ProB gets Nauty: Effective Symmetry Reduction for B and Z Models

Corinna Spermann and Michael Leuschel
Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{spermann,leuschel}@cs.uni-duesseldorf.de

Abstract

Symmetry reduction holds great promise to counter the state explosion problem. However, currently it is "conducting a life on the fringe", and is not widely applied, mainly due to the restricted applicability of many of the techniques. In this paper we propose a symmetry reduction technique applied to high-level formal specification languages (B and Z). Not only does symmetry arise naturally in most models, it can also be exploited without restriction by our method. This method translates states of a formal model into directed graphs, and then uses graph canonicalisation to detect symmetries. We use the tool NAUTY *to efficiently perform graph canonicalisation, which we have interfaced with the model checker* PROB.

In this paper we present the general technique, show how states can be translated first into vertex-coloured graphs suitable for NAUTY. *We present empirical results, showing the effectiveness of our method as well as analysing the cost of graph canonicalisation.*

Keywords: *B-Method, Tool Support, Model Checking, Symmetry Reduction.*[1]

1. Introduction

The B-method [1] is a theory and methodology for formal development of computer systems based on set theory and predicate logic. It is used in industry in a range of critical domains and is supported by a variety of tools such as Atelier-B, the B-toolkit and B4Free.

The PROB [6] animator and model checker is complementary to the traditional B tools and is particularly useful to provide a quick validation and debugging support prior to the generally time consuming work of developing formal proofs.

Model checking means that the invariant of a specification, also called model, is tested if it is true in every reachable state. That is, every predicate of the invariant is evaluated for each state. It is well known that model checking suffers from the exponential state explosion problem; one way to combat this is via *symmetry reduction* [3, 12]. Indeed, often a system to be checked has a large number of states with symmetric behaviour, meaning that there are classes of states where each member of a class behaves like every other member of that class. Symmetry is particularly prominent in B because of the use of *deferred sets*. A deferred set is in essence a user-defined data type, whose cardinality and individual elements are left open. This means that two elements of the same deferred set can be exchanged without affecting the truth-value of predicates.

In previous work we have introduced symmetry reduction for model checking of B machines:

- In [7] we have presented the theoretical underpinnings of symmetry reduction for B, along with correctness results. We also presented a new symmetry reduction technique, called permutation flooding, which ensures that only one representative per symmetry group is examined. The advantage of the method is its simplicity and its ability to deal with complicated symbolic datastructures. Its disadvantage is that it usually cannot yield an exponential reduction in complexity, as the number of stored states is not reduced.
- In [15] we have pursued another approach, using graph isomorphism to detect symmetries. We developed a *canonicalisation function* for B states viewed as vertice- and edge-coloured graphs, i.e. a procedure which maps each state to a unique member of its equivalence class, called the *canonical form*.

The starting point of this paper is that the practical performance of [15] is rather disappointing. While it is still more efficient than model checking without symmetry, it is often slower than [7], and usually fails to deliver the potential exponential reduction of complexity. A crucial question is whether this is inherent because of the complexity of detecting graph isomorphism, or whether it is simply due to an

[1]This research is partially supported by the EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

inefficent graph canonicalisation implementation. It is this question we aim to answer in this paper. We show how the well known NAUTY package [10] can be used to check symmetry for B machines, by translating coloured state graphs into uncoloured graphs suitable for NAUTY. We have then implemented this translation and written a C library that allows to integrate NAUTY within PROB. We then provide a thorough empirical investigation, comparing this approach with our three existing approaches.

2. Background: Viewing States as Graphs

A model checker (potentially) has to examine every reachable state of a specification. The idea of symmetry reduction is that among the reachable states there are many distinct states which are symmetric wrt each other, i.e., they are virtually indistinguishable as far as the behaviour of the specification is concerned.

The orbit of a state s is the set of all states which are symmetric to s. The *orbit problem* is deciding whether two states s, s' are in the same orbit. This often amounts to checking whether there exists a permutation of datavalues (within some given symmetry group) which transforms s into s'. The orbit problem is the cornerstone of model checking with symmetry: every time a model checker with symmetry encounters a new state s, it needs to check whether there exists an already treated state s' which is in the orbit of s.

