Noname manuscript No.
(will be inserted by the editor)

Efficient Approximate Verification of B and Z Models via Symmetry Markers

Michael Leuschel · Thierry Massart

the date of receipt and acceptance should be inserted later

Abstract We present a new approximate verification technique for falsifying the invariants of 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, so that symmetric states are guaranteed to have the same marker (but not the other way around). The falsification 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 and therefore provides an efficient and complete verification method. Finally, we show that the technique can be applied to Z models as well.

Keywords Model Checking · Symmetry · B-Method · Formal Methods · Logic Programming

Mathematics Subject Classification (2000) 68N30 · 68Q60 · 68R10 · 03B70 · 68N17

1 Introduction

The B-method [1] is a theory and methodology for formal development of computer systems based on set theory and predicate logic. It is used in industry in a range of critical domains.

Invariant properties and refinement relations for B specifications can be expressed and then proven by the semi-automated theorem provers within tools like Atelier-B [49] and B4Free [11], the B-toolkit [4] or the Rodin platform [3] for Event-B [2]. Recently,

Part of this research has been supported by the EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

M. Leuschel
Institut für Informatik, Universität Düsseldorf
E-mail: leuschel@cs.uni-duesseldorf.de

T. Massart
Université Libre de Bruxelles (U.L.B.)
E-mail: tmassart@ulb.ac.be

PROB [33,36] 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* [10]. Indeed, often a system to be checked has a large number of states with symmetric behaviour, meaning that there are classes of states where each member of the class behaves like every other member of the class. Symmetry is particularly prominent in B because of the common use of sets of unnamed elements, called *deferred sets* in “classical” B and *carrier sets* in Event-B.

In previous work [35] we have presented a symmetry reduction technique, called permutation flooding, which ensures that only one representative per symmetry equivalence class 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 [24]. We define a hashing function, which can be computed efficiently and which returns the same value for symmetric states. 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 do not have a complete verification, but only an *approximate* verification technique (in the sense that non-symmetric states can obtain the same hash value); but a very fast one. However, if the algorithm finds a counter example, it is guaranteed to be genuine. Hence, the main objective of our technique is “falsification”[5], i.e., proving that a B model does not preserve its invariant. 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 cases but one, no loss of precision was induced (and all symmetry classes were visited) and a fundamental reduction of complexity was achieved for some examples.

The techniques presented in this paper can also be applied in the context of checking LTL properties [45]. Furthermore, the techniques also apply to Z models. Indeed, in [44] we have extended PROB to work also for Z specifications and the present technique can also be used in that setting, where given sets play the rôle of deferred sets in B (and carrier sets in Event-B). In principle, the technique applies to any formal specification language rooted in set theory and predicate logic, where sets of anonymous elements are commonly used. In the paper we also examine our technique on two case studies from the Z literature, and show that it works equally well in that setting. In recent work [8] we have also used the present work as inspiration for an approximate technique in the context of Promela.

In this paper, we give in Sect. 2, a brief introduction to B and symmetry reduction 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. 4 our symmetry reduction technique. We have integrated our method into the PROB tool. In Sect. 5 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. 7 related work in the field of symmetry reduction and model checking, particularly the tools Murϕ [27] and SMC [47].

This is an extended and revised version of [37]. E.g., the present paper contains a complete formalisation of the algorithm, with full proofs. We have also extended the experimental section, and provide more discussions about related work. The application to Z models is also new.

2 An Overview of Symmetry in B

In this section we present how symmetry arises in B formal models and how it can be exploited. The crucial aspect are the underlying set theory and predicate logic together with the notion of deferred sets. As such, our approach could as well have been presented in the context of Z. We return to this issue in Section 6.2.

B is based on the notion of *abstract machine*. The variables of an abstract machine can be either elements of basic sets (Boolean values, integers and user-defined sets), 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 with set theory and arithmetic. (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. Also, we will concentrate our presentation on "classical" B, but our techniques applie in exactly the same way to Event-B.)

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 machine 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 [33], which can also check if every reachable state satisfies the invariant. Unfortunately, this computation can be very expensive, but 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) [35].

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 [35] 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.

We will present symmetry induced by permuting deferred set elements more formally in Section 3.2. In the meantime, the following simple example from [35] 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 session management model in B

This machine contains the deferred set *Session*. The variable *active* contains an active set of sessions and is initialised with the empty set. The machine contains two operations:

– The *Login* operation which non-deterministically choses a session s which is not yet used and adds it to *active*. The operation also returns a value, namely the chosen session s.
– The *Logout* operation, which receives as parameter a session s and removes s from the set of active sessions. Note that this operation contains a precondition: it can only be called with an active session as parameter s.

Once the cardinality of the deferred set *Session* has been fixed, the state space of this machine can be obtained by starting out from the initial state and then repeatedly applying the operations. With a cardinality of 3 for *Session*, the full state space for this machine has 8 states (one for each possible subset of Session).

Fig. 2 shows the full state space of the "login" example, where we have denoted the three elements of *Session* by *Session1*, *Session2*, *Session3*. One can observe that the possible behaviours of a state depend solely on the cardinality of the set *active* and not on the identity of the elements in this set. In other words, 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*).

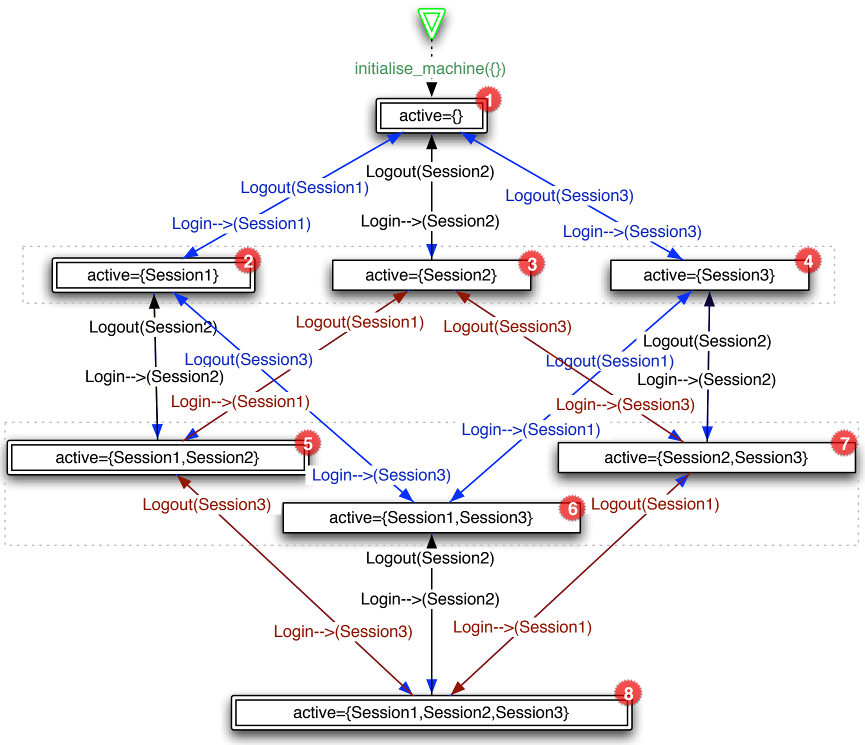


Fig. 2 Full state space; representatives are marked by double boxes

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, as depicted in Figure 3, where only one representative per equivalence class is kept. We will formalise this fact later in Section 3.2, leading up to Theorem 1.

