Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Symmetric Spin[⋆]

Dragan Bošnački[⋆⋆] and Dennis Dams[⋆⋆⋆] and Leszek Holenderski[†]

Address(es) of author(s) should be given

Received: date / Revised version: date

Abstract. We give a detailed description of SymmSpin, a prototype implementation of a symmetry-reduction package for the Spin model checker. It offers several heuristics for state-space reduction. A series of experiments is described, underlining the effectiveness of the heuristics and demonstrating the ability of the implementation to handle almost all of Spin's input language Promela.

1 Introduction

Model checking [7,10] is a highly automated approach to debugging and verification that is based on traversing the full state space of the system under consideration. Not surprisingly, the ability to effectively deal with large state spaces is one of the main challenges in the field. One option is to exploit symmetries in a system description. In order to grasp the idea of symmetry reduction, consider a mutual exclusion protocol based on semaphores. The (im)possibility for processes to enter their critical sections will be similar regardless of their identities, since process identities (pids) play no role in the semaphore mechanism. More formally, the system state remains behaviorally equivalent under permutations of pids. During state-space exploration, when a state is visited that is the same, up to a permutation of pids, as some state that has already been visited, the search can be pruned. The notion of behavioral equivalence used (bisimilarity, trace equivalence, sensitivity to deadlock, fairness, etc.) and the class of permutations allowed (full, rotational, mirror, etc.) may vary, leading to a spectrum of symmetry techniques.

The two main questions in practical applications of symmetry techniques are how to find symmetries in a system description, and how to detect, during exploration of the state-space, that two states are equivalent. To start with the first issue: as in any other state-space reduction technique based on behavioral equivalences, the problem of deciding equivalence of states requires, in general, the construction of the full state space. Doing this would obviously invalidate the approach, as it is precisely what we are trying to avoid. Therefore, most approaches proceed by listing sufficient conditions that can be statically checked on the system description. The second problem, of detecting equivalence of states, involves the search for a *canonical* state by permuting the values of certain, symmetric, data structures. In [6] it was shown that this *orbit problem* is at least as hard as testing for graph isomorphism, for which currently no polynomial algorithms are known. Furthermore, this operation must be performed for every state encountered during the exploration. For these reasons, it is of great practical importance to work around the orbit problem. In practice, heuristics for the graph isomorphism problem can be reused to obtain significant speedups. In case these do not work, one can revert to a suboptimal approach in which (not necessarily unique) *normalized* states are stored and compared.

The use of symmetry has been studied in the context of various automated verification techniques. We mention here only a few papers that are most closely related to our work, which is in the context of asynchronous systems. For a more complete overview we re-

⋆ This research has been supported by the VIRES project (Verifying Industrial Reactive Systems, Esprit Long Term Research Project #23498).

⋆⋆ Dept. of Computing Sci., Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands. e-mail: `D.Bosnacki@tue.nl`

⋆⋆⋆ Dept. of Electrical Eng., Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands. e-mail: `D.Dams@tue.nl`

† Dept. of Computing Sci., Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands. e-mail: `Leszek.Holenderski@philips.com`

fer to the bibliography of [24]. Emerson and Sistla have applied the idea to CTL model checking in [13], with extensions to fairness in [16] and [20]. In [13] they outlined a method for efficient calculation of a canonical representative for a special case when the global state vector consists only of individual process locations (program counters), i.e., no variables are allowed. In [14], Emerson and Trefler extended the concepts to real-time logics, while in [15] they considered systems that are *almost* symmetric and they also adapted the method for finding a canonical representative from [13] in the context of symbolic model checking. More recent papers to consider almost-symmetric systems are [11,19]. Clarke, Enders, Filkorn, and Jha used symmetries in the context of symbolic model checking in [6] where they proposed a heuristic involving multiple representatives of equivalence classes. Emerson, Jha, and Peled, and more recently Godefroid, have studied the combination of partial order and symmetry reductions, see [12,18].

Our work draws upon the ideas of Ip and Dill [22–24]. They introduce, in the protocol description language Murφ, a new data type called *scalarset* 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, that do not break symmetry; any violations can be detected at compile time.

We take the approach of Ip and Dill as the starting point. In order to work around the orbit problem, we follow their idea of *splitting* the state vector — the following is adapted from [22]:

> We separate the state into two parts. The leftmost (most significant) part is canonicalized (by picking the lexicographically smallest equivalent as a representative). Because the same lexicographical value may be obtained from different permutations, we may have a few canonicalizing permutations for this part of the state. The second, rightmost part is *normalized* by one of the permutations used to canonicalize the first part. The result is a normalized state of a small lexicographical value.

In this paper, we improve on this idea by exploiting another degree of freedom, namely the freedom to choose the ordering of variables in the state vector, on which the lexicographical ordering is based. Viewed differently, we reshuffle the positions of variables in the state vector — but only *conceptually* so! — before splitting it. In doing so, the goal is to move certain variables to the left so as to reduce the number of permutations that is determined by canonicalizing the leftmost part. Reshuffling of the state vector is done by searching for an array that is indexed by a scalarset type. This *main array* is then conceptually positioned at the leftmost end of the state vector. This results in a new heuristic for normalization, called the *sorted strategy*. A second improvement ensues by not using one of the permutations obtained from canonicalizing the leftmost part of the state vector, but using all of them in order to *canonicalize* the second part. This *segmented strategy* induces the same reduction as canonicalization of the state vector without splitting—which we have also implemented for reference purposes (*full strategy*)—but involves a much smaller overhead in time, as is demonstrated by our experiments.

We have also implemented a variation on this, which is particularly useful when no main array occurs in the system description. Namely, in the case that the process identities are of type scalarset, a main array can be coined by putting the program counters of the individual processes together in an array. The resulting strategies are called *pc-sorted* and *pc-segmented*.

We have implemented these strategies on top of the Spin model checker [21]. Building upon results reported in [12], our extension is compatible with Spin's partial order reduction algorithm, which is another, orthogonal approach to reduce the state space, and indeed one of the major strengths of the Spin tool. In addition, it handles *queues*, *multiple scalar sets*, as well as *multiple process families*. Although we can in principle handle all correctness properties that are symmetric (i.e. semantically closed under permutations of scalarset values), we focus in this paper on safety properties.

We are aware of only one other attempt to extend Spin with symmetry reductions [27]. With their implementation the user has to write a function that computes the normalized state. As a consequence, it is hard to see how this approach can be generalized. Moreover, it requires knowledge of Spin's internal workings.

This article is based on [3] and [4].

After the preliminary Section 2, the heuristic and the strategies based on it are presented in Section 3. The overall implementation in the context of Spin is described in Section 4. Having implemented this symmetry package, we have run several experiments. The results of these, and their evaluation, are the topic of Section 5.

2 Preliminaries

A *transition system* is a tuple $T = (\mathcal{S}, s_0, \rightarrow)$ where $\mathcal{S}$ is a set of states, $s_0 \in \mathcal{S}$ is an initial state, and $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation. We assume that $\mathcal{S}$ contains an error state $e \neq s_0$ which is a sink state (whenever $e \rightarrow s$ then $s = e$).

An equivalence relation on $\mathcal{S}$, say $\sim$, is called a *congruence* on T iff for all $s_1, s_2, s_1' \in \mathcal{S}$ such that $s_1 \sim s_2$ and $s_1 \rightarrow s_1'$, there exists $s_2' \in \mathcal{S}$ such that $s_1' \sim s_2'$ and $s_2 \rightarrow s_2'$. Any congruence on T induces a quotient transition system $T/{\sim} = (\mathcal{S}/{\sim}, [s_0], \Rightarrow)$ such that $[s] \Rightarrow [s']$ iff there exists a transition $q \rightarrow q'$, such that $q \in [s]$, $q' \in [s']$.

A bijection $h : \mathcal{S} \to \mathcal{S}$ is said to be a *symmetry* of T iff $h(s_0) = s_0$, $h(e) = e$, and for any $s, s' \in \mathcal{S}$, $s \to s'$ iff $h(s) \to h(s')$. The set of all symmetries of T forms a group (with function composition).

Any set A of symmetries generates a subgroup $G(A)$ called a *symmetry group* (induced by A). $G(A)$ induces an equivalence relation $\sim_A$ on states, defined as

$$s \sim_A s' \text{ iff } h(s) = s', \text{ for some } h \in G(A)$$

Such an equivalence relation is called a *symmetry relation* of T (induced by A). The equivalence class of s is called the *orbit* of s, and is denoted by $[s]_A$.

Any symmetry relation of T is a congruence on T (Theorem 1 in [23]), and thus induces the quotient transition system $T/\!\sim_A$. Moreover, s is reachable from s_0 if and only if $[s]_A$ is reachable from $[s_0]_A$ (Theorem 2 in [23]). This allows to reduce the verification of safety properties of T to the reachability of the error state $[e]$ in $T/\!\sim_A$ (via observers, for example).

