Efficient Approximate Verification of Promela Models Via Symmetry Markers

Dragan Bošnački[1], Alastair F. Donaldson[2], Michael Leuschel[3], and Thierry Massart[4]

[1] Department of Biomedical Engineering, Eindhoven University of Technology
dragan@win.tue.nl
[2] Codeplay Software Ltd., Edinburgh
ally@codeplay.com
[3] Institut für Informatik, Universität Düsseldorf
leuschel@cs.uni-duesseldorf.de
[4] Département d'informatique, Université Libre de Bruxelles
tmassart@ulb.ac.be

Abstract. We present a new verification technique for Promela which exploits state-space symmetries induced by *scalarset* values used in a model. The technique involves efficiently computing a *marker* for each state encountered during search. We propose a complete verification method which only partially exploits symmetry, and an approximate verification method which fully exploits symmetry. We describe how symmetry markers can be efficiently computed and integrated into the SPIN tool, and provide an empirical evaluation of our technique using the TopSPIN symmetry reduction package, which shows very good performance results and a high degree of precision for the approximate method (i.e. very few non-symmetric states receive the same marker). We also identify a class of models for which the approximate technique is precise.

1 Introduction

The design of concurrent systems is a non-trivial task where generally a lot of time is spent on simulation to track design errors. *Model checking* methods and tools [Hol03, McM93, CGP99] can be used to help in this effort by automatically analysing finite-state models of a system. In practice, exhaustive exploration of a state-space is impractical due to the infamous *state explosion problem*, which has motivated the development of more efficient exploration techniques. In particular, the model to be checked often consists of a large number of states which are indistinguishable up to rearrangement of process identifiers. As a result, the model is partitioned into classes of states where each member of a given class behaves like every other member of the class (with respect to a logical property that does not distinguish between individual processes). *Symmetry reduction* techniques [CGP99, ID96, CEFJ96, ES96] allow the restriction of model checking algorithms to a *quotient* state-space consisting of one representative state from each symmetric equivalence class. One successful symmetry reduction technique

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 300–315, 2007.

[ID96] relies on a special data type, called *scalarset*, used in the specification to identify the presence of symmetry. Symmetry reduction using scalarsets has been initially implemented in the Murϕ verifier [ID96], and in previous work we have adapted the same idea for Promela in the SymmSpin tool [BDH02]. We have also proposed in [DM05] a method to automatically detect, before search, structural symmetries in Promela specifications, and such symmetries can be exploited by the TopSpin tool [DM06]. Both SymmSpin and TopSpin perform symmetry reduction on-the-fly: each time a global state s is reached, its representative state $rep(s)$ is computed and used instead of s. If $rep(t) = rep(s)$ for every state t in the same class as s then symmetry reduction is said to be *full* (i.e. it is memory-optimal). However, the computation of a unique representative for an equivalence class (known as the *constructive orbit problem*) is NP-hard [CEJS98]; and for some practical examples full symmetry reduction strategies can be too time consuming. Therefore, *partial symmetry* reduction strategies have been defined, which take less time to compute a representative element. The price to pay is that multiple representatives may be computed for a given equivalence class and therefore the reduction factor in the number of global states explored can be smaller than with a full reduction method.

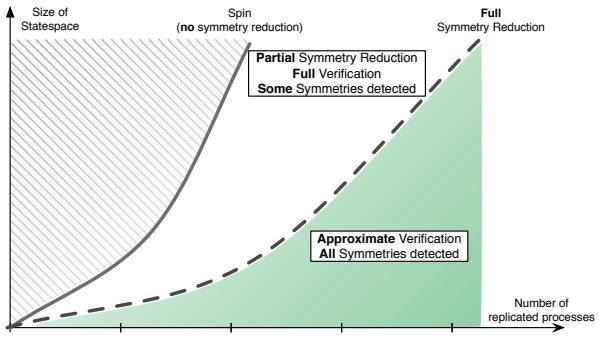

Fig. 1. Illustrating partial symmetry reduction and approximate verification

Fig. 1 illustrates the possible size of the explored state-space for the various verification methods, as a function of the number of replicated processes in the system. The solid line represents exhaustive state-space exploration without symmetry reduction, the dashed line a complete verification of the quotient state-space modulo full symmetry reduction. Any complete verification method using partial symmetry reduction would yield a function in the area between these lines. Recently, we have proposed the *symmetry marker* method [LM07] in the framework of the B specification language [Abr96]. This reduction technique is inspired by the success of Spin's bitstate hashing technique [Hol88], which regards two states as equivalent if they hash to the same value. Bitstate hashing reduces the per-state storage requirement to a single bit, at the expense of excluding a small percentage of states due to hash collisions. In a similar vein, we define a *marker* function on global states, invariant under symmetry, which can be computed efficiently. We avoid the underlying complexity induced by the constructive orbit problem by assuming that two states with the same marker value are symmetric. As this assumption

can be wrong in general, we only have an *approximate* verification technique (in the sense that collisions may occur where non-symmetric states obtain the same marker value); but a very fast one. If we refer to Fig. 1, our approximate verification method would give a function on the dashed line when it provides complete verification, or below this line in case of collisions when it only provides approximate verification.

In this paper, inspired by our previous works on symmetry, we propose two new symmetry reduction methods for Promela: a *complete* verification method which may compute more than one representative for each symmetric equivalence class, and an *approximate* verification method which guarantees unique representatives, but results in a small number of collisions between equivalence classes. We detail a TopSpin-based implementation of these methods, and provide encouraging experimental results. Note that although we present our methods in the context of Promela/Spin, they are clearly transferrable to other explicit-state model checking frameworks.

