# A Heuristic for Symmetry Reductions with Scalarsets*

Dragan Bošnački[1], Dennis Dams[2], and Leszek Holenderski[1]

[1] Dept. of Computing Sci., Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
[2] Dept. of Electrical Eng., Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands

`{D.Bosnacki,D.Dams,L.Holenderski}@tue.nl`

**Abstract.** We present four versions of a new heuristic for coping with the problem of finding (canonical) representatives of symmetry equivalence classes (the so-called *orbit problem*), in symmetry techniques for model checking. The practical implementation of such techniques hinges on appropriate workarounds of this hard problem, which is equivalent to graph isomorphism. We implemented the four strategies on top of the Spin model checker, and compared their performance on several examples, with encouraging results.

## 1 Introduction

One way to combat the state explosion problem in model checking [5, 8] 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 state-space exploration, that two states are equivalent. To start with the first issue: as in any other state-space reduction method 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

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

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 [4] 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 refer to the bibliography of [19]. Emerson and Sistla have applied the idea to CTL model checking in [10], with extensions to fairness in [13] and [15]. In [11], Emerson and Trefler extended the concepts to real-time logics, while in [12] they considered systems that are *almost* symmetric. Clarke, Enders, Filkorn, and Jha used symmetries in the context of symbolic model checking in [4]. Emerson, Jha, and Peled, and more recently Godefroid, have studied the combination of partial order and symmetry reductions, see [9, 14]. Our work draws upon the ideas of Dill and Ip [17–19]. They introduce, in the protocol description language Mur$\varphi$, 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 [17]:

> 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 processed together in an array. The resulting strategies are called *pc-sorted* and *pc-segmented.*

In order to compare these 5 strategies, we have implemented them on top of the Spin model checker [16]. Building upon results reported in [9], 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. A more detailed description of the implementation can be found in [3]. In this paper we concentrate more on the theoretical aspects and the methodology.

We are aware of only one other attempt to extend Spin with symmetry reductions [21]. With their implementation the user himself (herself) 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 might require quite a deep knowledge of Spin's internal workings.

Our current implementation preserves all safety properties. A large number of experiments have been conducted that show good results (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 $s \rightarrow s'$.

A bijection $h : \mathcal{S} \rightarrow \mathcal{S}$ is said to be a *symmetry* of $T$ iff $h(s_0) = s_0$, $h(e) = e$, and for any $s, s' \in \mathcal{S}$, $s \rightarrow s'$ iff $h(s) \rightarrow h(s')$. 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 [18]), 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 [18]). 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 principle, it is not difficult to extend an enumerative model checker to handle symmetries, i.e., to explore $T/\sim_A$ instead of $T$. Provided with a set $A$ of symmetries and a function $rep : \mathcal{S} \to \mathcal{S}$ which for a given state $s$ picks up a representative state from $[s]_A$, 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*.

When choosing a reduction strategy one has to deal with two contradictory requirements: *rep* should both lead to a substantial reduction of the explored state space and be computationally inexpensive. Obviously, the best reduction of the state space is obtained if *rep* is *canonical* (i.e., $rep(s) = rep(s')$ whenever $s \sim_A s'$) since then the explored part of $T$ is isomorphic with $T/\sim_A$. On the other hand, a canonical *rep* may be computationally costly and the time overhead caused by using it can substantially exceed the time savings gained by exploring a smaller state space.

In Section 3 we formally define several reduction strategies and then, in Section 5, compare their performance on several examples.

## 3 Reduction Strategies

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 $I \mapsto E$ of types where $I$ is the type of indices and $E$ is the type of elements. $I$ must be simple while $E$ can be any type that does not depend on $I \mapsto E$. Let $\mathcal{D}_T$ denote the set of all values of type $T$ then a value of an array type $I \mapsto E$ is a function $a : \mathcal{D}_I \to \mathcal{D}_E$. We write $a[i]$ instead of $a(i)$.

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 *Pid* 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 of processes $C = \lambda\, i : \mathit{Pid}.C_i$. Notice that $\mathit{Pid}$ can be used as other simple types, for example, to index arrays.

