Formal Model-Based Constraint Solving and Document Generation

Michael Leuschel

Universität Düsseldorf
Institut für Informatik,
Universitätsstr. 1, D-40225 Düsseldorf
leuschel@cs.uni-duesseldorf.de

Abstract. Constraint solving technology for formal models has made considerable progress in the last years, and has lead to many applications such as animation of high-level specifications, test case generation, or symbolic model checking. In this article we discuss the idea to use formal models themselves to express constraint satisfaction problems and to embed formal models as executable artefacts at runtime. As part of our work, we have developed a document generation feature, whose output is derived from such executable models. This present article has been generated using this feature, and we use the feature to showcase the suitability of formal modelling to express and solve various constraint solving benchmark examples. We conclude with current limitations and open challenges of formal model-based constraint solving.

1 Animation and Constraint Solving for B

The B-Method [2] is a formal method rooted in predicate logic and set theory, supporting the generation of code "correct by construction" via successive refinement. Initially, the B-method was supported by two tools, BToolkit [4] and Atelier B [7], which both provided automatic and interactive proving environments, as well as code generators. To be able to apply the code generators, one has to *refine* an initial high-level specifications into lower-level B (called B0). It is of course vital that the initial high-level specification correctly covers the requirements of the application being developed. To some extent suitability of the high-level specification can be ensured by stating and proving invariants and assertions. In addition, the BToolkit provided an interactive animator, where the user had to provide values for parameters and existentially quantified variables, the validity of which was checked by the BToolkit prover. However, quite often these techniques are far from satisfactory and sufficient. The ProB validation tool [24, 25] was developed to satisfy this need in the tooling landscape, and provide a more convenient and extensive validation of high-level specifications. The first problem that ProB set out to solve was to provide automatic animation, freeing up the user from providing values for parameters and quantified variables. This was achieved by providing a constraint solver for the B language. On top of the animator, a model checker was developed, in order to automatically construct the state space of a formal B model and check temporal properties.

Constraint Solving, Execution and Proof

What distinguishes constraint solving from proof and execution (e.g., of generated code) in the context of B:

- the expression $\{2, 3, 5\} \cap 4..6$ can be executed, yielding the value $\{5\}$. The characteristics of execution for B are: no non-determinism arises, no search is required, and there is a clear procedure on how to obtain the result. An example for execution is the running of code generated from B0.
- The sequent or proof obligation $x \geq 0 \wedge n > 0 \vdash x + n > 0$ can be proven. The characteristics of proof for B are: usually a non-deterministic search for a proof is required; human intervention is also often required. Proof can deal with infinite values and infinitely many possibilities; e.g., the above sequent holds for infinitely many values for x and n. A proof attempt either yields a proof or it does not. In the latter case, we do not know the status of the proof obligation and in either case no values are obtained.
- The predicate $x \geq 0 \wedge n > 0 \wedge x + n \in \{2, 3\}$ can be solved yielding a solution $x = 0, n = 2$. The characteristics of constraint solving are that, in contrast to execution and just like for proof, a non deterministic search for possible solutions is required. In contrast to proof, the process is fully automatic and provides concrete values. On the downside, constraint solving usually can only deal with a bounded number of finite values for the variables.

Challenge The major challenge of animating or validating B is the expressiveness of its underlying language. B is based on predicate logic, augmented with arithmetic (over integers), (typed) set theory, as well as operators for relations, functions and sequences. (A similar point can be made for other formal methods who share a similar foundation, such as TLA+ [21] or Z [38].) As such, B provides a very expressive foundation which is familiar to many mathematicians and computer scientists. For example, Fermat's Last Theorem can be written in B as follows:

$$\forall n.(n > 2 \Rightarrow \neg(\exists(a, b, c).(a^n + b^n = c^n)))$$

In B's ASCII syntax (AMN or Abstract Machine Notation) this is written as follows:

```
!n.(n>2 => not(#(a,b,c).(a**n + b**n = c**n)))
```

A more typical example in an industrial formal specifications would be the integer square root function, which can be expressed in B as follows:

$$isqrt = \lambda n.(n \in \mathbb{N} | max(\{i | i^2 \leq n\}))$$

Here, the λ operator allows us to construct an infinite function, whose domain are the natural numbers and whose result is the largest integer whose square is less or equal to the function parameter n.

Due to arithmetic and the inclusion of higher-order functions, the satisfiability of B formulas is obviously undecidable. As such, animation is also undecidable, as operation preconditions or guards in high-level models can be arbitrarily complex. We cannot expect to be able to determine the truth value of Fermat's Last Theorem automatically, but ProB *is* capable of computing with the integer square root function above, e.g., determining that $isqrt(101) = 10$ or $isqrt(1234567890) = 35136$.[1] The relational composition operator ";" can actually be used as the higher-order "map" function in functional programming, and ProB can compute $([99, 100, 101]; isqrt) = [9, 10, 10]$.

In essence, the challenge and ultimate goal of ProB is to solve constraints, for an undecidable formal method with existential and universal quantification, higher-order functions and relations, unbounded variables. Ideally, infinite functions should be dealt with symbolically, while large finite relations should be stored efficiently. Moreover, we generally need not just to find one solution for a predicate, but all solutions. For example, when evaluating a set comprehension, all solutions must be found. Similarly, when using model checking we need to find all solutions for the guard predicates, to ensure that the complete state space gets constructed.

