Symmetry Reduction Criteria for Software Model Checking *

Radu Iosif

Computer and Information Sciences Department,
318 Nichols Hall, Kansas State University
Manhattan, KS 66502, USA
iosif@cis.ksu.edu

Abstract. Symmetry reduction techniques exploit symmetries that occur during the execution of a system, in order to minimize its state space for efficient verification of temporal logic properties. This paper presents a framework for concisely defining and evaluating two symmetry reductions currently used in software model checking, involving heap objects and, respectively, processes. An on-the-fly state space exploration algorithm combining both techniques is also presented. Second, the relation between symmetry and partial order reductions is investigated, showing how one's strengths can be used to compensate for the other's weaknesses. The symmetry reductions presented here were implemented in the dSPIN model checking tool. We performed a number of experiments that show significant progress in reducing the cost of finite state software verification.

1 Introduction

The increasing complexity in the design of concurrent software artifacts demands new validation techniques. Model checking [4] is a widespread technique for automated verification of concurrent systems that has been recently applied to the verification of software. Unfortunately, the use of model checking tools [13] is often limited by the size of the physical memory, due to the state explosion problem. In order to deal with this problem, various reduction techniques have been proposed in the literature. Among those, symmetry reductions [3], [8] and partial-order reductions [10], [22] have gained substantial credibility over the past decade. Both techniques are automatic and can be applied on-the-fly, during model checking. The reduction achieved can be significant, in the best cases exponential in the size of the state space.

Symmetry reductions exploit the structure of states in order to identify symmetries that occur during verification. The intuition behind these strategies is

* This work was supported in part by NSF under grant CCR-9703094, by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement number DAAD190110564, and from the Formal Verification of Integrated Modular Avionics Software cooperative agreement, NCC-1-399, sponsored by Honeywell Technology Center and NASA Langley Research Center.

that the order in which state components (processes, objects) are stored in a state does not influence the future behavior of the system. That is, the successors of two symmetric states are also symmetric. Many criteria have been proposed to decide whether two states are symmetric on-the-fly, without any information about the future states. They usually exploit the ordering of processes [6], communication channels and the structure of temporal logic formulas used to express correctness requirements [8]. Ideally, the reduced state space will have only one state representing each symmetry equivalence class. Unfortunately, detecting all symmetries usually requires very expensive computations, that may make such reductions impractical.

Partial order reductions exploit the commutativity of concurrent transitions, which result in the same state when executed in different orders. The decision whether two transitions are independent, so that they can be safely swapped, is usually made using compile-time static analysis. In practice, this information is a conservative approximation of the real run-time independence. As in the case of symmetry reductions, using more information about the system helps detecting more independence, however it is computationally more expensive. It has been shown [7] that symmetry and partial order reductions are orthogonal strategies and can be used in combination to achieve better verification results.

The main contribution of this paper is applying both techniques to a particular class of software, namely dynamic programs, for which the number of state components (processes, objects) is continuously modified as a result of their ongoing execution. This concept can be used to formalize the semantics of most high-level object-oriented programs, such as the ones written in Java or C++. We show how existing reduction techniques can be specialized to exploit the dynamic nature of software systems in order to achieve more effective verification results.

The present paper is, to some extent, the continuation of our work reported in [19]. There we presented a canonical symmetry reduction that applies only to the heap of the program. Here we combine the heap symmetry reductions with more traditional approaches, such as process symmetries [6]. We first define a framework that allows us to express both reductions formally and compare their efficiency, in terms of canonical properties. Then we describe an explicit-state exploration algorithm that combines heap with process symmetry reduction on-the-fly. Finally, we investigate further optimizations by relating heap symmetries with partial order reductions. Preservation of temporal logic properties is discussed throughout the paper. A prototype implementation of the ideas described in this paper has been done in dSPIN [17], an extension of SPIN [13], especially designed for software model checking. We performed a number of experiments on two non-trivial test cases in order to obtain a practical assessment of our ideas.

1.1 Related Work

Among the first to use symmetries in model checking were Clarke, Filkorn and Jha [3], Emerson and Sistla [8] and Ip and Dill [20]. These approaches consider systems composed of a fixed number of active components (processors) [3], variables of a special symmetry-preserving data type (scalarset) [20] as well as symmetries of specifications [8]. Using sorting permutation to reduce the complexity of representatives computations has been addressed by the work of Bosnacki and Dams [6]. The problem of exploiting heap symmetries in software model checking has been informally addressed by Visser and Lerda in [21]. To our knowledge, they are the only other group that have addressed heap symmetries to date. Their approach looks attractive due to its simplicity, but no formal evidence of its canonical properties has yet been provided by the authors.

2 Preliminaries

In this section we present some background notions regarding symmetry. The classical framework [3], [8] starts from the basic notion of *group of automorphisms* in order to define symmetry as an equivalence between states. Since automorphisms preserve graph structure, it can be shown that the symmetry induced by a group of automorphisms is a bisimulation in the sense of Milner [12]. It is therefore possible to define a quotient structure in which each state is a (representative of a) symmetry equivalence class. Model checking the reduced structure preserves all properties that can be expressed using temporal logics [4].

Unfortunately, applying this framework directly to software model checking faces the difficulty of giving the automorphisms appropriate semantic definitions. Indeed, when considering a program in which the number of state components (such as objects or threads) may experience an unbounded growth along an execution path, one cannot consider only one group of permutations as the group of system automorphisms. Instead, we consider a (possibly infinite) family of such groups and chose one at each step, by keeping track of the number of components in every state.

Let G_n denote the set of all permutations on the set $\{1, \ldots, n\}$. It is easy to see that G_n forms a group with function composition, inverse and the identity mapping as neutral element. Formally, we represent program executions by an (augmented) Kripke structure $K = (S, R, L, \mathcal{N})$ over a set of atomic propositions $\mathcal{P}$ and a set of actions Σ, where:

- S is a set of states,
- $R \subseteq S \times \Sigma \times S$ is a transition relation,
- $L : S \to 2^{\mathcal{P}}$ is a function that labels states with sets of atomic propositions,
- $\mathcal{N}$ is a family of functions $\eta_\tau : S \to \mathbb{N}$, where $\eta_\tau(s)$ is the number of components of type τ occurring in state s.

In cases where the last ($\mathcal{N}$) component is irrelevant for the discussion, we may omit it. A transition $(s, \alpha, t) \in R$ is also denoted by $s \xrightarrow{\alpha} t$. We consider that permutations on numbers induce permutations on states. Let $\pi \in G_n$ be a permutation. We denote by $\pi_\tau(s)$ the application of π only to the components of type τ in s, given that $\eta_\tau(s) = n$. More precisely, let $S_{\tau,n} = \{s \in S \mid \eta_\tau(s) =$

$n\}$ be the set of all states whose number of τ-components is n. Any bijection $\pi_\tau : S_{\tau,n} \rightarrow S_{\tau,n}$ is a state permutation. In Section 3 we formally express π_τ in function of π for two types of state components: heap-allocated objects and processes.

