Efficient Approximate Verification of B via Symmetry Markers

Michael Leuschel[1], Thierry Massart[2]

[1] Institut für Informatik, Universität Düsseldorf, leuschel@cs.uni-duesseldorf.de
[2] Université Libre de Bruxelles (U.L.B.), tmassart@ulb.ac.be

Abstract. We present a new approximate verification technique for B models. The technique employs symmetry of B models induced by the use of deferred sets. The basic idea is to efficiently compute markers for states, which are such that symmetric states are guaranteed to have the same marker (but not the other way around). The approximate verification algorithm then assumes that two states with the same marker can be considered symmetric. We describe how symmetry markers can be efficiently computed and empirically evaluate an implementation, showing both very good performance results and a high degree of precision (i.e., very few non-symmetric states receive the same marker). We also identify a class of B models for which the technique is precise.

1 Introduction

The B-method [Abr96] 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.

Invariant properties of B specifications can be expressed and then proven by the semi-automated theorem provers within tools like Atelier-B [Ste96], the B-toolkit [BCUL99] or B4Free. Recently, PROB [LB03] has increased the set of tools available for B with an animator and a model checker. PROB 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. However, it is well known that model checking suffers from the exponential state explosion problem; one way to combat this is via *symmetry reduction* [CGP99]. 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. Symmetry is particularly prominent in B because of *deferred sets*. In previous work [LBST07] we have presented a symmetry reduction technique, called permutation flooding, which ensures that only one representative per symmetry group is checked. This technique can provide substantial speedups, but cannot produce an exponential reduction in complexity. In this paper we present a novel symmetry reduction technique, inspired by the success of Spin's bitstate hashing approximate verification [Hol88]. We define a hashing function, invariant under symmetry, which can be computed efficiently. We avoid the underlying complexity of checking whether two states are symmetric (which basically amounts to checking graph isomorphism), by "assuming" that two states with the same hash value are symmetric.

As this assumption can be wrong in general, we only have an *approximate* verification technique (in the sense that non-symmetric states can obtain the same hash value); but a very fast one. We identify conditions where our method provides a full verification and show cases where it cannot avoid approximations. In experiments we conducted, we show that in all except one case no loss of precision was induced (and all symmetry groups were visited) and a fundamental reduction of complexity was achieved for some examples.

In this paper, we give in Sect. 2, a brief introduction of B and symmetry and briefly explain the link between the symmetry detection and the graph isomorphism problems. We explain why symmetry is particularly prominent and natural in B. We present in Sect. 3 our symmetry reduction technique. We have integrated our method into the PROB tool. In Sect. 4 we evaluate this implementation on a series of examples, comparing it with a naive exploration as well as the precise permutation flooding technique. We also discuss in Sect. 5 related work in the field of symmetry reduction and model checking, particularly the tools Murϕ [ID96] and SMC [SGE00].

2 Symmetry in B

B is based on the notion of *abstract machine*. The variables of an abstract machine can be either elements of basic sets (including Boolean values and integers), pairs of values, or sets of values. Each machine has a certain number of operations that can update the variables of the machine, as well as an invariant specified using predicate logic. (Note that, while refinement is an important concept in B, in this paper we concentrate on consistency of B machines, i.e., checking that the invariant is always satisfied.) There are two ways to introduce basic sets into a B machine: either as a parameter of the machine 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 sets are called *deferred sets*. Operations define, with a high-level of abstraction, substitutions that can transform the state of a machine. Properties that the system must preserve are expressed by an *invariant*. When the cardinalities of all the deferred sets of a B machine have been fixed, the possible behaviours of a B abstract machine can be modelled as a transition system whose nodes are the reachable states and the transitions correspond to possible executions of the operations. This transition system is computed by the model checking tool PROB [LB03], which can also check if every reachable state satisfies the invariant. However, the high-level abstract mathematical formalism used by B means that it is often computationally expensive to compute this transitions system and to check whether all the reachable states satisfy the invariant. Detecting symmetries in the transition system can lead to a considerable reduction of that cost.

Informally, we define two states as being symmetric if the invariant has the same truth value in both states, and when both can execute the same sequences of operations (possibly up to some renaming of data values in the parameters) [LBST07].

Elements of deferred sets are not specified a priori and have no name or identifier. Hence, inside a B machine one cannot select a particular element of such deferred sets. It has been proven in [LBST07] that for any state of B machine, permutations of elements inside the deferred sets preserve the truth value of B predicates in general and the invariant in particular. Furthermore, the structure of the transition relation is also preserved. A reduction technique that exploits symmetries caused by deferred sets is likely to significantly reduce the time to model check many B specifications, since such sets are commonly used in B.

The following simple example from [LBST07] illustrates the basic idea of symmetry in B. Further below we describe a more involved example, which will enable us to show how symmetries can be detected.