For the B-method, symmetries are induced by deferred sets. Take for example the following excerpt from a B Machine:

```
MACHINE Sym
SETS Trains; Tracks
VARIABLES pos, stopped
INVARIANT pos: Trains +->Tracks
          & stopped <: Trains
```

Here Trains and Tracks are two deferred sets: the cardinality of the sets is not specified and the elements of the sets are not enumerated and given no name. Hence, there is no way to directly refer to the elements of those sets in B predicates or expressions. This allowed us to prove in [7], that given a state s we could permute deferred set elements for each other, yielding s', without changing the truth value of any B predicate. Also, the value of an expression in s' would simply be the permuted value of the expression in s'. For example, given the deferred sets $Trains = \{thomas, gordon\}$, $Tracks = \{t1, t2, t3\}$ (instantiated for model checking purposes) and given the state $s \equiv (pos = \{thomas \mapsto t2\} \wedge stopped = \{thomas\})$, the state $(pos = \{gordon \mapsto t1\} \wedge stopped = \{gordon\})$ is in the orbit of s, but $(pos = \{gordon \mapsto t1\} \wedge stopped = \{thomas\})$ is not.

The crucial question now is: how do we solve the orbit problem as efficiently as possible. An essential point of our approach is the translation of individual states of a B machine into *state graphs*. Indeed, binary relations are at the heart of the B method (and are used to represent more complicated data structures such as functions and sequences). Binary relations can be translated into directed coloured graphs in a natural way, thus translating the orbit problem of symmetry reduction into graph isomorphism (see also [4, 12] or Section 14.4.1 of [3]). This will allow us to reuse many years of research which have lead to efficient algorithms for graph isomorphism.

Take for example a relation $v \in S \leftrightarrow T$. The state graph for a state where $v = \{(s_0, t_0), \ldots, (s_n, t_m)\}$ is depicted in Fig. 3. In this graph, the value of the relation, v is represented by edges that indicate specific ordered pairs, whose edge labels denote the variable they encode. A special 'root' vertex is also present, whose use will be explained later in this section. Note that the nodes of the graph are coloured as well: each deferred set is assigned its own unique colour. Furthermore, all elements of enumerated sets, as well as all other non-symmetric elementary datavalues (integers, booleans) get their own unique colour.

What about variables which are not of type relation? Let us first discuss simpler data types. Such values in B are either elements of basic sets (including Boolean values and integers), or sets of values. For an element of a set, $v \in S$, where $v = s_0$, we have the graph in Fig. 1. The graph of a set, $v \in \mathbb{P}(S)$, where $v = \{s_0, \ldots, s_n\}$ is shown in Fig. 2. Note that both representations now use the special 'root' node. Also, although our graph representation does not distinguish $v = \{s_0\}$ from $v = s_o$, the B type system does and we only work with well-typed machines (typing is decidable in B).

For more complicated, nested data types, we have to introduce intermediary nodes into the graph; one for each nested value. Full details about this translation have been presented in [15], and further details can be found in [14]. By composing the individual graphs that represent each value of a variable in a state, we obtain its *state graph*. Let $\mathcal{G}$ denote the function translating a state to its state graph. As an example, the graphical form of the state, $\langle v_1 = \{(\{s_0\}, \{s_1\})\}, v_2 = \{\{s_2\}\}\rangle$ is given in Fig. 4.

3. Using NAUTY to detect symmetry for B

In the previous section we described how B-states can be transformed into graphs. Now we are going to concentrate on the graph isomorphism problem for B state graphs, which are vertice- and edge- coloured graphs.

To decide whether two graphs are isomorphic we have implemented in [15] a procedure to compute a *canonical label* for both. Canonical labelling functions find a

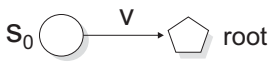

Figure 1. Graph for an atom

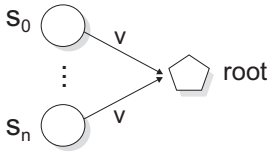

Figure 2. Graph for a set

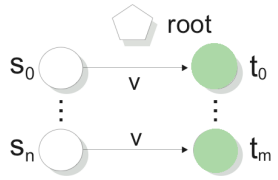

Figure 3. Graph for pairs

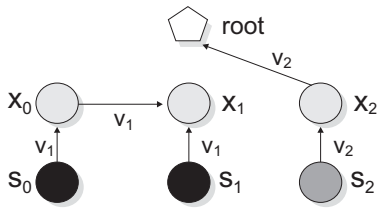

Figure 4. The state graph $\mathcal{G}$ ($\langle v_1 = \{(\{s_0\},\{s_1\})\}$, $v_2 = \{\{s_2\}\}\rangle$).