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 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 $p : \mathit{Pid} \to \mathit{Pid}$ can be lifted to states, via $p^* : \mathcal{S} \to \mathcal{S}$. Intuitively, $p^*$ applies $p$ to all values of type $\mathit{Pid}$, including the pids used as array indices. Formally, for $s = (s_{\mathcal{V}}, s_{PC})$, $p^*(s) = (\lambda\, v.\overline{p}_{type(v)}(s_{\mathcal{V}}(v)), s_{PC})$ where $\overline{p}_T : \mathcal{D}_T \to \mathcal{D}_T$ is defined as

$$\begin{aligned}
\overline{p}_T(d) &= d \qquad \text{if } T \text{ is a simple type other than } \mathit{Pid}\\
\overline{p}_{\mathit{Pid}}(d) &= p(d)\\
\overline{p}_{I\mapsto E}(a) &= \lambda\, i \in \mathcal{D}_I.\, \overline{p}_E(a[\overline{p}_I(i)])
\end{aligned}$$

In fact, $p^*$ is a symmetry of $T_P$ (Theorem 3 in [18]). 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 $\mathit{rep} : \mathcal{S} \to \mathcal{S}$ defined as

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

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 $\pi$ and $\mu$. A *concrete* reduction strategy is obtained from *rep* by fixing some $\pi$, and is denoted by $\mathit{rep}_\pi$. In the sequel, we consider several concrete strategies which we call full, sorted, segmented, pc-sorted and pc-segmented. The names denote the respective $\pi$ functions in $\mathit{rep}_\pi$, and whenever we say "strategy $\pi$" we really mean "strategy $\mathit{rep}_\pi$".

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 $\mu$, 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 the 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] 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).

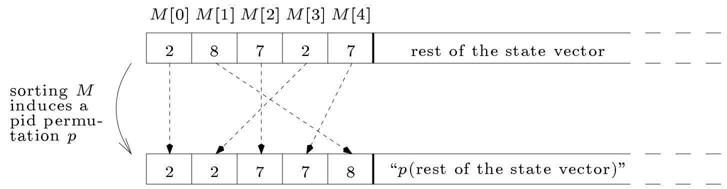
Second, we assume that program $P$ uses a variable $M : Pid \mapsto E$ which is an array indexed by pids and whose elements do not involve pids. $M$ is called a *main array*. Most real protocols specified in a language based on shared variables, and of the form $B||C_0|| \cdots ||C_{n-1}$, use such an array, either directly or indirectly. (Notice that each local variable declared in the parametric process $C = \lambda i : Pid.C_i$, and whose type does not involve $Pid$, can be considered an element of an array indexed by $Pid$, when lifted to the global level.) We further assume that $M$ dominates in the total order used by our particular choice function $\mu$, 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 $E$, due to our particular choice of $\mu$. So instead of considering all pid permutations in $\mathsf{full}$ 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 $\mathsf{sorted}$ 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 $rep_{\mathsf{sorted}}$ strategy is faster then $rep_{\mathsf{full}}$. 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 $\mathsf{segmented}$ 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 the following picture showing a state vector before and after sorting a main array $M$.




[1] This is not inconsistent with the assumption that $Pid$ 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).

[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 all the segments independently.

$M$ 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 $Pid$. Suppose that the particular sorting algorithm being used sorts $M$ as indicated by the dashed arrows. This particular sorting induces the pid permutation $p = \{0 \mapsto 0, 3 \mapsto 1, 2 \mapsto 2, 4 \mapsto 3, 1 \mapsto 4\}$, 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 $p' = \{3 \mapsto 0, 0 \mapsto 1, 4 \mapsto 2, 2 \mapsto 3, 1 \mapsto 4\}$, 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 (see Theorem 1 below), so they are equally good, as far as the reduction of a state space is considered.

The following result allows us to drop the assumption about our particular $\mu$ function.

**Theorem 1.** *For any choice function $\mu$, $rep_{\mathsf{segmented}}$ 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 $\mu$, 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 $p_1$ that sorts the array $M$. Observe that $s = (p^{-1})^*(s')$ where $p^{-1}$ is the inverse of $p$. Hence, $s_1 = p_1^*((p^{-1})^*(s'))$. Since $p_1^* \circ (p^{-1})^*$ also sorts the array $M$, $s_1 \in P(s')$. 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 $\pi$ 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 practical consequence. 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 $\mu$ 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 $\mu$ 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 $\mu$ is not needed. Also, the array of program counters, used in pc-sorted and pc-segmented, need not even be be contiguous. As a consequence, we do not need to reshuffle a state vector in order to use the presented strategies.

Finally, the assumption about the particular form of programs as $B||C_0||\cdots||C_{n-1}$, and the special role the scalarset *Pid* plays in indexing the family $C = \lambda i : Pid.C_i$, is only needed to formalize the pc-sorted and pc-segmented strategies. For segmented and sorted, the only essential assumption about *Pid* is that it is a distinguished scalarset.

In Section 5 we compare, on several examples, the performance of all the concrete strategies introduced above.