In order to extend an enumerative model checker to handle symmetries, i.e., to explore $T/\!\sim_A$ instead of T, a representation for equivalence classes is needed, as well as a way to find the successors of an equivalence class. A practical choice is to use states of T as representatives of their class; any mechanism for finding successor states in T can then be reused at the level of $T/\!\sim_A$. The remaining question is how to detect whether a newly-visited state is equivalent to any state that was visited before. Clearly, the more often such equivalences are detected when present, the more reduction we get from the symmetry. In the context of using states as representatives for their class, this quality depends on how many states per class are entitled to act as representatives. A *representative function* is a function $rep : \mathcal{S} \to \mathcal{S}$ that, given a state s, returns an equivalent state from $[s]_A$. For an equivalence class C, the states in $\{rep(s) \mid s \in C\}$ are called the *representatives* of C. Clearly, if $rep(s) = rep(t)$ then states s and t are equivalent. The reverse does not hold in general, but the smaller the set of all representatives of a class, the more often it will hold. The two extremes are $rep = id$ (the identity function), which can obviously be implemented very efficiently but will never succeed to detect the equivalence of two different states ("high speed, low precision"); and $\lambda\, s \in \mathcal{S}.\, c_{[s]}$ which returns for any input s a canonical state $c_{[s]}$ that is the *unique* representative for the class $[s]$. Such a *canonical* representative function will detect all equivalences but is harder to compute ("low speed, high precision"). Now, one can explore $T/\!\sim_A$ by simply exploring a part of T, using $rep(s)$ instead of s. A generic algorithm of this type is given in Section 4. In the sequel, by a *reduction strategy* we mean any concrete *rep*.

The definition of the representatives is usually based on some partial ordering on states, e.g. by taking as representatives those states that are minimal in a class, relative to the ordering. If the ordering is total, then the representatives are unique for their class.

In practice, a transition system is given implicitly as a *program*, whose possible behaviours it models. In order to simplify the detection of symmetries on the level of the program, we assume that a simple type can be marked as *scalarset*, which essentially means that it is considered to be an unordered, finite enumerated type, with elements called *scalars*. As a representation for scalars, we use integers from 0 to $n-1$, for some fixed n. Syntactical criteria can be identified under which scalarsets induce symmetries on the program's transition system, see [23]. In this presentation, we usually restrict ourselves to one scalarset which is then denoted I. Our implementation is able to deal with multiple scalarsets however.

3 Reduction Strategies

In this section we present the heuristic to solve the orbit problem and the 4 strategies that are based on it. A complementary description of the heuristic is given in [4].

Assume a model is specified by a program in a programming language based on shared variables. The variables are typed, and the types are of two kinds only: *simple* types (say, finite ranges of integers) and *array* types. An array type can be represented by a pair $X \mapsto Y$ of types where X is the type of indices and Y is the type of elements. X must be simple while Y can be any type. Let $\mathcal{D}_T$ denote the set of all values of type T then a value of an array type $X \mapsto Y$ is a function $a : \mathcal{D}_X \to \mathcal{D}_Y$. We write $a[x]$ instead of $a(x)$.

We assume that a program specifies a set of processes run in parallel, and that the processes use only global variables. (The latter assumption is not essential. We use it only to simplify the presentation.) Let $\mathcal{V}$ denote the set of variables used in the program and $\mathcal{D}$ denote the union of $\mathcal{D}_T$, for all types T used in the program. A program P induces a transition system T_P, via a formal semantics of the programming language. We assume that in the semantics states are pairs $(s_{\mathcal{V}}, s_{PC})$ where $s_{\mathcal{V}} : \mathcal{V} \to \mathcal{D}$ is the valuation of program variables and s_{PC} represents the values of program counters, for each process in P. For $s = (s_{\mathcal{V}}, s_{PC})$ and $v \in \mathcal{V}$, $s(v)$ means $s_{\mathcal{V}}(v)$.

In order to simplify the detection of symmetries in T_P, we assume a special simple type called I which is a scalarset (i.e., an unordered subrange of integers from 0 to $n-1$, for some fixed n). Its elements are called *pids* (for *process identifiers*). We assume that all programs are of the form $B||C_0|| \cdots ||C_{n-1}$ where B is a *base* process and $C_0, \ldots, C_{n-1}$ are instances of a parameterized family $C = \lambda\, i : I\ .\ C_i$ of processes.

For such a class of programs, it is convenient to treat the program counters of the family C as an array indexed by pids, and to consider it as a global variable. Thus, in $s = (s_{\mathcal{V}}, s_{PC})$, the program counters of the family C are in fact "stored" in the component $s_{\mathcal{V}}$, and s_{PC} represents only the program counter of B.

Let $\mathcal{P}$ denote the set (in fact, the group) of all pid permutations. A pid permutation $p : I \to I$ can be lifted to states, via $p^* : \mathcal{S} \to \mathcal{S}$. Intuitively, p^* applies p to all values of type I that occur in a state, including the pids used as array indices. Formally, for $s = (s_\mathcal{V}, s_{PC})$, we have $p^*(s) = (\lambda\, v.\overline{p}_{type(v)}(s_\mathcal{V}(v)), s_{PC})$ where $\overline{p}_T : \mathcal{D}_T \to \mathcal{D}_T$ is defined as

$$\begin{aligned} \overline{p}_T(d) &= d \qquad \text{if } T \text{ is a simple type other than } I \\ \overline{p}_I(d) &= p(d) \\ \overline{p}_{X \mapsto Y}(a) &= \lambda\, x \in \mathcal{D}_X \,.\, \overline{p}_Y(a[\overline{p}_X^{-1}(x)]) \end{aligned}$$

In fact, p^* is a symmetry of T_P (Theorem 3 in [23]). Moreover, $\mathcal{P}^* = \{p^* : p \in \mathcal{P}\}$ is a symmetry group of T_P (since $(\cdot)^*$ preserves the group operations of $\mathcal{P}$). In fact, $\mathcal{P}$ and $\mathcal{P}^*$ are isomorphic (with $(\cdot)^*$ as an isomorphism).

All the reduction strategies considered in this paper have the same pattern. For a given state, a set of pid permutations is generated, and each permutation is applied to the given state. All the states obtained in this way are equivalent w.r.t. the symmetry relation, and we choose one of them as a representative. Formally, this process is specified by a function $rep : \mathcal{S} \to \mathcal{S}$ defined as

$$rep(s) = \mu(\{p^*(s) : p \in \pi(s)\})$$

where $\mu : 2^\mathcal{S} \to \mathcal{S}$ is a choice function (i.e., $\mu(X) \in X$, for any nonempty $X \subseteq \mathcal{S}$) and $\pi : \mathcal{S} \to 2^\mathcal{P}$ generates a set of pid permutations for a given state.

We call such *rep* a *general* reduction strategy since it is parameterized by π and μ. A *concrete* reduction strategy is obtained from *rep* by fixing some π, but still leaving μ free, and is denoted by rep_π. In the sequel, we consider several concrete strategies which we call full, sorted, segmented, pc-sorted and pc-segmented. The names denote the respective π functions in rep_π, and whenever we say "strategy π" we really mean "strategy rep_π".

In the full strategy, all pid permutations are taken into account, thus $\mathsf{full}(s) = \mathcal{P}$. Since this strategy is canonical, for any choice function μ, it leads to the best reduction of the state space. However, it is computationally intensive.

In order to improve the full strategy we make two assumptions. In fact, the assumptions are needed only for presentation purposes (we want to *derive* the improvements to full, and not just state them). As it will turn out later, they can be dropped.

First, we assume a choice function of a particular kind, namely the one which picks up the lexicographically smallest state, under some lexicographical ordering of states. Formally, let us assume that each simple type used in a program is totally ordered, and that the order of pids is just the natural order of the set $\{0, \ldots, n-1\}$.[1]

[1] This is not inconsistent with the assumption that I is a scalarset, and thus unordered. In fact, a scalarset can be ordered but a program that uses such a scalarset must not depend on the order (all the models of the program obtained under different orderings must be isomorphic).

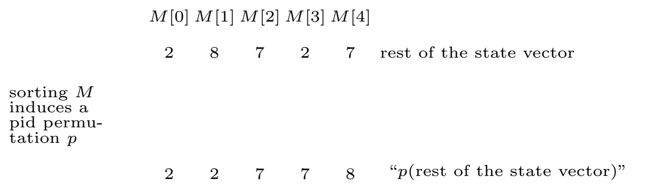
M[0] M[1] M[2] M[3] M[4]
2 8 7 2 7
rest of the state vector
sorting M induces a pid permutation p
2 2 7 7 8
"p(rest of the state vector)"

Fig. 1. A state vector before and after sorting the main array M.

As usual, such a total order can be lifted to the lexicographical order on arrays (by considering them as vectors), and then to states (by considering them as vectors).