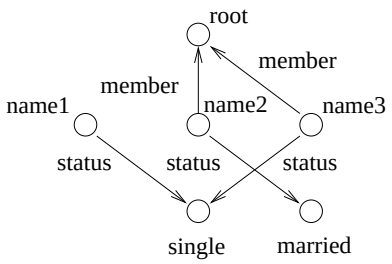

Figure 5.

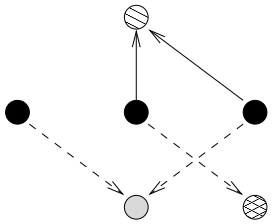

Figure 6.

uniquely determined label for a graph, which is the same for all symmetric graphs. These algorithms rely on the permutation of graph vertices. Let us denote all vertex permutations with the relation π. Now, consider a graph, g with vertices $V = \{v_1, v_2, \ldots, v_n\}$ and a sequence of vertices $v_1, v_2, \ldots, v_n$. All possible vertex orderings are $\Pi = \{o \mid o = v_1^\pi, v_2^\pi, \ldots, v_n^\pi\}$, each of which corresponds to an adjacency matrix that encodes the graph; thus, Π is an ordered set. Typically, an implementation of the algorithm will compute a subset of Π and will choose the least element as the canonical label. This implementation was done in Prolog, and was an extension of the core algorithm of [11] for vertice- and edge-coloured graphs.

Now we use the implementation of the canonical labelling algorithm from the NAUTY package [10] directly. Since NAUTY can handle only vertice-coloured graphs we need to transform our vertice- and edge-coloured graphs into graphs with only labels on the vertices. We show here briefly how we do this. Consider a machine with a dereferred set NAME, a given set MARITAL_STATUS $= \{single, married\}$, a set variable *member*, and a relation *status*. A valid state of this machine is

$$\begin{aligned} &member = \{name2, name3\},\\ &status = \{(name1 \mapsto single), (name2 \mapsto married),\\ &\qquad (name3 \mapsto single)\} \end{aligned}$$

We have already seen in Section 2 how such a state is transformed into a graph with labels on both nodes and edges, yielding the graph in Fig. 5.

Considering that the set NAME is dereferred, we get the vertice- and edge-coloured graph in Fig. 6. Note, we gave the nodes a different shade for different colours and solid and dashed arrows to distinguish the set variable *member* from the relation *status*.

Now we need to transform the vertice- and edge-coloured graph into an only vertice-coloured graph, before it can be handed over to NAUTY. The NAUTY User's Guide [10] suggests in Chapter 12 a method how an edge-coloured graph can be transformed into a vertice-coloured graph. We implemented a simpler version, introducing one level per edge-colour, because it is easier to prove its correctness. We describe our adapted method here briefly.

For each colour (different label) on the edges, there is a layer with all nodes of the vertice- and edge-coloured graph constructed. The layers are connected with undi-

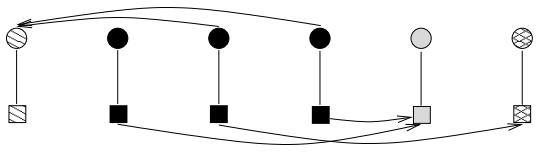

Figure 7.

rected edges[2]. We give a description of the first step of the transformation process. Let $g = \langle V, C_N, C_E, L, E\rangle$ the original vertice- and edge-coloured graph, with C_N and C_E the colours of the nodes and edges respectively, $L : N \to C$ a node labelling function, and $E \subseteq C_E \times N \times N$. Then $\hat{g} = \langle \widehat{V}, \widehat{C}_N, \widehat{E}, \widehat{L}\rangle$ is the layered graph, with respective sets of nodes, colours on the nodes and edges, and labelling function. Let n_E be the number of colours on the edges of g. For each $u \in V$ there are $u_i \in \widehat{V}$, where $i = 1, \cdots, n_E$, such that $\{(u_i, u_{i+1}), (u_{i+1}, u_i) \mid i = 1, \cdots, n_E\} \subseteq \widehat{E}$ holds.

