Comparing the use of symmetry in constraint processing and model checking

Alastair Donaldson, Alice Miller, Muffy Calder
Department of Computing Science
University of Glasgow, Scotland
{ally,alice,muffy}@dcs.gla.ac.uk

Abstract

Both *model checking* and *constraint processing* involve the searching of graphs: in model checking to establish the truth of a temporal logic formula; in constraint processing to determine whether or not solutions to a problem exist which satisfy a set of constraints. In both fields, the presence of symmetry in the model or problem can result in redundant search over equivalent areas of the graph or search space. Recently there has been much interest in *symmetry reduction* in model checking, and *symmetry breaking* in constraint satisfaction problems (CSPs). The n-queens problem is a CSP to which symmetry breaking has been applied. To illustrate the approaches to exploiting symmetry in both model checking and CSPs we show how the n-queens problem can be solved using model checking, and how symmetry reduction can be used to make larger instances of the problem tractable. Through this example we highlight the similarities and differences between the approaches.

1 Introduction

Searching for *all* solutions of a constraint satisfaction problem (CSP) can take a prohibitively long time due to symmetry inherent in the problem [14]. Most of the search effort is dedicated to exploring variable assignments which are symmetrically equivalent to previous assignments, and are thus redundant. Similarly in model checking, the state space of a model may be intractably large, but consist of many symmetrically equivalent components which are indistinguishable in terms of the global behaviour of the system being modelled [7].

Recently there has been much research into *symmetry breaking* in constraint processing—adding constraints to a problem either before or during search to avoid the exploration of symmetrically equivalent assignments [12; 13; 14]. The goal of this approach is to quickly obtain only a subset of all solutions to the problem; the set of *all* solutions is obtained from this subset by applying symmetries. Similarly in model checking there has been significant interest in *symmetry reduction* [7; 10; 19]. If symmetries of a model can be identified then they can be exploited during model checking to construct a *quotient model*. The quotient model is generally smaller than the original model, but they are behaviourally equivalent in the sense that they satisfy the same set of (symmetrically invariant) properties. Thus model checking can be performed over the smaller quotient model.

The similarity between model checking and constraint processing has been investigated, and constraints technology has been applied to model checking problems [9; 11]. Model checking has been used to solve the *rehearsal problem* [16], generally accepted as a constraint satisfaction problem [21]. In this paper we illustrate the relationship between symmetry breaking in constraint programming and symmetry reduction in model checking, using the n-queens problem as a case study. We do not suggest that model checking is more suitable than constraints technology for solving this problem—we use the example as a way to demonstrate the theory of symmetry in model checking to the constraint programming community.

1.1 Overview of results

We present the n-queens problem using the Promela modelling language, and show how the SPIN model checker [17] can be used to find all solutions. We then show how symmetry reduction can be applied to ensure that only symmetrically distinct solutions are found. Through this example we show that model checking over a quotient Kripke structure is similar both to the method of symmetry breaking during search [13], and to the approach of finding the smallest solution in each equivalence class under an appropriate ordering [4] in constraint processing.

For all verification runs we used a PC with a 2.4GHz Intel Xenon processor, 3Gb of available main memory, running Linux (2.4.18), with SPIN version 4.0.7.

2 Preliminaries

We give a brief description of model checking in section 2.1, and of the SPIN model checker and the Promela modelling language in section 2.2. For a thorough introduction to model checking see e.g. [6], and for the definitive reference on SPIN and Promela see [17].

2.1 Model checking, Kripke structures and temporal logic

Model checking is a technique whereby temporal logic properties of a system can be checked by building an abstract model of the system and checking whether the model satisfies the properties. The model is constructed using a specification language, and checked using an automatic model checker. Failure of the *model* to satisfy a property of the system indicates either that the model does not accurately reflect the behaviour of the system, that the property has been inaccurately specified in temporal logic, or that there is an error (bug) in the original system. Examination of counter-examples provided by the model checker enable the user to either refine the model, refine the property, or, more importantly, to debug the original system.

To reason about a model, we refer to its underlying Kripke structure, which is defined over a set of atomic propositions. Atomic propositions are logical statements representing the values of variables in the model.