Simple login Fig. 1 models a system where a user can login and logout with session identifiers being attributed upon login.

```
MACHINE $LoginVerySimple$
SETS $Session$
VARIABLES $active$
INVARIANT $active \subseteq Session$
INITIALISATION $active := \varnothing$
OPERATIONS
 $res \leftarrow Login$ = ANY $s$ WHERE $s \in Session \wedge s \notin active$ THEN
              $res := s \mid\mid active := active \cup \{s\}$  END;
 $Logout(s)$ = PRE $s \in active$ THEN
              $active := active - \{s\}$  END
END
```

Fig. 1. Simple sessions

This machine contains the deferred set *Session*. With a cardinality of 3 for *Session*, the full state space for this machine has 8 states (one for each possible subset of Session); but the possible behaviours of a state depends solely on the cardinality of the set *active* and not on the identity of the elements in this set. Fig. 2 shows the full state space of the "login" example presented in Section 2, where we have denoted the three elements of *Session* by $s1, s2, s3$. 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 $Logout(Session2)$.

The same holds for the states 5,6 and 7. Therefore, a reduced state space with one representative state for each class (4 classes in this case) can be used. In practice, the reduction method must not build the complete state graph and can proceed on the fly on the reduced one.

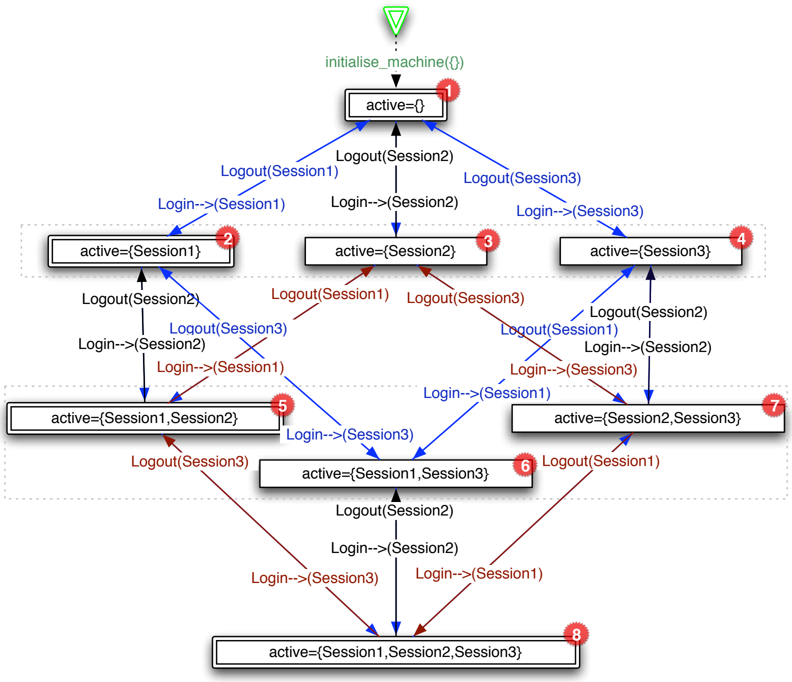

Fig. 2. Full state space; representatives are marked by double boxes

Dining philosophers Another example, with more involved data structures and using constants, can be found in Fig. 3. This will allow us to explain how symmetries can be detected in general. Fig. 3 models the well known dining philosophers problem, where the topology is described by B constants. Notice that we do not specify a protocol for the philosophers. The machine has two *finite* sets $Phil$ and $Forks$, two (constant) total bijections ($\rightarrowtail\!\!\!\!\twoheadrightarrow$) $lFork$ and $rFork$ which assign a left and a right fork to every philosopher, and a variable $taken$ which is a partial function ($\leftrightarrow$) recording for each fork, which philosopher, if any, has taken it into his or her hand. The properties impose that $Phil$ and $Forks$ have the same cardinality, which we denote by n below, and that for all philosophers the right fork must be different from the left one. Initially, no forks are taken and the operations *TakeLeftFork*, *TakeRightFork* and *DropFork* are possible when the corresponding preconditions are true. The (valid) invariant also expresses that a philosopher only takes *his* forks.

This specification is quite general and several initial topologies are possible (n must be at least 2) and for a cardinality n bigger than 3, we can have topologies with several groups of philosophers (e.g. with $n = 4$ two tables of two philosophers are possible). We can easily impose a ring topology with only one big table by adding the following property, excluding subtables: $\forall st.(st \subset Phil \wedge st \neq \varnothing \Rightarrow rFork^{-1}[lFork[st]] \neq st)$.

```
MACHINE Philosophers
SETS Phil; Forks
CONSTANTS lFork, rFork
PROPERTIES
  $lFork \in Phil \rightarrowtail\!\!\!\!\!\twoheadrightarrow Forks \wedge$
  $rFork \in Phil \rightarrowtail\!\!\!\!\!\twoheadrightarrow Forks \wedge$
  $card(Phil) = card(Forks) \wedge$
  $\forall pp.(pp \in Phil \Rightarrow lFork(pp) \neq rFork(pp))$
VARIABLES taken
INVARIANT
  $taken \in Forks \rightarrowtail\!\!\!\!\!\!\!+\;\, Phil \wedge$
  $\forall xx.(xx \in dom(taken) \Rightarrow (lFork(taken(xx)) = xx \vee rFork(taken(xx)) = xx))$
INITIALISATION $taken := \varnothing$
OPERATIONS
  $TakeLeftFork(p, f) =$
   PRE $p \in Phil \wedge f \in Forks \wedge f \notin dom(taken) \wedge lFork(p) = f$ THEN
            $taken(f) := p$
   END;
  $TakeRightFork(p, f) =$
   PRE $p \in Phil \wedge f \in Forks \wedge f \notin dom(taken) \wedge rFork(p) = f$ THEN
            $taken(f) := p$
   END;
  $DropFork(p, f) =$
   PRE $p \in Phil \wedge f \in Forks \wedge f \in dom(taken) \wedge taken(f) = p$ THEN
            $taken := f \mathbin{\lhd\!\!\!-} taken$
   END
END
```