In each layer of the graph $\hat{g}$ there is an $u_i \in \widehat{V}, i \in 1, \cdots, n_E$, being a duplicate of $u \in V$. Every such u_i gets a colour depending on the colour of $u \in V$ in the original graph and i, the number of the layer in $\hat{g}$.

In Figure 7 we gave the nodes in each layer a different shape, since the nodes have already different shades for different colours, to differ the nodes of the layers.

There is now one layer for each colour on the edges of g. The edges with colour one are inserted in layer one, edges with colour two in layer two and so on, connecting the respective nodes of the original graph g. For example, let $e = (u, v) \in E$ have colour 02, then the corresponding edge is inserted in $\hat{g}$ in layer 2, from node u_2 to node v_2. So we have $(u_2, v_2) \in \widehat{E}$.

In general, for every $e = (u, v) \in E$ with colour $i, i \in 1, \cdots, n_E$ there is an edge $(u_i, v_i) \in \widehat{E}$, where u_i is the corresponding node to u and v_i is the corresponding node to v in layer i. The edges in $\hat{g}$ are not coloured anymore as the corresponding ones in g. Finally we get a vertice but no longer edge coloured graph, so that it can be handled by NAUTY.

Correctness of Encoding Label Colours as Node Colours

We only formalise and prove the transformation of an edge-coloured graph into a vertice-coloured graph. The extension to a vertice- and edge-coloured graph is straightforward. Full proofs can be found in the appendix for referees. Readers who are not interested in the correctness of the translation can skip directly to the description of the model checking algorithm.

Add ref to tech report with proofs.

[2] equivalent to two directed edges in both directions

Definition 3.1 An edge-coloured graph is a tuple $\langle N, C, E\rangle$ where N is a set of nodes, C a set of colours, and $E \subseteq C \times N \times N$.

A vertice-coloured graph is a tuple $\langle N, C, L, E\rangle$ where N is a set of nodes, C a set of colours, $L : N \to C$ a node labeling function, and $E \subseteq N \times N$.

Give an edge-coloured (resp. vertice-colorued) graph g we denote by $nodes(g)$ the set of nodes of g.

Definition 3.2 A *permutation* π from N to N' is a bijection from N to N'.

A permutation π of N can be applied to an edge-coloured graph $g = \langle N, C, E\rangle$ as follows: $\pi(g) = \langle N', C, E'\rangle$, where $N' = \{\pi(n) \mid n \in N\}$ and $E' = \{(c, \pi(n_1), \pi(n_2)) \mid (c, n_1, n_2) \in E\}$.

A permutation π of N can be applied to a vertice-coloured graph $g = \langle N, C, L, E\rangle$ as follows:
$\pi(g) = \langle N', C, L', E'\rangle$, where $N' = \{\pi(n) \mid n \in N\}$, $\forall n \in N : L'(\pi(n)) = L(n)$ and $E' = \{(\pi(n_1), \pi(n_2)) \mid (n_1, n_2) \in E\}$.

Two label-coloured (resp. vertice-coloured) graphs g_1, g_2 are isomorphic iff there exists a permutation function π from $nodes(g_1)$ to $nodes(g_2)$ such that $\pi(g_1) = g_2$.

We now show how to formally translate an edge-coloured graph into a vertice-coloured graph, encoding every label-colour as a level in the vertice-coloured graph:

Definition 3.3 Let $g = \langle N, C, E\rangle$ be an edge-coloured graph. We denote by $level(g)$ the vertice-coloured graph $\langle N \times C, C, L^\times, E^\times\rangle$, where

- $L^\times((n, c)) = c$,
- $E^\times = \{((n, c), (n, c')) \mid n \in N \wedge c, c' \in C\} \cup \{((n_1, c), (n_2, c)) \mid n_1, n_2 \in N \wedge c \in C \wedge (c, n_1, n_2) \in E\}$

Proposition 3.4 Let g, g' be two label-coloured graphs. If g and g' are isomorphic then $level(g)$ and $level(g')$ are isomorphic.

For showing the opposite direction we need the following Lemma, that says that if a node (n, c) is mapped onto a node $(n'c)$ under π then for any colour $c' \in C$ the corresponding node (n, c') is mapped to the corresponding node (n', c') with the same n'.