Definition 1 *Let AP be a set of atomic propositions. A* Kripke structure *$\mathcal{M}$ over AP is a quadruple $\mathcal{M} = (S, R, L, s_0)$ where:*

- *S is a non-empty, finite set of states,*
- *$R \subseteq S \times S$ is a* total *transition relation, that is for each $s \in S$ there exists $t \in S$ such that $(s, t) \in R$,*
- *$L : S \to 2^{AP}$ is a mapping that labels each state in S with the set of atomic propositions true in that state,*
- *$s_0 \in S$ is an initial state.*

In model checking, properties of models are usually specified in the temporal logic CTL*, or in one of its sub logics, LTL (linear temporal logic) or CTL (computation tree logic). In this paper we make use the SPIN model checker, which allows verification of LTL properties. The properties we are interested in use the operators $\Box$ (always), and $\Diamond$ (eventually). For further details on temporal logic see e.g. chapter 3 of [6].

2.2 Promela and SPIN

Promela is the specification language for the SPIN model checker [17]. Although similar in syntax to the C programming language, Promela includes constructs to specify non-determinism and concurrency in a model, but not many of the advanced features of the C language such as memory management functions. SPIN is a bespoke model checker for the Promela language. Given a Promela model and a temporal logic formula, SPIN attempts to show that the model *does not* satisfy the formula by finding a counter example path through the model which violates the formula. In SPIN, LTL properties are converted into Büchi automata [23], expressed in the form of *never-claims*. A never-claim can be thought of as an additional process which makes a transition every time one of the other process in the model has made a transition. If no such transition is possible (if the current path cannot possibly lead to an error for example) the current path is blocked, and the search backtracks.

Usually when a single error is found, model checking terminates so that the error can be resolved. In the approach we use here, we require *all* errors to be reported before model checking terminates. Conveniently SPIN provides an option which supports this and provides a separate counterexample for each error.

3 Symmetry in CSPs

A symmetry α for a constraint problem C is a function that maps any assignment A which is a solution of C to a symmetric assignment $\alpha(A)$ which is also a solution of C [1]. The set of all symmetries for a problem C forms a group, which partitions the set of all possible variable assignments into equivalence classes. In any class, either every assignment is a solution, or none is.

According to Gent and Smith [14], the three aims for a symmetry-exclusion method are:

- to guarantee that we never allow search to find two symmetrically equivalent solutions,
- to respect heuristic choice as much as possible, and
- to allow arbitrary forms of symmetry.

One approach to symmetry breaking in CSPs involves adding symmetry breaking constraints to the CSP before search, converting it into a less symmetric (ideally asymmetric) problem [8; 20]. Another approach, known as *symmetry breaking during search* (SBDS) involves adding symmetry breaking constraints to the problem as the search proceeds, by detecting symmetries which remain unbroken when the search tries a fresh branch on backtracking [13; 14]. A third approach [4] defines an ordering on the set of assignments and finds only the smallest solution in each equivalence class under this ordering.

4 Symmetry in model checking

While there are several approaches to symmetry breaking in CSPs, the vast majority of work on symmetry reduction in model checking centres around one approach. This approach uses symmetries of a Kripke structure to automatically construct a *quotient* structure.

Let $\mathcal{M} = (S, R, L, s_0)$ be a Kripke structure. An *automorphism* of $\mathcal{M}$ is a bijection $\alpha : S \to S$ which satisfies the following condition:

$$\forall s, t \in S, \quad (s, t) \in R \Rightarrow (\alpha(s), \alpha(t)) \in R.$$

In a model of a concurrent system with many replicated processes, Kripke structure automorphisms may involve the permutation of process identifiers throughout all states of the model. On the other hand, a model may include a data structure which has geometrical symmetry. In this case Kripke structure automorphisms involve applying the geometrical symmetries throughout all states of the model. All Kripke structure automorphisms arising in this paper are geometrical.

The set of all automorphisms of the Kripke structure $\mathcal{M}$ forms a group under composition of mappings. This group is denoted $Aut(\mathcal{M})$. A subgroup G of $Aut(\mathcal{M})$ induces an equivalence relation $\equiv_G$ on the states of $\mathcal{M}$ by the rule

$$s \equiv_G t \Leftrightarrow s = \alpha(t) \text{ for some } \alpha \in G.$$

The equivalence class under $\equiv_G$ of a state $s \in S$, denoted $[s]$, is called the *orbit* of s under the action of G. The orbits can be used to construct a *quotient* Kripke structure $\mathcal{M}_G$ as follows:

Definition 2 *The quotient Kripke structure $\mathcal{M}_G$ of $\mathcal{M}$ with respect to G is a quadruple $\mathcal{M}_G = (S_G, R_G, L_G, [s_0])$ where:*

- $S_G = \{[s] : s \in S\}$ *(the set of orbits of S under the action of G),*
- $R_G = \{([s],[t]) : (s,t) \in R\}$*,*
- $L_G([s]) = L(rep([s]))$ *(where $rep([s])$ is a unique representative of $[s]$),*
- $[s_0] \in S_G$ *(the orbit of the initial state $s_0 \in S$).*

In general $\mathcal{M}_G$ is a smaller structure than $\mathcal{M}$, but $\mathcal{M}_G$ and $\mathcal{M}$ are equivalent in the sense that they satisfy the same set of logic properties which are *invariant* under the group G (that is, properties which are "symmetric" with respect to G). For a proof of the following theorem, together with a description of the temporal logic CTL*, and sub logics LTL and CTL, see [6].

