Three is a Crowd: SAT, SMT and CLP on a Chessboard

Sebastian Krings, Michael Leuschel, Philipp Körner,
Stefan Hallerstede, Miran Hasanagić

This is a pre-print version archived by P. Körner at
https://pkoerner.github.io/pages-output/publications/.

The final publication is available at Springer via
https://doi.org/10.1007/978-3-319-73305-0_5.

Three is a crowd: SAT, SMT and CLP on a chessboard

Sebastian Krings[1], Michael Leuschel[1], Philipp Körner[1], Stefan Hallerstede[2], and Miran Hasanagić[2]

[1] Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{krings,leuschel}@cs.uni-duesseldorf.de
p.koerner@uni-duesseldorf.de
[2] Department of Engineering, Aarhus University
Aarhus, Denmark
{sha,miran.hasanagic}@eng.au.dk

Abstract. Constraint solving technology for declarative formal models has made considerable progress in recent years, and has many applications such as animation of high-level specifications, test case generation, or symbolic model checking. In this article we evaluate the idea of using very high-level declarative models themselves to express constraint satisfaction problems. In particular, we study an old mathematical puzzle from 100 years ago, called the crowded chessboard. We study various high-level and low-level encodings and solutions, covering SAT, SMT and CLP-based solutions of the puzzle. Additionally, we present a new technique combining SAT-solving with CLP which is able to solve the puzzle efficiently.

1 Motivation: Model-Based Constraint Solving

Logic programming and constraint programming are key members of the declarative language paradigm. Logic programs and constraint (logic) programs tend to be much more declarative than traditional imperative programs, but developers still have to consider considerable operational aspects. High-level formal methods languages like B, TLA$^+$, or Z, are more declarative still: they were developed to be specification languages, with little concern for execution.[3] In between logic programs and formal methods are logic-based encodings like SMT-LIB.

In this paper, we study a non-trivial constraint satisfaction problem, investigating both the ease of expressing the problem and the solving performance for a range of declarative languages, from Prolog, onto SAT, SMT and high-level formal specification languages. One popular specification language is B [1], which has its roots in first-order predicate logic, with (higher-order) set theory and arithmetic. In that respect, it is quite similar to other formal methods such as TLA$^+$, Z or even VDM. Constraint solvers have made a big impact for formal

[3] Some even argue that formal specifications should be non-executable [11].

methods in general and B or TLA$^+$ in particular, by providing validation technology for *proof* [8, 18], *animation* [15], *bounded or symbolic model checking* [13], and *test case generation* [21].

We want to turn our focus from constraint solving technology for validating models towards using formal 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 [16], we already argued that B is well suited for expressing constraint satisfaction problems in other domains as well. This was illustrated on the Jobs puzzle challenge [24] and we are now solving various time tabling problems [23].

In this paper, we want to present one particular benchmark puzzle, and various ways to solve it. One motivation was that the puzzle was formulated exactly 100 years ago. A more academic motivation is that the puzzle is relatively easy to explain and hence should be relatively easy for other researchers to provide their own solutions in their favorite declarative formalism and compare it with ours.

Indeed, a real-life problem such as the time-tabling problem in [23] is very arduous to describe in an article, and would require considerable investment to write a solution in another formalism, requiring many weeks or months of effort. A puzzle such as the N-queens puzzle, on the other hand, is too simple and allows many very special encodings, which cannot be easily used in real-life, practical problems. The encoding is not really a challenge, and a solution to the N-queens puzzle only provides limited insights into practicality of a formalism.

We feel that the crowded chessboard problem provides the almost perfect middle ground, even though it is a combinatorical problem: the problem can still be explained in a paper, and has some entertaining aspects as well. Furthermore, solutions can be easily checked by a human, provided the solution is rendered graphically. In the following, we investigate different declarative encodings of the problem, considering ease of understanding, correspondance to the problem statement and solving performance. Constraint solving technology can be classified into the following broad categories, which we all experiment with:

- Constraint programming, used in Sections 3 and 4 and used by ProB's default solver in Section 2.1,
- Translation to boolean satisfiability and using SAT solvers in Section 5,
- Translation to SMT-LIB and using SMT solvers in Section 6.

All models used in this paper are available at:
`https://github.com/leuschel/crowded-chessboard`

2 The Crowded Chessboard Problem and its B Solution

The Crowded Chessboard Problem

The crowded chessboard is a 100 year old problem appearing as problem 306 in Dudeney's book [9]. The book provides the following description of the problem:

> "The puzzle is to rearrange the fifty-one pieces on the chessboard so that no queen shall attack another queen, no rook attack another rook, no bishop attack another bishop, and no knight attack another knight. No notice is to be taken of the intervention of pieces of another type from that under consideration - that is, two queens will be considered to attack one another although there may be, say, a rook, a bishop, and a knight between them. And so with the rooks and bishops. It is not difficult to dispose of each type of piece separately; the difficulty comes in when you have to find room for all the arrangements on the board simultaneously."

board size	♕	♖	♗	♘
5	5	5	8	5
6	6	6	10	9
7	7	7	12	11
8	8	8	14	21
9	9	9	16	29
10	10	10	18	37
11	11	11	20	47
12	12	12	22	57
13	13	13	24	69
14	14	14	26	81
15	15	15	28	94
16	16	16	30	109

Table 1: Configurations

Table 1 shows the maximum number of pieces for which the puzzle can still be solved. To gain unsatisfiable benchmarks, we will increment the number of knights.

