

# Using State Symmetries to Speed up Symmetry Reduction in Model Checking

Christian Appold

Chair of Computer Science V  
University of Würzburg  
appold@informatik.uni-wuerzburg.de

## Abstract

Concurrent systems with many replicated components often exhibit a lot of symmetries. When using model checking to verify such systems, this leads to a redundant search over equivalent parts of the state-space. Verification can even be intractable for concurrent systems due to the state-space explosion problem which model checking suffers from. Considerable memory savings can be achieved by using symmetry reduction when verifying systems which contain symmetries. However, sometimes verification time can significantly increase with symmetry reduction because of the time required to compute representatives of equivalent states. Therefore we investigate the efficient use of *state symmetries*. The use of state symmetries aims to avoid redundant computations of representatives.

## 1 Introduction

Today the presence of concurrent systems is steadily increasing, for example with the growing dispersion of multi-core CPUs or upcoming technologies like sensor networks. With the growing importance of concurrent systems, the need for reliable methods for their verification increases. Currently mostly simulation is used in industry to verify the correctness of systems. But it is known that especially complex errors in distributed systems are hard to find by simulation. A technique which exhaustively examines the state-space of a system is temporal logic model checking [Clarke and Emerson, 1982], [Queille and Sifakis, 1982]. Model checking is an automated formal verification technique, where properties are formulated in a temporal logic (e.g. CTL [Ben-Ari *et al.*, 1981] or LTL [Pnueli, 1981]). A problem, which model checking suffers from is the state-space explosion problem. This especially appears in the verification of concurrent systems, where the state-space grows exponentially with the number of components. Therefore symmetry reduction techniques [Ip and Dill, 1996] have been developed to diminish this problem. They try to exploit symmetries, which often exist in concurrent systems. It has been shown that in model checking of concurrent systems which exhibit a lot of symmetries often significant memory savings can be achieved (see e.g. [Ip and Dill, 1993]).

Symmetry reduction techniques in verification of concurrent systems generally exploit symmetries by restricting state-space search to representatives of equivalence classes. The calculation of the equivalence class representatives is central to all model checking methods which use symmetry reduction. It is known that for arbitrary symmetries their computation is a hard and time-consuming problem. But it has been shown that for certain classes of symmetries this problem can be solved efficiently. Although the state-space can be reduced considerably by using symmetry reduction, their usage can lead to a significant increase in runtime.

In constraint satisfaction problems and propositional satisfiability problems, symmetry breaking is used to gain computational savings by restricting the search space in the presence of symmetries. These symmetries can be exploited by introducing symmetry breaking constraints. These circumvent redundant variable assignments through symmetries. Ideally they restrict the search space to exactly one member of each class of symmetric variable assignments. Sometimes exponential savings in time can be achieved. Symmetry breaking in connection with problems represented in propositional logic has been investigated in [Crawford *et al.*, 1996]. In model checking *state symmetries* are used to break the execution of symmetric transitions in a global state of a concurrent system. Whereas symmetry breaking in constraint satisfaction problems aims to avoid redundant variable assignments, the usage of state symmetries circumvents redundant calculations of the same state equivalence class representative. They have been first introduced in [Emerson and Sistla, 1996], and [Gyuris *et al.*, 1997] integrated their use into a model checking algorithm. Especially in systems with many replicated components exploiting state symmetries can lead to big runtime savings.

SMC (Symmetry based Model Checker) [Sistla *et al.*, 1999] is a model checker which implements symmetry reduction and exploits state symmetries. In [Sistla *et al.*, 1999] the authors presented experimental results about the usage of state symmetries. But they presented just a few experimental results and have given no advice how to extend their usefulness for further improvements. To our knowledge there is no other work where state symmetries have been investigated experimentally.

Therefore we implemented the use of state symmetries for the model checker Murphi ([Dill *et al.*, 1992] and [Ip and

Dill, 1996]). Its advantage is that its input language allows the efficient use of state symmetries for a broader range of verification models. With verification experiments we tested achievable runtime improvements and developed and examined several extensions regarding the use of state symmetries.

The paper is organized as follows. In the next Section we present related work about using symmetries. Section 3 gives a short introduction to model checking, before symmetry reduction in model checking is discussed in Section 4. In Section 5 we present details about the explicit-state model checker Murphi, which we used for our verification experiments. Thereafter we present our enhancements to the use of state symmetries in Section 6 and then we discuss experimental results in Section 7. The paper closes with a conclusion and an outlook.

## 2 Related Work

To our knowledge the only work which presented experimental results about the potential of exploiting state symmetries was [Sistla *et al.*, 1999]. In this paper the model checker SMC for verifying concurrent systems has been presented. It is able to use symmetry reduction and state symmetries. The authors presented little experimental results about the possibilities of state symmetries and extensions haven't been discussed. Also the input language of SMC doesn't allow the specification of program variables which can store component identities directly. To verify algorithms like the peterson mutual exclusion protocol (see subsection 7.3), which needs arrays of global variables with component identities as values, a complicated modeling by a multi-dimensional array is necessary. This increases the size of a global system state and also the runtime when using state symmetries can significantly rise. In contrast, Murphi's input language hasn't these restrictions. Also with SMC only exploitation of state symmetries for all system states or no system state is possible. We discuss extensions to state symmetries and their benefits in our work. It can be shown that they can lead to remarkable runtime improvements.

