Symmetry Reduced Model Checking for B

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

[1] *School of Electronics and Computer Science*
University of Southampton
SO17 1BJ, UK
{*ent03r,mjb*}*@ecs.soton.ac.uk*

[2] *Institut für Informatik*
Universität Düsseldorf Universitätsstr. 1
D-40225 Düsseldorf, Germany
{*leuschel,spermann*}*@cs.uni-duesseldorf.de*

Abstract

Symmetry reduction is a technique that can help alleviate the problem of state space explosion in model checking. The idea is to verify only a subset of states from each class (orbit) of symmetric states. This paper presents a framework for symmetry reduced model checking of B machines, which verifies a unique representative from each orbit. Symmetries are induced by the deferred set; a key component of the B language. This contrasts with strategies that require the introduction of a special data type into a language, to indicate symmetry. An extended version of the graph isomorphism program, nauty, is used to detect symmetries, and the symmetry reduction package has been integrated into the PROB *model checker. Relevant algorithms are presented, and experimental results illustrate the effectiveness of the method, where exponential speedups are sometimes possible.*

1. Introduction

The B-Method [1] is a theory and methodology used for the formal development of computer programs. It includes a concise language based on set theory and predicate logic, called B, and is used by industries in a range of critical domains, notably in railway control (e.g., the control system for the automatic, unmanned Paris Métro Line 14).

Proof activities in B are usually carried out using the interactive theorem provers, Atelier-B [25] or the B-toolkit [2]. Model checking is a useful, complementary approach that can perform these tasks *automatically*, if bounds are placed on system types; as with the combined B-animation/model checker, PROB [19].

A major challenge facing model checking is the problem of *state space explosion*. This is where a linear increase in the size of a specification leads to an exponential increase in the number of states, which the model checker must explore. Thus, checking larger specifications becomes intractable. Much research in model checking focuses on methods to combat this problem, including partial order reduction [12], induction [6] and abstraction [5]. Symmetry reduction is another approach, which exploits symmetries inherent in the system [6] by constraining the search to representatives of symmetric states; often resulting in significant savings in memory and time. A successful technique relies on a special data type, called a *scalarset*, being introduced into the language of the model checker, to indicate symmetric structures (e.g., the Murφ Verifier [13] and SymmSpin [3]). This requires the user to indicate symmetries of the model, and is therefore error prone, and compromises the automation of model checking.

This paper presents an *automatic* method for exploiting symmetries caused by a key component of the B language, the *deferred set*. The work uses a very different technique to an alternate strategy called *permutation flooding*, which we present in [18]. In the next section, we introduce deferred sets in B, and the notion of symmetry. Section 3 follows with a detailed example that elaborates on the types of symmetries exploited. The integration of our symmetry reduction into the PROB model checker is given in Section 4. We then describe a two-phase technique that identifies system symmetries, using a graph isomorphism algorithm based on *nauty* [21]. Finally, we illustrate the effectiveness of the technique by applying it to several typical B models, and we discuss its drawbacks and future improvements.

2. Symmetry and Deferred Sets in B

In B there are two ways to introduce 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 whose elements are explicitly enumerated are called *enumerated sets*; the other types of sets are called *deferred sets*. An example of both kinds of given sets is given below:

```
SETS
    ExitMsg = {success,fail}; // an enumerated set
    Proc // a deferred set
```

Deferred sets consist of abstract elements. For instance, *Proc* is a set of abstract processors. The only information known about a given element of such a set is the *identity* of this deferred set. It follows that the permutation of one abstract element for another will have no effect on semantics; no information is gained or lost. Extending this idea, one finds a set containing *n* elements of *Proc* to be symmetric to another set containing *n* elements of *Proc*. Indeed, the use of *abstract* elements in a B specification gives rise to symmetries in data structures used within the system, which our scheme can exploit. This is similar to the symmetries induced by scalarsets [13, 3], which are sets of permutable scalar values. In the next section, we describe a concrete example to elaborate on the types of symmetries that our method can exploit.

3. Motivation

Let us indicate the type of symmetry we want to exploit by considering a simple B model of a phone book, see Fig. 1 below. The model (or machine) has three operations: *add*, to add entries into the phone book; *delete*, to remove entries; and *lookup*, to query a person's phone number.

The machine has two deferred sets, *Name* and *Code*, modelling sets of names and phone numbers respectively. A single variable, *db*, which is a partial function, stores the contents of the phone book; hence, a *Name* can have at most one *Code* associated with it.

An exhaustive search of the state space of this machine requires bounds to be placed on the types used [19]. If we set the cardinality of the deferred sets to 2, the full state space has 10 distinct states. Fig. 2 shows the state space, where the label of each state is the current value of *db* (e.g., {(Name1,Code1)}). For clarity, the parameters of the *add* operation are hidden, and the *delete* and *lookup* operations are not depicted (this does not affect the set of reachable states).