2.1 B Solution

We now try to formalise this problem in the B language, as clearly as possible. Our goal here is to make a human readable formalisation of the model, where a human can be convinced that the problem has been modelled correctly. Indeed, in constraint programming it is quite often the case that subtle errors creep into a formalisation, which can go unnoticed for quite some time. Thus, below we also intersperse the formal model with a few visualisations and other sanity checks, to ensure that the model is correct. The LaTeX of this section has been derived by executing the model, see [14].

For the visualisations, we first set the dimension n of the board to 5 and thus have the following set of possible indexes on the chessboard: $Idx = \{1, 2, 3, 4, 5\}$. Furthermore, we define the set of chess pieces, including a special piece $Empty$ for empty squares: $PIECES = \{Queen, Bishop, Rook, Knight, Empty\}$.

2.2 Specifying Movements

First, we compute for every position on the board which squares can be reached by a horizontal or vertical move, i.e., a function which returns a set of coordinates

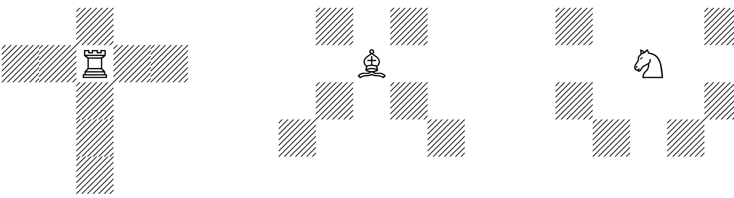


Fig. 1: Rook Attack Fig. 2: Bishop Attack Fig. 3: Knight Attack

that can be attacked from a given position. For example, we have $moveHV(2 \mapsto 3) = \{(1 \mapsto 3), (2 \mapsto 1), (2 \mapsto 2), (2 \mapsto 4), (2 \mapsto 5), (3 \mapsto 3), (4 \mapsto 3), (5 \mapsto 3)\}$, visualized in Fig. 1. This can be expressed in B using a lambda function:

$$moveHV = \lambda(i,j).(\{i,j\} \subseteq Idx \mid \{k,l \mid \{k,l\} \subseteq Idx \wedge (i,j) \neq (k,l) \wedge \\ (i = k \vee j = l)\})$$

Now, we compute the diagonal moves, e.g., we have $moveDiag(2 \mapsto 3) = \{(1 \mapsto 2), (1 \mapsto 4), (3 \mapsto 2), (3 \mapsto 4), (4 \mapsto 1), (4 \mapsto 5)\}$, visualized in Fig. 2.

$$moveDiag = \lambda(i,j).(\{i,j\} \subseteq Idx \mid \{k,l \mid \{k,l\} \subseteq Idx \wedge (i,j) \neq (k,l) \wedge \\ (k - i = l - j \vee i - k = l - j)\})$$

To describe the moves of the knight, we introduce the following auxiliary function to compute the absolute distance between indices:

$$dist = \lambda(i,j).(i \in \mathbb{Z} \wedge j \in \mathbb{Z} \mid \text{ IF } i \geq j \text{ THEN } i - j \text{ ELSE } j - i \text{ END}).$$

Using it, the knight's moves can now be described as follows:

$$moveK = \lambda(i,j).(\{i,j\} \subseteq Idx \mid \{k,l \mid \{k,l\} \subseteq Idx \wedge i \neq k \wedge \\ j \neq l \wedge dist(i,k) + dist(j,l) = 3\})$$

For example, we have $moveK(2 \mapsto 3) = \{(1 \mapsto 1), (1 \mapsto 5), (3 \mapsto 1), (3 \mapsto 5), (4 \mapsto 2), (4 \mapsto 4)\}$, visualized in Fig. 3.

We can now assemble all of these into a single higher-order function, which for each piece returns the attacking function (which in turn for each position on the board returns the set of attacked positions):

$$attack = \{(Rook \mapsto moveHV), (Bishop \mapsto moveDiag), (Knight \mapsto moveK), \\ \big(Queen \mapsto \lambda(p).(p \in (Idx \times Idx) \mid moveHV(p) \cup moveDiag(p))\big), \\ (Empty \mapsto ((Idx) \times (Idx)) \times \{\varnothing\})\}$$

2.3 Specifying the Number of Pieces

After having modelled how the pieces move, we now describe how many pieces of each type we want to place on the board as follows. Note that the second

conjunct asserts the number of all placed figures, including the empty field, to be n^2. That is, it computes the number of empty squares from the given figures. The only hard-coded part is the number of knights (5 in this case):

$$nrPcs \in PIECES \to 0 \dots n^2 \wedge \Sigma(p).(p \in PIECES | nrPcs(p)) = n^2 \wedge$$
$$\big((Queen \mapsto n) \in nrPcs \wedge (Rook \mapsto n) \in nrPcs \wedge$$
$$(Bishop \mapsto 2 * n - 2) \in nrPcs \wedge (Knight \mapsto 5) \in nrPcs\big)$$