## 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 [16]. 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 $T = (\mathcal{S}, s_0, \rightarrow)$, as depicted in Fig. 1, one explores the quotient $T/\!\sim$ using a simple modification of the algorithm, as depicted in Fig. 2. (This modification is borrowed from [18].)

```
reached := unexpanded := {s0};
while unexpanded ≠ ∅ do
    remove a state s from unexpanded;
    for each transition s → s' do
        if s' = error then
            stop and report error;
        if s' ∉ reached then
            add s' to reached and unexpanded;
```

**Fig. 1.** A standard exploration algorithm

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. 2. In

```
reached := unexpanded := {rep(s_0)};
while unexpanded ≠ ∅ do
    remove a state s from unexpanded;
    for each transition s → s' do
        if s' = error then
            stop and report error;
        if rep(s') ∉ reached then
            add rep(s') to reached and unexpanded;
```

**Fig. 2.** A standard exploration algorithm with symmetry reductions

our[3] algorithm, the original states, and not their representatives, are used to generate the state space to be explored, as depicted in Fig. 3.

```
reached := {rep(s_0)}; unexpanded := {s_0};
while unexpanded ≠ ∅ do
    remove a state s from unexpanded;
    for each transition s → s' do
        if s' = error then
            stop and report error;
        if rep(s') ∉ reached then
            add rep(s') to reached and s' to unexpanded;
```

**Fig. 3.** Our exploration algorithm with symmetry reductions

Obviously, our algorithm is still sound. Moreover, it allows to easily regenerate an erroneous trace from the set *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 directly be dumped as the erroneous trace. In the algorithm from Fig. 2, 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.

Notice that although both algorithms explore the same state space (since they agree on the *reached* set), there could still be a difference in their execution times, if the numbers of the successor states $s'$ considered in the loop "for each transition $s \to s'$ do" were different. It can easily be proved that this is not a case. Whenever our algorithm computes the successors of $s$, the algorithm in Fig. 2 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).

We also put an effort into efficiently implementing the 5 variants of the `rep` function. For example, it turns out that all the sets of pid permutations used

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

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 [3].

In the verification experiments described in the next section, 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 [9] 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.

It can be shown [1] that the nested depth-first search algorithm of [6], which is used in Spin for cycle detection, remains correct with the symmetry reduction. This implies that we can go beyond safety properties, or more precisely, the full class of $\omega$-regular correctness properties can be handled.

## 5 Experimental Results

We tried our prototype implementation on several examples. 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 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 three of the examples. All experiments were performed on a Sun Ultra-Enterprise machine, with three 248 MHz UltraSPARC-II processors and 2304MB of main memory, running the SunOS 5.5.1 operating system. Verification times are given in seconds ($s.x$), minutes ($m$:$s$), or hours ($h$:$m$:$s$); the number of states 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.

**Peterson's Mutual Exclusion algorithm [20].** For this well known example[4] we verified the mutual exclusion property. The results for different numbers $N$ of processes are shown below.

[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 |
| | states | 154 | 263 | 4992 | 11318 | 202673 | 542921 | 9.1M | *o.m.* | *o.m.* | *o.m.* | *o.m.* | *o.m.* |
| no sym. | time | 6.5 | 5.6 | 6.8 | 6.3 | 25.0 | 1:09 | 19:04 | — | — | — | — | — |
| | states | 89 | 134 | 980 | 1976 | 9634 | 24383 | 86112 | 262749 | 700543 | 2.5M[a] | 5.3M[a] | *o.m.* |
| full | time | 6.5 | 5.6 | 6.7 | 5.9 | 10.4 | 19.3 | 2:35 | 12:02 | 2:04:03 | *o.t.* | *o.t.* | — |
| seg. | time | 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. | time | 6.5 | 5.6 | 6.6 | 5.8 | 8.4 | 11.3 | 37.4 | 1:59 | 10:01 | 40:46 | 4:26:32 | — |
| | states | 113 | 160 | 1877 | 3967 | 35644 | 88489 | 595843 | 1.8M | 8.8M | *o.m.* | *o.m.* | *o.m.* |
| sorted | time | 6.5 | 5.6 | 6.6 | 6.1 | 11.0 | 21.6 | 1:42 | 8:13 | 28:37 | — | — | — |
| | states | 92 | 137 | 1149 | 2396 | 20339 | 46804 | 401423 | 942786 | 9.6M | *o.m.* | *o.m.* | *o.m.* |
| pc-sorted | time | 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.

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

**Data Base Manager [22].** The system that we consider 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 form 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.