Size of the State Space For the *LoginVerySimple*, the size of the unreduced state space is $1 + 2^n$, where n is n is the cardinality of the deferred set *Session*. For example, in Fig. 2 we have $1 + 2^3 = 9$ reachable states (including the root state before initialisation). The size of the reduced state space is just $1 + n$, e.g., 4 in Fig 3. We thus get an exponential reduction of the size of the state space.

Dining philosophers Another example, with more involved data structures and using constants, can be found in Fig. 4. This will allow us to explain how symmetries can be detected in general. Fig. 4 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 ($\nrightarrow$) recording for each

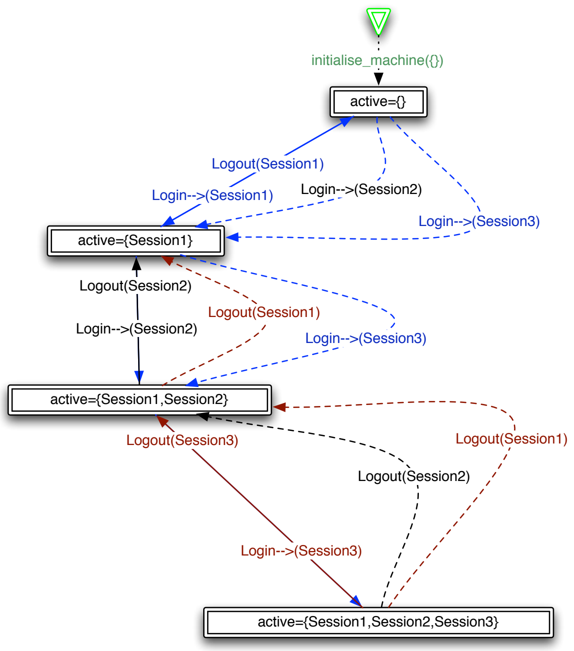


Fig. 3 Symmetry Reduced state space; dashed lines represent "redirected" operations

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

Size of the State Space For the *Philosophers* machine of Fig. 4 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 17,713 states. As we will see later, a lot of those states are symmetric. Indeed, the symmetry reduced state space has only 48 states.

```
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
 $taken \in Forks \mathrel{+\!\!\!\!\rightarrow} Phil \wedge$
 $\forall xx.(xx \in dom(taken) \Rightarrow (lFork(taken(xx)) = xx \vee rFork(taken(xx)) = xx))$
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
              $taken := f \mathbin{\triangleleft\!\!\!-} taken$
  END
END
```

Fig. 4 Dining Philosophers specification

3 Formal Definition of Symmetry in B Models

3.1 States and Statespace

We first need to formalise the notation of a state of a B machine. The values of variables in B expressions and predicates are inductively defined to be either elements of given sets (including the set $BOOL$ of Boolean values and the set $\mathbb{Z}$ of integers), pairs of values, or sets of values. We denote the set of all possible values as $DATA$. Note that for pairs, we use the B notation: a pair consisting of the two components x and y is denoted by $x \mapsto y$.

A state s of a B machine is denoted by a tuple $\langle c_1, \ldots, c_n \rangle$ of values of its ordered variables or constants $v_1, \ldots, v_n$ (denoted V) where the order is fixed a priori. The set of all states is denoted by $STATE$.

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. (Some additional details may be found in [34].) B models also contain an initialisation substitution, which can be characterised by a predicate $Init(V)$. This induces a labelled transition system as follows:

Definition 1 The *state space* of a B machine is defined to be the labelled transition system (S, S_0, T), with

– the set of states $S = STATE$,
– the set of initial states $S_0 = \{s \in STATE \mid Init(s)\}$,
– the transition relation $T = \{(s1, L, s2) \mid s_1 \rightarrow^M_L s_2\}$.

The *reachable states* of a labelled transition system (S, S_0, T) is the set of states $\bigcup_{i>0} T^i(S_0)$, where T^i is inductively defined by $T^1(S) = S$ and $T^{i+1}(S) = T(T^i(S))$ for $i > 0$.

Figure 2 depicts the state space of the machine from Fig. 1, where we have linked the initial states of the B machine to a single special root node.

3.2 Symmetry induced by Permutation of Deferred Set Elements

We recall the definitions from [35], where symmetry induced by deferred set elements is described.

Definition 2 Let $\mathcal{DS}$ be a set of disjoint sets. A *permutation* f *over* $\mathcal{DS}$ is a total bijection from $\cup_{S \in \mathcal{DS}} S$ to $\cup_{S \in \mathcal{DS}} S$ such that $\forall S \in \mathcal{DS}$ 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 3 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:
– $f(x) = x$ if $x : \mathbb{Z}$ or $x : BOOL$ or $x : ES_j$ (for some j)
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
– $f(\langle v_1, \ldots, v_k\rangle) = \langle f(v_1), \ldots, f(v_k)\rangle$

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

Definition 4 Let s, s' be two states of a B Machine with deferred sets $DS_1, \ldots, DS_k$. The state s' is a *permutation of the state* s iff there exists a permutation f over $\{DS_1, \ldots, DS_i\}$ such that $s' = f(s)$. The set of all permutation states of s is called the *orbit* of s, denoted by $\theta(s)$.

In [35] it is proven that a predicate is true in a state s iff it is true in all permutation states of s. This in turn can be used to prove that permutations induce a symmetry in the state space of a B machine:

[1] 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 $f(c) = c$.

Theorem 1 *Every state permutation f over the deferred sets of a B machine M with invariant I satisfies*

– $\forall s \in S : s \models I$ *iff* $f(s) \models I$

– $\forall s_1 \in S, \forall s_2 \in S$: $\quad s_1 \rightarrow^{M}_{op.a.b} s_2 \quad \Leftrightarrow \quad f(s_1) \rightarrow^{M}_{op.f(a).f(b)} f(s_2)$.