This gives rise to the following solution for $n = 5$:

$$nrPcs = \{(Queen \mapsto 5), (Rook \mapsto 5), (Bishop \mapsto 8),$$
$$(Knight \mapsto 5), (Empty \mapsto 2)\}$$

2.4 Solving the Crowded Chessboard Puzzle

Solving the crowded chessboard puzzle now amounts to solving the predicates:

$$board \in Idx \times Idx \to PIECES$$
$$\forall P.(P \in PIECES \Rightarrow card(\{p | p \in dom(board) \wedge board(p) = P\}) = nrPcs(P))$$
$$\forall (pos, piece).((pos \mapsto piece) \in board$$
$$\Rightarrow \forall pos2.(pos2 \in attack(piece)(pos) \Rightarrow board(pos2) \neq piece))$$

Observe how compact the core B representation of the problem is. The first predicate sets up the board and specifies the possible pieces that can be put on the board. The second predicate specifies how many pieces of each kind should be placed on the board. The third predicate posits that no piece can attack another piece of the same kind. In case we want to add a new kind of piece or change the rules for an existing piece, these three predicates would remain unchanged; one would only have to adapt the definition of the *attack* function. The first solution found by ProB for n=5 is visualized in Fig. 4.

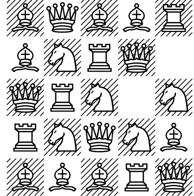


Fig. 4: Solution

Of course, the model could be improved by adding further (implied) constraints, e.g., by asserting that bishops have to be placed on border cells. However, by doing so, we would decrease correspondence to the original problem formulation, sacrificing comprehensibility for solving speed.

2.5 New CLP and SAT Integration

We believe that the above encoding is compact, elegant, easy to understand, validate and adapt by a human. Unfortunately, as it stands, this encoding can be solved by ProB only for small values of n. Given the success of the SAT approach later in Section 5, one may wonder why ProB's SAT backend [20] cannot be applied to the B model instead of the default CLP solver.

The reason is the higher order nature of the model: the SAT backend relies on Kodkod [26], a relational model finder translating its input language to SAT. However, Kodkod can only translate certain first-order constructs and data structures. For this reason, the technique in [20] statically splits a predicate P to be solved into a part P_{SAT} that can be translated by Kodkod and another one P_{CLP} that will be solved by ProB's default solver. The solving now proceeds by first performing deterministic propagation in P_{CLP}, possibly instantiating values in P_{SAT}. Then, P_{SAT} is solved, the solution of which is fed into ProB for solving P_{CLP}. The core problem is that, when solving P_{CLP} we cannot generate new constraints to be shipped to the SAT solver. This is exactly what we would need in our case: let ProB evaluate the higher-order *attack* function from Section 2.2, unroll the involved quantifiers and then ship simple first-order constraints to a SAT solver.

We have implemented exactly this style of integration in reaction to the challenge posed by the crowded chessboard. The idea is to allow the user to annotate parts of the constraint as to be treated by a SAT solver. Note that these parts can be inside quantifiers, in which case *ProB* would expand these quantifiers and (possibly) compute relevant values using higher-order datastructures.

We annotate implications, equalities, inequalities and cardinality constraints for SAT and let ProB deal with the rest, e.g., determining and applying the higher-order *attack* function. Doing so, the first two predicates given in Section 2.4 can completely be given to a SAT solver via Kodkod. The third one however is unrolled by ProB, only the inner predicate gets solved by Kodkod:

$$\overbrace{\forall(piece,i,j,i2,j2).(i\mapsto j\in(1\dots n)\times(1\dots n)\wedge i2\mapsto j2\in attack(piece)(i\mapsto j)}^{ProB\ (CLP)}$$
$$\Rightarrow \underbrace{board1((i-1)*n+j)=piece\Rightarrow board1((i2-1)*n+j2)\neq piece}_{Kodkod\ (SAT)})$$

Observe that we have rewritten the quantification over $(piece,i,j,i2,j2)$ slightly, to enable ProB to completely expand the quantifier and apply the higher-order function *attack*.

3 A Prolog CLP(FD) Solution

Given that ProB's default CLP solver does not scale for the high-level B model, we have written a direct encoding of the crowded chessboard problem in SICStus Prolog using the finite domain library CLP(FD) [5]. As the following code snippet shows, the chessboard is encoded as a list of length n of lists of n finite domain variables each, in the range 0 to 4. The value 0 denotes an empty square, 1 a queen, 2 a rook, 3 a bishop, and 4 a knight. The entry predicate is `solve(N,K,Sol)`, where N specifies the size of the board and K the number of knights to be placed. The solution is returned in *Sol*; by backtracking all solutions can be found.

```
solve(N,Knights,Sol) :-
   length(Sol,N),
   maplist(pieces(N),Sol),
   append(Sol,AllPieces),
   ... constraint setup ...
   labeling([ffc],AllPieces).
pieces(N,L) :- length(L,N), clpfd:domain(L,0,4).
```

Above, *AllPieces* is a flattened list of the domain variables. Below, we explain the most important part of the constraint setup. To ensure that we place the correct number of pieces of each type onto the board we use the `global_cardinality` constraint of CLP(FD):

```
Bishops is 2*N-2,
Empty is N*N - 2*N - Bishops - Knights,
global_cardinality(AllPieces,[0-Empty,1-N,2-N,3-Bishops,4-Knights])
```

We use the following auxiliary predicates for rows, columns and diagonals to ensure that queens, rooks and bishops do not attack each other:

```
exactly_one(Piece,List) :- count(Piece,List,'#=',1).
at_most_one(Piece,List) :- count(Piece,List,'#<',2).
```

For example, `maplist(exactly_one(1),Sol)` ensures that no queens attack each other on rows. We can see that this is a lower-level encoding than the B-solution: There is special code for knights, which are treated quite differently from the other pieces. Furthermore, the model checks explicitly that there is exactly one queen on every row and column.