Second, we assume that program P uses a variable $M : I \mapsto Y$ which is an array indexed by pids and whose elements do not involve pids. M is called a *main array*. Most real protocols specified in a language based on shared variables, and of the form $B||C_0||\cdots||C_{n-1}$, use such an array, either directly or indirectly. (Notice that each local variable declared in the parametric program $C = \lambda\, i : I \,.\, C_i$, and whose type does not involve I, can be considered an element of an array indexed by I, when lifted to the global level.) We further assume that M dominates in the total order used by our particular choice function μ, in the sense that the main array is located at the beginning of the state vector. (If it is not the case, the state vector can be reshuffled.)

Let us consider a state s. Notice that if $rep_{\mathsf{full}}(s) = s'$ then $s'(M)$ must be sorted, w.r.t. the ordering of Y, due to our particular choice of μ. So instead of considering all pid permutations in full it is enough to consider only those which sort $s(M)$. (Notice that there may be more than one sorting permutation, if $s(M)$ contains repeated values.) Let $\vec{s}(M)$ denote the set of all sorting permutations.

In the sorted strategy we consider just one pid permutation $\vec{p} \in \vec{s}(M)$ obtained by applying a particular sorting algorithm to $s(M)$. (Formally, $\mathsf{sorted}(s) = \{\vec{p}\}$.) Obviously, the rep_{sorted} strategy is faster then rep_{full}. Unfortunately, it is not canonical since $\vec{p}$ minimizes only $s(M)$ and not necessarily the whole s. (If $s(M)$ contains repeated values then there may be another sorting permutation p such that $p^*(s)$ is smaller than $\vec{p}^*(s)$.)

In the segmented strategy we consider all the permutations in $\vec{s}(M)$.[2] (Formally, $\mathsf{segmented}(s) = \vec{s}(M)$.) Obviously, $rep_{\mathsf{segmented}}$ is canonical, for the particular choice function we have assumed. Moreover, $rep_{\mathsf{segmented}}(s) = rep_{\mathsf{full}}(s)$, for any s.

As an example, consider Figure 1 showing a state vector before and after sorting its main array M. M is indexed by pids, which in this case range from 0 to 4. M's elements are the numbers 2, 7, and 8, taken from some type that differs from I. Suppose that the particular

[2] The name "segmented" comes from the way we have implemented the computation of $\mathsf{segmented}(s)$. We first sort $s(M)$, locate all the segments of equal values in the sorted array, and permute these segments independently.

sorting algorithm being used sorts M as indicated by the dashed arrows. This particular sorting induces the pid permutation $p = \{0 \mapsto 0, 1 \mapsto 4, 2 \mapsto 2, 3 \mapsto 1, 4 \mapsto 3\}$, which is then applied to the rest of the state vector in order to obtain the representative under the sorted strategy. If we applied *all* pid permutations to the upper state vector in the picture, then the lexicographical minimum among the results, call it $s_{\min}$, 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 initial values, for example $p' = \{0 \mapsto 1, 1 \mapsto 4, 2 \mapsto 3, 3 \mapsto 0, 4 \mapsto 2\}$, but may give smaller results than p. The segmented strategy applies all the permutations that sort M (in this example there are four of them) on the whole state vector, and selects the smallest among the results, which is then guaranteed to be $s_{\min}$.

The requirement that program P uses an explicit main array can be dropped, by observing that in every program of the form $B||C_0|| \cdots ||C_{n-1}$ there is, in fact, an implicit array indexed by pids, namely the array of program counters for processes $C_0, \ldots, C_{n-1}$. Thus, we can consider the variants of sorted and segmented in which we use the array of program counters instead of M. The variants are called pc-sorted and pc-segmented, respectively. If P contains a main array as well then both sorted/segmented and pc-sorted/pc-segmented are applicable to P, so the question arises which of the combinations is better. We cannot say much about sorted versus pc-sorted, in general. However, segmented and pc-segmented are both canonical, so they are equally good, as far as the reduction of a state space is concerned.

The following result allows us to drop the assumption about our particular μ function.

Theorem 1. *For any choice function μ, $rep_{\mathsf{segmented}}$ is canonical.*

Proof. Let $P(s) = \{p^*(s) : p \in \mathsf{segmented}(s)\}$. We have to show that if $s \sim s'$, i.e., there exists a pid permutation p such that $p^*(s) = s'$, then $\mu(P(s)) = \mu(P(s'))$. In order to show it for any choice function μ, it is enough to show that $P(s) = P(s')$. Assume $s_1 \in P(s)$ then $s_1 = p_1^*(s)$ for some pid permutation p_1, where p_1^* sorts the array M in s. Observe that $s = (p^{-1})^*(s')$ where p^{-1} is the inverse of p. Hence, $s_1 = p_1^*((p^{-1})^*(s'))$. Since $p_1^* \circ (p^{-1})^*$ sorts the array M in s', $s_1 \in P(s')$. Similarly, if $s_1 \in P(s')$ then $s_1 \in P(s)$. □

Remark 1. In the proof, the only step specific to the segmented strategy is the observation that if p establishes the equivalence of states s and s', and p_1^* sorts the array M, then also $p_1^* \circ (p^{-1})^*$ sorts the array M. Thus, the theorem can be generalized to any strategy π that preserves pid permutations in the following sense: if p establishes the equivalence of states s and s' then for any $p_1 \in \pi(s)$, $p_1^* \circ (p^{-1})^* \in \pi(s')$. For example, this condition holds for the full and pc-segmented strategy as well.

The theorem has an important consequence in practice. Suppose we want to extend an existing enumerative model checker with symmetry reductions. Usually, a model checker stores a state in a continuous chunk of memory. A very efficient way of implementing μ is to choose the lexicographically smallest chunk by simply comparing such memory chunks byte by byte (for example, in C, the built-in function `memcmp` could be used for this purpose). Obviously, such μ is a choice function, no matter where the main array resides in the memory chunk. Thus, the requirement that the main array dominates in the total order used by μ is not needed. Also, the array of program counters, used in pc-sorted and pc-segmented, need not even be contiguous. As a consequence, we do not need to reshuffle a state vector in order to use the presented strategies.

Finally, the assumption that programs are of the form $B||C_0|| \cdots ||C_{n-1}$with process indices in the scalarset I is only needed to formalize the pc-sorted and pc-segmented strategies. For segmented and sorted, the only essential assumption about I is that it is a distinguished scalarset.

In Section 4.4.4 we estimate the theoretical complexity of our strategies. Their factual performance is compared, on several examples, in Section 5.

4 Extending Spin with Symmetry Reductions

In order to compare the performance of the various reduction strategies in practice, we have embedded them into the enumerative model checker Spin [21]. The result of our extension of Spin with a symmetry package is called SymmSpin.

When extending an enumerative model checker, in the spirit of [23], two essential problems must be solved. First, the input language must be extended, to allow the usage of scalarsets in a program that specifies a model. Second, a symmetry reduction strategy must be added to the state space exploration algorithm. Both problems are non-trivial. As far as the extension of the input language is concerned, a compiler should be able to check whether the scalarsets really induce a symmetry relation (i.e., whether a program does not rely on the order of the range of integers that represents a scalarset). As far as adding a reduction strategy is concerned, the *rep* function should be implementable in an efficient way.

In order for a model checker to be a successful tool for the verification of symmetric systems, good solutions are needed for both problems. However, there does not seem to be much sense in putting an effort into solving the language extension problem without having an efficient reduction strategy. That is why in SymmSpin we have mainly concentrated on the second problem.

```
1 $reached := unexpanded := \{s_0\}$;
2 while $unexpanded \neq \emptyset$ do
3     remove a state $s$ from $unexpanded$;
4     for each transition $s \to s'$ do
5         if $s' = \textbf{error}$ then
6             stop and report error;
7         if $s' \notin reached$ then
8             add $s'$ to $reached$ and $unexpanded$;
```

Fig. 2. A standard exploration algorithm

```
1 $reached := \{rep(s_0)\}$; $unexpanded := \{s_0\}$;
2 while $unexpanded \neq \emptyset$ do
3     remove a state $s$ from $unexpanded$;
4     for each transition $s \to s'$ do
5         if $s' = \textbf{error}$ then
6             stop and report error;
7         if $rep(s') \notin reached$ then
8             add $rep(s')$ to $reached$ and $s'$ to $unexpanded$;
```

Fig. 4. Our exploration algorithm with symmetry reductions

```
1 $reached := unexpanded := \{rep(s_0)\}$;
2 while $unexpanded \neq \emptyset$ do
3     remove a state $s$ from $unexpanded$;
4     for each transition $s \to s'$ do
5         if $s' = \textbf{error}$ then
6             stop and report error;
7         if $rep(s') \notin reached$ then
8             add $rep(s')$ to $reached$ and $unexpanded$;
```

Fig. 3. A standard exploration algorithm with symmetry reductions

As far as the first problem is concerned, it could be solved just by lifting the syntactical extensions of Murφ [23] to Spin. Although not entirely straightforward, it should not pose fundamental difficulties. Unfortunately, to do it in the right way, one would need to extensively modify the existing Promela parser. In SymmSpin, we have tried to avoid this effort, as explained in Section 4.3.

4.1 A Modified State Space Exploration Algorithm

In principle, extending an existing enumerative model checker to handle symmetries is not difficult, once the *rep* function is implemented. Instead of using a standard algorithm for exploring $T = (\mathcal{S}, s_0, \to)$, as depicted in Fig. 2, one explores the quotient $T/{\sim}$ using a simple modification of the algorithm, as depicted in Fig. 3. (This modification is borrowed from [23].) Obviously, the algorithm in Fig. 3 deals with smaller number of states than the algorithm in Fig. 2.

In practice, we had to overcome several problems, due to idiosyncrasies of Spin. For example, it turned out that the operation "add $rep(s')$ to *unexpanded*" is difficult to implement reliably, due to a peculiar way Spin represents the set *unexpanded* as a set of "differences" between states rather than states themselves. For this reason, we had to change the exploration algorithm given in Fig. 3. In our[3] algorithm, the original states, and not their representatives, are used to generate the state space to be explored, as depicted in Fig. 4. (The algorithms differ only in lines 1 and 8.)

Using the notation of Section 2, consider a state transition system $T = (\mathcal{S}, s_0, \to)$ and some group $G(A)$ of symmetries. Since the symmetry relation $\sim_A$ is a congruence, obviously the algorithm in Fig. 4 visits at least one representative of each equivalence class (orbit) induced by $\sim_A$. In other words, after the algorithm ends, provided that no error is reported, for each state $s \in S$, there is at least one $s' \in$ *reached* such that $s' \in [s]_A$. Using the results of [23], this implies directly the correctness of the algorithm in Fig. 4, i.e., the algorithm reports an error iff there exists an error state $e \in \mathcal{S}$.

If the *rep* function is canonical, the algorithms from Fig. 3 and 4 explore the same number of states, provided that no error is reported. This follows from the following reasoning. The total number of visited states is bounded by the number of representatives, i.e., the number of different states s such that $s = rep(s')$, for some $s' \in S$. This is because the number of visited states is equal to the number of states which are removed from *unexpanded* in line 3. Since a state s is added to *unexpanded* only if its representative is not already in *reached* (lines 7 and 8), the total number of states which are added to *unexpanded* cannot exceed the number of representatives. On the other hand, as we already mentioned, the algorithm is guaranteed to visit at least one representative of each equivalence class. Thus, if *rep* is canonical, the upper and lower bounds of visited states are the same and they are equal to the number of equivalence classes. Moreover, the number of explored transitions is also the same in both algorithms. This follows from the fact that for each state s for which our algorithm computes its successors with the statement "for each transition $s \to s'$ do" (line 4), the algorithm in Fig. 3 computes the successors of $rep(s)$. Since s and $rep(s)$ are related by a symmetry, they must have the same number of successors (recall that any symmetry is a bijection).

If an error is reported, one can show that, under some constraints on the nondeterministic parts of the algorithms (and provided that *rep* is canonical), both of them visit the same number of states and explore the same number of transitions:

Proposition 1. *Given a transition system T, let rep be canonical and let the following holds for the algorithms in Figures 3 and 4:*

1. *The order in which a state s is chosen from the set unexpanded in line 3* (remove a state s from *unex-*

[3] We acknowledge a remark from Gerard Holzmann which led us to this algorithm.

panded) in the algorithm in Fig. 4 is such that: For each two states s_1, s_2, it is the case that s_1 is chosen before s_2 iff in the same line in the algorithm in Fig. 3 $rep(s_1)$ is chosen before $rep(s_2)$.

2. The order in which transitions are explored in line 4 (for each transition $s \rightarrow s'$ do) *in the algorithm in Fig. 4 is such that: For each two transitions $s \rightarrow s'$, $s \rightarrow s''$, it is the case that $s \rightarrow s'$ is chosen before $s \rightarrow s''$ iff in the same line in in the algorithm in Fig. 3 the transition $rep(s) \rightarrow s_1$, where $rep(s_1) = rep(s')$, is chosen before the transition $rep(s) \rightarrow s_2$, where $rep(s_2) = rep(s'')$.*

Then, when applied on T, for both algorithms the total number of states that are removed from unexpanded in line 3 is the same, as well as the total number of transitions that are explored in line 4.

Proof. For brevity, we denote the algorithms in Fig. 3 and 4 with A3 and A4, respectively. In order to prove the property we show that each execution step of A3 can be mimicked by an execution step of A4. To this end we prove that at each point the following invariants hold:

1. the set of *reached* states is the same in both executions,
2. there is a bijection between the sets *unexpanded* such that each state s from *unexpanded* of A4 is mapped into the state $rep(s)$ in *unexpanded* of A3.

The proof is by induction on the length of the execution sequences of A3 and A4. *Base case*: Obviously the invariants are preserved after the initialization of *reached* and *unexpanded* in line 1. *Inductive step*: Constraint (1) and invariant (2) (induction hypothesis) imply that, if a state s is chosen in line 3 of A3, then a state s' is chosen in the same line of A4 such that $rep(s') = s$. Similarly, because of constraint (2), if a transition $s \rightarrow s_1$ is chosen in line 4 of A3, then the transition $s' \rightarrow s'_1$ is chosen in the same line of A4 such that $s = rep(s')$ and $s_1 = rep(s'_1)$. (As any symmetry is bijection, the same number of transitions are chosen in both executions in line 4.) Thus, from the discussion above it follows that, if an error is reported (line 5) by A3, then it is reported also by A4. Further, having in mind invariant (1) (induction hypothesis) we can conclude that the boolean expression "$rep(s') \notin reached$" (line 7) has the same value in both executions. This in combination with constraint (2) also implies that the invariants are preserved by the statements in line 8 that update *reached* and *unexpanded*. □

If *rep* is not canonical, then the number of explored states and transitions differs between the two algorithms and depends on the choice of *rep*. The algorithms are incomparable in this case, because it can happen that our algorithm explores fewer states and/or transitions than the algorithm in Fig. 3, or vice versa.

An advantage of our algorithm is that it allows (regardless of the fact whether *rep* is canonical or not) to easily regenerate an erroneous trace from *unexpanded*, in case an error is encountered. Since Spin explores the state space in a depth first manner, the set *unexpanded* is, in fact, structured as a stack. When an error is encountered, the stack contains the sequence of states that lead to the error, and its contents can then directly be dumped, giving the erroneous trace. In the algorithm from Fig. 3, the stack would contain the representatives of the original states, and since the representatives are not necessarily related by the transition relation in the original model, the stack would not necessarily represent an existing trace in the original model.

We have put considerable effort into efficiently implementing the 5 variants of the `rep` function. For example, it turns out that all the sets of pid permutations used in our reduction strategies can always be enumerated starting with an initial permutation and then composing it with transpositions (permutations that swap two elements). As a consequence, the most costly operation p^* (that applies a given pid permutation to a given state) can be optimized by using two versions of p^*. In addition to the general version that is applicable to any pid permutation, we also use a restricted (and more efficient) version that is only applicable to transpositions. Our implementation is described in more detail below in 4.2.

In the verification experiments described in Section 5, we used Spin in two ways: with and without its partial order reduction (POR) algorithm. Allowing Spin to use its POR algorithm together with our symmetry reductions is sound due to Theorem 19 in [12] which guarantees that the class of POR algorithms to which the Spin's POR algorithm belongs, is compatible with the generic symmetry reduction algorithm. With a straightforward modification, the theorem's proof is valid for our algorithm as well.

4.2 An Overview of SymmSpin

Our goal was first to experiment with various reduction strategies, to check whether they perform well enough to undertake the effort of fully extending Spin with a symmetry package (such as modifying the Promela parser). So the problem was how to extend Spin in a minimal way that would be sufficient to perform various verification experiments with some symmetric protocols. In fact, we have managed to find a way which does not require any change to the Spin tool itself, but only to the C program generated by Spin.

The C program generated by Spin is kept in several files. Two of them are of particular interest to us: `pan.c` that implements a state exploration algorithm, and `pan.h` that contains the declarations of various data structures used in `pan.c`, including a C structure called `State` that represents a state of the transition system being verified. Our current extension of Spin with symmetry reductions simply adds a particular reduction strategy to `pan.c`. This is done in two steps.

First, we locate in `pan.c` all calls to the procedures that add a newly visited state to the set of already visited states. The calls have the generic form `store(now)` where `now` is a global variable that keeps the current state. All the calls are changed to `store(rep(now))` where `rep` is the name of a C function that implements the reduction strategy. In this way, the representatives, and not the states themselves, are stored in the set of already visited states. The `pan.c` code that computes the successor states of `now` is left unchanged. As a consequence, the original states, and not their representatives, are used to generate the state space to be explored. This agrees with the exploration algorithm in Fig. 4.

Second, the C code for `rep` is generated and added to `pan.c`. This step is not straightforward since we have to scan `pan.h` for various pieces of information needed for the implementation of `rep`.

The two steps are performed by a Tcl script called `AdjustPan`. The current version of the script is about 1800 lines, not counting comments.