In the remainder of the paper, we briefly recall in Sect. 2 some relevant features of Spin and Promela, the notion of a *scalarset* and the main methods employed by the SymmSpin and TopSpin tools. We outline in Sect. 3 the initial *symmetry marker* method presented in [LM07], then describe in Sect. 4 a first naïve method for Promela and Spin, which is directly inspired by our symmetry marker method. In Sect. 5 we describe our "complete" new methods based on markers for Promela and Spin. In Sect. 5 we also provide theoretical results for particular classes of systems where our method is precise, and in Sect. 6 some results which empirically validate both methods compared to methods without symmetry reduction and existing methods implemented in SymmSpin and TopSpin.

2 Scalarsets in Promela

The Spin model checker [Hol03] allows verification of concurrent systems specified in Promela — a C like language extended with Dijkstra's guarded commands and communication primitives from Hoare's CSP. In Promela, system components are specified as *processes* that can interact either by message passing, via buffered or rendez-vous *channels*, or memory sharing, via *global variables*. Concurrency is asynchronous and modelled by interleaving. Spin can verify various safety and liveness properties of a Promela model including any LTL formula. To cope with the problem of state-space explosion, standard Spin employs several reduction techniques, such as partial-order reduction, state-vector compression, and bitstate hashing. In Spin each state has an explicit representation called the *state vector*. The state vector has the form $(G, R_1, \ldots, R_n)$, where G comprises the values of global variables, and $R_1, \ldots, R_n$ are records corresponding to the processes in the system. Each process record contains the parameters, local variables and the program counter for the particular process. The marker algorithms that we present in the sequel are independent of this representation, but in the presentation we will refer to the process vector structure and in particular its form for symmetric models.

SymmSpin [BDH02] is an extension of Spin with symmetry reduction based on the *scalarset* data type [ID96], by which the user can point out (full) symmetries to the verification tool. The values of scalarsets are finite in number, unordered, and allow only a restricted number of operations which do not break symmetry. Intuitively, in the context of Spin, the scalarsets are process identifiers (pids) of a family of symmetric processes. Such a family is obtained by instantiating a parameterized process type. One restriction is that applying arithmetic operations to pids is forbidden. Also, since formally there is no ordering between the scalarset values, the pids can be tested only for equality. We consider models in Promela that are collections of parallel processes of the form $B \| P_1 \| \ldots \| P_n$. Processes P_i are instances of a parameterized process template and differ only in their pid. Process B is a *base* process and it represents the "non-symmetric" part of the model (though B must behave symmetrically with respect to the P_i). Further, we assume that each state is represented explicitly by a state vector as described above. To illustrate the main ideas behind the strategies for finding representatives in SymmSpin, consider the following example adapted from [BDH02]. Let us assume that we want to choose as a (unique) representative of each symmetry class (orbit) the state from that class represented by the lexicographically minimal state vector. Further, suppose that there is an array M (we call it the *main* array) at the very beginning (of the global part G) of the state vector of our model. Let M be indexed by pids (scalarsets, here 1 to 5), but the elements of M are not of scalarset type. The *sorted* strategy computes a representative of a state s by sorting M and applying the pid permutation p corresponding to this sorting operation to the rest of the state vector. This involves sorting the process records, and rearranging the values of scalarset variables. Fig. 2 shows the state vector before and after sorting an example main array M with permutation $p = \{1 \mapsto 1, 2 \mapsto 5, 3 \mapsto 3, 4 \mapsto 2, 5 \mapsto 4\}$.

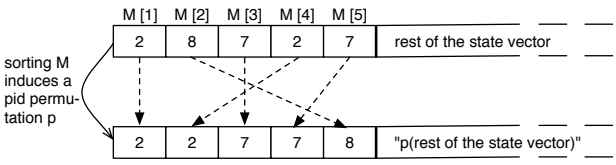

Fig. 2. A state vector before and after sorting the main array M

Note that when M contains several instances of the same value (here 2 and 7), several orbit representatives can be computed for the same class of states. Hence, the sorted strategy only performs partial symmetry reduction. Suppose we applied *all* pid permutations to the upper state vector in Fig. 2. Then the lexicographical minimum among the resulting states, $s_{\min}$ say, would start with the same values as the lower state vector, namely 2, 2, 7, 7, 8. However, the rest of $s_{\min}$ need not coincide with the rest of the lower state vector. The reason is that there are other pid permutations that yield the same ordering of the array M, for example $p' = \{1 \mapsto 2, 2 \mapsto 5, 3 \mapsto 4, 4 \mapsto 1, 5 \mapsto 3\}$, but may give smaller results than p when applied to the rest of the state vector. The *segmented* strategy applies all pid permutations which sort M (in this example there are four

of them) to the whole state vector, and selects the smallest among the resulting states, which is then guaranteed to be $s_{\min}$. The price to pay for this *full* reduction strategy is factorial complexity in the worst case: if all values of M are identical in a state then all $n!$ scalarset permutations must be considered in order to compute a representative for the state. However, for many states very few permutations need to be considered, so this approach is more efficient than the basic approach of considering every pid permutation at each state. Note that a main array is always available and can be selected automatically by the model checker – if no suitable array is explicitly declared in the model then the array of process program counters can be used, by default.

The TopSpin symmetry reduction package [DM06] builds on the ideas introduced with SymmSpin, supporting more general types of symmetry than just full symmetry, and providing support for automatic symmetry detection based on techniques presented in [DM05]. We use TopSpin for implementation of our techniques, so it is important to briefly explain the relationship between TopSpin and SymmSpin. TopSpin includes symmetry reduction strategies based on the SymmSpin *sorted* and *segmented* strategies. The TopSpin *sorted* strategy is a generalisation of the SymmSpin *sorted* strategy: instead of sorting with respect to one particular *main* array, sorting is performed in a more general fashion by repeatedly applying swap permutations to the entire target state. This generalised *sorted* strategy sometimes performs better than the original SymmSpin *sorted* strategy, but both approaches share the problem that symmetry reduction may result in storage of multiple orbit representatives. The TopSpin *segmented* strategy generalises the SymmSpin *segmented* approach, and is described in detail in [DM07]. For the special case of full symmetry, the TopSpin and SymmSpin *segmented* strategies are analogous.

3 Symmetry Markers

Our symmetry marker technique, initially proposed in [LM07] for the B language, is partially inspired by Holzmann's successful bitstate hashing technique [Hol88]. 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. Our adapted model checking algorithm stores those markers rather than the states and checks a new state only if its marker has not yet been seen before. The advantages over classical symmetry reduction are two-fold. First, a precise symmetry marker can be computed very efficiently (depending on the system, basically linear or quadratic in the size of the state for which the marker is computed), while classical symmetry reduction has an inherent factorial complexity (in terms of the number of the symmetric data values). The second advantage is the size of the state-space explored with our marker method, which is equal or less than the size of the state space explored with a full symmetry reduction method (smaller if collisions occur). The price we pay is that – just as with the bitstate hashing technique – we no longer have a complete verification method: two non-symmetric states s_1, s_2 can have

the same marker meaning that the second state s_2 would not be checked, even though it could lead to an error while no error is reachable from s_1.

In the B language [Abr96], the *deferred sets* construct gives rise to *symmetric* data values similar to scalarsets [LM07]. The value of a global state s can be given as a vector of values of its global variables and constants, which are classified by the following types: simple non-symmetric (e.g. booleans, and integer subranges), simple scalarset (i.e. a deferred set), pair, or finite (multi)sets. In this setting, a set of pairs defines a relation and an *array* is defined as a total function (i.e. a particular type of relation) between its indexes and the value of its components. We adopt a standard structured view of a state as a *rooted acyclic graph* whose nodes are labelled by their type and whose leaves are values. The root has n ordered children corresponding to n variables or constants. Simple values are *leaves* of the graph, pair values have two ordered children and (multi)sets have unordered children, one for each element in the (multi)set. The idea of our marking function is to transform a state s into a marker by replacing the scalarset values by so-called *vertex invariants*. In graph theory, a vertex invariant *inv* is a function which labels the vertices of an arbitrary graph with values so that symmetrical vertices are assigned the same label. Examples of simple vertex invariants include the in-degree and the out-degree for the specified vertex. Our technique uses a more involved vertex invariant for scalarset elements. Informally, a symmetry marker $m(s)$ for a given state s is computed as follows: (1) For every scalarset element d used in s, compute structural information about its occurrence in s, invariant under symmetry. This is computed as the multiset of *paths* that lead to an occurrence of d in s. (2) Replace all scalarset elements by the structural information computed above and compute a marker with an algorithm similar to the computation of the canonical form in the tree isomorphism problem [Val02]. The resulting complexity is quadratic in the size of the state in the worst case. We have proved in [LM07] that our definition is indeed invariant under symmetry, i.e. that if s_1 and s_2 are symmetric states in a system M then $m(s_1) = m(s_2)$; we have also identified classes of systems where the marker method is precise, i.e. it provides a full symmetry reduction method. Note that the method is quite general and abstract and could be instantiated in other contexts than those of SPIN/Promela and B.

4 A First Naïve Approach

As a stepping stone towards our approximate techniques, we first describe a naïve strategy that we call *flattened*. In this approach we "flatten" the state vector by assigning to each scalarset variable the same value. (The choice of the concrete value is irrelevant – it can be from the range of the scalarset or some other "neutral" value.) Basically, this amounts to distinguishing processes by the values of their non-scalarset local variables only. Then we apply to the flattened state vector TopSPIN's *sorted* strategy, described in Sect. 2. Obviously, because of the flattening, states that are not symmetric may have the same representative. As a result a full state-space coverage is not guaranteed. We illustrate this basic technique using Peterson's n-process mutual exclusion protocol [Pet81],

```
byte flag[6] = 0;    // an array from PID to byte (flag[0] is not used)
pid turn[5] = 0;     // an array from [0..4] to PID
byte inCR = 0        // number of processes in critical region
proctype user () {
   byte k; bool ok;
   do :: k = 1;
         do :: k < 5 ->  flag[_pid] = k; turn[k] = _pid;
again:         atomic {
                  ok = ((_pid==1)||(_pid!=1 && flag[1]<k))&&
                       ((_pid==2)||(_pid!=2 && flag[2]<k))&&
                       ((_pid==3)||(_pid!=3 && flag[3]<k))&&
                       ((_pid==4)||(_pid!=4 && flag[4]<k))&&
                       ((_pid==5)||(_pid!=5 && flag[5]<k));
                  if :: ok || turn[k] != _pid
                     :: else -> goto again
                  fi
               };
               k++
            :: else -> break
         od;
         atomic { inCR++; assert(inCR == 1) };  inCR--;  flag[_pid] = 0;
   od;
}
 init {   // start the processes
   atomic{ run user(); run user(); run user(); run user(); run user(); }
 }
```

Fig. 3. Promela code for Peterson's mutual exclusion protocol, with five processes

which has been used for experiments with SymmSpin [BDH02]. We consider various configurations of a Promela specification of this protocol, adapted from the specification used in [BDH02]. Promela code for the specification with five processes is given in Fig. 3. We check the mutual exclusion property, which is embedded into the specification using an assertion, and also verify that the model associated with each specification is deadlock-free. For various configurations of the protocol, Fig. 4 shows the state-space size and time (in seconds) for unreduced verification, and verification using the SymmSpin *segmented* and *sorted* strategies, the TopSpin *sorted* strategy, and our *flattened* strategy. An entry marked '−' indicates that memory requirements for verification exceeded available resources, or that verification did not complete within five hours. All experiments were performed on a PC with a 1.7 GHz Pentium processor, 760 Mb of main memory, using Spin version 4.2.6. Recall that the SymmSpin *segmented* strategy is guaranteed to give memory-optimal symmetry reduction. For this

Peterson	**Spin (unreduced)**		**SymmSpin segmented**		**SymmSpin sorted**		**TopSpin sorted**		**TopSpin flattened**	
n	states	time	states	time	states	time	states	time	states	time
3	2636	0.4	494	0.3	907	0.4	494	0.4	251	0.1
4	60577	0.6	3106	0.4	9373	0.4	3106	0.4	1177	0.1
5	1.56×10^6	11	17321	1	95303	2	17321	1	5148	0.3
6	4.48×10^7	2666	89850	7	885399	18	89850	7	21752	2
7	-	-	442481	85	7.94×10^6	383	442481	56	89969	10
8	-	-	2.09×10^6	1166	-	-	2.09×10^6	412	366424	63
9	-	-	9.62×10^6	16673	-	-	9.62×10^6	3034	1.47×10^6	393

Fig. 4. Experimental results for Peterson's mutual exclusion protocol, using SymmSpin, TopSpin, and a naïve "markers"-based approach

example, therefore, we see that TopSpin *sorted* also provides memory-optimal symmetry reduction. In comparison, SymmSpin *sorted* performs visibly poorly on the *Peterson 7* configuration, and could not be practically applied to larger configurations. The *flattened* strategy yields very fast verification in comparison to all other strategies. Correspondingly, the state-space explored using this strategy is much smaller than the symmetry-reduced state-space explored using the SymmSpin *segmented* strategy. The speed-up gained using this approach is encouraging, but state-space coverage is clearly too low for this technique to be acceptable in practice. Motivated by the efficiency of the *flattened* strategy, we now develop more sophisticated symmetry marker techniques for Promela.

5 The New Marker Methods for Promela

The marker method developed for B [LM07] inspires the definition of efficient symmetry reduction techniques for other specification languages, such as Promela. However, adapting the techniques for Promela is not trivial. The requirement to extend Spin (more precisely SymmSpin or TopSpin) to include the new concepts means that we cannot simply use data structures like multisets of paths as defined in [LM07], but need to define an efficient encoding in the context of the data structures used for state-space representation by Spin. The methods we propose respect this constraint through transformations of state vectors which preserve its structure. We also derive a new technique, which can be used as a complete verification method.

Datatypes and state representation. We first define the following simple Promela datatypes:

- a single scalarset I (whose elements are called pids) of cardinality N with values 1..N. Sometimes we also need to use the special value 0, representing an undefined value (as in [ID96]). We define $I_0 = I \cup \{0\}$.
- simple non-scalar datatypes such as byte, bool and mtype (an enumerated message type included in the Promela language), denoted by *NS*.

We assume that we do not have nested arrays or queues (i.e., the elements of arrays or queues cannot be in turn arrays or queues), and that our Promela model is composed with a base process G in parallel with instances P_i of a parameterized family of processes.

Definition 1. *The state of a Promela specification is described by the following quadruple* $\langle \overrightarrow{n}, \overrightarrow{s}, \overrightarrow{sn}, \overrightarrow{ss} \rangle$ *where*

- $\overrightarrow{n}$ *is a vector of values from NS (i.e., of non-scalar type)*
- $\overrightarrow{s}$ *is a vector of values in* I_0
- $\overrightarrow{sn}$ *is a vector of arrays indexed by the scalarset* I *and with range values from NS*
- $\overrightarrow{ss}$ *is a vector of arrays indexed by the scalarset* I *and with range from* I_0

One can make the following observations. Conceptually there are no local process variables: they are treated as entries of a global array indexed by the pids. In

other words, the local variables become part of $\overrightarrow{sn}$ and $\overrightarrow{ss}$. The program counter pc_i of each process i is conceptually handled as part of $\overrightarrow{sn}$.

Some datastructures are missing from Def. 1. However, without loss of generality, they can be incorporated into the state as follows:

1. Arrays $NS \to I_0$ from non-scalar to scalarset values can be seen as part of $\overrightarrow{s}$ by expanding out the array and treating each array element as a distinct variable.
2. Similarly, arrays $NS \to NS$ from non-scalar to non-scalar values can be viewed as part of $\overrightarrow{n}$ by expanding them out.
3. Queues of (scalar or non-scalar) values are translated into arrays (indexed by non-scalar values) of the same size, padded with zeroes if the queue is not full, together with an integer to record the current length of the queue. For example, $queue = [2, 3]$ of length 4 becomes $array = [2, 3, 0, 0]$, $length = 2$. The resulting arrays can then be expanded according to points 1 and 2, depending on the type of values which they contain. A queue for which messages consist of multiple fields can be handled using a series of arrays, one per field.
4. Records can be handled by treating each field as an individual variable.

Example 1. Consider the Promela code for Peterson's mutual exclusion protocol [Pet81] with 5 processes, shown in Fig. 3 and introduced in Sect. 4. For this Promela specification the components of a state s from Def. 1 will look as follows (where x^s denotes the value of the global variable x in the state s and y_i^s denotes the value of the local variable y for process i in s):

- $\overrightarrow{n} = \langle inCR^s \rangle$
- $\overrightarrow{s} = \langle turn^s[0], \ldots, turn^s[4] \rangle$
- $\overrightarrow{sn} = \langle [pc_1^s, \ldots, pc_5^s], [flag_1^s, \ldots, flag_5^s], [k_1^s, \ldots, k_5^s], [ok_1^s, \ldots, ok_5^s] \rangle$
- $\overrightarrow{ss} = \langle\rangle$

The structure of s is also depicted in Fig. 5 below.

To compute our markers (see algorithm 5.1), we use the notions of *permutation* and *mapping* of a state s as defined below where $s = \langle \overrightarrow{n}, \overrightarrow{s}, \overrightarrow{sn}, \overrightarrow{ss} \rangle$ with $\overrightarrow{sn} = \langle \overrightarrow{sn_1}, \ldots \overrightarrow{sn_k} \rangle$ and $\overrightarrow{ss} = \langle \overrightarrow{ss_1}, \ldots \overrightarrow{ss_\ell} \rangle$.

Definition 2. *A* permutation π *is a bijection from* I *to* I.
We extend the application of a permutation π *to a data value* v*, denoted by* v^π*, as follows:* $v^\pi =$

- v *if* v *is a non-scalar value or* $v = 0$
- $\pi(v)$ *if* $v \in I$
- $\langle v_1^\pi, \ldots, v_k^\pi \rangle$ *if* $v = \langle v_1, \ldots, v_k \rangle$ *is an array or vector indexed by non-scalars*
- $\langle (v_{\pi^{-1}(1)})^\pi, \ldots, (v_{\pi^{-1}(k)})^\pi \rangle$ *if* $v = \langle v_1, \ldots, v_k \rangle$ *is an array indexed by scalarset values*

The application of a permutation π *to the state* s *is defined by* $s^\pi = \langle \overrightarrow{n}, \overrightarrow{s}^\pi, \langle \overrightarrow{sn_1}^\pi, \ldots \overrightarrow{sn_k}^\pi \rangle, \langle \overrightarrow{ss_1}^\pi, \ldots \overrightarrow{ss_\ell}^\pi \rangle \rangle$.

Finally, we say that state s' is symmetric to *s iff there exists a permutation π such that $s^\pi = s'$.*

We sometimes write permutations (and mappings) in explicit form as follows: $\{1 \mapsto j_1, \ldots, N \mapsto j_N\}$.

For example, let $\pi = \{1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 3\}$ and let $a = [1, 2, 2]$ be an array $NS \to I$. Then $a^\pi = [2, 1, 1]$. However, if a is of type $I \to I$ then $a^\pi = [1, 2, 1]$.

Definition 3. *A* mapping *ρ is a function (which may not be a bijection) from I to I. We extend the application of a mapping ρ to a data value v and state s, denoted resp. by $\rho(v)$ and $\rho(s)$ in a way similar to what we did for permutations, except for arrays $v = \langle v_1, \ldots, v_k \rangle$ indexed by scalarset values which is defined by $\langle \rho(v_1), \ldots, \rho(v_k) \rangle$.*

Note that, contrary to permutations, a mapping does not necessarily permute the indexes of vectors indexed by scalarset values.

Marker algorithms for approximate and exact verification. We will now present a way to efficiently compute for any given state s a marker $m(s)$. The central idea of our approach is to analyse the current state s of a Promela specification in order to compute information about every scalarset value p. This information $m_s(p)$ is called the marker of p in s and captures structurally how p is used within s.

Definition 4. *The marker $m_s(p)$ of a scalarset value $p \in I$ in the state $s = \langle \overrightarrow{n}, \overrightarrow{s}, \overrightarrow{sn}, \overrightarrow{ss} \rangle$ is the triple $\langle \overrightarrow{s'}, \overrightarrow{sn'}, \overrightarrow{ss'} \rangle$ where*

- *$\overrightarrow{s'}$ is a vector of bits of the same length as $\overrightarrow{s}$, where $\overrightarrow{s'}_i = 1$ iff $\overrightarrow{s}_i = p$*
- *$\overrightarrow{sn'}$ is a vector of non-scalar values and of the same length as $\overrightarrow{sn}$ where $\overrightarrow{sn'}_i = \overrightarrow{sn}_i[p]$*
- *$\overrightarrow{ss'}$ is a vector of non-scalar values and of the same length as $\overrightarrow{ss}$ where $\overrightarrow{ss'}_i =$ number of occurrences of p in the range of $\overrightarrow{ss}_i$*

For a particular Promela specification with possible states S we define the set of scalarset markers $\mathcal{M} = \{m_s(p) \mid s \in S \wedge p \in I\}$. By $<_\mathcal{M}$ we denote a total order relation $<$ on $\mathcal{M}$.[1]

Consider the Peterson-5 example (Fig. 3 and Ex. 1). For $p \in I$ we have that (see also Fig. 5):

- $\overrightarrow{s'}$ is a vector of 5 bits, one for each entry of *turn*, with $\overrightarrow{s'}_i = 1 \Leftrightarrow turn^s[i] = p$
- $\overrightarrow{sn'} = \langle pc_p^s, flag_p^s, k_p^s, ok_p^s \rangle$
- $\overrightarrow{ss'} = \langle\rangle$

Our algorithm takes a state s of a Promela specification and computes the marker $m(s)$ for the state. Ideally we want the property that if two states are

[1] Such an order is easy to define, e.g. using lexicographical ordering, as no scalarset values occur inside the markers.

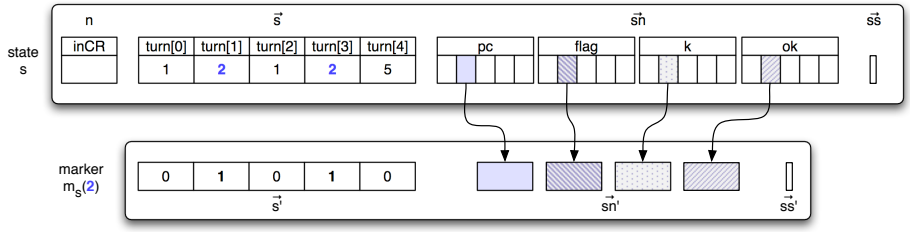

Fig. 5. The structure of Promela states and markers for Peterson-5

symmetric then they have the same marker, and vice versa. However, in order to make the computation of the marker more efficient, we are willing to accept a tradeoff. We will present two possible tradeoffs below. The method which uses the *exact markers* $m_{exact}(s)$ does not always detect that two symmetrical states are symmetric; the method which uses the *approximate marker* $m(s)$ may merge two states which are not symmetrical.