4 OscaR / Scala solution

As an additional solution, we have written an encoding in Scala using the OscaR Constraint Programming library [19]. This library supports a CLP(FD) modelling approach combined with various search heuristics. Compared to ProB, OscaR usually sacrifices completeness for efficiency, while also providing support for built-in constraint functions. Such functions have dedicated and optimised domain restriction and search algorithms. Variables in OscaR are explicitly declared with their domains, e.g., `val Pieces = Set(0 to k)`. The representation of the problem in the OscaR model follows closely that of the B model. However, implementation concerns need to be considered in OscaR. For instance, when dereferencing a 2-dimensional array, the first index must be of type `int` (but not of type `CPIntVar`). Hence, to refer to rows and columns usually the 2-dimensional array and its transpose are required. E.g., in OscaR the board is modelled by

```
val board = Array.fill(n, n)(CPIntVar(Pieces))
val board_t = board.transpose
```

Constraints are added to the constraint store by means of a function `add(c)`, where `c` is a constraint. Constraints may not contain quantifiers. As a consequence, OscaR models are less abstract than B models as well as not as easy to read and understand, e.g., concerning knight attacks:

```
for(i<-0 until n; j<-0 until n)
  (for(u<-Seq(-2, -1); v<-Seq(-2, -1, 1, 2);
    if(u.abs!=v.abs && i+u>=0 && i+u<n && j+v>=0 && j+v<n))
  yield board(i+u)(j+v)).foreach
    {t=>add(board(i)(j).isEq(k)==>t.isDiff(k))}
```

We have analyzed two differently structured versions of the model.

1. A monolithic version where constraints are propagated between the four sub-problems, similarly to the B and Prolog models.
2. A layered version where the sub-problems are solved in a predetermined order (first bishops, then rooks, then queens, finally knights) where constraints are propagated within the sub-problems and allocated positions are passed as constants from one to the next sub-problem. The only way the sub-problems communicate is by backtracking.

Three similar sets `Bset`, `Rset` and `Qset` are used in the layered model to pass the already allocated board positions to the next sub-problem dealing with bishops, rooks and queens, respectively. E.g., `Bset` is declared as

```
val Bset = scala.collection.mutable.Set[(Int, Int)]()
```

When the solver for the bishops problem has found a solution, the positions of the bishops are copied into `Bset`,

```
for(i<-0 until n; j <- 0 until n)
  if(board(i)(j).value.toInt!=0) Bset += (i, j)
```

The sub-model for the rooks contains constraints to block these positions,

```
for(i<-Bset) add(board(i._1)(i._2)!==r)
```

and adds the constraints for the placement of the rooks using global cardinality constraints `gcc`

```
for(i <- 0 until n){
  add(gcc(board(i), Array((r,CPIntVar(0 to 1)))))
  add(gcc(board_t(i), Array((r,CPIntVar(0 to 1)))))
}
add(gcc(board.flatten.toArray, Array((r,CPIntVar(nR)))))
```

The same scheme is followed for the queens and knights subproblems.

We have varied the order of the sub-problems discussed above and the alternatives have a worse performance than the one presented here. This is likely due to the degrees of liberty when placing the figures, influenced by the possible

moves and the number of figures. As an example, for models starting with the bishops sub-problem, the search performed better than for those starting with the queens sub-problem. This could be a consequence of having to place almost twice as many bishops as queens, i.e., we reduce the search space more when passing the positions occupied by bishops while still having enough liberty in placing the queens on the board.

The layered approach is not suitable for testing of unsatisfiability even for small board sizes: backtracking is more costly than direct constraint propagation. In addition, the search uses limits on the permitted number of iterations. In the benchmarks this is not an issue, because the runtime is dominated by the sub-problems. The ability to experiment easily with model representations and search heuristics is an advantage of OscaR. The price to pay for this is the lower level of abstraction which makes it more difficult to validate the model against the problem statement.

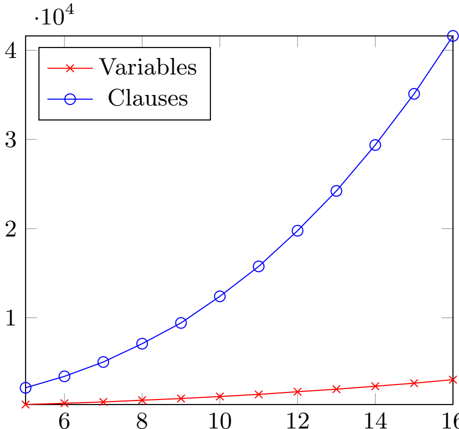
We have also experimented with modelling variants using a piece-centric model instead of a board-centric model. The models were generally less performant than the two discussed above. We see two reasons for this: (1) when the models are to be considered jointly, some optimisations usually applied to piece-centric models were not possible because they depend on specific data-models, (2) it requires replacing a few arithmetic calculations and comparisons with many boolean comparisons. Due to the small board sizes used there are no performance issues due to memory management. We also observed that when trying to break symmetries in these models, the low level of the programming and the restrictions on referencing arrays obscured the added constraints. This reduces the legibility of the models further and makes validation more difficult.

