Towards B as a High-Level Constraint Modelling Language

Solving The Jobs Puzzle Challenge

Michael Leuschel and David Schneider

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

Abstract. We argue that B is a good language to conveniently express a wide range of constraint satisfaction problems. We also show that some problems can be solved quite effectively by the ProB tool. We illustrate our claim on several examples, such as the *jobs puzzle* - for which we solve the challenge set out by Shapiro. Here we show that the B formalization is both very close to the natural language specification and can still be solved efficiently by ProB. Our approach is particularly interesting when a high assurance of correctness is required. Indeed, compared to other existing approaches and tools, validation and double checking of solutions is available for ProB and formal proof can be applied to establish important properties or provide an unambiguous semantics to the problem specification.

Keywords: B-method, constraint programming, logic programming, Alloy, Kodkod, optimization

1 Introduction

The B-method [1] is a formal method for specifying safety critical systems, reasoning about those systems and generating code that is correct by construction. The B-Language, part of this method, is a rich, mathematical language based on abstract machines and built around the concepts of first order logic, higher-order relations and set theory. Due to its expressiveness the B-Language allows its users to formalize and express complex problems in a succinct and elegant way on a high level of abstraction.

Initially, the B-method was supported by two tools, BToolkit and Atelier B, which both provided mainly automatic and interactive proving environments, as well as code generators. Later, the ProB validation tool provided automatic animation and model checking. Due to the characteristics of B, ProB gradually evolved into a constraint solving tool for the B language, in order to automatically determine values for parameters and quantified variables. This opens up

** Part of this research has been sponsored by the EU FP7 project 287563 (ADVANCE).

new uses of B beyond developing safety critical systems. Indeed, in this paper we want to show that B is a language well suited to express general constraint satisfaction problems. Moreover, in combination with the constraint solving capabilities of ProB, we can solve a non-trivial class of these problems.

First we present a case study which highlights the expressiveness of B as a constraint modelling language. The case study is based on the *jobs puzzle* [26] and takes on the challenges identified and discussed by Shapiro [21]. One of these challenges is to provide a formalization of the puzzle that follows closely the English text of the puzzle. This aspect makes it particularly interesting, as it allows us to showcase how, using B, these kinds of problems can be expressed very conveniently and still be solved efficiently with ProB. The puzzle, the challenge and our solution to the puzzle are discussed and compared to other solutions in Section 2.

Later in the paper, we establish that ProB as a tool can be used to solve an interesting class of constraint satisfaction problems efficiently, providing a good balance between the ease of expressing a problem on an abstract level using B and the efficiency of solving these problems. In Section 3 we discuss a series of problems that are expressed nicely in B and can be solved by ProB for a wide range of values, such as the *N-Queens* problem, the *Peaceable Armies of Queens* and the *Graph Isomorphism* problem. We compare the results to a selection of different tools to show that ProB gives competitive results, while still having room for improvement as discussed in Section 5. Finally, in Section 4 we present how the constraint solving features of ProB, showcased in this article, are being used in several industrial applications.

Alternate Approaches to Constraint Solving The mathematical language of B is quite close to that of Z and TLA$^+$. As ProB can deal with those formalisms [17, 8], the gist of the paper is also valid for those languages. Similarly, VDM and abstract state machines are probably also well suited to express constraint satisfaction problems.

Dedicated Constraint Solving Libraries Alternate approaches to our high-level formal methods approach are dedicated constraint-solving libraries embedded in general purpose programming languages. Examples are the CLP(FD) library of SICStus Prolog [4] or the ILOG solver. These libraries require a much higher modelling effort and a relatively high level of expertise, but can obviously obtain better performance. Another possible approach is the Zinc modelling language [14]. It provides a higher level encoding than for example CLP(FD), but still cannot deal with higher-order sets or relations. Also, to our knowledge, neither Zinc nor any other tool we are aware of can deal with unbounded constraint satisfaction problems.

SMT-based approaches It would be interesting to see how an expert in the Formula language [10], which maps to the Z3 SMT solver, would encode the problems in this paper, and how the solving times compare with those of ProB. Recently, an Event-B to SMT-LIB converter has become available for the Rodin platform [6]. It is very useful for proof, but as shown in [18] not suitable for constraint solving. For example, it was not possible to solve various simpler

problems, such as the *Who killed Agatha* puzzle or a graph colouring problem. As such, we did not attempt to use this B to SMT converter on the examples in this paper.