4.3 The `AdjustPan` *Script*

Conceptually, the symmetry relation is deduced from a Promela program, say `foo.pml`, that uses special types called *scalarsets* which are unordered ranges of integers (in SymmSpin, always of the form $0..n-1$, for some constant $n < 256$).

Actually, in order to avoid modifications to the Promela parser, there is no special declaration for scalarsets. Instead, the standard Promela type `byte` is used for this purpose. Also, the symmetry relation is not deduced from `foo.pml` itself. Instead, it is deduced from an additional file, say `foo.sym`, that must accompany the Promela program. The additional file (later called a *system description file*) must be prepared by a user, and must contain all information relevant to the usage of scalarsets in the original Promela program that specifies a symmetric concurrent system.

The precise description of the syntax of the system description file, and its meaning, is beyond the scope of this paper. In short, it resembles the declaration part of Promela, and allows to describe all the data structures and processes, appearing in the original Promela program, that depend on scalarsets. This description is then used by the `rep` function to locate all fragments of the Spin state vector that depend on a particular scalarset, and lift a permutation of the scalarset to the state.

In the current implementation we assume that the safety properties to be verified are symmetric, i.e., invariant under pid permutation (also see [6,13]). Indeed, we get the symmetry of the properties for free. This is because we deal only with safety properties which are expressed as assertions, i.e. with `assert` statements in the Promela model. These `assert` statements, just as any other statement, must obey the syntactic restrictions of [23] on the scalarset (pid) type. As a consequence, the properties they express are invariant under permutation of pids. Further, the syntactic criteria guarantee that the obtained state space remains symmetric in presence of such symmetric assertions.

The only important restriction in the current version of SymmSpin is that the concurrent system specified by a symmetric Promela program must be static, in the sense that all the processes must be started simultaneously, by the `init{atomic{...}}` statement. Lifting this restriction would probably substantially complicate the implementation.

The `AdjustPan` script is called with two parameters: the name of one of the 5 reduction strategies described in Section 3, and the name of a system description file. The script reads three files: `pan.h`, `pan.c` (both generated by Spin from `foo.pml`), and `foo.sym`. The information in `foo.sym` and `pan.h` is used to modify `pan.c`. The modified version of `pan.c` is stored under the name `pan-sym.c`, and is used to model check the symmetric Promela program.

In summary, SymmSpin is used in the following way:

– Write a symmetric Promela program, say `foo.pml`, and its system description file, say `foo.sym`.
– Run Spin (with the `-a` option) on `foo.pml` (this will generate `pan.h` and `pan.c` files).
– Run `AdjustPan` on `foo.sym` with a particular reduction strategy (this will generate `pan-sym.c` file).
– Compile `pan-sym.c` with the same options you would use for `pan.c`, and run the generated binaries to model check the symmetric Promela program.

4.4 The Implementation of Strategies

In this section we highlight some implementation details of SymmSpin, to give an idea of how the `rep` function is implemented in an efficient way. To simplify presentation, we assume that only one scalarset is used in a Promela program.

The canonical strategies full, segmented and pc-segmented are implemented by a C function of the following shape:

```
State tmp_now, min_now;

State *rep(State *orig_now) {
  /* initialize */
    memcpy(&tmp_now, orig_now, vsize);
  /* find the representative */
    memcpy(&min_now, &tmp_now, vsize);
    ...
  return &min_now;
}
```

The parameter `orig_now` is used to pass the current state `now`. In order to avoid any interference with the original code in `pan.c` that uses `now` for its own purposes (for example, to generate the successor states of the current state), we must assure that `rep` does not modify now. For this reason, we copy orig_now to the auxiliary state tmp_now.

The representative is found by enumerating permutations of a scalarset, and applying them to (the copy of) the current state. The lexicographically smallest result is kept in min_now. After each permutation, it is updated by the following statement

```
if (memcmp(&tmp_now, &min_now, vsize) < 0)
  memcpy(&min_now, &tmp_now, vsize);
```

In strategies sorted and pc-sorted, only one permutation is considered, so the auxiliary state min_now is not needed. Hence, the strategies are implemented by a function of the following shape:

```
State tmp_now;

State *rep(State *orig_now) {
  /* initialize */
    memcpy(&tmp_now, orig_now, vsize);
  /* find the representative */
    ...
  return &tmp_now;
}
```

In the rest of this section, we present the rep functions for the full, sorted and segmented strategies. The pc-sorted and pc-segmented strategies are similar to the sorted and segmented strategies.

4.4.1 Strategy full

```
State tmp_now, min_now;

State *rep_full(State *orig_now) {
  memcpy(&tmp_now, orig_now, vsize);
  /* find the representative */
    memcpy(&min_now, &tmp_now, vsize);
    permute_scalar(SIZE_OF_SCALAR);
  return &min_now;
}
```

The representative is found by the permute_scalar procedure that takes the size of a scalarset, say *size*, and generates all permutations of numbers from 0 to $size - 1$, excluding the identity permutation. Each permutation is then applied to the current state. Applying a permutation may be quite expensive. It can be implemented more efficiently if a permutation is a transposition (i.e, a swap of two numbers). For this reason, the permutations are generated incrementally, by composing successive transpositions (starting from the identity permutation). A tricky algorithm (borrowed from [25], and presented in Fig. 5) is used for this purpose.

It happens that the transpositions generated by the algorithm always swap two successive elements p and $p + 1$, but we do not rely on this feature. Whenever a new transposition is computed, permute_scalar calls apply_swap. The apply_swap procedure has the following header:

```
void permute_scalar(int size) {
  int i, p, offset,
      pos[MAX_SCALAR_SIZE],dir[MAX_SCALAR_SIZE];

  for (i = 0; i < size; i++) {
    pos[i] = 1; dir[i] = 1;
  }
  pos[size-1] = 0;

  i = 0;
  while (i < size-1) {
    for (i = offset = 0;pos[i] == size-i;i++) {
      pos[i] = 1; dir[i] = !dir[i];
      if (dir[i]) offset++;
    }
    if (i < size-1) {
      p = offset-1 +
          (dir[i] ? pos[i] : size-i-pos[i]);
      pos[i]++;

      /* apply transposition p <-> p+1 */
      apply_swap(&tmp_now, p, p+1);
      if (memcmp(&tmp_now, &min_now, vsize) < 0)
        memcpy(&min_now, &tmp_now, vsize);
    }
  }
}
```

Fig. 5. Generating permutations by transpositions

```
void apply_swap(State *state, int v1, int v2)
```

It lifts the transposition of scalar values v_1 and v_2 to a given state. The lifting is performed *in situ*, by modifying the given state. The body of apply_swap is generated by AdjustPan, using the information given in a system description file. The generated C code is straightforward. It consists of a sequence of guarded assignments that swap v_1 with v_2, for each variable that depends on a scalarset.

For example, if x is a global variable of a scalarset type then the following code fragment is generated:

```
if (state->x == v1) state->x = v2; else
if (state->x == v2) state->x = v1;
```

For arrays, the code is more complex. For example, if x is a global array of a scalarset type, and indexed by the same scalarset, then the following code fragment is generated:

```
/* swap values */
for (i = 0; i < SCALAR_SIZE; i++) {
  if (state->x[i] == v1) state->x[i] = v2; else
  if (state->x[i] == v2) state->x[i] = v1;
}
/* swap indices */
{ uchar tmp;
  tmp = x[v1]; x[v1] = x[v2]; x[v2] = tmp; }
```

In addition, for every family of processes indexed by a scalarset, say proctype P(scalar i), apply_swap swaps the two chunks of memory (in a state vector) that correspond to $P(v_1)$ and $P(v_2)$.

4.4.2 Strategy sorted

```
State *rep_sorted(State *orig_now) {
  int perm[MAX_SCALAR_SIZE];
  memcpy(&tmp_now, orig_now, vsize);
  /* sort the main array and compute
     the sorting permutation */
    ...
  /* find the representative */
    apply_perm(perm, &tmp_now);
  return &tmp_now;
}
```

The main array is sorted using a straightforward algorithm that successively finds minimal elements. Its quadratic complexity is acceptable since the size of a scalarset is usually small. On the other hand, it allows to compute the sorting permutation with minimal cost, since each element is swapped only once.

The `apply_perm` procedure has the following header:

```
void apply_perm(int *perm, State *state)
```

It lifts the given scalarset permutation to `state`. As in `apply_swap`, the lifting is performed *in situ*, and it consists of a sequence of guarded assignments, for each variable that depends on a scalarset. For example, if x is a global variable of a scalarset type then the following code fragment is generated:

```
if (state->x < SCALAR_SIZE)
  state->x = perm[state->x];
```

The guard is needed to solve a subtle problem with the initialization of scalarset variables. Since all variables used in a standard Promela program are automatically initialized with 0, it is common to use 0 to represent an undefined value. Unfortunately, this convention cannot be used for scalarsets. The reason is that in our implementation, a scalarset of size n is a range of integers from 0 to $n-1$, so 0 must be treated as other well-defined values (otherwise, the symmetry would be broken). Thus, a value outside of the range must be used for an undefined value. By convention, we use n for this purpose, and the guard guarantees that the undefined value is treated in a symmetric way (i.e., it is never permuted, as required in [23]). In fact, any value not less than n can be used to represent the undefined value.