5 SAT Encoding

A different approach towards solving the crowded chessboard puzzle is to encode it using pure propositional calculus and employ SAT solvers such as Minisat [10]. The general idea is as follows: For each kind of chess figure to be placed we introduce $n \times n$ boolean variables to encode the chessboard, e.g., $queen_{2,5}$. We set a variable to `true` to represent a figure on a certain square, while `false` represents an empty square.

Most placement rules can be encoded easily. First, we encode that no two figures can be placed on the same square. This is done by enforcing, that for each combination of indices i, j the fields encoded as $queens_{i,j}$, $rooks_{i,j}$, $bishops_{i,j}$ and $knights_{i,j}$ should not be true simultaneously. For example, for $queens$ and $rooks$ we assert $\forall i \in 1 \ldots n, j \in 1 \ldots n \cdot queens_{i,j} = \top \Rightarrow rooks_{i,j} = \bot$. Of course, using universal quantification is not allowed in the input of a common SAT solver. Thus, we have to unroll the formula and set the constraint up explicitly for all combinations of chess pieces, i and j.

Next, we have to encode the movement of figures. For a solution to be correct, we require that no two figures of the same kind can capture each other. Again, this can be encoded using implications, e.g., for the case of linear movement we



n	variables	clauses
5	276	2152
6	416	3434
7	564	5042
8	760	7096
9	948	9424
10	1176	12406
11	1412	15762
12	1712	19764
13	1996	24224
14	2328	29374
15	2668	35110
16	3064	41608

Fig. 5: Board Size vs. Variables / Clauses (Maximum Number of Knights)

assert $\forall i \in 1 \ldots n, j \in 1 \ldots n, x \in 1 \ldots n, x \neq j \cdot queens_{i,j} = \top \Rightarrow queens_{i,x} = \bot$. Obviously, diagonal movement can be encoded similarly. Again, we have to unroll the resulting constraints to remove the universal quantification.

Encoding the knights movement is more complicated and can not be expressed as easily using quantification. They key idea however remains the same: we iterate over all possible fields a knight can reach and set up implications preventing other knights from being placed there. To do so, we compute the set of fields a knight can reach from a current field i, j:

$$\begin{aligned} reachable_{i,j} = \{&(i+1, j+2), (i-1, j+2), (i+1, j-2), (i-1, j-2), \\ &(i+2, j+1), (i+2, j-1), (i-2, j+1), (i-2, j-1)\}. \end{aligned}$$

Of course, we have to keep in mind not to violate the borders of the chess board:

$$reachable2_{i,j,n} = \{(x, y) | (x, y) \in reachable_{i,j} \wedge x \in 1 \ldots n \wedge y \in 1 \ldots n\}.$$

Following, we can prevent knights from attacking each other:

$$\forall i \in 1 \ldots n, j \in 1 \ldots n, (x, y) \in reachable2_{i,j,n} knights_{i,j} = \top \Rightarrow knights_{x,y} = \bot$$

In contrast to movement, encoding the number of figures to be placed is quite involved. This is due to the fact the very low-level SAT encoding does not feature constructs like cardinality computation. There are extensions to SAT introducing *quantified Boolean satisfiability*, or QSAT for short. When the quantification of variables is not existential, SAT becomes PSPACE-complete.

To summarize, we have two possible ways to proceed: use a solver for quantified boolean formulas or encode the cardinality constraints ourselves. As we intended to present a low-level alternative to the high-level encoding in B and ProB, we went with the second alternative. Essentially, there are three ways to encode the sum of boolean variables by means of pure propositional calculus:

- Encode a bit-wise adder and treat every boolean variable as an input bit. The result can be compared to the required cardinality using bit-level arithmetic. This is the approach we will use in our benchmarks.
- An improved encoding of bitwise addition called the *totalizer* [3]. In contrast to the naive encoding it provides improved unit propagation. However, for the problem at hand we did not observe any speedup.
- Use a sorting network as outlined in [2].

As can be seen in Fig. 5, encoding the crowded chessboard puzzle using pure propositional calculus involves introducing numerous variables and connecting them by a high number of relatively simple constraints. In particular, the number of clauses rises quadratically, as expected due to the pairwise constraints.

6 SMT Encoding

In contrast to the very low-level encoding needed for SAT solvers, SMT solvers support a much richer logic. In particular, cardinality constraints can be expressed by means of addition. Furthermore, we can again use quantifiers to express the relations between chess pieces. We investigated three possible encodings of the crowded chessboard puzzle into SMT:

1. The board-centric approach, where we try to find a function mapping positions to chess pieces occupying them.
2. The piece-centric approach, where we try to find a function mapping pieces to their position on the chess board.
3. A low-level approach using a boolean encoding similar to Section 5. This time however, cardinality is expressed using integer arithmetic.