Applications of Constraint Solving

Over the years the constraint solving kernel of ProB has been improved, e.g., making use of the CLP(FD) library of SICStus Prolog [6] or using CHR [12]. This opened up many additional applications:

- Constraint-based invariant or deadlock checking [14].
  E.g., for deadlock checking, we ask the constraint solver to find a state of a B model satisfying the invariant, such that no event or operation is enabled.
- Model-based testing [34, 16, 31].
  Here we ask the constraint solver to find values for constants and operation parameters to construct test cases.
- Disproving and proving [17].
  Here we ask the constraint solver to find counter examples to proof obligations. Sometimes, when no counter example is found, the constraint solver can return a proof, e.g., when only finite domain variables occur.
- Enabling analysis [10].
  Here the constraint solver determines whether an event can disable or enable other events. The result is used for model comprehension, inferring control flow and for optimising the model checking process.
- Symbolic model checking [18].
  Here the constraint solver is used to find counter example traces for invariance properties.

[1] This is one of the specifications which is given as an example of a non-executable specification in [15].

2 Model-Based Constraint Solving

We now want to turn our focus from constraint solving technology for validating B models towards using B models to express constraint satisfaction problems.

The idea is to use the expressivity of the B language and logic to express practical problems, and to use constraint solving technology on these high level models. In other words, the B model is not refined in order to generate code but is "executed" directly.

Data validation in the railway domain [26, 27, 5, 22, 1] was a first practical application where B was used in this way, i.e., properties where expressed in B and checked directly by a tool such as ProB, PredicateB or Ovado. Here the B language was particularly well suited, e.g., to express reachability in railway networks. The constraint solving requirements are typically relatively limited and could still be solved by naive enumeration.

In the article [28] we later argued that B is well suited for expressing constraint satisfaction problems in other domains as well. This was illustrated on the Jobs puzzle challenge [37] and we are now using this approach at the University of Düsseldorf to solve various time tabling problems [35], e.g., determine whether a student can study a particular combination of course within a given timeframe.

A question is of course, why not encode these constraint satisfaction problems in a dedicated programming language such as CLP(FD) [6] or Zinc [29]. Some possible answers to this question are:

- By using B we obtain constraint programming with proof support B. For example, we can add assertions about our problem formulation and discharge them using proof. We also hope that optimisation rules can be written in B and proven for all possible values.
- B is a very expressive language, many problems can be encoded more elegantly in B than in other languages [28]
- we want to use a formal model not just as a design artefact but also at runtime; B can also be a very expressive query language, thereby enabling introspection, monitoring and analysis capabilities at runtime.
- We also wanted to stress test the constraint solver of ProB, identify weaknesses and improve the tool in the process.
- Finally, we hope to use B in this way for teaching mathematics, theoretical computer science and obviously B itself.

In the SlotTool project [35] we will compare the formal model based approach with a traditional constraint programming implementation, but it is still to early in the project to draw any conclusions.

In Section 4 we will present a few more constraint satisfaction benchmarks and problems which can be stated in the logic of the B notation. To this end, we will use another new feature of ProB: being able to generate "executable" Latex documentation. This feature was developed out of the necessity to understand complex models and complex situations in [35], as well as out of the need to

generate validation reports and summaries for data validation. This new feature is described in the following section.

3 Model-Based Document Generation

In this section we present a new feature of ProB, allowing one to generate readable documents from formal models. ProB can be used to process Latex [20] files, i.e., ProB scans a given "raw" Latex file and replaces certain ProB Latex commands by processed results, yielding a "proper" Latex file with all ProB commands replaced by evaluated results.

```
probcli FILE -init -latex RawLatex.tex FinalLatex.tex
```

The `FILE` and `-init` parameters are optional; they are required in case one wants to process the commands in the context of a certain model. Currently the ProB Latex commands mainly support B and Event-B models, TLA+ and Z models can also be processed but all commands currently expect B syntax. You can add more commands if you wish, e.g., set preferences using `-p PREF VAL` or run model checking `--model-check`. The Latex processing will take place after most other commands, such as model checking.

To some extent this feature was inspired by Z, where models are written in Latex format from the start. The Z Word Tools [13] were later developed to enable one to write Z models in Microsoft Word. A difference with our approach is that the B model is still kept separate from the Latex document, and that the Latex document may also contain commands to derive additional data, tables or figures. Moreover, multiple Latex documents can be attached to a B model and can also be re-used for the same model, with varying data inputs.

Applications We hope that some of the future applications of this Latex package are:

- **Model documentation**: generate an executable documentation for a formal model, that shows how to operate on the model. Moreover, provided ProB's Latex processing runs without errors, the documentation is guaranteed to be up-to-date with the current version of the model.
- **Worksheets**: for certain tasks the Latex document can replace a separate formal B model, the model is built-up incrementally by Latex commands and the are results shown in the final Latex output. This is probably most appropriate for smaller, isolated mathematical problems in teaching.
- **Validation reports**: on can automatically construct a summary of a validation task such as model checking or assertion checking.
- **Model debugging or information extraction**: here the processing of the executable document extracts and derives relevant information from a formal model, and presents it in a user friendly way. We use this feature regularly for our time tabling application [35] to depict conflicts either graphically or in a tabular fashion.

