Symmetry Reduction for B by Permutation Flooding

Michael Leuschel[1], Michael Butler[2], Corinna Spermann[1] and Edd Turner[2]

[1] Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{leuschel,spermann}@cs.uni-duesseldorf.de
[2] School of Electronics and Computer Science, University of Southampton
Highfield, Southampton, SO17 1BJ, UK
{mjb,ent03r}@ecs.soton.ac.uk

Abstract. Symmetry reduction is an established method for limiting the amount of states that have to be checked during exhaustive model checking. The idea is to only verify a single representative of every class of symmetric states. However, computing this representative can be non-trivial, especially for a language such as B with its involved data structures and operations. In this paper, we propose an alternate approach, called permutation flooding. It works by computing permutations of newly encountered states, and adding them to the state space. This turns out to be relatively unproblematic for B's data structures and we have implemented the algorithm inside the ProB model checker. Empirical results confirm that this approach is effective in practice; speedups exceed an order of magnitude in some cases. The paper also contains correctness results of permutation flooding, which should also be applicable for classical symmetry reduction in B.
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. It is used in industry in a range of critical domains. In addition to the proof activities it is increasingly being realised that validation of the initial specification is important, as otherwise a correct implementation of an incorrect specification is being developed. This validation can come in the form of *animation*, e.g., to check that certain functionality is present in the specification. Another useful tool is *model checking*, whereby the specification can be systematically checked for certain temporal properties. In previous work [14], we have presented the ProB animator and model checker to support those activities. Implemented in Prolog, the ProB tool supports automated consistency checking and deadlock checking of B machines and has been recently extended for

[1] This research is partially supported by the EU funded project: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

automated refinement checking [15], also allowing properties to be expressed in CSP [4].

However, it is well known that model checking suffers from the exponential state explosion problem; one way to combat this is via *symmetry reduction* [6]. Indeed, often a system to be checked has a large number of states with symmetric behaviour, meaning that there are groups of states where each member of the group behaves like every other member of the group. In the case of B machines this arises, e.g., when using deferred sets, where one set element can be replaced by another without affecting the behaviour of the machine. The classical approach to symmetry reduction requires the determination of a so-called *representative* for every symmetry group; the idea being that it is sufficient to check just the representative. Computing such a representative, at least for a formalism such as B with its sophisticated data structures and operations, is a non-trivial task. In this paper we present an alternate way to add symmetry reduction to B, which we have incorporated into ProB's model checking algorithm. Indeed, while it is not trivial to pick a (unique) representative for every symmetry group, it is relatively straightforward in B to generate from a given state a set of symmetric states, essentially by permuting the elements of deferred sets. Our new algorithm uses that fact to achieve symmetry reduction; the basic idea being that when a new state is added to the state space all symmetric states are also added. While this can result in a considerable number of symmetric states being added, we show that—in the absence of counterexamples—all of them would have been explored using the classical model checking algorithm anyway. We have implemented this algorithm, and have evaluated it on a series of examples. Our experiments show that big savings can be achieved (exceeding an order of magnitude).

2 Motivation and Overview

Let us examine the following simple B machine, modelling a system where a user can login and logout with session identifiers being attributed upon login.

```
MACHINE LoginVerySimple
SETS Session
VARIABLES active
INVARIANT  active<:Session
INITIALISATION active := {}
OPERATIONS
 res <-- Login = ANY s WHERE s:Session & s/: active THEN
                   res := s || active := active \/ {s} END;
 Logout(s) = PRE s: active THEN active := active - {s} END
END
```

This machine contains the deferred set `Session`. For animation, the user has to select some finite size for this set [14]. Figure 1 contains the full state space generated by ProB for this machine, where the cardinality of `Session` was set

to 3 (and ProB has automatically generated three elements *Session*1, *Session*2 and *Session*3 of that set). One can see that the states 2,3,4 are symmetric, in the sense that:

– the states can be transformed into each other by permuting the elements of the set `Session`;
– if one of the states satisfies (respectively violates) the invariant, then any of the other states must also satisfy (respectively violate) the invariant;
– if one of the states can perform a sequence of operations, then any other state can perform a similar sequence of transitions; possibly substituting operation arguments (in the same way that the state values were permuted). E.g., state 2 can perform *Logout*(*Session*1), state 3 can be obtained from state 2 by replacing Session1 with Session2, and, indeed, state 3 can perform *Logout*(*Session*2).

The same holds for the states 5,6 and 7.