Algorithm 1 Algorithm to construct a quotient Kripke structure

```
$reached := \{rep(s_0)\}$
$unexplored := \{rep(s_0)\}$
while $unexplored \neq \emptyset$ do
  remove a state $s$ from $unexplored$
  for all successor states $q$ of $s$ do
    if $rep(q)$ is not in $reached$ then
      append $rep(q)$ to $reached$
      append $rep(q)$ to $unexplored$
    end if
  end for
end while
```

Theorem 1 *Let $\mathcal{M}$ be a Kripke structure, G be a subgroup of $Aut(\mathcal{M})$ and f be a CTL* formula. If G is an invariance group for all the atomic propositions p occurring in f, then*

$$\mathcal{M}, s \models f \Leftrightarrow \mathcal{M}_G, [s] \models f$$

where $\mathcal{M}_G$ is the quotient structure corresponding to $\mathcal{M}$.

Thus by choosing a suitable symmetry group G, model checking can be performed over $\mathcal{M}_G$ instead of $\mathcal{M}$, often resulting in considerable savings in memory and verification time [7].

Of course it must be possible to construct the structure $\mathcal{M}_G$ without first constructing $\mathcal{M}$. Assuming that symmetries of the model are known in advance, this construction can be achieved using algorithm 1, adapted from [19].

5 The n-queens problem

The n-queens problem [2] is as follows: "Given an $n \times n$ board of squares, place n-queens on the board so that no queen can attack any other queen". The problem is a generalisation of the 8-queens problem (the case where $n = 8$). A solution to the problem is a configuration of the board which satisfies the requirement that no two queens can attack one another. Figure 1 shows two solutions to the 4-queens problem. Figure 3 on the other hand shows two configurations of the 4-queens problem which are not solutions.

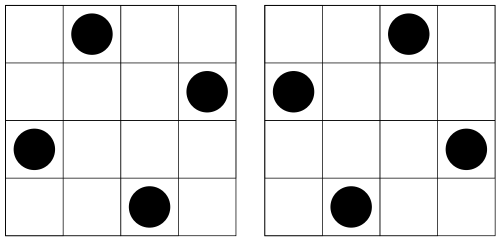


Figure 1: Symmetric solutions to the 4 queens problem.

5.1 Representing n-queens as a CSP

The standard formulation of n-queens as a constraint satisfaction problem uses n variables, $q_1, q_2, \ldots, q_n$, where the position of the queen on row i of the board is represented by q_i. The domain of each of the variables $q_1, q_2, \ldots, q_n$ is $\{1, 2, \ldots, n\}$. For all $i, j = 1, 2, \ldots, n,\ i \neq j$, the following constraints ensure that no queen can attack any other:

$$q_i \neq q_j \quad (1)$$
$$q_i + i \neq q_j + j \quad (2)$$
$$q_i - i \neq q_j - j \quad (3)$$

Constraint 1 ensures that the positions of queens in different rows must be different, so that queens cannot attack one another by moving vertically on the board. The requirement that no two queens can appear on the same diagonal is specified by constraints 2 and 3. No constraint is required to ensure that only one queen appears on each row: formulating the problem using one variable per row means that this requirement is implicitly satisfied.

5.2 Representing n-queens in Promela

We follow an approach similar to the one presented in [16] for solving a constraint satisfaction problem using model checking. Recall from section 2.1 that model checking is useful for finding errors in models of systems. To find all solutions to a CSP using model checking we write a model of the problem, and assert that no solutions to the problem exist. The model checker then regards a solution as an error, and reports it.

The approach involves first writing a verification model which non-deterministically assigns values to each variable in the problem, so that the state space of the model includes all possible assignments to variables. A temporal logic formula asserting that the constraints of the problem are *not* satisfied in *any* state of the model must then be specified. Finally the model checker must be instructed to verify the formula, reporting all errors paths. All solutions to the problem will thus be reported as errors by the model checker.