Definition 1. *Let $K = (S, R, L, \mathcal{N})$ be a structure. For some component type τ, a binary relation $\equiv_\tau \subseteq S \times S$ is a τ-symmetry iff, for all $s \equiv_\tau t$, the following hold:*

- *$L(s) = L(t)$,*
- *$\eta_{\tau x}(s) = \eta_x(t)$, for all $\eta_x \in \mathcal{N}$,*
- *$\pi_\tau(s) = t$ for some $\pi \in G_{\eta_\tau(s)}$.*

Using basic group theory, it can be proved that $\equiv_\tau$ is an equivalence relation. The equivalence class, also known as the *orbit*, of a state s is denoted by $[s]_\tau$. Throughout this paper we omit τ whenever it is implicit or irrelevant to the discussion. The *quotient* structure w.r.t. a τ-symmetry is defined as follows:

Definition 2. *Given a structure $K = (S, R, L, \mathcal{N})$ and a symmetry relation $\equiv_\tau$ on S, the quotient structure for K w.r.t to $\equiv_\tau$ is $K_{/\equiv_\tau} = (S_\tau, R_\tau, L_\tau, \mathcal{N}_\tau)$, where:*

- *$S_\tau = \{[s]_\tau \mid s \in S\}$,*
- *$R_\tau = \{([s]_\tau, \alpha, [t]_\tau) \mid (s, \alpha, t) \in R\}$,*
- *$L_\tau([s]_\tau) = L(s)$, for all $s \in S$,*
- *$\eta_x([s]_\tau) = \eta_x(s)$, for all $\eta_x \in \mathcal{N}$*

The states of a quotient structure are equivalence classes of states from the original structure and a transition occurs between two equivalence classes whenever a transition (labeled with the same action) occurs between states from the original structure. It is clear, from the first two points of Definition (1), that L_τ and $\mathcal{N}_\tau$ are well defined for the quotient structure. Since the set S_τ is a (possibly non-trivial) partition of S, it is potentially more efficient to model check a temporal logic formula on $K_{/\equiv_\tau}$ instead of K, provided that they represent equivalent computations. We use here the notion of bisimulation in the sense of Milner [12] strengthened with equivalence w.r.t to the set of atomic propositions $\mathcal{P}$:

Definition 3. *Let $K_1 = (S_1, R_1, L_1)$ and $K_2 = (S_2, R_2, L_2)$ be Kripke structures over the set of actions Σ. A binary relation $\approx \subseteq S_1 \times S_2$ is a bisimulation iff, for all $s_1 \approx s_2$ and $\alpha \in \Sigma$, all the following hold:*

- *$L_1(s_1) = L_2(s_2)$,*
- *$\forall\ t_1 \in S_1\ .\ (s_1, \alpha, t_1) \in R_1 \ \Rightarrow \exists\ t_2 \in S_2\ .\ (s_2, \alpha, t_2) \in R_2$ and $t_1 \approx t_2$,*
- *$\forall\ t_2 \in S_2\ .\ (s_2, \alpha, t_2) \in R_2 \ \Rightarrow \exists\ t_1 \in S_1\ .\ (s_1, \alpha, t_1) \in R_1$ and $t_1 \approx t_2$.*

If $\approx$ is total on S_1 and S_2 we say that K_1 and K_2 are bisimilar, and denote this by $K_1 \approx K_2$. It is known fact that bisimilar states cannot be distinguished by formulas of mu-calculus or any of its sub-logics, such as computation-tree logic (CTL) or linear-time temporal logic (LTL) [4].

Using the symmetry framework in explicit-state model checking requires computation of representatives for each equivalence class. Unfortunately, finding the general solution to this problem is known to be as hard as proving graph isomorphism, for which no polynomial-time solution is known to exist [3]. Solutions proposed in the literature either deal with incomplete equivalence classes for which the orbit problem has polynomial solution [3] (i.e., the *bounded orbit problem*), or use heuristic strategies [6], [19].

Definition 4. *Given a structure $K = (S, R, L)$ and a symmetry relation $\equiv_\tau$, a function $h : S \rightarrow S$ is said to be a canonical representative for $\equiv_\tau$ iff, for all $s, s' \in S$ both the following hold:*

- $s \equiv_\tau h(s)$, *and,*
- $s \equiv_\tau s' \iff h(s) = h(s')$.

Throughout this paper we use sorting heuristics, as the ones described in [6], [19]. Below we introduce a formal definition that captures the idea of such strategies.

Definition 5. *Let $K = (S, R, L, \mathcal{N})$ be a structure and $\xi : S \times \mathbb{N} \times \mathbb{N} \rightarrow \{true, false\}$ be a partial boolean mapping. Given a state s and component type τ, a permutation $\pi^\xi \in G_{\eta_\tau}$ is said to be sorting for s iff for all $0 \leq i, j < \eta_\tau(s)$, $\pi^\xi(i) < \pi^\xi(j) \iff \xi(s, i, j) = true$.*

In the following, we refer to the ξ function as to the *sorting criterion*. The reason why ξ is allowed to be partial is a rather technical formality: we are not interested in the values $\xi(s, i, j)$ where i or j is greater than $\eta_\tau(s)$. The intuition behind sorting criteria and sorting permutations are better explained with an example. Let $v : \{1, \ldots, n\} \rightarrow \mathbb{N}$ be a (finite) vector whose elements are natural numbers. Obviously, the vector is sorted when, for all $1 \leq i < j \leq n$ we have $v(i) \leq v(j)$ $(= \xi(v, i, j))$. Otherwise, for some $k < l$ the condition $\xi(v, k, l)$ is not met. In this case, a permutation $\pi \in G_n$ exists such that the new vector $v \circ \pi$ is sorted. Then we say that π is sorting for v w.r.t to the ξ criterion.

The heuristics used in this paper follow the same pattern. Given a state s and a sorting criterion ξ we compute a sorting permutation π^ξ for s w.r.t. ξ. The representative of the symmetry equivalence class $[s]_\tau$ will be $h(s) = \pi^\xi_\tau(s)$. Necessary and sufficient conditions for the representative function to be canonical in the sense of Definition (4) are given by the following theorem. Due to space limitations, all proofs are omitted from this paper.