The use of deferred sets give rise to symmetries among the states. Informally, we define two states as being symmetric if the machine invariant has the same truth value in both states, and if there is a permutation between the two states that permutes values of deferred sets. We also require this permutation to respect the typing e.g., a *Name* can only be permuted with another *Name*. In Fig. 2, the state $db = \{(Name1, Code1)\}$ is symmetric to $db = \{(Name1, Code2)\}$ since both are functions and there is the

```
MACHINE phonebook
SETS Name; Code
VARIABLES db
INVARIANT $db \in Name \rightarrowtail Code$
INITIALISATION $db := \emptyset$
OPERATIONS
  add( n , c ) $\widehat{=}$
    PRE $n \in Name \wedge c \in Code \wedge n \notin \mathrm{dom}(db)$ THEN
    $db := db \cup \{n \mapsto c\}$ END;
  delete( n , c ) $\widehat{=}$
    PRE $n \in Name \wedge c \in Code \wedge n \mapsto c \in db$ THEN
    $db := db \setminus \{n \mapsto c\}$ END;
  $c \leftarrow$ lookup( n ) $\widehat{=}$
    PRE $n \in Name \wedge n \in \mathrm{dom}(db)$ THEN
    $c := db(n)$ END
END
```

Figure 1. Phonebook Machine

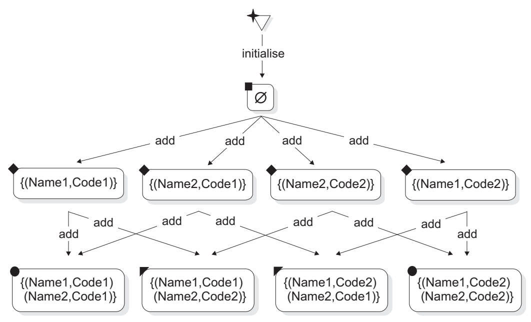
initialise
∅
add
add
add
add
{(Name1,Code1)}
{(Name2,Code1)}
{(Name2,Code2)}
{(Name1,Code2)}
add
add
add
add
add
add
add
add
{(Name1,Code1) (Name2,Code1)}
{(Name1,Code1) (Name2,Code2)}
{(Name1,Code2) (Name2,Code1)}
{(Name1,Code2) (Name2,Code2)}

Figure 2. State Space of Phonebook

permutation, $\{(Code1, Code2)\}$ between them. Symmetric states in Fig. 2 have been indicated by identical black shapes in their top left hand corner. As can be seen, three other states are symmetric to $db = \{(Name1, Code1)\}$.

Permutations that preserve the equivalence of two states must take care with values that are constants or elements of enumerated types (including Booleans and integers), since these values can break the symmetries described above. For example, an alternate phonebook machine might use a constant '999' $\in$ *Code* to represent the phone number of the emergency services. To ensure this number is reserved, we strengthen the precondition of the *add* operation with $n \neq$ '999'. Supposing $db = \{(Name1, Code1)\}$ and $db = \{(Name1, \text{'999'})\}$ are two reachable states for this new machine, we must find that they are *not* symmetric, despite the existence of a permutation on deferred sets between them. To handle this particular case, values of constants must not be permuted once model checking has started. A thorough definition of these permutation functions, and soundness results of the symmetries, can be found in [18].

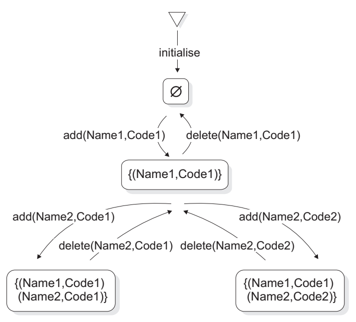

Figure 3. Reduced State Space of Phonebook

Our approach to symmetry reduction checks only one (unique) representative per symmetry class, using an algorithm for graph isomorphism that permits the permutations described. For proof of soundness with respect to standard model checking, see [18]. Fig. 3, illustrates the symmetry reduced state space for the phonebook machine. The technique exploits symmetries caused by deferred sets, so it is likely to significantly reduce the time to model check many B specifications, since such sets are commonly used in B. For instance, deferred sets often occur near the top of stepwise refinement chains, in the abstract specifications. In the next section, we present our approach for identifying symmetries.

4. Symmetry Detection

The task of identifying unique representatives from sets of symmetric states is closely related to the well known problem of solving graph isomorphism [16], which currently has no known polynomial algorithm. However, in practice some extremely efficient algorithms exist for most classes of graphs [15] that may contain several thousands of vertices [11]. The most efficient general purpose graph isomorphism program is *nauty* [22].

This section outlines our method for computing representatives of states in B using an extension to the underlying algorithm of nauty. The idea is to compute representatives in two phases, starting by translating a state to a graph, and then applying a graph isomorphism program to find its representative.

4.1. Symmetry in Model Checking

Let us recall that given a specification of a system, model checking involves the construction of a state transition system (*model*), representing the behaviour of the system, so that properties of it can be checked by exhaustive search.

Constructing models in ProB: A specification in B is described by means of constants and variables, whose evaluation determines a state, and a set of operations. To construct the model of a B machine, M, we represent machine constants and variables by a vector of variables $V = v_1, \dots, v_n$, and we characterise B operations operating on variables V with inputs x and outputs y by a predicate $P(x, V, V', y)$. Characterising B operations of the form $X \leftarrow op(Y)$ as predicates then gives rise to the model: a labelled transition system on states, where states s and s' are related by an operation $op.a.b$, denoted $s \rightarrow^M_{op.a.b} s'$, when $P(a, s, s', b)$ holds. A special *root* state is also added, where $root \rightarrow^M_{initialise}\ s$ denotes that state s satisfies the initialisation and properties clause. For further details, see [20]. In ProB, such models are automatically constructed and searched for states that violate properties specified on V (invariant violations), or are deadlocked.

We now formalise our modified model checking algorithm, to show how symmetry detection via graph isomorphism is integrated into the checking; see Algorithm 1. When a new state is encountered it is not explored further if its canonical form has already been explored. On line 4, $error$ is a function which returns true if the argument is an error state: usually, this means an invariant violation or a deadlock[1]. Also note the use of $random$ on line 11, and α, which is a user defined value. Its effect varies whether model checking progresses using a depth first or breadth first search.