**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 |
| | states | 100 | 5112 | 130 | 17506 | 164 | 59060 | 202 | 196842 | 244 | 649552 | 290 | 2.1M |
| no sym. | time | 0.0 | 0.5 | 0.0 | 1.9 | 0.0 | 8.2 | 0.0 | 32.1 | 0.0 | 2:11 | 0.0 | 8:27 |
| | states | 16 | 31 | 18 | 39 | 20 | 48 | 22 | 58 | 24 | 69 | — | — |
| full | time | 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. | time | 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. | time | 0.1 | 0.1 | 0.8 | 1.0 | 7.3 | 9.1 | 1:16 | 1:33 | 15:16 | 17:14 | *o.t.* | *o.t.* |
| | states | 27 | 250 | 31 | 505 | 35 | 1016 | 39 | 2039 | 43 | 4086 | 47 | 8181 |
| sorted | time | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.2 | 0.0 | 0.6 | 0.0 | 1.3 | 0.0 | 3.1 |
| | states | 16 | 58 | 18 | 91 | 20 | 155 | 22 | 245 | 24 | 415 | 26 | 659 |
| pc-sorted | time | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.1 | 0.0 | 0.3 |

Our experiments are in accord with the theoretically predicted results from [22]: 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 occuring in the sorted array – the only possible values of the main array for segmented are 0 and 1, while the pc counter has 7 different values. 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 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.

**Base Station.** The third example is a simplified version of MASCARA – a telecommunication protocol developed by the WAND (Wireless ATM Network Demonstrator) consortium [7]. The protocol is an extension of the ATM (Asynchronous Transfer Mode) networking protocol to wireless networks. Our 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 below are for checking for unreachable code, with $M = 2$.

**Table 3.** Results for the Base Station example.

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

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.

## 6 Conclusions and Future Work

We have presented four variants of a new heuristic for coping with the orbit problem in symmetry reduction. Their strength is based on the observation that the lexicographical ordering on state vectors may be tuned to optimize the splitting heuristic of [17]. Of the resulting strategies, the segmented/pc-segmented versions indeed produce canonical representatives of orbits, while sorted/pc-sorted are only normalizing. The sorted and segmented strategies presuppose the presence of a main array in the program. Their pc variants exploit the array of program counters in programs containing a family of similar processes.

We have implemented these strategies in Spin, as well as the full reference strategy, and compared their effects on three programs. The experimental 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 results obtained with this experimental implementation are sufficiently encouraging in order to integrate the symmetry strategies more completely into the Spin tool. One concern here is to ensure the compatibility of the symmetry reduction with Spin's other features. So far, we have focussed on the partial order reduction. Our recent results [1] show that the correctness of Spin's cycle detection algorithm, which lies at the heart of its capability to handle the full class of $\omega$-regular correctness properties, is preserved. Furthermore, the symmetry is compatible with the recent extension to timed systems, DTSpin ([2]). It should be added that the present implementation is able to deal with multiple scalar sets and multiple process families. Also, almost all Promela features are handled, including queues. For the latter there is still the restriction though that the the queue elements have to be simple, non-structured types.

Also, the restrictions on the scalarset variables are as yet not automatically checked. Next, we plan to extend the symmetry algorithm itself with other system topologies, like rings[5]. An interesting question is whether the choice of an appropriate strategy can be automated, based on syntactic clues.

## References


1. D. Bošnački, *Enhancing State Space Reduction Techniques for Model Checking*, Ph.D. Thesis, Department of Computer Science, Eindhoven University of Technology, expected in 2000
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, 1998.
3. D. Bošnački, D. Dams, L. Holenderski, Symmetric Spin, *SPIN'2000 (The 7th International SPIN Workshop on Model Checking of Software)*, to appear in LNCS, 2000.
4. 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.
5. E.M. Clarke, O. Grumberg, D.A. Peled, *Model Checking*, The MIT Press, 2000.
6. C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, Memory efficient algorithm for the verification of temporal properties, *Formal Methods in System Design I*, pp. 275-288, 1992


[5] The group of *shift* permutations can be handled by a simplification of our approach, where instead of sorting the main array, it is cyclically shifted until it starts with a minimal element.

7. I. Dravapoulos, N. Pronios, S. Denazis *et al*, *The Magic WAND, Deliverable 3D2, Wireless ATM MAC*, Sep 1997.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. G.J. Holzmann, *Design and Validation of Communication Protocols*, Prentice Hall, 1991. Also: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`
17. 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.
18. C.N. Ip, D.L. Dill, Better verification through symmetry. *Formal Methods in System Design*, Vol. 9, 41–75, 1996.
19. C.N. Ip, *State Reduction Methods for Automatic Formal Verification*, Ph.D. thesis, Department of Computer Science of Stanford University, Dec 1996.
20. N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996.
21. 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.
22. A. Valmari, Stubborn sets for reduced state space generation, *Advances in Petri Nets 1990*, LNCS 483, 491–515, Springer, 1991.