Theorem 1. *Let $K = (S, R, L, \mathcal{N})$ be a structure, $\equiv_\tau \subseteq S \times S$ be a symmetry relation and ξ be a sorting criterion. Then the sorting permutations induced by ξ are canonical representatives for $\equiv_\tau$ iff, for each state $s \in S$ and $0 \leq i, j < \eta_\tau(s)$, $i \neq j$, both the following hold:*

- *ξ remains invariant under permutations of s, i.e,* $\forall \pi \in G_{\eta_\tau(s)}$, $\xi(s, i, j) = \xi(\pi_\tau(s), \pi(i), \pi(i))$ *and,*

$$Store = Variable \longmapsto Location \qquad Process = ProcCnt \times Store$$
$$Heap = Location \longmapsto Store \qquad ProcPool = ProcId \longmapsto Process$$
$$StateHeap = Heap \times Location \qquad StateProc = ProcPool \times ProcId$$

Fig. 1. Semantic domains

– *ξ induces a strict total order on the set $\{0, \ldots, \eta_\tau(s) - 1\}$ i.e., $\xi(s, i, j) \vee \xi(s, j, i) = true$ and $\neg\xi(s, i, j) \vee \neg\xi(s, j, i) = true$.*

The above result leverages the difficult task of proving strategies canonical. It will be applied in Section 3 in order to compare two techniques, involving the detection of state symmetries induced by permutations of heap objects and processes. It will be also shown that the reduction strategy involving heap objects is canonical, while the one involving processes is not.

3 Semantic Definitions of State Symmetries

In this section we are concerned with defining state symmetries i.e., symmetries that can be discovered by inspecting the structure of the state. We present a (partial) semantic definition of programs that modify the number of state components (objects, processes) as part of their execution. This class of programs is also referred to in the literature as *dynamic* programs [17]. For space reasons, we are not going to enter here all the details of language definition. For more details, the interested reader is referred to [16]. Instead, in the following we define program configurations and give small-step operational semantic rules only for some of the allocator statements.

3.1 Domains and Rules

Consider the semantic domains in Figure 1. The definition of *Store* is the classical one: a partial mapping between variables and values. For simplicity reasons we assume that all variables will take memory reference values from the set *Location*. A *Heap* consists of a partial mapping between memory locations and stores. We may refer to the stores in the range of a heap as to *objects*. The second component of a *StateHeap* is a location used to describe the implementation of object allocator statements; it holds the last allocated location. A *Process* is a pair consisting of a program counter and a store for local variables. Processes are referred to by means of *ProcId* values, and the *ProcPool* domain represents a collection of active processes. Similarly, the second component of a *StateProc* represents the last allocated *ProcId*. We conclude our description of the semantic domains with the following assumptions:

1. there exists a strict total ordering $\prec_v \subseteq Variable \times Variable$.

2. there exists a strict total ordering $\prec_i \subseteq ProcId_\top \times ProcId_\top$, where $ProcId_\top = ProcId \cup \{\top\}$ and $\top$ is less than every element of $ProcId$.
3. there exists a strict total ordering $\prec_c \subseteq ProcCnt \times ProcCnt$ and a function $next : ProcCnt \to ProcCnt$ such that $next(c)$ always returns the next element w.r.t. to $\prec_c$ i.e, the program location of the next statement within the process; computations are assumed to be infinite; the least element in the order is denoted by $init$.
4. there exists a strict total ordering on $Location$ and a function $new : Location \to Location$ such that $new(l)$ always returns the next location in that ordering; the least element is denoted by $null$; the set $Location$ is assumed to be infinite and countable.

With the above definitions and assumptions, we consider a program configuration (state) to be an element of the $State$ set, defined as follows:

$$\sigma \in State = Store \times StateHeap \times StateProc$$

Intuitively, the first component of the triple σ is a store that holds global variables, the second is a heap referencing all existing objects, and the third is the thread pool referencing all active threads in σ.

Figure 2 presents structural rules that define the small-step semantics of object allocator statements. These rules are needed mostly for the discussion in Section 5. For some $j \in ProcId$, the notation $\sigma \vdash_j \texttt{ast} \Longrightarrow \sigma'$ expresses the fact that the process referred to by j in state σ, executing the statement given by the abstract syntax tree `ast` changes the program state into σ'.

$$\frac{\begin{array}{c}\sigma = (s,(h,l),(p,i)), \quad s(x) \neq \perp, \quad p(j) = (c, s') \\ c' = next(c), \qquad l' = new(l), \ o = \lambda v.null\end{array}}{\sigma \vdash_j \texttt{x} = \texttt{new}() \Longrightarrow ([x \to l']s, ([l' \to o]h, l'), ([j \to (c', s')]p, i))} \ (NEW1)$$

$$\frac{\begin{array}{c}\sigma = (s,(h,l),(p,i)), \ \ p(j) = (c, s'), \quad s'(x) \neq \perp, \ \ c' = next(c) \\ l' = new(l), \qquad s'' = [x \to l']s', \ o = \lambda v.null,\end{array}}{\sigma \vdash_j \texttt{x} = \texttt{new}() \Longrightarrow (s, ([l' \to o]h, l'), ([j \to (c', s'')]p, i))} \ (NEW2)$$

Fig. 2. Allocator Rules

The first rule (NEW1) describes the state changes that occur due to an object allocation where the left hand side of the statement is a global variable ($s(x) \neq \perp$). Analogous, the second rule (NEW2) describes the state changes caused by a heap allocation where the left hand side is a local variable. All rules reflect also the implicit change of control within the current process. It is to be noticed that the allocation strategies exploit the order on the set of memory locations. Namely, the next available element, as returned by the new function, is used for allocation of fresh components. Such allocation strategies are commonly used in real-life implementation of dynamic programming languages. For the purposes of this paper, we will refer to these techniques as to *next-free allocation* strategies.

We are now able to complete the formal definition of state symmetries by defining the meaning of a permutation π applied to the heap and process components of a state $\sigma = (s, (h, l), (p, i))$. Formally, since the set *Location* in Figure 1 has been considered countable, we have $Location = \{l_0, l_1, \ldots, \}$ and by $\pi(l_x)$ we actually denote $l_{\pi(x)}$. A similar notation is used for the application of permutations to the elements of the set *ProcId* below.

$$\pi_{heap}(\sigma) = (\pi_{heap}(s), (\pi_{heap}(h), l), (\pi_{heap}(p), i)) \tag{1}$$
$$\pi_{heap}(s) = \lambda v.\pi(s(v)) \tag{2}$$
$$\pi_{heap}(h) = \lambda lv.\pi(h(\pi^{-1}(l), v)) \tag{3}$$
$$\pi_{heap}(p) = \lambda i(\lambda cs.(c, \pi_{heap}(s)))p(i) \tag{4}$$

$$\pi_{proc}(\sigma) = (s, (h, l), (\pi_{proc}(p), i)) \tag{5}$$
$$\pi_{proc}(p) = \lambda i.p(\pi^{-1}(i)) \tag{6}$$

Informally, the equations (1 - 4) say that, applying a permutation to a state, will permute all locations that are values of reference variables in the global store, local stores within processes, and in each heap object. The objects in the heap are also permuted, by the inverse permutation, in order to consistently reflect this change. Permuting processes (5 - 6) is easier, since we consider that processes are not referenced by variables, in our simple language.