Below, by $\mid \overrightarrow{v} \mid$ we denote the length of a vector $\overrightarrow{v}$. We compute the markers for all $i \in I$ and based on the markers (which contain no scalarset values) find a way to permute the values in I. To handle the case where two values in I have the same marker, we also compute the information $local(i)$ for every $i \in I$ which captures which other markers i refers to in its entries of $\overrightarrow{ss}$ (which is usually part of its local state).

Algorithm 5.1[Computation of the markers $m(s)$ and $m_{exact}(s)$ for s]

Input: A state $s = \langle \overrightarrow{n}, \overrightarrow{s}, \overrightarrow{sn}, \overrightarrow{ss} \rangle$ of a Promela specification
Output: The markers $m(s)$ and $m_{exact}(s)$ for s
let $a = \langle m_s(1), \ldots m_s(N) \rangle$; **sort** a according to $<_{\mathcal{M}}$
let $mval_s(i) =$ **if** i=0 **then** 0 **else** $max(\{j \mid a[j] = m_s(i)\})$ **fi** ;
for $i \in I$ **do** *% compute which other markers does i refer to in its part of* $\overrightarrow{ss}$
  let $local_s[i] = \langle mval_s(\overrightarrow{ss}_1[i]), \ldots, mval_s(\overrightarrow{ss}_{|\overrightarrow{ss}|}[i]) \rangle$
od ;
let $b = \langle (m_s(1), local_s[1], 1), \ldots, (m_s(n), local_s[n], n) \rangle$;
sort b where $(m_1, l_1, n_1) < (m_1, l_2, n_2)$ iff $m_1 <_{\mathcal{M}} m_2$ or $m_1 = m_2$ and
  $l_1 <_{local_s} l_2$ (using some total order $<_{local_s}$ on arrays of numbers);
let $newval_s(i) = max(\{j \mid \exists k.b[j] = (m_s(i), local_s[i], k)\})$;
let $pos(i) =$ value j such that $b[j] = (m_s(i), local_s[i], i)$;
let $\pi = \{1 \mapsto pos(1), \ldots, N \mapsto pos(N)\}$;
let $m_{exact}(s) := s^{\pi}$; *% Apply permutation* π
let $\rho = \{pos(1) \mapsto newval_s(1), \ldots, pos(N) \mapsto newval_s(N)\}$; *% may not be a perm.*
let $m(s) := \rho(m_{exact}(s))$ *% Apply mapping* ρ

Example 2. Take the state s partially illustrated in Ex. 1. If the markers computation gives that $m_s(3) < m_s(4)$, we have $\pi = \{1 \mapsto 5, 2 \mapsto 4, 3 \mapsto 1, 4 \mapsto 2, 5 \mapsto 3\}$ and $m(s) = m_{exact}(s)$ as outlined by Fig. 6 where for $\overrightarrow{sn}$ we just showed that values initially in position 2 are now in position 4. Note that since $\overrightarrow{ss}$ is empty, a big part of the algorithm can be simplified.

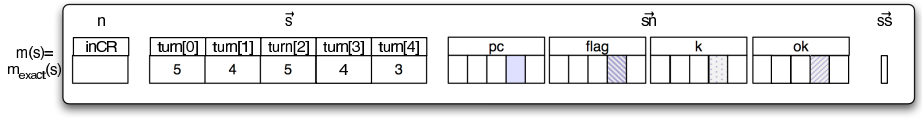

Fig. 6. $m(s)$ and $m_{exact}(s)$ for s in Fig 5

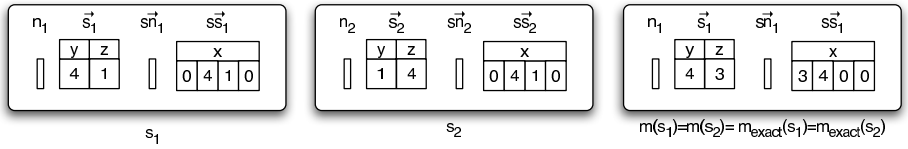

Fig. 7. $m(s)$ and $m_{exact}(s)$ for $s1$ and s_2

Example 3. In Fig. 7, the states s_1 and s_2 are symmetrical through the permutation $\pi = \{1 \mapsto 4, 2 \mapsto 3, 3 \mapsto 2, 4 \mapsto 1\}$. However, both for states s_1 and s_2, $m_s(2) = m_s(3)$ and hence without $local_s$ it would be unclear in which order to put the scalarset values 2 and 3. Our algorithm will guarantee that s_1 and s_2 have the same marker, as shown in the Fig. 7 and detailed in the following table:

element	value for s_1	value for s_2
a sorted	$m_{s_1}(2) = m_{s_1}(3) < m_{s_1}(1) < m_{s_1}(4)$	$m_{s_2}(2) = m_{s_2}(3) < m_{s_2}(4) < m_{s_2}(1)$
$local_s$	$\langle 0, 4, 3, 0\rangle$	$\langle 0, 3, 4, 0\rangle$
b sorted	$\langle\langle m_{s_1}(3), 3, 3\rangle, \langle m_{s_1}(3), 4, 2\rangle,$ $\langle m_{s_1}(1), 0, 1\rangle, \langle m_{s_1}(4), 0, 4\rangle\rangle$	$\langle\langle m_{s_2}(3), 3, 2\rangle, \langle m_{s_2}(3), 4, 3\rangle,$ $\langle m_{s_2}(4), 0, 4\rangle, \langle m_{s_2}(1), 0, 1\rangle\rangle$
π	$\langle 1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 1, 4 \mapsto 4\rangle$	$\langle 1 \mapsto 4, 2 \mapsto 1, 3 \mapsto 2, 4 \mapsto 3\rangle$