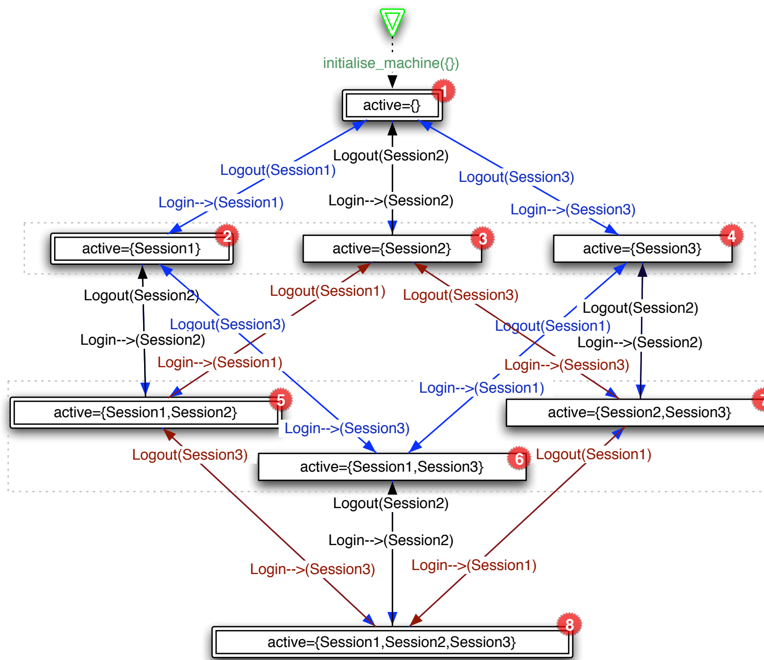

Fig. 1. Full state space; representatives are marked by double boxes

Classical Approach to Symmetry Reduction In classical symmetry reduction, one would compute a representative for every set of symmetric states. A possible

choice for such representatives for the above example would be the states 1,2,5,8; shown as boxes with double borders in Figure 1. When a model checker with symmetry reduction encounters state 3, it would compute its representative (i.e., state 2) which is already in the state space. Thus state 3 would not need to be checked. The same holds for state 4.

Deciding whether two states can be considered symmetric (also called the "orbit problem") is tightly linked to detecting graph isomorphisms (see, e.g., [6][Chapter 14.4.1]). Indeed, one can directly employ algorithms for detecting graph isomorphisms, by converting the system states into graphs and then checking whether these graphs are isomorphic. One efficient approach is by computing the canonical form of the graphs (called certificates in [12]). Such an approach also looks promising for B, but requires careful extension due to the data structures and operations of B.

New Approach The new algorithm we present in this paper works the other way around to classical symmetry reduction: when a new state is added to the state space, we at the same time (proactively so to speak) add all states which can be obtained by permuting the deferred set values of the state. This is based on the three insights below:

- **Insight 1**: Whereas it is difficult to find out if two states are symmetric and compute a representative, it is actually quite straightforward to generate symmetric permutation of a given state; at least in B. Indeed, symmetry in B occurs usually due to the use of deferred sets; this was the case in the example above. One thus simply has to permute the deferred set elements that occur in the given state.
- **Insight 2**: In order to prevent symmetric states from being checked, we can simply add the permutations to the state space and mark them as "already processed."
- **Insight 3**: The obvious drawback of generating all permutations is that there can be many such permutations, "flooding" the state space. However, they would all have been encountered during an exhaustive model check anyway! Thus—at least when compared to classical model checking in the absence of counterexamples—we have nothing to lose by using the new algorithm. We gain that for the permuted states we do not have to compute the invariant, nor compute the enabled operations and their effect. Furthermore we apply the permutation generation only to one representative of each symmetry group; all the other representatives will be detected by straightforward state hashing and identity checking using normalisation [14].

An illustration of the approach on the above example can be found in Figure 2. When adding the state $active = \{Session1\}$ the two symmetric states $active = \{Session2\}$ and $active = \{Session3\}$ are also added to the state space. Similarly, when adding $active = \{Session1, Session2\}$ its two symmetric states are also added. As can be seen, a reduction in the checking effort has been achieved: only 4 of the 8 states have to be checked (i.e., the invariant evaluated and the enabled operations computed).