3.2 Heap and Process Sorting Criteria

The other issue that remains to be dealt with in order to use heap and process symmetries in practical software model checking, is the complexity of computing the representatives of symmetry equivalence classes. As mentioned before, in Section 2, we rely on sorting heuristics in order to improve the performance of our reduction algorithm. In the remainder of this section, we will briefly explain the ideas behind such heuristics using sorting criteria, as introduced by Definition (5). Sorting heap objects is discussed in more detail in [19], while specific information regarding sorting processes can be found in [6].

Since the heap is not a linear structure, finding a canonical order can be done via topological sorting. However, a topological order is usually partial. Normally, a total order can be derived from a partial one by linearization and in our case we achieve that assuming a strict total order on variables ($\prec_v$) and process identifiers ($\prec_i$). In practice, it is often the case that a strict total order on the set of variables can be found at compile-time, and one might consider for instance alphabetical order, declaration order, etc. This automatically induces the required order on the set of sequences of variables prefixed by a process identifier. There is need for a process identifier as prefix in order to distinguish between local variables. Identical processes will contain multiple copies of the same local variable and they can only be ordered using unique process identifiers. Record fields can be distinguished from global or local variables by prefixing them with the name of the record, as it is done in most object-oriented compilers.

Intuitively, when sorting a heap structure, we take into account, for each object, reachability information that is, the chains of variables including global, local or field variables, that reach every object. Formally, let $Variable^*$ denote the set of sequences of variables and let $\prec_v^*$ be the lexicographical order induced by $\prec_v$ on sequences. Also, let $\prec^*$ be a strict total order on the set $Chain = ProcId_\top \times Variable^*$ naturally induced by both the order on $ProcId_\top$ ($\prec_i$) and $\prec_v^*$. As a convention, we use the literals i, j to denote process identifiers, v, u to denote sequences and x, y to denote variables. The notation min^* denotes the greatest lower bound with respect to $\prec^*$ and $\langle v, u \rangle$ is sequence concatenation. The $\perp$ symbol denotes undefinedness of partial mappings. Consider the following partial mappings:

$$reach : State \times Chain \rightarrow Location$$

$$reach(\sigma, v) = \begin{cases} s(x) & v = \langle \top, x \rangle \\ s'(x) & p(i) = (c, s') \wedge v = \langle i, x \rangle \\ h(reach(\sigma, u), x) & reach(\sigma, u) \neq \perp \wedge v = \langle u, x \rangle \\ \perp & otherwise \end{cases}$$

$$trace : State \times Location \rightarrow Chain$$

$$trace(\sigma, l) = \begin{cases} min^*\{v \mid reach(\sigma, v) = l\} & \exists u \in Chain \ . \ reach(\sigma, u) = l \\ \perp & otherwise \end{cases}$$

The sorting criterion for heap objects is denoted by ξ_{heap} and is defined as follows:

$$\xi_{heap}(\sigma, m, n) = (trace(\sigma, l_m) \prec^* trace(\sigma, l_n)) \tag{7}$$

In order to asses the performance of this sorting criterion, we will show that it actually can be the base for a canonical reduction strategy.

Lemma 1. *For all* $\pi \in G_{\eta_{heap}(\sigma)}, l \in Location$, $trace(\sigma, l) = trace(\pi_{heap}(\sigma), \pi(l))$.

The first condition of Theorem 1 holds as a consequence of Lemma 1. The second condition holds due to the fact that $\prec^*$ was assumed to be a strict total order on the set $Chain$ and that each chain uniquely identifies a reachable object location (one variable cannot point to two different objects, from the definitions of $Store$ and $Heap$). Consequently, the strategy based on heap objects is canonical, yielding optimal reductions.

The heuristics proposed in [6] use the idea of sorting processes. One such strategy, called *pc-sorted*, uses the values of the program counters in the sorting criterion. Let $c' \preceq_c c''$ stand for $c' \prec_c c'' \vee c' = c''$. Formally, we denote by ξ_{proc} the following predicate:

$$\sigma = (s, (h, l), (p, i))$$

$$\xi_{proc}(s, m, n) = (p(m) = (c', s') \wedge p(n) = (c'', s'') \wedge c' \preceq_c c'') \tag{8}$$

It is easy to see that the first condition of Theorem 1 is met by ξ_{proc}, while the second one is not aways met. Indeed, it can be often the case that two identical processes are at the same location, that is, the values of their program counters are equal. This situation violates the second requirement of Theorem 1, therefore the reduction strategy induced by ξ_{proc} is not necessarily canonical.

4 Combining Heap and Process Symmetries

The main contribution of this section is the presentation of a reduced state space search algorithm that combines the heap and process symmetry reduction strategies defined in Section 3 on-the-fly. For heap symmetries, we briefly describe the algorithm used to compute canonical sorting permutations.

Assuming the existence of a representative function *rep*, Figure 3 shows the basic depth first search algorithm [15] with symmetry reductions. The correctness of the algorithm in Figure 3 is ensured by the fact that for each s, we have $s \equiv rep(s)$ by Definition (4). In case $rep(s)$ is already in the state space when the search reaches s, all its outgoing transitions have been already explored by DFS and since all transitions originating in s are bisimilar to the ones originating in $rep(s)$, the search algorithm can safely backtrack. The extension of the correctness argument to the cycle detection algorithm [5], which is the base of the automata-theoretic approach [5] in SPIN, was reported in [2].

```
DFS(s)
if error(s) then report error fi
add(rep(s), Statespace)
for each successor t of s do
    if rep(t) not in Statespace then DFS(t) fi
od
end DFS
```

Fig. 3. Symmetry Reduced Depth First Search

In the following, we discuss the effective computation of $rep(s)$. Intuitively, the algorithm used to implement *rep* can be decomposed into two distinct phases. First we generate a sorting permutation π for s; the result of *rep* will be the application of this permutation to the family τ of components in s, i.e., $\pi_\tau(s)$. The rules for applying a permutation to heap objects and processes in s are the ones given by equations (1 - 6) in Section 3.

For heap objects, the algorithm used to compute sorting permutations is presented in Figure 4. Let us remember the fact that a total strict order $\prec_v$ on the set of variables is assumed to exist. We consider a function $ordered : Stores \rightarrow Variables^*$ that returns, for a given store, the $\prec_v$-ordered sequence of variables that are defined in that store.

