

Centre Fédéré en Vérification

Technical Report number 2007.92

Efficient Approximate Verification of B via Symmetry Markers

Michael Leuschel, Thierry Massart



This work was partially supported by a FRFC grant: 2.4530.02

<http://www.ulb.ac.be/di/ssd/cfv>

Efficient Approximate Verification of B via Symmetry Markers

Michael Leuschel¹, Thierry Massart²

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.

Keywords: Model Checking, Symmetry, B-Method, Formal Methods, Logic Programming.

AMS Classification: 68N30 Computer science, Software, Mathematical aspects of software engineering (specification, verification, metrics, requirements, etc.), 68Q60 Computer science, Theory of computing, Specification and verification (program logics, model checking, etc.), 68R10 Computer science, Discrete mathematics in relation to computer science, Graph theory, 03B70 Mathematical logic and foundations, General logic, Logic in computer science, 68N17 Computer science, Software, Logic programming.

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 Mur ϕ [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 ab-

stract 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 :=  $\emptyset$ 
OPERATIONS
  res  $\leftarrow$  Login = ANY s WHERE s  $\in$  Session  $\wedge$  s  $\notin$  active THEN
    res := s || 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(Session1)*, 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.

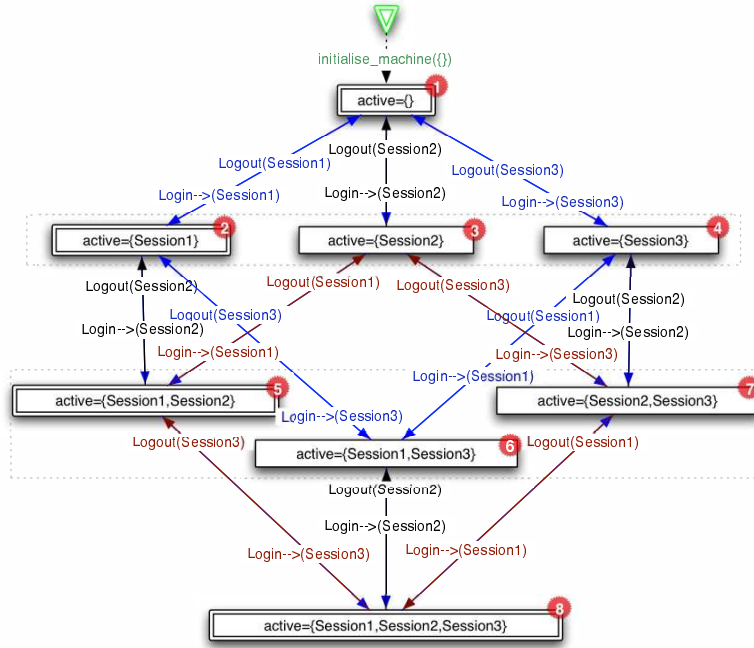


Fig. 2. Full state space; representatives are marked by double boxes

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 (\rightarrow) $lFork$ 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: $\forall st.(st \subset Phil \wedge st \neq \emptyset \Rightarrow rFork^{-1}[lFork[st]] \neq st)$.

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 into 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 reduc-

```

MACHINE Philosophers
SETS Phil; Forks
CONSTANTS lFork, rFork
PROPERTIES
  lFork ∈ Phil  $\mapsto$  Forks ∧
  rFork ∈ Phil  $\mapsto$  Forks ∧
  card(Phil) = card(Forks) ∧
   $\forall pp.(pp \in Phil \Rightarrow lFork(pp) \neq rFork(pp))$ 
VARIABLES taken
INVARIANT
  taken ∈ Forks  $\leftrightarrow$  Phil ∧
   $\forall xx.(xx \in dom(taken) \Rightarrow (lFork(taken(xx)) = xx \vee rFork(taken(xx)) = xx))$ 
INITIALISATION taken := ∅
OPERATIONS
  TakeLeftFork(p, f) =
    PRE p ∈ Phil ∧ f ∈ Forks ∧ f ∉ dom(taken) ∧ lFork(p) = f THEN
      taken(f) := p
    END;
  TakeRightFork(p, f) =
    PRE p ∈ Phil ∧ f ∈ Forks ∧ f ∉ dom(taken) ∧ rFork(p) = f THEN
      taken(f) := p
    END;
  DropFork(p, f) =
    PRE p ∈ Phil ∧ f ∈ Forks ∧ f ∈ dom(taken) ∧ taken(f) = p THEN
      taken := f  $\triangleleft$  taken
    END
END