Our model of the n-queens problem uses a byte array of length n, indexed from 0 to $n-1$. Each entry of this array represents a row of the board, and is set to u (unassigned) if no queen has been placed on that row. Setting entry $j-1$ to the value $i-1$ implies that a queen has been placed at row j, column i $(1 \leq i, j \leq n)$. Here is the Promela code for our model with $n = 4$:

```
#define u 255
byte q[4]=u;
bit result=0;
```

```
init {
  do
    :: result==1 -> skip
    :: else ->
       atomic { if
        :: q[0]==u ->
           if :: q[0]=0 :: q[0]=1 :: q[0]=2 :: q[0]=3 fi
        :: q[1]==u && q[0]!=u ->
           if :: q[1]=0 :: q[1]=1 :: q[1]=2 :: q[1]=3 fi;
           diagscheck(1,result);
           alldifferentcheck(1,result)
        :: q[2]==u && q[1]!=u && q[0]!=u ->
           if :: q[2]=0 :: q[2]=1 :: q[2]=2 :: q[2]=3 fi;
           diagscheck(2,result);
           alldifferentcheck(2,result)
        :: q[3]==u && q[2]!=u && q[1]!=u && q[0]!=u ->
           if :: q[3]=0 :: q[3]=1 :: q[3]=2 :: q[3]=3 fi;
           diagscheck(3,result);
           alldifferentcheck(3,result)
       fi }
  od
}
```

The keyword *atomic* ensures that the code enclosed in the following curly braces is treated as one step, so that a single state change occurs on execution of the enclosed block. The *do :: Seg od* construct causes the code segment *Seg* to be unconditionally repeated, and the construct *if* Seg_1 *::* ... *::* Seg_m *fi* allows one of the code segments Seg_i to be executed nondeterministically if it is executable ($1 \leq i \leq m,\ m > 0$).

To make the problem tractable our code uses two simple optimisations. The rows are assigned in order: $q[j]$ cannot be set until $q[k]$ has been set for all $k < j$ ($0 \leq j < n$). This is achieved by the boolean guards of the form

```
q[j]==u && q[j-1]!=u && ... && q[0]!=u
```

This optimisation ensures that there is exactly *one* path through the Kripke structure leading to each total variable assignment—without this optimisation there would be multiple paths to each solution as illustrated by figure 2, so the model checker would report the same solution many times. Note that the rows could be assigned in any fixed order.

The second optimisation is that on assigning each variable, a check is made to see whether or not the partial assignment satisfies the diagonal constraints and the column constraint. This optimisation makes use of two functions, *diagscheck(k,result)* and *alldifferentcheck(k,result)*. The function *diagscheck(k,result)* sets *result* to 1 if the assignment just made to row *k* breaks the diagonal constraints. This forces the search to backtrack if a partial assignment cannot be a solution. The function *alldifferentcheck(k,result)* works similarly for the column constraint. This optimisation dramatically reduces the state space of the model.

The following LTL formula is used to find solutions to the problem:

$$\neg\Big(\Diamond\Big(\Box(\text{finished} \wedge \text{alldifferent} \wedge \text{updiag} \wedge \text{downdiag})\Big)\Big)$$

Here *alldifferent*, *updiag* and *downdiag* correspond to constraints 1, 2 and 3 in section 5.1 respectively. The proposition *finished* is used to ensure that only total assignments are considered as solutions.

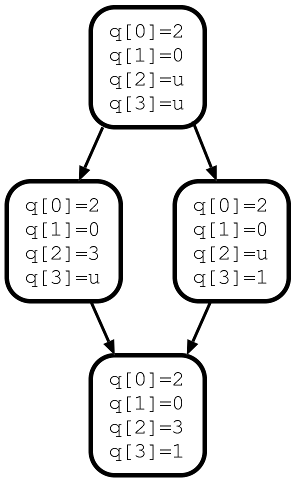


Figure 2: Two paths to the same solution. Our model ensures that only the left hand path is explored.

$$\begin{aligned}
\text{alldifferent} &= \bigwedge_{i \neq j} (q[i] \neq q[j]) \\
\text{updiag} &= \bigwedge_{\substack{0 \leq i,j < n \\ i \neq j}} (q[i] + i \neq q[j] + j) \\
\text{downdiag} &= \bigwedge_{\substack{0 \leq i,j < n \\ i \neq j}} (q[i] - i \neq q[j] - j) \\
\text{finished} &= \bigwedge_{0 \leq i < n} (q[i] \neq u)
\end{aligned}$$

Recall from section 5.1 that in modelling the problem using one variable for each row we eliminate the need for a constraint specifying that there should be no more than one queen on each row. In English, the above LTL formula asserts that "at every state in the model, for all paths, it is not true that the constraints eventually always hold". A violation of this property clearly must be a solution to the problem. The $\Box$ operator is necessary to ensure that duplicates of solutions are not reported, due to the way SPIN verifies LTL formulae.[1]

