Symmetric Spin*

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 give a detailed description of SymmSpin, a symmetry-reduction package for Spin. It offers four strategies for state-space reduction, based on the heuristic that we presented in [3], and a fifth mode for reference. A series of new experiments is described, underlining the effectiveness of the heuristic and demonstrating the generalisation of the implementation to multiple scalar sets, multiple process families, as well as almost the full Promela language.

1 Introduction

One way to combat the state explosion problem in model checking [5, 7] 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 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

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

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 [9], with extensions to fairness in [12] and [15]. In [10], Emerson and Trefler extended the concepts to real-time logics, while in [11] 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 [8, 14].

Our work draws upon the ideas of Ip and Dill [17–19]. 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.

Following up on the approach of Ip and Dill, we have presented in [3] a new heuristic for finding representatives of symmetry equivalence classes. In Section 3, this heuristic and the four strategies based on it are summarised. This is done by redeveloping them on the basis of a generalisation of the notion of lexicographical ordering. By introducing *partial weight functions*, lexicographical orderings are generalised to partial orderings that are not necessarily total, which allows to uniformly capture both the canonical and the non-canonical strategies.

[3] also reports on a number of experiments obtained with a prototype implementation on top of the Spin tool. Based on the results of those experiments, it was deemed worthwhile to extend the implementation so as to provide a more complete integration into Spin. Presently, this extension has been carried out. Compared to the prototype implementation, the current symmetry package has been generalised in a number of essential directions. The main improvements are its capabilities to handle *queues*, *multiple scalar sets*, as well as *multiple process families*. One of the current paper's contributions is to describe the overall implementation in the context of Spin (Section 4). Having implemented this symmetry package, we were also able to run more and larger experiments. The results of these, and their evaluation, form the second main contribution of this paper (Section 5). In the sequel we assume that we are dealing with safety properties (see also Section 6).

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 order to extend an enumerative model checker to handle symmetries, i.e., to explore $T/\sim_A$ instead of T, a practical representation for equivalence classes is needed. A common approach is to use a *representative function*, which is a function $rep : \mathcal{S} \rightarrow \mathcal{S}$ 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 [18]. 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

We start with a summary of the strategies from [3].

In the approach of [17], the definition of representatives is based on a lexicographic ordering on state vectors. Normally this is a total ordering, but by defining it relative to a splitting of the state vector in two parts, a hierarchy of non-total orderings is obtained, parameterized by the point at which the vector is split. More precisely, the representatives of a class are those state vectors whose leftmost part (relative to the splitting) is the lexicographical minimum among the leftmost parts of all the vectors in that class. The trade-off between speed and precision may now be tuned by varying the split point. This is a degree of freedom that allows to explore a variety of representative functions, including the two extremes *id* (split point at the beginning of the state vector) and $\lambda\, s \in \mathcal{S}.\, c_{[s]}$ (split point at the end) mentioned in the previous section.

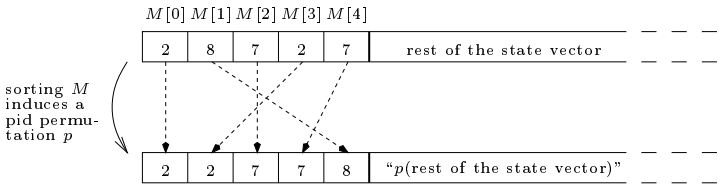
We generalise this approach. Recall that a lexicographic ordering is an ordering on sequences of elements, and it is based on an underlying ordering on the type of the elements. Normally, in lifting this underlying ordering to sequences, "weights" are attached to all positions in the sequence, where the weight increases as the position is further to the left[1] (therefore *ab* would precede *ba* in a dictionary). Splitting the state vector and letting only the leftmost part determine the lexicographic value, as done in [17], may be seen as assigning weights only to the leftmost k positions, for some k. In other words, the weight function from positions to weights becomes partial. We generalise this further by dropping the requirement that weights increase as positions are further to the left. Hence, we consider lexicographic orderings that are parameterized by arbitrary partial weight functions. Viewed differently, such a weight function not only truncates the state vector (as in [17]), but also reshuffles it, before computing the lexicographic value in the standard way. Thus, we have extended the freedom in selecting a representative function. In order to explain how this freedom is exploited, we first look into the way the equivalence on states is defined in our particular case.