SAT-based approaches The Alloy language [9] was designed from the outset to be able to effectively translated to SAT problems. This leads to certain expressivity restrictions, e.g., higher-order relations and sets are not allowed as their SAT encoding would become too large to be tractable. ProB follows another principle: it accepts the B language in full with all its consequences, and tries to solve constraints for as many relevant models as possible. Note that ProB also has a backend [18] which translates B constraints into SAT. This uses the same Kodkod library [25] that Alloy employs, and can deal much better with certain relational constraints, but similarly only translates first order sets and relations (the rest are left for the traditional ProB solver). In this paper we discuss and compare several B solutions with Alloy counterparts and also discuss the ProB Kodkod backend.

Finally, one could think of using model checking rather than constraint solving. In fact, we have experimented with various solutions for the puzzles using efficient model checkers such as Spin or TLC. However, for constraint satisfaction problems model checking amounts to naive, brute force search and is rarely able to solve more complicated constraints.

2 On The Expressiveness of B - The Jobs Puzzle

The first and most detailed example we discuss is the *jobs puzzle*. This puzzle was originally published in 1984 by Wos et al. [26] as part of a collection of puzzles for automatic reasoners. A reference implementation of the puzzle, by one of the authors of the book, using OTTER [15] can be found online.[1]

The puzzle consists of eight statements that describe the problem domain and provide some constraints on the elements of the domain. The problem is about a set of people and a set of jobs; the question posed by the puzzle is: who holds which job? The text of the puzzle as presented in [21] is as follows:

- There are four people: Roberta, Thelma, Steve, and Pete.
- Among them, they hold eight different jobs.
- Each holds exactly two jobs.
- The jobs are: chef, guard, nurse, clerk, police officer (gender not implied), teacher, actor, and boxer.
- The job of nurse is held by a male.
- The husband of the chef is the clerk.
- Roberta is not a boxer.
- Pete has no education past the ninth grade.
- Roberta, the chef, and the police officer went golfing together.

[1] `http://www.mcs.anl.gov/~wos/mathproblems/jobs.txt`

What makes this puzzle interesting for automatic reasoners, is that not all the information required to solve the puzzle is provided explicitly in the text. The puzzle can only be solved if certain implicit assumptions about the world are taken into account, such as: the names in the puzzle denote gender or that some of the job names imply the gender of the person that holds it.

2.1 Shapiro's Challenge

Shapiro [21], following the original authors' remarks, that formalizing the puzzle was at times hard and tedious, identified three challenges posed by the puzzle with regard to automatic reasoners. According to Shapiro [21], the challenges posed by the *jobs puzzle* are to:

- formalize it in a non-difficult, non-tedious way
- formalize it in a way that adheres closely to the English statement of the puzzle
- have an automated general-purpose commonsense reasoner that can accept that formalization and solve the puzzle quickly.

Any formalization also needs to encode the implicit knowledge used to solve the puzzle for the automatic reasoners while still trying to satisfy the aspects mentioned above. Addressing this challenge makes this puzzle a good case-study for the expressiveness of B to formalize such a problem.

2.2 A Solution to the Jobs Puzzle using B

The B encoding of the puzzle uses plain predicate logic, combined with set theory and arithmetic. We will show how this enables a very concise encoding of the problem, staying very close to the natural language requirements. Moreover, the puzzle can be quickly solved using the constraint solving capabilities of ProB. Following the order of the sentences in the puzzle we will discuss one or more possibilities to formalize them using B.

To express "*There are four people: Roberta, Thelma, Steve, and Pete*" we define a set of people, that holds the list of names:

```
PEOPLE={"Roberta", "Thelma", "Steve", "Pete"}
```

We are using strings here to describe the elements of the set. This has the advantage, that the elements of the set are implicitly different.[2] Alternatively, we could use enumerated or deferred sets defined in the `SETS` section of a B machine.

As stated above we need some additional information that is not included in the puzzle to solve it. The first bit of information is that the names used in the puzzle imply the gender. In order to express this information we create two

[2] This encoding allows us to input the puzzle directly into the ProB console (available at `http://stups.hhu.de/ProB/index.php5/ProB_Logic_Calculator`).

sets, MALE and FEMALE which are subsets of PEOPLE and contain the corresponding names.

```
FEMALE={"Roberta", "Thelma"} & MALE={"Steve", "Pete"}
```

The next statement of the puzzle is: "*among them, they hold eight different jobs*". This can be formalized in B using a function that maps from a job to the corresponding person that holds this job using a total surjection from JOBS to PEOPLE:

```
HoldsJob : JOBS -->> PEOPLE
```

Although redundant, as we will see below, to express "*Among them, they hold eight different jobs*" we can add the assertion that the cardinality of HoldsJob is 8. This is possible, because in B functions and relations can be treated as sets of pairs, where each pair consists of an element of the domain and the corresponding element from the range of the relation.

```
card(HoldsJob) = 8
```