For arrays, the code is more complex. For example, if x is a global array of a scalarset type, and indexed by the same scalarset, then the following code fragment is generated:

```
/* permute values */
for (i = 0; i < SCALAR_SIZE; i++) {
  if (state->x[i] < SCALAR_SIZE)
    state->x[i] = perm[state->x[i]];
}
/* permute indices */
{ uchar buf[SCALAR_SIZE];
  memcpy(buf, state->x, sizeof(state->x));
  for (i = 0; i < SCALAR_SIZE; i++)
    state->x[perm[i]] = buf[i];
}
```

Notice that when permuting indices we have to use a buffer.

4.4.3 Strategy segmented

```
State *rep_segmented(State *orig_now) {
  int perm[MAX_SCALAR_SIZE];
  memcpy(&tmp_now, orig_now, vsize);
  /* sort the main array and compute
     the sorting permutation */
    ...
  /* locate blocks */
    ...
  /* find the representative */
  apply_perm(perm, &tmp_now);
  memcpy(&min_now, &tmp_now, vsize);
  if (num_of_blocks > 0)
    permute_blocks(0, block_start[0],
                   block_size[0]);
  return &min_now;
}
```

First, the main array is sorted as in the sorted strategy. Second, the segments of equal values are located, in the sorted main array, by a straightforward linear algorithm. The information about the segments (called blocks henceforth) is stored in the following global data structures:

```
int num_of_blocks;
int block_start[MAX_SCALAR_SIZE],
    block_size[MAX_SCALAR_SIZE];
```

Finally, the canonical representative is found by procedure `permute_blocks` that generates all permutations of indices in successive blocks, excluding the identity permutation. Its code is given in Fig. 6. Each permutation is then applied to the current state, and the lexicographically smallest result is chosen.

The procedure uses double recursion, to assure that the number of calls to `apply_swap` and comparisons between `tmp_now` and `min_now` is minimal. It is a generalization of `permute_scalar` used in `rep_full`. Conceptually, the indices of separate blocks, when considered relative to the start of a respective block, can be perceived as separate scalarsets. Inside one block, all transpositions of the block's indices are generated as in `permute_scalar`. The double recursion is used to properly compose the permutations of separate blocks, by chaining the invocations of `permute_blocks`.

4.4.4 Cost of full, sorted and segmented

Let us recall the general formula for computing a representative in all our strategies:

$$rep(s) = \mu(\{p^*(s) : p \in \pi(s)\})$$

```
void permute_blocks(int block, int start,
                    int size) {
  int i, p, offset,
      pos[MAX_SCALAR_SIZE],dir[MAX_SCALAR_SIZE];

  /* go to the last block */
    if (++block < num_of_blocks)
      permute_blocks(block, block_start[block],
                     block_size[block]);
    block--;

  /* the same as permute_scalar, but apply
     transposition is changed to */
  ...
    swap_in_block(block, p, p+1);
  ...
}

void swap_in_block(int block, int p1, int p2) {
  /* apply transposition p1 <-> p2 */
    apply_swap(&tmp_now, p1, p2);
    if (memcmp(&tmp_now, &min_now, vsize) < 0)
      memcpy(&min_now, &tmp_now, vsize);
  /* permute the next block */
    if (++block < num_of_blocks)
      permute_blocks(block, block_start[block],
                     block_size[block]);
}
```

Fig. 6. Permuting blocks

Let $|s|$ denote the size of state s (say, in bytes), $|\pi(s)|$ denote the cardinality of set $\pi(s)$, and $\langle x\rangle$ denote the cost of computing x. Using this notation,

$$\langle rep(s)\rangle = \langle\pi(s)\rangle + |\pi(s)|\cdot\langle p^*(s)\rangle + \langle\mu(X)\rangle$$

where $X = \{p^*(s) : p \in \pi(s)\}$.

In all our strategies, $\mu(X)$ finds the lexicographically smallest state in set X, using the linear search algorithm that performs $|X|-1$ comparisons between states. Notice that $|X| = |\pi(s)|$ so $\langle\mu(X)\rangle$ can be approximated by $(|\pi(s)|-1)\cdot|s|$ where the number of state comparisons is $|\pi(s)|-1$ and the cost of one such comparison comes down to $|s|$. On the other hand, $\langle p^*(s)\rangle$ (i.e., the cost of lifting a pid permutation to a state) can be approximated by $|s|$. Thus,

$$\langle rep(s)\rangle \approx (2|\pi(s)|-1)\cdot|s| + \langle\pi(s)\rangle.$$

For full, $\langle\pi(s)\rangle$ is the cost of generating all pid permutations, so it is proportional to $n!$ where n is the size of scalarset I. Also, $|\pi(s)| = n!$. Thus, $\langle rep_{\mathsf{full}}(s)\rangle \approx n!\cdot|s|$.

For sorted, $\langle\pi(s)\rangle$ is the cost of generating one permutation of pids that sorts an array of size n. This cost depends on the algorithm used to sort the array. In our implementation, we use a naive n^2 algorithm. There is no need for a more sophisticated $n\log(n)$ algorithm since n is usually very small (say, less than 5). Notice that $|\pi(s)| = 1$. Thus, $\langle rep_{\mathsf{sorted}}(s)\rangle \approx n^2 + |s|$.

For segmented, $\langle\pi(s)\rangle$ is proportional to $n^2+|\pi(s)|-1$ since we first generate a sorting permutation of pids (in cost n^2) and then generate the rest of permutations in the set $\pi(s)$ (in cost $|\pi(s)|-1$). For $n > 0$,

$$\langle rep_{\mathsf{sorted}}(s)\rangle \le \langle rep_{\mathsf{segmented}}(s)\rangle \le \langle rep_{\mathsf{full}}(s)\rangle.$$

5 Experimental Results

We tried our prototype implementation on several examples. The obtained reductions were often very close to the theoretical limits, thus, we were able to obtain reductions of several orders of magnitude in the number of states. Also, due to the significantly smaller number of states that had to be explored, in all the cases the verification with symmetry reduction was faster than the one without it. The experiments showed that there is no favorite among the reduction strategies regarding the space/time ratio. This suggests that it makes sense to have all strategies (maybe except full) as separate options of the extended model-checker.

In the verification experiments we used Spin and SymmSpin both with and without the partial order reduction (POR) option. In most of the cases there was a synergy between the symmetry and the partial order reductions. The two reduction techniques are orthogonal because they exploit different features of the concurrent systems, therefore, their cumulative effect can be used to obtain more efficient verification.

In the sequel, we present the results for four examples. All experiments were performed on a Sun Ultra-Enterprise machine, with three 248 MHz Ultra SPARC-II processors and 2304MB of main memory, running the SunOS 5.5.1 operating system. In the tables below the verification times (in the rows labeled with "t") are given in seconds (*s.x*), minutes (*m:s*), or hours (*h:m:s*); the number of states (in the rows labeled with "s") is given directly or in millions (say, 9.1M); *o.m.* stands for out of memory, and *o.t.* denotes out of time (more than 10 hours); +POR and -POR mean with and without POR, respectively.

5.1 Peterson's Mutual Exclusion algorithm

For this well known example [26][4] we verified the mutual exclusion property. The results for different numbers N of processes are shown in Table 1.

The gain due to the symmetry reduction is obvious. The obtained reductions, ranging from 49% (for $N = 2$) to 99% and more (for $N \ge 5$) are close to the theoretical maxima, which can be explained with the high degree

[4] In our implementation the global predicate that guards the entry in the critical section is checked atomically. As this guard ranges over all process indices, the atomicity was necessary due to the restrictions on statements that can be used such that the state space symmetry is preserved.

Table 1. Results for Peterson's mutual exclusion algorithm.

N		2		3		4		5		6		7	
		+POR	-POR	+POR	-POR	+POR	-POR	+POR	-POR	+POR	-POR	+POR	-POR
no sym.	s	154	263	4992	11318	202673	542921	9.1M	*o.m.*	*o.m.*	*o.m.*	*o.m.*	*o.m.*
	t	6.5	5.6	6.8	6.3	25.0	1:09	19:04	—	—	—	—	—
	s	89	134	980	1976	9634	24383	86112	262749	700543	2.5M[a]	5.3M[a]	*o.m.*
full	t	6.5	5.6	6.7	5.9	10.4	19.3	2:35	12:02	2:04:03	*o.t.*	*o.t.*	—
seg.	t	6.7	5.6	6.6	5.8	8.5	12.4	38.0	2:31	7:36	44:40	1:56:00	—
pc-seg.	t	6.5	5.6	6.6	5.8	8.4	11.3	37.4	1:59	10:01	40:46	4:26:32	—
	s	113	160	1877	3967	35644	88489	595843	1.8M	8.8M	*o.m.*	*o.m.*	*o.m.*
sorted	t	6.5	5.6	6.6	6.1	11.0	21.6	1:42	8:13	28:37	—	—	—
	s	92	137	1149	2396	20339	46804	401423	942786	9.6M	*o.m.*	*o.m.*	*o.m.*
pc-sort.	t	6.5	5.6	6.6	5.8	10.2	15.0	1:36	4:53	54:53	—	—	—

[a] The number of states is obtained with segmented and pc-segmented.

of symmetry in the protocol. The verification times are also better, even with the straightforward full strategy, due to the smaller number of states that are generated during the search. We used both symmetry and partial order reduction (which is default in standard Spin), separately and in combination. Standard Spin with partial order reduction could not handle more than 5 processes. However, the segmented versions of the symmetry heuristics alone were sufficient for $N = 6$. For $N = 7$ we had to use a combination of both reduction techniques in order to stay inside the available memory.