We have written a PERL script which generates a Promela model and never claim for the n-queens problem, given $n > 0$. The code for this script is available on our website [5]. Table 1 shows the number of solutions to the n-queens problem found by our setup for $n \leq 10$, along with the time and memory requirements. For $n \leq 9$, all solutions are found. It was not possible to find any solutions for $n = 10$ before available main memory was exhausted.[2] This shows that although the

[1] In SPIN all finite paths are extended to infinite paths by the creation of a loop from a final state to itself. As we have asked the model checker to report *all* errors, without the $\Box$ operator a different error would be reported depending on how many times this loop is executed, thus the same solution would be returned multiple times.

[2] We could use the minimised automaton method of SPIN [17] to

Table 1: Verification results for the n-queens Promela model without symmetry reduction

n	**solutions found**	**states searched**	**time**	**memory (Mb)**
4	2	332	0.01 s	2.3
5	10	875	0.01 s	2.3
6	4	14,850	0.05 s	2.7
7	40	121,282	0.41 s	6.4
8	92	1.11×10^6	4.50 s	43.9
9	352	1.12×10^7	1 m 47 s	428.3
10	0	7.70×10^7	4 m 31 s	3039.0

problem can in principle be solved using model checking, it is not an effective approach.

6 Symmetry in the n-queens problem

Consider the solution to the 4-queens problem shown on the left in figure 1. Observe that flipping the board through the vertical axis results in the solution shown on the right. These solutions are said to be *symmetric*, the symmetry being reflection through the vertical axis.

In general there are 8 symmetries of the n-queens problem for any n. Geometrically these symmetries are: the identity, rotation by 90°, 180° and 270°, reflection through the vertical axis, and reflection through the vertical axis combined with rotation through 90°, 180° or 270°. These symmetries form a group (called the dihedral group of order 8) that preserves solutions to the n-queens problem. The solution preserving property is illustrated by both figure 1 which shows symmetric solutions, and figure 3 which shows two symmetric assignments which are not solutions.

Symmetry in the 8-queens problem was investigated as early as 1874 by Glaisher [15].

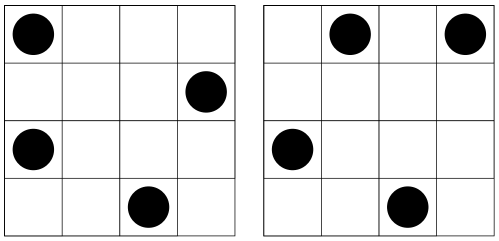


Figure 3: Symmetric configurations of the 4 queens problem which are not solutions.

6.1 Symmetry breaking in the CSP approach

The n-queens problem has been used as an example by various authors investigating symmetry in constraint processing [4; 14]. Three methods of symmetry breaking which have been used are symmetry breaking before search, symmetry breaking during search, and symmetry breaking by assignment ordering. We briefly summarise the advantages and disadvantages of these techniques for the n-queens problem.

Symmetry breaking before search

It is possible to add symmetry breaking constraints to a CSP representation of the n-queens problem before search to reduce the number of symmetrically equivalent solutions that are reported [14; 22]. For example, to eliminate 180° rotational symmetry we could add constraints in the following manner [14]:

- $q_1 \leq n + 1 - q_n$
- if $q_1 = n + 1 - q_n$ then $q_2 \leq n + 1 - q_{n-1}$
- if $q_1 = n + 1 - q_n$ and $q_2 = n + 1 - q_{n-1}$ then $q_3 \leq n + 1 - q_{n-2}$

etc. This method requires a large number of constraints to be added to the problem if all symmetries are to be broken. Breaking all symmetries in practice can prove difficult [22], and symmetry breaking constraints do not always work well with variable ordering heuristics, especially if only one solution is required [14].

Symmetry breaking during search

In [14], the method of symmetry breaking during search is applied to the n-queens problem, using ILOG Solver. Seven functions are written, one for each non-trivial symmetry of the problem. Run time is shown to be cut by up to 75%, and the number of backtracks is greatly reduced as n increases. All symmetry of the problem is exploited, so that all the solutions found are symmetrically distinct. Since the number of symmetries of the problem is small, and does not change as n increases, the SBDS approach is very effective when applied to the n-queens problem [14].

Symmetry breaking by assignment ordering

A symmetric backtracking algorithm with symmetry is presented in [4]. Given a symmetry group, the algorithm works by imposing an ordering on variable assignments, and uses a function, *Symtest*, to check for each partial assignment whether or not it is equivalent to a smaller one using the symmetry group.