Fig. 3. Dining Philosophers specification

Size of the State Spaces without Symmetry: For the *LoginVerySimple* machine of Fig. 1 and a cardinality of 6 for the deferred set *Session*, 2188 states are reachable in PROB (including a root state and a first initialisation). For the *Philosophers* machine of Fig. 3 with a cardinality of 4 for the sets *Phil* and *Fork*, 216 initial topologies are possible split into 144 topologies with all philosophers at the same table and 72 topologies with 2 tables and 2 philosophers at each one. The full *state space* for this machine has 17713 states. As we will see below, a lot of those states are symmetric.

Symmetry Reduction by Graph Canonicalisation: Deciding whether two states can be considered symmetric (also called the "orbit problem") is tightly linked to detecting graph isomorphisms (see, e.g., [CGP99][Chapter 14.4.1]). Indeed, to detect on the fly if two states are symmetric, one can directly employ algorithms for detecting graph isomorphisms, by converting the system states into graphs and then checking whether these graphs are isomorphic. Graph isomorphism currently has no known polynomial algorithm. However, in practice some efficient algorithms exist for most classes of graphs [KK04]. The most efficient general purpose graph isomorphism program is *nauty* [McK]. In related work [TLSB07], inspired by *nauty*, we have implemented a *canonicalisation function* for B states viewed as graphs, i.e. a procedure which maps each state to a unique member of its equivalence class, called the *canonical form.*

Symmetry Reduction by Permutation Flooding: Informally, we know that two states are definitely symmetric if there exists a permutation of the deferred set elements which transforms one state ino the other. The idea of permutation flooding [LBST07] is thus, for every newly encountered state, compute all permutations states of this state and add them to the state space. Those new states are marked as already processed, and hence will not be checked for invariant violations, nor will the enabled transitions be computed. The first encountered state of each equivalence class becomes de facto the representative element for the class. If the state space fits into memory, this provides a simple symmetry reduction method that can be quite efficient (especially for complicated datastructures) with execution time similar to the graph normalisation method. Further information on these permutation functions, and the soundness results of the symmetries, can be found in [LBST07].

3 Symmetry Markers

Even with symmetry reduction via normalisation or flooding, complete verification of a B model may take too much time or use too much space to be practical. To address this issue, we propose a new approximate verification technique based on *symmetry markers.* The technique is partially inspired by successful Holzmann's bitstate hashing technique [Hol88] which computes a hash value for every reached state: if another state with the same hash value has already been checked, the new state is not analysed any further. As hash collisions can arise, some reachable states are *not* checked. Holzmann's method is therefore, no longer an exhaustive model checking method but an *approximate* verification method (or intensive testing), which is able to discover errors if an error state is reached, but in general cannot certify that the model is error-free.

In our case, the hash value is replaced by a *marker.* This marker has a more complicated structure, but integrates the notion of symmetry: two symmetric states will have the same marker and there is a "small chance" that two non-symmetric states have the same marker. In our model checking algorithm, we will store those markers rather than the states and we will check a new state only if its marker has not yet been seen before. Similarly to the bitstate hashing algorithm, part of the (symmetry reduced) state space may not be checked in case of a collision (i.e., non symmetrical states having the same marker). In the rest of the paper, we will formally present a way to compute such markers, discuss in which case our markers are precise, and present an empirical evaluation exhibiting big speedups (over classical model checking and even over other symmetry approaches) with few collisions (actually in only one example in the experiments).

Formal Definition of Markers:

A marking function is given a state s of a B machine and computes the associated marker. The idea of our marking function is to transform s into a marker by replacing the deferred set elements by so-called *vertex-invariants.*

In graph theory, an invariant [KS99][Sect. 7.2] is a function which does not depend on the presentation of the graph. A vertex-invariant [McK] *inv* is a function which labels the vertices of an arbitrary graph with values so that symmetrical vertices are assigned the same label. Vertex-invariants can be used to speed up graph isomorphism checks. Examples of simple vertex-invariants include the in-degree and the out-degree for the specified vertex. Below we present a more involved vertex-invariant for deferred set elements in B, generalising the ideas of in- and out-degrees.

We denote a state s as a vector $\langle c_1, \ldots, c_n \rangle$ of value of its variables or constants $v_1, \ldots, v_n$ where an order is fixed between them. We also denote multisets by using $\{| \ldots |\}$ and multiset union by $\uplus$. We denote sequences by $\langle \ldots \rangle$. The concatenation of two sequences α and β is denoted by $\alpha.\beta$. If $B = \{| \beta_1, \ldots, \beta_n |\}$ is a multiset of n sequences and α a sequence, we also define $\alpha.B = \{| \beta'_1, \ldots, \beta'_n |\}$ with $\beta'_i = \alpha.\beta_i$.