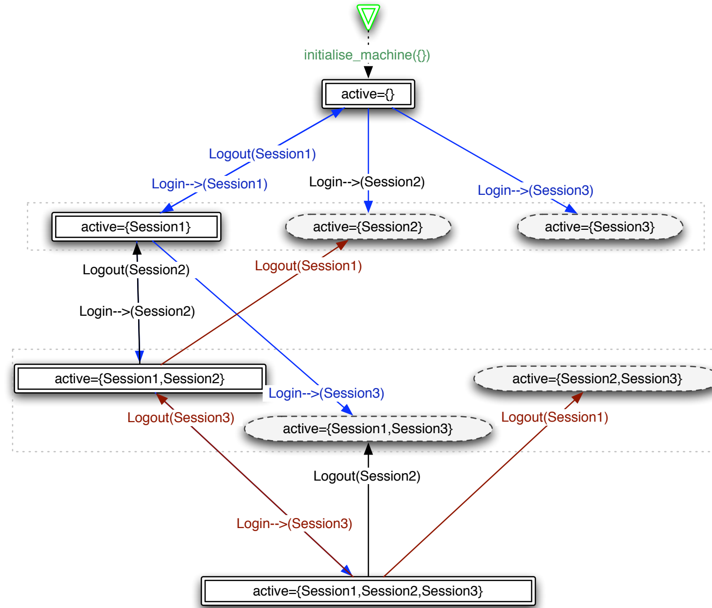

Fig. 2. Full state space after permutation flooding

3 The Algorithm

Informal Explanation Recall, that in B there are two ways to introduce sets into a B machine: either as a parameter of the machine (by convention parameters consisting only of upper case letters are sets; the other parameters are integers) or via the SETS clause. Sets introduced in the SETS clause are called *given sets*. Given sets which are explicitly enumerated in the SETS clause are called *enumerated sets*, the other given sets are called *deferred sets*.

Informally, two states are symmetric when the invariant has the same truth value in both states and when they can both execute the same sequences of operations (possibly up to some renaming of data values). While in the general case it is undecidable whether two states are symmetric or not, we can generate for a given state s a set of states which are guaranteed to be symmetric. The simplest approach is to permute the deferred set elements within s. This is what we have done in the example in the previous section. One may wonder why we only permute the deferred set elements, and not the elements of enumerated sets. Indeed, in the example above, if we replace `Session` by an enumerated set `Session = {Session1, Session2, Session3}` then the state space would remain unchanged (Figure 1 would remain exactly the same) as would the symmetry groups. However, without further knowledge about the machine, we have no guarantee that this would always be the case. Indeed, as

the elements Session1, Session2, Session3 can now be referenced by name, they could be used in the invariant or in the precondition of a machine. For example, `Session1:active => not(Session2:active)` could be used as a precondition of an operation. This would break the symmetry, meaning that we would generate permutation states which are *not* symmetric to the original state. In other words, by not checking the permutation states we may fail to detect an invariant violation or deadlock. Similarly, if the above condition appears inside the invariant, the state $active = \{Session1, Session3\}$ would satisfy the invariant, while the permutation $active = \{Session1, Session2\}$ would not. Later, in Section 6, we will explain how this restriction can be somewhat relaxed; but for the time being we will only permute deferred set elements. Integers, booleans and enumerated sets will not be permuted.

Formalisation The state space of a machine is defined as the cartesian product of the types of each of the machine variables and constants. We represent the machine constants and variables by a vector of variables $v1, \ldots, vn$ (denoted V). The normal form for a B operation operating on the variables V with inputs x and outputs y is characterised by a predicate $P(x, V, V', y)$. Characterising a B operation of the form $X \longleftarrow op(Y)$ as a predicate in this way gives rise to a labelled transition relation on states: state s is related to state s' by event $op.a.b$, denoted by $s \rightarrow^M_{op.a.b} s'$, when $P(a, s, s', b)$ holds. Further details may be found in [15]. We also add a special state $root$, where we define $root \rightarrow^M_{initialise} s$ if s satisfies the initialisation and the properties clause. We now describe how to generate permutation states.

Definition 1. *Let DS be a set of disjoint sets. A* permutation *f* over *DS is a total bijection from $\cup_{S \in DS} S$ to $\cup_{S \in DS} S$ such that $\forall S \in DS$ we have $\{f(s) \mid s \in S\} = S$.*

We can now define permutations for B machines, which permute deferred set elements, respecting the typing (i.e., we only permute within each deferred set).

Definition 2. *Let M be a B Machine with deferred sets $DS_1, \ldots, DS_k$ and enumerated sets $ES_1, \ldots, ES_m$. A function f is called a* permutation for *M iff it is a permutation over $DS_1, \ldots, DS_k$. We extend f to B's other basic datatypes, requiring that f must not permute integer, boolean or enumerated values:*
- $f(x) = x \quad \textit{if} \quad x : \mathbb{Z} \textit{ or } x : BOOL \textit{ or } x : ES_j$ *(for some j)*

Values in B are either elements of given sets (including boolean values and integers), pairs of values, or sets of values. We recursively lift such an f to pairs and sets as follows:
- $f(x \mapsto y) = f(x) \mapsto f(y)$
- $f(\{x_1, \ldots, x_n\}) = \{f(x_1), \ldots, f(x_n)\}$

We also extend the domain of this function f to state vectors by defining
- $f(\langle v_1, \ldots, v_k \rangle) \quad = \quad \langle f(v_1), \ldots, f(v_k) \rangle$