Input: configuration $\sigma = (s, (h, l), (p, i))$
Output: sorting permutation $\pi_{heap} \in G_{\eta_{heap}}(\sigma)$

```
SORT(store)                              begin main
for next v from ordered(store) do        k = 0; π_heap = λx. ⊥
    l_i = store(v)                       SORT(s)
    if l_i not marked do                 for each 0 ≤ i ≤ η_proc(σ) do
       mark l_i                          (*) (c, s') = p(i)
       π_heap = [i → k]π_heap                SORT(s')
       k = k + 1                         od
       SORT(h(l_i))                      end main
    od
od
end SORT
```

Fig. 4. Generation of Sorting Permutations for Heap Objects

The correctness of the algorithm in Figure 4 has been discussed in great detail in [19]. In this case, correctness implies that the generated permutation π_{heap} always meets the sorting criterion ξ_{heap}, defined in Section 3. Informally, it can be noticed that, every (reachable) object stored at location l in state σ, will be eventually reached by a call to the SORT procedure. The complexity of the sorting permutation algorithm for heap objects is $O(\eta_{heap}(\sigma))$, since SORT visits every object and every field connecting two objects only once. Let us notice that in this case, the maximum number of outgoing edges from an object is bounded by a compile-time constant which is the maximum number of variables declared in the program.

The problem of computing sorting permutations for processes reduces to the vector sorting problem, which can be solved by existing algorithms in time $O(\eta_{proc}(\sigma)\log(\eta_{proc}(\sigma)))$. As a remark, the process ordering strategies presented in [6] do not explicitly separate sorting permutation computation and application, but rather compute representatives in one step. Here we need to keep that distinction in order to describe the composition of the two reduction strategies. The following discussion will present the combined strategy.

The idea of combining the two reduction techniques originates from the observation that the application of two permutations ρ and π to heap objects and processes respectively, as defined by equations (1 - 6), operate independently on different types of components. Therefore their composition could be easily defined, i.e., $\rho_{heap}(\pi_{proc}(\sigma))$. It is clear from the equations (1 - 6), that the composition is commutative, in the following sense: $\rho_{heap}(\pi_{proc}(\sigma)) = \pi_{proc}(\rho_{heap}(\sigma))$. However, using this straightforward composition to define the representative function *rep* for the algorithm in Figure 3 faces the following problem: if ρ has been computed in σ using the sorting criterion ξ_{heap}, it might be the case that ρ is no longer sorting, according to ξ_{heap}, for $\pi_{proc}(\sigma)$. Analogously, computing ρ

in $\pi_{proc}(\sigma)$ might not satisfy ξ_{heap} for σ. As a result, applying the heap permutations computed according to ξ_{heap} (by the algorithm in Figure 4) does not give the canonical representatives for heap symmetric states. The reason lies within the definition of ξ_{heap} (Section 3), since a chain that reaches a location may be prefixed with a process identifier, and therefore the minimal chain $trace(\sigma, l)$ may depend on the order of processes. In other words, permuting processes may affect the canonical property of the heap symmetry reduction. In order to overcome this difficulty, we need to record information that allows us to establish a fixed order on processes during the state space search. The following definition captures the formal meaning of the combined symmetry:

Definition 6. *Let $K = (S, R, L, \mathcal{N})$ be a structure, with $\mathcal{N} = \{\eta_{heap}, \eta_{proc}\}$. Two states σ, σ' are said to be fully symmetrical, denoted by $\sigma \equiv_{full} \sigma'$ iff the following hold:*

- $L(\sigma) = L(\sigma')$,
- $\eta_{heap}(\sigma) = \eta_{heap}(\sigma')$ *and* $\eta_{proc}(\sigma) = \eta_{proc}(\sigma')$,
- $\zeta_{heap}(\pi_{proc}(\sigma)) = \sigma'$ *for some* $\zeta \in G_{\eta_{heap}}(\sigma)$ *and* $\pi \in G_{\eta_{proc}}(\sigma)$.

We will proceed under the simplifying assumption that all processes are created (statically) in the initial state of the program[1]. We consider two functions $sort_{heap} : S \times G \to G$ and $sort_{proc} : S \to G$ that generate sorting permutations according to the ξ_{heap} and ξ_{proc} sorting criteria, respectively. Let us notice that $sort_{heap}$ now takes into account a process permutation in order to produce a canonical heap permutation. The state space search algorithm with combined reductions is presented in Figure 5.

```
RDFS(s, π)
if error(s) then report error fi
add(s, Statespace)
for each successor t of s do
    π' = sort_proc(t)
(#)ρ = sort_heap(t, π)
    t' = ρ_heap(π'_proc(t))
    if t' not in Statespace then RDFS(t', π' ∘ π)
od
end DFS
```

Fig. 5. Depth First Search Combining Heap and Process Symmetry

Informally, the search algorithm in Figure 5 keeps track of the process permutation resulting from the cumulative composition of all process permutations computed along every path within the quotient structure. Formally, let

[1] The extension of the algorithm to handle the dynamic creation of processes is considered as future work.

$\Pi(w,k) = \pi_0 \circ \pi_1 \circ \ldots \circ \pi_{k-1}$ where $w = s_0, s_1, \ldots, s_{k-1}$ and $\pi_i = sort_{proc}(s_i)$. Intuitively, $\Pi(w,k)$ gives the information needed to restore, in each state, the initial order of processes. It is easy to show that, in a recursive call to $RDFS$ in Figure 5 such that w is sequence of states passed as first parameter, $\Pi(w,|w|)$ represents the permutation passed as second parameter.

The implementation of the $sort_{heap}$ function uses a modified version of the heap sorting algorithm in Figure 4, in which the line marked by (*) has been changed into:

$$(*) \quad (c, s') = p(\pi^{-1}(i))$$

Here by π we denote the second argument in the invocation of $sort_{heap}$, as in the line marked with (#) in Figure 5. The idea is to use the inverse permutation in order to restore the original order of processes and maintain the canonical properties of the algorithm in Figure 4.

The following result gives sufficient conditions under which our combined algorithm still performs a canonical heap reduction. Let Id_n denote the identity permutation (neutral element) of G_n.