Constraining the jobs each person holds, the puzzle states: "*Each holds exactly two jobs*". To express this we use the inverse relation of HoldsJob, it maps a PERSON to the JOBS associated to her. The inverse function or relation is expressed in B using the ~ operator. For readability we assign the inverse of HoldsJob to a variable called JobsOf. JobsOf is in this case is a relation, because, as stated above, each person holds two jobs.

```
JobsOf = HoldsJob~
```

Because JobsOf is a relation and not a function, in order to read the values, we need to use B's relational image operator. This operator maps a subset of the domain to a subset of the range, instead of a single value. To read the jobs Steve holds, the relational image of JobsOf is used as shown below:

```
JobsOf[{"Steve"}]
```

Using the JobsOf relation we can express the third sentence of the puzzle using a universally quantified expression over the set PEOPLE. The Universal quantification operator (∀) is expressed in B using the ! symbol followed by the name of the variable that is quantified. This way of expressing the constraint is close to the original text of the puzzle, saying that the set of jobs each person holds has a cardinality of two.

```
!x.(x : PEOPLE => card(JobsOf[{x}]) = 2)
```

The fourth sentence assigns the set of job names to the identifier JOBS. This statement also constraints the cardinality of HoldsJob to 8.

```
JOBS = {"chef", "guard", "nurse", "clerk", "police", "teacher", "actor", "boxer"}
```

The following statements further constrain the solution. First "*The job of nurse is held by a male*", which we can express using the `HoldsJob` function and the set `MALE` by stating that the element of `PEOPLE` that `HoldsJob("nurse")` points to is also an element of the set `MALE`.

```
HoldsJob("nurse") : MALE
```

Additionally, we add the next bit of implicit information, which is that typically a distinction is made between actress and actor, and therefore the job name actor implies that it is held by a male. This information is formalized, similarly as above.

```
HoldsJob("actor") : MALE
```

The next sentence: "*The husband of the chef is the clerk*" contains two relevant bits of information, based on another implicit assumption, which is that marriage usually is between one female and one male. With this in mind, we know that the chef is female and the clerk is male. One possibility is to do the inference step manually and encode this as:

```
HoldsJob("chef") : FEMALE & HoldsJob("clerk") : MALE
```

Alternatively, and in order to stay closer to the text of the puzzle we can add a function `Husband` that maps from the set `FEMALE` to the set `MALE` as a partial injection. We use a partial function, because we do not assume that all elements of `FEMALE` map to an element of `MALE`.

```
Husband : FEMALE >+> MALE
```

To add the constraint using this function we state that the tuple of the person that holds the job as chef and the person that holds the job as clerk are an element of this function when treated as a set of tuples.

```
(HoldsJob("chef"), HoldsJob("clerk")) : Husband
```

The next piece of information is that "*Roberta is not a boxer*". Using the `JobsOf` relation we can express this close to the original sentence, by stating: boxer is not one of Roberta's jobs. This can be expressed using the relational image of the `JobsOf` relation:

```
"boxer" /: JobsOf[{"Roberta"}]
```

The next sentence provides the information that "*Pete has no education past the ninth grade*". This again needs some contextual information to be useful in order to find a solution for the puzzle [21]. To interpret this sentence we need to know that the jobs of police officer, teacher and nurse require an education of more than 9 years. Hence the information we get is that Pete does not hold any of these jobs. Doing this inference step we could, as above, state something along the lines of `HoldsJob("police") /= "Pete"`, etc. for each of the jobs. The

solution used here, tries to avoid doing the manual inference step. Although we still need to provide the information needed to draw the conclusion that Pete does not hold any of these three jobs. We create a set of those jobs that need higher education:

```
QualifiedJobs = {"police", "teacher", "nurse"}
```

Using the relational image operator we can now say that Pete is not among the ones that hold any of these jobs. The relational image can be used to get the set of items in the range of function or relation for all elements of a subset of the domain.

```
"Pete" /: HoldsJob[QualifiedJobs]
```

Finally, the last piece of information is that "*Roberta, the chef, and the police officer went golfing together*", from this we can infer that Roberta, the chef, and the police officer are all different persons. We write this in B stating that the set of Roberta, the person that holds the job as chef, and the person that is the police officer has cardinality 3, using a variable for the set for readability.

```
Golfers = {"Roberta", HoldsJob("chef"), HoldsJob("police")} & card(Golfers) = 3
```

By building the conjunction of all these statements, ProB searches for a valid assignment to the variables introduced that satisfies all constraints, generating a valid solution that answers the question posed by the puzzle "*who holds which job?*" in form of the `HoldsJob` function. The solution found by ProB is depicted in Fig. 1.[3]

This satisfies, in our eyes, the challenges identified by Shapiro. In the sense that the formalization, is not difficult, although it uses a formal language. The elements of this language are familiar to most programmers or mathematicians and it builds upon well understood and widely known concepts. The brevity of the solution shows that using an expressive high-level language it is possible to encode the puzzle without having tedious tasks in order to be able to solve the puzzle at all.