The variable *Queue*, stores the states with transitions yet to be explored, and $Visited$ records states already reached by checking. $SGraph$ stores the section of the model explored so far. The function $\mathcal{G}$ (line 9) converts a state of a B machine into a labelled, directed graph, and is explained in Section 4.2 below. The function $canon$ computes a canonical form for such a graph. We explain how the function works in Section 4.4. Note that all elements of *Queue* and $Visited$ have associated hash values. It is therefore usually efficient to decide whether $sg \notin Visited$. We have implemented this algorithm within ProB, and we provide empirical results later in Section 5.

4.2. The Graph of a B State

Let us first consider an example of a graph that represents a state, which we will refer to later. Fig. 4 shows the state graph of the state, $db = \{(Name1, Code1), (Name2, Code2)\}$ in Fig. 2.

In this graph, the value of the relation, *db* is represented by edges that indicate specific ordered pairs, whose edge

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

Algorithm 1 Symmetry Reduced Model Checking in ProB

```
Require: An abstract machine $M$
 1: $Queue := \{root\}$ ; $Visited := \{root\}$; $SGraph := \{\}$;
 2: while $Queue \neq \emptyset$ do
 3:   state := $pop(Queue)$;
 4:   if $error$(state) then
 5:     return counter-example trace in $SGraph$ from $root$ to $state$
 6:   else
 7:     for all $succ$,$Op$ such that $state \rightarrow^{M}_{Op} succ$ do
 8:       $SGraph := SGraph \cup \{state \rightarrow_{Op} succ\}$
 9:       $sg := canon(\mathcal{G}(succ))$
10:       if $sg \notin Visited$ then
11:         if random(1) $< \alpha$ then
12:           add $succ$ to front of $Queue$
13:         else
14:           add $succ$ to end of $Queue$
15:         end if
16:         $Visited := Visited \cup \{sg\}$
17:       end if
18:     end for
19:   end if
20: end while
21: return ok
```

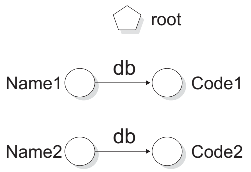

Figure 4. A phonebook state as a graph

labels denote the variable they encode. A special 'root' vertex is also present (different from the root in Algorithm 1), which is explained later in this section.

Values in B are either elements of sets (including Boolean values and integers), pairs of values, or sets of values. If we first ignore nested values in B, we can use three simple rules to translate a value to its concrete graphical representation. For an element of a set, $v \in S$, where $v = s_0$, we have the graph in Fig. 5. The graph of a set, $v \in \mathbb{P}(S)$, where $v = \{s_0, \ldots, s_n\}$ is shown in Fig. 6. Finally, a relation, $v \in S \leftrightarrow T$, where $v = \{(s_0, t_0), \ldots, (s_n, t_m)\}$ is depicted in Fig. 7. Also, although our graph representation does not distinguish $v = \{s_0\}$ from $v = s_o$, the B type system does and we only work with well-typed machines (typing is decidable in B).

We extend this idea for nested data structures, such as sets of sets, through the introduction of a set of special, symmetric vertices X, which contains an element $x \in X$

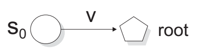

Figure 5. Graph for an atom

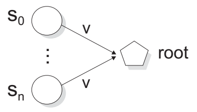

Figure 6. Graph for a set

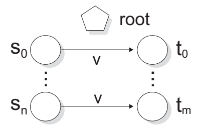

Figure 7. Graph for pairs

for each nested value, V. For nested sets, $v = \{V_0, \ldots, V_n\}$, we create $n + 1$ special vertices, and we translate the set $\{x_0, \ldots, x_n\}$ to a graph, as in Fig. 6. Then, we recurse on each nested value V_i, $0 \leq i \leq n$ and draw the corresponding graph with x_i as the new 'root'.

Similarly, for nested relations, $v = \{(V_0, V_1), \ldots, (V_{n-1}, V_n)\}$, we have $n + 1$ special vertices, and we translate the relation $\{(x_0, x_1), \ldots, (x_{n-1}, x_n)\}$ to a graph, as in Fig. 7. Then, we recurse on each nested value V_i, $0 \leq i \leq n$ and draw the corresponding graph with x_i as the new 'root'.

By composing the individual graphs that represent each value of a variable in a state, we obtain its *state graph*. Let $\mathcal{G}$ denote the function translating a state to its state graph. As an example, the graphical form of the state, $\langle v_1 = \{(\{s_0\}, \{s_1\})\}, v_2 = \{\{s_2\}\}\rangle$ is given in Fig. 8. Note that the colours used should be ignored; they will be explained in the next section.

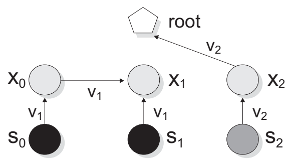

Figure 8. $\mathcal{G}(\langle v_1 = \{(\{s_0\}, \{s_1\})\}, v_2 = \{\{s_2\}\}\rangle)$

4.3. Relating Graph Isomorphism to State Equivalence

A convenient technique for establishing whether two graphs are isomorphic is to compute a *canonical label* for both. Canonical labelling functions find a unique label for some graph, which is the same for all symmetric graphs.

These algorithms rely on the permutation of graph vertices. Let us denote all vertex permutations with the relation γ. Now, consider a graph, G with vertices $V = \{v_1, v_2, \ldots, v_n\}$ and a sequence of vertices $v_1, v_2, \ldots, v_n$. All $n!$ possible vertex orderings are $\Gamma = \{o \mid o = v_1^\gamma, v_2^\gamma, \ldots, v_n^\gamma\}$, each of which corresponds to an adjacency matrix that encodes the graph; thus, Γ is an ordered set. Typically, an implementation of the algorithm will compute a subset of Γ and will choose the least element as the canonical label.

The assignment of labels to vertices is one method to encode extra information in a graph. For example, the vertices of two disjoint sets could be assigned one of two labels. By convention, vertex labels are called *colours*, and a vertex labelled graph is called a *coloured graph*. Canonical labels can be computed for coloured graphs by requiring vertex permutations take place over vertices with the same colour.

Definition 4.1 Permutation function *canon* computes the canonical label for a coloured graph by permuting vertices with the same colour, such that for two graphs g_1 and g_2, *canon*(g_1) = *canon*(g_2) $\Leftrightarrow$ g_1 is isomorphic to g_2.

We now describe how coloured graphs relate to state graphs. First, recall that in Section 3 we gave an informal definition of symmetries in B. We say that two states are symmetric if the invariant has the same truth value in both, and there exists a permutation between them over certain elements of deferred sets. In previous work [18], we prove this by defining a permutation function, f over symmetric elements of deferred sets used in a machine.

Definition 4.2 Let DS be the set of disjoint sets in a machine M. A *permutation f over DS* is a bijection from $\cup_{S \in DS} S$ to $\cup_{S \in DS} S$ such that $\forall S \in DS$ we have $\{f(s) \mid s \in S\} = S$. f is a fixpoint for enumerated types, including Boolean values and integers.

We can extend our definition of a state graph (Section 4.2) by colouring vertices. We assign the same colour to any pair of vertices *iff* f permutes their corresponding state values. For example, the vertex colours used in Fig. 8 would reflect that $s_0, s_1 \in DS_1$ and $s_2 \in DS_2$, given that $DS_1, DS_2 \subseteq DS$. Also, note the same colour is given to vertices x_0, x_1 and x_2, which belong to the set of special, symmetric vertices, X.

By lifting f to machine states we show in [18] that state σ satisfies predicate P (including any invariants), *iff* $f(\sigma)$ satisfies P. The goal now is to show that two states are symmetric *iff* $canon(\mathcal{G}(\sigma)) = canon(\mathcal{G}(f(\sigma)))$. This is accomplished since *canon* permutes only vertices with the same colour (Definition 4.1).

State graphs cannot be applied immediately to nauty, which works on undirected, unlabelled and coloured graphs. In the next section we extend this to directed and labelled graphs.

4.4. Computing Canonical Labels for Unlabelled, Undirected Graphs

The canonicalisation function, *canon*, has been implemented in SICStus Prolog, and integrated into PROB, using a technique called 'partition refinement', as used by the efficient graph isomorphism programs of nauty and saucy [8]. For a thorough introduction to partition refinement, see [15, 21]. Our implementation modifies the partition refinement step used by nauty. Before explaining our new algorithms for computing canonical labels, we first introduce some key concepts, and recall McKay's algorithm for unlabelled, undirected graphs [21].

A naïve approach to computing canonical labels would make $n!$ analyses for a graph with n vertices (Section 4.3, paragraph 1). Ideally, however, this should be as few as possible. A step towards this goal is to first sort the vertices by degree (the number of adjacent vertices in the graph), and consider only those orderings that preserve the degrees of the vertices. Let us extend this idea with the notion of *equitable partitions*; a relatively simple but effective technique that constrains further the number of orderings to consider.

Consider a simple graph G with the set of vertices V. An *ordered partition* π of V is a sequence $(V_1, V_2, \cdots, V_n)$ of disjoint non-empty subsets of V, called *cells*, whose union is V. A partition is called *discrete,* if for, $1 \leq i \leq n$, V_i is *trivial* (i.e., a singleton set). Let $v \in V$ and $W \subseteq V$, then the number of elements of W, which are adjacent to v in G, is denoted as $d(v, W)$. An ordered partition π is *equitable* if, for all $V_1, V_2 \in \pi$ and $v_1, v_2 \in V_1$ we have $d(v_1, V_2) = d(v_2, V_2)$. Since discrete partitions contain only trivial cells, we have the property that all discrete partitions are equitable; however, an equitable partition is not necessarily discrete.

An equitable partition π_β is computed by a procedure called *refine*, by analysing degrees of vertices in a graph, given some initial ordered partition ${\pi_\alpha}^2$. The computation ensures each cell in π_β is a subset of some cell in π_α. π_β is said to be *finer* than π_α; denoted $\pi_\beta \leq \pi_\alpha$. Algorithm 2 shows the use of *refine* for computing *canonical labels*.

The function *compute_label* on line 4 is used to encode a graph structure in a non-graphical form. Given a discrete partition π, take its adjacency matrix for graph G and define an integer $n(G)$ by writing down the entries in the upper-triangle, row by row[3]. Interpreting the result as an n^2-bit binary string, gives the return value of *compute_label*. Furthermore, given a set of discrete partitions, the lexicographic order on the corresponding set of binary strings induces the presence of a unique, least element, which we take

[2]In the context of partition refinement, *refine* bares no relation with refinement in B.

[3]We only analyse the upper-triangle as undirected graphs have triangular matrices.