Proposition 1. *Let s, s' be states. Then the following hold:*

1. $m(s) = m(s^\pi)$ *for any permutation* π
2. $m_{exact}(s) = m_{exact}(s') \Rightarrow \exists \pi. s' = s^\pi$
3. $m_{exact}(s) = m_{exact}(s') \Rightarrow m(s) = m(s')$

Proof. Point 1 can be proven as follows. It is easy to see that $m_s(\pi(i)) = m_{s^\pi}(i)$ and hence the sorted arrays a in Alg. 5.1 are identical for s and s^π. Hence, $mval_s(\pi(i)) = mval_{s^\pi}(i)$. This in turn implies $local_s[\pi(i)] = local_{s^\pi}[i]$ and that $newval_s(\pi(i)) = newval_{s^\pi}(i)$. The only potential difference between i and $\pi(i)$ could be the value of *pos*. However, in that case there must exist another $j \in I$ with $m_s(j) = m_s(i) \wedge local_s[i] = local_s[j] \wedge newval(j) = newval(i)$ with the same value of *pos* as $\pi(i)$; and hence the resulting markers must be identical. Point 2 can be proven by composing π from Alg. 5.1 for s with the inverse of π from Alg. 5.1 for s'. Point 3 follows directly from the two other points $(m(s') = m(s^\pi) = m(s))$.

Point 1 means that all symmetries are detected by our approximate markers. Point 2 means that using exact markers yields a complete verification method.

In general the ordinary markers do not provide a complete verification method, but in the next proposition we establish a class of models for which ordinary markers do:

Peterson	SPIN (unreduced)		SymmSpin segmented		TopSPIN sorted		TopSPIN markers	
n	states	time	states	time	states	time	states	time
3	2636	0.4	494	0.3	494	0.4	494	0.3
4	60577	0.6	3106	0.4	3106	0.4	3106	0.4
5	1.56×10^6	11	17321	1	17321	1	17321	1
6	4.48×10^7	2666	89850	7	89850	7	89850	3
7	-	-	442481	85	442481	56	442481	24
8	-	-	2.09×10^6	1166	2.09×10^6	412	2.09×10^6	175
9	-	-	9.62×10^6	16673	9.62×10^6	3034	9.62×10^6	1333

Fig. 8. Symmetry markers applied to Peterson's mutual exclusion example

Proposition 2. *Let s, s' be two states. If $s = \langle \overrightarrow{n}, \overrightarrow{s}, \overrightarrow{sn}, \overrightarrow{ss} \rangle$ with $\overrightarrow{ss} = \langle\rangle$ and $m(s) = m(s')$ then $\exists \pi . s' = s^\pi$.*

Proof. We will prove that $\overrightarrow{ss} = \langle\rangle$ implies that for any s: $m_{exact}(s) = m(s)$. Hence, by Point 2 of Proposition 1 we have proven our result. First it is clear to see that if for all $i \in I$ all markers are distinct, then $m_{exact}(s) = m(s)$ as $newval(i) = pos(i)$. Let $j \in I$ be such that $newval(j) \neq pos(j)$ (i.e., there must be at least one other $k \in I$ with $k \neq j$ with $m_s(k) = m_s(j)$). In this case we know that j does not occur as a value in $\overrightarrow{s}$ (otherwise we cannot have another $k \neq j$ with the same marker). But then, as $\overrightarrow{ss} = \langle\rangle$, applying ρ has no effect for j. This reasoning can be applied to all $j \in I$ and hence our result holds.

6 Empirical Results

We have implemented symmetry reduction using symmetry markers in the TopSPIN symmetry reduction package. The result is two new TopSPIN strategies: *exact markers* and *approx markers*. Use of the *exact markers* strategy results in complete verification since the strategy guarantees that at least one state from each symmetric equivalence class is explored. On the other hand, the *approx markers* strategy does *not* guarantee sound model checking since several equivalence classes may be represented by the same state. However, our results show that this strategy can provide a reduction in verification time whilst maintaining high state-space coverage. We illustrate our implementation using two families of Promela specifications: the Peterson mutual exclusion protocol (see Sect. 4), and an email system adapted from [CM03] (and similar to an example used for experiments in [DM06]). A configuration of the email example consists of n *client* processes which exchange messages via a *mailer* process. The *mailer* is modelled using the Promela *init* process, and can be viewed as part of the *base* process discussed earlier. Experiments were carried out on the platform described in Sect. 4, and once again a '$-$' result indicates that verification was intractable, or took longer than 5 hours, for a given configuration. For each configuration we check basic safety properties expressed using assertions, and for deadlock-freedom. Note that symmetry reduction can be used, in principle, for model checking symmetric CTL* formulas [ES96]; our implementation is limited to basic safety properties due to restrictions of SPIN and TopSPIN [DM06].

Email	SPIN (unreduced)		TopSPIN segmented		TopSPIN sorted		TopSPIN exact markers		TopSPIN approx markers	
n	states	time	states	time	states	time	states	time	states	time
2	938	0.5	471	0.3	471	0.3	471	0.3	470	0.4
3	37793	0.5	6335	0.4	6361	0.5	6337	0.4	6316	0.4
4	1.33×10^6	11	56631	4	60566	2	57398	1	55711	1
5	-	-	399534	64	481964	30	430212	12	380040	9
6	-	-	2.42×10^6	1415	3.40×10^6	366	3.05×10^6	131	2.21×10^6	82
7	-	-	-	-	-	-	-	-	1.17×10^7	766

Fig. 9. Symmetry markers applied to a Promela email specification

Fig. 8 shows state-space sizes and verification times for various configurations of the Peterson protocol. To ease readability, some of the results from Fig. 4 are duplicated in Fig. 8. For the Peterson examples, the set $\overrightarrow{ss}$ is empty. Therefore, by Propositions 1 and 2, we anticipate that the *exact markers* and *approx markers* strategies should both provide full symmetry reduction *and* complete verification. This is indeed the case – the *states* column for the TopSPIN *exact markers* and SymmSPIN *segmented* strategies are identical. Results for the *approx markers* strategy are not shown in Fig. 8, as they are the same as for the *exact markers* strategy. Verification using symmetry markers is significantly faster than using the TopSPIN *sorted* or SymmSPIN *segmented* strategies.