Take for example a B machine with deferred sets $DS_1 = \{s_1, s_2\}$ and $DS_2 = \{r_1, r_2\}$. Then $f = \{s_1 \mapsto s_2, s_2 \mapsto s_1, r_1 \mapsto r_1, r_2 \mapsto r_2\}$ is a permutation over

$\{DS_1, DS_2\}$. Applying f to states we have for example $f(\langle s_1 \rangle) = \langle s_2 \rangle$, $f(\langle r_1, 5 \rangle) = \langle r_1, 5 \rangle$, $f(\langle \{s_1, s_2\} \rangle) = \langle \{s_1, s_2\} \rangle$, $f(\langle \{s_2\}, s_1 \rangle) = \langle \{s_1\}, s_2 \rangle$, $f(\langle \{s_1\}, \{1 \mapsto s_1\}, \{\{\}, \{s_2\}\} \rangle) = \langle \{s_2\}, \{1 \mapsto s_2\}, \{\{\}, \{s_1\}\} \rangle$. Observe that constants are part of the state and are thus also permuted by f.[2] We can now define when two states are a permutation of each other:

Definition 3. *Let s, s' be two states of a B Machine with deferred sets $DS_1, \ldots, DS_k$. The state s is a* permutation the state *of s' iff there exists a permutation f over $\{DS_1, \ldots, DS_i\}$ such that $s' = f(s)$.*

In order to generate all permutation states of a given state s we thus simply need to enumerate all possible permutations over $\{DS_1, \ldots, DS_i\}$. In our implementation we have added one improvement: if the state s only contains the deferred set values $V \subset \cup_{i=1..k} DS_i$ then we only need to generate "partial" permutations (i.e., we do not have to map elements which do not occur in s).

We can now formalise our model checking algorithm. Below, *error* is a function which returns true if the argument is an error state: usually, this means an invariant violation or a deadlock.

Algorithm 3.1[*Consistency Checking with Permutation Flooding*]

> **Input:** An abstract machine M
> $Queue := \{root\}$; $Visited := \{\}$; $Graph := \{\}$
> **while** $Queue$ is not empty **do**
>   state := pop($Queue$); $Visited := Visited \cup \{state\}$
>   **if** $error(state)$ **then**
>   **return** counter-example trace in *Graph* from *root* to *state*
>   **else**
>   **for all** *succ*,*Op* **such that** $state \rightarrow^M_{Op} succ$ **do**
> $Graph := Graph \cup \{state \rightarrow_{Op} succ\}$
>   **if** $succ \notin Visited$ **then**
>   **if** random(1) $< \alpha$ **then** add *succ* to front of *Queue*
>   **else** add *succ* to end of *Queue* **end if**
> $Visited := Visited \cup \{s | \exists f.f$ is a permutation function $\wedge s = f(succ)\}$
>   **end if**
>   **end for**
>   **end if** **od**
> **return** ok

Note that all elements of *Queue* and *Visited* have associated hash values. It is therefore usually quite efficient to decide whether $succ \notin Visited$. We have implemented this algorithm within ProB, and we provide empirical results later in Section 5. In the following section we will justify the soundness of the approach.

[2] If that is not desired then one could simply impose on the allowed permutations, that for all deferred set elements c occurring in the constants we have $f(c) = c$.

4 Soundness Results

The permutation flooding algorithm optimises the standard exhaustive checking algorithm of ProB by assuming that if a state satisfies the invariant, then all permutations of that state also satisfy the invariant. It also assumes that if a state can reach another state violating the invariant (or exhibiting a deadlock), then all permutations of that state can also reach a state violating the invariant (or with a deadlock). This section outlines the correctness of these assumptions. A key theorem in showing this is that given a state s and a permuted state $f(s)$, the truth value of predicate P in state s is equivalent to the truth value of P in state $f(s)$.

The values of free variables in B expressions and predicates are either elements of given sets (including Boolean values and integers), pairs of values, or sets of values. We find it convenient to represent the state as a substitution of the form $[v1, \ldots, vn := c1, \ldots, cn]$, where $v1, \ldots, vn$ (denoted V) are the variables in any B expression/predicate and $c1, \ldots, cn$ (denoted C) are the values. Such variables include state variables, machine constants, quantified variables and local operations variables.