Experimental results using the n-queens problem show that for $n \leq 6$ the overhead of symmetry computations means the algorithm using symmetry takes longer than one without symmetry, but for larger values of n the algorithm with symmetry is 2.6 times faster than without. The algorithm with symmetry could also handle cases of the problem which were not tractable without symmetry.

6.2 Symmetry reduction in the model checking approach

We show how the approach to symmetry reduction using quotient structures can be applied to the Promela model described in section 5.2. Definition 2 of a quotient Kripke

try to verify larger configuration. This method trades memory for time. While it can handle extremely large state spaces, verification can be very slow.

structure, and algorithm 1 for constructing a quotient Kripke structure, both make use of a function *rep* which, given a state s, computes a unique representative from the equivalence class $[s]$ such that for all $t \in [s]$, $rep(t) = rep(s)$. It would be possible to modify the SPIN search algorithm to store a unique representative of each state, and this is done in [3] for total symmetries. However, such a modification to SPIN would be quite tricky, and would tie the symmetry reduction technique to the current version of the model checker. For this reason we alter our model so that once a state has been reached, but *before* SPIN has stored the state, the state is *canonicalised*, i.e. it is converted into a unique representative from its equivalence class. This has *exactly* the same effect as altering the SPIN search according to algorithm 1. The Promela code for the symmetry reduced model is the same as that given in section 5.2, with a call to the function *rep* added as follows:

```
init {
  do
    :: result==1 -> skip
    :: else ->
       atomic { if
        :: q[0]==u ->
          .....
       fi;
       rep() }
  od
}
```

The function *rep* applies each symmetry of the problem to the array q in turn, and selects (as a representative) the assignment which is smallest using the natural lexicographical ordering on the array. This is similar to the approach used in [4] of assigning an ordering on assignments and returning only the smallest solution in each equivalence class under this ordering. Here is an excerpt of the Promela code:

```
inline rep() {
  d_step {
    copy(q,min); wipe(temp); r90(q,temp);
    if
      :: lt(temp,min) -> copy(temp,min) :: else -> skip
    fi;
    wipe(temp); r180(q,temp);

    if
      :: lt(temp,min) -> copy(temp,min) :: else -> skip
    fi;
    wipe(temp); r270(q,temp);

    .....

    copy(min,q);
  }
```

As can be seen from the code, there is a function for each symmetry in the problem. The function *r90(a,b)*, which turns b into a configuration of a rotated through 90°, is implemented (for the case $n = 4$) as follows:

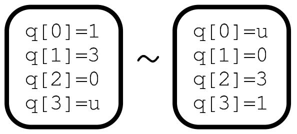
```
inline r90(a,b) {
  if
    :: a[0]==u
    :: else b[a[0]]=n-1
  fi;
  if
    :: a[1]==u
    :: else b[a[1]]=n-2
  fi;
  if
    :: a[2]==u
    :: else b[a[2]]=n-3
  fi;
  if
    :: a[3]==u
    :: else b[a[3]]=n-4
  fi;
}
```

Table 2: Verification results for the n-queens problem with symmetry reduction

n	solutions found	states searched	time	memory (Mb)
4	1	137	0.01 s	2.3
5	2	742	0.01 s	2.3
6	1	4,901	0.01 s	2.4
7	6	37,994	0.18 s	3.5
8	12	337,070	1.61 s	14.5
9	46	3.35×10^6	23.06 s	129.1
10	92	3.70×10^7	17 m 20 s	1478.7
11	46	6.93×10^7	6 m 33 s	3039.0

The others functions are defined in a similar way. The function *copy(a,b)* makes copies the array a into the array b. This has been implemented since Promela does not support the operation of assigning one array to another. Similarly, *wipe(a)* sets each element of the array a to zero. The function *lt(a,b)* returns true if array a is lexicographically smaller than array b.

Providing a function for each nontrivial symmetry of the problem is similar to the way SBDS is applied to the problem in [14].



```
q[0]=1        q[0]=u
q[1]=3   ~    q[1]=0
q[2]=0        q[2]=3
q[3]=u        q[3]=1
```

Figure 4: These assignments are symmetrically equivalent, but the one on the right is not reachable in the original model.