For the first two approaches, fields are encoded as pairs of two integers ranging from 1 to n. The two functions $first$ and *second* are used to access the first and second entry. A set of common predicates is used to set up the constraints:

$$\begin{aligned}
on_board(x) &\Leftrightarrow 1 \leq first(x) \leq n \wedge 1 \leq second(x) \leq n \\
not_same_row(x, y) &\Leftrightarrow first(x) \neq first(y) \\
not_same_col(x, y) &\Leftrightarrow second(x) \neq second(y) \\
not_same_diag(x, y) &\Leftrightarrow |first(x) - first(y)| \neq |second(x) - second(y)|
\end{aligned}$$

All of them can directly be encoded using SMT-LIB, the common input language of SMT solvers. For the first approach, we try to find *board*, a mapping of pairs to integers ranging from -1 to 3, where -1 represents an empty square, 0 a queen, 1 a rook, 2 a bishop and 3 a knight. We assert that the board may not hold other values and that a figure may not be placed outside of the field: $\forall x \cdot -1 \leq board(x) \leq 3 \wedge \neg on_board(x) \Rightarrow board(x) = -1$.

Movement and attacking is also modeled using universal quantification. For instance, the fact that two rooks may not attack each other is encoded as follows:

$$\begin{aligned}
&\forall x, y \cdot board(x) = 1 \wedge board(y) = 1 \wedge x \neq y \\
&\qquad \Rightarrow not_same_row(x, y) \wedge not_same_col(x, y)
\end{aligned}$$

Cardinality is encoded by enforcing the existence of n pairs where a queen is placed. The first universal quantification can be expressed efficiently in SMT-LIB by a *distinct* constraint:

$$\exists q_1, \ldots, q_n \cdot \big((\forall i, j \in [1, n], i \neq j \cdot q_i \neq q_j) \wedge (\forall i \in [1, n] \cdot board(q_i) = 0)\big)$$

This first approach was not successful, as it could not compete with the SAT approach even for small board sizes. While the usage of SMT instead of SAT greatly increases expressiveness and therefore understandability of the encoded problem, it also causes a performance decline if used as above.

The second approach, the piece-centric view of the puzzle can be extracted out of the first one by unrolling *board*. Essentially, we set up a variable for each queen, rook, bishop and knight. In consequence, we do not have to consider empty squares anymore. Furthermore, by using the correct number of variables, checking cardinality boils down to checking inequality.

Now that we have immediate access to the different figures, we can hardcode some simple symmetry reductions directly into the constraints. For instance, we know that the queens cannot share a row at all. Hence, we can sort them by asserting $\forall i \in [1, n] \cdot first(q_i) = i$.

We could again use quantifiers for the attack relation. For instance, in case of the queens we could reuse the *not_same_diag* predicate defined above:

$$\forall x, y \cdot \left(\bigvee_{i=1}^{n} x = q_i\right) \wedge \left(\bigvee_{i=1}^{n} y = q_i\right) \wedge x \neq y \Rightarrow not_same_diag(x, y).$$

However, SMT solvers such as Z3 [7] and CVC4 [4] currently do not detect saturation of the left hand side, i.e., they do not realize that the quantifier in fact handles all combinations of two queens. In consequence, we decided to unroll the universal quantifier and assert all *not_same_diag* combinations individually.

7 Related Work and Other Encodings

There are well-known approaches (e.g., [12]) to encoding constraint problems using integer programming (IP). Instead of asking for logical satisfaction one asks for the solution to a minimisation problem. An IP formulation for the crowded chessboard problem has been proposed in [6] where two approaches are discussed: (i) counting the number of attacks or (ii) a direct binary model for the logical constraints. The objective function for model (i) is binary discarding the extra information provided by counting the number of attacks. The XPRESS-MP models that are mentioned are not accessible. However, a MiniZinc model and a Picat encoding based on the direct binary model of [6] is available. For the MiniZinc version, a comment in the model states that for $n = 8$ the computation of a solution takes 108 seconds using ECLiPSe/eplex [22, 25], however, in our benchmarks solutions were found much faster. A more low-level implementation of the direct binary model has been implemented directly in ECLiPSe/eplex. As one would expect, this implementation obscures the clear abstract description of [6] when using a Prolog-like notation.

n	k	B plain	B +Kodkod	CLP(FD)	OscaR plain	OscaR split	SAT	SMT board	SMT figure	SAT	ECLiPSe plain	ECLiPSe MiniZinc	Picat
5	5	6.1	8.3	2.6	2.8	1.5	0.7	117.6	1.9	0.4	0.6	0.3	1.3
*5	6	28.4	8.3		2.2		0.8			0.7	0.6	0.3	3.0
6	9		10.8	2.0	85.9	3.0	0.8		636.4	1.7	0.6	0.5	3.5
*6	10		11.9		485.6		0.7			3.8	3.9	1.2	12.8
7	11		16.7	42.8		7.4	0.9			41.4	1.1	0.6	4.0
*7	12		16.1	10.9			0.2			48.6	0.7	0.5	3.3
8	21		75.5			41.5	13.2			402.1	1.0	4.8	8.5
*8	22		374.1				160.1			1535.0	54.9	93.7	516.0
9	29		709.5				184.7				16.9	13.5	11.7
*9	30		1385.1				1257.4				59.8	1633.1	245.0
10	37		1408.9				189.6				1.2	153.6	23.6
*10	38		1413.5								300.6		726.2
11	47		1449.5								38.2		56.1
*11	48		1455.3								275.2		
12	57		1509.1								46.0		81.2
*12	58		1516.6								1106.4		
13	69		1599.5								201.2		
*13	70		1615.0								430.2		
14	81		1768.4								59.5		
*14	82		1755.4										
15	94										2.5		
*15	95												