The sorted strategies are comparable with their segmented counterparts only for small values of N. For greater values they deteriorate and even the possible gain in time over the segmented versions disappears because of the greater number of states that have to be explored.

One can also expect that as N increases pc-segmented and pc-sorted will loose the advantage they have over segmented and sorted, for smaller values of N. The reason is that the number of different elements in the main array of segmented and sorted increases as N increases, while the number of values of the pc counter stays the same. (We use as a main array the array of flags. Whenever process i enters the competition for the critical section, it sets flag i to a value between 1 and $N-1$. The default value is 0. Notice that the values of the flag array are not of the scalarset type I, although their range is the same as the range of I.) Intuitively, the greater versatility of the values in the main array, the fewer permutations have to be generated on average, in order to canonicalize the state. This tendency is already visible for $N = 6$ (for which sorted is winning over pc-sorted) as well as for $N = 7$ (for which segmented is better than pc-segmented).

5.2 Data Base Manager

In this example, borrowed from [28], one considers a system that consists of $N \geq 2$ data base managers, which modify a data base and exchange messages to ensure the consistency of the data base contents. Our model deals with the procedural part of the protocol, i.e., with the message exchange, by abstracting from the actual modification of the data base. Initially all managers are in inactive state until one of them modifies the data base. This manager in one atomic step reserves the data base for itself and sends a message to every other manager. After that it waits for acknowledgments from the other managers. All other managers concurrently perform a two step sequence: reception of the message, and sending of an acknowledgment. When all acknowledgments are available, the manager who initially modified the data base and started the whole procedure, reads them. At the same moment it also releases the data base so that it can be modified by the other managers, after which it returns to inactive state. We checked the model for absence of deadlock. The results are given in Table 2.

Our experiments are in accord with the theoretically predicted results from [28]: both symmetry and POR applied separately reduce the exponential growth of the state space to quadratic, and in combination they give linear growth. Unlike in the previous example, this time pc-segmented strategy is a clear winner over segmented. The explanation is again in the diversity of the elements occurring in the sorted array – the only possible values of the main array for segmented are 0 and 1, while the pc counter has seven different values. (The main array consists of boolean flags which are set to 1 when a manager reserves the data base and sends a message to all the other processes.) It is interesting that pc-sorted in combination with POR is by far the most successful strategy. It achieves the same reduction as the canonical

Table 2. Results for the Data Base Manager example.

N		7		8		9		10		11		12	
		+POR	-POR	+POR	-POR	+POR	-POR	+POR	-POR	+POR	-POR	+POR	-POR
no sym.	s	100	5112	130	17506	164	59060	202	196842	244	649552	290	2.1M
	t	0.0	0.5	0.0	1.9	0.0	8.2	0.0	32.1	0.0	2:11	0.0	8:27
	s	16	31	18	39	20	48	22	58	24	69	—	—
full	t	0.5	2.0	4.6	24.5	48.0	6:16	9:08	1:27:40	1:55:09	22:07:36	*o.t.*	*o.t.*
seg.	t	0.3	0.4	2.8	3.8	28.3	39.1	5:22	7:20	1:06:32	1:30:53	*o.t.*	*o.t.*
pc-seg.	t	0.1	0.1	0.8	1.0	7.3	9.1	1:16	1:33	15:16	17:14	*o.t.*	*o.t.*
sorted	s	27	250	31	505	35	1016	39	2039	43	4086	47	8181
	t	0.0	0.0	0.0	0.1	0.0	0.2	0.0	0.6	0.0	1.3	0.0	3.1
pc-sort.	s	16	58	18	91	20	155	22	245	24	415	26	659
	t	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.1	0.0	0.3

strategies, but within much shorter time. It remains to be seen whether this is just a peculiarity of this model, or it occurs for a wider class of examples.

5.3 The Initialization Protocol

Assume that we are given a system of n processes each consisting of an initial part and a main part. The data structures can have arbitrary initial values. The role of the Initialization Protocol from [17] is to synchronize the processes such that none of them can enter its main part before all the others have finished their initial parts.[5] Process synchronization is done via shared boolean variables. Each process has so called co-component process whose task is to manage the synchronization of these boolean variables. We refer the reader to [17] for more details.

The protocol is an example of a system with one scalar variable and two process families indexed by this scalar – the family of processes that are synchronized and the family of their co-components. In such a case one can expect the same maximal amount of reduction of $n!$ (where n is the family size) as with one family.

We verified the correctness of the algorithm for several values of n by placing before the main part of each process an assertion which checks that all the other processes have terminated their initial parts. The results are given in Table 3.

Because of the intensive use of global variables, partial order reduction had almost no effect in this example. Thus, the state space reduction was mostly due to the symmetry. The canonical heuristic segmented and pc-segmented prevailed over the non-canonical ones regarding both space and time. Compared to full, they were significantly faster for larger values of n. Better performance of pc-segmented and pc-sorted over segmented and sorted respectively, can be explained by the fact that the main array that was used in the latter strategies is of boolean type. This means that the elements of the main array can have only two values, in contrast with the program counter which ranges over at least 15 different values. Intuitively, the greater versatility of values in the pc array means that fewer permutations have to be generated on average in order to canonicalize the state.

[5] Note that this would be trivial if we could assume that the variables' initial values were known.

5.4 Base Station

This example is a simplified version of MASCARA – a telecommunication protocol developed by the WAND (Wireless ATM Network Demonstrator) consortium [9]. The protocol is an extension of the ATM (Asynchronous Transfer Mode) networking protocol to the wireless domain. We present the results for two models of the protocol.

Our first model represents a wireless network connecting $N \geq 2$ mobile stations (MS) that may communicate with each other, using a limited number ($M \geq 1$) of radio channels provided by one base station BS. More specifically, when MS A wants to send a message to MS B it must request a channel from BS. Provided there are channels available, A is granted one, call it c. If B wants to receive messages, it queries BS. As there is a pending communication for B through c, BS assigns c to B. After the communication has taken place, both A and B return the channel to BS. The results given in Table 4 are for checking for unreachable code, with $M = 2$.

For $N = 2$ the symmetry reduction approaches the theoretical limit of 50%. For $N = 3$ standard Spin ran out of memory, while with symmetry reduction it was possible to verify the model. In this example POR did not play a significant role. Also, there was no clear preference between segmented, pc-segmented and sorted.

In the second model, instead of having N stations which can both receive and send messages, we consider a network connecting $n_s \geq 1$ sending mobile stations and $n_r \geq 1$ receiving mobile stations. Another important

Table 3. Results for the Initialization Protocol example.

n		2		3		4		5	
		+POR	-POR	+POR	-POR	+POR	-POR	+POR	-POR
no sym.	s	2578	2578	131484	131484	7.0M	7.0M	*o.m.*	*o.m.*
	t	6.4	1.1	19.3	12.2	18:50	16:31	—	—
	s	1520	1521	26389	26399	386457	386536	5.3M	*o.m.*
full	t	6.2	1.0	11.4	6.0	4:53	4:46	5:38:05	—
seg.	t	6.3	1.0	11.1	5.4	3:19	3:04	2:28:20	—
pc-seg.	t	6.3	0.9	10.1	4.4	1:54	1:39	47:03	—
sorted	s	2344	2344	98602	98602	4.2M	4.2M	*o.m.*	*o.m.*
	t	6.3	1.0	17.9	11.0	12:13	12:22	—	—
pc-sort.	s	1607	1607	40395	40395	987830	987830	*o.m.*	*o.m.*
	t	6.3	0.1	11.8	5.7	3:55	3:17	—	—

Table 4. Results for the Base Station example (first model).

N		2		3	
		+POR	-POR	+POR	-POR
no sym.	s	15613	15621	*o.m.*	*o.m.*
	t	7.6	6.4	—	—
	s	7808	7812	3.4M	3.4M
full	t	7.5	6.3	17:39	13:00
seg.	t	5.8	6.3	12:10	9:46
pc-seg.	t	5.9	6.4	13:24	10:18
sorted	s	7856	7860	3.9M	3.9M
	t	7.2	6.3	12:58	9:48
pc-sort.	s	8282	8286	5.1M	5.1M
	t	7.5	6.3	16:48	12:57

Table 5. Results for the Base Station example (second model).