Lemma 3.5 Let π be a permutation such that $\pi(level(g)) = level(g')$. Let $level(g) = \langle N \times C, C, L, E\rangle$ and $level(g') = \langle N' \times C, C, L', E'\rangle$. Then for any $n \in N, n' \in N$ and $c \in C$ we have: $\pi((n, c)) = (n', c) \Rightarrow \forall\, c'.(c' \in C \Rightarrow \pi((n, c')) = (n', c'))$.

Proposition 3.6 Let g, g' be two label-coloured graphs. g and g' are isomorphic if $level(g)$ and $level(g')$ are isomorphic.

Model checking algorithm

We now formalise our modified model checking algorithm. To show how symmetry detection via graph isomorphism is integrated into the checking, see Algorithm 3.7 adapted from [15]. When a new state is encountered it is not explored further if its canonical form has already been explored. In the algorithm $error$ is a function which returns true if the argument is an error state: usually, this means an invariant violation or a deadlock[3]. Also observe our use of the $random$ function, and α, which is a user defined value. Its effect varies whether model checking progresses using a depth first or breadth first search.

Algorithm 3.7[*Model Checking with Symmetry Reduction*]

Input: An abstract machine *M*
Queue := {*root*} ; $Canon$:= {}; $SGraph$:= {}
while *Queue* is not empty **do**
 if random(1) $< \alpha$
  then state := pop_from_front(*Queue*); /* depth-first */
  else state := pop_from_end(*Queue*); /* breadth-first */
 end if
 if $error$(*state*) **then**
  return counter-example trace in $SGraph$ from *root* to *state*
 else
  for all $succ$,*Op* **such that** *state* $\rightarrow^{M}_{Op}$ $succ$ **do**
   sg := *nauty_canon*($\mathcal{G}(succ)$)
   if $\exists\, s$ such that $(sg, s) \in Canon$ **then**
    $SGraph := SGraph \cup \{state \rightarrow_{Op} s\}$
   else
    add $succ$ to front of *Queue*
    $Canon := Canon \cup \{(sg, succ)\}$
    $SGraph := SGraph \cup \{state \rightarrow_{Op} succ\}$
   end if
  end for
 end if
od
return ok

The variable *Queue* stores the states with transitions yet to be explored, and $Canon$ records canoncial forms of states already reached, along with the corresponding state. $SGraph$ stores the section of the model explored so far. The function $\mathcal{G}$ converts a state of a B machine into a labelled, directed graph, as explained in Section 2. The function $nauty\_canon$ computes a canonical form for such a graph using NAUTY, as explained earlier. Note that all elements of *Queue* and $Canon$ have associated hash values. It is therefore usually efficient to decide whether $(sg, s) \in Canon$. We have implemented this algorithm within PROB, and we provide empirical results later in Section 4.

[3]We do not deal with liveness properties in this algorithm. In B such properties are encoded via refinement.

4. Empirical Results

In the following we give the results of some runtime experiments. The tests were conducted under Debian Linux on a AMD Dual Core 3800+, 2GHz system.

We have compared the method presented in this paper, abbreviated as *nausym* in the tables and the following, with the other symmetry methods described in [7] (flood) and [15] (canon), as well as the approximate method of [8] (hash). Runtime results presented in those articles may differ, since different computer architectures were used. As baseline we have also conducted experiments with PROB, where symmetry reduction was disabled (wo).

We have used a variety of B specifications in our experiments, and we have partitioned them into two tables: Table 1 contains those experiments were all symmetry reduction methods examined the same number of nodes (sym) and Table 2 contains those experiments were the symmetry reduction methods differed in the number of states examined. (Hence, Table 1 contains only a single column for the number of states computed by all symmetry methods; whereas Table 2 contains one column per method). The machines are the same as in [8], except for TokenRing which is a new B model of a token ring network. We have also varied the cardinality of the deferred sets in our experiments; the cardinality used is shown in the first column (card). Runtimes are expressed in seconds; Table 1 also contains columns for the speedup of our new method compared the other methods (a value above 1 means that our method is faster).