– Finally, we also plan to use the Latex package to produce documentation for some of ProB's features (such as this latex package or ProB's external functions).

Some Commands The `\probexpr` command takes a B expression as argument and evaluates it. By default it shows the B expression and the value of the expression, for example:

– `\probexpr{{1}\/{2**100}}` in the raw Latex file will yield:
$\{1\} \cup \{2^{100}\} = \{1, 1267650600228229401496703205376\}$

The `\probrepl` command takes a REPL command and executes it. By default it shows only the output of the execution, e.g., in case it is a predicate TRUE or FALSE.

– `\probrepl{2**10>1000}` in the raw Latex file will yield:
$TRUE$
– `\probrepl{let DOM = 1..3}` outputs a value and will define the variable `DOM` for the remainder of the Latex run:
$\{1, 2, 3\}$
– there is a special form for the let command: `\problet{DOM}{1..3}`, it has the same effect as the command above, but also prints out the let predicate itself:
$let\ DOM = 1..3 \rightsquigarrow \{1, 2, 3\}$

The `\probprint` command takes an expression or predicate and pretty prints it, for example:

– `\probprint{bool({1|->2,2|->3}|>>{4}:NATURAL+->INTEGER)}` yields:
$bool(\{(1 \mapsto 2), (2 \mapsto 3)\} \rhd \{4\} \in \mathbb{N} \nrightarrow \mathbb{Z})$

The `\probif` command takes an expression or predicate and two Latex texts. If the expression evaluates to TRUE the first branch is processed, otherwise the other one is processed. Here is an example:

– `\probif{2**10>1000}{$\top$}{$\bot$}` in the raw Latex file will yield:
$\top$

The `\probfor` command takes an identifier, a set expression and a Latex text, and processes the Latex text for every element of the set expression, setting the identifier to a value of the set. For example, below we embed the command: `\probfor{i}{2..3}{\item square of $\probexpr{i}$: $\probexpr{i*i}$}` within an itemize environment to generate a list of entries:

– square of $i = 2$: $i * i = 4$
– square of $i = 3$: $i * i = 9$

The \probtable command takes a B expression as argument, evaluates it and shows it as a table. For example, the command:
\probtable{{i,cube|i:2..3 & cube=i*i*i}}{no-row-numbers} in the raw Latex file will yield:

i	$cube$
2	8
3	27

Finally, the \probdot command takes a B expression or predicate as argument, evaluates it and translates it into a graph rendered by dot [3].

4 A Portfolio of Constraint Solving Examples in B

The following examples were generated (on $1/10/2016-11h383s$) using the Latex package described in Sect. 3 with ProB version $1.6.1 - beta4$.

4.1 Graph Colouring

The graph colouring problem consists in assigning colours to nodes of a graph, such that any two neighbours have different colours. Let us first define some arbitrary directed graph $gr = \{(1 \mapsto 3), (2 \mapsto 4), (3 \mapsto 5), (5 \mapsto 6)\}$ (using integers as nodes). Suppose we want to color this graph using the colours $cols = \{red, green\}$. We now simply set up a total function from nodes to $cols$ and require that neighbours in gr have a different colour:

$$\exists col.(col \in 1..6 \rightarrow cols \wedge \forall(x,y).(x \mapsto y \in gr \Rightarrow col(x) \neq col(y)))$$

The graph and the first solution found by ProB for col are shown in Fig. 1 using the \probdot command.

4.2 Graph Isomorphism

Let us define two directed graphs $g1 = \{(v1 \mapsto v2), (v1 \mapsto v3), (v2 \mapsto v3)\}$ and $g2 = \{(n1 \mapsto n2), (n3 \mapsto n1), (n3 \mapsto n2)\}$. The nodes of $g1$ are $V = \{v1, v2, v3\}$ and of $g2$ are $N = \{n1, n2, n3\}$. These two graphs are isomorphic if we can find a bijection between V and N, such that the successor relation is preserved. We can compute the successors of a node by using the relational image operator [.], e.g., the successors of $v1$ in $g1$ are $g1[\{v1\}] = \{v2, v3\}$. In B we can thus check $g1$ and $g2$ for isomporhism by trying to find a solution for:

$$\exists iso.(iso \in V \rightarrowtail\!\!\!\!\!\!\rightarrow N \wedge \forall v.(v \in V \Rightarrow iso[g1[\{v\}]] = g2[iso[\{v\}]]))$$

The graph and the first solution found by ProB for iso are shown in Fig. 2 using the \probdot command.

An industrial application of this constraint solving task — expressed in B — for reverse engineering can be found in [8].