An important point to note is that our model is only partially symmetric. We assign values to the array positions $q[0], q[1], \ldots, q[n-1]$ in a fixed order, and so the configuration shown on the left in figure 4 is a reachable configuration, but the one on the right is not. However, the two configurations are symmetrically equivalent. This means that the original model is not *closed* under the symmetries of the problem: given a state and a symmetry it may be that the state resulting from applying the symmetry is not a reachable state of the model. The work of Emerson and Trefler [10] on *rough symmetry* in model checking ensures that in this case the quotient model obtained by applying symmetries of the problem is still behaviourally equivalent to the original model for symmetric properties even though the symmetry is only partial. The justification for this is that each variable $q[i]$ $(0 \leq i < n)$ can be seen as a prioritised process which assigns itself a value, such that process $q[i]$ is blocked until process $q[j]$ has assigned itself a value for each $j < i$. For more details of symmetry in prioritised systems see [10].

We have extended our PERL script to allow the inclusion of symmetry reduction in the generated Promela model. Table 2 shows the number of solutions to the n-queens problem found by our model using symmetry reduction for $n \leq 11$, along with the time and memory requirements. Since all symmetries of the problem have been exploited, only symmetrically distinct solutions are returned. For $n \leq 10$, all symmetrically distinct solutions are found. Despite the overhead of the *rep* function, using symmetry reduction is always at least as fast as not using it. For the case $n = 8$, using symmetry reduces the number of explored states by a factor of more than 3, and search is 4.7 times faster. We were also able to find all (symmetrically distinct) solutions for the case $n = 10$, which was not possible without exploiting symmetry. For the case $n = 11$, it was possible to find 46 of the 341 symmetrically distinct solutions before memory was exhausted.

Recall from section 6.1 the three aims of a symmetry exclusion method proposed by Gent and Smith [14]. We have demonstrated that symmetry reduction using quotient structures fulfills the first and last of these aims—this approach has avoided finding symmetrically equivalent solutions, and we could deal with different forms of symmetry in other problems by coding the *rep* function differently. In fact it would be possible to automatically generate the *rep* function given generators of a symmetry group, using a similar approach to [4]. The second aim, that heuristic choice should be represented, is not applicable in this situation since search heuristics are not usually used in model checking (this is one of the reasons that model checking performs so poorly in solving the n-queens problem). Although symmetry reduction has made larger instances of the problem tractable, the largest instance we can handle using model checking is still much smaller than the largest instances successfully handled by other approaches [4; 14].

7 Related work

Symmetry reduction using SPIN is applied to a model of the game *Tic Tac Toe* in [18]. The symmetries of this model are the same as the rotation and reflection symmetries of the n-queens problem. Symmetry reduction is implemented in a way similar to our approach, by incorporating a canonicalisation function into the model rather than changing the SPIN model checker. In [16], model checking with SPIN is used to solve the *rehearsal problem*, a constraint satisfaction problem. Symmetry is exploited in the model by adding additional constraints rather than by building a quotient structure of unique representatives. This is analogous to the technique of symmetry reduction before search in CSPs, discussed in section 6.1

In order to exploit the partial symmetry in our Promela verification model we use the results of Emerson and Trefler [10] on systems of identical processes which are "nearly" symmetric. Emerson and Trefler have also worked on the problem of exploiting symmetry at the *source code level*, translating a symmetric program into a reduced program *before* model checking, then model checking the reduced program. It would be interesting to compare this with the approach of converting a symmetric CSP into an asymmetric one before search to avoid duplicate solutions [20; 8].

8 Conclusions and future work

We have presented a Promela verification model to find all solutions of the n-queens problem, and have applied symmetry reduction techniques to speed up search, and to make larger instances of the problem tractable. We have also summarised approaches to symmetry breaking for this problem in constraint processing. Through this example it is clear that symmetry reduction in model checking shares similarities with SBDS in constraint programming, in that functions must be provided for each nontrivial symmetry in the problem, and these functions are used during search to avoid the exploration of symmetrically equivalent portions of the search space. The method of symmetry breaking by assignment ordering [4], which returns only the smallest solution in each equivalence class under a defined ordering, is also similar to symmetry reduction using quotient structures. The states of a quotient structure consist of a unique representative from each equivalence class of the original structure under the symmetry group, and we have followed the common approach of using the lexicographically smallest state from each class as a representative [3].

In this paper we have concentrated on the concepts of symmetry reduction and symmetry breaking rather than on the theoretical details. However, we intend to formalise our observations at a theoretical level, and try to establish exactly what the correspondence is between symmetry breaking in constraint processing and symmetry reduction in model checking.

It would be possible to apply the approach of symmetry breaking *before* search to our verification model. Although this technique has been used in model checking for specific examples [16], to our knowledge applying this idea to model checking has not been studied for arbitrary forms of symmetry. Emerson and Trefler [10] have investigated symmetry reduction at the source code level for *fully* symmetric systems. We intend to investigate this further.