Informally, we will compute a symmetry marker for a given state s of a B machine as follows:

1. For every deferred set element d used inside s we compute structural information about its occurrence in s, invariant under permutation (and thus symmetry). For this, we compute the multiset of *paths* that lead to an occurrence of d in s. This is formalised in Def. 1 below.
2. Replace all deferred set elements by the structural information computed above. This is formalised in Def. 2.

Definition 1. *Let $d \in \mathcal{D}$ be a deferred set element and e a data value of a variable or constant of a B machine.*

- $paths(d, e) = \{| \langle\rangle |\}$ *if $e = d$,*
- $paths(d, e) = \langle left \rangle.paths(d, x) \uplus \langle right \rangle.paths(d, y)$ *if $e = (x \mapsto y)$ is a pair,*
- $paths(d, e) = \uplus_{x \in e} \langle el \rangle.paths(d, x)$ *if e is a set,*
- $paths(d, e) = \varnothing$ *otherwise.*

For a state s of a B machine with variables and constants V (ordered as $v_1, \ldots, v_n$) we define

– $paths(d, s) = \{\mid \langle v_i \rangle.paths(d, c_i) \mid s = \langle c_1, \ldots, c_n \rangle \mid\}$

$paths(d, s)$ computes structural information on how the deferred set element d is used within s. It identifies which variables and constants use this element and the various *paths* to d in the structure of s (seen as a graph).

$paths(d, s)$ is a vertex-invariant and in the particular case where s is a single binary relation g over D, representing a graph, then $paths(d, s)$ effectively computes the in- and out-degree of the vertex d. E.g., if d has one outgoing and two incoming edges, we will have $paths(d, s) = \{\mid \langle r, el, left \rangle, \langle r, el, right \rangle, \langle r, el, right \rangle \mid\}$.

The following definition simply replaces all deferred set elements within a state by their paths in order to compute the symmetry marker.

Definition 2. *Let s be the state of a B machine with ordered variables and constants $v_1, \ldots, v_n$. We define the* marking function *m, computing markers for data values as follows:*

– $m(s) = \{\mid v_i \mapsto m_s(c_i) \mid s = \langle c_1, \ldots c_n \rangle \mid\}$

where m_s is inductively defined by:

– $m_s(e) = e \quad$ *if* $e \in \mathbb{Z}$ *or* $e \in BOOL$ *or* $e = \varnothing$ *or* $e \in \mathcal{E}$,
– $m_s(e) = (m_s(x) \mapsto m_s(y)) \quad$ *if* $e = (x \mapsto y)$ *is a pair,*
– $m_s(e) = \{\mid m_s(e_1), \ldots, m_s(e_k) \mid\} \quad$ *if* $e = \{e_1, \ldots, e_k\}$ *is a set,*
– $m_s(d) = paths(d, s)$ *if* $d \in \mathcal{D}$.

Fig. 4 illustrates the concepts for machine with the deferred set $D = \{d_1, d_2\}$, the variables c, r where $c \in D$ and $r \subseteq D \times D$, and the state $s = \langle d_1, \{(d_1 \mapsto d_2)\} \rangle$. Note that the state $s_2 = \langle d_2, \{(d_2 \mapsto d_1)\}\}$ is symmetric to the state s of Fig. 4 (the permutation is $f = \{d_1 \mapsto d_2, d_2 \mapsto d_1\}$) and the symmetry markers are identical.

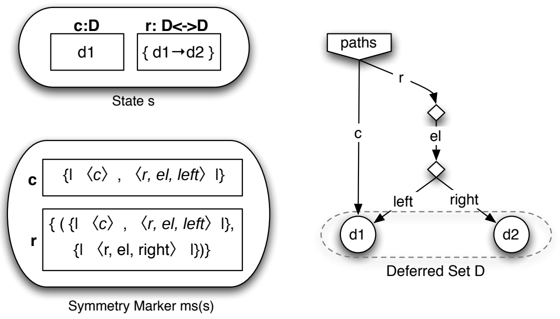

Fig. 4. Illustrating the paths for deferred set elements

Let us examine a few more examples, all with the deferred set $\mathcal{D} = \{d_1, d_2\}$. Take the two states $s_1 = \langle \{d_1 \mapsto 0\}, \{d_1\} \rangle$, $s_2 = \langle \{d_2 \mapsto 0\}, \{d_1\} \rangle$ with variables x

and y. These two states are not symmetric and have also different symmetry markers $m(s_1) \neq m(s_2)$ as $m_{s_1}(d_1) = \{\mid \langle x, el, \mathit{left}\rangle, \langle y, el\rangle \mid\}$, $m_{s_2}(d_1) = \{\mid \langle y, el\rangle \mid\}$, $m_{s_2}(d_2) = \{\mid \langle x, el, \mathit{left}\rangle \mid\}$. For $s_3 = \langle\{d_2 \mapsto 0\}, \{d_2\}\rangle$ we have that $m(s_1) = m(s_3)$, and indeed s_1 and s_3 are symmetric. So far, our symmetry markers have been perfectly precise, i.e., two states had the same marker iff they were symmetric. It is, however, not too difficult to construct cases where this is no longer true and collisions occur. Take the states $s_4 = \langle\{d_1 \mapsto 1, d_2 \mapsto 2\}, \{d_1 \mapsto 1, d_2 \mapsto 2\}\rangle$ and $s_5 = \langle\{d_1 \mapsto 2, d_2 \mapsto 1\}, \{d_1 \mapsto 1, d_2 \mapsto 2\}\rangle$. Those states are not symmetric but they have the same symmetry marker. Such situations (state variables which map deferred set elements to non-symmetric data values) are quite common, and the following improvement to Def. 1 stores more information in the symmetry marker to avoid collisions in those cases:

Let *NonSym* be the smallest set satisfying $((\mathbb{Z} \cup BOOL \cup \mathcal{E} \cup \{\varnothing\} \subseteq NonSym) \wedge (\forall x, y : x \in NonSym \wedge y \in NonSym \Rightarrow (x, y) \in NonSym))$. We extend Def. 1 by replacing the second rule by the following rules:

- $paths(d, e) = \langle to, n\rangle.paths(x)$ if $e = (x \mapsto n) \wedge n \in NonSym \wedge x \notin NonSym$
- $paths(d, e) = \langle from, n\rangle.paths(x)$ if $e = (n \mapsto x) \wedge n \in NonSym \wedge x \notin NonSym$
- $paths(d, e) = \langle leftright\rangle.paths(x)$ if $e = (x \mapsto x) \wedge x \notin NonSym$.
- $paths(d, e) = \langle left\rangle.paths(x) \uplus \langle right\rangle.paths(y)$ if $e = (x \mapsto y) \wedge x \notin NonSym \wedge y \notin NonSym \wedge x \neq y$.

The adapted definition is more precise and now distinguishes s_4 and s_5. We will return to the issue of precision below. We first prove that our definition is indeed invariant under permutation:

Proposition 1. *Let s_1, s_2 be two states for B Machine with deferred sets $\{D_1, \ldots, D_i\}$ such that there exists a permutation f over $\{D_1, \ldots, D_i\}$ where $s_2 = f(s_1)$. Then for any element $d_1 \in \{D_1, \ldots, D_i\}$ we have $paths(d_1, s_1) = paths(f(d_1), s_2)$.*

Proof. If $paths(d_1, s_1) = \{||\}$, then d_1 does not occur in s_1, and hence $f(d_1)$ cannot occur in s_2 either as $s_2 = f(s_1)$. Generally, as $s_2 = f(s_1)$, $f(d_1)$ must occur in exactly the same places in s_2 where d_1 occurred in s_1. This can be formally proven by a straightforward induction on the length of the paths.

From the above proposition we can conclude that for $\forall . d \in \mathcal{D}$ and for every permutation function we have $m_{s_1}(d) = m_{f(s_1)}(f(d))$. We thus have:

Corollary 1. *Let s_1, s_2 be two states of a B machine M. If s_1 and s_2 are permutation states of each other then $m(s_1) = m(s_2)$.*

When are symmetry markers precise Our *Dining Philosopher* example from Sect. 2 can be used to show the limits of our method. Already with 2 philosophers a collision occurs between the state where each philosopher has taken a fork in his left hand with the state where each philosopher has taken a fork in his right hand. Fig. 5 gives a graphical representation of these two states where, to simplify the graph we have represented each pair by an arrow. It is not straightforward

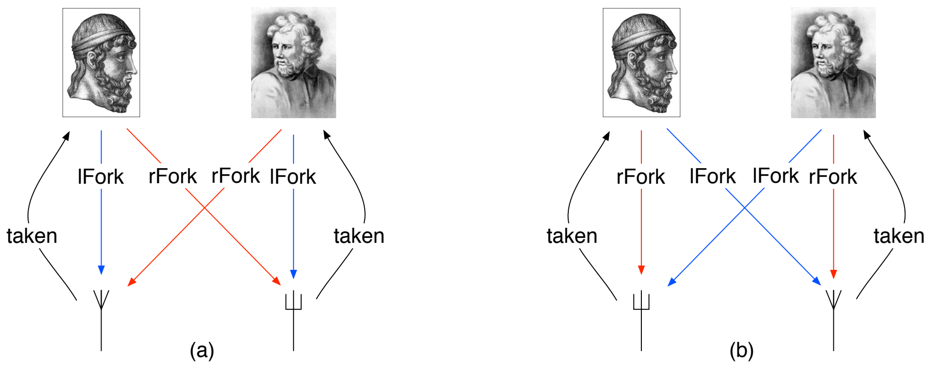

Fig. 5. Two states in collision for the symmetry markers method

to see why these two state are not symmetric.[1] Let us consider the predicate $\forall p.(p : Phil \Rightarrow taken(lFork(p)) = p)$. This predicate is true for the left state (a) but not for the right state (b); hence the two states cannot be symmetrical. We could strengthen our method and detect cycles of length 2 (like the ones occuring in Fig. 5 (a): *p1-f1-p1* and *p2-f2-p2*), but in the end, only a full graph isomorphism algorithm is sufficient to properly identify all symmetries.

Notice however that our method correctly distinguishes between the state where one philosopher has a fork in his left hand with the one where he has one in his right hand.