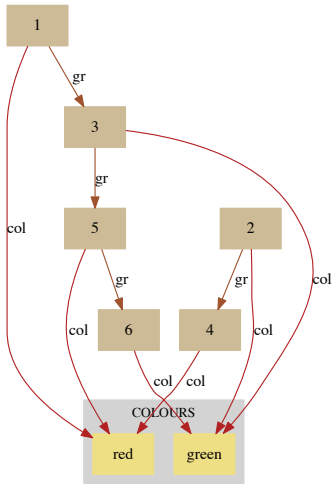


Fig. 1. A solution to a graph colouring problem

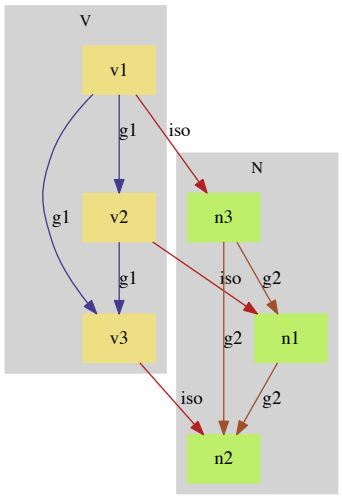


Fig. 2. A solution to a graph isomorphism problem

4.3 N-Queens and Bishops

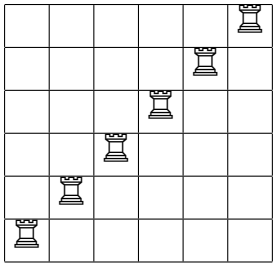
The N-Queens puzzle is a famous benchmark within constraint programming. The task is to place n queens on a $n \times n$ chessboard so that no two queens attack each other. Initially, we solve the puzzle for n=6.

In a first step, we place one queen on each row and column by using a total injection constraint:

$$\exists queens.(queens \in 1..n \rightarrowtail 1..n)$$

Here, *queens* is a function which for every queen stipulates the column it is placed on. By stipulating that the function is injective, we ensure that no two queens can be on the same column. By numbering queens from 1 to n, we have implicitly placed one queen on each row.

We still need to ensure that queens cannot attack each other on the diagonals, above we have actually described the N-Rook problem. The first solution found by ProB is shown below `\probfor` command and the skak package.[2]



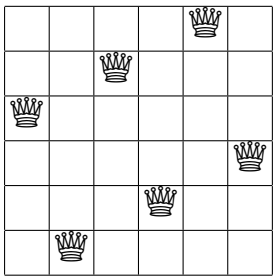
Dealing with the diagonals requires a more involved universal quantification:

$queens \in 1..n \rightarrowtail 1..n \wedge \forall(q1, q2).(q1 \in 1..n \wedge q2 \in 2..n \wedge q2 > q1 \Rightarrow queens(q1) + (q2 - q1) \neq queens(q2) \wedge queens(q1) + (q1 - q2) \neq queens(q2))$

The first solution found by ProB is

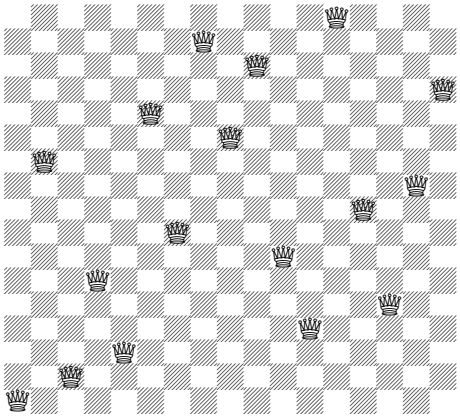
$$queens = \{(1 \mapsto 5), (2 \mapsto 3), (3 \mapsto 1), (4 \mapsto 6), (5 \mapsto 4), (6 \mapsto 2)\}$$

which can be depicted graphically as follows:



For n=17 we obtain the following first solution (after about 20 ms):

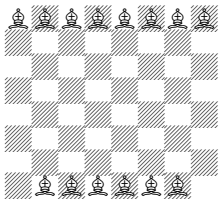
[2] See `https://www.ctan.org/pkg/skak`.



Related to the N-Queens puzzle is the Bishops problem: how many bishops can one place on an n by n chess board without any bishop attacking another bishop. In this case one can place multiple bishops on the same row and column; hence our encoding in B must be slightly different. Below we represent the placement of the bishops as a subset of $(1..n) \times (1..n)$ and solve the puzzle for n=8. The following constraint encodes the proper placement of the bishops:

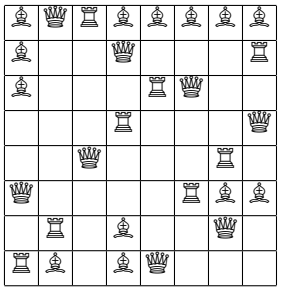
$\exists bshp.(bshp \subseteq (1..n) \times (1..n) \wedge \forall (i,j).(\{i,j\} \subseteq 1..n \Rightarrow (i \mapsto j \in bshp \Rightarrow \forall k.(k \in i+1..n \Rightarrow (k \mapsto (j+k)-i) \notin bshp \wedge (k \mapsto (j-k)+i) \notin bshp))))$

To find the optimal solution one can solve the above predicate with an additional constraints about the cardinality of *bshp*, and continuously use the size of the previous solution as a strict lower bound for the next solution. Below is solution of the above with 14 bishops (found in about half a second); there is no solution with 15 bishops.