Algorithm 2 $canon(\pi, G)$: Computing a canonical label

```
Require: Unlabelled, undirected graph, $G$, and ordered partition $\pi$
 1: $\pi_e$ = refine($\pi$); {refine $\pi$ to an equitable partition}
 2: if $\pi_e$ is discrete then
 3:     {compare with smallest label so far}
 4:     v = compute_label($G, \pi_e$);
 5:     if v $\leq$ best then
 6:         best = v; {update label}
 7:     end if
 8: else
 9:     {$\pi_e$ is not discrete – attempt to make non-equitable}
10:     $C$ = first non-discrete cell in $\pi_e$;
11:     for all $u \in C$ do
12:         make a copy $\pi_u$ of $\pi_e$ in which $C$ is split into $u$ and $C - u$
13:         canon($\pi_u$);
14:     end for
15: end if
```

as the canonical label.

The procedure *refine* (line 1) generates for an initial partition π_α, an equitable partition π_β, such that $\pi_\beta \leq \pi_\alpha$. Typically, π_β has significantly more cells than π_α [15], and consequently, π_β has significantly fewer discrete partitions (vertex orderings) finer than π_α. This premise forms the basis of Algorithm 2. Starting with an initial partition, create an equitable, finer partition. If this does not represent a vertex ordering, attempt to make a non-equitable, but finer partition (lines 10-12) and recurse. The overall procedure constructs a search tree, whose leaves are discrete partitions that must be considered when finding the canonical label. The size of the tree is, however, constrained through the use of *refine*, which progressively eliminates a large number of vertex orderings. An in-depth description of this procedure, together with algorithmic optimisations, can be found in [21]; for a somewhat gentler introduction, see [23].

In more detail, *refine compares* each cell with every other cell in π_α. Comparing two cells $V_1, V_2 \in \pi_\alpha$ means that for every element $v \in V_1$, we compute $d(v, V_2)$. Elements of V_1 with the same number of edges to V_2 form a new cell; thus, V_1 is split into several non-empty cells. These cells are ordered by increasing degree, such that cell V_i comes prior to cell V_j if $deg_i \leq deg_j$, where $\forall v_i \in V_i \Rightarrow d(v_i, V_2) = deg_i$, and $\forall v_j \in V_j \Rightarrow d(v_j, V_2) = deg_j$. New partition $\pi_{\beta 1}$ is formed by replacing the cell $V_1 \in \pi_\alpha$, with the newly created cells. This process repeats with $\pi_{\beta 1}$ until it remains unchanged or is discrete; the resulting partition, π_β is equitable. For correctness proofs of this procedure, see [21].

New Extended Algorithm: In order to represent the values of individual variables and constants, as well as to faithfully represent more complicated B data structures as graphs, we use directed, labelled, coloured (state) graphs. The move to directed graphs is straightforward (mainly consists of using full adjacency matrices in the *compute_label* procedure rather than only triangular ones).Similarly, treating coloured graphs is also not difficult; one simply defines an order on the colours and uses an initial partition where the vertices have already been partitioned according to the various colours (see also [21]). However, the move to graphs with labelled edges is less obvious. Below we describe how we have adapted McKay's procedure to handle such graphs. The main algorithm for computing the canonical label of a graph with directed and labelled edges is the same except for the *compute_label* and *refine* sub-procedures.

It should be noted that our implementation of *canon does not* use several intricate programming optimisations used in nauty (e.g., two variables for all partition nests [21]). Our implementation should be viewed as a proof of concept.

First we describe our change to the *compute_label* subprocedure. For labelled edges, an ordering is placed on the set of labels L (the variable names), so that labelled graphs can be encoded as a single matrix, where each entry is a binary string of size $\mid L \mid$. For directed edges, we ensure *compute_label* takes the row-by-row binary string for the full matrix. So for example, in Fig. 8, given a variable ordering v_1, v_2, the matrix entry at index $[x_0, x_1]$ would be '$1, 0$', since between x_0 and x_1 there exists only the single edge labelled v_1.

Algorithm 3 below shows our new *refine* procedure. Since we work on graphs with directed and labelled edges, we must first adapt the function $d(v, W)$ and the definition of equitable.

Definition 4.3 Let G be a graph with directed, labelled edges and set of vertices V, $v \in V, W \subseteq V$ and $L = (l_1, \cdots, l_l)$ the labels on the edges. Then $d_{in}(v, W, l_\nu)$ is the number of elements in W, that have an edge with the label $l_\nu \in L$ leading to v and $d_{out}(v, W, l_\nu)$ is the number of elements in W, that have an edge with the label l_ν coming from v.

Definition 4.4 Let G be a graph with directed, labelled edges and set of vertices V and $L := (l_1, \cdots, l_l)$ the labels on the edges. An ordered partition π of V is called *label equitable* if, for all $V_1, V_2 \in \pi$, $v_1, v_2 \in V_1$ and label $l_\nu \in L$ we have:

$$d_{in}(v_1, V_2, l_\nu) = d_{in}(v_2, V_2, l_\nu) \text{ and}$$
$$d_{out}(v_1, V_2, l_\nu) = d_{out}(v_2, V_2, l_\nu).$$

Example 4.5 We shall now integrate the methods described in Sections 4.2-4.4, to show how to compute the canonical label of an example B state, whose state graph G_x is given in Fig. 9.

