prob.cli.InterruptRefPattern.setValue(Interr
[2018-06-04 09:48:48,873, T+212076835] "ProB Output Logger for instance 7587ef64" de.prob.cli.Pr
[2018-06-04 09:48:48,885, T+212076847] "ProB Output Logger for instance 7587ef64" de.prob.cli.Pr
```

```
Out[69]: Loaded machine: signals : []
```

```
In [78]: :time :solve prob nxt =
             {PL01 |-> PL02, PL02 |-> PL03, PL03 |-> PL04, PL04 |-> PL05,
              PL05 |-> PL06, PL06 |-> PL07, PL07 |-> PL08, PL08 |-> PL09,
              PL09 |-> PL10, PL10 |-> PL11, PL11 |-> PL11, PL12 |-> PL13,
              PL13 |-> PL14, PL14 |-> PL15, PL15 |-> PL16, PL16 |-> PL17,
              PL17 |-> PL18, PL18 |-> PL19, PL19 |-> PL20, PL20 |-> PL20} &
              res = SIGNAL \ nxt[SIGNAL]
```

```
Execution time: 0.010791768 seconds
```

```
Out[78]: TRUE

         Solution:
                 res = {PL01,PL12}
                 nxt = {(PL01↦PL02),(PL02↦PL03),(PL03↦PL04),(PL04↦PL05),(PL05↦PL06),(PL06↦
```

```
In [76]: :time :solve z3 nxt =
             {PL01 |-> PL02, PL02 |-> PL03, PL03 |-> PL04, PL04 |-> PL05,
              PL05 |-> PL06, PL06 |-> PL07, PL07 |-> PL08, PL08 |-> PL09,
              PL09 |-> PL10, PL10 |-> PL11, PL11 |-> PL11, PL12 |-> PL13,
              PL13 |-> PL14, PL14 |-> PL15, PL15 |-> PL16, PL16 |-> PL17,
              PL17 |-> PL18, PL18 |-> PL19, PL19 |-> PL20, PL20 |-> PL20} &
              res = SIGNAL \ nxt[SIGNAL]
```

```
        :time: :solve: Computation not completed: time out
```

```
In [77]: :time :solve kodkod nxt =
             {PL01 |-> PL02, PL02 |-> PL03, PL03 |-> PL04, PL04 |-> PL05,
              PL05 |-> PL06, PL06 |-> PL07, PL07 |-> PL08, PL08 |-> PL09,
              PL09 |-> PL10, PL10 |-> PL11, PL11 |-> PL11, PL12 |-> PL13,
              PL13 |-> PL14, PL14 |-> PL15, PL15 |-> PL16, PL16 |-> PL17,
              PL17 |-> PL18, PL18 |-> PL19, PL19 |-> PL20, PL20 |-> PL20} &
              res = SIGNAL \ nxt[SIGNAL]

[2018-06-04 09:54:55,813, T+212443775] "ProB Output Logger for instance 7587ef64" de.prob.cli.Pr
[2018-06-04 09:54:55,814, T+212443776] "ProB Output Logger for instance 7587ef64" de.prob.cli.Pr
Execution time: 0.046603293 seconds
```

```
Out[77]: TRUE

         Solution:
                 res = {PL01,PL12}
                 nxt = {(PL01↦PL02),(PL02↦PL03),(PL03↦PL04),(PL04↦PL05),(PL05↦PL06),(PL06↦
```

```
In [86]: ::load
         MACHINE
             signals

         SETS
         /* Ensemble des signaux */
                    SIGNAL =
                    {    PL01
                    ,    PL02
                    ,    PL03
                    ,    PL04
                    ,    PL05
                    ,    PL06
                    ,    PL07
                    ,    PL08
                    ,    PL09
                    ,    PL10
                    ,    PL11
                    ,    PL12
                 ,         PL13
                    ,    PL14
                    ,    PL15
                    ,    PL16
                    ,    PL17
                    ,    PL18
                    ,    PL19
                    ,    PL20
                    ,    PL21
                    ,    PL22
                 ,         PL23
                    ,    PL24
                    ,    PL25
                    ,    PL26
                    ,    PL27
                    ,    PL28
                    ,    PL29
                    ,    PL30
                    ,    PL31
                    ,    PL32
                 ,          PL33
                    ,    PL34
                    ,    PL35
                    ,    PL36
                    ,    PL37
                    ,    PL38
                    ,    PL39
                    ,    PL40
                    }
        END