The first point of Theorem 1 establishes invariance of f wrt the invariant predicate I, and point two establishes that f is an automorphism.

The idea of "traditional" symmetry reduction is to apply the model checking to the symmetry quotient of the state space, also called the symmetry reduced state space. It is obtained by choosing one representative per orbit $\theta(s)$, denoted by $rep(\theta(s))$, defined as follows:

Definition 5 A representative function for a B machine, is a function *rep* from sets of states to states, such that for all sets of states $S \subseteq STATE$, we have $rep(S) \in S$. Given a B machine M and a representative function *rep*, we define the *symmetry reduced state space* of M wrt *rep* to be the labelled transition system (S', S'_0, T'), with

– the set of states $S' = \{rep(\theta(s)) \mid s \in STATE\}$,
– the set of initial states $S'_0 = \{rep(\theta(s)) \mid Init(s)\}$,
– and where the transition relation is defined by:
$(s_1, L, s_2) \in T'$ iff $s_1 \in S'$ and there exists s'_2 with $s_2 = rep(\theta(s'_2))$ and $s_1 \rightarrow^{M}_{L} s'_2$.

Figure 3 contains the symmetry quotient of the state space in Fig. 2, where the states 1,2,5,8 have been used as representatives.

As a corollary of Theorem 1 we have:

Corollary 1 *Let L be the state space of a B machine with invariant I and let L' be a symmetry reduced state space. There exists a reachable state s of L such that $s \models \neg I$ iff there exists a reachable state s' of L' such that $s' \models \neg I$.*

3.3 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., [10][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 many classes of graphs [30]. The most efficient general purpose graph isomorphism program is NAUTY [40]. In related work [50], inspired by NAUTY, we have implemented a *canonicalisation function* for B states viewed as graphs, i.e. a procedure which maps each state to the representative of its orbit, also called the *canonical form.* This has been further refined in [48], actually making use of NAUTY itself to detect the symmetries.

3.4 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 [35] 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 canonization method. Further information on these permutation functions, and the soundness results of the symmetries, can be found in [35].

4 Symmetry Markers

Even with symmetry reduction via canonization 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 Holzmann's successful bitstate hashing technique [24] 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 method), 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 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).

4.1 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 *see s* as a graph and transform it into a marker by replacing the deferred set elements by so-called *vertex-invariants*.

In graph theory, an invariant [31][Sect. 7.2] is a function which does not depend on the presentation of the graph. A vertex-invariant [40] *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.

4.1.1 Symmetry Markers

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, which is not affected by permutation. Hence, the same structural information will be computed in all symmetric states.
   For this, we compute the multiset of *paths* that lead to an occurrence of d in s. This is formalised in Def. 6 below.
2. Replace all deferred set elements by the structural information computed above. This is formalised in Def. 7.

Fig. 5 (b) provides, for the deferred set $D = \{d_1, d_2\}$, a graphical representation of the state $s = \langle d_1, \{d_1 \mapsto d_2\}\rangle$ given in Fig.5 (a) of a machine with the ordered variables c, r where $c \in D$ and $r \subseteq D \times D$. Diamonds are used to denote larger, non-atomic expressions (such as $d_1 \mapsto d_2$).

Below we denote the elements of the deferred (resp. enumerated) sets by $\mathcal{D}$ (resp. by $\mathcal{E}$). For a set V of variables and constants, we also denote by $\mathit{MPATHS}(V)$ all multisets of sequences over $V \cup \{\mathit{left}, \mathit{right}, \mathit{el}\}$.

We also denote multisets by using $\{|\ \ldots\ |\}$, multisets union by $\uplus$ and 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 $\alpha.B = \{|\ \alpha.\beta_1, \ldots, \alpha.\beta_n\ |\}$.

Definition 6 Let $d \in \mathcal{D}$ be a deferred set element and e a data value of a variable or constant of a B machine. We define the function $\mathit{paths} : \mathit{DATA} \times \mathcal{D} \mapsto \mathit{MPATHS}(\varnothing)$ as follows:

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

For a state $s = \langle c_1, \ldots, c_n\rangle$ of a B machine with variables and constants V (ordered as $v_1, \ldots, v_n$) we define $\mathit{vpaths} : \mathit{STATE} \times \mathcal{D} \mapsto \mathit{MPATHS}(V)$ by

– $\mathit{vpaths}(s, d) = \{|\ \langle v_1\rangle.\mathit{paths}(c_1, d) \ldots \langle v_n\rangle.\mathit{paths}(c_n, d)\ |\}$

$\mathit{vpaths}(s, d)$ 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).

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

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

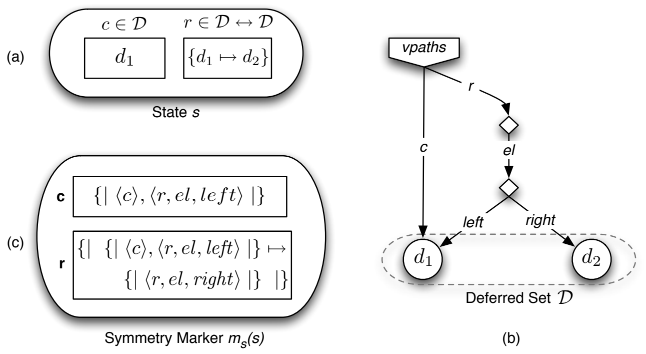
Definition 7 Let s be the state of a B machine with ordered variables and constants $v_1, \ldots, v_n$. We define the *marking function* m, computing markers for data values as follows:

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

where m_s is inductively defined by:

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

Fig. 5 (c) completes the example of Fig. 5 with the symmetry marker corresponding to the state s. Note that the state $s_2 = \langle d_2, \{d_2 \mapsto d_1\}\}$ is symmetric to the state s of Fig. 5 (the permutation is $f = \{d_1 \mapsto d_2, d_2 \mapsto d_1\}$) and the symmetry markers are identical.



(a)
$c \in \mathcal{D}$
$r \in \mathcal{D} \leftrightarrow \mathcal{D}$
d_1
$\{d_1 \mapsto d_2\}$
State s
(c)
c
$\{\mid \langle c \rangle, \langle r, el, left \rangle \mid\}$
r
$\{\mid \{\mid \langle c \rangle, \langle r, el, left \rangle \mid\} \mapsto \{\mid \langle r, el, right \rangle \mid\} \mid\}$
Symmetry Marker $m_s(s)$
vpaths
r
c
el
left
right
d_1
d_2
Deferred Set $\mathcal{D}$
(b)

Fig. 5 State s, graphical view and its symmetry marker $m_s(s)$

The above generalises the in- and out-degree: the marker of a deferred element d records all the *positions* where it is used in the state seen as a data structure, and in particular if d is used within a relation r (i.e., a set of pairs) representing a directed graph, then m_s will effectively count the number of incoming and outgoing edges in r.

