Specifying and Solving Constraint Satisfaction Problems in B

Michael Leuschel and Daniel Plagge

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

ProB [3] is an animator and model checker for the B-method, supporting classical B as well as Event-B. In order to be able to animate B models, ProB has to find solutions for various predicates expressed in predicate logic, set theory and arithmetic:

- finding solutions for the parameters of events, which make the guard of the event become true,
- finding solutions for constants within contexts, which satisfy the axioms,
- and generally within formulas, determining whether solutions for existentially quantified variables exist.

In recent work, the kernel of ProB has been improved [4], proving more fine grained constraint propagation. In this paper, we study to what extent classical constraint satisfaction problems can be conveniently expressed as B predicates, and then solved by ProB. We study problems such as the n-Queens problem, graph colouring, graph isomorphism detection, time tabling, Sudoku, Hanoi, magic squares, Alphametic puzzles, and several more. We also compare the performance with respect to other tools, such as the model checker TLC [5] for TLA$^+$ [2], AnimB for Event-B, and Alloy [1].

The experiments show that some constraint satisfaction problems can be expressed very conveniently in B and solved very effectively with ProB. Some problems, such as Hanoi, can also be more easily expressed and solved as a model checking problem (rather than as finding solutions to predicates). For some others, the performance of ProB is still sub-optimal with respect to, e.g., Alloy, and we will describe how we plan to overcome this shortcoming in the future. Our long term goal is that B can not only be used to as a formal method for developing safety critical software, but also as a high-level constraint programming language.

N-Queens

Due to space constraints, we only describe one experiment in this abstract: the well-known n-Queens puzzle[1]. It can be expressed very succinctly by specifying a constant q, which has to satisfy the following axioms:

[1] http://en.wikipedia.org/wiki/Eight_queens_puzzle

$$q : 1..n \rightarrowtail\!\!\!\!\!\rightarrow 1..n \ \wedge$$

$$\forall(i,j).(i : 1..n \wedge j : 2..n \wedge j > i \Rightarrow q(i) + j - i \neq q(j) \wedge q(i) - j + i \neq q(j))$$

Experimental results for finding the first solution to this predicate with ProB can be found in Table 1. ProB can deal with this problem quite effectively, solving the predicate for up to $n = 17$ in less than 0.1 seconds each on a MacBook Pro 3.06 GHz Core2 Duo. For $n = 70$ the problem is solved in about 9 seconds.

The animator AnimB can solve this predicate only for $n = 5$. (AnimB cannot solve it for $n = 4$; nor can it determine that there are no solutions for $n = 3$.)

We can translate the above B predicate into TLA$^+$ [2] quite easily (cf. Appendix B.1). Note that, as far as we know, TLA$^+$ does not provide a built-in way to declare q' to be an injection; we have therefore to add an additional inequality inside the universally quantified expression. The model checker TLC (version 3.5 of the TLA Toolbox) [5] can solve this predicate for n up to 6 quickly, but already takes 4 minutes and 3 seconds for $n = 8$. For $n = 9$, TLC took over 1 hour and 45 minutes, i.e., more than 5 orders of magnitude slower than ProB(0.02 seconds). Note that we are only testing TLC's capability to solve predicates, not its (very effective) disk-based model checking capabilities.

The reason for this big performance difference is that TLC deals with conjuncts from left-to-right (see page 239 of [2]). As such, TLC enumerates all possible total functions from $1..n$ to $1..n$, then checking each one of them whether it satisfies the universally quantified formula. (Note: One cannot change the order of the conjuncts above, otherwise TLC complains that q is undefined.)

We have also experimented with Alloy [1] (version 4.1.10), which translates predicates to propositional logic formulas fed to a SAT solver. Solving the equivalent Alloy model (cf. Appendix B.3) for $n = 8$ takes 0.80 seconds with the default SAT4J sat solver. With minisat rather than SAT4J as backend, Alloy takes 0.24 seconds. In both cases, this is considerably faster than TLC. However, as Table 1 shows, ProB is still considerably more efficient for this task (in the presentation we will also discuss other constraint satisfaction problems where Alloy is considerably more efficient).