Results for configurations of the email example are given in Fig. 9. It was not possible to apply SymmSPIN to these examples due to limitations of the prototype SymmSPIN implementation; therefore we used the TopSPIN *segmented* strategy to compute (where practical) the optimal symmetry-reduced state-space for each configuration (see Sect. 2). Fig. 9 and Fig. 10 back up the predictions of our theory: that, with systems with arrays both indexed by scalarset and range from scalarset, the *approx markers* strategy may sometimes regard inequivalent states as equivalent, whereas the *exact markers* strategy may not always recognise equivalent states as such. The left of Fig. 10 in particular (see also Fig. 1) highlights the precision of our methods – note how close the respective curves lie to the curve for full symmetry reduction (TopSPIN *segmented*). For this example, TopSPIN *sorted* also computes multiple representatives from each orbit. The results of Fig. 9 clearly illustrate the benefits of using symmetry markers:

- The *exact markers* strategy outperforms TopSPIN *sorted* both in terms of memory requirements and verification time
- The difference between the state-space size using full symmetry reduction compared with *exact markers* is relatively small
- The *approx markers* strategy provides very good coverage of the symmetry-reduced state-space, and runs significantly faster than the other strategies.

The value of the *approx markers* strategy is further illustrated by the fact that exact verification of the *email 7* configuration was not possible: full symmetry reduction using the *segmented* strategy is not feasible (for some states, as many as 7! symmetries would need to be considered), and the state-spaces generated using *exact markers* and standard TopSPIN strategies exceed memory requirements.

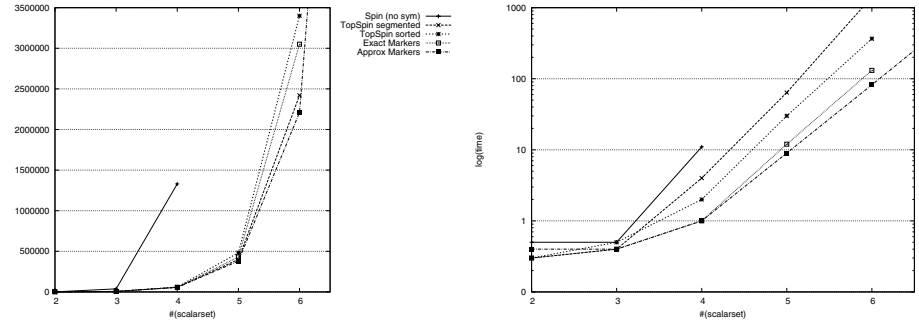

Fig. 10. Number of states and log of model checking times for the Email benchmark

Verification of deadlock-freedom using *approx markers* does not *guarantee* deadlock-freedom for the full model, but the high coverage rate of this strategy provides us with a reasonable degree of confidence that the model does not deadlock.

7 Related and Future Work

The SMC symmetry reduction tool [SGE00] employs a somewhat similar approach to our symmetry markers in order to determine state equivalence. Given two states to be tested for equivalence, SMC computes a *checksum* for each state. If the checksums are not equal then the states are not symmetrically equivalent. This simple pre-test quickly identifies many inequivalent states, but (as with markers), equality of checksums does not guarantee equivalence. The tool applies an approximate strategy to conservatively determine whether states with equal checksums are genuinely equivalent. Symmetry markers are more precise than the checksums computed by SMC, and can effectively handle the complications introduced by pid variables and pid-indexed arrays, which are not supported in the SMC input language.

Symmetry markers are currently limited to apply to fully symmetric systems. Although full symmetry occurs most frequently in practice, concurrent systems with ring or tree-like structures can exhibit other forms of structural symmetry [CEJS98, DM05]. It should be straightforward to extend the markers approach to handle multiple families of symmetric processes. A more challenging research topic involves generalising the markers approach to apply in the presence of an arbitrary symmetry group.

References

[Abr96] Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
[BDH02] Bosnacki, D., Dams, D., Holenderski, L.: Symmetric Spin. STTT 4(1), 92–106 (2002)
[CEFJ96] Clarke, E., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Form. Methods Syst. Des. 9(1-2), 77–104 (1996)

[CEJS98] Clarke, E., Emerson, E., Jha, S., Sistla, A.: Symmetry reductions in model checking. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 147–158. Springer, Heidelberg (1998)

[CGP99] Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)

[CM03] Calder, M., Miller, A.: Generalising feature interactions in email. In: FIW'03, pp. 187–204. IOS Press, Amsterdam (2003)

[DM05] Donaldson, A., Miller, A.: Automatic symmetry detection for model checking using computational group theory. In: Fitzgerald, J.A., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 481–496. Springer, Heidelberg (2005)

[DM06] Donaldson, A., Miller, A.: Exact and approximate strategies for symmetry reduction in model checking. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 541–556. Springer, Heidelberg (2006)

[DM07] Donaldson, A., Miller, A.: Extending symmetry reduction techniques to a realistic model of computation. ENTCS 185, 63–76 (2007)

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

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

[Hol03] Holzmann, G.: The SPIN model checker: Primer and reference manual. Addison-Wesley, Reading (2003)

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

[LM07] Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. In: Proc. of the International Symmetry Conference, Edinburgh, UK, pp. 71–85 (January 2007)

[McM93] McMillan, K.: Symbolic Model Checking. PhD thesis, Boston (1993)

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

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

[Val02] Valiente, G.: Algorithms on Trees and Graphs. Springer, Heidelberg (2002)