Table 2: Solving times (in seconds), * means unsat, empty means timeout

8 Empirical Evaluation

In this section, we evaluate the performance of the different approaches. Each encoding is executed once for several board sizes. The amount of knights is varied in order to check both a satisfiable and unsatisfiable instance. All benchmarks were run on an AMD Opteron with 2 GHz and 4 physical cores. Up to 3 benchmarks were run in parallel. After 30 minutes without a result, the execution was aborted. Results are given in Table 2 and Fig. 6, showing the runtimes in seconds.

We benchmarked the CLP(FD) encoding introduced in Section 3 using SICStus Prolog. For $n = 5$ and 5 knights, solving takes $2.6s$. For $n = 6$ and 9 knights it takes only $2s$, but to determine that there is no solution for 10 knights it takes > 30 minutes. For $n = 7$ and 12 knights however, unsatisfiability is detected in $11s$. We did not manage to solve the full puzzle for $n = 8$ and 21 knights.

In summary, this encoding is more efficient than the higher level one written in B in Section 2, as the high level one can only solve the puzzle for $n = 5$. This shows that there is still scope to reduce the overhead of ProB's default CLP backend, given that it is based on CLP(FD) and SICStus as well. However, it is disappointing not to be able to solve the original puzzle for $n = 8$. In the future, we will investigate whether we can replace the `global_cardinality` constraint by a more effective encoding.

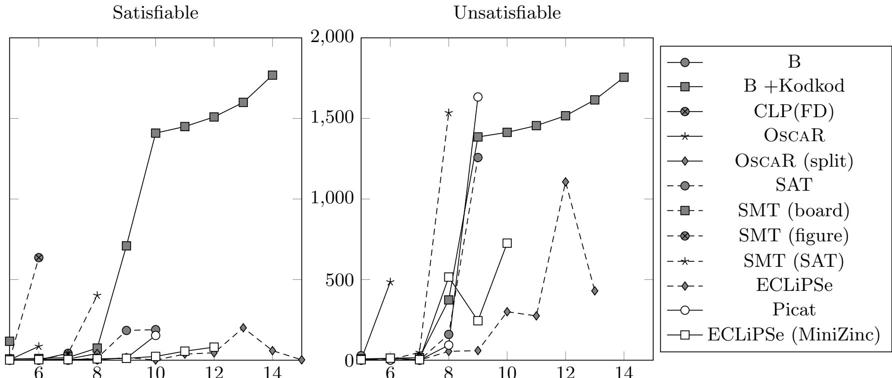


Fig. 6: Runtimes for Satisfiable and Unsatisfiable Instances

Furthermore, integrating ProB's default backend with other solvers is beneficial. Thanks to the improved integration with Kodkod presented in Section 2.5, ProB can solve all satisfiable and unsatisfiable instances with $n \leq 14$. Combining SAT and CLP proves to be stronger than both working independently.

In Section 4, we presented another attempt at a constraint programming solution to this problem; this time using the OscaR library in Scala. Unfortunately, the monolithic solution also does not scale to $n = 8$, but an optimized layered version, splitting the constraints and sacrificing completeness for efficiency, does scale (but is in principle not guaranteed to find a solution if there is one).

With respect to the search, the layered approach showed a superior performance. For $n = 5$, the execution time was roughly split in half, while for $n = 6$ the layered version was nearly 30 times faster. Until $n = 8$ the performance is very high in comparison, which is likely down to the small amount of backtracking that occurs during the search up to this point.

For the low-level encoding into SAT, we benchmarked using the winner of the SAT competition 2017, Maple [17]. Despite the blowup in complexity the resulting performance of a translation to SAT is often better than the one exhibited by our other encoding for large n. Both for satisfiable and unsatisfiable benchmarks Maple outperforms CLP(FD), ProB and OscaR. In particular, for $n = 8$ and 22 knights, the low-level SAT encoding using Maple reports unsatisfiability after $160.1s$. Furthermore, it can compute solutions for $n = 10$.

Our last approach, replacing the cardinality constraint by integer arithmetic, proves not to be an advantage over the plain SAT encoding. In fact, the context switch between SAT and Z3's arithmetic solver causes overhead and, in consequence, reduces performance. For $n = 8$, Z3 takes $402s$ to find a model. If the number of knights is increased to 22, Z3 detects unsatisfiability in $1535s$.

The related encodings discussed in Section 7 perform surprisingly well: Using the MiniZinc encoding we find a solution for $n = 8$ and 21 knights in $8.5s$. We are not sure what caused the speedup since the model was introduced in [6], but suspect a combination of improvements in ECLiPSe/eplex and faster CPUs.