Finally, we have also adapted the TLC model so that the solution can be found by model checking. In other words, we have added an event which places the next queen on a row such that it attacks none of the already placed queens. This is a more low level model; for example, it encodes in which order the queens are placed (namely from left to right). As you can see in Table 1, this is more efficient that the high-level TLC solution, but still takes more than 45 minutes for 14 queens.

References

1. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.
2. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

3. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
4. 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, 2009.
5. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA$^+$ specifications. In L. Pierre and T. Kropf, editors, *CHARME*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.

A N-Queens Table

B N-Queens Models

B.1 TLA$^+$ high-level model

```
---- MODULE queens ----

EXTENDS Naturals, FiniteSets

VARIABLE q, n, solved
----

Init == /\ q=[i \in 1..2 |-> 0]
        /\ n=8
        /\ solved = 0

Solve == /\ solved=0
         /\ q' \in [1..n -> 1..n]
         /\ \A i \in 1..n : (\A j \in 2..n : i<j =>
              q'[i] # q'[j] /\ q'[i]+i-j # q'[j] /\ q'[i]-i+j # q'[j])
         /\ solved'=1
Spec == Init /\ [] [Solve]_<<n,q>>
=======
```

B.2 TLA$^+$ lower-level model for model checking

```
---- MODULE queensMC ----

EXTENDS Naturals, FiniteSets

VARIABLE q, n, cur, pos
----

Init == /\ q=[i \in 1..14 |-> 0]
        /\ n=14
        /\ cur=1
        /\ pos=0
```

n	TLC	Alloy Sat4J	Alloy Minisat	ProB 1.3.2	TLC MC
6	1	0.224	0.133	0.005	1
7	10	0.56	0.194	0.01	1
8	243	0.80	0.241	0.01	1
9	8747	0.69	0.406	0.02	3
10		1.01	0.44	0.02	2
11		2.05	0.54	0.03	13
12		4.20	0.72	0.04	75
13		5.10	0.85	0.06	498
14		5.48	1.15	0.05	2737
15		9.84	2.08	0.06	
16		21.74	3.61	0.06	
17		48.45	4.62	0.08	
18		43.56	7.58	0.10	
19		80.38	14.70	0.11	
20		100.87	16.03	0.13	
21		127.20	21.36	0.15	
22		139.94	24.84	0.17	
23		238.36	27.09	0.20	
24		254.56	31.50	0.22	
25		284.82	32.23	0.26	
26		332.35	56.13	0.29	
27		439.77	52.02	0.30	
28		458.32	81.49	0.34	
29		600.50	91.61	0.38	
30		775.44	87.28	0.43	
31		905.08	161.94	0.47	
32		1925.09	245.55	0.52	
40			454.22	1.08	
50				3.25	
60				5.63	
70				9.09	
100				80.41	

Fig. 1. Time to find first solution for N-Queens

```
Step == /\ cur<=n
         /\ pos' \in 1..n
         /\ cur'=cur+1
         /\ q' = [q EXCEPT ![cur] = pos']
         /\ n'=n
         /\ \A i \in 1..(cur-1) :  q[i] # pos' /\ q'[i]+i-cur # pos' /\ q'[i]-i+cur # pos'
GINV == cur<=n
Spec == Init /\ [] [Step]_<<n,q,cur,pos>>
=======
```

B.3 Alloy model

Note that the Alloy model uses the built-in Int type with a bit width of 5 for n up to 15, a bit width of 6 for $n > 15$ and $n < 32$, etc.

```
sig Queens {
  row : Int,
  col: Int
} {
 row >= 0 and row < #Queens
 and col >= 0 and col < #Queens
}

pred nothreat(q1,q2 : Queens) {
q1.row != q2.row
and q1.col != q2.col
and q1.row+q2.col-q1.col != q2.row
    and q1.row-q2.col+q1.col != q2.row
}

pred valid { all q1,q2 : Queens |
    q1 != q2 => nothreat[q1, q2]
 }

fact card {#Queens = 8}
run valid for 8 Queens, 5 int
```