Analysis of the results

nausym vs wo: We can see that for very small cardinalities of the dereferred set, the runtimes for each symmetry method or even without symmetry do not differ much in most cases. Except for machines like *USB_4Endpoints*, where the runtime exceeds reasonable timeperiods already for cardinality greater than three. The greater the cardinality of the dereferred sets, the greater is also the speedup of *nausym* compared to using no symmetry (see speedup wo). For a cardinality of five using *nausym* is already more than ten times faster for all four machines.

nausym vs canon: We can see that in every instance our new implementation is more efficient than canon from [15]. Sometimes the difference is dramatic, exceeding 3 orders of magnitude. Recall that [15], has the same mathematical foundation as our new method: The B-states are translated into coloured graphs and a canonical label is computed, to decide if the respective state has been computed already or not (see Section 2 and [15]). However, in [15] the standard canonical labelling algorithm was extended to vertice- and edge-coloured graphs and implemented in Prolog. In this paper we transform the B state graphs into vertice-coloured graphs as explained in Section 3 and then apply NAUTY. In the introducction we stated that the runtime of the approach

in [15], was dissapointingly slow and therefore the question was, if this was due to the method based on graph isomorphism or the implementation. Now we can answer this question and say that the disapointing runtime results were due to the implementation. Probably part of the blame goes to the implementation in Prolog, the other part is that the canonical labelling algorithm itself in [15] did not include many of the optimisations and years of refinement that make NAUTY such an effective tool.

nausym vs flood: flood employs a different approach to symmetry reduction, called permutation flooding [7]. For small cardinalities it behaves similarly (sometimes even slightly better) than our new implementation. But for higher cardinalities the flooding of the state space induces often too big of an overhead; meaning that our approach is generally faster and much more scalable (see, e.g., scheduler1 or TokenRing).

nausym vs hash: The only method which is faster than our new implementation is the hash marker method from [8]. Indeed, it is the only other method that scales well for the scheduler1 and TokenRing examples. The method using hash markers is about twice as fast as symmetry with NAUTY in all four examples from Table 1 for most cardinalities. Although it is precise for these examples, the symmetry reduction with hash markers is not an exact method. That means that error states of a machine could be missed out. This can be seen in Table 2: the hash markers often computes less states than required to exhaustively model check the B machine. In fact, Table 2 shows that the number of states computed by each method differs. Permutation flooding and *nausym* compute the same number of states, which is the minimum number for a correct model checking.[4]

Analysis of the graph canonicalisation time After comparing the runtime of the symmetry reduction with NAUTY with our other symmetry reduction methods, we asked ourselves, how much time NAUTY actually takes to compute the canonical forms, as compared to the total runtime. Furthermore we wanted to know, if there were any graphs for which the computation of the canonical form was exceedingly expensive? In theory the answer to the latter question is "yes": in fact we were able to construct a small graph with 15 nodes where the computation of the canonical form took about a minute. Luckily, such graphs never occured during our empirical tests. We measured for several machines the maximum time taken to compute the canonical form of a single graph. In all experiments, this time was less than a millisecond. Considering that the graphs can be quite big, for greater cardinalities of the dereferred sets, that was quite a surprising result. A graphical representation of the size of the state graphs which are fed to NAUTY, can be found in Figure 8. E.g., fo the *USB*4*_Endpoints* machine there are more than 800 graphs with 45 nodes each, but the total runtime of NAUTY is still quite small: most of the runtime of 1.14 seconds (see table 1) comes from ProB's model checking and interpretation of the B machine. We believe that the graphs constructed from B-machine states have a special structure, which makes it easy for NAUTY to compute the canonical form. Still, we believe that there is some potential in optimizing the code of the interface which performs the translation of the state graphs, see section 3, for NAUTY.

[4]The Prolog implementation of the canonical labelling algorithm does not detect symmetric states that arise during the constant setup phase, therefore more states need to be model checked.

5. Conclusion and Related Work

So far this paper has concentrated on model checking B models. However, in [13] it was shown how Z models could be translated into B models and how PROB could be used to model check Z specifications. Also, in [9], we have developed an LTL model checker for PROB, which can be correctly applied without restriction in the presence of symmetry reduction. Hence, our new method is also applicable to Z as well as for LTL model checking (something which we have already successfully used in practice).

We have presented a new symmetry reduction implementation for high-level B and Z models. We have shown (and proven correct) a way to translate B states into graphs suitable for NAUTY, and have shown how the NAUTY tool can be integrated into PROB. We have conducted extensive empirical experiments, analyzing runtimes and various other aspects related to the graph canonicalisation procedure.