One of the reasons that model checking performs so poorly in solving constraint satisfaction problems is that it stores the entire search space in main memory. This is because Kripke structures typically have cycles leading back to old states. However, in our formulation of this problem it is clear that the underlying Kripke structure is a tree (apart from the stuttering extension of final states), so it would be possible to use stateless search to find solutions. Although a stateless variation of the SPIN search algorithm is discussed in [17], it is not currently implemented in SPIN.

An important problem in using symmetry with model checking and constraint processing is *symmetry detection*, that is determining symmetries of a models or problems from their specifications. The challenge is to allow symmetry to be exploited without requiring knowledge of group theory on the part of the modeller. Ip and Dill proposed a new data type called *scalarset* which makes the specification and detection of symmetries trivial for models of concurrent systems [19]. It would be interesting to try to apply the notion of a scalarset to constraint satisfaction problems as a way to specify problem symmetries easily.

References

[1] Rolf Backofen. Symmetries: To break or not to break, that's the question. (extended abstract). In *Proceedings of the Joint Annual Workshop of the ERCIM Working Group on Constraints and the CoLogNET area on Constraint and Logic Programing*, Budapest, Hungary, 2003.

[2] R.A. Bosch. Peaceably coexisting armies of queens. *Optima (Newsletter of the Mathematical Programming Society)*, 62:6–9, 1999.

[3] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. *International Journal on Software Tools for Technology Transfer*, 4(1):65–80, 2002.

[4] C.A. Brown, L. Finkelstein, and P.W. Purdom. Backtrack searching in the presence of symmetry. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 of *Lecture Notes in Computer Science*, pages 99–110, 1988.

[5] M. Calder and A. Miller. Veriscope publications website: `http://www.dcs.gla.ac.uk/research/veriscope/publications.html`.

[6] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Masachusetts, 1999.

[7] E.M. Clarke, R. Enders, T. Filkhorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1–2):77–104, 1996.

[8] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings of Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 149–159, 1996.

[9] G. Delzanno and A. Podelski. Model checking in CLP. In *tacas99*, pages 223–239, 1999.

[10] E. Allan Emerson and Richard J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In Laurence Pierre and Thomas Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 142–156, Bad Herrenalp, Germany, September 1999. Springer-Verlag.

[11] L. Fribourg. Constraint Logic Programming Applied to Model Checking. In Annalisa Bossi, editor, *Logic Programming Synthesis and Transformation, 9th International Workshop, LOPSTR'99*, volume 1817 of *Lecture Notes in Computer Science*, pages 30–41, Venezia, Italy, September 2000. Springer.

[12] I. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic SBDD using computational group theory. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming 9th International Conference - CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 333–347, Kinsale, Ireland, September 2003. Springer.

[13] I.P. Gent, W. Harvey, and T. Kelsey. Groups and constraints: Symmetry breaking during search. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming 8th International Conference - CP 2002*, volume 2470 of *Lecture Notes in Computer Science*, pages 415–430, Ithaca, NY, USA, September 2002. Springer.

[14] I.P. Gent and B. Smith. Symmetry breaking in constraint programming. In W. Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence, ECAI 00*, pages 599–603, Berlin, Germany, August 2000. John Wiley and Sons, Chichester.

[15] J.W.L. Glaisher. On the problem of the eight queens. *Philosophical Magazine*, 4(48):457–467, 1874.

[16] P. Gregory, A. Miller, and P. Prosser. Solving the rehearsal problem with planning and with model checking. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004), workshop W14: Modelling and Solving Problems with Constraints*, Valencia, Spain, 2004. to appear.

[17] Gerard J. Holzmann. *The SPIN model checker: primer and reference manual*. Addison Wesley, Boston, 2003.

[18] G.J. Holzmann and R. Joshi. Model-driven software verification. In Susanne Graf and Laurent Mounier, editors, *Proceedings of the 11th International SPIN Workshop (SPIN 2004)*, volume 2989 of *Lecture Notes in Computer Science*, pages 76–91, Barcelona, Spain, April 2004. Springer-Verlag.

[19] C.Norris Ip and D. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.

[20] J.F. Puget. On the satisfiability of symmetrical constrainted satisfaction problems. In J. Komorowski and Z.W. Ras, editors, *Methodologies for Intelligent Systems (Proceedings of ISMIS '93)*, volume 689 of *Lecture Notes in Artificial Intelligence*, pages 350–361, 1993.

[21] B. Smith. Constraint programming in practice: Scheduling a rehearsal. Technical Report APES-67-2003, APES resesarch group, 2003.

[22] Barbara Smith. Personal communication, 2004.

[23] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings, Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society, June 1986.