[2018-06-04 10:14:49,390, T+213637352] "Shell-0" de.prob.cli.PrologProcessProvider.makeProcess(P
[2018-06-04 10:14:50,859, T+213638821] "Shell-0" de.prob.cli.PortPattern.setValue(PortPattern.ja
[2018-06-04 10:14:50,860, T+213638822] "Shell-0" de.prob.cli.InterruptRefPattern.setValue(Interr
[2018-06-04 10:14:50,867, T+213638829] "ProB Output Logger for instance 758ad42d" de.prob.cli.Pr
[2018-06-04 10:14:50,881, T+213638843] "ProB Output Logger for instance 758ad42d" de.prob.cli.Pr
```

```
Out[86]: Loaded machine: signals : []
```

```
In [98]: :time :solve kodkod nxt =
             {PL01 |-> PL02, PL02 |-> PL03, PL03 |-> PL04, PL04 |-> PL05,
              PL05 |-> PL06, PL06 |-> PL07, PL07 |-> PL08, PL08 |-> PL09,
              PL09 |-> PL10, PL10 |-> PL11, PL11 |-> PL12, PL12 |-> PL13,
              PL13 |-> PL14, PL14 |-> PL15, PL15 |-> PL16, PL16 |-> PL17,
              PL17 |-> PL18, PL18 |-> PL19, PL19 |-> PL20, PL20 |-> PL20,
              PL30 |-> PL31, PL31 |-> PL32, PL33 |-> PL34, PL34 |-> PL35} &
              cl1 = closure1(nxt) &
              nrs = %x.(x:SIGNAL|card(cl1[{x}])) &
              mx = nrs~[{max(ran(nrs))}]

[2018-06-04 10:21:44,461, T+214052423] "ProB Output Logger for instance 758ad42d" de.prob.cli.Pr
[2018-06-04 10:21:44,461, T+214052423] "ProB Output Logger for instance 758ad42d" de.prob.cli.Pr
Execution time: 0.047563731 seconds
```

```
Out[98]: TRUE

        Solution:
                cl1 = ∃197∈{(PL01↦PL02),(PL01↦PL03),...,(PL33↦PL35),(PL34↦PL35)}
```

```
                  mx = {PL01}
                  nxt = {(PL01↦PL02),(PL02↦PL03),(PL03↦PL04),(PL04↦PL05),(PL05↦PL06),(PL06↦
                  nrs = {(PL01↦19),(PL02↦18),(PL03↦17),(PL04↦16),(PL05↦15),(PL06↦14),(PL07
```

```
In [97]: :time :solve prob nxt =
            {PL01 |-> PL02, PL02 |-> PL03, PL03 |-> PL04, PL04 |-> PL05,
             PL05 |-> PL06, PL06 |-> PL07, PL07 |-> PL08, PL08 |-> PL09,
             PL09 |-> PL10, PL10 |-> PL11, PL11 |-> PL12, PL12 |-> PL13,
             PL13 |-> PL14, PL14 |-> PL15, PL15 |-> PL16, PL16 |-> PL17,
             PL17 |-> PL18, PL18 |-> PL19, PL19 |-> PL20, PL20 |-> PL20,
             PL30 |-> PL31, PL31 |-> PL32, PL33 |-> PL34, PL34 |-> PL35} &
             cl1 = closure1(nxt) &
             nrs = %x.(x:SIGNAL|card(cl1[{x}])) &
             mx = nrs~[{max(ran(nrs))}]
```

```
Execution time: 0.033897454 seconds
```

```
Out[97]: TRUE

         Solution:
                  cl1 = ∃197∈{(PL01↦PL02),(PL01↦PL03),...,(PL33↦PL35),(PL34↦PL35)}
                  mx = {PL01}
                  nxt = {(PL01↦PL02),(PL02↦PL03),(PL03↦PL04),(PL04↦PL05),(PL05↦PL06),(PL06↦
                  nrs = {(PL01↦19),(PL02↦18),(PL03↦17),(PL04↦16),(PL05↦15),(PL06↦14),(PL07
```