The encoding of the sentences follows the structure of the English statements very closely. We avoid the use of quantification wherever possible and use set based expressions that relate closely to the puzzle. We are able to encode the additional knowledge needed to solve puzzle in a straight forward way, that is also close to how this would be expressed as statements in English. Lastly it is worth to note that the formalization of "*Each holds exactly two jobs*" is the one furthest away from the English expression, using quantifications and set cardinality expressions.

[3] We used the "Visualize State as Graph" command and then adapted the generated graph using OmniGraffle

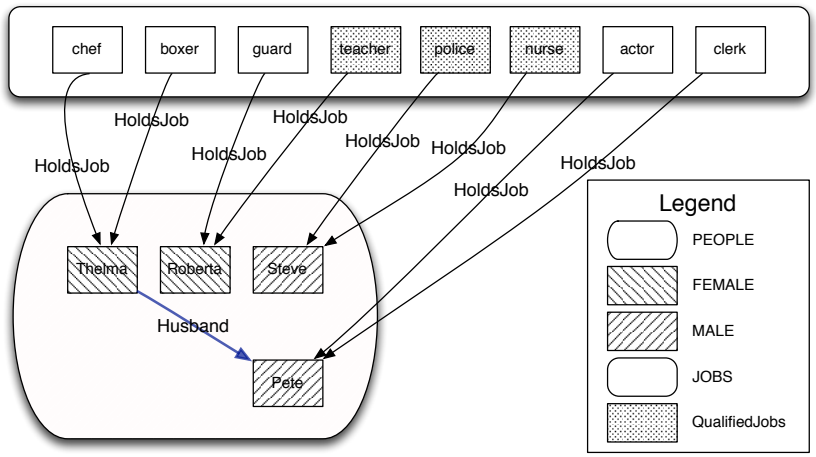

Fig. 1. The solution to the Jobs puzzle, depicted graphically

2.3 Related Work

In his paper Shapiro discusses several formalizations of the puzzle with regard to the identified challenges. A further formalization using controlled natural language and answer set programming (ASP) was presented in [19] by Schwitter et al.

The first of the solutions discussed by Shapiro is a solution from the TPTP website, encoded as a set of clauses and translated to FOL. The main disadvantages of this encoding is that it requires 64 clauses to encode the problem and many of them are needed to define equality among jobs and names. This is in contrast to our B encoding using either enumerated sets or strings, where all elements are implicitly assumed to be different. Thus the user does not have to define the concept of equality for simple atoms.

The second solution discussed by Shapiro uses SNePS [22], a common sense and natural language reasoning system designed with the goal to "have a formal logical language that captured the expressibility of the English language" [21]. The language has a unique name assumption and set arguments making the encoding simpler and less tedious. On the other hand the lack of support for modus tolens requires rewriting some of the statements in order to solve the puzzle.

The last formalization discussed by Shapiro uses Lparse and Smodles [16] which uses stable model semantics with an extended logic programming syntax. According to Shapiro several features of Lparse/Smodels are simmilar to those of SNePS. This formalization also simplifies the encoding of the puzzle, but according to Schwitter et al. both solutions still present a "considerable conceptual gap between the formal notations and the English statements of the puzzle" [19].

Schwitter et al. in their paper "The jobs puzzle: Taking on the challenge via controlled natural language processing" [19] present a solution to the *jobs puzzle* using controlled natural language and a translation to ASP to solve the *jobs puzzle* in a novel way that stays very close to the English statements of the puzzle and satisfying the challenges posed by Shapiro. To avoid the mismatch between natural and controlled natural languages Schwitter et al. describe the use of a development environment that supports the user to input valid statements according to the rules of the controlled language. A solution using a mathematical, but high level language like B avoids this problems by having a formal and, for most, familiar language used to formalize the problem.

3 Solving Constraint Problems with B and ProB

ProB is able to solve our formalization of the *jobs puzzle*, presented in the previous Section in about $10ms$, finding the two possible instantiations of the variables that represent the (same) solution to the puzzle.

In this section we will present four examples, that can be elegantly expressed with B and discuss how these can be solved with the constraint solving features of ProB.[4] There are many more examples, which unfortunately we cannot present here due to space restrictions.