Lemma 2. *Let $\sigma = (s,(h,l),(p,i))$ and $\sigma' = (s',(h',l),(p',i))$ be two states such that $\sigma \equiv_{full} \sigma'$.*

1. *Let $\pi \in G_{\eta_{proc}(\sigma)}$ be a process permutation such that $p' = \pi_{proc}(p)$. Let $\zeta = sort_{heap}(\sigma, Id_{\eta_{proc}}(\sigma))$ and $\zeta' = sort_{heap}(\sigma', \pi)$ be two heap permutations computed by the algorithm in Figure 4 with the (*) modification. Then $\zeta_{heap}(\pi_{proc}(\sigma)) = \zeta'_{heap}(\sigma')$.*
2. *Let $w = \sigma_0, \sigma_1, \ldots, \sigma_k = \sigma$ and $w' = \sigma_0, \sigma'_1, \ldots, \sigma'_l = \sigma'$ be two paths. Let $\zeta = sort_{heap}(\sigma, \Pi(w,k))$ and $\zeta' = sort_{heap}(\sigma', \Pi(w',l))$ be two heap permutations computed by the algorithm in Figure 4 with the (*) modification. Then $\zeta(\sigma) \equiv_{proc} \zeta'(\sigma')$.*

Informally, Lemma 2 shows that using the algorithm in Figure 5 and computing heap permutations using the modified version of the algorithm in Figure 4 still preserves the properties of the original heap symmetry reduction, without process symmetry.

5 Symmetry versus Partial Order Reductions

In this section we investigate the relation between symmetry and partial order reductions applied to the model checking of dynamic programs that execute allocation statements. The previous work of Godefroid [11] also uses partial order information to detect symmetries between states, however it focuses mostly on flat programs, by defining permutations of actions and inferring that symmetric states are reached from the initial state by transition-symmetric paths. Our approach exploits the nature of dynamic programs that make use of the next-free allocation policy for which a semantics has been provided in Section 3. The notion of independence is extended via symmetry to define *symmetric independence*. It can be shown that paths differing only by a permutation of adjacent symmetric independent actions lead to symmetric states. In practice, this corresponds to the very common situation in which various interleavings of threads that perform heap allocations generate heap symmetric states. By conservatively exploiting this observation, when using partial order reductions in combination with symmetry reductions we can achieve better results when dynamic models of behavior are considered.

For the rest of this section, let $K = (S, R, L)$ be a Kripke structure over the set of actions Σ. An action α is said to be *enabled* in state s if there exists a state t such that $s \stackrel{\alpha}{\to} t$ in R. By $enabled_K(s)$ we denote the set of all actions enabled in s, according to the structure K. We can now introduce the concept of independent actions.

Definition 7. *A symmetric irreflexive relation $I \in \Sigma \times \Sigma$ is said to be an independence relation for K iff for all $(\alpha, \beta) \in I$ and for each $s \in S$ such that $\alpha, \beta \in enabled_K(s)$, we have:*

- *if $s \stackrel{\alpha}{\to} t$ then $\beta \in enabled_K(t)$*
- *if for some $s', s'' \in S$, $s \stackrel{\alpha}{\to} s' \stackrel{\beta}{\to} t$ and $s \stackrel{\beta}{\to} s'' \stackrel{\alpha}{\to} t'$, then $t = t'$.*

All partial order reduction algorithms [10], [22], [14] exploit (conservative under-approximations) of action independence. In practice, it has been shown that larger independence relations yield better partial order reductions. The contribution of this work to improving partial order reductions is based on defining and exploiting a weaker notion than the one from Definition 7.

Definition 8. *Given a symmetry relation $\equiv$ on S, a symmetric irreflexive relation $I_S \in \Sigma \times \Sigma$ is said to be a symmetric independence relation for K iff for all $(\alpha, \beta) \in I_S$ and for each $s \in S$ such that $\alpha, \beta \in enabled_K(s)$, we have:*

- *if $s \stackrel{\alpha}{\to} t$ then $\beta \in enabled_K(t)$*
- *if for some $s', s'' \in S$, $s \stackrel{\alpha}{\to} s' \stackrel{\beta}{\to} t$ and $s \stackrel{\beta}{\to} s'' \stackrel{\alpha}{\to} t'$, then $t \equiv t'$.*

The only change with respect to the Definition (7) is that, in I_S, two transitions are allowed to commute modulo symmetry. An independence relation is trivially a symmetric independence. Let us notice however that I_S can be much larger than I, since the number of states in a symmetry equivalence class can be exponential in the number of state components e.g., objects, processes. Dually, one can refer to the notion of *dependence*, which is defined as $D = \Sigma \times \Sigma \setminus I$. Similarly, we can define the notion of *symmetric dependence* as $D_S = \Sigma \times \Sigma \setminus I_S$. We can now formally relate the two notions of independence.

Lemma 3. *Given a symmetry relation $\equiv \subseteq S \times S$, I is a symmetric independence for K iff I is an independence for $K_{/\equiv}$.*

A second point of our discussion concerns visibility of actions. An action α is said to be *invisible* with respect to a set of atomic propositions $P \subseteq \mathcal{P}$ iff, for all $s, t \in S$ such that $s \stackrel{\alpha}{\rightarrow} t$ it is the case that $L(s) \cap P = L(t) \cap P$. Given the quotient structure $K_{/\equiv} = (S', R', L')$, by Definition (2) we have that $L(s) = L'([s])$ for each $s \in S$, therefore an action is invisible in K iff it is invisible in $K_{/\equiv}$.

(a)	(b)
DFS($[s]$)	DFS(s)
add($[s]$, Statespace)	add($rep(s)$, Statespace)
push($[s]$, Stack)	push($rep(s)$, Stack)
for each l in $ample_a([s])$ do	for each l in $ample_b(s)$ do
let $[t]$ such that $[s] \stackrel{l}{\rightarrow} [t]$	let t such that $s \stackrel{l}{\rightarrow} t$
if $[t] \notin$ Statespace DFS($[t]$) fi	if $rep(t) \notin$ Statespace DFS(t) fi
od	od
pop(Stack)	pop(Stack)
end DFS	end DFS

Fig. 6. Depth First Search with Partial Order and Symmetry Reductions

The correctness result of this section is based on the main result of [7]: performing partial order reduction on an already built quotient structure yields the same structure as using an on-the-fly algorithm that combines both partial order and symmetry reduction. Figure 6 (a) shows a classical state space exploration algorithm with partial order reductions on the already built quotient structure $K_{/\equiv} = (S', R', L')$, while Figure 6 (b) depicts the changes done to the algorithm in order to use both partial order and symmetry reduction on-the-fly. Assume that $rep : S \rightarrow S$ is a canonical representative function (see Definition 4). We consider two functions $ample_a : S' \rightarrow \Sigma$ and $ample_b : S \rightarrow \Sigma$ that return, for a state s, a subset of the set of enabled actions in s for the quotient and original structures respectively, i.e., $ample_a(s) \subseteq enabled_{K_{/\equiv}}(s)$ and $ample_b(s) \subseteq enabled_K(s)$. In order for the reduction to be sound[2], $ample_a$ must satisfy the following requirements [4], for each state s:

- C0-a $ample_a([s]) \neq \emptyset \iff enabled_{K_{/\equiv}}([s]) \neq \emptyset$
- C1-a on every path that starts with $[s]$ in $K_{/\equiv}$, an action that is *dependent* on some action in $ample_a([s])$ cannot be taken before an action from $ample_a([s])$ is taken.
- C2-a if $ample_a([s]) \subset enabled_{K_{/\equiv}}([s])$ then every $\alpha \in ample_a([s])$ is invisible.
- C3-a if $ample_a([s]) \subset enabled_{K_{/\equiv}}([s])$ then for every $\alpha \in ample_a([s])$ such that $[s] \stackrel{\alpha}{\rightarrow} [t]$, then $[t] \notin Stack$ holds.