Symmetry relations between states are generated by permuting scalars. A given *scalar permutation* $p : I \to I$ can be lifted to states. The precise definition

[1] Note that it is only the relative weight of an element w.r.t. other elements in the sequence that counts, not its absolute value.

of this lifting can be found in [3]. Intuitively, the application of a scalar permutation p to a state vector s replaces every scalar i that occurs in s by $p(i)$. Because scalars may also occur as indices of arrays, array values may have to be reshuffled in order to get a proper state vector again. An equivalence class is a minimal non-empty set of states that is closed under applying scalar permutations. The representatives of such a class are those states that are minimal under some given (generalised) lexicographic ordering. These representatives are the states that are actually stored as the exploration of the transition system proceeds.

First, we explain how to choose a lexicographic ordering so as to further optimise on the *speed*, i.e. the computational effort in mapping a state s to its representative $rep(s)$. Assume that the program uses an array indexed by scalars and whose elements do not involve scalars; we call this a *main array*. Then take such an array, say M, and choose the partial weight function that assigns weights only to the elements of M (and to nothing else in the state vector), with weights decreasing from the first element $M[0]$ to the last $M[n-1]$. Now observe that under the lexicographic ordering based on this partial weight function, in all representatives (in any equivalence class), M is *sorted*. This gives rise to the following idea in order to compute a representative $rep(s)$: Instead of applying different scalar permutations to s until a minimal state vector is found, *sort* M in s. Depending on the particular sorting algorithm used, this gives rise to a certain scalar permutation, that is then applied to the rest of s in order to obtain a representative. As an example, consider the following picture showing a state vector before and after sorting a main array M, where M has been depicted at the beginning of the state vector to stress the fact that its elements have the highest lexicographic weight.



M is indexed by scalars, 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 sorting algorithm being used sorts M as indicated by the dashed arrows. This particular sorting induces the scalar 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. This is called the sorted strategy. Note that it is not canonical.

Another strategy presented in [3] is based on the sorted strategy, and trades some speed for *precision*. It is called the segmented strategy. Instead of sorting M in one particular way, it considers all sortings of M. To continue with the example, observe that there are 3 more ways to sort M, depending on how the two 2's and the two 7's are moved around. An example of a corresponding scalar permutation is $p' = \{3 \mapsto 0, 0 \mapsto 1, 4 \mapsto 2, 2 \mapsto 3, 1 \mapsto 4\}$. Furthermore, the

segmented strategy is based on a lexicographic ordering which, just as sorted, gives the highest weights to M's elements, but in addition it assigns weights to all other elements in the state vector as well, i.e., it is a total function. The segmented strategy considers the results of applying the scalar permutations corresponding to all ways to sort M, and selects the lexicographically smallest among these. Clearly, this results in *canonical* representatives[2].

The assumption of an explicit array M can be dropped if the program contains a family $C_0, \ldots, C_{n-1}$ of processes $C = \lambda\, i : I.C_i$ parameterized by scalars. In every such program there is, in fact, an implicit array indexed by scalars, namely the array of program counters for processes C_i. Thus, we can consider the variants of sorted and segmented in which we use the array of program counters for M. These variants are called pc-sorted and pc-segmented, respectively, the latter one being canonical again.

Finally, full is the (canonical) strategy that generates all scalar permutations and applies each of them to the state s, selecting the (standard) lexicographic minimum as representative. It is used for reference purposes.

4 Extending Spin with Symmetry Reductions

The result of our extension of Spin with a symmetry package is called SymmSpin. When extending an existing enumerative model checker, in the spirit of [18], two essential problems have to 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.

As far as the first problem is concerned, it could be solved just by lifting the syntactical extensions of Murφ [18] 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.

[2] In [3], we prove a more general result stating that for canonicity it is not necessary to extend the partial weight function of the sorted strategy to a full function, but that an arbitrary choice function can be used.

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 $T = (\mathcal{S}, s_0, \rightarrow)$, 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 := {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 s' ∉ reached then
            add s' to reached and unexpanded;
```

Fig. 1. A standard exploration algorithm

```
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

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 the particular 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 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.