This paper started with the question, if the disappointing performance results of [15] were due to the computation of the canonical labelling or the inplementation of the algorithm. In this paper, we have clearly answered this question, and shown that our new technique and implementation outperforms [15]; often by several orders of magnitude. Our technique also is generally much more effective than permutation flooding [7]. We have also shown that graph canonicalisation is not the bottleneck, at least for the experiments conducted. Our technique is generally slower than the approximate method of [8], but scales equally well while being fully precise.

In summary, we believe that we now have a tool which can effectively exploit symmetries in B and Z models. Contrary to other approaches, such as, e.g., [2], the user has to perform no annotation; symmetries induced by deferred sets are exploited automatically. Furthermore, partial symmetries can be exploited (such as in the Dining example from Table 2). Our long term goal is to provide a tool that can both deal with a high-level formalism such as B or Z, while at the same time providing performance comparable to tools

Table 1. Empirical Results I

				Runtimes of Symmetry Methods				Speedup			
card	states (wo)	states (sym)	wo	flood [7]	hash [8]	canon [15]	nausym	wo	flood [7]	hash [8]	canon [15]
scheduler0											
2	16	10	0,04	0,03	0,03	0,05	0,03	1,3	0,9	0,9	1,4
3	55	17	0,23	0,08	0,08	0,16	0,09	2,5	0,9	0,8	1,8
4	190	26	1,10	0,24	0,17	0,60	0,21	5,4	1,1	0,8	2,9
5	649	37	5,10	0,94	0,33	2,76	0,41	12,4	2,3	0,8	6,7
6	2188	50	23,08	6,11	0,61	17,12	0,76	30,5	8,1	0,8	22,7
7	7291	65	115,01	55,07	1,00	139,05	1,28	89,9	43,0	0,8	108,7
scheduler1											
2	27	14	0,06	0,04	0,03	0,26	0,05	1,2	0,7	0,7	5,12
3	145	29	0,46	0,12	0,10	1,29	0,16	2,0	0,8	0,7	8,08
4	825	51	3,36	0,43	0,25	6,27	0,39	8,6	1,1	0,6	16,03
5	5201	81	27,13	2,28	0,54	35,56	0,84	32,4	2,7	0,7	42,42
6	37009	120	333,82	35,71	0,96	674,26	1,61	207,9	22,2	0,6	419,93
7	-	169	*	*	1,66	*	2,80	-	-	0,6	-
10	-	386	*	*	6,51	*	11,37	-	-	0,6	-
15	-	1041	*	*	35,67	*	67,08	-	-	0,5	-
20	-	2171	*	*	141,96	*	257,16	-	-	0,6	-
RussianPostalPuzzle											
1	15	15	0,03	0,03	0,03	0,07	0,06	0,5	0,5	0,5	1,1
2	81	48	0,21	0,14	0,13	0,42	0,25	0,8	0,6	0,5	1,7
3	441	119	1,40	0,54	0,43	2,20	0,81	1,7	0,7	0,5	2,7
4	2325	248	9,36	2,37	1,14	11,53	2,23	4,2	1,1	0,5	5,2
5	11985	459	64,52	15,91	2,58	63,97	5,57	11,6	2,9	0,5	11,5
USB_4Endpoints											
1	29	29	0,21	0,21	0,23	24,67	1,14	0,2	0,2	0,2	21,7
2	694	355	10,69	5,80	7,74	547,17	21,97	0,5	0,3	0,4	24,9
3	16906	3013	1533,54	265,19	208,40	*	297,43	5,2	0,9	0,7	-