More generally, the symmetry marker method may fail to properly identify symmetry classes already with a single binary relation over a deferred set,. For example, the states $s_1 = \langle\{d1 \mapsto d2, d2 \mapsto d3, d3 \mapsto d4, d4 \mapsto d1\}\rangle$ and $s_2 = \langle\{d1 \mapsto d2, d2 \mapsto d1, d3 \mapsto d4, d4 \mapsto d3\}\rangle$ have same symmetry marker, but they are not symmetric.

Still, we can identify the following cases where our method is precise:

Proposition 2. *Let s_1, s_2 be two states for B Machine with deferred sets $\{D_1, \ldots, D_i\}$. Let all the values v of variables and constants in s_1 and s_2 be either:*

- *a value not containing any element from one of the sets $D_1, \ldots, D_i$, or*
- *a value not containing a set, or*
- *a set of values $\{x_1, \ldots, x_n\} \subseteq D_k$ for some $1 \leq k \leq i$, or*
- *a set of pairs $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ such that either all x_i are in NonSym and all y_i are elements of some deferred set D_j, or all x_i are in NonSym and all y_i are elements of some deferred set D_j.*

Then $m(s_1) = m(s_2)$ implies that there exists a permutation function f over $\{D_1, \ldots, D_i\}$ such that $f(s_1) = s_2$.

[1] And actually for the current machine in Fig. 3 these two states could be confounded; but if we add a protocol or other predicate which distinguishes left forks from right forks this will no longer be the case.

Proof. (outline) Let us only consider the case where all variable values are sets of deferred elements. The other cases can be translated to such a situation (a variable of type D_j can be seen as a singleton set of elements of D_j; a set of pairs containing a *NonSym* mapped to an element of D_j can be set as a series of sets of elements of D_j: one for every occuring value in *NonSym*; and for values v not containing an element of D we simply have $m_{s_1}(v) = v = m_{s_2}(v)$ and for all permutation functions we have $f(v) = v$). We now construct the function f with the property $m_{s_1}(d) = m_{s_2}(f(d))$ for all $d \in D$. If all the deferred set elements have a different marker, then this function is uniquely defined and must be a bijection. If there are some deferred set elements $d_1, \ldots, d_k$ with the same marker in s_1 then we must also have k elements $d'_1, \ldots, d'_k$ of D which have this same marker in s_2, and hence we can also construct a function f which is a bijection (e.g., choosing $f(d_i) = d'_i$). One can now easily prove that f is such that $s_2 = f(s_1)$ and hence s_1 and s_2 are symmetric.

As a corolloray of the above we know that if the variables and constants used in a B machine fulfill conditions of Prop. 2, our symmetry marker method provides a full verification. Fortunately, in practice quite a lot of specifications seem to fulfill the conditions of Prop. 2. It can also be possible to change a specification to satisfy the conditions. E.g. if we enumerate the forks of the dining philosophers example[2], our method is precise but of course has less symmetry.

Discussion: Our markers could be further improved, to avoid more collisions, by moving from multiset of paths to a tree structure. This would mainly require changing the last rule for computing *paths* above to the following:

– $\mathit{paths}(d,e) = (\langle \mathit{left}\rangle.\mathit{paths}(d,x)\ ,\ \langle \mathit{right}\rangle.\mathit{paths}(d,y))$ if $e = (x \mapsto y) \wedge x \notin \mathit{NonSym} \wedge y \notin \mathit{NonSym}$

This extension has not been implemented in our tool.

4 Empirical Evaluation

We have implemented the technique presented in the previous section and incorporated into PROB. In our implementation, the marker is built up as a Prolog term, and the multisets are sorted and individual occurrences counted.

Below, we give an empirical evaluation of this implementation. We have performed classical consistency and deadlock checking without symmetry reduction (wo) and with our permutation flooding (flood) and symmetry markers (markers) reduction methods, on a series of examples using PROB's model checker. The results can be found in Table 1. RussianPostalPuzzle is a B model of a cryptographic puzzle. (see, e.g., [Fla01]). Scheduler0.mch and scheduler1.ref are the machines presented in [LB05]. Peterson is the specification of the mutual exclusion protocol for n processes as defined in [Pet81]. Philosophers is the dining philosopher example presented above.

The column "Nodes" in Table 1 contains the number of nodes for which the invariant was checked and the outgoing transitions computed.

[2] and use it e.g. in the invariant since otherwise PROB change its specification to deferred set

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.[3]

Table 1. Model checking with symmetry markers compared against classical checking and permutation flooding