3.1 Subset Sum

The first example is the subset sum problem 7.8.1 from page 340 of "Optimization Models For Decision Making: Volume 1".[5] Expressing this problem just takes this one line of B and ProB can solve it in less than 5 ms:

```
coins = {16,17,23,24,39,40} & stolen : coins --> NATURAL & SIGMA(x).(x:coins|stolen(x)*x)=100
```

The goal is to determine how many bags of coins were lost to amount for 100 missing coins. The bags can have different sizes, specified in the set named `coins`. In order to find the result with B, we create a function that maps from the different coin-bag sizes to a number; this number represents how many bags of that size were stolen. The instantiation of the function `stolen` is constrained by the last expression, which states that the sum of all coins in the missing bags is 100. This is expressed using the `SIGMA` operator, which returns the sum of all values calculated in the associated expression, akin the mathematical Σ operator. An interesting aspect is that we have not explicitly expressed an upper bound on `coins` (`NATURAL` stands for the set of mathematical natural numbers); ProB determines the upper bound itself during the constraint solving process. Finally, we can check that there is only one solution by checking:

```
card({c,s|c = {16,17,23,24,39,40} & s : c --> NATURAL & SIGMA(x).(x:c|s(x)*x)=100})=1
```

[4] The source code of the examples can be obtained at the following web site: `http://stups.hhu.de/w/Pub:Constraint_Solving_in_B`

[5] `http://ioe.engin.umich.edu/people/fac/books/murty/opti_model/`

3.2 N-Queens

The well known *N-Queens* problem[6] is a further problem, that can be expressed very succinctly in B by specifying a constant `queens`, which has to satisfy the following axioms:

```
queens : 1..n >-> 1..n & !(q1,q2).(q1:1..n & q2:2..n & q2>q1
   => queens(q1)+(q2-q1) /= queens(q2) & queens(q1)+(q1-q2) /= queens(q2))
```

The total and injective function `queens` maps the index of each column to the index of the row where the queen is placed in that column. The formula states that for each pair of columns, the queens placed on those columns are not on the same diagonal. From the set of functions from $1..n$ to $1..n$ ProB discards those candidates that violate the condition on the diagonals and instantiates queens to the first solution that satisfies it.

To get a better impression of how ProB performs for this and hopefully similar problems we compared, as shown in Table 1, the B implementation on ProB 1.3.7-beta9 to a C iterative implementation, a version in Prolog using CLP(FD)[7] running on SWI-Prolog 6.6.4, a version in Prolog taken from [24] Chapter 14[8] , a version written in Alloy using the Minisat and Sat4J SAT-solvers of Alloy 4.2. We ran the examples on a MacBook Pro with a four core Intel i7 Processor with 2.2 Ghz and 8 Gb. RAM for increasing values of n as reported below. All reported times represent the time needed to find a first valid configuration. For each tool we stopped collecting data after the first computation that took more than 10000 ms. to find a solution.

Table 1. Time in *ms.* to find a first solution to the *N-Queens* problem.

n	ProB	C	swipl	swipl CLP(FD)	Alloy Sat4J	Alloy Minisat
8	1	4	1	7	75	81
10	1	4	1	6	190	120
20	28	34	4941	30	3483	10019
30	67	11467	-[9]	49	6296	-
40	122	-	-	77	57992	-
50	250	-	-	483	-	-
60	349	-	-	479	-	-
70	426	-	-	178	-	-
80	623	-	-	278	-	-
90	885	-	-	759	-	-
100	1028	-	-	442	-	-

Table 2. LoC for the different solutions compared

Language	LoC
B (ProB)	2
Alloy	21
Prolog	60
swipl CLP(FD)	122
C	171

There are some pathological cases, not shown here, where ProB, but also other tools perform very badly (such as $n = 88$) but for most inputs of $n < 101$ ProB finds a solution in up to 1.2 seconds.

[6] `http://en.wikipedia.org/w/index.php?oldid=587668943`
[7] `http://www.logic.at/prolog/queens/queens.pl`
[8] `http://bach.istc.kobe-u.ac.jp/llp/bench/queen_p.pl`
[9] We canceled this run after 40 minutes without result.

The results show that constraint based solutions to this problem, with increasing board sizes give better results than the brute force versions. Among the constraint based results, the direct encoding in Prolog using CLP(FD) is generally faster than ProB; considering the higher abstraction level of B these results are to be expected. Taking the size of the implementations into account (as reported in Table 2) gives evidence that using B to encode such a problem and ProB to solve it is a good trade-off between the size, the complexity of the implementation and the time required to find a solution.[10]

3.3 Peaceable Armies of Queens

A challenging constraint satisfaction problem, related to the previous, was proposed by Bosch (Optima 1999) and taken up in [23]. It consists of setting up opposing armies of queens of the same size on a $n \times n$ chessboard so that no queen attacks a queen of the opposing colour.