Algorithm 3 *refine*(π, G): Extended partition refinement

```
Require: Directed, labelled graph $G$, $\pi = (V_1, \cdots, V_n)$,
    $L = (l_1, \cdots, l_l)$
 1: $\tilde{\pi} := \pi$;
 2: $\alpha = (V_1, \cdots, V_n)$;
 3: while $\tilde{\pi}$ is not discrete and $\alpha \neq \emptyset$ do
 4:     Remove an element $W$ from $\alpha$;
 5:     for all $\nu \in 1 \ldots l$ do
 6:         for all $k \in 1 \ldots n$ do
 7:             Compute ordered partition $(X_1, \cdots, X_s)$ from
                $V_k$, where $\forall i,j. 1 \leq i,j \leq s \wedge x \in X_i \wedge y \in X_j \Rightarrow i < j \Leftrightarrow d_{in}(x, W, l_\nu) < d_{in}(y, W, l_\nu)$
 8:             if $s > 1$ then
 9:                 update $\tilde{\pi}$ by replacing the cell $V_k$ with the
                    cells $X_1, \cdots, X_s$;
10:                 $\alpha$ = concatenate$(\alpha, (X_1, X_2, \cdots, X_s))$;
11:             end if
12:         end for
13:     end for
14:     Repeat lines 5 - 13, but use alternate condition
        $d_{out}(x, W, l_\nu) < d_{out}(y, W, l_\nu)$ on line 7.
15: end while
16: return Label equitable partition, $\tilde{\pi}$;
```

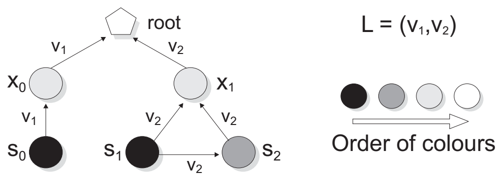

Figure 9. Example state graph

The example state makes use of two deferred sets, DS_1 and DS_2, and uses two variables, $v_1 \in \mathbb{P}(\mathbb{P}(D_1))$ and $v_2 \in \mathbb{P}(D_1 \leftrightarrow D_2)$. Let us assume $s_0, s_1 \in DS_1$, $s_2 \in DS_2$, and the variables values $v_1 = \{\{s_0\}\}$ and $v_2 = \{\{(s_1, s_2)\}\}$. Note, Fig. 9 also depicts the special vertices $x_0, x_1 \in X$ for nested values, and the orders of variables and colours. Thus, we start with the initial partition $\pi = (\{s_0, s_1\}, \{s_2\}, \{x_0, x_1\}, \{root\})$, such that vertices in the same cell have the same colour, and the cells are ordered by their colours.

Initially, line 1 of Algorithm 2 requests the partition refinement of π. We then enter Algorithm 3 with π, and $\alpha = (\{s_0, s_1\}, \{s_2\}, \{x_0, x_1\}, \{root\})$. In the first traversal of the **while** loop, $W = \{s_0, s_1\}$. The algorithm considers only edges with label v_1 in the first cycle of the outer **for**-loop. For the first cell of π, $V_1 = \{s_0, s_1\}$, no edges labelled v_1 originate from W and lead to an element of V_1; so $d_{in}((1), W, a) = d_{in}((3), W, a) = 0$ and π remains unchanged. The second cell of π, $V_2 = \{s_2\}$, is trivial and cannot be split further, so the algorithm continues. For the third cell, $V_3 = \{x_0, x_1\}$, we have $d_{in}((x_0), W, v_1) = 1 > 0 = d_{in}((x_1), W, v_1)$. So, V_3 is split into two cells, $\{x_1\}, \{x_0\}$; which must be ordered by their values for d_{in}. The algorithm updates π and α, where $\pi := (\{s_0, s_1\}, \{s_2\}, \{x_1\}, \{x_0\}, \{root\})$ and $\alpha := (\{s_2\}, \{x_1, x_0\}, \{root\}, \{x_1\}, \{x_0\})$. Since the last cell $V_4 = \{root\}$ is trivial, the algorithm progresses and begins considering edges labelled, v_2 (line 5, second iteration).

Splitting next occurs when d_{out} is analysed (execution of line 14), for $W = \{s_2\}$ and the edge label, v_2. When, $V_1 = \{s_0, s_1\}$, we have, $d_{out}((s_0), W, v_2) = 0 < 1 = d_{out}((s_1), W, v_2)$ and so partition π is updated to $\pi := (\{s_0\}, \{s_1\}, \{s_2\}, \{x_1\}, \{x_0\}, \{root\})$, which is now discrete. Further splitting is not possible, so Algorithm 3 terminates with this discrete, label equitable partition[4]. With execution returning to line 1 of Algorithm 2, π_e is discrete and the procedure terminates with the canonical label *compute_label*(G_x, π_e). Fig. 10 shows the first two rows of the adjacency matrix of G, which constitute the twelve most significant bits of the canonical label.

	s_0	s_1	s_2	x_1	x_0	*root*
s_0	00	00	00	00	10	00
s_1	00	00	01	01	00	00
⋮		⋮		⋮		⋮

Figure 10. Adjacency matrix corresponding to the canonical label of G_x

5. Empirical Results

In Table 1 we present the results from applying symmetry reduction in ProB to five typical B specifications. For each specification, we vary the size of deferred sets (Card) and record the time required for model checking to terminate. The table also shows the number of states and transitions in the reachable state space, with and without (wo) symmetry reduction. All experiments are performed on a PC with a 2.8GHz Intel Pentium 4 processor, 1Gb of available main memory, running SICStus Prolog 3.12.0 (x86-win32-nt-4) and ProB version 1.2.0. scheduler0 defines a process scheduling specification, and is given in [20]. scheduler is a variation of scheduler0, taken from [17]. The deferred sets in both cases are the process identifiers. Russian Postal Puzzle is a specification of a cryptographic puzzle [10], where the deferred sets are sets of available

[4]Note that a label equitable partition may not be discrete; see Definition 4.4.

keys and locks. phonebook is a slightly more elaborate version of the phonebook defined in Fig. 2, and has 3 variables. phonebook_err is the phonebook specification with an error in the precondition of the *delete* operation, which leads to an invariant violation.