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 $m_{s_1}(d_1) = \{\mid \langle x, el, left \rangle, \langle y, el \rangle \mid\}$, $m_{s_2}(d_1) = \{\mid \langle y, el \rangle \mid\}$, $m_{s_2}(d_2) = \{\mid \langle x, el, left \rangle \mid\}$. 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. 6 stores more information in the symmetry marker to avoid collisions in those cases:

First, we define the set of non-symmetrical *NonSym* datavalues as follows:

Definition 8 *NonSym* is the smallest set satisfying

– $\mathbb{Z} \cup BOOL \cup \mathcal{E} \cup \{\varnothing\} \subseteq NonSym$

– $\forall x, y : x \in \mathit{NonSym} \wedge y \in \mathit{NonSym} \Rightarrow (x, y) \in \mathit{NonSym}$.

We extend Def. 6 by replacing the second rule by the following rules:

2a. $\mathit{paths}(e, d) = \langle \mathit{to}, n\rangle.\mathit{paths}(x, d)$ if $e = (x \mapsto n) \wedge n \in \mathit{NonSym} \wedge x \notin \mathit{NonSym}$
2b. $\mathit{paths}(e, d) = \langle \mathit{from}, n\rangle.\mathit{paths}(x, d)$ if $e = (n \mapsto x) \wedge n \in \mathit{NonSym} \wedge x \notin \mathit{NonSym}$
2c. $\mathit{paths}(e, d) = \langle \mathit{leftright}\rangle.\mathit{paths}(x, d)$ if $e = (x \mapsto x) \wedge x \notin \mathit{NonSym}$.
2d. $\mathit{paths}(e, d) = \langle \mathit{left}\rangle.\mathit{paths}(x, d) \uplus \langle \mathit{right}\rangle.\mathit{paths}(y, d)$ if $e = (x \mapsto y) \wedge x \notin \mathit{NonSym} \wedge y \notin \mathit{NonSym} \wedge x \neq y$.

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 of m_s is indeed not affected by permutation of deferred set elements:

Proposition 1 *Let s be a state for B Machine with deferred sets $\{D_1, \ldots, D_i\}$ and f a permutation over $\{D_1, \ldots, D_i\}$. Then for any element $d \in \{D_1, \ldots, D_i\}$ we have $\mathit{paths}(d, s) = \mathit{paths}(f(d), f(s))$.*

Proof In Appendix A.

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

Corollary 2 *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 $m(s_1) = m(s_2)$.*

4.2 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. 6 gives a graphical representation of these two states, where we have represented each pair by an arrow between a philosopher and a fork. It is not straightforward to see why these two states are not symmetric.[2] Let us consider the predicate $\forall p.(p : \mathit{Phil} \Rightarrow \mathit{taken}(\mathit{lFork}(p)) = p)$. 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 record cycles of length 2 together with the links followed (like the ones occurring in Fig. 6 (a) following links *lFork* and *taken*: *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 element. For example, the states $s_1 = \langle\{d_1 \mapsto d_2, d_2 \mapsto d_3, d_3 \mapsto d_4, d_4 \mapsto d_1\}\rangle$ and $s_2 = \langle\{d_1 \mapsto d_2, d_2 \mapsto$

[2] And actually for the current machine in Fig. 4 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.

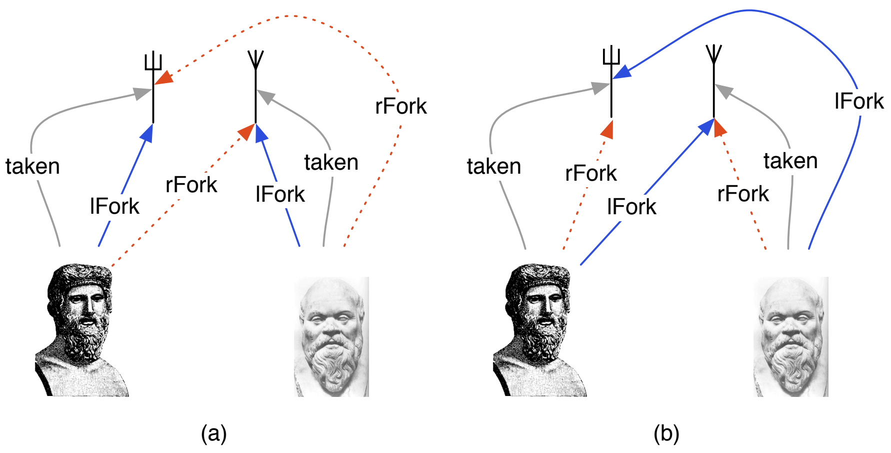


Fig. 6 Two states in collision for the symmetry markers method

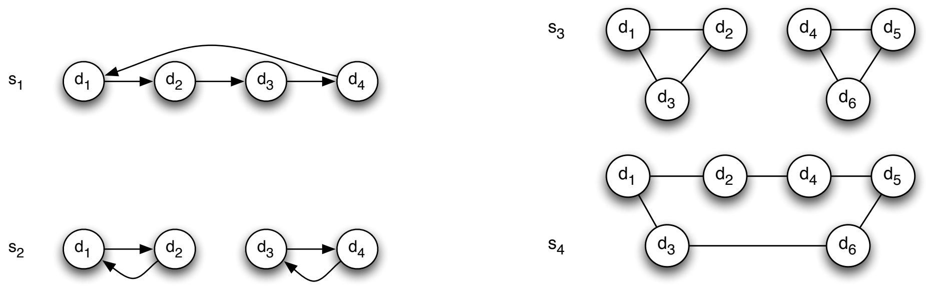


Fig. 7 Further collisions for the symmetry markers method

$d_1, d_3 \mapsto d_4, d_4 \mapsto d_3\}\rangle$ have same symmetry marker (assuming $D = \{d_1, d_2, d_3, d_4\}$ is a deferred set; see the left of Figure 7 for a graphical representation) but they are not symmetric.

As we can encode a symmetric relation r as a set of sets $\{\{x, y\} \mid (x \mapsto y) \in r\}$, this means that sets of sets of deferred elements may also lead to imprecision. For example, the right of Figure 7 shows two symmetric relations s_3 and s_4 which are not distinguished by symmetry markers.

Still, in the following cases where pairs of deferred values is allowed, 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:*

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

5. or a pair of values inductively satisfying one of the above requirements.