We can also try to solve these various puzzles together, e.g., place 8 queens, 8 rooks and 13 bishops on the same eight by eight board. For this, we simply conjoin the four problems above and add constraints linking them, to ensure that a square is occupied by one piece at most. This is a simplified version of the crowded chess board problem from [11].

A solution found after about 0.4 seconds is shown below. Note, that while ProB can solve the problem quite efficiently for 13 bishops, solving time for the optimal 14 bishops together with 8 queens and rooks is dramatically higher (about 560 seconds). Here, a custom low-level encoding will probably be much more efficient than the B version (but also more tedious to write).



4.4 Golomb Ruler

Another well-known constraint solving benchmark is the Golomb ruler. The task is to set marks on a ruler of a given length so that no two marks have the same distance. The marks have to be put at integer positions and the ruler is also of integer length.

We now solve this puzzle for $n = 7$ marks and a length $len = 25$.

The following expresses the problem in B:

$\exists a.(a \in 1..n \to 0..len \wedge \forall i.(i \in 2..n \Rightarrow a(i-1) < a(i)) \wedge \forall (i1, j1, i2, j2).(i1 > 0 \wedge i2 > 0 \wedge j1 \leq n \wedge j2 \leq n \wedge i1 < j1 \wedge i2 < j2 \wedge (i1 \mapsto j1) \neq (i2 \mapsto j2) \Rightarrow a(j1) - a(i1) \neq a(j2) - a(i2)))$

The first solution found by ProB (in about 130 ms) is the following one:

$$a = \{(1 \mapsto 0), (2 \mapsto 2), (3 \mapsto 6), (4 \mapsto 9), (5 \mapsto 14), (6 \mapsto 24), (7 \mapsto 25)\}$$

The solution is depicted graphically below, using the `\probfor` command within a Latex picture environment.

It takes 320 ms to compute all 10 solutions using a set comprehension. Note that some of the rulers can be obtained from the other rules by reversing the order of the marks.

We can filter out these rulers using the B function $\lambda r.(r \in seq(\mathbb{Z}) | rev((r; \lambda i.(i \in \mathbb{Z} | 25 - i))))$ yielding the table below:

a
$[0, 2, 6, 9, 14, 24, 25]$
$[0, 1, 4, 10, 18, 23, 25]$
$[0, 2, 3, 10, 16, 21, 25]$
$[0, 1, 7, 11, 20, 23, 25]$
$[0, 3, 4, 12, 18, 23, 25]$

4.5 Sudoku and Latin Squares

Sudoku is a popular puzzle in constraint programming circles. We first define the domain for our numbers: $let\ D = 1..9$. Let us first construct a 9×9 square containing numbers in D, such that on all rows and columns we have different numbers, i.e., we just construct a **Latin square** of order 9.

We first compute the pairs of positions on columns that need to be different:

$$let\ Diff1 = \{x1, x2, y1, y2 | \{x1, x2, y1\} \subseteq D \wedge x1 < x2 \wedge y1 = y2\}$$

This gives rise to $card(Diff1) = 324$ pairs of positions. Now we do the the same for rows:

$$let\ Diff2 = \{x1, x2, y1, y2 | \{x1, y1, y2\} \subseteq D \wedge x1 = x2 \wedge y1 < y2\}$$

A solution to the constraint $\exists Board.(Board \in D \rightarrow (D \rightarrow D) \wedge \forall (x1, x2, y1, y2).(x1 \mapsto x2 \mapsto y1 \mapsto y2 \in Diff1 \cup Diff2 \Rightarrow Board(x1)(y1) \neq Board(x2)(y2)))$ is depicted below, again using the `\probfor` command:

2	5	4	1	3	6	7	8	9
1	4	2	3	6	5	9	7	8
4	9	3	2	1	8	5	6	7
3	1	8	7	9	2	4	5	6
6	3	9	8	7	1	2	4	5
5	2	1	9	8	7	6	3	4
7	6	5	4	2	9	8	1	3
8	7	6	5	4	3	1	9	2
9	8	7	6	5	4	3	2	1

Now we take into account difference constraints on the nine relevant 3×3 sub squares. We define a set containing three sets of indices:

$$let\ Sub = \{\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}\}$$

Observe that this is a set of sets. We can now compute the pairs of positions that need to be different within each sub square:

$let\ Diff3 = \{x1, x2, y1, y2 | x1 \geq x2 \wedge (x1 \mapsto y1) \neq (x2 \mapsto y2) \wedge \exists (s1, s2).(s1 \in Sub \wedge s2 \in Sub \wedge \{x1, x2\} \subseteq s1 \wedge \{y1, y2\} \subseteq s2)\}$

Observe that above we have quantified over sets (for $s1$ and $s2$). The constraint $x1 \geq x2$ is not strictly necessary; it just reduces the number of conflict positions to be checked. As a further improvement, one could add the additional symmetry breaking constraint that $x1 = x2 \Rightarrow y1 > y2$.