Analysis of the results: Results obtained are encouraging. As can be seen, symmetry reduction reduces verification time up to some point, in each correct machine. Also, there is a large reduction in the number of states and transitions. The most prominent savings are for the phonebook machine, where a linear increase in size of deferred sets leads to a combinatorial saving in time; see the 'Speedup' column in Table 1. In the case where deferred sets have a size of 6, reduced checking is 231 times faster than standard checking. The results for the phonebook_err machine highlights PROB's effectiveness at finding counterexamples, even without reduction strategies. In fact, symmetry reduced checking requires more time to find the errors (even though fewer states are encountered) due to the overhead of computing representatives. However, the time required was still small (< 2s).

Predicting the magnitude of a speedup for some machine is non-trivial since it depends on the behaviour of the machine and the number of elements of deferred sets being used. In all cases, the speedups eventually drop off. The bottlenecks occur when the benefit of performing a constrained search outweighs the overhead of computing representatives. Fig. 11 shows how the speedup varies with the size of deferred set, for 3 machines. Data points correspond to the cardinality of deferred set used i.e., the *n*th point on a line shows the speedup when deferred sets contain *n* element(s). The drop off points illustrated are affected by programming inefficiencies. For instance, the Gauge profiling module in SICStus Prolog shows that more than 70% of computation time is spent accessing Prolog terms that model arrays; this figure would be *significantly* reduced if C-language arrays were used, which have constant-time access. Additionally, several major optimisations used in nauty are not used by our algorithm. The result of any algorithmic optimisation would be to increase the speedups achieved (i.e., increase the gradient of lines in Fig. 11) and delay the speedup drop off points.

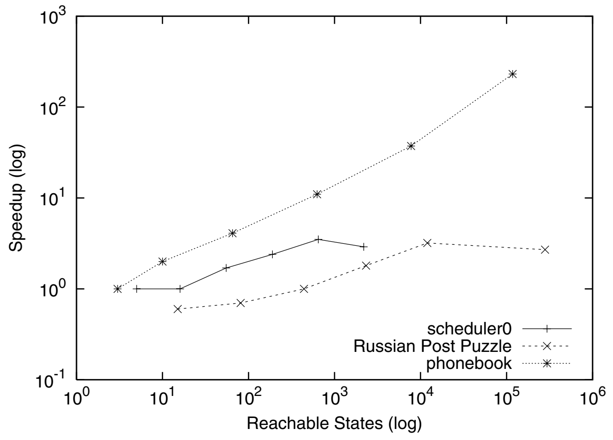

Figure 11. Variation of speedups with cardinality of deferred sets

6. Related Work

Symmetry reduction in model checking dates back to [4, 3], and more recently [9]. A key difference with these works is that ours does not consider temporal logic formulas, but B's criteria of invariant violations, deadlocks and refinement. Despite this, the fundamental problem of efficiently computing representatives, namely the orbit problem, is common to all approaches.

The line of research that has developed symmetry reduction techniques based on *scalarsets* [13] is the inspiration for the present article. The SymmSpin package [3] uses scalarsets to integrate symmetry reduction into the SPIN model checker. These special types contain only symmetric data values and are thus similar to deferred sets in B. However, the scalarsets approach is not automatic; it requires the user to identify and assert the presence of symmetries. Our approach is fully automatic since the deferred set is a key component of the B language. Furthermore, our reduction strategy will always apply to a large proportion of B machines since deferred sets are frequently used.

In Section 4.2, we show how to encode a B state as a graph. Various other research also addresses this general problem, such as [24], which describes an approach for states of Java programs. For each Java object one encodes its associated type (or program counter, if it is a thread), and a finite set of values. The state graph, or *shape graph*, is then a directed, labelled graph, which intuitively represents the state's contents, and which also makes use of a root node. This differs from B state graphs as shape graphs are entirely deterministic, except for the use of special ϵ transitions (i.e., indistinguishable labels that can indicate nodes corresponding to active threads) – both features are not present in B state graphs. In addition, B state graphs encode non-comparable abstract elements of deferred sets, for which there is no equivalent notion in Java. However, it remains that both methods share the use of state/shape graphs to achieve the goal of exploiting symmetries in their respective systems.

An interesting orthogonal concept for symmetry reductions is to use *symmetry breaking predicates* [7], as used by the Alloy Analyser [14], whose reductions are often several orders of magnitude. Its premise prevents redundant sym-