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

Proof In Appendix A.

As a corollary 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 (and use it, e.g., in the invariant since otherwise PROB changes its specification to deferred set), our method is precise but of course has less symmetry. Finally, observe that these conditions are cheap to check and have been implemented in our tool (PROB warns the user if the conditions do not apply).

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:

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

This extension has not been implemented in our tool.

5 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. The tool also checks whether the condition in Prop. 2 applies (and warns the user if it does not and no counter example was found).

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 (mark) 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., [22]). Sched0 and Sched1 are the machines presented in [34]. Peterson is the specification of the mutual exclusion protocol for n processes as defined in [43]. USB is a specification of a USB protocol, developed by the French company ClearSy. Towns is a specification from the Schneider B Book [46]; here the overhead is in the closure computation of a query operation. Dining is the dining philosopher example presented above. Finally, we have also added one example without any symmetry (no deferred sets), the Volvo vehicle function from [33].

The column "card" indicates the cardinality that was used for the deferred sets. The column "Nodes" in Table 1 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.[3]

[3] Note that neither SICStus Prolog nor PROB take advantage of multiple processors. The Volvo examples were actually run on SICStus Prolog 4.0.7 and PROB 1.3.1.

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	mark	wo	flood	mark	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
Sched0	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
Sched1	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
Towns	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	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
Dining	2	123.64	5.99	0.15	11809	26	20	799.36	38.73
Volvo	-	8.77	8.68	9.21	1361	1361	1361	0.95	0.94

The results are very good, and for most examples the symmetry marker is much faster than 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. 8 for the scheduler0 example. Note that permutation flooding does not achieve such a fundamental reduction. The same is actually true for symmetry reduction by computing canonical forms, as implemented in [50], with speedup factors dropping below 3 for scheduler0 with $card > 5$. The more recent implementation in [48] based upon NAUTY is considerably faster (only 28 % slower than symmetry markers for scheduler0 with $card = 7$), but for most examples roughly half the speed of our symmetry marker method.

The Volvo example shows that the overhead of the marker computation is relatively small. Indeed, there are no deferred sets in the Volvo example, and hence the computed markers correspond to the original state and there is no reduction in the state space. As we can see, we pay about a 5% performance penalty for computing the markers.

Also note that in all but one case (Philosophers), the symmetry marker method was precise; for this last, our method is very efficient but indeed does not provide an exhaustive verification. The criterion from Prop. 2 applies to all but three cases (Philosophers, Russian and Towns). Note that the Russian model has one variable *has_keys* of type `POW(PERSONS*POW(KeyIDs))`, where *KeyIDs* is a deferred set but *Person* is enumerated. In principle our method could have been imprecise. Indeed, a function f from *NonSym* to sets of deferred elements can be "translated" to a symmetric binary relation R between deferred set elements as follows: $R = \{x, y \mid \exists n \in dom(f) \wedge \{x, y\} \subseteq f(n)\}$. As we have seen in the right of Figure 7, such a relation can lead to collisions. But in this model, all the sets in the domain of *has_keys* are disjoint, which means that no collisions occur.

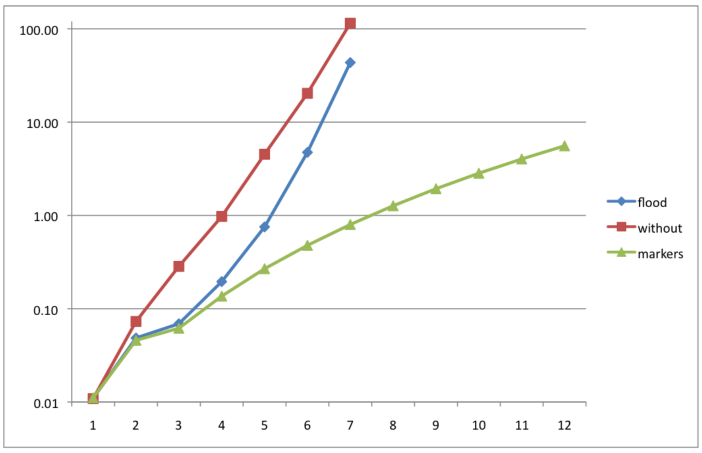


Fig. 8 Model Checking time (in seconds) for scheduler0.mch; log scale

Space complexity The theoretical space complexity of the marker method is function of the possible size of a marker and the number of states. The worst case is when no symmetry exists. In that case, since in our method, each deferred set element d is represented by the set of paths to d, a marker takes more memory space than a state, and the space complexity is worse than for the other methods. In practice, the space taken by each method is roughly proportional to the respective number of nodes in Table 1. Note, however, that flooding also adds "virtual" nodes. Hence, flooding will take almost the same space than the model checking without symmetry reduction (slightly less: because there are less transitions).

For instance, for scheduler1 and cardinality of 5, we have that the original model checking yields 5201 nodes, flooding generates 5120 virtual nodes and 81 "real" nodes and our markers method generates 81 nodes. The actual measured space consumption is 4667 kB for the original, 3197 kB for flooding and 78 kB for the markers method.

Figure 9 shows the memory consumption for storing the statespace for the scheduler0 example.

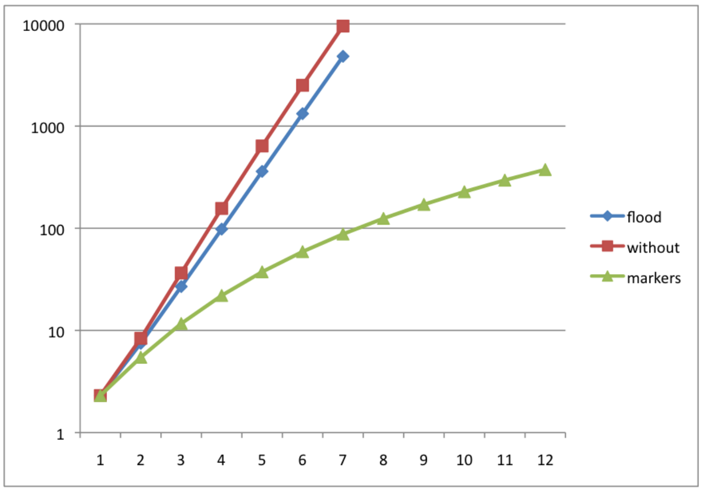


Fig. 9 Memory consumption (in kBytes) for the statespace of scheduler0.mch; log scale

6 Comparison with other Tools

To our knowledge, the only other model checker for B is [39]. However, it is not available for download and does not include any state space reduction techniques.