* means test has been cancelled or not done, because of excessive runtime

Table 2. Empirical Results II (where the methods calculate different number of nodes)

	number of states						Runtime of Symmetry Methods			
card	wo	flood [7]	hash [8]	canon [15]	nausym	wo	flood [7]	hash [8]	canon [15]	nausym
Token Ring										
2	35	19	19	35	19	0,07	0,06	0,05	0,11	0,07
3	295	60	60	148	60	0,49	0,19	0,16	0,65	0,22
4	3097	174	141	646	174	6,57	1,44	0,46	6,47	0,86
5	38521	480	278	2248	480	175,61	42,90	0,97	61,98	2,96
6	-	-	495	8460	1252	*	*	2,09	921,79	9,90
7	-	-	816	-	3160	*	*	4,27	*	33,86
Dining										
2	21	8	7	11	8	0,07	0,05	0,04	0,06	0,05
3	337	13	11	29	13	1,50	0,18	0,08	0,27	0,10
4	17713	48	17	165	48	145,53	18,13	0,15	3,39	0,54
Towns										
2	17	11	11	11	11	0,34	0,20	0,21	0,24	0,21
3	513	105	105	105	105	67,78	13,45	13,73	15,68	13,93
4	65537	3045	3011	-	3045	*	1721,73	1748,45	*	1732,03

* means test has been cancelled or not done, because of excessive runtime

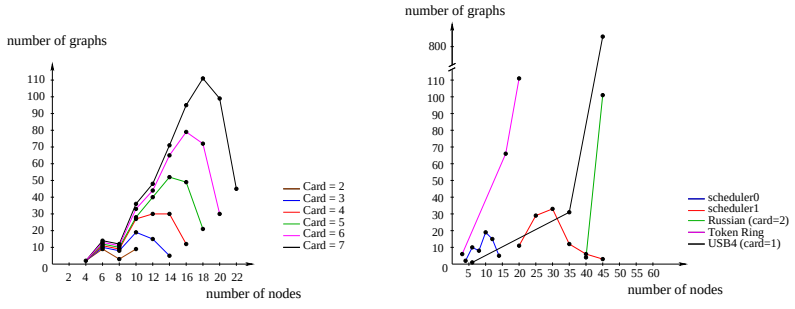

Figure 8. Size of state graphs for scheduler0 (with varying cardinality) and for varying B machines

such as SPIN working on lower-level Promela models. In first preliminary experiments, we have hand-translated the scheduler1 example from Table 1, with the help of Promela experts. The outcome was that for 4 processes PROB with NAUTY symmetry reduction is actually quite competitive compared to SPIN with partial order reduction, despite the much higher-level input language. Furthermore, if we actually measure the total time taken to display the result to the user, PROB is faster (for SPIN there is the overhead to generate and compile the C code). For 12 processes we were actually not able to perform the model checking in SPIN, despite various attempts, whereas PROB checked the model in less than 15 seconds. Of course, this is just one particular experiment and it is too early to draw any general conclusions. Also, in addition to partial order reduction, one could also try and use symmetry reduction for SPIN, e.g., by using the SymmSPIN tool [2] or TopSPIN tool [5], However, exploiting symmetries at the Promela level is more difficult and limited (no partial symmetries; only one symmetric scalar set, etc...). In conclusion, we believe we have presented a model checking technique and tool that can effectively and accurately exploit symmetries in high-level B and Z models.

References

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric Spin. *STTT*, 4(1):92–106, 2002.

[3] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

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

[5] A. F. Donaldson and A. Miller. Exact and approximate strategies for symmetry reduction in model checking. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, LNCS 4085, pages 541–556. Springer, 2006.

[6] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[7] M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry reduction for B by permutation flooding. In *Proceedings B2007*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.

[8] M. Leuschel and T. Massart. Efficient approximate verification of B via symmetry markers. *Proceedings International Symmetry Conference*, pages 71–85, Januar 2007.

[9] M. Leuschel and D. Plagge. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In *Proceedings Isola 2007*, Revue des Nouvelles Technologies de l'Information RNTI-SM-1, pages 73–84, December 2007.

[10] B. D. McKay. Nauty users guide. Available via `http://cs.anu.edu.au/people/bdm/nauty/`.

[11] B. D. McKay. Practical graph isomorphism. *Congress Numerantium*, pages 45–87, 1981. Available via `http://cs.anu.edu.au/ bdm/nauty/PGI/`.

[12] A. Miller, A. F. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3), 2006.

[13] D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Proceedings IFM 2007*, LNCS 4591, pages 480–500. Springer-Verlag, 2007.

[14] E. Turner. *Improving the Process of Model Checking through State Space Reductions*. PhD thesis, University of Southampton, 2007.

[15] E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry reduced model checking for B. In *Proceedings Symposium TASE 2007*, pages 25–34, Shanghai, China, June 2007. IEEE.