The performance of the Picat and the MiniZinc model differs, although they are based on the same encoding. While Picat is slower for $n \leq 8$, it is faster for larger instances. The direct encodings in ECLiPSe and Picat show that tailoring towards the particular strengths of solvers is beneficial regarding performance. ECLiPSe outperforms the more general solution written in MiniZinc on the larger benchmarks. In particular, it is the only combination of encoding and solver able to solve the puzzle for $n \geq 14$. However, as argued above, the price is readability.

9 Conclusion, Outlook and Discussion

In conclusion, the crowded chessboard problem turned out to be surprisingly difficult to solve. The problem also allowed us to compare a variety of approaches.

- The high-level B model from Section 2.1 is a very readable, mathematical formulation of the problem. It cannot be solved for $n = 8$ using the current CLP(FD)-based backend of ProB, but it can be used to validate solutions found by other approaches. Other backends of ProB based on SAT [20] or SMT cannot be applied, due to the higher-order nature of the model.
- The lower-level, direct CLP(FD) encoding in Section 3 is faster, but also cannot scale to $n = 8$.
- The monolithic encoding using the OscaR constraint library in Section 4 also does not scale, but an optimized, layered version, sacrificing completeness for efficiency, does.
- Various attempts at a lower-level direct SMT-LIB encoding in Section 6, also cannot be solved for $n = 8$ using Z3.
- The high-level B model from Section 2.5 is slightly less declarative than the B model from Section 2.1, but scales surprisingly well.
- Among the fastest of our solutions is a direct SAT encoding, generated by a Python program in Section 5. However, this solution is hardest to read by a human: neither the SAT encoding nor the Python program are ideal for human validation and reviewing.

The ultimate goal is to be able to solve the high-level, readable model from Section 2.1, fully automatically. We hope that further refinements of the approaches in Section 2.5 and Section 4 will enable this, and pave the road for very declarative but tractable modelling of complex constraint problems.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality Networks and Their Applications. In *Proceedings SAT*, LNCS 5584, pages 167–180. Springer, 2009.
3. O. Bailleux and Y. Boufkhad. Efficient CNF Encoding of Boolean Cardinality Constraints. In *Proceedings CP*, LNCS 2833, pages 108–122. Springer, 2003.

4. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings CAV*, LNCS 6806, pages 171–177. Springer, 2011.
5. M. Carlsson and G. Ottosson. An Open-Ended Finite Domain Constraint Solver. In *Proceedings PLILP*, LNCS 1292, pages 191–206. Springer, 1997.
6. M. J. Chlond. IP modeling of chessboard placements and related puzzles. *INFORMS Transactions on Education*, 2(2):56–57, 2002.
7. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings TACAS*, LNCS 4963, pages 337–340. Springer, 2008.
8. D. Deharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In *Proceedings ABZ*, LNCS 7316, pages 194–207. Springer, 2012.
9. H. E. Dudeney. *Amusements in Mathematics*. 1917. Available at `https://www.gutenberg.org/ebooks/16713`.
10. N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proceedings SAT*, LNCS 2919, pages 502–518, 2003.
11. I. Hayes and C. Jones. Specifications are not (necessarily) executable. *Softw. Eng. J.*, 4(6):330–338, 1989.
12. J. N. Hooker. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*. John Wiley, New York, 2000.
13. S. Krings and M. Leuschel. Proof Assisted Symbolic Model Checking for B and Event-B. In *Proceedings ABZ*, LNCS 9675. Springer, 2016.
14. M. Leuschel. Formal Model-Based Constraint Solving and Document Generation. In *Proceedings SBMF*, pages 3–20, 2016.
15. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
16. M. Leuschel and D. Schneider. Towards B as a High-Level Constraint Modelling Language. In *Proceedings ABZ*, LNCS 8477, pages 101–116. Springer, 2014.
17. M. Luo, C.-M. Li, F. Xiao, F. Manyà, and Z. Lü. An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers. In *Proceedings IJCAI-17*, pages 703–711, 2017.
18. S. Merz and H. Vanzetto. Automatic Verification Of TLA$^+$ Proof Obligations With SMT Solvers. In *Proceedings LPAR-18*, LNCS, Mérida, Venezuela, 2012. Springer.
19. OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.
20. D. Plagge and M. Leuschel. Validating B, Z and TLA+ using ProB and Kodkod. In *Proceedings FM*, LNCS 7436, pages 372–386. Springer, 2012.
21. A. Savary, M. Frappier, and M. Leuschel. Model-Based Robustness Testing in Event-B using Mutation. In *Proceedings SEFM*, LNCS 9276, pages 132–147, 2015.
22. J. Schimpf and K. Shen. ECLiPSe - From LP to CLP. *TPLP*, 12(1-2):127–156, 2012.
23. D. Schneider, M. Leuschel, and T. Witt. Model-Based Problem Solving for University Timetable Validation and Improvement. In *Proceedings FM*, LNCS 9109, pages 487–495. Springer, 2015.
24. S. C. Shapiro. The Jobs Puzzle: A Challenge for Logical Expressibility and Automated Reasoning. In *AAAI SS*, 2011.
25. K. Shen and J. Schimpf. Eplex: Harnessing Mathematical Programming Solvers for Constraint Logic Programming. In *Proceedings CP*, LNCS 370, pages 622–636. Springer, 2005.
26. E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In *Proceedings TACAS*, LNCS 4424, pages 632–647. Springer, 2007.