6.1 Spin

A comparison with the explicit state model checker Spin [25,26,6] for the Promela language and its partial order reduction can be found in [32]. While in principle Spin can deal with much larger state spaces as PROB, the exploitation of symmetry means that our algorithm was often much more efficient on high-level B models than Spin on the equivalent low-level models in Promela. In addition to partial order reduction, one can also use symmetry reduction for Spin, e.g., by using the SymmSpin tool [7] or TopSpin tool [17].

However, compared to PROB's approach to symmetry, we can make the following observations:

1. In Promela the user has to declare the symmetry: if he or she makes a mistake the verification procedure will be unsound; (there is, however, the work [16] which automatically detects some structural symmetries in Promela). In B symmetries can be inferred automatically very easily (by looking at the deferred sets).

2. Symmetry is much more natural and prevalent in B and we can take advantage of partial symmetries (see, e.g., the generic dining philosophers example from Figure 4) and one can have multiple symmetric types (for SPIN typically only a single scalarset, namely the process identifiers, is supported).

To further illustrate point 2, [32] presented a B model of a server farm with multiple deferred sets as well as the corresponding Promela model. To the best of our knowledge, this Promela model cannot be put into a form so that TopSPIN can exploit the symmetries. For cardinality of 7 (i.e., 7 servers and 7 users), the best result obtainable with SPIN in [32] ran in about 20 seconds. For cardinality of 8 it was not possible to verify the model using SPIN. Complete model checking with our hash marker method takes 0.77 seconds (on the same hardware that was used in [32]), generating 46 representative states. For a cardinality of 9, our new algorithm takes 1.16 seconds with our hash marker method, generating 56 representative states. Our hash marker method was again precise in all cases. In summary, the symmetry that can be inferred and exploited in the high-level B model leads to a dramatic reduction in model checking time compared to a low-level model.

6.2 Z Models and Z2SAL

In this subsection we apply our technique on Z models, and compare the results with existing tools for Z. Unfortunately, there does not seem to be a Z model checker available for download and experimentation. However, an alternate model checker for Z, translating Z to SAL, is under development [14,13,12]. Those papers also contain two Z models, which we will use for our experiments.

[14] contains a simple Z model of an organisation with members and people trying to join. The paper also provides timings for model checking several LTL formulas. As our tool can also handle Z specifications and LTL formulas, we were able to load the exact same case study and model check it. In [14], the validating the third property (that we always have `G {card(waiting)+card(member)<=3}`) took 3 seconds using the best translation to SAL, and 12 hours using canonical and original encodings in SAL. PROB takes 0.044 s without symmetry and 0.015 s with hash symmetry to validate the same property. In Figure 11 we show how our techniques fares on this model for various sizes of the given set *NAME*. Note that these experiments were run on the same hardware as above, but using PROB 1.3.2. We have performed exhaustive model checking[4] without symmetry and with symmetry markers. We have also applied the flooding technique, confirming that our symmetry markers are precise for this model (as expected by the application of Proposition 2). We can see that the performance advantage of symmetry markers increase with the size of *NAME*. For cardinality 7, the symmetry markers are more than 50 times faster than PROB without symmetry. We can also observe that the flooding technique is less efficient, and suffers a drastic drop in performance for size 7.

Some new experimental results of Z2SAL are reported in [13,12] for a video shop case study. We were again able to load the exact same case study and model check it. Figure 11 contains the results for exhaustively model checking the specification, for various cardinalities of the given sets *PERSON* and *TITLE* (and using a value of

[4] This amounts to deadlock checking here, as Z specifications preserve the invariant by definition.

Cardinality	Time	States	Edges	Time	States	Edges	Time	States
NAME	(without symmetry)			(symmetry markers)			(flood)	
3	0.03 s	29	149	0.01 s	12	55	0.02 s	12
4	0.54 s	165	1,196	0.12 s	33	218	0.14 s	33
5	1.45 s	733	6,761	0.14 s	67	571	0.31 s	67
6	4.84 s	2,191	23,892	0.22 s	88	896	1.61 s	87
7	16.10 s	6,565	82,263	0.31 s	112	1,326	25.70 s	112

Fig. 10 Experiments with the organisation Z model from [14]

Cardinality		Time	States	Edges	Time	States	Edges	Time
PERS.	TIT.	(without symmetry)			(with symmetry markers)			(flood)
2	2	1.32 s	526	2,733	0.51 s	178	953	0.50 s
2	3	32.10 s	7,130	52,533	4.57 s	862	6,521	4.85 s
3	3	210.31 s	42,345	334,450	12.40 s	2,046	16,938	16.38 s
3	4	>3600 s	>405,000		126.79 s	11,477	122,394	268.60 s

Fig. 11 Experiments with the video shop Z case study from [13,12]

MAXINT=4). Again, we can see the most a dramatic difference in performance for the last line (3 persons and 4 titles): PROB without symmetry reduction did not finish after one hour (and had generated over 405,000 states at that time), while with symmetry markers the model checking took 2 minutes and 7 seconds. Note that the symmetry markers were precise for all but the last entry (3 persons and 4 titles). Here, the flooding technique generates 11,692 nodes, whereas the symmetry marker technique generates only 11,477.

Unfortunately, [13,12] do not contain timings for exhaustive model checking. [13, 12] contains several experiments, were SAL is used to find counter examples for certain false theorems. Unfortunately, the hardware used is not described in [12]. It is probably slower than our reference hardware. Still, we try to provide some rough comparison between Z2SAL and PROB here. Note that the experiments in [13,12] were done with a maximum integer value of 3 and with 3 persons and 3 titles (the encoding for SAL uses 3 real titles and one bottom value for representing undefined function entries; the same scheme is used for integers). We have used the same setting for PROB below. For the property "th1" from [12] PROB takes 0.11 seconds with symmetry markers (and 0.07 seconds without), versus 4.40 seconds reported in [12] for Z2SAL. For "th4" PROB takes 0.06 seconds with symmetry markers (and 0.10 without), versus 4.40 seconds reported for Z2SAL in [12]. Finally, "th6" takes 2.00 seconds with PROB and symmetry markers (and 0.50 seconds without), versus 6.52 seconds in [12]. Note that the actual timings of PROB vary due to the random component of the mixed depth-first breadth-first search, which also explains why PROB with symmetry markers is in this case sometimes slower than without symmetry. We did restart PROB from scratch every time (even though one of the advantages of PROB is that it does remember the state space explored so far).

In summary, the results show that our symmetry reduction can be applied to Z models and that it can also drastically reduce the model checking time in that context.