Machine	Card	Model Checking Time			Number of Nodes			Speedup over	
		wo	flood	markers	wo	flood	markers	wo	flood
Russian	1	0.05	0.05	0.05	15	15	15	1.04	1.04
	2	0.32	0.21	0.21	81	48	48	1.51	0.97
	3	1.32	0.46	0.34	441	119	119	3.92	1.35
	4	8.73	1.90	0.89	2325	248	248	9.81	2.13
	5	54.06	12.18	2.05	11985	459	459	26.35	5.94
scheduler0	1	0.01	0.01	0.01	5	5	5	0.98	0.99
	2	0.07	0.05	0.05	16	10	10	1.59	1.06
	3	0.28	0.07	0.06	55	17	17	4.60	1.12
	4	0.98	0.20	0.14	190	26	26	7.15	1.43
	5	4.52	0.75	0.27	649	37	37	16.87	2.81
	6	20.35	4.74	0.48	2188	50	50	42.60	9.93
	7	114.71	43.47	0.80	7291	65	65	143.61	54.43
scheduler1	1	0.01	0.01	0.01	5	5	5	1.09	1.12
	2	0.05	0.06	0.05	27	14	14	1.12	1.26
	3	0.41	0.11	0.09	145	29	29	4.50	1.17
	4	2.96	0.34	0.18	825	51	51	16.62	1.93
	5	23.93	1.70	0.37	5201	81	81	64.24	4.56
	6	192.97	13.37	0.70	37009	120	120	275.75	19.10
	7	941.46	167.95	1.22	297473	169	169	771.39	137.61
Peterson	2	0.28	0.28	0.15	49	27	27	1.87	1.89
	3	8.80	2.00	1.73	884	174	174	5.08	1.16
	4	861.49	60.13	20.66	22283	1134	1134	41.69	2.91
Philosophers	2	0.11	0.05	0.04	21	8	7	3.02	1.30
	3	1.56	0.15	0.05	337	13	11	28.83	2.80
	4	123.64	5.99	0.15	11809	26	20	799.36	38.73
Towns.mch	1	0.01	0.01	0.01	3	3	3	1.03	1.00
	2	0.37	0.33	0.34	17	11	11	1.08	0.97
	3	63.95	12.78	12.95	513	105	105	4.94	0.99
USB.mch	1	0.21	0.20	0.22	29	29	29	0.96	0.90
	2	8.42	4.74	6.17	694	355	355	1.36	0.77
	3	605.25	277.59	232.93	16906	3013	3013	2.60	1.19

USB is a specification of a USB protocol, developed by the French company ClearSy. Towns is a specification from the Schneider B Book [Sch01]; here the overhead is in the closure computation of a query operation. The results are very good, and for most examples the symmetry marker is much faster than

[3] Note that neither SICStus Prolog nor PROB take advantage of multiple processors.

the permutation flooding approach. In Towns and USB both fare equally well, which can be explained by the complexity of the specification. For example, in the Towns specification the major bottleneck is the computation of the `closure` of the connectivity graph of the towns; both symmetry markers and permutation flooding get rid of the overhead in the same manner leaving the same number of residual `closure` operations to be computed. In all other examples the symmetry marker method fares substantially better than permutation flooding, sometimes actually achieving a fundamental reduction of the exponential complexity This can be seen in Fig. 6 for the scheduler0 example. Note that permutation flooding does not achieve such a fundamental reduction (note that symmetry reduction by computing canonical forms has a very similar curve, with speedups dropping below 3 for $Card > 5$; see [TLSB07]).

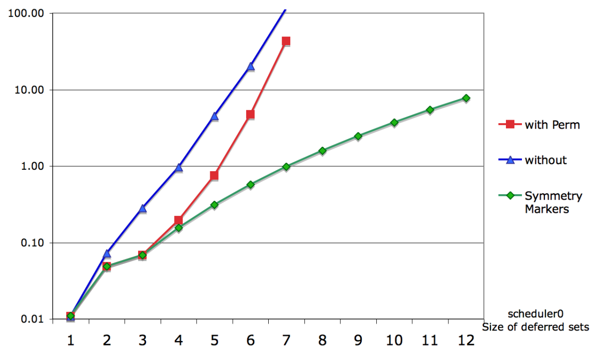

Fig. 6. Model Checking time (in seconds) for scheduler0.mch; log scale

5 Related and Future Work

Symmetry reduction in model checking has been studied extensively since the nineties [ES96,ID96,CEFJ96]. The two sources of symmetry mostly analyzed are *data symmetry* generally identified through the use of special data types, and structural symmetry due to concurrent (isomorphic) processes.

Two major works have initially studied data symmetry. Ip and Dill [ID96] introduced the *scalarset* datatype which is an integer subrange with restricted operations. These restrictions allow to identify symmetries in the state-space. Scalarset is implemented in the tool Murϕ [DDHY92]. The second precursor work in this line is the work of Clarke, Jha et al [CEFJ96,Jha96] which combines data symmetry with a BDD approach. The approach taken with the scalarset data type has been taken and extended in various works on untimed [BDH02,DMC05b] and timed systems [HBL$^+$03]. Data equivalence is also exploited by Jackson et al. [JJD98] in relational specifications where data and operations are specified as relations. Note that since it is relational, the language NP used by Jackson et al. is somehow, in the same family as Z, VDM and B.

In a pioneer work [ES96,ES95], Emerson and Sistla studied *structural symmetry*. They use concurrent systems of processes together with some communication topology using shared variables. They studied fairness in that setting. The tool SMC [SGE00] implements this theory. It is worth noticing that, except for the work of Jackson et al., symmetry is always specified by hand by the designer. Our approach does not require this; the symmetry arises naturally from the (common) use of deferred sets.

The problem of efficient identification of equivalent states was already discussed in [ES96] and a very simple hashing function invariant to symmetry was proposed as a first step to identify states equivalence classes. To our knowledge, our work is the first elaborate approach to replace the standard symmetry reduction method based on normalisation to an efficient approximation method.

Other works studied structural symmetry in concurrent systems and in particular with other kind of communication such as signal or message passing [MHB98,DM05,DMC05a].