To conclude, we simply combine all position pairs into a single set:

$$let\ Diff = Diff1 \cup Diff2 \cup Diff3$$

To generate a valid Sudoku solution we now need to solve the following constraint:

$\exists Board.(Board \in D \rightarrow (D \rightarrow D) \wedge \forall(x1, x2, y1, y2).(x1 \mapsto x2 \mapsto y1 \mapsto y2 \in Diff \Rightarrow Board(x1)(y1) \neq Board(x2)(y2)))$

The first solution found in about 50 ms is shown below:

2	7	5	1	4	3	8	6	9
1	3	6	7	9	8	2	4	5
8	4	9	5	6	2	7	1	3
7	1	2	8	3	5	4	9	6
4	6	3	2	1	9	5	7	8
5	9	8	4	7	6	1	3	2
6	5	4	3	2	1	9	8	7
3	2	1	9	8	7	6	5	4
9	8	7	6	5	4	3	2	1

4.6 Coins Puzzle

This is a puzzle from chapter 7 of [32]. One interesting aspect is the use of an aggregate constraint (Σ) and the fact that decision variables are in principle unbounded.

The puzzle is as follows. A bank has various bags of money, each containing differing number of coins $coins = \{16, 17, 23, 24, 39, 40\}$. In total 100 coins are stolen; how many bags are stolen for each type of bag?

We can express this puzzle in B as the solution to the following predicate:

$$\exists stolen.(stolen \in coins \rightarrow \mathbb{N} \wedge \Sigma(x).(x \in coins | x * stolen(x)) = 100)$$

A solution found by ProB is: $stolen = \{(16 \mapsto 2), (17 \mapsto 4), (23 \mapsto 0), (24 \mapsto 0), (39 \mapsto 0), (40 \mapsto 0)\}$, also depicted as a table as follows:

coins	16	17	23	24	39	40
stolen	2	4	0	0	0	0

All solutions can be found by computing the following set comprehension:

$$\{s | s \in coins \rightarrow \mathbb{N} \wedge \Sigma(x).(x \in coins | x * s(x)) = 100\}$$

The solution computed by ProB contains just the single solution already shown above: $\{\{(16 \mapsto 2), (17 \mapsto 4), (23 \mapsto 0), (24 \mapsto 0), (39 \mapsto 0), (40 \mapsto 0)\}\}$. Observe that here the constraint solver needs to find all solutions the predicate inside the set comprehension. This is made more difficult by the fact that the range of the *coins* variable is not bounded explicitly, and only bounded implicitly by the summation constraint. The bounds on *coins* can only be inferred during the constraint solving process itself.

5 External Data Sources and Data Validation

The core B language does not provide any features for input and output. Moreover, the operations for data types such as strings are quite limited (only equality and inequality are provided). This has lead us to extend the B language via so-called **external functions**. Basically, these are B DEFINITIONS which get mapped to code in the ProB kernel. Some of these functions have been taken over and implemented by ClearSy in their PredicateB secondary toolchain. Here we briefly showcase these features, in particular in the context of data validation.

5.1 External Data Sources

ProB can read in XML and CSV files using various external functions. In this section we read in a CSV file called "elementdata.csv" containing data about chemical elements:

$$let\ data = READ\_CSV\_STRINGS(\text{"elementdata.csv"})$$

The read in data is of type $seq(STRING \mapsto STRING)$ and contains $size(data) = 118$ entries.

The first entry has $card(data(1)) = 20$ fields in total, for example the fields ("Atomic_Number" $\mapsto$ "1"), ("Atomic_Weight" $\mapsto$ "1.00794"), as well as the fields ("Name" $\mapsto$ "Hydrogen") or ("Symbol" $\mapsto$ "H").

Note that the external read function is generic: it works for any CSV file where the field names are stored in the first row; empty cells lead to undefined fields in the B data.

5.2 Data Validation Example

Data validation is one area where B's expressivity is very useful, and we illustrate this on the data we have read in above. We can check that the index in the data sequence correspond to the atomic number using the following predicate:

$$\forall i.(i \in dom(data) \Rightarrow i = STRING\_TO\_INT(data(i)(\text{"Atomic\_Number"})))$$

This property is $TRUE$. `STRING_TO_INT` is another external function, converting strings to integer. `DEC_STRING_TO_INT` is a variation thereof, also dealing with decimal numbers and expects a precision as argument. It is often useful for a user to define other auxiliary functions. In that respect, B is almost like a functional programming language:

$$let\ aw = \lambda i.(i \in dom(data) | DEC\_STRING\_TO\_INT(data(i)(\text{"Atomic\_Weight"}), 4))$$

The above function can now be applied, e.g., $aw(1) = 10079$.

We can check if the atomic weights are ordered by atomic number:

$$\forall(i,j).(i \in dom(aw) \wedge j \in dom(aw) \wedge i < j \Rightarrow aw(i) \leq aw(j)) \rightsquigarrow FALSE$$

Maybe surprisingly, this property has been evaluated to false. One counter example is $i = 18 \wedge j = 19 \wedge awi = 399480 \wedge awj = 390983 \wedge namei =$ "Argon" $\wedge$ $namej =$ "Potassium". All counter examples are shown in the table below:

$Element1$	$aw1$	$Element2$	$aw2$
"Argon"	"39.948"	"Potassium"	"39.0983"
"Cobalt"	"58.9332"	"Nickel"	"58.6934"
"Plutonium"	"244.0642"	"Americium"	"243.0614"
"Tellurium"	"127.6"	"Iodine"	"126.90447"
"Thorium"	"232.0381"	"Protactinium"	"231.03588"
"Uranium"	"238.0289"	"Neptunium"	"237.048"

In summary, in this section we have shown how to read in and manipulate data in B, how to validate properties in the data and how validation reports with counter example tables can be generated.