7 More Related and Future Work

Symmetry reduction in model checking has been studied extensively since the nineties [21,27,9]. 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 [27] 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 [15]. The second precursor work in this line is the work of Clarke, Jha et al [9,29] 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 [7,19] and timed systems [23]. Data equivalence is also exploited by Jackson et al. [28] in relational specifications where data and operations are specified as relations. Note that the language NP used by Jackson et al. is relational and hence in the same family as Z, VDM and B.

In a pioneer work [21,20], 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 [47] 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 [21] 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 canonization 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 [38,16,18].

Symmetry on the formula allows another kind of reduction and has been investigated in [38].

The link between the finding of orbits and the graph isomorphism problems was studied in [9]. The computation of a representative element for a global state can therefore be done by the powerful algorithm [41] and the NAUTY tool [40] developed by McKay. The paper of Miller et al. [42] gives a nice survey to symmetry in model checking.

In future we plan to adapt our results to improve automatic refinement checking [34]. 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 could also combine our symmetry markers with the graph canonicalisation approach [50,48], 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 and Z, employing symmetry induced by deferred sets in B and given sets in Z. The technique computes symmetry markers for states and two states with the same symmetry marker are considered symmetric by the approximate verification algorithm. We have shown that for important classes of systems our method gives a complete verification algorithm. In our empirical evaluation we have also shown that our technique is both very precise (very few non-symmetric states are identified) and very efficient, sometimes achieving a fundamental reduction of the underlying exponential verification complexity.

Acknowledgements

First, we would like to thank the anonymous referees of SC'2007 and of AMAI for their very extensive and thorough feedback, which helped us considerably to improve paper. We also would like to thank John Derrick and Anthony Simons for answering our questions about Z2SAL and for performing some additional experiments.

References

1. Jean-Raymond Abrial. *The B-Book*. Cambridge University Press, 1996.
2. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
3. Jean-Raymond Abrial, Michael Butler, and Stefan Hallerstede. An open extensible tool environment for Event-B. In *ICFEM06*, LNCS 4260, pages 588–605. Springer, 2006.
4. B-Core (UK) Ltd, Oxon, UK. *B-Toolkit, On-line manual*, 1999. Available at `http://www.b-core.com/ONLINEDOC/Contents.html`.
5. Sharon Barner and Orna Grumberg. Combining symmetry reduction and under-approximation for symbolic model checking. *Formal Methods in System Design*, 27(1–2):29–66, 2005.
6. Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.
7. Dragan Bosnacki, Dennis Dams, and Leszek Holenderski. Symmetric spin. *STTT*, 4(1):92–106, 2002.
8. Dragan Bosnacki, Alastair F. Donaldson, Michael Leuschel, and Thierry Massart. Efficient approximate verification of promela models via symmetry markers. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Proceedings ATVA 2007*, LNCS 4762, pages 300–315. Springer, 2007.
9. 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.
10. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
11. ClearSy, Aix-en-Provence, France. *B4Free: Tool and Manuals*, 2006. Available at `http://www.b4free.com`.
12. John Derrick, Siobhán North, and Anthony Simons. Z2sal: a translation-based model checker for z. *Formal Aspects of Computing*. To appear. Available as Online First.
13. John Derrick, Siobhán North, and Anthony J. H. Simons. Z2SAL - building a model checker for Z. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *Proceedings ABZ 2008*, LNCS 5238, pages 280–293, 2008.
14. John Derrick, Siobhán North, and Tony Simons. Issues in implementing a model checker for Z. In Zhiming Liu and Jifeng He, editors, *ICFEM*, LNCS 4260, pages 678–696. Springer, 2006.
15. 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.
16. 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, *Proceedings FM 2005*, LNCS 3582, pages 481–496. Springer, 2005.
17. Alastair F. Donaldson and Alice Miller. Exact and approximate strategies for symmetry reduction in model checking. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings FM'2006*, LNCS 4085, pages 541–556. Springer, 2006.
18. 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.

19. 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.
20. E. Allen Emerson and A. Prasad Sistla. Utilizing symmetry when model checking under fairness assumptions: An automata-theoretic approach. In Pierre Wolper, editor, *Proceedings CAV'95*, LNCS 939, pages 309–324. Springer, 1995.
21. E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.
22. Sarah Flannery. *In Code: A Mathematical Adventure.* Profile Books Ltd, 2001.
23. 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, *Proceedings FORMATS 2003*, LNCS 2791, pages 46–59. Springer, 2003.
24. Gerard J. Holzmann. An improved protocol reachability analysis technique. *Softw., Pract. Exper.*, 18(2):137–161, 1988.
25. Gerard J. Holzmann. The model checker Spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
26. Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.
27. C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.
28. 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.
29. Somesh Jha. *Semmetry and Induction in Model Checking.* PhD thesis, School of Computer Science, Carnegie Mellon University, October 1996.
30. William Kocay and Donald L. Kreher. *Graphs, Algorithms and Optimization.* Chapman & Hall/CRC, 2004.
31. Donald L. Kreher and Douglas R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, Search.* CRC Press, 1999.
32. Michael Leuschel. The high road to formal validation. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *Proceedings ABZ 2008*, LNCS 5238, pages 4–23, 2008.
33. 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.
34. 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.
35. Michael Leuschel, Michael Butler, Corinna Spermann, and Edd Turner. Symmetry reduction for B by permutation flooding. In *Proceedings B2007*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.
36. Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
37. Michael Leuschel and Thierry Massart. Efficient approximate verification of B via symmetry markers. In *Proceedings International Symmetry Conference*, pages 71–85, Edinburgh, UK, January 2007.
38. Gurmeet Singh Manku, Ramin Hojati, and Robert K. Brayton. Structural symmetry and model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings CAV'98*, LNCS 1427, pages 159–171. Springer, 1998.
39. Paulo J. Matos, Bernd Fischer, and João P. Marques Silva. A lazy unbounded model checker for event-b. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 485–503. Springer, 2009.
40. Brendan McKay. Nauty user's guide. Available via `http://cs.anu.edu.au/people/bdm/nauty/`.
41. Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
42. Alice Miller, Alastair Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3):8, 2006.
43. Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
44. Daniel Plagge and Michael Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Proceedings IFM 2007*, LNCS 4591, pages 480–500. Springer-Verlag, 2007.

45. Daniel Plagge and Michael Leuschel. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. *STTT*, 11:9–21, 2010.
46. Steve Schneider. *The B-method, an introduction.* Computer Science - The Cornerstones of Computing Series. Palgrave, macmillan, 2001.
47. 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.
48. Corinna Spermann and Michael Leuschel. ProB gets nauty: Effective symmetry reduction for B and Z models. In *Proceedings TASE 2008*, pages 15–22, Nanjing, China, June 2008. IEEE.
49. France Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com`.
50. Edd Turner, Michael Leuschel, Corinna Spermann, and Michael J. Butler. Symmetry reduced model checking for B. In *Proceedings TASE 2007*, pages 25–34. IEEE Computer Society, 2007.