		$n_s = 2$ $n_r = 2$		$n_s = 3$ $n_r = 2$		$n_s = 2$ $n_r = 3$	
		+POR	-POR	+POR	-POR	+POR	-POR
no sym.	s	4.6M	9.0M	*o.m.*	*o.m.*	*o.m.*	*o.m.*
	t	5:02	13:48	—	—	—	—
	s	1.2M	2.3M	9.0M	*o.m.*	9.4M	*o.m.*
full	t	1:41	5:03	28:57	—	30:32	—
pc-seg.	t	1:32	4:54	19:29	—	20:36	—
pc-sort.	s	1.7M	3.5M	*o.m.*	*o.m.*	*o.m.*	*o.m.*
	t	1:58	6:03	39:02	—	—	—

difference is that the communication is modeled with Promela channels, instead of global variables. For the rest the description of the protocol given for the first model applies also to the second one.

Thus, there are two families of symmetric processes in the second model: the family of sending stations and the family of receiving stations. Unlike in the Initialization Protocol, the two families are independent, and, consequently, indexed with two different scalar sets.

On this example we could apply only pc-segmented and pc-sorted because there was no main array that could be used for sorted and segmented. One can see that for the canonical strategies the reduction indeed approaches the theoretical limit $n_s! \cdot n_r!$. (In general, for a system with k independent process families indexed by k scalar sets, the reduction due to symmetry is limited by $n_1! \cdot n_2! \cdot \ldots \cdot n_k!$, where $n_i, 1 \leq i \leq k$, is the number of processes in the i-th family.)

With the usage of Promela channels instead of global variables, the contribution of the partial order reduction became significant (Table 5), which was not at all the case in the first model. One can also notice the orthogonality of the two heuristics – namely, the factor of reduction due to symmetry was almost the same with or without POR.

6 Conclusions and Future Work

We have presented SymmSpin, an extension of Spin with a symmetry-reduction package. The package is based on a heuristic for solving instances of the orbit problem. It provides the four reduction strategies based on this heuristic (sorted/pc-sorted and segmented/pc-segmented) as well as the full reduction strategy. Our implementation deals with almost all Promela features, including queues, multiple scalar sets, and multiple process families. The only remaining restrictions are that the elements of queues are as yet restricted to simple, non-structured types, and that no dynamic process creation is allowed.

The resulting package has been described at a detailed level. For maximal modularity, the implementation is in the form of a Tcl script that operates on the verification engine produced by Spin. The full source code is available from the authors.

We performed an extensive series of experiments. The results show that there is no uniform winner: It depends on the program which of the strategies performs best.

In some cases, this was in fact predicted by the form of the program. Thus, it make sense to have all strategies (maybe except full) as separate options of the extended model-checker.

The most natural next step would be to extend the Promela syntax to allow specifications of scalarsets directly instead of via an accompanying file. This will facilitate the automatic verification of the syntactic conditions along the lines of [23], that are needed to ensure the soundness of the analysis. With the present implementation this is still a task for the user. A more ambitious attempt would be the automatic detection of scalarsets directly from the Promela sources.

Another extension is to also allow the verification of (symmetric) liveness properties. It can be shown [1] that the nested depth-first search algorithm of [8], which is used in Spin for cycle detection, remains correct under the symmetry reduction. This implies that we can go beyond safety properties, or more precisely, the full class of ω-regular correctness properties can be handled by SymmSpin.However, as also shown in [1], Spin's algorithmic notion of process fairness, which is usually needed in the verification of liveness aspects, is not compatible with symmetry reduction. Hence, this is a topic for future research.

References

1. D. Bošnački, Nested Depth First Search Algorithms for Symmetry Reduction in Model Checking, in *Enhancing State Space Reduction Techniques for Model Checking*, Ph.D. Thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, 2001.
2. D. Bošnački, D. Dams, Integrating real time into Spin: a prototype implementation, in S. Budkowski, A. Cavalli, E. Najm (eds), *Proc. of FORTE/PSTV'98 (Formal Description Techniques and Protocol Specification, Testing and Verification)*, 423–438, Paris, France, Oct. 1998.
3. D. Bošnački, D. Dams, L. Holenderski, A Heuristic for Symmetry Reductions with Scalarsets, *Proc. of FME'2001 (Fomal Methods Europe)*, LNCS 2021, 518–533, Springer, 2001.
4. D. Bošnački, D. Dams, L. Holenderski, Symmetric Spin, *Proc. of SPIN'2000 (The 7th International SPIN Workshop on Model Checking of Software), LNCS 1885, 1–19, Springer, 2000.*
5. *D. Bošnački, D. Dams, L. Holenderski, N. Sidorova, Model checking SDL with Spin,* Proc. of TACAS'2000 (Tools and Algorithms for the Construction and Analysis of Systems), *LNCS 1785, 363–377, Springer, 2000.*
6. E.M. Clarke, R. Enders, T. Filkorn, S. Jha, Exploiting symmetry in temporal logic model checking, *Formal Methods in System Design*, Vol. 19, 77–104, 1996.
7. E.M. Clarke, O. Grumberg, D.A. Peled, *Model Checking*, The MIT Press, 2000.
8. C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, Memory efficient algorithm for the verification of temporal properties, *Formal Methods in System Design I*, 275-288, 1992.
9. I. Dravapoulos, N. Pronios, S. Denazis *et al*, *The Magic WAND, Deliverable 3D2, Wireless ATM MAC*, Sep 1997.
10. E.A. Emerson, Temporal and modal logic, in Jan van Leeuwen (ed.), *Formal Models and Semantic*, Vol. B of *Handbook of Theoretical Computer Science*, Chap. 16, 995–1072, Elsevier/The MIT Press, 1990.
11. E.A. Emerson, J. Havlicek, R.J. Trefler, Virtual Symmetry Reduction, in *Proc. of 15th Annual IEEE Symposium on Logic in Computer Science (LICS'00)*, 121–131, Springer, 2000.
12. E.A. Emerson, S. Jha, D. Peled, Combining partial order and symmetry reductions, in Ed Brinksma (ed.), *Proc. of TACAS'97 (Tools and Algorithms for the Construction and Analysis of Systems)*, LNCS 1217, 19–34, Springer, 1997.
13. E.A. Emerson, A.P. Sistla, Symmetry and model checking, in C. Courcoubetis (ed.), *Proc. of CAV'93 (Computer Aided Verification)*, LNCS 697, 463–478, Springer, 1993.
14. E.A. Emerson, R.J. Trefler, Model checking real-time properties of symmetric systems, *Proc. of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 427–436, Aug. 1998.
15. E.A. Emerson, R.J. Trefler, From asymmetry to full symmetry: new techniques for symmetry reduction in model checking, *Proc. of CHARME'99 (The 10th IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods)*, Bad Herrenalb, Germany, Sep. 1999.
16. E.A. Emerson, A.P. Sistla, Utilizing symmetry when model-checking under fairness assumptions: an automata-theoretic approach, *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, July 1997.
17. W.H.J. Feijen, A.J.M. van Gasteren, *On a method of multiprogramming*, Springer-Verlag, 1999.
18. P. Godefroid, Exploiting symmetry when model-checking software, *Proc. of FORTE/PSTV'99 (Formal Methods for Protocol Engineering and Distributed Systems)*, 257–275, Beijing, Oct. 1999.
19. P. Godefroid, A.P. Sistla, Symmetry and Reduced Symmetry in Model Checking, in G. Berry, H. Comon, A. Finkel (eds.), *Proc. of CAV'2001 (13th Conference on Computer Aided Verification)*, LNCS 2102, 91–103, Springer, 2001.
20. V. Gyuris, A.P. Sistla, On-the-fly model checking under fairness that exploits symmetry, in O. Grumberg (ed.), *Proc. of CAV'97 (Computer Aided Verification)*, LNCS 1254, 232–243, Springer, 1997.
21. G.J. Holzmann, *Design and Validation of Communication Protocols*, Prentice Hall, 1991. Also: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`
22. C.N. Ip, D.L. Dill, Better verification through symmetry, in D. Agnew, L. Claesen, R. Camposano (eds), *Proc. of the 1993 Conference on Computer Hardware Description Languages and their Applications*, Apr. 1993.
23. C.N. Ip, D.L. Dill, Better verification through symmetry. *Formal Methods in System Design*, Vol. 9, 41–75, 1996.
24. C.N. Ip, *State Reduction Methods for Automatic Formal Verification*, PhD thesis, Department of Computer Science of Stanford University, Dec 1996.

25. V. Lipskiy, Kombinatorika dlya programmistov, Mir, Moscow, 1988. (In Russian)
26. N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996.
27. R. Nalumasu, G. Gopalakrishnan, Explicit-enumeration based Verification made Memory-efficient, *Proc. of CHDL'95 (Computer Hardware Description Languages)*, 617-622, Chiba, Japan, Aug. 1995.
28. A. Valmari, Stubborn sets for reduced state space generation, *Advances in Petri Nets 1990*, LNCS 483, 491–515, Springer, 1991.