Smith et. al [23] report that the integer linear programming tool CPLEX took 4 hours to find an optimal solution for $n = 8$ (which is 9 black and 9 white queens). Optimal here means placing as many queens as possible, i.e., there is a solution for 9 queens but none for 10 queens. In order to determine the optimal solution for $n = 8$ the ECLiPSe and the ILOG solver are reported to take just over 58 minutes and 27 minutes and 40 seconds respectively (Table 1 in [23]). After applying the new symmetry reduction techniques proposed in [23], the solving time was reduced to 10 minutes and 40 seconds for ECLiPSe and 5 minutes 31 seconds for the ILOG solver.

In a first instance, we have encoded the puzzle as a constraint satisfaction problem, i.e., determining whether for a given board size n and a given number of queens q we can find a correct placement of the queens. We first introduce the following `DEFINITION`:

```
ORDERED(c,r) == (!i.(i:1..(q-1) => c(i) <= c(i+1)) &
                 !i.(i:1..(q-1) => (c(i)=c(i+1) => r(i) < r(i+1))))
```

The encoding of the problem is now relatively straightforward:

```
blackc : 1..q --> 1..n &  blackr : 1..q --> 1..n &
whitec : 1..q --> 1..n &  whiter : 1..q --> 1..n &
ORDERED(blackc,blackr) & ORDERED(whitec,whiter) &

!(i,j).(i:1..q & j:1..q => blackc(i) /= whitec(j)) &
!(i,j).(i:1..q & j:1..q => blackr(i) /= whiter(j)) &
!(i,j).(i:1..q & j:1..q => blackr(i) /= whiter(j)+(blackc(i)-whitec(j))) &
!(i,j).(i:1..q & j:1..q => blackr(i) /= whiter(j)-(blackc(i)-whitec(j))) &

 whitec(1) < blackc(1) /* simple symmetry breaking */
```

The solving time on a MacBook Pro with a four core Intel i7 Processor with 2.2 Ghz and 8 Gb. RAM is as follows using ProB 1.3.7-beta9:

[10] We are aware that the different solutions compared might not represent the best possible solution in each formalism.

Table 3. Runtime in seconds to solve the *Peaceable Queens* problem

Board Size n	Queens q	Result	ProB	Alloy Minisat	Alloy Sat4J
7	7	sat	0.29 sec	3.04 sec	14.90 sec
7	8	unsat	17.6 sec	-	-
8	1	sat	0.00 sec	0.01 sec	0.02 sec
8	2	sat	0.00 sec	0.02 sec	0.03 sec
8	3	sat	0.01 sec	0.03 sec	0.05 sec
8	4	sat	0.01 sec	0.09 sec	0.17 sec
8	5	sat	0.02 sec	0.21 sec	0.45 sec
8	6	sat	0.02 sec	0.22 sec	1.29 sec
8	7	sat	0.06 sec	1.29 sec	2.10 sec
8	8	sat	0.51 sec	2.58 sec	6.68 sec
8	9	sat	12.10 sec	122.87 sec	106.74 sec
8	10	unsat	661 sec	-	-

We have also encoded the problem in Alloy. Table 3 shows the results for board sizes of 7 and 8 for ProB and Alloy. We have used Alloy 4.2. For a board size of 8 and 9 queens of each colour the solving time was 107 seconds compared to ProB's 13 seconds. We were unable to confirm the absence of a solution for 10 queens of each colour (Alloy was still running after more than 4 hours). We have also directly used the Kodkod library, through our B to Kodkod translation [18]. Here the runtime was typically a bit slower than running Alloy (e.g., 149 seconds instead of 107 using SAT4J and 123 seconds using MiniSat for 9 queens of each colour). ProB can also solve the puzzle for $n = 8$, $q = 9$ and an additional two kings (on of each colour; see [7]). The solving time is about one hour.

This example has shown that even problems considered challenging by the constraint programming community can be solved, and that they can be expressed with very little effort. The graphical visualization features of ProB were easy to setup (basically just defining an animation function in B and declaring a few images; see [13]) and were extremely helpful in debugging the model.

3.4 Extended Graph Isomorphism

The *Graph Isomorphism* Problem[11] is the final problem discussed here. To determine if two graphs are isomorphic we search for a bijection of the vertices which preserves the neighbourhood relationship.

Using B, graphs can be represented as relations of nodes, where the vertices are represented by the tuples in the relation, seen as a set. An undirected graph can hence be easily represented as the union of the directed graph relation with the inverse of the relation, basically duplicating all vertices.