A Proofs

Proof of Proposition 1.

Proof If $paths(d, s) = \{| \, |\}$, then d does not occur in s, and hence $f(d)$ cannot occur in $f(s)$, as f is injective. Generally, $f(d)$ must occur in exactly the same places in $f(s)$ where d occurred in s. This can be formally proven by a straightforward induction on the length of the paths. More concretely, we have that $paths(f(e), f(d))$ is equal to:

1. if $f(e) = f(d)$:
   $paths(f(e), f(d))$
   $=$ (by Def. 6)
   $\{| \, \langle\rangle \, |\}$
   $=$ (because $e{=}d$)
   $paths(e, d)$
2. if $f(e) = (x \mapsto y)$ is a pair:
   $paths(f(e), f(d))$
   $=$ (by Def. 6)
   $\langle left\rangle.paths(x, d) \uplus \langle right\rangle.paths(y, d)$
   $=$ (by induction assumption)
   $\langle left\rangle.paths(f^{-1}(x), f(d)) \uplus \langle right\rangle.paths(f^{-1}(y), f(d))$
   $=$ (as $e = (f^{-1}(x) \mapsto f^{-1}(y))$ by Def. 3)
   $paths(e, d)$
3. if $f(e)$ is a set:
   $paths(f(e), f(d))$
   $=$ (by Def. 6)
   $\uplus_{x \in f(e)} \langle el\rangle.paths(x, f(d))$
   $=$ (by induction assumption)
   $\uplus_{x \in f(e)} \langle el\rangle.paths(f^{-1}(x), f(d))$
   $=$ (as $e = \cup_{x \in e} f^{-1}(x)$ by Def. 3)
   $paths(e, d)$
4. otherwise:
   $paths(f(e), f(d))$
   $=$ (by Def. 6)
   $\varnothing$
   Case $f(e)$ is a deferred set element with $f(e) \neq f(d)$
   $=$ (by Def. 6, as $e \neq d$)
   $paths(e, d)$
   Case $f(e)$ is a boolean, an integer or an enumerated set element
   $=$ (by Def. 6, as $f(e) = e$)
   $paths(e, d)$.

This proof uses the original version of Def. 6. For the extended version, the proof for the second case ($f(e) = (x \mapsto y)$ is a pair) proceeds in a similar fashion to above; using the fact that $e \in NonSym$ iff $f(e) \in NonSym$ and $x \neq y$ iff $f(x) \neq f(y)$.

Proof of Proposition 2.

Proof In the following we will make use of the following lemma:
Lemma 1: $m_s(v)$ is a data value with no deferred set elements iff $m_s(v) = v$.

Now, let $s_1 = \langle v_1, \ldots, v_n\rangle$ and $s_2 = \langle w_1, \ldots w_n\rangle$. We know that for $1 \leq i \leq n$ we have $m_{s_1}(v_i) = m_{s_2}(w_i)$.

We now define f to be a permutation over $\{D_1, \ldots, D_i\}$, with the property that for all $d \in D$ $\mathit{paths}(d, s_1) = \mathit{paths}(f(d), s_2)$. This function must exist because:

- All deferred set elements d are replaced by their $\mathit{paths}(d, s)$. Hence, s_1 and s_2 must have the same number of positions at which deferred set elements appear. Also, from $\mathit{paths}(d, s_1)$ we can uniquely determine how many occurrences of d there are in s_1.
- If for some $d \in D$ we have that for all $d' \in D$ $\mathit{paths}(d, s_1) \neq \mathit{paths}(f(d), s_2)$, then necessarily, $m(s_1) \neq m(s_2)$.
  - Indeed, either d occurs in s_1, then d is replaced by $\mathit{paths}(d, s_1)$ to obtain the marker $m(s_1)$, and hence $m(s_1) \neq m(s_2)$ as $\mathit{paths}(d, s_1)$ cannot occur in $m(s_2)$.
  - Or d does not occur in s_1, i.e., $\mathit{paths}(d, s_1) = \{||\}$. Here, we can again infer that $m(s_1) \neq m(s_2)$, as there now must be at least one other $e \in D$ occurring in s_1, such that no $e' \in D$ occurs the same number of times in s_2.

  In other words, as $m(s_1) = m(s_2)$, we have that for every $d \in D$ we can find a corresponding d' such that $\mathit{paths}(d, s_1) = \mathit{paths}(d', s_2)$.
- We can generalise the above reasoning, to conclude that for every $d \in D$ with $p = \mathit{paths}(d, s_1)$ we have $\mathit{card}(\{e \in D \mid \mathit{paths}(e, s_1) = p\}) = \mathit{card}(\{e' \in D \mid \mathit{paths}(e', s_2) = p\})$.

We now prove that $f(s_1) = s_2$, by proving that $f(v_i) = w_i$ for $1 \leq i \leq n$, inspecting the cases of Definition 2:

1. In this case $m_{s_1}(v_i) = v_i$ and hence for any permutation f, we have $f(v_i) = v_i$. By Lemma 1, we also know $m_{s_1}(w_i) = w_i$; hence $f(v_i) = w_i$.
2. In this case $m_{s_1}(v_i) = \mathit{paths}(v_i, s_1)$ and w_i must also be a deferred set element (by Lemma 1). We know that $\langle V\rangle \in \mathit{paths}(v_i, s_1)$ and $\langle V\rangle \in \mathit{paths}(w_i, s_2)$, where V is the name of the ith variable. We also know that v_i is the *only* element $x \in D$ such that $\langle V\rangle \in \mathit{paths}(x, s_1)$. Similarly, w_i is the *only* element $y \in D$ such that $\langle V\rangle \in \mathit{paths}(y, s_2)$. Hence, we must have by construction of f that $f(v_i) = w_i$.
3. Let $v_i = \{x_1, \ldots, x_n\}$. $\{x_1, \ldots, x_n\}$ are the only values x such that $\langle V.el\rangle \in \mathit{paths}(x, s_1)$. By Lemma 1 we know that w_i must also be of the form $\{y_1, \ldots, y_n\}$, which also are the only values y such that $\langle V.el\rangle \in \mathit{paths}(y, s_2)$. Hence, $f(x_i) \in w_i$. Furthermore, as f is a bijection, we must have $f(v_i) = w_i$.
4. set of pairs
5. A pair of values can simply be viewed as representing two separate variables. Hence, we can inductively apply the above reasoning on the components.