For expression E, we write $E[V := C]$ to denote the value of E in state $[V := C]$. This value will be an element of some type constructed from the given sets of a machine. Similarly for predicate P, we write $P[V := C]$ to denote the boolean truth value of P in state $[V := C]$. Most B set operators are defined in terms of other more basic operators and/or set comprehension[3]. This means we can focus on the core predicate and expression syntax as defined in [1]. This core syntax is shown in figures 3 and 4[4].

$$
\begin{array}{rll}
E & ::= & Var \\
 & | & Enum \\
 & | & (E, E) \\
 & | & E \times E \\
 & | & \mathbb{P}(E) \\
 & | & \{x | x \in S \wedge P\} \\
 & | & E(E)
\end{array}
$$

Fig. 3. Core syntax for expressions

$$
\begin{array}{rll}
P & ::= & P \wedge P \\
 & | & \neg P \\
 & | & E = E \\
 & | & \forall x.(x \in S \implies P) \\
 & | & E \in E
\end{array}
$$

Fig. 4. Core syntax for predicates

The goal is now to prove that the permutation function f used in permutation flooding will preserve the evaluation of *any expression or predicate*. This is expressed by the following theorem.

[3] For example, $S \subseteq T \Leftrightarrow \forall x.(x \in S \implies x \in T)$

[4] To simplify the presentation we have ignored integer and boolean expressions. These are never permuted by the algorithm. However an integer expression may contain a subexpression of the form $max(S)$ or $card(S)$, where S is a set. The set S in $max(S)$ must be a set of integers and therefore will never be permuted. The set S in $card(S)$ can be any finite set and therefore may be permuted. Such permutation is sound since the injectivity of f means that for any set S, $card(S) = card(f(S))$.

Theorem 1. *For any expression E, predicate P, state $[V := C]$ and permutation function f:*

$$f(E[V := C]) = E[V := f(C)]$$
$$P[V := C] \Leftrightarrow P[V := f(C)]$$

The theorem can be proved by structural induction over expression and predicate terms. The induction is mutual since expressions may contain predicates and vice versa. We don't present the full proof here, but it is instructive to consider two case of the structural induction. Firstly, consider the base case where E is an enumerated value ev:

$$\begin{array}{ll} & f(ev[V := C]) \\ = & f(ev) \quad \textit{ev has no free variables} \\ = & ev \quad \textit{f(ev)=ev} \\ = & ev[V := f(C)] \quad \textit{ev has no free variables} \end{array}$$

The case of an equality predicate makes use of the injectivity of f:

$$\begin{array}{ll} & (E1 = E2)[V := f(C)] \\ \Leftrightarrow & E1[V := f(C)] = E2[V := f(C)] \quad \textit{substitution distributes} \\ \Leftrightarrow & f(E1[V := C]) = f(E2[V := C]) \quad \textit{induction hypothesis} \\ \Leftrightarrow & E1[V := C] = E2[V := C] \quad \textit{f is injective} \\ \Leftrightarrow & (E1 = E2)[V := C] \end{array}$$

Corollary 1. *From Theorem 1 we can conclude that every state permutation f for a B machine M satisfies*

- $\forall s \in S : s \models I$ *iff* $f(s) \models I$
- $\forall s_1 \in S, \forall s_2 \in S$: $\quad s_1 \rightarrow^M_{op.a.b} s_2 \quad \Leftrightarrow \quad f(s_1) \rightarrow^M_{op.f(a).f(b)} f(s_2)$.

In terms of the terminology of [6][Chapter 14], we have thus shown that our permutations are *automorphisms* wrt B's transition relation between states and that the truth value of the invariant is preserved by our permutations. Note that by induction, it follows from Corollary 1 that, if we can execute a trace t from a state s_1 to another state s_2, then we can execute a corresponding trace t' from $f(s_1)$ to $f(s_2)$. This ensures that we do not miss out deadlocks or reachable classes of symmetric states by checking just a single representative of a class. It also ensures that we do not miss out on traces (up to renaming); which is important for B's refinement notion and will enable us (in future) to use permutation flooding for symmetry reduction during refinement checking [15].

Proof. (Sketch) for Corollory 1.
The first point about invariant preservation is obvious. It is also trivial to show that enabling of *op* is preserved by f by applying Theorem 1 to the guard and pre-condition of *op*. The fact that the parameters and return values of an operation are linked can be easily proven by adding new variables to the machine for the arguments and return values and applying Theorem 1 with P being the characteristic predicate of the operation *op*.

From the above results we can also derive an efficiency result for permutation flooding, namely that all permutation states of some reachable state are also

part of the reachable state space (this is nicely illustrated in Figures 1 and 2). In practical terms, this means, in case we exhaustively explore the entire state space, we have nothing to lose by applying permutation reduction.