Using B we can state the problem using an existential quantification over the formula used to define the *graph isomorphism* problem, following closely the mathematical problem definition. Existential quantification is expressed in B using the `#` operator, which corresponds to the $\exists$ symbol in mathematical notation. We state that there is a total bijection that maps the vertex set from one graph to the other such that two nodes are adjacent in the domain iff they are adjacent in the range of the bijection. Additionally we only need to encode

[11] `http://en.wikipedia.org/w/index.php?oldid=561096064`

the entities needed in the quantification in order to solve this problem for two specific graphs.

```
MACHINE CheckGraphIsomorphism
SETS Nodes = {a,b,c,d,e, x,y,z,v,u}
DEFINITIONS
  G1 == {a|->b, a|->c, a|->d, b|->c, b|->d, c|->e, d|->e};
  G2 == {x|->v, x|->u, x|->z, y|->v, y|->u, z|->v, z|->u}
CONSTANTS graph1, graph2, relevant
PROPERTIES
  graph1: Nodes <-> Nodes & graph2: Nodes <-> Nodes &
  graph1 = G1\/G1~ & graph2 = G2 \/ G2~ &  /* generate undirected graphs */
  relevant = dom(graph1) \/ dom(graph2) \/ ran(graph1) \/ ran(graph2) &
  #p.(p : relevant >->> relevant &
    !(x, y).(x : relevant & y : relevant =>
      (x|->y : graph1 <=> p(x)|->p(y) : graph2)))
END
```

The above specification can easily be extended with additional constraints. An industrial application of such an extended *graph isomorphism* problem was presented by ClearSy [5]. Here, ClearSy used B and ProB to find graph isomorphisms between high level control flow graphs and control flow graphs extracted from machine code gathered through a black box compiler. In addition, the memory mapping used by the compiler had to be inferred by constraint solving. For the main problem on graphs with 192 nodes each, the solution was found by ProB in 10 seconds. The ability to easily express graph isomorphism and pair it with other domain specific predicates was an important aspect in this application.

4 Industrial Applications

As hinted in the previous section where we describe how ClearSy used B and ProB to decompile machine code, the expressivity of B in combination with the constraint solving capabilities of ProB are being used in several industrial applications, in order to validate inputs for models or to solve problems similar to those shown in this paper.

A further application by Siemens is described in [12]. Siemens use the B-method to develop safety critical software for various driverless train systems. These systems make use of a generic software component, which is then instantiated for a particular line by setting a larger number of parameters (i.e., the topology, the style of trains with their parameters). In order to establish the correctness of the generic software, a large number of properties about these parameters and the system in general are expressed in B.

The data validation problem is then to validate these properties for particular data values. One difficulty is the size of the parameters; the other is the fact that some properties are expressed in terms of abstract values and not in terms of concrete parameter values (which are linked to abstract values via a gluing invariant in the B refinement process). Initially, the data validation was carried out in Atelier-B using custom proof rules [3]. However, many properties could not be checked in this way (due to excessive runtime or memory consumption) and had to be checked manually. As described in [12], Siemens now use the

ProB constraint solver to this end and have thus dramatically reduced the time to validate a new parametrisation of their generic software. This success led to this technique also being applied in Alstom and the development of a custom tool DTVT with the company ClearSy [11]. The company Systerel is also using B for data validation for a variety of customers [2]. To this end, Systerel uses a B evaluation engine which is also at the heart of Brama [20] combined with ProB.

It is interesting to note, that data validation is now also being applied in contexts where the system itself was not developed using B; the B language is "just" used to clearly stipulate important safety properties or regulations about data. This shows a shift from using B to formally prove systems or software correct, to using B as an expressive specification language for various constraint satisfaction problems. In the traditional use of B to develop software or systems correct by construction, refinement and the proving process play a central role. In this novel use of B, those aspects of B are almost completely absent. There often is no use of refinement but the properties to be checked or solved become larger and more difficult, making the use of traditional provers nigh impossible.

The ProB tool is now used by various companies for similar data validation tasks, sometimes even in contexts where B itself is not used for the system development process. In those cases, the underlying language of B turns out to be very expressive and efficient to cleanly encode a large class of data properties.

5 Conclusion and Future Work

In the first part of this article we focused on the language B and its power to express constraint satisfaction problems. Using B, we have taken on the challenges identified by Shapiro regarding the *jobs puzzle*. Our primary goal was to show that while B is a formal specification language, it is also very expressive and can be used to encode constraint satisfaction problems in a readable and concise way. Our solution to the *jobs puzzle* addresses all the challenges posed by this puzzle, we were able to create an encoding, that using mainly simple B constructs, creates a simple and straight-forward formalization. We only have to additionally provide an encoding of the implicit information required to solve the puzzle, which can also be achieved in a way that is not complex and close to a translation to English. Our encoding follows the original text closely and ProB is able to solve the puzzle efficiently.