6 Discussion

Above we have shown the promises of using the B language to express constraint satisfaction problems. In practice, there are of course still limitations to this approach. The B approach will often engender a computational overhead compared to a direct encoding in a lower-level constraint programming language. Future research will try to minimise this overhead.

A crucial aspect of the constraint solving is the treatment of quantifiers and (nested) set comprehensions. ProB has techniques to expand quantifiers of bounded scope, or some special forms such as $\forall x.(x \in S \Rightarrow ...)$.[3] When these cannot be applied, the quantifiers will delay until all relevant variables are known: this can lead to performance degradations.

Debugging is another issue, which is problematic for constraint programming in general and B is no exception here. We have added external functions for debugging, e.g., to print values or observe how values are instantiated. ProB can now also provide performance warning messages, e.g., when universal or existential quantifiers cannot be dealt with efficiently.

Below we discuss some related approaches (and repeat some of the points made in the not easily accessible article [23]).

Comparison with non-constraint solving tools We have already discussed the proof-based BToolkit animator. A variety of other tools have been developed

[3] See, `https://www3.hhu.de/stups/prob/index.php/Tips:_Writing_Models_for_ProB` for more details.

for animating or model checking high-level specifications: Brama [36] and AnimB [30] for Event-B or TLC [40] for TLA$^+$. These tools rely on naive enumeration and can be used if the models are relatively concrete. However, there is little chance in using such tools for more challenging constraint solving tasks. For example, TLC takes hours to find an isomorphism for two graphs with 9 nodes (see [27]). TLC on the other hand can be very efficient for concrete models, where the overhead of constraint solving provides no practical advantage.

Comparison with other technologies In the past years we have also investigated a variety of alternative technologies to replace or complement the constraint solver of ProB: BDD-Datalog based approaches, SAT- and SMT-solving techniques. For SAT, we have implemented an alternative backend for first-order B in [33] using the Kodkod interface [39]. For certain complicated constraints, in particular those involving relational operators, this approach fared very well. The power of clause learning and intelligent backtracking are a distinct advantage here over classical constraint solvers. However, for arithmetic the SAT approach usually has problems scaling to larger integers.

Quite often, the SAT approach is better for inconsistent predicates, while the ProB constraint solver fared better when the predicates were satisfiable. Also, the SAT approach typically has problems dealing with large data and cannot deal with unbounded values or with infinite or higher-order functions. Here, an SMT-based approach could be more promising. We have also experimented with SMT-solvers, in particular a SMT-plugin for Event-B [9] and now also provide a Z3 backend for ProB[19]. For proof, SMT solving has proven very useful for B. In [18] have also used SMT to complement ProB for symbolic model checking. But for constraint solving, the results are thus far still rather disappointing.

In conclusion, constraint solving has provided the foundation for many novel tools and techniques to validate formal models. While SAT and SMT-based techniques also have played an increasingly important role in this area, constraint solving approaches have advantages when dealing with large data. In future, we are striving for an approach which can reconcile the advantages of all of these approaches.

Acknowledgements

I would like to thank all those people who have contributed towards the development of ProB and without whom the tool would not be where it is now: Jens Bendisposto, Michael Butler, Ivaylo Dobrikov, Marc Fontaine, Fabian Fritz, Dominik Hansen, Philipp Körner, Sebastian Krings, Lukas Ladenberger, Daniel Plagge, David Schneider, Corinna Spermann, and many more. I also thank Stefan Hallerstede for feedback and discussions about this article.