[2] Property preservation for partial order reductions uses the notion of *stuttering path equivalence*, a weaker notion than bisimulation. For more details, the interested reader is referred to [22]

In order to define the $ample_b$ function (used by the algorithm in Figure 6 (b)), we change conditions [C0-a] and [C2-a] into [C0-b], [C2-b] by syntactically replacing $ample_a$ with $ample_b$, $[s]$ with s and $K_{/\equiv}$ with K. Since K and $K_{/\equiv}$ are bisimilar, conditions [C0-a] and [C0-b] are actually equivalent. From the previous discussion concerning visibility of actions, we can conclude that also [C2-a] and [C2-b] are equivalent. The rules [C1-b] and [C3-b] are as follows:

C1-b on every path that starts with s in K, an action that is *symmetric dependent* on some action in $ample_b(s)$ cannot be taken before an action from $ample_b(s)$ is taken.

C3-b if $ample_b(s) \subset enabled_K(s)$ then for every $\alpha \in ample_b(s)$ such that $s \stackrel{\alpha}{\to} t$, then $rep(t) \notin Stack$ holds.

A consequence of Lemma (3) is that conditions [C1-a] and [C1-b] are equivalent. Equivalence of [C3-a] and [C3-b] can be shown as an invariant of the lockstep execution of the algorithms in Figure 6. The proof of the following theorem can be done between the lines of Theorem 19 in [7].

Theorem 2. *The state space explored by the algorithm (a) running on the quotient structure $K_{/\equiv}$ is isomorphic to the one explored by the algorithm (b) running on the original structure K.*

According to [22], partial order reduction preserves all formulas of the LTL_X (next-free LTL) logic. An algorithm for partial order reduction that preserves properties expressible in CTL^*_X can be found in [9]. As a consequence of this and Theorem 2, combining partial order with symmetry reductions will preserve all properties written as next-free temporal logic formulas.

Having discussed the correctness of our partial order reduction that uses directly symmetric independence, we need to identify actions (program statements) that are (globally) symmetric independent without inspecting the program executions described by K or the reduced structure $K_{/\equiv}$. The operational semantics defined in Section 3 comes into place now. In particular, we are interested by the rules (NEW1) and (NEW2) that define object allocator actions. It can be noticed that the first-free allocation policy used by both (NEW1) and (NEW2) actions is sufficient to obtain the second point of Definition (8). non-deterministic choices in our language. In the following, let `a` and `b` denote two distinct program variables.

Lemma 4. *Let $\sigma = (s, (h, l_k), (p, i))$ be a state and $\alpha = [\mathtt{a} = \mathtt{new}]$, $\beta = [\mathtt{b} = \mathtt{new}]$ be two actions whose semantics are described by either one of the rules (NEW1) or (NEW2). If $\sigma', \sigma'', \theta'$ and θ'' are states such that $\sigma \stackrel{\alpha}{\to} \sigma' \stackrel{\beta}{\to} \theta'$ and $\sigma \stackrel{\beta}{\to} \sigma'' \stackrel{\alpha}{\to} \theta''$, then $\theta' \equiv_{heap} \theta''$.*

In order to meet the first requirement of Definition (8), one can take the classical [14] approach of defining *safe* actions. A safe action belonging to a process $p(i)$ is globally independent from all actions belonging to other processes $p(j)$ $(i \neq j)$ and invisible with respect to the set of atomic propositions that occur in a property expressible as a temporal logic formula. Both requirements are met by actions `x = new` where `x` is a local variable, in cases where the property refers only to global variables. Otherwise, static analysis can be used to compute a conservative over-approximation of the set of aliases in the program and consequently, conservatively under-approximate the set of safe actions enabled in a state.

To conclude, we have shown how the concept of symmetry can be used to extend the notion of independence used by partial order reductions. Identifying symmetric independent actions can be done by a syntactic analysis of the program and using them in the model checking algorithm may result in a better partial order reduction. As a remark, none of the two reduction techniques considered in this paper can fully replace the other. Since it uses a conservative under-estimation of the symmetric independence relation, partial order reduction might not always detect all symmetric states caused by different interleavings of allocators. Symmetry reduction is therefore needed in order to deal with these cases. On the other hand, it has been shown that partial order reduction that uses symmetric independence is equivalent to performing classical partial order reduction on an already built quotient structure, the result being a subset of the quotient structure that preserves meaningful properties. In cases where all symmetrical states are generated by interleavings of safe allocator actions, partial order reductions alone can actually outperform symmetry reductions.

6 Implementation and Experience

The heap symmetry and partial order reductions with symmetric independence have been implemented in the dSPIN model checker [17]. We performed experiments involving two test cases: the first one is a model of an ordered list shared between multiple updater threads, and the second models an interlocking protocol used for controlling concurrent access to a shared B-tree structure. Both models are verified for absence of deadlocks, as we performed these tests mainly to assess the effectiveness of our reduction techniques.

dSPIN is an automata theoretic explicit-state model checker designed for the verification of software. It provides a number of novel features on top of standard SPIN's [13] state space reduction algorithms, e.g., partial-order reduction and state compression. The input language of dSPIN is a dialect of the PROMELA language [13] offering, C-like constructs for allocating and referencing dynamic data structures. On-the-fly garbage collection is also supported [18]. The presence of garbage collector algorithms in dSPIN made the implementation of heap symmetry reductions particularly easy. The algorithm used to compute sorting permutations is in fact an instrumented *mark and sweep* garbage collector. The explicit representation of states allowed the embedding of such capabilities directly into the model checker's core. This served to bridge the semantic gap between high-level object oriented languages, such as Java or C++, and formal description languages that use abstract representations of systems, such as finite-state automata.

The first test case represents a dynamic list ordered by node keys. The list is updated by two processes that use node locks to synchronize: an inserter that adds given keys into the list, and an extractor that removes nodes with given keys from the list. The example scales in the maximum length of the list (L).