In the second part of this article we focused on solving constraint satisfaction problems written in B using ProB. We outlined on two similar examples that it is possible to efficiently solve problems encoded on a high abstraction level.

The pairing of ProB and B can be a good trade-off between the efficiency of low level constraint solvers that make the encoding of problems very hard and high level systems that have to pay the price of abstraction by the increased amount of computation needed to solve problems. Surprisingly, for some problems (see Section 3.3) we are competitive with low-level modern constraint solving techniques. But obviously, many large scale industrial optimization prob-

lems are still out of reach of our approach. Also, compared to Alloy, the standard ProB solver is weak for certain relational operators such as image or transitive closure. But we work on further improving the constraint solving capabilities of ProB and thus reducing the overhead associated with the abstraction level of B, allowing to use ProB on more problems and domains.

As shown in the *Peaceable Queens* example, B and ProB are still awkward for solving optimization problems. The current solution is to setup a problem twice and search for a solution where one problem is solved and the other one not. This is an area we intend to do further research in the future.

Finally we discussed the industrial uses cases of B in combination with the constraint solving features of ProB. All the aspects discussed in this article show the advantages of using a feature rich and high-level language such as B to encode complex problems and at the same time making use of a high-level constraint solver to solve them. Compared to other approaches to constraint solving, ours has the advantage of extensive validation of the tool along with a double chain [11] to cross-check results, and the ability to apply proof to (parts of) B models. This makes B and ProB particularly appealing to solving constraints in safety critical application areas.

Acknowledgements We thank Daniel Plagge and Dominik Hansen for interesting discussions and feedback. Daniel Plagge also devised the Alloy solutions for the N-Queens puzzle and the peaceable armies of queens. We thank anonymous referees for their useful feedback. We also thank Mats Carlsson and Per Mildner for their support around SICStus Prolog and CLP(FD).

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. F. Badeau and M. Doche-Petit. Formal data validation with Event-B. *CoRR*, abs/1210.7039, 2012. Proceedings of DS-Event-B 2012, Kyoto.
3. O. Boite. Méthode B et validation des invariants ferroviaires. Master's thesis, Université Denis Diderot, 2000. Mémoire de DEA de logique et fondements de l'informatique.
4. 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.
5. ClearSy. Data Validation & Reverse Engineering. June 2013. Available at `http://www.data-validation.fr/data-validation-reverse-engineering/`.
6. D. Deharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In *Proceedings ABZ'2012*, LNCS 7316, pages 194–207. Springer, 2012.
7. I. P. Gent, K. E. Petrie, and J.-F. Puget. Symmetry in constraint programming. *Foundations of Artificial Intelligence*, 2:329–376, 2006.
8. D. Hansen and M. Leuschel. Translating TLA+ to B for validation with ProB. In *Proceedings iFM'2012*, LNCS 7321, pages 24–38. Springer, 2012.
9. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.
10. E. K. Jackson, T. Levendovszky, and D. Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In J. Whittle, T. Clark, and T. Kühne, editors, *MoDELS*, LNCS 6981, pages 653–667. Springer, 2011.

11. 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.
12. 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.
13. M. Leuschel, M. Samia, J. Bendisposto, and L. Luo. Easy Graphical Animation and Formula Viewing for Teaching B. *The B Method: from Research to Teaching*, pages 17–32, 2008.
14. 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.
15. W. Mccune. Otter 3.3 reference manual, 2003.
16. I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A system for answer set programming. *CoRR*, cs.AI/0003033, 2000.
17. D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Proceedings IFM 2007*, LNCS 4591, pages 480–500. Springer-Verlag, 2007.
18. 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.
19. R. Schwitter. The jobs puzzle: Taking on the challenge via controlled natural language processing. *Theory and Practice of Logic Programming*, 13:487–501, 7 2013.
20. 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.
21. S. C. Shapiro. The jobs puzzle: A challenge for logical expressibility and automated reasoning. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, 2011.
22. S. C. Shapiro and The SNePS Implementation Group. *SNePS 2.7.1 User's Manual*. Department of Computer Science and Engineering University at Buffalo, The State University of New York, Dec. 2010.
23. B. M. Smith, K. E. Petrie, and I. P. Gent. Models and symmetry breaking for peaceable armies of queens. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 271–286. Springer Berlin Heidelberg, 2004.
24. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.
25. 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.
26. L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning: Introduction and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1984.