Obviously, our algorithm is still sound. Beside avoiding the problem with "add $rep(s')$ to *unexpanded*", it has yet another advantage over the algorithm in Fig. 2: 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,

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

```
reached := {rep(s_0)}; unexpanded := {s_0};
while unexpanded ≠ Ø 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. A modified exploration algorithm with symmetry reductions

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 successor states s' considered in the loop "for each transition $s \to s'$ do" were different. It can easily be proven that this is not a case. Whenever the algorithm in Fig. 3 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).

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 demand 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. 3.

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` [1]. 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.

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.

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 succesor 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.

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 [20], and presented in Fig. 4) is used for this purpose.

It happens that the transpositions generated by the algorithm always swap two succesive 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 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 $P(v_2)$.

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 permutation of scalarset values 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 [18]). 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.

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. 5. 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`.

5 Experimental Results

We tried our implementation on several examples, like Peterson's mutual exclusion algorithm [21], or Data Base Manager from [23] (see also [3] for more details about these and other examples.) The obtained reduction 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 in general there is no favorite among the reduction strategies regarding the space/time ratio. This justifies our decision to have all strategies (maybe except full) as separate options of the extended model-checker.

In the verification experiments we used Spin both with and without the partial order reduction (POR) option. Allowing Spin to use its POR algorithm together with our symmetry reductions is sound due to Theorem 19 in [8] 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. 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 two examples. They were chosen so as to test the capabilities to deal with queues, multiple scalar sets and multiple process families. All experiments were performed on a Sun Ultra-Enterprise machine, with three 248 MHz UltraSPARC-II processors and 2304 MB 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.

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 [13] is to synchronize the processes such that none of them can enter its main part before all the others have finished their initial parts.[4] Process synchronization

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

is done via shared boolean variables. Each process has so called co-component process whose task is to manage the synchronization of the boolean variables. We refer the reader to [13] 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.

Table 1. Results for the Initialization Protocol example.

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

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.

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.

Base Station. This example is a simplified version of MASCARA – a telecommunication protocol developed by the WAND (Wireless ATM Network Demon-

strator) consortium [6]. The protocol provides an extension of the ATM (Asynchronous Transfer Mode) networking protocol to wireless networks. Our model represents a wireless network connecting $n_s \geq 1$ sending mobile stations and $n_r \geq 1$ receiving mobile stations 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 sender station A wants to send a message to receiver station 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 $m = 2$ radio channels.

Table 2. Results for the Base Station example.

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

There are two families of symmetric processes in the 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.)

A version of this protocol was analyzed in [3]. There, we had to rewrite the original model such that all the communication was done via global variables. For the experiments in this paper we were able to use the original model in which the communication was modeled with Promela channels. As a result, the contribution of the partial order reduction became visible. One can also notice

the already mentioned 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 the heuristic presented in [3]. It provides the 4 reduction strategies based on this heuristic (sorted/pc-sorted and segmented/pc-segmented) as well as the full reduction strategy. Compared to the prototype implementation that was used for the feasibility study reported in [3], the current implementation covers a more general class of symmetric systems. Namely, it is able to deal with multiple scalar sets and multiple process families. In addition, almost all Promela features are now handled, including queues. 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. We point the interested reader to the web site accompanying this paper, [1], for the full source code.

This enhanced implementation enabled a more extensive series of experiments, exploiting the additional symmetric structure in models and the richer variety of Promela constructs handled. Two of the new experiments are reported here. Naturally, also the experiments reported in [3] have been repeated and successfully reproduced in the process of testing the new implementation.

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 [18], 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.

On a theoretical level, the symmetry package is compatible with Spin's LTL verification algorithm combined with partial order reduction. What needs to be verified is that our arguments are also valid for the particular implementations in Spin, more precisely, the cycle detection algorithm. Experience has taught us that this kind of issues should be treated cautiously.

References

1. AdjustPan script, `http://www.win.tue.nl/~lhol/SymmSpin`, April 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, Oct. 1998.
3. D. Bošnački, D. Dams, L. Holenderski, A Heuristic for Symmetry Reductions with Scalarsets, submitted to *FORTE'2000 (The 13th Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols)*.

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. I. Dravapoulos, N. Pronios, S. Denazis *et al*, *The Magic WAND, Deliverable 3D2, Wireless ATM MAC*, Sep 1997.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. W.H.J.Feijen, A.J.M. van Gasteren, *On a method of multiprogramming*, Springer-Verlag, 1999
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*, PhD thesis, Department of Computer Science of Stanford University, Dec 1996.
20. V. Lipskiy, Kombinatorika dlya programmistov, Mir, Moscow, 1988. (In Russian)
21. N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996.
22. 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.
23. A. Valmari, Stubborn sets for reduced state space generation, *Advances in Petri Nets 1990*, LNCS 483, 491–515, Springer, 1991.

```
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. 4. Generating permutations by transpositions

```
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. 5. Permuting blocks