5 Empirical Results

In a first phase we have performed classical consistency and deadlock checking with and without permutation flooding, on a series of examples using ProB's model checker. The results can be found in Table 1. The column "Nodes" contains the number of nodes for which the invariant was checked and the outgoing transitions computed. The experiments were all run on a multiprocessor system with 4 AMD Opteron 870 Dual Core 2 GHz processors, running SUSE Linux 10.1, SICStus Prolog 3.12.5 (x86_64-linux-glibc2.3) and ProB version 1.2.0.[5] scheduler0.mch and scheduler1.ref are the machines presented in [15]. The scheduler machine is a variation of scheduler0.mch, and is taken from [13]. In all the schedulers the deferred sets are the process identifiers. USB is a specification of a USB protocol, developed by ClearSy. The deferred set are the data transfers. RussianPostalPuzzle is a B model of a cryptographic puzzle (see, e.g., [10]). In this case, the deferred set is the number of available keys and locks.

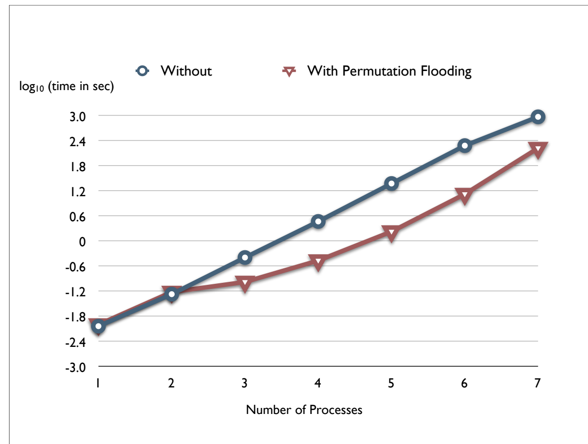

Fig. 5. Model Checking time and evaluated nodes and transitions for scheduler1.ref

Analysis of the results: The results are very encouraging. As can be seen, the permutation flooding algorithm pays off, sometimes achieving an order of magnitude reduction. However, we do not get a fundamental change of the runtime complexity, as Fig. 5 clearly shows. Still, Table 1 shows that a considerable

[5] Note that neither SICStus Prolog nor ProB take advantage of multiple processors.

Table 1. Model checking with and without (wo) permutation flooding

Machine	Card	Time wo (s)	Time with	Speedup	Nodes wo	Nodes with	Transitions wo	Transitions with
scheduler (from [13])	1	0.01	0.01	1.0	4	4	5	5
	2	0.04	0.03	1.6	11	7	25	16
	3	0.09	0.04	2.1	36	11	121	38
	4	0.43	0.07	6.0	125	16	561	75
	5	1.97	0.23	8.6	438	22	2481	131
	6	8.63	1.22	7.1	1523	29	10489	210
	7	36.63	9.87	3.7	5232	37	42617	316
Russian PostalPuzzle	1	0.05	0.05	1.0	15	15	24	24
	2	0.32	0.21	1.6	81	48	177	105
	3	1.32	0.46	2.9	441	119	1227	331
	4	8.73	1.90	4.6	2325	248	7869	838
	5	54.06	12.18	4.4	11985	459	47795	1826
scheduler0 (from [15])	1	0.01	0.01	1.0	5	5	6	6
	2	0.07	0.05	1.5	16	10	37	23
	3	0.28	0.07	4.1	55	17	190	59
	4	0.98	0.20	5.0	190	26	865	121
	5	4.52	0.75	6.0	649	37	3646	216
	6	20.35	4.74	4.3	2188	50	14581	351
scheduler1 (refines scheduler0)	1	0.01	0.01	1.0	5	5	6	6
	2	0.05	0.06	0.9	27	14	62	32
	3	0.41	0.11	3.8	145	29	447	94
	4	2.96	0.34	8.6	825	51	2948	211
	5	23.93	1.70	14.1	5201	81	19925	405
	6	192.97	13.37	14.4	37009	120	145926	701
	7	941.46	167.95	5.61	297473	169	506084	1127
USB.mch	1	0.21	0.26	0.8	23	23	652	652
	2	7.70	3.03	2.5	415	214	13120	6736
	3	283.05	47.92	5.9	7663	1398	248540	45302

amount of nodes do not need to be evaluated for invariant violations or computing the possible outgoing transitions. Hence, our method will pay off especially for machines with more complicated invariants or complicated operations. In the machines above the invariants were actually very simple. For example, scheduler1's invariant only contains typing information. We have thus produced an elaboration of scheduler 1 with a more complicated invariant. The results can be found in Figure 2, and they confirm our expectation that for more complicated machines the permutation flooding algorithm can be even more beneficial (e.g., being 68.5 times faster than classical model checking for 5 processes). However, for simple machines such as the scheduler or scheduler0, the bottleneck is the generation of the permutations. This explains why the relative performance improvements drop off for these simple machines for higher cardinalities of the deferred sets.