Symmetry on the formula allows another kind of reduction and has been investigated in [MHB98].

The link between the finding of orbits and the graph isomorphism problems was studied in [CEFJ96]. The computation of a representative element for a global state can therefore be done by the powerful algorithm [McK81] and the nauty tool [McK] developed by McKay. The paper of Miller et al. [MDC06] gives a nice survey to symmetry in model checking.

In future we plan to adapt our results to improve automatic refinement checking [LB05]. Another promising work is to employ symmetry reduction when checking logical predicates containing existential or universal quantification, in order to cut down on the number of values that need to be tested for the quantified variables. We also plan to combine our symmetry markers with the graph canonicalisation approach, so that the canonical form only has to be computed when two states have the same symmetry marker.

In conclusion, we have presented a new approximate verification technique for B, employing symmetry induced by B's deferred sets. The technique computes symmetry markers for states of B machines and two states with the same symmetry marker are considered symmetric by the approximate verification algorithm. In our empirical evaluation we have shown that our technique is both very precise (very few non-symmetric states are identified) and very efficient, sometimes achieve a fundamental reduction of the underlying exponential verification complexity.

References

[Abr96] Jean-Raymond Abrial. *The B-Book*. Cambridge University Press, 1996.

[BCUL99] UK B-Core (UK) Limited, Oxon. *B-Toolkit, On-line manual*, 1999. Available at `http://www.b-core.com/ONLINEDOC/Contents.html`.

[BDH02] Dragan Bosnacki, Dennis Dams, and Leszek Holenderski. Symmetric spin. *STTT*, 4(1):92–106, 2002.

[CEFJ96] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des.*, 9(1-2):77–104, 1996.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*, pages 522–525, 1992.

[DM05] Alastair F. Donaldson and Alice Miller. Automatic symmetry detection for model checking using computational group theory. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 481–496. Springer, 2005.

[DMC05a] Alastair F. Donaldson, Alice Miller, and Muffy Calder. Finding symmetry in models of concurrent systems by static channel diagram analysis. *Electr. Notes Theor. Comput. Sci.*, 128(6):161–177, 2005.

[DMC05b] Alastair F. Donaldson, Alice Miller, and Muffy Calder. Spin-to-grape: A tool for analysing symmetry in promela models. *Electr. Notes Theor. Comput. Sci.*, 139(1):3–23, 2005.

[ES95] E. Allen Emerson and A. Prasad Sistla. Utilizing symmetry when model checking under fairness assumptions: An automata-theoretic approach. In Pierre Wolper, editor, *CAV*, volume 939 of *Lecture Notes in Computer Science*, pages 309–324. Springer, 1995.

[ES96] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.

[Fla01] Sarah Flannery. *In Code: A Mathematical Adventure.* Profile Books Ltd, 2001.

[HBL$^+$03] Martijn Hendriks, Gerd Behrmann, Kim Guldstrand Larsen, Peter Niebert, and Frits W. Vaandrager. Adding symmetry reduction to uppaal. In Kim Guldstrand Larsen and Peter Niebert, editors, *FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2003.

[Hol88] Gerard J. Holzmann. An improved protocol reachability analysis technique. *Softw., Pract. Exper.*, 18(2):137–161, 1988.

[ID96] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[Jha96] Somesh Jha. *Semmetry and Induction in Model Checking.* PhD thesis, School of Computer Science, Carnegie Mellon University, October 1996.

[JJD98] Daniel Jackson, Somesh Jha, and Craig Damon. Isomorph-free model enumeration: A new method for checking relational specifications. *ACM Trans. Program. Lang. Syst.*, 20(2):302–343, 1998.

[KK04] William Kocay and Donald L. Kreher. *Graphs, Algorithms and Optimization.* Chapman & Hall/CRC, 2004.

[KS99] Donald L. Kreher and Douglas R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, Search.* CRC Press, 1999.

[LB03] Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[LB05] Michael Leuschel and Michael Butler. Automatic refinement checking for B. In Kung-Kiu Lau and Richard Banach, editors, *Proceedings ICFEM'05*, LNCS 3785, pages 345–359. Springer-Verlag, 2005.

[LBST07] Michael Leuschel, Michael Butler, Corinna Spermann, and Edd Turner. Symmetry reduction for b by permutation flooding. In *B'2007, the 7th Int. B*

Conference - Tool Session, volume 4355 of *LNCS*, Besancon, France, January 2007. Springer. To appear.

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

[McK81] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[MDC06] Alice Miller, Alastair Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3):8, 2006.

[MHB98] Gurmeet Singh Manku, Ramin Hojati, and Robert K. Brayton. Structural symmetry and model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 159–171. Springer, 1998.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[Sch01] Steve Schneider. *The B-method, an introduction.* Computer Science - The Cornerstones of Computing Series. Palgrave, macmillan, 2001.

[SGE00] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. Smc: a symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133–166, 2000.

[Ste96] France Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com/index_uk.html`.

[TLSB07] Edd Turner, Michael Leuschel, Corinna Spermann, and Michael Butler. Symmetry reduced model checking for B. In *Proceedings Symposium TASE 2007 (Theoretical Aspects of Software Engineering)*, Shanghai, China, June 2007. IEEE. To appear.