References

1. R. Abo and L. Voisin. Formal implementation of data validation for railway safety-related systems with OVADO. In S. Counsell and M. Núñez, editors, *Proceedings SEFM'2013 Collocated Workshops*, LNCS 8368, pages 221–236. Springer, 2013.
2. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
3. AT&T Labs-Research. Graphviz - open source graph drawing software. Obtainable at `http://www.research.att.com/sw/tools/graphviz/`.
4. B-Core (UK) Ltd, Oxon, UK. *B-Toolkit*. Available at `https://github.com/edwardcrichton/BToolkit`.
5. F. Badeau and M. Doche-Petit. Formal data validation with Event-B. *CoRR*, abs/1210.7039, 2012. Proceedings of DS-Event-B 2012, Kyoto.
6. M. Carlsson and G. Ottosson. An Open-Ended Finite Domain Constraint Solver. In H. G. Glaser, P. H. Hartel, and H. Kuchen, editors, *Proceedings PLILP'97*, LNCS 1292, pages 191–206. Springer-Verlag, 1997.
7. ClearSy. *Atelier B, User and Reference Manuals*. Aix-en-Provence, France, 2009. Available at `http://www.atelierb.eu/`.
8. ClearSy. Data Validation & Reverse Engineering. June 2013. Available at `http://www.data-validation.fr/data-validation-reverse-engineering/`.
9. D. Deharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In *Proceedings ABZ'2012*, LNCS 7316, pages 194–207. Springer, 2012.
10. I. Dobrikov and M. Leuschel. Enabling analysis for event-b. In *Proceedings ABZ 2016*, volume 9675 of *LNCS*. Springer, 2016.
11. H. E. Dudeney. *Amusements in Mathematics*. 1917. Available at `https://www.gutenberg.org/ebooks/16713`.
12. T. Frhwirth. *Constraint Handling Rules*. Cambridge University Press, August 2009.
13. A. Hall. Integrating Z into large projects tools and techniques. In E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, editors, *Proceedings ABZ'2008*, LNCS 5238, page 337. Springer-Verlag, 2008.
14. S. Hallerstede and M. Leuschel. Constraint-based deadlock checking of high-level specifications. *TPLP*, 11(4–5):767–782, 2011.
15. I. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Softw. Eng. J.*, 4(6):330–338, Nov. 1989.
16. A. Idani and Y. Ledru. B for modeling secure information systems - the b4msecure platform. In M. Butler, S. Conchon, and F. Zaïdi, editors, *Proceedings ICFEM 2015*, LNCS 9407, pages 312–318. Springer, 2015.
17. S. Krings, J. Bendisposto, and M. Leuschel. From Failure to Proof: The ProB Disprover for B and Event-B. In *Proceedings SEFM'2015*, LNCS 9276, pages 199–214. Springer, 2015.
18. S. Krings and M. Leuschel. Proof assisted symbolic model checking for b and event-b. In *Proceedings ABZ 2016*, LNCS 9675, pages 136–150. Springer, 2016.
19. S. Krings and M. Leuschel. SMT solvers for validation of B and Event-B models. In *Proceedings iFM 2016*, LNCS 9861, pages 361–375. Springer, 2016.
20. L. Lamport. *Latex: A Document Preparation System*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
21. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
22. T. Lecomte, L. Burdy, and M. Leuschel. Formally checking large data sets in the railways. *CoRR*, abs/1210.6815, 2012. Proceedings of DS-Event-B 2012, Kyoto.

23. M. Leuschel, J. Bendisposto, I. Dobrikov, S. Krings, and D. Plagge. From animation to data validation: The ProB constraint solver 10 years on. In J.-L. Boulanger, editor, *Formal Methods Applied to Complex Systems: Implementation of the B Method*, chapter Chapter 14, pages 427–446. Wiley ISTE, Hoboken, NJ, 2014.
24. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
25. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
26. M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale B models. In A. Cavalcanti and D. Dams, editors, *Proceedings FM 2009*, LNCS 5850, pages 708–723. Springer-Verlag, 2009.
27. M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale B models with ProB. *Formal Asp. Comput.*, 23(6):683–709, 2011.
28. M. Leuschel and D. Schneider. Towards B as a high-level constraint modelling language. In Y. Ait Ameur and K.-D. Schewe, editors, *Proceedings ABZ'2014*, LNCS 8477, pages 101–116. Springer Berlin Heidelberg, 2014.
29. K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. G. de la Banda, and M. Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.
30. C. Métayer. *AnimB: Animator of B system model in the Rodin platform*, 2010. Available at `http://wiki.event-b.org/index.php/AnimB`.
31. A. M. Moreira, C. Hentz, D. Déharbe, E. C. B. de Matos, J. B. S. Neto, and V. de Medeiros Jr. Verifying code generation tools for the b-method using tests: A case study. In J. C. Blanchette and N. Kosmatov, editors, *Proceedings TAP'2015*, LNCS 9154, pages 76–91. Springer, 2015.
32. K. G. Murty. *Optimization Models For Decision Making: Volume 1*. 2005. Available at `http://www-personal.umich.edu//~murty/books/opti_model/`.
33. D. Plagge and M. Leuschel. Validating B, Z and TLA+ using ProB and Kodkod. In D. Giannakopoulou and D. Méry, editors, *Proceedings FM'2012*, LNCS 7436, pages 372–386. Springer, 2012.
34. A. Savary, M. Frappier, and M. Leuschel. Model-based robustness testing in Event-B using mutation. In R. Calinescu and B. Rumpe, editors, *Proceedings SEFM'15*, LNCS 9276, pages 132–147, 2015.
35. D. Schneider, M. Leuschel, and T. Witt. Model-Based Problem Solving for University Timetable Validation and Improvement. In *Proceedings FM'2015*, LNCS 9109, pages 487–495. Springer-Verlag, 2015.
36. T. Servat. Brama: A new graphic animation tool for B models. In J. Julliand and O. Kouchnarenko, editors, *Proceedings B'2007*, LNCS 4355, pages 274–276. Springer-Verlag, 2007.
37. S. C. Shapiro. The Jobs Puzzle: A Challenge for Logical Expressibility and Automated Reasoning. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, 2011.
38. J. M. Spivey. *The Z Notation: a reference manual*. Prentice-Hall, 1992.
39. E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *Proceedings TACAS'07*, LNCS 4424, pages 632–647. Springer-Verlag, 2007.
40. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA^+ specifications. In L. Pierre and T. Kropf, editors, *Proceedings CHARME'99*, LNCS 1703, pages 54–66. Springer-Verlag, 1999.