Table 2. Further Experiments

Machine	Card	Time wo (s)	Time with	Speedup	Nodes wo	Nodes with	Transitions wo	Transitions with
scheduler1$^+$	2	0.23	0.14	1.7	27	14	62	32
(with more	3	1.81	0.37	4.9	145	29	447	94
complicated	4	32.48	1.97	16.4	825	51	2948	211
invariant)	5	766.19	11.18	68.5	5201	81	19925	405

6 Extending the Algorithm for Enumerated Sets

In this section we discuss how to extend our algorithm for enumerated given sets. It is clear that if an enumerated set element is referenced in the invariant or in the precondition or guard of an operation, that this can cause unsoundness of permutation flooding. We have seen that in Section 3. But what if the set element is only referenced in the body of an operation? Let us look at the following example.

```
MACHINE SymCounterEx
SETS  S={s1,s2,s3}
VARIABLES x
INVARIANT  x:POW(S) & card(x)=1
INITIALISATION  x : (x:POW(S) & card(x)=1)
OPERATIONS  add = BEGIN x := x \/ {s1} END
END
```

Here we have three initial states: $x = \{s1\}$, $x = \{s2\}$, $x = \{s3\}$. They are all symmetric wrt the INVARIANT as well as wrt all Preconditions and Guards of all operations. Still, it is unsound to just examine one representative. Suppose we check $x = \{s1\}$. This state does not violate the invariant and we detect that executing `add` in the state loops back itself and we would incorrectly report that the invariant is not violated by the machine, while it is when executing the `add` operation from either $x = \{s2\}$ or $x = \{s3\}$.

It is instructive to examine exactly where the proof of Section 4 fails if we permute enumerated sets. It is in Theorem 1, when we have an element of a given set as the expression. In this case we no longer have $f(ev) = ev$. As a result, we no longer have that evaluating the expression and then applying the permutation gives the same result as applying the permutation first and then evaluating the expression; breaking the symmetry results. However, if we do not use an enumerated set element inside *any* expression, the proof still goes through. This thus provides a way to extend our correctness results and methodology for enumerated sets.

This idea has also been implemented inside ProB. The basic idea is that, for every enumerated given set ES_i, we compute the values $D_{ES_i} \subseteq ES_i$ which are *not* referenced syntactically inside the invariant, properties, initialisation or the operations of a B machine. If $card(D_{ES_i}) > 1$ then we will permute the values in

D_{ES_i} in the same way as deferred set values were substituted for each other. We thus extend Definition 2 by allowing f to be a permutation over $\{DS_1, \ldots, DS_k\}$ $\cup$ $\{D_{ES_i} \mid 1 \leq i \leq m \wedge card(D_{ES_i}) > 1\}$.

Let us examine this extended algorithm on the example above. Figure 6 shows the correct state space computed by ProB, where $s2$ and $s3$ are considered symmetric wrt each other but not wrt $s1$. As can be seen, the invariant violation is detected. (It can also be seen that ProB inserts artificial permute operations. This is to provide the user with a better feedback when using the animator.)

One may wonder how often in practice a given set would be defined and only part of its elements referenced inside the machine. One typical example is when the set is actually a deferred set, but was enumerated for animation purposes (to give meaningful names to the set elements). This case is actually quite common.

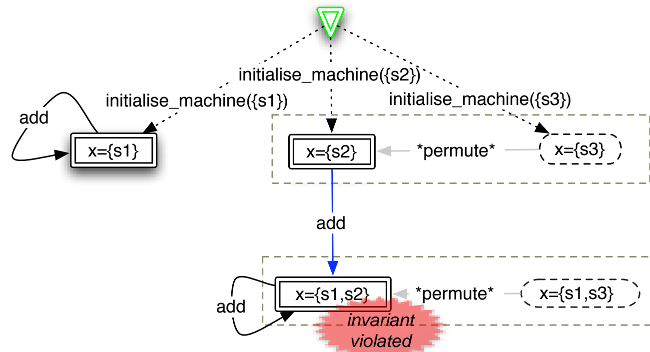

Fig. 6. Full state space after permutation flooding for SymCounterEx

7 Discussion, Related and Future Work

Future Work One interesting avenue for further research is to use the permutation flooding idea also for refinement checking. This could be achieved by suitably taking the "permute" transitions inserted by ProB (see, e.g., Fig. 6) into account. Another idea is to use symmetry reduction when evaluating predicates with existential or universal quantification, to cut down on the number of values that need to be tested for the quantified variables. For example, to evaluate the formula $\forall x.x \subseteq DS \Rightarrow P$ one would only have to evaluate P for one representative per symmetry group. Finally, ProB has recently been extended to be able to treat set comprehensions and lambda abstractions symbolically. These are converted into a closure and are only evaluated on demand. For this extension, we would need to adapt the permutation so that it permutes the values stored inside the closures.