The second example is an interlocking protocol that ensures the consistency of a B-tree* data structure accessed concurrently by a variable number of replicated updater processes. Various mutual exclusion protocols for accessing concurrent B-tree* structures are described in [1] and our example has been inspired by this work. The example scales in the number of updater processes (N), B-tree order (K) and maximum depth of the structure (D).

Table 1. Experimental Results

i. Ordered List Example

L	SI+SR	SR	SI	-
8	296159	296159	766297	766297
9	727714	727714	2.29669e+06	2.29669e+06
10	1.75287e+06	1.75287e+06	4.62012e+06	4.62012e+06

ii. B-Tree* Example

N, K, D	SI+SR	SR	SI	-
2, 2, 3	1259	6816	1259	94105
2, 4, 3	3027	18773	3027	766842
2, 4, 4	32998	142371	out of memory	out of memory

Symmetries arise in both examples because different interleavings of the updater processes cause different allocation orderings of nodes with the same data. The results of our experiments are shown in Table 1. The table shows the number of states generated by the model checker with standard partial order reduction only (-), with partial order based on symmetric independence only (SI), with symmetry reductions only (SR) and with combined partial order and symmetry reductions (SI+SR).

In the first example (Ordered List) partial order reductions using symmetric independence do not contribute to the overall reduction of the state space. The reason is that the allocator statements in this model handle only global variables, being therefore labeled as “unsafe” by the dSPIN transition table constructor. On the contrary, in first two instances of the second example (Btree*) partial order reductions using symmetric independence manage to detect all heap symmetries arising as result of interleaving allocators, therefore symmetry reductions do not contribute any further to the overall reduction. The results show that combining partial order with symmetry reductions can outperform each reduction technique applied in isolation.

7 Conclusions

In this work, we have tackled issues related to the application of model checking techniques to software verification. Programs written in high-level programming languages have a more dynamic nature than hardware and network protocols. The size of a program state is no longer constant, as new components are added along executions. We have formalized this matter by means of semantic definitions of program states and actions. This semantics allows definition of various symmetry criteria for programs. We gave such criteria formal definitions, and described algorithms for on-the-fly symmetry reductions in automata theoretic model checking. In particular, we have discussed the combination of two orthogonal symmetry reductions, related to heap objects and processes. We have also shown how our heap symmetry reduction technique relates with partial order reductions. The emphasis is on how to adapt existing state space reduction techniques to software model checking. The ideas in this paper have been implemented in a software model checker that extends SPIN with dynamic features. Using this prototype implementation, a number of experiments have been performed. Preliminary results are encouraging, making us optimistic about the role symmetry and partial order reductions can play in enhancing model checking techniques for software.

Acknowledgments The author wishes to thank Dennis Dams, Dragan Bosnacki and Willem Visser, for interesting discussions and suggestions, Matthew Dwyer and John Hatcliff for valuable corrections of the previous versions of this paper, and the anonymous referees for comments leading to the discovery of an erroneous claim in the submission draft.

References

1. R. Bayer and M. Schkolnick: Concurrency of Operations on B-Trees. Acta Informatica, Vol. 9 (1977) 1–21
2. Dragan Bosnacki: Enhancing State Space Reduction Techniques for Model Checking. PhD Thesis, Technical University of Eindhoven (2001)
3. Edmund M. Clarke, Somesh Jha, Reinhard Enders and Thomas Filkorn: Exploiting Symmetry In Temporal Logic Model Checking. Formal Methods in System Design, Vol.9, No. 1/2 (1996) 77–104
4. Edmund M. Clarke, Orna Grumberg and Doron Peled: Model Checking. MIT Press (2001)
5. Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper and Mihalis Yannakakis: Memory-Efficient Algorithms for the Verification of Temporal Properties. Formal Methods in System Design, Vol. 1, No 2/3 (1992) 275–288
6. Dennis Dams, Dragan Bosnacki and Leszek Holenderski: A Heuristic for Symmetry Reductions with Scalarsets. Proc. Formal Methods Europe (2001) 518–533
7. E. Emerson, S. Jha and D. Peled: Combining Partial Order and Symmetry Reductions. Proc. Tools and Algorithms for Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1217 (1997) 19–34

8. E. Emerson and A. P. Sistla: Symmetry and Model Checking. Formal Methods in System Design, Vol.9, No. 1/2 (1996) 105–131
9. R. Gerth, R. Kuiper, D. Peled and W. Penczek: A Partial Order Approach to Branching Time Logic Model Checking. Proc. 3rd Israel Symposium on Theory on Computing and Systems (1995) 130–139
10. P. Godefroid: Partial-Order Methods for the Verification of Concurrent Systems. Lecture Notes in Computer Science Vol. 1032 (1996)
11. P. Godefroid: Exploiting Symmetry when Model-Checking Software. Proc. Formal Methods for Protocol Engineering and Distributed Systems (FORTE/PSTV) (1999) 257–275
12. M. Hennessy and R. Milner: Algebraic Laws for Nondeterminism and Concurrency. Journal of the ACM Vol. 32 (1985) 137–161
13. G.J. Holzmann: The SPIN Model Checker. IEEE Trans. on Software Engineering Vol. 23 (1997) 279–295
14. G. J. Holzmann and D. Peled: An Improvement in Formal Verification. Formal Description Techniques, Chapman & Hall, (1994) 197–211
15. G. Holzmann, D. Peled and M. Yannakakis: On Nested Depth First Search. Proc. 2nd SPIN Workshop (1996)
16. R. Iosif: Symmetric Model Checking for Object-Based Programs. Technical Report KSU CIS TR 2001-5 (2001)
17. R. Iosif and R. Sisto: dSPIN: A Dynamic Extension of SPIN. Proc. 6th SPIN Workshop, Lecture Notes in Computer Science Vol. 1680 (1999) 261–276
18. R. Iosif and R. Sisto: Using Garbage Collection in Model Checking. Proc. 7th SPIN Workshop, Lecture Notes in Computer Science Vol. 1885 (2000) 20–33
19. R. Iosif: Exploiting Heap Symmetries in Explicit-State Model Checking of Software. Proc. 16th IEEE Conference on Automated Software Engineering (2001) 254 – 261
20. C. Ip and D. Dill: Better Verification Through Symmetry. Formal Methods in System Design, Vol.9, No. 1/2 (1996) 41–75
21. F. Lerda and W. Visser: Addressing Dynamic Issues of Program Model Checking. Proc. 8th SPIN Workshop, Lecture Notes in Computer Science Vol. 2057 (2001) 80–102
22. D. Peled: All from One, One from All: on Model Checking using representatives. Proc. 5th Conference on Computer Aided Verification, Lecture Notes in Computer Science Vol. 697 (1993) 409–423