Table 1. Experimental results for five typical B-specifications

Machine	Card	Time		Speedup	States		Trans	
		wo (s)	with		wo	with	wo	with
scheduler0 (from [20])	1	0.01	0.01	1.0	5	5	6	6
	2	0.05	0.05	1.0	16	10	37	23
	3	0.26	0.15	1.7	55	17	190	59
	4	1.24	0.52	2.4	190	26	865	121
	5	7.78	2.20	3.5	649	37	3646	216
	6	35.35	12.03	2.9	2188	50	14581	351
scheduler (from [17])	1	0.01	0.01	1.0	4	4	5	5
	2	0.03	0.02	1.5	11	7	25	16
	3	0.14	0.08	1.8	36	11	121	37
	4	0.64	0.30	2.1	125	16	561	71
	5	3.02	2.78	1.1	438	22	2418	121
	6	22.93	27.03	0.8	1523	29	10489	190
Russian Postal Puzzle	1	0.05	0.08	0.6	15	15	24	24
	2	0.31	0.42	0.7	81	48	177	105
	3	2.13	2.05	1.0	441	119	1277	331
	4	17.34	9.80	1.8	2325	248	7869	838
	5	158.61	48.83	3.2	11985	459	47795	1826
	6	738.14	266.37	2.8	60981	780	279969	3571
phonebook	1	0.01	0.01	1.0	3	3	4	4
	2	0.06	0.03	2.0	10	5	37	17
	3	0.69	0.17	4.1	65	8	433	50
	4	12.03	1.09	11.0	626	13	6001	125
	5	280.38	7.51	37.3	7777	20	97201	269
	6	13944.97	60.36	231.0	117650	31	1815157	541
phonebook_err	4	0.01	0.90	<0.1	46	17	57	108
	5	0.01	0.20	<0.1	76	10	88	88
	6	0.01	1.92	<0.1	131	19	156	178

metries being computed through the addition of predicates that constrain the search. Although originally aimed at SAT based search methods, the key ideas appear applicable to a constraint logic programming environment such as SICStus Prolog, as used by PROB.

7. Conclusions and Future Work

In this paper, we have presented the first technique to achieve classical symmetry reduction for model checkers of B specifications. The algorithm computes representatives in two phases. First we compute the state graph of state *s*, and then we compute its canonical label, corresponding to the unique representative of *s*. We have described the relation of symmetries in B to canonical labelling algorithms, and have presented a translation of B states to graphs. We have also presented our extension to McKay's algorithm to find canonical labels. We have implemented the algorithm inside the PROB toolset and have evaluated the approach on a series of examples. The empirical results were very encouraging, where exponential speedups are sometimes possible. Furthermore, the techniques developed may be generalised for application to model checkers of other formal languages.

In future work, we will increase the speedups our method can achieve by optimising our implementation of the canonical labelling algorithm, so that its performance is closer to that of nauty. Immediate changes include translating Prolog procedures into C/Java, and using two variables per partition nest [21], during search tree exploration, to significantly improve performance by reducing memory usage and computation time. Furthermore, we can optimise our implementation of partition refinement to better suit the properties of state graphs, and break more symmetries in each step (similar to how *saucy* [8] optimises nauty for CNF). One example is to use the fact that edges never originate from the set of vertices, X, introduced to represent nested values. Future work also includes constructing symmetry breaking predicates from the automorphisms found when computing canonical labels, and using them to constrain the search space *before* symmetric states are encountered.

References

[1] J. R. Abrial. *The B Book: Assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] B-Core (UK) Limited, Oxon, UK. *B-Toolkit, On-line manual*, 1999.

[3] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric SPIN. *STTT International Journal on Software Tools for Technology Transfer*, 4(1):92–106, 2002.

[4] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In C. Courcoubetis, editor, *CAV*, volume 697 of *Lecture Notes in Computer Science*, pages 450–462, London, UK, 1993. Springer-Verlag.

[5] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. The MIT Press, 1999.

[7] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR*, pages 148–159, 1996.

[8] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for CNF. In S. Malik, L. Fix, and A. B. Kahng, editors, *DAC*, pages 530–534. ACM, 2004.

[9] A. F. Donaldson and A. Miller. Exact and approximate strategies for symmetry reduction in model checking. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 541–556. Springer, 2006.

[10] S. Flannery and D. Flannery. *In Code: A Mathematical Journey*. Algonquin Books of Chapel Hill, Chapel Hill, NC, USA, 2001.

[11] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

[12] G. J. Holzmann and D. Peled. An improvement in formal verification. In D. Hogrefe and S. Leue, editors, *FORTE*, volume 6 of *IFIP Conference Proceedings*, pages 197–211. Chapman & Hall, 1994.

[13] C.-W. N. Ip and D. L. Dill. Better verification through symmetry. In D. Agnew, L. J. M. Claesen, and R. Camposano, editors, *CHDL*, volume A-32 of *IFIP Transactions*, pages 97–111. North-Holland, 1993.

[14] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[15] W. Kocay. *On Writing Isomorphism Programs*, chapter Chapter 6, pages 135–175. Computational and Constructive Design Theory. Kluwer, 1996.

[16] D. L. Kreher. *Combinatorial Algorithms: Generation, Enumeration and Search*. Discrete Mathematics and its Applications. CRC Press, 1998.

[17] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In L.-H. Eriksson and P. A. Lindsay, editors, *FME*, volume 2391 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002.

[18] M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry reduction for B by permutation flooding. In J. Julliand and O. Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 79–93. Springer-Verlag, 2007.

[19] M. Leuschel and M. J. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer, 2003.

[20] M. Leuschel and M. J. Butler. Automatic refinement checking for B. In K.-K. Lau and R. Banach, editors, *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2005.

[21] B. D. McKay. Practical graph isomorphism. In *Numerical mathematics and computing, Proc. 10th Manitoba Conf., Congr. Numerantium 30*, pages 45–87, 1981.

[22] B. D. McKay. *NAUTY user's guide (version 1.5), Technical report TR-CS-90-02*. Australian National University, Computer Science Department, ANU, 1990.

[23] T. Miyazaki. The complexity of Mckay's canonical labeling algorithm. In L. Finkelstein and W. M. Kantor, editors, *Groups and Computation II*, volume 28 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 239–256. American Mathematical Society, Providence, Rhode Island 02940, USA, 1997.

[24] Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic software. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.

[25] Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 1996.