Related Work The line of research developing symmetry reduction for temporal logic model checking is the inspiration for the present article. Symmetry reduction in model checking dates back to [11] and [5], more recent works being [17, 18]. One difference with those works is that in our case we do not consider temporal logic formulas, but B's criteria of invariant violations, deadlocks and refinement. Another important difference is the complexity of B's data structures and operators, making the orbit problem [6] particularly tricky. Still, it should be possible to extend this line of research for B, by using algorithms from graph theory [12] and systems such as NAUTY [16].

As our experiments have indicated, classical symmetry reduction by computing a normal form (i.e., a representative of the set of symmetric states) may in principle be able to achieve even better results than our approach. The drawback of our method is that all permutations are added all of the time (for one representative per class). Depending on the B machine being checked, this may be unnecessary work as by pruning symmetric states initially, many of the later symmetric states may not be reachable anymore. E.g., in Fig 1, this is not the case, but one could imagine that by pruning 3 and 4, states 6 and 7 became unreachable. A classical symmetry reduction algorithm would thus not have to compute a normal form for 6 and 7, whereas we will still add the states 6 and 7 as permutations of 5 to the state space. Furthermore, when invariant violations are present, adding permutations could result in the model checker taking longer to find the first counterexample. However, there will also be cases where permutation flooding is better than a classical symmetry reduction algorithm, namely when the data values inside the individual states get complicated, thus making the computation of the normal form of the state graphs expensive. In our case, the complexity of computing the permutations does not depend on the complexity of the data values being used (they just need to be traversed once); only on the number of deferred set elements occurring inside the state. In summary, a main advantage of our approach lies in its simplicity, along with the fact that it can naturally deal with complicated data structures (such as the closures discussed above).

Another class of related work is the one using symmetry for efficient testing of satisfiability of boolean formulas. These works (e.g., [7], [2, 3] or [8]) use symmetry breaking predicates, which are determined using algorithms from graph theory and which ensure that a subset of the state space will be ignored. Other related work in the formal methods area is the BZ testing tool (BZTT) [13]. This is a test-case generation tool, that also contains an animator. In contrast to ProB, this animator keeps constraints about the variables of a machine, rather than explicitly enumerating possible values. As a side benefit, this gives a simple form of symmetry reduction for the deferred set elements (e.g., the states 2,3,4 in Fig. 1 could be represented by a single state of the form $active = \{s\}$ with the constraint $s \in Session$), but in general not for enumerated sets. Also, the symbolic approach seems difficult to scale up to more complicated B operators (such as set comprehensions, lambda abstractions, existential quantifications, etc., which are not supported by BZTT).

Conclusion In conclusion, we have presented a new way to achieve symmetry reduction for model checking B specifications. The algorithm proceeds by computing permutations of the states encountered during the model checking, and adding those permutations to the state space, marking them as already processed. We have presented a formalisation of this approach, along with correctness results. We have also compared our approach to classical symmetry reduction, arguing that either approach has its advantages and drawbacks. We have implemented the algorithm inside the ProB toolset and have evaluated the approach on a series of examples. The empirical results were very encouraging, the speedups exceeding an order of magnitude in some cases.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *DAC*, pages 731–736. ACM, 2002.
3. F. A. Aloul, K. A. Sakallah, and I. L. Markov. Efficient symmetry breaking for boolean satisfiability. *IEEE Trans. Computers*, 55(5):549–558, 2006.
4. M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.
5. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In C. Courcoubetis, editor, *CAV'93*, LNCS 697, pages 450–462. Springer-Verlag, 1993.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
7. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR 1996*, pages 148–159, 1996.
8. P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for CNF. In S. Malik, L. Fix, and A. B. Kahng, editors, *DAC*, pages 530–534. ACM, 2004.
9. E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.
10. S. Flannery. *In Code: A Mathematical Adventure*. Profile Books Ltd, 2001.
11. C. N. Ip and D. L. Dill. Better verification through symmetry. In *Computer Hardware Description Languages and their Applications*, pages 97–111, 1993.
12. D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, Search*. CRC Press, 1999.
13. B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proceedings FME'02*, LNCS 2391, pages 21–40. Springer-Verlag, 2002.
14. 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.
15. M. Leuschel and M. Butler. Automatic refinement checking for B. In K.-K. Lau and R. Banach, editors, *Proceedings ICFEM'05*, LNCS 3785, pages 345–359. Springer-Verlag, 2005.
16. B. McKay. Nauty users guide. Available at `http://cs.anu.edu.au/people/bdm/nauty/`.
17. A. P. Sistla. Employing symmetry reductions in model checking. *Computer Languages, Systems & Structures*, 30(3-4):99–137, 2004.
18. A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, 26(4):702–734, 2004.