Using state symmetries in model checking is similar to symmetry breaking in constraint satisfaction problems. Therefore the knowledge of details of both techniques can possibly be used to develop further improvements for model checking and constraint satisfaction problems. Symmetry breaking with regard to model checking has only been investigated for model checking by using SAT solvers. In [Rintanen, 2003], the authors treated the handling of symmetries in representations of transition systems in propositional logic. They stated that symmetry breaking constraints used in constraint satisfaction problems and satisfiability testing only remove all the symmetry at one point in time. They couldn't be generalized directly to symmetries in sequences of states. To enable the use of symmetry breaking in SAT representations of transition systems they order sequences of transitions and allow only one sequence from each class of symmetrical transition sequences. The authors state that removing of symmetries from transition sequences in many cases also removes all symmetries from the sequences of states which these sequences generate. Their concept to restrict the feasibility of

transitions is similar to the concept of exploiting state symmetries.

A method to add symmetry breaking predicates to problems formulated in propositional logic has been presented in [Crawford *et al.*, 1996]. Their approach works as a preprocessor to any satisfiability algorithm and modifies the problem which has to be solved by adding symmetry breaking predicates. They are satisfied by exactly one member of each symmetric point of the search space. Because the computation of the full symmetry breaking predicates seems intractable, they investigated the usage of partial symmetry breaking predicates. In several cases symmetries can be broken either fully or partially by using a polynomial number of symmetry breaking predicates.

## 3 Model Checking

Model checking [Clarke and Emerson, 1982], [Queille and Sifakis, 1982] is an automatic technique for verifying finite state concurrent systems. Given a finite state model describing the behavior of a system and a property, a model checker determines if the property is satisfied by the model. The finite state model of a system is usually described in the form of a *Kripke structure*.

**Definition 1** Let  $AP$  be a finite set of atomic propositions. A *Kripke structure*  $M$  over  $AP$  is a quadruple  $M = (S, R, L, S_0)$ , with the following components:

- $S$  is a nonempty, finite set of states,
- $R \subseteq S \times S$  is the transition relation,
- $L : S \rightarrow 2^{AP}$  is a function, which maps each state in  $S$  with the set of atomic propositions which are true in that state and
- $S_0 \subseteq S$  is the set of initial states.

A path in  $M$  is a nonempty sequence of states  $\pi = (s_0, s_1, \dots)$ , such that  $s_0 \in S_0$  and  $\forall i \geq 0, (s_i, s_{i+1}) \in R$ . If a path is finite, the length  $|\pi|$  of a path is the number of its transitions and  $i$  is restricted to  $|\pi| - 1$ . A path of a single state has length zero. A state  $t \in S$  is reachable in  $M$  if there is a path in  $M$  from some initial state to  $t$ . Properties usually can be specified in a temporal logic. Examples of temporal logics are CTL and LTL, which are sublogics of the temporal logic CTL\* [Emerson and Halpern, 1986]. Temporal logics extend propositional logic with temporal operators. These allow to specify properties like: eventually some designated state has to be reached, or that some atomic proposition has to be true in every state of a path.

## 4 Symmetry Reduction in Temporal Logic Model Checking

This section gives an introduction to symmetries in model checking. For further information about the use of symmetries in model checking see e.g. [Miller *et al.*, 2006] or [Clarke *et al.*, 1996].

### 4.1 Symmetry Reduction

Symmetries of a Kripke structure form a group.

**Definition 2** A group is a non-empty set  $G$  together with a binary operation  $\circ : G \times G \rightarrow G$  which satisfies:

- For all  $\alpha, \beta, \gamma \in G$ ,  $\alpha \circ (\beta \circ \gamma) = (\alpha \circ \beta) \circ \gamma$ .
- There is an identity element  $e \in G$ , such that for all  $\alpha \in G$ ,  $\alpha \circ e = \alpha = e \circ \alpha$ .
- For all  $\alpha \in G$  there is an inverse element  $\alpha^{-1} \in G$  such that  $\alpha \circ \alpha^{-1} = \alpha^{-1} \circ \alpha = e$ .

Permutations are used to define symmetries of a Kripke structure. Given a non-empty set  $X$ , a permutation of  $X$  is a bijection  $\alpha : X \rightarrow X$ . We extend  $\alpha$  to a mapping  $\alpha : R \rightarrow R$  on the transition level of a Kripke structure by defining  $\alpha((s, t)) = (\alpha(s), \alpha(t))$ .

**Definition 3** A permutation  $\alpha$  on  $S$  is said to be a **symmetry** of a Kripke structure  $M = (S, R, L, S_0)$ , if:

- $R$  is invariant under  $\alpha : \alpha(R) = R$ ,
- $L$  is invariant under  $\alpha : L(s) = L(\alpha(s))$  for any  $s \in S$ , and
- $S_0$  is invariant under  $\alpha : \alpha(S_0) = S_0$ .

The symmetries of  $M$  form a group under function composition. A model  $M$  is said to be **symmetric**, if its symmetry group  $G$  is non-trivial (i.e. does not consist only of the identity permutation).

A state  $(\vec{g}, l_1, \dots, l_n)$  in a concurrent system with  $n$  components consists of the values  $\vec{g}$  of all global variables (not associated with any process) and the *local state*  $l_i$  of each process  $i \in \{1, \dots, n\}$  (values of all local variables of process  $i$ ). A symmetry  $\alpha$  is derived from a permutation on  $\{1, \dots, n\}$  and acts on a state  $s = (\vec{g}, l_1, \dots, l_n)$  as  $\alpha(s) = (\vec{g}^\alpha, l_{\alpha(1)}, \dots, l_{\alpha(n)})$ . The local states of the processes are permuted by permuting their positions in the state vector. Further,  $\alpha$  acts on  $\vec{g}$  by acting component-wise on each global variable  $g$ . The action of  $\alpha$  on  $g$  depends on the nature of  $g$ , for more details see [Emerson and Wahl, 2003].

A group  $G$  of symmetries induces an equivalence relation  $\equiv_G$  on the states of  $M$  by the rule  $s \equiv_G t \Leftrightarrow s = \alpha(t)$  for some  $\alpha \in G$ . The equivalence class under  $\equiv_G$  of a state  $s \in S$ , denoted  $[s]_G$ , is called the *orbit* of  $s$  under the action of  $G$ . Observe that  $s \equiv_G t$  implies  $L(s) = L(t)$ , since  $L$  is invariant under permutations of  $G$  (see Definition 3). The orbits can be used to construct a *quotient* Kripke structure  $M_G$ .

**Definition 4** The *quotient Kripke structure*  $M_G$  of  $M$  with respect to  $G$  is a quadruple  $M_G = (S_G, R_G, L_G, S_G^0)$  where:

- $S_G = \{[s]_G : s \in S\}$  (the set of orbits of  $S$  under the action of  $G$ ),
- $R_G = \{([s]_G, [t]_G) : (s, t) \in R\}$ ,
- $L_G([s]_G) = L(\text{rep}_G([s]_G))$  (where  $\text{rep}_G([s]_G)$  is a unique representative of  $[s]_G$ ),
- $S_G^0 = \{[s]_G : s \in S_0\}$  (the orbits of the initial states  $S_0$  under the action of  $G$ ).

The quotient structure  $M_G$  is smaller than  $M$ , if  $G$  is non-trivial. For any  $s$ , the size of  $[s]_G$  is bounded by  $|G|$ , and so the theoretical minimum size of  $S_G$  is  $|S|/|G|$ . In highly

symmetric systems we may have  $|G| = n!$ , where  $n$  is the number of components. It has been shown that  $M$  and  $M_G$  are equivalent in the sense that they satisfy the same set of logic properties which are invariant under permutations of  $G$ . A proof of the following theorem can be found in [Clarke et al., 1996].

**Theorem 1** Let  $M = (S, R, L, S_0)$  be a Kripke Structure,  $G$  be a symmetry group of  $M$ , and  $h$  be a CTL\* formula. If  $h$  is invariant under the group  $G$ , then

$$M, s \models h \Leftrightarrow M_G, [s]_G \models h \quad (1)$$

where  $M_G$  is the quotient structure corresponding to  $M$ .

As a consequence, by choosing a suitable symmetry group  $G$ , model checking can be done using  $M_G$  instead of  $M$ , which often leads to considerable savings in memory (see e.g. [Clarke et al., 1996]).

## 4.2 Quotient structure exploration

In this subsection we present an algorithm which explores a quotient Kripke structure. Algorithm 1 shows the search algorithm used by the model checker Murphi to generate the quotient Kripke structure [Ip and Dill, 1996].

```

reached = {rep_G(s) : s in S_0}
unexplored = {rep_G(s) : s in S_0}
while unexplored != empty do
  remove a state s from unexplored
  for all successor states q of s do
    if rep_G(q) is not in reached then
      add rep_G(q) to reached
      add rep_G(q) to unexplored
    end if
  end for
end while

```

Algorithm 1: Exploration of the quotient Kripke structure

In Algorithm 1,  $\text{rep}_G(s)$  denotes the unique representative of the equivalence class of state  $s$  of the Kripke structure. The calculation of the unique representative for an orbit  $[s]_G$  requires a canonicalization function. Unique representatives for the orbits are necessary to decide, if a state which is symmetric to a newly discovered state has already been reached during the exploration of the Kripke structure. The calculation of the representatives is one of the most important operations in connection with symmetry reduction and is known as the *orbit problem*. Under arbitrary symmetries, it has been shown that the orbit problem is as hard as the *graph isomorphism problem* [Clarke et al., 1998], for which no polynomial-time algorithm is known. But despite this discouraging result, it has been shown that this problem can be solved efficiently for certain classes of symmetry groups. A convenient approach is to use the lexicographically least element in an orbit as the unique representative. Also approximate solutions, like e.g. using multiple representatives ([Miller et al., 2006]), have been developed to diminish this problem. The techniques the model checker Murphi offers to solve this problem are described in 5.2. Algorithm 1 checks in the for-loop for every newly explored successor of a state, if its canonical representative has been visited before. If a representative hasn't

been visited before, it adds the representative to the visited and the unexplored states and explores its successors later. Because Murphi doesn't allow to specify properties in a temporal logic, properties expressible in Murphi (see subsection 5.1) can be verified on-the-fly for every state of the quotient structure during execution of Algorithm 1. This means, when a new state is found, it is verified if the specified properties are fulfilled for this state. Thereby an error can be reported as soon as it has been detected and exhaustive exploration and generation of the quotient Kripke structure in such cases isn't necessary.

### 4.3 State Symmetry

Another type of symmetry which can be used in model checking of concurrent systems with replicated components is state symmetry (see [Gyuris *et al.*, 1997] and [Sistla *et al.*, 1999]). Beneath heuristics for canonicalization which sometimes lead to more than one representative for a state equivalence class (see also subsection 4.2), state symmetries can be used to reduce the runtime requirements of canonicalization. Especially in systems with many components, their use can lead to high savings of runtime.

State symmetries use the internal symmetries of a single global state of the Kripke structure. In [Sistla *et al.*, 1999] two components  $i$  and  $j$  are said to be *equivalent in a global state  $s$* , if  $\pi_{i,j}(s) = s$ , where  $\pi_{i,j}$  is the permutation that interchanges  $i$  and  $j$  but fixes all other components. This relation is an equivalence relation among the components, which induces a partition on the set of components and is called the *state symmetry partition of  $s$* . All components in the same local state and to which the described partition can be applied and leads to  $\pi_{i,j}(s) = s$  are in the same group of this partition. It is only necessary to execute every enabled transition for one representative component of every group of a *state symmetry partition of  $s$* . Executable transitions from other components of the partition group don't have to be executed, because the same transitions are executable for each component of a group and the representatives of the successor states of the components of a group are the same. Thus the computation overhead of redundant representative calculations can be avoided. Utilization of this type of state symmetry has been implemented in SMC. [Emerson and Sistla, 1996] have proposed a less restrictive notion of state symmetry, which doesn't consider just permutations that interchange local states of components. If necessary they regard all permutations including affiliated permutations of values of global variables to find state symmetries.

### 4.4 Symmetry Detection

To diminish the state-space explosion problem in the presence of symmetries, the symmetries first have to be detected. One approach to find symmetries is to build a Kripke structure  $M$  and then to find a symmetry group of  $M$  by using standard algorithms for graph automorphisms (e.g. by using the tool *nauty* [McKay, 2007]). The symmetry group then could be used to build the quotient Kripke structure  $M_G$ . But the biggest advantage when using symmetry reduction techniques in model checking is that they allow to check a symmetry reduced model, even if the unreduced model would be

intractably large. Thus for exploiting the possibilities of symmetry reduction, methods to find symmetries without building the Kripke structure  $M$  explicitly are needed.

One Method to simplify symmetry detection is to restrict the input specification language of a model checker to guarantee full symmetry between processes of the same type. This approach has been used in the input language of the SMC tool. In its input language symmetry is indicated by declaring multiple instances of a given module type. The corresponding group of permutations can then be determined easily. Another approach to simplify symmetry detection is to introduce a new data type. The input language of Murphi has been extended with the data type *scalarset* [Ip and Dill, 1996] to facilitate the detection of symmetries. A scalarset is an integer subrange with restricted operations. The restrictions are necessary to ensure that consistent permutation of scalarset variables in all states corresponds to an automorphism of the state-space. Scalarset variables support assignment, array indexing and testing for equality and inequality. They don't support arithmetic operations and comparisons other than equality or inequality testing. The permutation process can be summarized as follows:

- By applying a permutation to a scalarset variable, the value of the variable is modified to the corresponding permuted value.
- When an array indexed by a scalarset variable is permuted, the contents of the array elements are permuted and the elements are rearranged according to the permutation.

## 5 The Model Checker Murphi

For our experiments we used the explicit-state model checker Murphi [Dill *et al.*, 1992], which has been extended in [Ip and Dill, 1996] to detect and exploit symmetries.

### 5.1 Properties of Murphi

Murphi is an explicit-state model checker with its own high-level input language. The input language has been designed to define finite-state asynchronous concurrent systems. Murphi possesses a compiler, which generates a C++ program from an input description and code for a verifier. Verification runs can be done by compiling and executing the generated C++ program. The behavior of a system can be described with the input language by using a set of rules, which are guarded commands. They consist of a condition and an action. The condition is a boolean expression consisting of constants, declared variables and operators. If the condition of a rule is satisfied in a state, the corresponding action can be executed. An action thereby is a sequence of statements, which update the current values of the state variables of the system. If conditions of more than one rule are satisfied in a state, one rule is chosen non-deterministically for execution. Therewith the input language of Murphi is well-suited for an asynchronous, interleaving model of concurrency. If more than one process has been specified, a process can execute any number of rules before a rule for another process is executed.

Murphi is able to detect deadlocks and to check invariants. Invariants are boolean expressions, which have to be true in every state of a system. They are specified by the user in the input language of Murphi. If an invariant is violated, Murphi generates an error trace that can be used for debugging. Additionally Murphi offers *Error* and *Assert* statements to detect design errors. It doesn't allow the expression of properties of sequential behavior directly in a temporal logic. Because the temporal logic CTL\* allows to specify a superset of the properties which can be specified for Murphi, Theorem 1 (see subsection 4.1) is also valid for Murphi. During verification the verifier exhaustively examines all states of a system. This can be done with a breadth-first search or a depth-first search. The user can select which of these search algorithms Murphi has to use. Before symmetries can be exploited in model checking, they first need to be detected. To facilitate the detection of symmetries, the Murphi input language has been extended with the datatype *scalarset* (see subsection 4.4 and [Ip and Dill, 1996]).

## 5.2 Representative Calculation with Murphi

For efficiently computing the unique representatives of the orbits Murphi offers *canonicalization* and *normalization* algorithms. Canonicalization algorithms basically compute unique representatives for the orbits, whereas normalization algorithms sometimes use more than one representative for an orbit. Because we want to examine the effect of state symmetry on verification time by using as much state-space savings as possible, we limited our experiments to the canonicalization algorithms of Murphi. Murphi has two canonicalization algorithms. One which exhaustively generates all permutations of a global state (denoted *sym1* in section 7) and then chooses the lexicographically smallest state as the canonical representative. The other canonicalization algorithm first heuristically restricts the set of permutations it applies. During canonicalization it exhaustively applies all these permutations and also chooses the lexicographically smallest generated state (denoted *sym2* in section 7).

## 6 Extending the usage of State Symmetries

In this section we describe how the use of state symmetries can be optimized and extended in explicit-state model checking. Experimental results about runtime changes through using the extensions can be found in section 7.

### 6.1 Model Checker improvements

The runtime improvements achieved by using state symmetries can be further increased by optimizing the implementation of the model checker. As mentioned in subsection 5 the behavior of an input model for Murphi is described by a set of rules. During state-space exploration Murphi tests the feasibility of all rules of the model. If a rule is feasible it is executed and the representative of the corresponding next state is generated. When using state symmetries only one representative of every rule which leads to an equivalent next state has to be executed. We inserted the test for state symmetries after the feasibility test of the rules. This has the big advantage that state symmetries only need to be computed between components which can execute the same type of rules

in a state. For example if in a global state all components can execute only different types of rules, no state symmetries will be computed. To compute the state symmetries it is necessary to calculate equivalence classes of components which generate the same new states. This step can consume a considerable amount of runtime, particularly if the components contain many local states and also global variables have to be regarded. Therefore, if state symmetries are computed before regarding if the same rules are enabled for components, this often can lead to a big runtime increase if the computation is complex.

When testing a rule type for feasibility, Murphi examines for every component if the rule type is enabled. In systems with a lot of components the overhead for testing all components for feasibility of a rule often can be avoided if the representatives of global states are sorted lexicographically by considering feasibility conditions of the rules. When verifying software, the *program counter* has a special role. The feasibility of rules which can be executed for components of a concurrent software system always depends on the current value of the program counter of the component. Therefore if the program counter has been taken as the sorting criterion in lexicographically sorted states, the feasibility test often can be broken. It can be broken for a rule when a component has been reached in the global state-vector, whose program counter has the value which is needed for feasibility of the rule and if the next local component state in the state-vector has another program counter value. Then no local component state in the state-vector which follows has a program counter value which is required for feasibility of the rule and testing for feasibility can be broken. Especially if a system contains many components, this can lead to large runtime reductions.

### 6.2 Exploiting particularities of the verification task

Another optimization to the use of state symmetries is the consideration of information about the verification model. If one can decide through analysis of the verification model that never or only in a few cases state symmetries can be used for a rule, the insertion of the state symmetry test can be avoided for that rule. Therewith the overhead to detect them if they can never be used is eliminated. For the peterson mutual exclusion algorithm, which gives access to the critical section in a series of competitions (see subsection 7.3), for example, we avoided the insertion of state symmetries for components which are in the upper rounds of the tournament. It's expected that state symmetries can be used for them only in a few cases. Even if the algorithm contains failures, there should not be much possibilities to use state symmetries in these states. Therefore the runtime of predominantly useless computing of state symmetries sometimes can be avoided through analysis of the verification model.

Examination of global states can be avoided completely if they satisfy the properties we want to check, or another similar state that possesses the necessary behavior of the state is surely visited. Information about such states can be obtained from an analysis of the input model. As our experiments have shown the usage of such information promises a great potential for runtime reductions. Beneath the canonicalization

overhead and the overhead for computing state symmetries also the visiting of irrelevant states for the verification tasks can be avoided. Then no memory is required to store such visited states. It's only necessary to ensure that the model doesn't lose any behaviors which are relevant for the verification task. A technique which already is used in model checking of concurrent systems and aims to avoid the visiting of sequences of states which are equivalent up to reordering of transitions is *partial order reduction*. The combination of symmetry reduction and partial order reduction has been discussed in [Emerson *et al.*, 1997]. But we couldn't find any experimental results about the combination of symmetry reduction, the use of state symmetries and techniques to avoid the visiting of states which are irrelevant for the verification task. Therefore we made some verification experiments whose results are presented in the next section. We have circumvented the visiting of states in our experiments only where this could have been decided by simplest static analysis. Nevertheless, this already leads to large runtime decreases and even lessened the memory requirements.

## 7 Experimental Evaluation

In this section we present the results of our verification experiments. The mentioned runtimes are averages of multiple verification runs. We have done the experiments on a computer with an Intel Pentium Core 2 CPU with 2.4 GHz and 3 GB main memory by using a single core. As operating system we used Debian 4.0. Breadth-first search of Murphi has been used as state-space search strategy for our results presented here, but verification runs using depth-first search lead to similar results. Our verification runs with symmetry reduction have been made with the Murphi canonicalization algorithms *sym1* and *sym2* (see also subsection 5.2). Algorithm *sym1* thereby is restricted in Murphi to a system size of 9 components, if the system to verify consists only of components of the same type. For the verification example of subsection 7.2 which contains two different component types, *sym1* could only be used for at maximum 6 components of each type. Therefore, for our experiments with a higher number of components, we could use only *sym2*. In the column *canonicalization algorithm* of the tables with the experimental results, beneath the algorithm used for canonicalization of the representatives, it is denoted which symmetries have been used in the verification run. There *without symmetry* means that no symmetry reduction has been used and *without sSym* means that only symmetry reduction but no state symmetries have been used. *sSym* stands for the usage of state symmetries in addition to symmetry reduction and *sSym + ext* says that additionally extensions from subsection 6 have been used.

### 7.1 Simple Mutual Exclusion Algorithm

In this subsection we report about the verification of a simple mutual exclusion algorithm. The verification results can be seen in table 1. In this algorithm every component has only the three local states *non-critical*, *trying* and *critical*. There are state changes from *non-critical* to *trying* and from *critical* to *non-critical* which can be executed without restrictions. State changes from *trying* to *critical* are only allowed for a

number of components	canonicalization algorithm	number of states	time
9	without symmetry	2816	3 sec
	sym1 without sSym	19	31 sec
	sym1 sSym	19	10 sec
	sym1 sSym + ext	19	10 sec
	sym2 without sSym	19	1 sec
	sym2 sSym	19	1 sec
	sym2 sSym + ext	19	1 sec
200	without symmetry	mem ov	-
	sym2 without sSym	401	50 sec
	sym2 sSym	401	2 sec
	sym2 sSym + ext	401	2 sec
2000	without symmetry	mem ov	-
	sym2 without sSym	timeout	> 4 days
	sym2 sSym	7999	35 min
	sym2 sSym + ext	7999	28 min

Table 1: Verification results for the mutual exclusion example

component if currently no other component is in the state *critical*. The possible state changes of the algorithm show that the mutual exclusion property, which we used for our verification experiments with the algorithm, is always true. But because of the very small number of local states this algorithm suited very well to study the effect of using state symmetries and our enhancements for systems with a large number of components. As optimizations for the verification runs with *sSym + ext* in the table we used the improvement of breaking the feasibility test of rules.

The verification results show that even if we used only 9 components, a runtime reduction of two-thirds can be achieved for canonicalization algorithm *sym1* through using state symmetries. For 200 components and *sym2* one can see that a considerable decrease in runtime can be achieved by using state symmetries. The verification runs with 2000 components showed that additionally to the use of state symmetries our improvements lead to further runtime reductions. Therefore when verifying systems with a large number of components, using state symmetries greatly improves verification time. Nevertheless additional enhancements can lead to further runtime improvements. Due to ever increasing memory capacities and computing power and therewith the possibility to verify larger systems, such optimizations can be even more valuable in future.

### 7.2 Readers-Writers Problem

In the readers-writers problem there are multiple readers and writers, which share a common memory. It is allowed that multiple readers get access to the shared memory at the same time. If a writer has access to the shared memory, no reader and no other writer should have access to the shared memory. This has also been the property which we used for our verification experiments with this example. We have made verification runs with different numbers of components, whereas we always have chosen equal numbers of readers and writers. Our verification results are presented in table 2. In this testcase every reader has the three local states *idle*, *trying* and *reading*, while every writer has the local states *idle*, *trying*

number of readers/writers	canonicalization algorithm	number of states	time
6	without symmetry	58944	8 sec
	sym1 without sSym	238	542 sec
	sym1 sSym	238	194 sec
	sym1 sSym + ext	238	194 sec
	sym2 without sSym	238	3 sec
	sym2 sSym	238	3 sec
	sym2 sSym + ext	238	3 sec
	50	without symmetry	mem ov
sym2 without sSym		70176	770 sec
sym2 sSym		70176	54 sec
sym2 sSym + ext		70176	51 sec
150	without symmetry	mem ov	-
	sym2 without sSym	timeout	> 4 days
	sym2 sSym	1755526	498 min
	sym2 sSym + ext	1755526	483 min

Table 2: Verification results for the readers-writers problem

and *writing*. Both component types can always execute transitions from *idle* to *trying* and from *reading* and *writing* respectively to *idle*. Readers are allowed to execute the transition from *trying* to *reading* if currently no writer is in the state *writing*. Writers can change their state from *trying* to *writing* if no reader is in the state *reading* and no other writer is in the state *writing*. Because two different component types exist in the readers-writers problem, the states of the readers and the writers have to be canonicalized apart. Their canonicalized states are then combined to get a canonicalized global state of the system. Therefore the symmetry reduced state-space here consists of all combinations of canonicalized states of the readers and the writers which can occur. State symmetries can only be used for the readers and the writers separately. In our verification experiments with this example we used as enhancements, as in subsection 7.1, the improvement of breaking the feasibility test of rules for the testcases mentioned *sSym + ext* in table 2.

The results of our verification experiments with 6 readers and 6 writers show that when using symmetry reduction and canonicalization algorithm *sym1*, the verification time increases considerably in contrast to verification runs without symmetry reduction. But by using state symmetries a runtime reduction of nearly two-thirds can be achieved. When doing verification runs with 50 readers and 50 writers, the symmetry reduced state-space is large in comparison to the symmetry reduced state-spaces of subsection 7.1. This has been caused by the large number of combinations of canonicalized local states of the readers and writers which appear. Without symmetry reduction verification even hasn't been possible because of the high memory requirements. Despite state symmetries can only be used for the readers and writers separately, through the high number of symmetry reduced global states where state symmetries can be used, also big runtime improvements can be achieved. For verification runs with 150 readers and writers, the runtime reductions are very large when using state symmetries. Also runtime improve-

number of components	canonicalization algorithm	number of states	time
7	without symmetry	628868	34 sec
	sym1 without sSym	163298	24 min
	sym1 sSym	163298	20 min
	sym1 sSym + ext	149283	16 min
	sym2 without sSym	163298	16 sec
	sym2 sSym	163298	14 sec
	sym2 sSym + ext	155890	12 sec
	11	without symmetry	mem ov
sym2 without sSym		43554583	1006 min
sym2 sSym		43554583	882 min
sym2 sSym + ext		40605664	614 min

Table 3: Verification results for the peterson mutual exclusion protocol

ments occur when using our extensions to state symmetries. This verification example shows that the use of state symmetries also can be very beneficial for systems with different component types.

### 7.3 Peterson Mutual Exclusion Protocol

Here we present and discuss verification results for the Peterson mutual exclusion protocol [Peterson, 1981]. Results of our verification experiments are summarized in table 3. In this protocol entry to the critical section is gained by a single process via a series of  $n - 1$  competitions. There is at least one loser for each competition and the protocol satisfies the mutual exclusion condition since at most one process can win the final competition. For the competitions the protocol needs a global array with component identifiers as values. Because of this and the cumbersome and inefficient specification of such variables in SMC, we decided to test the possibilities of state symmetries and our extensions with this protocol. A component of the protocol can have considerably more local states than a component of the algorithms from the last two subsections. Therefore verification hasn't been tractable for systems with as many components as there.

Our results show that the usage of *sym1* leads to a very strong increase in runtime compared to verification without symmetry reduction. The reason therefore is that all possible permutations of a global state are used by *sym1* to compute a canonical representative. These computations need a lot of runtime. Whereas *sym2* heuristically restricts the set of permutations it uses. This even leads to a runtime decrease. But despite this differences of the canonicalization algorithms, without restricting the feasibility of transitions the same number of states have been visited during state-space search. Therefore algorithm *sym2* seems to work very well and achieves great runtime reductions. Because the protocol works in rounds it is expected that only a few components will be in local states in which they are in upper rounds of the tournament. Therefore we here omitted the use of state symmetries in rules when components have been in these rounds of the competition for the verification runs mentioned *sSym + ext* in table 3. Also we restricted there for two control states of the protocol the feasibility of rules which only change local variables of the same component. This leads

to a decreasing number of states which have to be considered for verification and didn't impact the property that only one component is allowed to be in the critical section at the same time, which we used for verification. Therewith the visiting of some irrelevant states for verifying this property can be avoided. To use this enhancement we haven't exhaustively analyzed the input model but only inserted restrictions of the feasibility if they have been obvious and easy to detect through examining the rules of the model and the property to verify. Nevertheless this change lead to the greatest runtime improvements.

Table 3 shows that the runtime of verification runs with *sym1* and 7 components can be improved by using state symmetries. Also further runtime improvements are possible by using our proposed extensions. When using *sym2* for 7 components the verification time is already very small, but also runtime improvements are observable. Interestingly the number of visited states for 7 components and the testcases with *sSym+ext* differs for the canonicalization algorithms *sym1* and *sym2*. This effect has been caused through different canonical states which *sym1* and *sym2* generated through their possible permutations. Therewith our restriction of the feasibility of transitions could lead to the execution of different transitions in symmetric global states which lead to different successor states. Depending on the possible successor states the number of visited states then can vary. This effect also could be observed during verification runs with the example of the next subsection. When verifying the protocol for 11 components big runtime savings can be achieved by using only state symmetries and even more by using our extensions. Therefore especially in verification of systems with a large state-space and whose verification needs a lot of time, our extensions should be used additionally to state symmetries to bring much more runtime reductions. This also shows that for enabling model checking to verify large systems with many states, it is recommendable to combine symmetry reduction and the use of state symmetries with several other abstraction techniques to get the most benefits.

#### 7.4 Distributed List Protocol

This subsection discusses the verification of a distributed list protocol. Experimental results for it can be seen in table 4. This protocol has the particularity that not as much state symmetries as in the previous algorithms can be exploited for it and their detection is more complex. The system we verified consisted of a number of components and a global buffer for messages. The messages thereby contained component identities as source and destination addresses. Its Murphi input description had rules with conditions about local component states and also some possessed the availability of certain message types as condition. In the local states of the components information about the next component in the list is stored. Because of this the detection of state symmetries is more time-consuming for this protocol (e.g. always the content of all messages in the message buffer has to be analyzed to detect state symmetry between two components). There have also been rules where one could easily decide through static analysis of the verification model that no usage of state symmetry would be possible for them. As properties we verified for this

number of components	canonicalization algorithm	number of states	time
7	without symmetry	mem ov	-
	sym1 without sSym	3619472	264 min
	sym1 sSym	3619472	263 min
	sym1 sSym + ext	2700608	193 min
	sym2 without sSym	3619472	19 min
	sym2 sSym	3619472	20 min
	sym2 sSym + ext	3231984	16 min

Table 4: Verification results for the distributed list protocol

protocol that in the stable states of the protocol the predecessors and successors of the list members are always in the list and the next pointer of components which aren't in the list should point to themselves. For our verification experiments we didn't generate all permutations of a single global state to detect state symmetries but used transpositions of components and the corresponding messages to detect state symmetries. We have chosen this approach because of the lower runtime needed for it. We expect that when using state symmetries and by always generating all permutations to detect them, the runtime would be considerably higher for this example. Therewith also the impact of extensions to the use of state symmetries would grow considerably.

Our verification results confirmed that the global states of this system don't contain as much state symmetries. In the verification experiments with canonicalization algorithm *sym1* little runtime reductions can be achieved by using state symmetries. With canonicalization algorithm *sym2* and using state symmetries, we even got a slight increase in runtime. This has been caused by time-consuming state symmetry checks for many cases where state symmetries didn't exist. The increase of runtime occurred for canonicalization algorithm *sym2* in contrast to the experiments with *sym1*, because the computation time for canonicalization when using *sym1* is higher. Therefore larger runtime improvements through existing state symmetries can be achieved for algorithm *sym1*. Also we made verification runs by using state symmetries and enhancements from section 6. There we eliminated the test for state symmetries from rules where we could easily decide through static analysis of the input model that no state symmetries exist for them. Also we restricted the feasibility of rules in cases where it could be easily decided that omitting them and the resulting global states wouldn't affect the verification result. The verification results show that even when the use of only state symmetries can lead to an increase of runtime, by using further extensions runtime reductions can be achieved for both canonicalization algorithms. For this verification model the biggest improvements in runtime have been made through omitting the visiting of states which have no influence on the verification result. When using only state symmetries it has been sufficient to remove state symmetry detection where state symmetries cannot be observed in that model, for eliminating the increase of runtime for *sym2* and even to get small runtime improvements.

## 8 Conclusion and Outlook

In this paper we investigated the benefits of using state symmetries in addition to symmetry reduction for the model checker Murphi. Also we developed and examined extensions to the use of state symmetries. From our results we conclude that the use of state symmetries can lead to high runtime improvements. This effect could not only be observed for very large systems but already for systems with 7 components. We saw very large runtime reductions for the peterson protocol and only 11 system components. The experiments also showed that in systems with few components and few state symmetries an increase in runtime can occur, if state symmetry detection is time-consuming. Experiments with further improvements to the use of state symmetries also showed promising results. We observed that improvements through simple static analysis of the verification model often lead to significant runtime reductions. Therefore the combination of symmetry reduction, the use of state symmetries and techniques for analyzing a verification input model have to be examined deeper to get further runtime reductions.

In the future we plan to further analyze the potential of using state symmetries only partially. Sometimes it could be better to break state symmetry detection and to execute a rule if the computation load for an exact decision about the presence of state symmetries seems to be high. Also we want to examine existing automatic techniques for static analysis which help to improve the benefits of using state symmetries. Additionally stronger methods for restricting the feasibility of transitions which limit the state-space have to be examined. Therefore we want to analyze existing methods for breaking symmetries in constraint satisfaction problems to exploit similarities in breaking symmetries.

## References

- [Ben-Ari *et al.*, 1981] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 164–176. ACM, 1981.
- [Clarke and Emerson, 1982] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.
- [Clarke *et al.*, 1996] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des.*, 9(1-2):77–104, 1996.
- [Clarke *et al.*, 1998] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In *CAV98, LNCS 1427*, pages 147–158. Springer-Verlag, 1998.
- [Crawford *et al.*, 1996] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. pages 148–159. Morgan Kaufmann, 1996.
- [Dill *et al.*, 1992] D. L. Dill, A. J. Drexler, A. J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.
- [Emerson and Halpern, 1986] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. In *Journal of the ACM*, volume 33, pages 151–178. ACM, 1986.
- [Emerson and Sistla, 1996] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.
- [Emerson and Wahl, 2003] A. Emerson and T. Wahl. On combining symmetry reduction and symbolic representation for efficient model checking. In *Correct Hardware Design and Verification Methods*, 2003.
- [Emerson *et al.*, 1997] E. A. Emerson, S. Jha, and D. Peled. Combining partial order and symmetry reductions. In *TACAS '97: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 19–34, London, UK, 1997. Springer-Verlag.
- [Gyuris *et al.*, 1997] V. Gyuris, A. P. Sistla, and E. A. Emerson. On-the-fly model checking under fairness that exploits symmetry. In *CAV97, LNCS 1254*, pages 232–243. Springer-Verlag, 1997.
- [Ip and Dill, 1993] C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234. IEEE Computer Society, 1993.
- [Ip and Dill, 1996] C. N. Ip and D. L. Dill. Better verification through symmetry, 1996.
- [McKay, 2007] B. D. McKay. *nauty user's guide* (version 2.4), 2007.
- [Miller *et al.*, 2006] A. Miller, A. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3):8, 2006.
- [Peterson, 1981] G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
- [Pnueli, 1981] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
- [Queille and Sifakis, 1982] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
- [Rintanen, 2003] Jussi Rintanen. Symmetry reduction for sat representations of transition systems. In *ICAPS*, pages 32–41, 2003.
- [Sistla *et al.*, 1999] A. P. Sistla, V. Gyuris, and E. A. Emerson. Smc: A symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering Methodologies*, Vol 9, No, 2:133–166, 1999.