

A Light-weight Algorithm for Model Checking with Symmetry Reduction and Weak Fairness

Dragan Bošnački

Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: dragan@win.tue.nl

Abstract. We present an algorithm for (explicit state) model checking under weak fairness that exploits symmetry for state space reduction. It is assumed that the checked properties are given as Büchi automata. The algorithm is based on the Nested Depth First Search (NDFS) algorithm by Courcoubetis, Vardi, Wolper and Yannakakis. The weak fairness aspect is captured by a version of the Choueka flag algorithm. As the presented algorithm allows false positives, it is mainly intended for efficient systematic debugging. However, we show that for this more modest goal our algorithm has significant advantages over the existing full-fledged model checking algorithms that exploit symmetry under weak fairness. The prototype implementation on top of Spin showed encouraging results.

1 Introduction

Symmetry reduction (c.f. [10, 6]) is one of the most effective weapons against the state space explosion problem in model checking [6].

Many systems, like mutual exclusion algorithms, cache coherence protocols, or bus communication protocols, exhibit significant degree of symmetry. In order to grasp the idea behind symmetry reduction, consider a typical mutual exclusion protocol. The (im)possibility for processes to enter their critical sections simultaneously will stay the same if the process identities are permuted. As a consequence, when during state-space exploration a state is visited that is the same, up to a permutation of pids, as some state that has already been visited, the search can be pruned. More formally, the symmetry of the system is represented by a given group G of permutations that act on the global states of the system. It turns out that two states s and s' are behaviorly equivalent if there exists a permutation $\pi \in G$ such that s' can be obtained by applying π on s . Thus, the system state space T is partitioned into equivalence classes. We define a selection function h which selects a unique representative from each equivalence class. Next, the quotient state space $h(T)$ is constructed that contains only these representatives and the property is checked using $h(T)$ instead of T . As $h(T)$ is in general much smaller than T , the gain in memory and time needed for the verification algorithms can be significant.

In this paper we confine our attention to the notion of weak fairness on process level. Thus, in a quite standard way we assume that the system is specified as a collection of concurrent (sequential) processes. Under this assumption we require that for every execution sequence of the system, if some process becomes continuously enabled at some point of time (i.e. can always execute some of its statements), then at least one statement from that process will eventually be executed. This kind of fairness is most often associated with mutual exclusion algorithms, busy waiting, simple queue-implementations of scheduling, resource allocation, etc. Weak fairness will guarantee the correctness of properties like eventually entering the critical region for every process which is continuously trying to do this (in the mutual exclusions) or eventually leaving the waiting queue for each process that has entered it (in the scheduling) [13].

Thus, combining the algorithms for model-checking under weak fairness with reduction techniques, and in particular symmetry, is a prerequisite for the verification of many interesting properties in practice. However, when coupling the two concepts special care should be taken, because of the possible incompatibilities between the particular algorithms.

The main contribution of this paper is an algorithm for model checking under weak fairness (in the sense described above) that exploits symmetry reduction. We assume that the properties that are checked are specified as Büchi automata [19]. As a consequence the problem of checking whether some given property holds for the system under consideration can be reduced to the problem of finding acceptance cycles (i.e., cycles that contain acceptance states) in the graph representing the product of the system state space with the property automaton (c.f. [7, 6]).

Our basic idea is conceptually simple. We apply a (symmetry) reduction algorithm in order to reduce the state space and on the reduced state space we run an algorithm for detecting weakly fair acceptance cycles. (In accord with what was said above, along these cycles each process must be either disabled or contribute a transition to the cycle.) The main challenge is how to combine the symmetry and weak fairness algorithms efficiently. This means that we want the two algorithms to be run simultaneously (instead of sequentially) and on-the-fly (i.e. the state space is generated only on demand). The on-the-fly feature is very important in debugging because it often happens that there is not enough memory or time to deal with the whole state space although it can still be easy and useful to detect possible erroneous executions with the existing resources. Unfortunately, with our symmetry reduction algorithm it is not guaranteed (at least in theory) that if there exists a fair acceptance cycle in the original state space, it will also exist one in the reduced state space. As a consequence, the combined algorithm can give false positives, i.e., it can miss some acceptance cycles. As such, our algorithm is mainly intended for debugging.¹

¹ At this stage it is not clear for what kind of acceptance cycles (i.e. properties) our algorithm will fail. The counterexample which was suggested to the author by Dennis Dams was quite contrived and we were not able to produce a corresponding Promela model on which our algorithm should fail. For all examples on which the algorithm

The algorithms which are presented in this paper are based on the so-called nested depth first search (NDFS) algorithm by Courcoubetis, Vardi, Wolper and Yannakakis [7] for detecting acceptance cycles in the state space.

As a stepping stone toward the final goal, we first describe an NDFS based algorithm for model checking under weak fairness without symmetry reduction. This algorithm is a minor modification of the weak fairness algorithm of the model checker Spin, as implemented by Holzmann. The last algorithm is actually a version of the Choueka flag algorithm [5]. The idea of this upgrade of NDFS for weak fairness is to do the search for an acceptance cycle in an extended state space in which it is guaranteed that each cycle which is detected is a fair one. The final algorithm which features both fairness and symmetry is a merge of the weak fairness algorithm with a NDFS based algorithm for symmetry reduction from [3]. The main idea behind the resulting algorithm is to find a path in the original state space for which it can be proved that it is a fragment of a weakly fair cycle. Along such a fragment each process has already been disabled or contributed a transition. In fact the weakly fair cycle can be obtained by concatenating symmetric images of the fragment.

The algorithms for combining symmetry with (weak) fairness that we found in the literature [11, 14] are based on finding maximal strongly connected components (MSCC) in the state space graph. Because of that our algorithm has the advantages that standard NDFS has over the MSCC based algorithms. More precisely, we argue that in practice both the time and space complexities of our algorithm are better. Also, unlike the algorithms from [11, 14], our algorithm is compatible with the approximative verification techniques from [15, 21], which are important in the context of debugging. Shortly before the submission of this paper it was suggested to the author that an efficient model checking algorithm that exploits symmetry with fairness based on the LTL model checking algorithm from [8] can be designed. It is quite difficult to compare the efficiency of our algorithm with such a hypothetical algorithm. However, as the algorithm of [8] is also based on MSCC, it is evident that it is not going to be compatible with the above mentioned approximative verification techniques from [15, 21].

2 Preliminaries

2.1 Labeled Transition Systems

In the sequel we adopt the automata theoretic approach to model checking [20]. In particular, we assume that the properties that are checked are given as Büchi automata. As it was mentioned in the introduction, we work with the graph obtained as a product of the state space graph representing the system (more precisely: the system model) with the automaton (graph) representing the (negation of the) property. The algorithms to obtain the product state space are quite standard (c.f. [7, 6]). In the sequel we assume that the product state space is given.

was tested, if an acceptance cycle existed, the algorithm successfully reported an error. Of course, this is not enough to draw any definite conclusions.

(An on-the-fly integration of the algorithms for obtaining the product state space with the algorithms presented in this paper should be trivial (c.f. [6]).)

Because we deal with weak process fairness it is natural to assume that we model the system as a collection of processes. To capture this in the semantics we assume a finite sequence of processes P_1, P_2, \dots, P_N , $N \geq 1$, which are represented by their indices (process IDs, pids) from the set $I = \{1, 2, \dots, N\}$. For our purposes, we represent the final state space graph as a *labeled transition system* formally defined as follows:

Definition 1. *Let Prop be a set of atomic propositions. A labeled transition system (LTS) is a 7-tuple $T = (S, A, I, R, L, \hat{s}, F)$, where*

- S is a finite set of states,
- A is a finite set of actions,
- I is a finite set of processes (process IDs),
- $R \subseteq S \times A \times I \times S$ is a transition relation (we write $s \xrightarrow{a,i} s' \in R$ for $(s, a, i, s') \in R$),
- $L : S \rightarrow 2^{Prop}$ is a labeling function which associates with each state a set of atomic propositions that are true in the state,
- \hat{s} is the initial state,
- $F \subseteq S$ is the set of acceptance states.

Unless stated differently, we fix T to be $(S, A, I, R, L, \hat{s}, F)$ for the rest of the paper.

Intuitively, if a transition is labeled by the pid i , this means that it is obtained as a result of a statement executed by the process P_i . An action a is *enabled* in a state $s \in S$ iff $s \xrightarrow{a,i} s' \in R$ for some $s' \in S$ and $i \in I$. We say that the process P_i is *enabled* in $s \in S$ iff there exists $a \in A, s' \in S$, such that $s \xrightarrow{a,i} s' \in R$. An *execution sequence* or *path* is a finite or infinite sequence of subsequent transitions, i.e., for $s_i \in S, a_i \in A, j_i \in I$ the sequence $s_0 \xrightarrow{a_0, j_0} s_1 \xrightarrow{a_1, j_1} s_2 \dots$ is an execution sequence in T iff $s_i \xrightarrow{a_i, j_i} s_{i+1} \in R$ for all $i \geq 0$. An infinite execution sequence is said to be *accepting* iff there is an acceptance state $s \in F$ that occurs infinitely many times in the sequence. A finite execution sequence $c = s_0 \xrightarrow{a_0, j_0} s_1 \xrightarrow{a_1, j_1} \dots \xrightarrow{a_{n-1}, j_{n-1}} s_n, n \geq 1$ is a *cycle* iff the start and end states coincide, i.e. $s_0 = s_n$. Given a finite or infinite execution sequence $\sigma = s_0 \xrightarrow{a_0, j_0} s_1 \xrightarrow{a_1, j_1} s_2 \dots$, a process $P_i, 1 \leq i \leq N$, and a state s_j from the execution sequence, we say that P_i is *executed* in σ in s_j iff $s_j \xrightarrow{a,i} s_{j+1}$ is a transition in σ . A state s is *reachable* iff there exists a finite execution sequence that starts at \hat{s} and ends in s . A cycle c is *reachable* iff there exists a state in c which is reachable. A cycle c is an *acceptance cycle* if it contains at least one acceptance state.

2.2 The Standard Nested Depth-First Search Algorithm

The algorithms presented in this paper are based on the algorithm of [7] for memory efficient verification of LTL [9] properties, called nested depth first search (NDFS) algorithm. The standard NDFS algorithm is given in Fig. 1.

```

1  proc dfs1(s)
2    add {s,0} to States
3    for each process i := 1 to N do
4      for each transition (s,a,i,s') do
5        if {s',0} not in States then dfs1(s') fi
6      od
7    od
8    if accepting(s) then seed:={s,1}; dfs2(s) fi
9  end
10
11 proc dfs2(s) /* the nested search */
12   add {s,1} to States
13   for each process i := 1 to N do
14     for each transition (s,a,i,s') do
15       if {s',1} == seed then report cycle
16       else if {s',1} not in States then dfs2(s') fi
17     od
18   od
19 end

```

Fig. 1. Nested depth first search (NDFS) algorithm.

The algorithm consists of two depth first search (DFS) procedures: `dfs1` which implements the “basic” DFS and `dfs2` which performs the “nested” DFS. In order to detect possible acceptance cycles, at each acceptance cycle the basic DFS is temporarily suspended and the nested DFS is launched (by calling `dfs2` in line 8), with the current (acceptance) state as a `seed`. The procedures `dfs1` and `dfs2` work in separate state spaces. If `dfs2` matches the `seed`, this means that an acceptance cycle is found. In this case the cycle is reported and the program is stopped (line 16). If, however, the nested DFS ends without matching the `seed`, the basic DFS (`dfs1`) is resumed until a possible new acceptance state is encountered. It should be emphasized that, although it might look quadratic, the NDFS algorithm is linear in the number of states in the state space [7]. (See also [3].)

The fact that for each state s of the original state space the copies in the first and second DFS differ only in the second (bit) component can be used to save memory space [17]. The states $(s, 0)$ and $(s, 1)$ can be stored together as (s, b_1, b_2) , where b_1 (respectively b_2) is a bit which is set to 1 iff $(s, 0)$ (resp. $(s, 1)$) has been already visited during the first (resp. second) DFS. Thus in this way, the (doubled) state space used by NDFS takes, for realistic systems, virtually the same amount of memory as the original state space.

The following claim (essentially Theorem 1 from [7]) establishes the correctness of the algorithm:

Theorem 1 ([7]). *Given an LTS T , the NDFS algorithm in Fig. 1, when called on \hat{s} , reports a cycle iff there is a reachable acceptance cycle in T .*

2.3 Symmetry Reduction

Given a LTS $T = (S, A, I, R, L, \hat{s}, F)$ let $Perm(I)$ and $Perm(S)$ be the groups of permutations of the sets $I = \{1, \dots, N\}$ of pids and S of states, respectively. Both $Perm(I)$ and $Perm(S)$ are groups under the functional composition \circ defined as: For any two permutations π_1, π_2 , $(\pi_1 \circ \pi_2)(x) \stackrel{def}{=} \pi_1(\pi_2(x))$. We assume that each permutation $\pi \in Perm(I)$ can be lifted into the permutation π^* on the state set S . This assumption is quite natural regarding the way symmetry reduction is handled in practice (see for instance [18, 10, 4]). Formally, we require that there exists a mapping $(\cdot)^* : Perm(I) \rightarrow Perm(S)$ which maps each $\pi \in Perm(I)$ into $\pi^* \in Perm(S)$.

Definition 2. *Given a LTST $T = (S, A, I, R, L, \hat{s}, F)$ and a mapping $(\cdot)^* : Perm(I) \rightarrow Perm(S)$, a subgroup G of $Perm(I)$ is called a symmetry group of T iff for all $\pi \in G$*

- $s \xrightarrow{a,i} s' \in R$ iff $\pi^*(s) \xrightarrow{a,\pi(i)} \pi^*(s') \in R$.
- For all $s \in S$ $\pi^*(s) = \hat{s}$ iff $s = \hat{s}$
- For all $s \in S$ $L(s) = L(\pi^*(s))$.
- $s \in F$ iff $\pi^*(s) \in F$.

We say that the states $s_1, s_2 \in S$ are in the same *orbit* iff there exists $\pi \in G$ such that $\pi^*(s_1) = s_2$. The symmetry group G induces the *orbit* relation $\Theta_G \subseteq S \times S$ defined as $\Theta_G = \{(s_1, s_2) \mid s_1 \text{ and } s_2 \text{ are in the same orbit}\}$. It is trivial to show that Θ_G is an equivalence relation. It is convenient to work with representatives of these equivalence classes (orbits). To this end we introduce the function $h : S \rightarrow S$ which maps a given state s of T into its representative. More precisely, given $s_1, s_2 \in S$, s_1 and s_2 are in the same orbit iff $h(s_1) = h(s_2)$.

Directly from Def. 2 above we have:

Corollary 1. *Let $\rho = s_0 \xrightarrow{a_0, j_0} s_1 \xrightarrow{a_1, j_1} \dots \xrightarrow{a_{n-1}, j_{n-1}} s_n, n \geq 1$ be a path in T and π a permutation in G . Then the path $\pi(\rho) = \pi^*(s_0) \xrightarrow{a_0, \pi(j_0)} \pi^*(s_1) \xrightarrow{a_1, \pi(j_1)} \dots \xrightarrow{a_{n-1}, \pi(j_{n-1})} \pi^*(s_n)$ is also in T .*

We can exploit the symmetry for state space reduction by pruning the state space search from the currently visited state, if some symmetric state (i.e. from the same equivalence class) has already been visited. An adaptation of the NDFS algorithm that employs symmetry is given in Fig. 2.

The only changes compared to the standard NDFS from Fig. 1 are in lines 2, 5, 8, 12, 15, and 16, where s and s' are replaced with their representatives $h(s)$ and $h(s')$, respectively. (Notice though that the procedures `dfs1` and `dfs2` are called with s and s' as parameters, instead of $h(s)$ and $h(s')$ respectively.) The net effect is the same as if we do the exploration of the state space instead in the original state space T in a reduced LTS $h(T)$. The states of $h(T)$ are representatives of the symmetry equivalence classes from T obtained by means of the function h .

```

1  proc dfs1(s)
2    add {h(s),0} to States
3    for each process i := 1 to N do
4      for each transition (s,a,i,s') do
5        if {h(s'),0} not in States then dfs1(s') fi
6      od
7    od
8    if accepting(s) then seed:={h(s),1}; dfs2(s) fi
9  end
10
11 proc dfs2(s) /* the nested search */
12  add {h(s),1} to States
13  for each process i := 1 to N do
14    for each transition (s,a,i,s') do
15      if {h(s'),1} == seed then report cycle
16      else if {h(s'),1} not in States then dfs2(s') fi
17    od
18  od
19 end

```

Fig. 2. Nested depth first search (NDFS) algorithm with symmetry reduction.

In [3] it is shown that the algorithm in Fig. 2 reports a cycle iff there is an acceptance cycle in the original LTS T . Notice though that the preservation of fair acceptance cycles is not guaranteed. In other words, given a fair acceptance cycle c in T , the acceptance cycle which is detected (and reported) in the reduced state space $h(T)$, and which corresponds to c , is not necessarily weakly fair. Intuitively this is because in general the cycles in $h(T)$ can be shorter and the process indices are “scrambled”.

It is worth emphasizing again that in order to exploit symmetry the state space must be symmetric. As we assume that we work with state spaces which incorporate the property (originally given as a Büchi automaton), this means that implicitly we require that the property which is checked is also symmetric, i.e., invariant under the permutations from G applied on pids from I .

2.4 Weak Fairness

We consider weak fairness with regard to processes, i.e. we say that a given execution sequence is fair if for each process that becomes continuously enabled starting at some point in the execution sequence, a transition belonging to this process is executed infinitely many times. Formally:

Definition 3. *An infinite execution sequence $s_0 \xrightarrow{a_0, p_0} s_1 \xrightarrow{a_1, p_1} s_2 \dots$ is (weakly) fair iff for each process $P_l, 1 \leq l \leq N$ the following holds: If there exists $i \geq 0$ such that P_l is enabled in s_j for all $j \geq i$, then there are infinitely many $k \geq 0$ such that P_l is executed in s_k .*

A cycle $c = s_0 \xrightarrow{a_0, p_0} s_1 \xrightarrow{a_1, p_1} \dots s_{n-1} \xrightarrow{a_{n-1}, p_{n-1}} s_0$ is (weakly) fair iff whenever a process P is enabled in all states s_i , $0 \leq i < n$, then P is executed in some state s_j , $0 \leq j < n$.

When solving the model-checking problem under the (weak) fairness assumption we are interested only in fair accepting execution sequences. As we work with finite LTSs, it is obvious that, translated in terms of acceptance cycles, the fairness assumption means that we require that the acceptance cycles we detect in the state space are fair.

3 Combining Weak Fairness with Symmetry Reduction

3.1 Model Checking under Weak Fairness without Symmetry

In the sequel we informally discuss an algorithm for model checking with weak fairness (WF) without symmetry reduction. The algorithm which is described below is a variant of Choueka's flag algorithm [5], based on the version implemented in the model checker Spin by Holzmann.

In order to capture weak fairness, one also has to modify the standard NDFS algorithm. This is because the latter only guarantees that it will find some acceptance cycle, if there exists one, but not all of them. Thus, one cannot use the straightforward idea to just ignore the detected cycles which are not fair until one finds a fair one, or there are no more cycles.

To bypass this limitation of the NDFS, we apply it not to the original state space, but to an extended state space. The latter is constructed such that the existence of a fair acceptance cycle in the original state space implies that there exists an acceptance cycle in the extended state space, and vice versa. We first give the intuition behind the extended state space and the modified NDFS algorithm. The set of states of the extended state space consists of $N + 1$ copies of the original state set, where N is the number of processes. The extended state space and its relation to the original state space are shown in Figure 3.

T and F denote the original state space and its acceptance states, respectively. The extended state space is denoted with $\mathcal{F}(T)$, the set of its acceptance states with F_f , while T_i , $0 \leq i \leq N$ are the copies of the original state space T . With $T_0 - F_f$ we denote copy 0 of the original state space without the acceptance states F_f . The label a, l ($a, i \neq l$) on the arrows between the copies express the fact that we can pass between the copies l and $l + 1$ (resp. stay inside copy l) by executing a transition belonging to process l .

The intuition behind the copies of T is that when the algorithm works in the i -th copy of the state space ($1 \leq i \leq N$), this means that it waits for process i to contribute a transition or to become disabled. (In order to treat these two cases uniformly, we will assume that when process i is disabled in the original state space, in the extended state space it executes a *default transition*, labeled with a special action ϵ_i .) When the algorithm is in copy i and process i executes a transition, then the algorithm passes to copy $i + 1$. From copy N it goes back to the special copy 0.

All acceptance states of $\mathcal{F}(T)$ (i.e. the set F_f) reside in copy 0. When the algorithm comes across an acceptance state of $\mathcal{F}(T)$ it jumps immediately via a special ϵ_0 -transition to copy 1. (The ϵ_0 -transitions can be considered as meta transitions which do not belong to any process and do not correspond to any transition from the original system.) The algorithm stays inside copy 0 as long as transitions originating from a non-acceptance state are explored. The ϵ_0 -transitions are the only possible transitions from an acceptance state. Thus, from the structure of $\mathcal{F}(T)$ it is obvious that in order to get back to the acceptance state one has to pass through all N copies of T . This implies that each process i has contributed a transition (ordinary or ϵ) along the cycle, and therefore the latter is weakly fair.

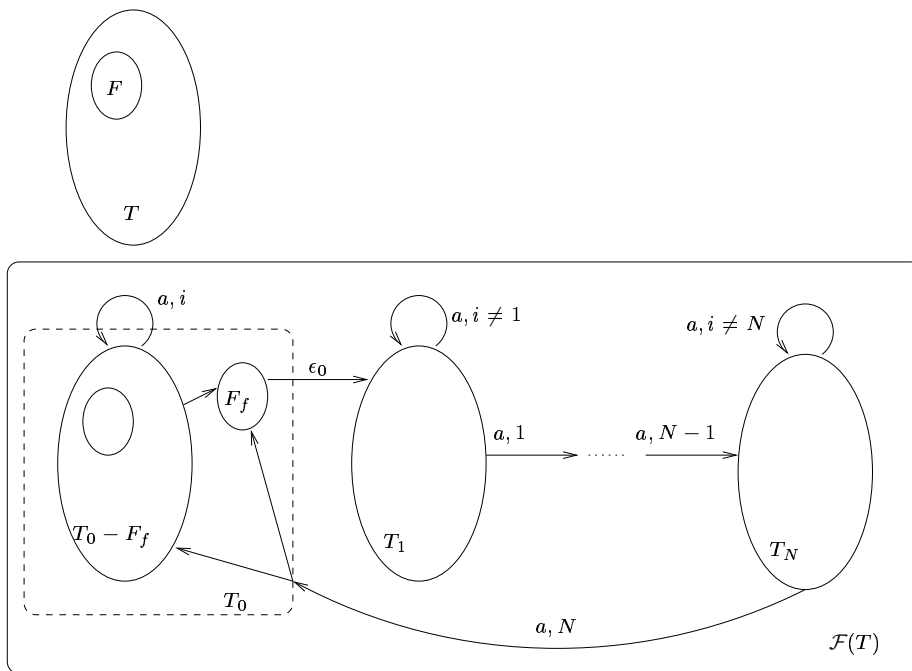


Fig. 3. The extended state space for the weak fairness algorithm.

3.2 Weak Fairness and Symmetry Together On-the-fly

The most direct way to model check under weak fairness is to first unfold the original state space T into the corresponding weakly fair extension $\mathcal{F}(T)$ and then to apply the NDFS algorithm on $\mathcal{F}(T)$. Instead, for efficiency reasons, we do the unfolding of the state space and the NDFS simultaneously. In this way

we keep the advantage of the on-the-fly approach that, if there exists a fair acceptance cycle, usually we do not have to generate the whole $\mathcal{F}(T)$. Once NDFS is adapted to generate the extended state space, it remains to add the symmetry reduction part. We do this by merging the obtained on-the-fly NDFS with weak fairness with the NDFS based symmetry reduction algorithm from [3], given in Fig. 2. Thus, the state space search is pruned from the current state s if a symmetric state has already been saved in the state space. The pseudo-code of the algorithm is given in Figure 4.

We represent the states of the extended state space as triples of the form (s, C, b) . They are obtained in a straightforward way by extending each state s , apart from the bit b discriminating between the first and second DFS, also with the integer counter C that keeps track of the current copy of the original state space.

There are several changes in the standard NDFS algorithm which are needed for the generation of $\mathcal{F}(T)$. The ϵ_0 -transitions are generated with the true branch of the outermost if statement (lines 3 and 4, and 23 and 24). The increment of the counter component by 0 or 1, depending on whether the counter C matches the pid of the currently considered process or not, takes place in lines 7 and 27. The ϵ_i -transitions corresponding to the disabled processes are implemented with the true branch of the corresponding if statements in lines 8 and 9, and 28-30). Notice that also the transition between the states in copy 0, i.e., when the current state is in copy 0 and it is not an acceptance state, is also implicitly implemented through line 7 and 27. As variable i is never 0, if C is 0, then the assignment $C' = C$ is executed, which leaves the counter component unchanged.

The symmetry reduction part is implemented analogously to the algorithm in Fig. 2, i.e., by replacing some occurrences of states with their representatives.

The correctness of the algorithm is given with the following claim:

Theorem 2. *Given an LTS T , if the algorithm from Fig. 4, started with arguments \hat{s} and 0 reports a cycle, then there exists a reachable weakly fair acceptance cycle in T .*

Proof. Suppose that the algorithm reports a cycle. The intuition is that in that case it has detected a path $\rho^0 = s_0^0 \xrightarrow{a_0, j_0} s_1 \xrightarrow{a_1, j_1} \dots \xrightarrow{a_{n-1}, j_{n-1}} s_0^1$, $n \geq 1$, which can be extended into a weakly fair cycle in the original state space T . More precisely ρ^0 is such that

- $h(s_0^0) = h(s_0^1)$, and
- ρ^0 is a “fair fragment”, i.e., for each process i it holds that there exists on ρ^0 a state s_k such that i is disabled in s_k or executed in s_k .

This is implied by the following reasoning. Let s_0^0 be the acceptance state from which we have obtained (in line 17) the seed state which is matched by the algorithm. Thus, $(h(s_0^0), C, 1) = \text{seed}$. (Recall that we consider that our algorithm works in the extended state space $\mathcal{F}(T)$, where each state in the extended state space is a triple of the form (s, C, b) .) The algorithm can make a move, i.e., generate a transition in $\mathcal{F}(T)$ from the currently visited state (s, C, b) in the extended state space only in the following cases:

```

1  proc dfs1(s,C)
2    add {h(s),C,0} to States
3    if C == 0 and accepting(s) then
4      if {h(s),1,0} not in States then dfs1(s,1) fi /* epsilon_0 move */
5    else
6      for each process i := 1 to N do
7        if C == i then C' := (C+1) mod (N+1) else C' := C fi
8        if process i is disabled then /* epsilon_i move */
9          if {h(s),C',0} not in States and C == i then dfs1(s,C') fi
10       else
11         for all (s,a,i,s') do
12           if {h(s'),C',0} not in States then dfs1(s',C') fi
13         od
14       fi
15     od
16   fi
17   if C == 0 and accepting(s) then seed := {h(s),C,1}; dfs2(s,C) fi
18 end
19
20
21 proc dfs2(s,C) /* the nested search */
22   add {h(s),C,1} to States
23   if C == 0 and accepting(s) then
24     if {h(s),1,1} not in States then dfs2(s,1) fi /* epsilon_0 move */
25   else
26     for each process i := 1 to N do
27       if C == i then C' := (C+1) mod (N+1) else C' := C fi
28       if process i is disabled then /* epsilon_i move */
29         if {h(s),C',1} == seed then report cycle
30       else if {h(s),C',1} not in States and C == i then dfs2(s,C') fi
31     else
32       for all (s,a,i,s') do
33         if {h(s'),C',1} == seed then report cycle
34       else if {h(s'),C',1} not in States then dfs2(s',C') fi
35     od
36   fi
37   od
38 fi
39 end

```

Fig. 4. An Algorithm that combines weak fairness and symmetry reduction.

- s is an acceptance state and $C = 0$ (lines 3 and 4, and 23 and 24 – ϵ_0 -transition), or
- Process i is disabled in s and $C = i$ (lines 8 and 9, and 28-30 – ϵ_i -transition), or
- there exists a transition in T from s to s' (lines 12–34).

The first two cases (the ϵ -transitions) do not correspond to “real” transitions that exist in T . We obtain ρ^0 by removing such transitions from the sequence of moves (transitions) taken by the algorithm. This can be done because the source and target states of the ϵ -transitions always have the same s -component. By omitting these transitions no state (of the original LTS T) is lost. The states s_0^0 and s_0^1 belong to the same symmetry equivalence class because also $h(s_0^1, C, 1) = \text{seed}$, and therefore $h(s_0^0) = h(s_0^1)$. Thus, by omitting the C and b components of the states in the transition sequence which the algorithm generates in $\mathcal{F}(T)$ without the ϵ -transitions, we obtain the desired path ρ^0 which exists in T .

Now we show the “fairness” of ρ^0 . From the structure of the algorithm it is clear that the counter C is increased first from 0 to 1 (line 4, 24), then from 1 to N (lines 7, 27), and then reset back to 0 (again lines 7, 27). The increment of the counter C from value i to $i + 1$ (or reset to 0, if $C = N$) is possible only if process i is disabled in the current state or a transition is taken that belongs to process i (code fragments between lines 7 and 14, and 27 to 34.) The increment from 0 to 1 is immediate in case s is an acceptance state. This means that the algorithm generates a sequence of $N + 1$ states s_{j_0}, \dots, s_{j_N} which are on ρ^0 , and for each process i it holds that in s_{j_i} process i is disabled or contributes a transition along ρ^0 .

It remains to show that the “fair fragment” ρ can be extended to obtain a fair cycle. If ρ^0 is a cycle, i.e., $s_0^0 = s_0^1$, then we are done. So, suppose that $s_0^0 \neq s_0^1$. Consequently, there exists a permutation π_1 such that $s_0^1 = \pi_1(s_0^0)$. If we now apply the permutation π_1 to each transition in ρ^0 we will obtain the path $\pi_1(\rho^0) = \rho^1$. By Corollary 1 this path exists in T . (Notice that the end state of ρ^0 is the same with the start state of ρ^1 .) In an obvious way we can continue this procedure to obtain new paths $\rho^{i+1} = \pi_{i+1}(\rho^i)$. From the finiteness of the equivalence class to which the starting states s_0^i belong, it follows that sooner or later one of the states s_0^k will repeat itself. In other words, there exists $0 \leq k < l$ such that $s_0^k = s_0^l$. From the discussion above, it is obvious that the cycle which is obtained as a concatenation of the paths ρ^k to ρ^{l-1} is a weakly fair cycle in T .

□

Comparison with Related Work. We compare our algorithm with the algorithms of Emerson and Sistla (ES95) [11] and Gyuris and Sistla (GS97) [14]. These were the only algorithms for combining weak fairness and symmetry that we could find in the literature. GS97 is an improved version of ES95, so we will refer to the former for comparison. The GS97 algorithm is on-the-fly and it is based on the graph algorithm for finding all maximal strongly connected components (MSCC) from [1]. Unlike our algorithm, ES95 and GS97 are model checking

algorithms that work in both directions, i.e., they do not report false positives. However, in the sequel we argue that if one considers only systematic debugging, our algorithm is more efficient in general. This is mainly because the algorithm in Fig. 4 has the usual advantages that NDFS have over the algorithms that work with MSCCs.

Regarding the time complexity it is assumed that finding a representative can be done in an efficient way, more precisely, in our calculations we assume a constant time. Unfortunately, this is the case only for some special systems and symmetries. In general, no polynomial algorithm is known to compute a unique representative of a symmetry equivalence class. There are however efficient heuristics that work reasonably well in practice (c.f. [18, 4]). The same assumption is made also for the complexity calculations for the GS97 algorithm, therefore, this feature does not have any impact on the comparison.

Under the above assumption our algorithm and GS97 have the same worst case time complexity $O(N \cdot |h(T)|)$, where N is the number of processes in the system and $|h(T)|$ is the size (in number of transitions) of the reduced LTS $h(T)$. (see Appendix A).

In practice we reduce the space complexity of our algorithm by using a similar trick which allowed us to avoid doubling of the state space in the case of the standard NDFS algorithm as described in Section 2.2. Namely, instead of using $N + 1$ copies of the state space $h(T)$ we can extend the state description with $2 \times (N + 1)$ bits, each corresponding to the $(N + 1)$ copies of $h(T)$ in the basic and nested DFS, respectively. (Factor 2 comes because of the two NDFS copies.) This overhead per state is less than the overhead required by GS97. Namely, the MSCC algorithm from [1] keeps two long integers per state as state identifiers. The bottom line is that it can be shown by simple calculation that in practice our algorithm requires less memory in the worst case than GS97. (see Appendix A for more details).

If a fair acceptance cycle is reported, our algorithm has an obvious advantage regarding both time and space because it does not have to finish finding the whole MSCC. GS97 has to construct the whole MSCC in order to detect a violation of the property. This means that the whole MSCC is kept in the memory in contrast to our algorithm which keeps only a part of it – in the best case only the acceptance cycle. Also, exploring a bigger portion of the state space takes more time.

An important practical advantage of our algorithm is that, unlike the two existing algorithms, it is compatible with the memory efficient approximate verification techniques like bit-state hashing [15] and hash-compact [21]. This is not the case with GS97, because of the above mentioned two integers per state needed in the MSCC computation. Finally, for practical reasons our algorithm is also easier to implement because it has a simpler theory basis behind it and does not require some complex structures which are used in GS97.

4 Experiments

A prototype implementation of the algorithm is included in SymmSpin [4], an extension of the model checker Spin [15] with symmetry reductions. We tried it on the case studies from [4] with encouraging results. The obtained reductions for correct properties were usually of several orders of magnitude, very similar with the results for the same examples for safety properties, reported in [4]. Also significant reductions were obtained for incorrect properties, i.e., when a fair acceptance cycle was reported. Due to space constraints we give only the results for one of the examples, which is nevertheless quite illustrative. Table 1 gives the results for Peterson’s mutual exclusion algorithm for an incorrect property of bounded response type (LTL formula: $\Box(p \rightarrow \Diamond q)$). (SR+ and SR- mean with and without symmetry reduction, respectively.) The symmetry reduction algorithm used a selection function h which corresponds to the “pc-segmented” heuristic from [4]. One can see that the reduction factor significantly improves as N increases, which is typical for symmetry reduction.

Table 1. Results for Peterson’s mutual exclusion protocol.

N	2		3		4		5	
	+SR	-SR	+SR	-SR	+SR	-SR	+SR	-SR
nr. of states	81	153	636	2295	4150	84707	77064	out of memory
time [min:sec]	0.1	0.1	0.2	0.6	2.1	1:51.4	5:06.3	—

5 Conclusion and Future Work

In this paper we presented an algorithm for efficient debugging of properties under weak fairness while exploiting symmetry reduction. It was argued that, compared with the similar algorithms that exist in the literature, our algorithm has advantages regarding the time and memory efficiency, and the compatibility with other verification (debugging) techniques. The efficiency of the algorithm was further supported with some encouraging experiments with a prototype implementation on top of the model checker Spin.

A natural avenue for future work is to design an NDFS-based algorithm for full-fledged model checking, i.e., to avoid false positives. Further, it would be interesting to combine our algorithm with other state space reduction techniques, like, for instance, partial order reduction (POR), [6]. In the literature there are successful attempts [12] of combining symmetry and POR. Despite the seemingly widespread belief that weak fairness and partial order are inherently irreconcilable, we believe that this is not true. We are confident that the POR algorithm from [16] can be adapted so that it becomes compatible with fairness. A previous attempt in that direction is described in [2].

References

1. A.V. Aho, J.E. Hopcroft, J.D. Ulmann, *The design and Analysis of Computer Algorithms*, Addison Wesley, 1974.
2. D. Bošnački, *Partial Order Reduction in Presence of Rendez-vous Communications with Unless Constructs and Weak Fairness*, Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, LNCS 1680, pp. 40–56, Springer, 1999.
3. D. Bošnački, *A Nested Depth First Search Algorithm for Model Checking with Symmetry Reduction*, Proc. of FORTE'02, LNCS, Springer, 2002.
4. D. Bošnački, D. Dams, L. Holenderski, *Symmetric Spin*, 7th Int. SPIN Workshop on Model Checking of Software SPIN 2000, pp. 1-19, LNCS 1885, Springer, 2000.
5. Y. Choueka, *Theories of Automata on ω -tapes: a Simplified Approach*, Journal of Computer and System Science, Vol. 8, pp. 117-141, 1974.
6. E.M. Clarke, Jr., O. Grumberg, D.A. Peled, *Model Checking*, The MIT Press, 2000.
7. C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, *Memory Efficient Algorithms for the Verification of Temporal Properties*, Formal Methods in System Design I, pp. 275-288, 1992.
8. J.-M. Couvreur, *On-the-fly Verification of Linear Temporal Logic*, Proc. of FM'99, pp. 253–271, LNCS 1708, 1999.
9. E.A. Emerson, *Temporal and Modal Logic*, in J. van Leeuwen (ed.), Formal Models and Semantics, pp. 995–1072, Elsevier, 1990.
10. E.A. Emerson, A.P. Sistla, *Symmetry and model checking*, Proc. of CAV'93 (Computer Aided Verification), LNCS 697, pp. 463–478, Springer, 1993.
11. E.A. Emerson, A.P. Sistla, *Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata Theoretic Approach*, Proc. of CAV'95 (Computer Aided Verification), LNCS 697, pp. 309–324, Springer, 1995.
12. E.A. Emerson, S. Jha, D. Peled, *Combining partial order and symmetry reductions*, in Ed Brinksma (ed.), Proc. of TACAS'97 (Tools and Algorithms for the Construction and Analysis of Systems), LNCS 1217, pp. 19–34, Springer, 1997.
13. N. Francez, *Fairness*, Springer, 1986.
14. V. Gyuris, A.P. Sistla, *On-the fly model checking under fairness that exploits symmetry*, in O. Grumberg (ed.), Proc. of CAV'97 (Computer Aided Verification), LNCS 1254, pp. 232–243, Springer, 1997.
15. G. J Holzmann, *Design and Validation of Communication Protocols*, Prentice Hall, 1991. Also: <http://netlib.bell-labs.com/netlib/spin/whatispin.html>.
16. G. Holzmann, D. Peled, *An Improvement in Formal Verification*, FORTE 1994, Bern, Switzerland, 1994.
17. G. Holzmann, D. Peled, M. Yannakakis, *On Nested Depth First Search*, Proc. of the 2nd Spin Workshop, Rutgers University, New Jersey, USA, 1996.
18. C.N. Ip, D.L. Dill, Better verification through symmetry. *Formal Methods in System Design*, Vol. 9, pp. 41–75, 1996.
19. W. Thomas, *Automata on Infinite Objects*, in J. van Leeuwen (ed.), Formal Models and Semantics, pp. 995–1072 Elsevier, 1990.
20. M. Vardi, P. Wolper, *Automata Theoretic Techniques for Modal Logics of Programs*, Journal of Computer and System Science, 32(2), pp. 182–221, 1986.
21. P. Wolper, D. Leroy, *Reliable Hashing without Collision Detection*, Proc. of CAV'93 (Computer Aided Verification), LNCS 697, pp. 59–70, Springer, 1993.

A Comparison with Related Work

This appendix is not part of the paper and it is intended only for the reviewing process.

In this section we give some more detail regarding the time and space complexity of our algorithm, as well as the ES95 and GS97 algorithms.

Regarding the time complexity it is assumed that finding a representative can be done in an efficient way, more precisely, in our calculations we assume a constant time. Unfortunately, this is the case only for some special systems and symmetries. In general, no polynomial algorithm is known to compute a canonical representative. There are however efficient heuristics that work reasonably well in practice (c.f. [18, 4]). A constant time for finding a representative is also assumed in the complexity calculations for the GS97 algorithm, therefore, this feature does not have any impact on the comparison.

Under the above assumption, the algorithms in [11, 14] have worst case time complexity of $O(N^2 \times |h(T)|)$ and $O(N \times |h(T)|)$, where N is the number of processes in the system and $|h(T)|$ is the size (in number of transitions) of the reduced LTS $h(T)$. As our algorithm is an NDFS in the extended state space consisting of N copies of $h(T)$, its worst case time complexity is also $O(N \times |h(T)|)$. However, in practice our algorithm will be most of the time faster than the algorithm in [14]. This is because, especially in the cases when the checked property does not hold, it is usually much faster to find an acceptance cycle instead of the whole MSCC to which this cycle belongs. Besides, as explained below, GS97 manipulates quite complex data structures which also slows down this algorithm.

In practice we can reduce the space complexity of our algorithm by using a similar trick which allowed us to avoid doubling of the state space in the case of the standard NDFS algorithm as described in Section 2.2. Namely, instead of using $N + 1$ copies of the state space $h(T)$ we can extend the state description with $2 \times (N + 1)$ bits, each corresponding to the $(N + 1)$ copies of $h(T)$ in the basic and nested DFS, respectively. (Thus the factor 2 comes because of the two NDFS copies.) Similarly as in Section 2.2, if now the i -th bit ($0 \leq i \leq N = 1$) of these extra bits is set to 1, this means that i -th copy of s has been visited by the algorithm during the basic DFS. (If $i > N + 1$ then $i - (N + 1)$ -th copy of s has been visited.) If $|s|$ is the size of the state description, in practice we have that $|s| \gg 2 \times (N + 1)$. This means that we virtually need to store only $|h(T)|$ states. GS97 also has memory complexity $O(|h(T)|)$. However, it uses several extra data structures of which two integers are essential. For computing of the maximal strongly connected components, GS97 algorithm for each state has to keep two special unique numbers (called NFNUMBER and LOWLINK in [1]). Therefore $2 \cdot \log|h(T)|$ extra bits are needed in the state vector. To the best of our knowledge nothing similar to the above described efficient storage technique is used in GS97. But even if it were used, in general $2N \times \log|h(T)|$ extra bits would be needed for the two unique numbers for each of the N copies of the state. Thus, even with this minimal overhead, for systems where $N \times \log|h(T)| > (N + 1)$, in practice: always, our algorithm will have shorter description of the state vector.