```

Fig. 3. Dining Philosophers specification

tion 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 test-

ing), 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 $\langle c_1, \dots, c_n \rangle$ of value of its variables or constants v_1, \dots, v_n where an order is fixed between them. We also denote multisets by using $\{ | \dots | \}$ and multiset union by \uplus . We denote sequences by $\langle \dots \rangle$. The concatenation of two sequences α and β is denoted by $\alpha.\beta$. If $B = \{ | \beta_1, \dots, \beta_n | \}$ is a multiset of n sequences and α a sequence, we also define $\alpha.B = \{ | \beta'_1, \dots, \beta'_n | \}$ 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,
- $paths(d, e) = \emptyset$ otherwise.

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

$$- paths(d, s) = \{ | \langle v_i \rangle.paths(d, c_i) | s = \langle c_1, \dots, c_n \rangle | \}$$

$paths(d, s)$ 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 $paths(d, s) = \{ | \langle r, el, left \rangle, \langle r, el, right \rangle, \langle r, el, right \rangle | \}$.

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, \dots, v_n . We define the marking function m , computing markers for data values as follows:

$$- m(s) = \{ | v_i \mapsto m_s(c_i) | s = \langle c_1, \dots, c_n \rangle | \}$$

where m_s is inductively defined by:

- $m_s(e) = e$ if $e \in \mathbb{Z}$ or $e \in \text{BOOL}$ or $e = \emptyset$ or $e \in \mathcal{E}$,
- $m_s(e) = (m_s(x) \mapsto m_s(y))$ if $e = (x \mapsto y)$ is a pair,
- $m_s(e) = \{ | m_s(e_1), \dots, m_s(e_k) | \}$ if $e = \{e_1, \dots, e_k\}$ 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 $s_2 = \langle d_2, \{d_2 \mapsto d_1\} \rangle$ 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.

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 $m_{s_1}(d_1) = \{ | \langle x, el, left \rangle, \langle y, el \rangle | \}$, $m_{s_2}(d_1) = \{ | \langle y, el \rangle | \}$, $m_{s_2}(d_2) = \{ | \langle x, el, left \rangle | \}$. 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:

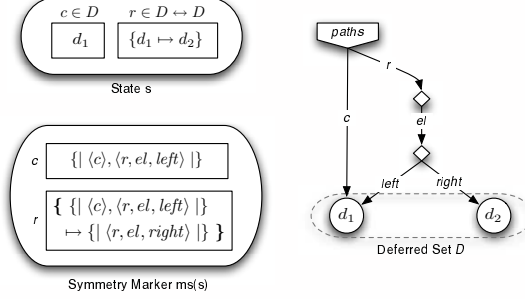


Fig. 4. Illustrating the paths for deferred set elements

Let $NonSym$ be the smallest set satisfying $((\mathbb{Z} \cup \mathbb{B} \cup \mathcal{E} \cup \{\emptyset\}) \subseteq NonSym) \wedge (\forall x, y : x \in NonSym \wedge y \in NonSym \Rightarrow (x, y) \in NonSym)$. 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 $e = (n \mapsto x) \wedge n \in NonSym \wedge x \notin NonSym$
- $paths(d, e) = \langle leftrightight \rangle.paths(x)$ 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 $\{D_1, \dots, D_i\}$ such that there exists a permutation f over $\{D_1, \dots, D_i\}$ where $s_2 = f(s_1)$. Then for any element $d_1 \in \{D_1, \dots, D_i\}$ we have $paths(d_1, s_1) = paths(f(d_1), s_2)$.*

Proof. (Sketch) 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

ERROR: ioerror
OFFENDING COMMAND: image

STACK:

