7. Nested Depth First Search Algorithms for Symmetry Reduction in Model Checking
Nested Depth First Search Algorithms for Symmetry Reduction in Model Checking

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

Abstract. We propose an algorithm for model checking under weak fairness that exploits symmetry for state space reduction. As intermediate results we also discuss two other algorithms which deal separately with weak fairness and symmetry reduction. The algorithms presented in this chapter are based on the Nested Depth First Search (NDFS) algorithm by Courcoubetis, Vardi, Wolper and Yannakakis. We argue that the worst case time complexity of our algorithm for model checking under weak fairness with symmetry is of the same order of magnitude as the time complexity of the algorithms known in the literature. Moreover, as these algorithms require finding all strongly connected components (SCC) in the state space graph, our algorithm preserves the advantages of the NDFS over the SCC approach.

1 Introduction

Model checking [6, 22] is a widespread technique for the automated verification of concurrent systems. The technique lends itself to implementation and as such it is successfully used in the construction of verification tools [23]. However, model-checking tools are often limited by memory requirements because of the problem of state space explosion. One of the important techniques to alleviate this problem is symmetry reduction.

Symmetry is present in many systems, like mutual exclusion algorithms, cache coherence protocols, bus communication protocols, etc. 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 behaviorally equivalent if there exists a permutation $\pi \in G$ such that $s'$ can be obtained by applying $\pi$ 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.

The issue of fairness is an inherent and important one in the study of concurrency and nondeterminism, in particular in the area of the verification of concurrent systems. Since fairness is used as a generic notion there is a broad taxonomy of fairness concepts. In this chapter we confine our attention to the notion of weak fairness on the level of processes. This means that we require that for every execution sequence of the concurrent program which is a composition of several processes, 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, and resource allocation. Weak fairness will guarantee the correctness of statements 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 the chapter is an algorithm for model checking under weak fairness (in the sense described above) that exploits symmetry reduction. As intermediate results we also discuss two other algorithms which deal separately with weak fairness and symmetry reduction.

We assume that the properties that are checked are specified as Büchi automata [21]. 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 model state space with the property automaton (c.f. [7, 5]).

All the algorithms that are presented in the chapter 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. In the NDFS algorithm from each acceptance state which is visited during the standard DFS exploration of the state space a check procedure is called in order to detect a possible acceptance cycle through the state. This nested cycle check is itself a DFS, which explains the name “nested DFS”.

In order to capture weak fairness, one 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. In the chapter we present an
algorithm for model checking under weak fairness which is a minor modification of the weak fairness algorithm by Gerard Holzmann implemented in the model checker Spin. The idea of the upgrade for weak fairness is to do the acceptance cycle search in an extended state space in which it is guaranteed that each cycle that is detected is a fair one. In fact, we need a more general result that claims that there exists a fair acceptance cycle in the original state space if and only if there exists an acceptance cycle in the extended state space. We show the correctness of our modified fairness algorithm by arguing that it is equivalent to the standard NDFS algorithm applied to the extended state space.

Our second stepping stone towards the main algorithm is presented in a more general framework of state space reductions that preserve bisimulation, introduced by Emerson, Jha and Peled [12]. The symmetry based reductions are only a special case of such reductions. The algorithm that we consider is a straightforward generalization of the symmetry reduction algorithms from [19, 10, 12, 3]. After presenting an NDFS version of this generalization, we prove its correctness.

Finally, we present the main algorithm that reconciles the issues of fairness and symmetry. We begin by discussing why a straightforward combination of the NDFS versions of the fairness and bisimulation preserving reduction algorithms fails. After that we briefly introduce symmetry reduction as a special case of a bisimulation preserving reduction. The algorithm for fairness and symmetry is based on the theory developed by Emerson and Sistla [11]. In this chapter we do the necessary adjustments of the theory so that we can fit it into the NDFS concept. The main idea (borrowed from [11]) is to work with a reduced state space $h_G(T)$ which, unlike for the ordinary symmetry reduction algorithm, has transitions annotated with permutations. In this way the process identities from the original state space $T$, which are scrambled during the reduction, can be restored and the corresponding fair acceptance cycles detected. In order to facilitate the implementation of the cycle detection we further unfold the annotated reduced state space into a threaded state space $h^*_G(T)$. Besides using the NDFS concept, the main novelty of our approach is to reduce the search for a fair acceptance cycle in the annotated state space $h_G(T)$ to a search of $N$ acceptance cycles in the threaded state space $h^*_G(T)$, where $N$ is the number of processes in the system. There is one to one correspondence between these acceptance cycles and the processes. Moreover, each of the cycles is weakly fair with regard to its corresponding process, meaning that the process either executes a statement along the cycle or it is disabled in some state of the cycle. The main benefit of such an approach is that one can avoid that the annotating permutations are kept as part of the representation of the state or on the DFS stack. The former is necessary in order to keep the advantages of the NDFS algorithm.

The unfolding into a threaded structure augments the size of the annotated state space roughly by a factor of $N$. The search for fair cycles contributes an additional factor of $N$, which gives in total $N^2$ times bigger structure than the reduced annotated state space. However, the gain in memory is substantial because the annotated state space is often a factor $N!$ smaller than the original
one. In fact, using efficient storage techniques \cite{16} one can show that in practice the factor $N^2$ can be avoided, i.e., often the threaded structure also can be stored in virtually the same memory size as the annotated state space.

In the literature we could find two algorithms for exploiting symmetry under weak fairness. The first one is from the already mentioned paper of Emerson and Sistla \cite{11}, while the second one is by Gyuris and Sistla \cite{14} and it is an improvement of the first algorithm. We show that the time complexity of our algorithm is the same as for the above mentioned algorithms. However, as the algorithms in \cite{11,14} require finding all strongly connected components (SCC) in the state space graph, our algorithm capitalizes on the advantages that the NDFS approach has over the SCC based algorithms. Probably the most important one among these advantages is that, unlike the two existing algorithms, our algorithm is compatible with the memory efficient approximative verification techniques like bit-state hashing \cite{15} and hash-compact \cite{24}. In practice, when the property which is being verified does not hold, the NDFS based algorithms are faster and require less memory to find an error. Also, with the NDFS based algorithms it is much easier to reconstruct the counter example execution which witnesses the error, by simply dumping the contents of the stack. Our algorithm is also easier to implement because it does not require complex structures to keep the annotating permutations.\footnote{For the algorithm from \cite{14} the authors claim that it is not necessary to keep the permutations as a part of the state. However, it is hard to see how this can be achieved without a significant time overhead.}

Chapter layout. In the next section we give the basic notions used throughout the chapter, as well as the standard NDFS algorithm. In Section 3 we describe the weak fairness algorithm and prove its correctness using the notion of weakly fair extension of labeled transition system. Section 4 introduces bisimulation preserving reduction. In this section we give the NDFS algorithm for model checking by exploiting these kind of reductions, without fairness. In the next section we first recall the basics of symmetry reduction and the theory of \cite{11}. After presenting the extensions of the theory we give the main algorithm that combines symmetry reduction with weak fairness, and show its correctness. Finally we discuss its space and time complexity. The last section summarizes the chapter and provides some guidelines for future work.

2 Preliminaries

2.1 Model-checking Problem, Labeled Transition systems and Bisimulations

In the sequel we adopt the automata-theoretic approach to model checking. In particular, we assume that the properties are given as Büchi automata \cite{21}. This allows us to reduce the model checking problem to the problem of finding a cycle.
in the finite graph representing the combination between the state space of the model and the property.

There are several ways to get this graph which depend on the languages in which the model and the property are specified, as well as on the formalisms used to represent their semantics. One typical way is the following:

Assume that we are given the model $M$ and the property $f$, represented in some modeling language and temporal logic, respectively. We want to check if all execution sequences (computations) of $M$ satisfy $f$. From the description of $M$, usually given as a parallel composition of processes $P_i$, we can obtain the global behavior of $M$ semantically represented by a graph $T$ whose nodes are the global states of $M$ and the edges are the transitions between them. The negation of $f$, $\neg f$, is translated into the Büchi automaton $A_{\neg f}$, which can be also regarded as a state space graph with a special set of designated states. In order to check the property we need the state space graph $G$ which is obtained as a product of $T$ and $A_{\neg f}$. The intuition is that $A_{\neg f}$ monitors $T$ by guessing nondeterministically the infinite execution sequences in $T$. $A_{\neg f}$ induces designated states also in $G$. In order for property $f$ to hold for $M$, the set of infinite execution sequences in $G$ that pass through infinitely many such designated states should be empty. Otherwise, there is an execution which is in accord with the negation of $f$, and therefore it is a counterexample that the property does not hold. As $G$ is finite, finding an infinite execution which is a counter example boils down to detecting cycles containing designated states.

The state space graph $T$ usually grows exponentially with the size of the description of $M$, which causes also an exponential growth of $G$. Building the whole $T$ can be avoided using on-the-fly techniques. In this approach we build only $A_{\neg f}$ before starting the construction of $G$. $A_{\neg f}$ is used to reduce the size of $G$ by generating only the part of $T$ which is needed for the product, i.e., which is in accord with $A_{\neg f}$. The property is checked simultaneously while building $G$. This increases the efficiency of the model-checking procedure because often in practice we can come up with a counterexample in an early phase of the state space graph exploration, which means that the complete graph $G$ does not have to be stored in the memory.

We refer the interested reader, for instance, to [7], for a formal description of the scheme given above. In the sequel we work directly with the final state space graph $G$ representing the product of the semantics of the model and the (negation of the) property, abstracting from the way it is obtained. From the model description or the property, we keep only the information which might be needed in the verification (e.g., process IDs). For a more detailed description how the composition can be done on-the-fly see for example [5, 20, 17].

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 6-tuple $T = (S, R, L, A, \hat{s}, F)$, where
- $S$ is a finite set of states,
- $R \subseteq S \times A \times S$ is a transition relation (we write $s \xrightarrow{a} s' \in R$ for $(s, a, s') \in R$),
- $L : S \rightarrow 2^{\text{Prop}}$ is a labeling function which associates with each state a set of atomic propositions that are true in the state,
- $A$ is a finite set of actions,
- $\hat{s}$ is the initial state,
- $F \subseteq S$ is the set of acceptance states.

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

As mentioned above, we specify a model as a collection of processes. To capture this in the semantics we assume that a finite sequence of processes $P_1, P_2, \ldots, P_N$, $N \geq 1$, is associated with the LTS. We introduce the mapping $\text{Pid} : R \rightarrow \{1, \ldots, N\}$, which assigns to each transition a process index. Intuitively, $\text{Pid}(s \xrightarrow{a} s') = i$ means that the transition $s \xrightarrow{a} s'$ is generated by some statement executed by the process $P_i$. An action $a$ is enabled in a state $s \in S$ iff $s \xrightarrow{a} s' \in R$ for some $s' \in S$. We say that the process $P_i$ is enabled in $s \in S$ iff there exists $s'$ such that $s \xrightarrow{a} s' \in R$ and $\text{Pid}(s \xrightarrow{a} s') = i$. An execution sequence or path is a finite or infinite sequence of subsequent transitions, i.e., for $s_i \in S$, $a_i \in A$, the sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$ is an execution sequence in $T$ iff $s_i \xrightarrow{a_i} s_{i+1} \in R$ 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 $c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n, n \geq 1$ is a cycle iff the start and end states coincide, i.e. $s_0 = s_n$. Given a finite or infinite execution sequence $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$, 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 $\sigma$ in $s_j$ iff $\text{Pid}(s_j \xrightarrow{a_j} s_{j+1}) = i$. A state $s$ is reachable iff there exists a finite execution sequence that starts at $s$ 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.

Next, we define bisimulation between two LTSs:

Definition 2. Given two LTSs $T_1 = (S_1, R_1, L_1, A, \hat{s}_1, F_1)$ and $T_2 = (S_2, R_2, L_2, A, \hat{s}_2, F_2)$, an equivalence relation $\mathcal{B} \subseteq S_1 \times S_2$ is called a bisimulation between $T_1$ and $T_2$ iff the following conditions hold:
- $\hat{s}_1 \mathcal{B} \hat{s}_2$;
- If $s \mathcal{B} s'$, then:
  - $L_1(s) = L_2(s')$;
  - $s \in F_1$ iff $s' \in F_2$;
  - Given an arbitrary transition $s \xrightarrow{a} s_1 \in R_1$, there exists $s_2 \in S_2$ such that $s' \xrightarrow{a} s_2 \in R_2$ and $s_1 \mathcal{B} s_2$;
  - The symmetric condition holds: Given an arbitrary transition $s' \xrightarrow{a} s_2 \in R_2$, there exists $s_1 \in S_1$ such that $s' \xrightarrow{a} s_1 \in R_1$ and $s_1 \mathcal{B} s_2$;
We say that $T_1$ and $T_2$ are bisimilar iff there exists a bisimulation between $T_1$ and $T_2$.

2.2 The Standard Nested Depth-First Search Algorithm

The algorithms presented in this chapter are based on the algorithm of [7] for memory efficient verification of LTL [9] properties, called nested depth-first search (NDFS) algorithm. The algorithm is implemented in the model-checker Spin [15]. In the rest of this section we give a brief overview of the NDFS algorithm.

We begin by considering the basic depth-first search algorithm in Fig. 1. When it is started in the initial state $\hat{s}$ of a given LTS $T$, the basic depth-first search algorithm generates and explores the reachable part of $T$, i.e., all reachable states and all transitions between them.

```
1 proc dfs1(s)
2   add s to Stack
3   add s to States
4   for each transition (s,a,s') do
5     add {s,a,s'} to Transitions
6     if s' not in States then dfs1(s') fi
7   od
8   delete s from Stack
9 end
```

Fig. 1. Basic DFS algorithm.

Note that in the algorithm in Fig. 1, as well as in all other algorithms in this chapter, we need to represent only the states of the LTS; the representation of the transitions is not needed. Another remark is that also no explicit representation of the depth-first search stack is needed. Thus, the variables Transitions, Stack, and the statements from lines 2, 5, and 8 are only used in the presentation and in some of the proofs of the algorithms.

The basic DFS cannot detect cycles. Therefore, in order to do model checking we extend it with a call to a procedure that checks for a cycle, as soon as an acceptance state is encountered. The new algorithm is given in Fig. 2.

The cycle check procedure is again a DFS which reports a cycle and stops the algorithm if the seed acceptance state is matched. If a cycle through the acceptance state does not exist, then the basic DFS is resumed at the point in which it was interrupted by the nested cycle check. In the sequel we also call the basic DFS first DFS, while the nested cycle checks all together are called second DFS. We need to work with two distinct copies of the state space in order to ensure that the second DFS does not fail to detect a cycle by cutting the search because it has hit a state already visited by the first DFS.
proc dfs1(s)
    add s to Stack1
    add s to States1
    if accepting(s) then States2:=empty; seed:=s; dfs2(s) fi
    for each transition (s,a,s') do
        add {s,a,s'} to Transitions
        if s' not in States1 then dfs1(s') fi
    od
    delete s from Stack1
end

proc dfs2(s) /* the nested search */
    add s to Stack2
    add s to States2
    for each transition (s,a,s') do
        add {s,a,s'} to Transitions2
        if s' == seed then report cycle
        else if s' not in States2 then dfs2(s') fi
    od
    delete s from Stack2
end

Fig. 2. Nested depth first search (NDFS) algorithm, version 1 (“preorder”).

An important feature of the algorithm in Fig. 2 is that before the cycle check is called its copy of the state space is reset in line 4 with the statement States2:=empty. Hence, the cycle check is always started from scratch. As a consequence, some states can be visited several times (by different cycle checks), which increases the time complexity of the algorithm. The reinitialization of States2 is needed, because otherwise some acceptance cycles can be missed. An example of an LTS for which the algorithm without reinitialization fails is shown in Fig. 3. The states $\hat{s}$ and $s_1$ are acceptance states. All the states are visited and put in States2 during the first cycle check which is started from $\hat{s}$. Thus, when the second cycle check is called from $s_1$, it is stopped at $s_2$ which was entered in

---

Fig. 3. A counterexample LTS for the NDFS with state space preservation.
States2 by the previous cycle check. As a result, the cycle $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_1$ is not reported.

In order to be able to preserve the States2 between cycle checks and reuse the previous calls of dfs2, we need a small modification of the algorithm – the cycle check should start only after all the successors of an acceptance state are explored, i.e., when the recursion retracts from an acceptance state [7]. This is achieved by moving line 4 to the end of the procedure dfs1, i.e., after line 8, and removing the statement States2 := empty which is not needed anymore.

Also, in the new version of the algorithm we use only one States (and Transitions) variable and extend the state representation with an additional bit to distinguish between the two different copies of the state space, belonging to the first and second DFS, respectively. The variable States is preserved between the calls of dfs1 and dfs2 and, as a result, each state is visited only once also during the second DFS. The resulting algorithm is shown in Fig. 4.

```
1 proc dfs1(s)
2   add s to Stack1
3   add {s,0} to States
4   for each transition (s,a,s') do
5     add {{s,0},a,{s',0}} to Transitions
6     if {s',0} not in States then dfs1(s') fi
7   od
8   if accepting(s) then seed:={s,1}; dfs2(s) fi
9   delete s from Stack1
10 end

11 proc dfs2(s) /* the nested search */
12   add s to Stack2
13   add {s,1} to States
14   for each transition (s,a,s') do
15     add {{s,1},a,{s',1}} to Transitions
16     if {s',1} == seed then report cycle
17     else if {s',1} not in States then dfs2(s') fi
18   od
19   delete s from Stack2
20 end
```

Fig. 4. Nested depth first search (NDFS) algorithm.

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

**Theorem 1 ([7]).** Given an LTS $T$, the NDFS algorithm in Fig. 4, when called on $s$, reports a cycle iff there is a reachable acceptance cycle in $T$. 

The cycle which is reported is contained in Stack2, while Stack1 contains the path from the initial state that leads to it. Thus, the counterexample execution can be reproduced by concatenating the contents of the stacks.

The fact that for each state $s$ the copies in the first and second DFS differ only in the second (bit) component can be used to save memory space [18]. 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.

The NDFS algorithm can be optimized by reporting a cycle when a state is visited which is already on the stack of the first DFS [18]. Unfortunately, this optimization does not work when NDFS is combined with partial order reduction [18].

3 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 $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$ is 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 $c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_0$ is 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.

3.1 Description of the Weak Fairness Algorithm

The weak fairness (WF) algorithm presented in the sequel is a modification of the algorithm implemented in the model checker Spin by Gerard Holzmann. It is a variant of Choueka’s flag algorithm [16].

Consider the following straightforward adaptation of the standard NDFS:

Each time the second DFS is resumed from a seed (acceptance) state, keep track which processes have contributed a transition or have been disabled so far, starting from the seed state. Whenever the seed state is matched, i.e., an acceptance cycle is detected, check if all the processes have contributed a transition or have been disabled in some state. If this is not true, then the cycle is not fair and simply ignore it, backtrack and
continue the search until a fair cycle is found or all acceptance states are checked without detecting a cycle.

This approach does not work because the NDFS algorithm guarantees to report at least one acceptance cycle (if there are any), but not all of them. Consequently, fair acceptance cycles could remain undetected.

To bypass the above described limitations 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 \( N \) copies of the original state set, where \( N \) is the number of processes. The intuition behind the copies is that when the algorithm works in the \( i \)-th copy of the state space, 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 \( \epsilon_i \).) 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 copy 1. Let us suppose that the acceptance states of the extended system are the acceptance states of the original system which reside in copy 1, and that the cycle check always starts in copy 1. Obviously, if a cycle goes through all \( N \) copies and arrives back to copy 1, then it is fair. Unfortunately, cycles through the acceptance states can be closed without leaving copy 1. A remedy for this problem is to ensure that from an acceptance state we can only go to a state which is in the next copy. Apparently, copy 1 (or any other copy) cannot in general satisfy this requirement because it is in conflict with the requirement that we pass to copy \( i + 1 \) only by taking transitions of process \( i \).

Therefore, we introduce a new special copy 0, which does not correspond to any process and to which we move the acceptance states. Copy 0 is inserted between copy \( N \) and copy 1, i.e., now we pass from copy \( N \) to copy 0, instead of copy 1. Whenever we are in an acceptance state in copy 0 we go via a special \( \tau \)-transition to copy 1. The \( \tau \)-transitions do not belong to any process and do not correspond to any transition from the original system. Note that the only possible transitions from an acceptance state of the extended state space are \( \tau \)-transitions. We stay inside copy 0 as long as transitions originating from a non-acceptance state are explored. The extended state space with copy 0 and its relation to the original state space is shown in Figure 5.

\( T \) 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 \( F_T \), while \( T_i, 1 \leq i \leq N \) 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 \( \text{Pid}(t) = i \ (\text{Pid}(t) \neq i) \) on the arrows between the copies express the fact that we can pass between the copies \( i \) and \( i + 1 \) (resp. stay inside copy \( i \)) by executing a transition belonging to process \( i \). One can see that in the
new setting all the cycles that pass through an acceptance state must also pass through all the copies and, consequently, they are all fair.

For representing the extended state space, we extend each original state with an integer counter that denotes the current copy of the state set. Formally, the extended state space is defined as follows:

**Definition 4.** Given an LTS $T = (S, R, L, A, \hat{s}, F)$ and processes $P_1, \ldots, P_N$, $N \geq 2$, we define its (weakly) fair extension to be the LTS $F(T) = (S_f, R_f, L_f, A \cup \{\tau, \epsilon_1, \ldots, \epsilon_N\}, \hat{s}_f, F_f)$ as:

- $S_f = S \times \{0, 1, \ldots, N\}$
- $R_f$ is defined with the following cases:
  1. $(s,0) \xrightarrow{a} (s',0) \in R_f$ iff $s \notin F$ and $s \xrightarrow{a} s' \in R$;
  2. $(s,0) \xrightarrow{\tau} (s,1) \in R_f$ iff $s \in F$;
  3. $(s,C) \xrightarrow{a} (s',C) \in R_f$ iff $C > 0$ and $s \xrightarrow{a} s' \in R$ and $\text{Pid}(s \xrightarrow{a} s') \neq C$;
  4. $(s,C) \xrightarrow{a} (s',(C + 1) \mod (N + 1)) \in R_f$ iff $C > 0$ and $s \xrightarrow{a} s' \in R$ and $\text{Pid}(s \xrightarrow{a} s') = C$;
  5. $(s,C) \xrightarrow{\epsilon_C} (s,(C + 1) \mod (N + 1)) \in R_f$ iff $C > 0$ and $P_C$ is disabled in $s$;
- for all $(s,C) \in S_f, L_f((s,C)) = L(s)$.
- $\hat{s}_f = (\hat{s},0)$
- $F_f = \{(s,0) \mid s \in F\}$. 

---

**Fig. 5.** The extended state space for the weak fairness algorithm.
The most direct way to do the model checking under weak fairness is to first unfold the original state space \( T \) into its weakly fair extension \( F(T) \) and then to apply the NDFS algorithm on \( F(T) \). 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 \( F(T) \).

The merge of the two stages is straightforward. The pseudo-code of the algorithm is given in Figure 6. The code is basically the standard NDFS model-checking algorithm modified for manipulation of the new counter component in order to generate the extended state space according to Definition 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.

The line

\[
\text{for each transition } (s,a,s') \text{ do}
\]

that occurs in the standard NDFS algorithm in Fig. 4 in the WF algorithm in Fig. 6 is expanded into

\[
\text{for each process } i = 1 \text{ to } N \text{ do}
\]

\[
\quad \text{nxt = all transitions enabled in } s \text{ with Pid(t)=i}
\]

\[
\quad \text{for all } (s,a,s') \text{ in nxt do}
\]

This refinement is necessary because in the WF algorithm the process identities play a significant role.

The other modifications that are added to the standard NDFS scheme are because of Definition 4, more precisely, to capture the transition relation \( R_f \). The \( \tau \)-transitions (case 2 of the definition) are generated with the true branch of the outermost \( \text{if} \) (lines 4-6 and 28-30). The increment of the counter component by 0 or 1 in the cases 3 to 5 is reflected in the line 9 and 33. The \( \epsilon \)-transitions (case 5) are implemented with the true branch of the \( \text{if} \) statement (line 11-13 and 35-38). Finally, case 0 is also implicitly implemented through line 9 and 33. As variable \( i \) is never 0, if \( C \) is 0, then the assignment \( C' = C \) is executed, which leaves the counter component unchanged.

### 3.2 Correctness of the Weak Fairness Algorithm

We first show that the WF algorithm is in fact the NDFS algorithm applied to the fair extension \( F(T) \). The main idea is to execute in lock-step the two algorithms and show that they produce the same state space.
proc dfs1(s,C)
  add {s,C} to Stack1
  add {s,C,0} to States
  if C == 0 and accepting(s) then
    add {{s,0,0},tau,{s,1,0}} to Transitions
    if {s,1,0} not in States then dfs1(s,1) fi /* tau move */
  else
    for each process i := 1 to N do
      if C == i then C' := (C+1) mod (N+1) else C' := C fi
      nxt := all transitions t enabled in s with Pid(t) == i
      if nxt == empty then /* epsilon move */
        if C == i then add {{s,C,0},epsilon,{s,C',0}} to Transitions fi
        if {s,C',0} not in States and C == i then dfs1(s,C') fi
      else
        for all (s,a,s') in nxt do
          add {{s,C,0},a,{s',C',0}} to Transitions
          if {s',C',0} not in States then dfs1(s',C') fi
        od
      fi
    od
    if C == 0 and accepting(s) then seed := {s,C,1}; dfs2(s,C) fi
    delete {s,C} from Stack1
  end

proc dfs2(s,C) /* the nested search */
  add {s,C} to Stack2
  add {s,C,1} to States
  if C == 0 and accepting(s) then
    add {{s,0,1},tau,{s,1,1}} to Transitions
    if {s,1,1} not in States then dfs2(s,1) fi /* tau move */
  else
    for each process i := 1 to N do
      if C == i then C' := (C+1) mod (N+1) else C' := C fi
      nxt := all transitions t enabled in s with Pid(t) == i
      if nxt == empty then /* epsilon move */
        if C == i then add {{s,C,1},epsilon,{s,C',1}} to Transitions fi
        if {s,C',1} not in States and C == i then dfs2(s,C') fi
      else
        for all (s,a,s') in nxt do
          add {{s,C,1},a,{s',C',1}} to Transitions
          if {s',C',1} not in States then dfs2(s',C') fi
        od
      fi
    od
    if C == 0 and accepting(s) then delete {s,C} from Stack2
  end

Fig. 6. Weak fairness (WF) algorithm.
Lemma 1. Given an LTS $T$, for every execution $E$ of the NDFS algorithm from Fig. 4, started in $(\hat{s}, 0) \in \mathcal{F}(T)$, there exists an execution $E'$ of the WF algorithm, started in $\hat{s} \in T$, such that the parts of $\mathcal{F}(T)$ which are generated and explored by $E$ and $E'$ are the same.

Proof. We will construct an execution $E'$ of the WF algorithm applied to $T$ while tracing the execution $E$ applied to $\mathcal{F}(T)$. We denote the $j$-th element from the bottom of the stack $s_i$, $i = 1, 2$, with $s_i(j)$, i.e., $s_i(0)$ is the bottom element. The function $\text{length}(s_i)$ returns the number of elements in $s_i$. Thus $s_i(\text{length}(s_i) - 1)$ is the top element. The superscript of a variable denotes the execution, i.e., algorithm, it belongs to.

We will show that at each point the following invariants hold:

- $\text{length}(s_i) = \text{length}(s_i'), i = 1, 2$, and $s_i(j) = s_i'(j), i = 1, 2, 0 \leq j \leq \text{length}(s_i') - 1$, and
- $\text{States}_i = \text{States}'_i$ and $\text{Transitions}_i = \text{Transitions}'_i$.

Initially, the invariants hold because both $E$ and $E'$ begin by adding $(\hat{s}, 0)$ to $\text{States}$ and $\text{Stack}$. Now, we advance $E$ and show how it can be mimicked by $E'$, while preserving the invariants. Suppose that at some point in $E$ a $\tau$-transition is generated and added to the state space. From the Def. 4 it follows that the state which is on the top of the $s_i$ and which is currently visited in by the NDFS, i.e., in the execution $E$, is an acceptance state. By the invariant, the same state is also on the top of the $s_i'$. Thus, as the counter component of this state is 0, because of the lines 4-6 and 28-30 in dfs1 and dfs2, respectively, the same $\tau$-transition is added to the $\text{States}'_i$. This is because the state which is generated by the $\tau$-transition is the same in both executions. Also, as the $\text{States}$ variables contain the same states, the search is continued via a recursive call to dfs1 (dfs2) in NDFS ($E$) if and only if the same is done in WF ($E'$).

Moreover, in both algorithms the same state is passed as an argument to the called procedure, which implies that the same state will be pushed in their $\text{Stack}$ variables by both algorithms.

In a similar way one can show using Def. 4 that one can mimic in $E'$ the $\epsilon$- (lines 10–12 and 35–37), as well as the ordinary transitions (lines 14–16 and 39–42). We assume that in the for each statements of NDFS the transitions from $s$ are taken in the same order as in the nested loops (for each and for all) in the WF algorithm. In other words, the processes are checked for enabled transitions in increasing order, starting at process 1, and the transitions of each process are chosen in the same order as in the statement

```
for all (s,a,s') in nxt do
```

in lines 14 and 39 of WF. \hfill $\Box$

It is worth noting that $C = k$, for some $1 \leq k \leq N$, implies that the processes with indices less than $k$ have been executed or been disabled in some state after the last “restarting” of the counting via a $\tau$-transition from some acceptance state.
The next step is to prove the correspondence between the fair acceptance cycles in the original state space and the acceptance cycles in its weakly fair extension.

**Lemma 2.** There exists a reachable fair acceptance cycle in $T$ iff there exists a reachable acceptance cycle in $F(T)$.

**Proof.** Let $c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ be a reachable fair acceptance cycle in $T$. Let us suppose without loss of generality that there is a path $p$ from the initial state $\hat{s}$ to $s_0$. By applying the definition of $R_f$ (Def. 4) $p$ can be mapped in $F(T)$ into a path $p'$ from $(\hat{s}, 0)$ to $(s_0, C)$, for some $C$.

We can continue extending $p'$ by mapping $c$, starting from $s_0$ and according to the definition of $R_f$. (Note that we might have to pass the cycle $c$ several times.)

The only ambiguity that can occur during the mapping is resolved such that the $\epsilon$-transitions (i.e. Case 5 in Def. 4) have priority over the ordinary transitions (i.e. Case 3 in Def. 4). More precisely, when we are extending from some state $(s, C)$ and the corresponding transition in $c$ is $s \xrightarrow{a} s'$, then if $P_C$ is disabled in $s$, the path in $F(T)$ is extended with $(s, C) \xrightarrow{\epsilon} (s, (C + 1) \mod (N + 1))$, otherwise the extension is $(s, C) \xrightarrow{a} (s', C)$. As $c$ is fair and contains acceptance states we will arrive eventually in $F(T)$ in some acceptance state $(s_a, 0)$. (The number of steps can be zero, if $(s_0, C)$ is already an acceptance state.) If we keep cycling and mapping along $c$, again because of the fact that $c$ is a fair acceptance cycle, we will come again across an acceptance state $(s_a, 0)$. If $s_a = s_a$, we are done because we have found the desired acceptance cycle in $F(T)$. Otherwise, we repeat the same procedure starting from $(s_a, 0)$. Because the number of acceptance states along $c$ is finite we will eventually end up in $F(T)$ in some acceptance state $(s_{a_k}, 0)$ that we have already visited. It is obvious from the above construction that the obtained acceptance cycle passing through $(s_{a_k}, 0)$ is reachable from the initial state $(\hat{s}, 0)$.

For the reverse direction, we show that each acceptance cycle $c$ from $F(T)$ is projected into a fair acceptance cycle $c'$ in $T$. The last property follows directly from the definition of $F(T)$. Each acceptance cycle in $F(T)$ contains a $\tau$-transition from some acceptance state $(s, 0)$ to the state $(s, 1)$. This transition is the only one from $(s, 0)$ and the counter component along the cycle can be changed only by increasing it modulo $N + 1$. Thus, in order to get back to $(s, 0)$ each process must execute a transition along the cycle (ordinary or $\epsilon$). The desired fair acceptance cycle in $T$ is obtained by simply omitting the counter component of the states in $c$ and eliminating the $\epsilon$- and $\tau$-transitions by merging the states which are connected via these. As $c$ is reachable in $F(T)$, some $(s, C)$ on $c$ is also reachable via some path $p$ in $F(T)$ which starts at the initial state $(\hat{s}, 0)$. The path $p$ too can be projected by omitting the counter components and the $\epsilon$-transitions into a path $p'$ in $T$ that leads to the state $s$ which is on $c'$.

Finally, Lemma 1, Lemma 2 and Theorem 1 put together give the correctness of the WF algorithm:
Theorem 2. Given an LTS $T$, the weak fairness algorithm (WF) from Figure 6, when called on $(\hat{s}, 0)$, reports a cycle if and only if there exists a reachable fair acceptance cycle in $T$.

4 Combining Symmetry Reduction and Weak Fairness

In this section we combine the ideas of the weak fairness algorithm from the previous section with reduction techniques that exploit symmetry. We first treat the simpler case of a NDFS algorithm for symmetry reduction without fairness. Using the approach of [12] we present this algorithm in the framework of bisimulation preserving reduction, which is a generalization of the symmetry reduction.

It should be emphasized once again that we assume that we work on a state space which is the product of the system model and the property, i.e., a Büchi automaton. In order to ensure bisimulation preservation, both the model and the property (Büchi automaton) have to satisfy certain conditions. Most of the time these conditions are just sufficient conditions which are efficiently checkable in practice, preferably on syntactic level [19, 10, 11]. In what follows we abstract from them and assume that they are captured in the selection function which we define below.

4.1 Bisimulation Preserving Reduction without Fairness

We begin by recalling some definitions and results from [12].

The main idea is to perform the model checking on an abstract state space, which is usually much smaller than the original one. To this end, the original state set $S$ is partitioned into equivalence classes, by means of some function $h : S \rightarrow S$, such that two states $s_1$ and $s_2$ are in the same class iff $h(s_1) = h(s_2)$. The abstract state space consists of the representatives of these classes with transitions between them as defined below.

Definition 5. Given a function $h : S \rightarrow S$ on LTS $T = (S, R, L, A, \hat{s}, F)$, we define the corresponding abstract LTS $h(T)$ to be $(h(S), h(R), h(L), A, h(\hat{s}), h(F))$, where

- $h(S)$, the set of representatives,
- $r_1 \xrightarrow{a} r_2 \in h(R)$ with $\text{Pid}(r_1 \xrightarrow{a} r_2) = i$ iff there exists $s \in S$ such that $r_1 \xrightarrow{a} s \in R$ with $\text{Pid}(r_1 \xrightarrow{a} s) = i$ and $h(s) = r_2$.
- for all $r \in h(S)$, $h(L)(r) = L(r)$, and
- $h(F) = h(F)$.

Usually, several different transitions from $T$ are represented by the same transition in $h(T)$. As a result, only the pids of the representative transitions are preserved, while those of the other transitions are lost by the reduction.

In order to obtain a correspondence between the acceptance cycles in $T$ and $h(T)$ we need to impose some additional constraints on the function $h$.

Definition 6. For a given LTS $T$, a function $h : S \rightarrow S$ is a selection function iff there exists a bisimulation $B \subseteq S \times S$ between $T$ and $T$ such that
Proof. 1. From \( \exists s \in S, sBh(s) \),
2. \( sBs' \) implies that \( h(s) = h(s') \).

In the sequel we assume that \( h \) is a selection function. We say that \( h \) preserves
the bisimulation relation \( B \). Intuitively, the function \( h \) picks a representative for
each equivalence class of \( S \) induced by \( B \).

The following result (Lemma 8 from [12]) is implied directly by the definitions
given above:

**Lemma 3 ([12]).** Given an LTS \( T = (S, R, I, A, s, F) \) and a selection function
\( h, T \) and \( h(T) \) are bisimilar.

As a consequence we can do the model checking in the reduced state space. In
the sequel we show how this can be done with a variation of the NDFS algorithm
applied to the reduced state space \( h(T) \). We begin with the following claim:

**Lemma 4.** Given an LTS \( T \) and a selection function \( h : S \to S \) for \( T \)
1. for each path \( p = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n \) in \( T \), there exists a corres-
dponding path \( q = r_0 \xrightarrow{a_0} r_1 \xrightarrow{a_1} \ldots r_{n-1} \xrightarrow{a_{n-1}} r_n \) in \( h(T) \), such that \( r_i = h(s_i) \),
   \( 0 \leq i \leq n \).
2. if \( h \) is a selection function, then for each path \( p = r_0 \xrightarrow{a_0} r_1 \xrightarrow{a_1} \ldots r_{n-1} \xrightarrow{a_{n-1}} r_n \)
in \( h(T) \) and every state \( s_0 \) such that \( h(s_0) = r_0 \), there exists a corresponding
path \( q = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n \) in \( T \), such that \( r_i = h(s_i) \), \( 0 \leq i \leq n \).

**Proof.** 1. From \( r_0 = h(s_0) \) by point 1 of Def. 6 we conclude that \( s_0B_{r_0} \). Thus,
   there exists \( s'_1 \in S \) such that \( r_0 \xrightarrow{a_0} s'_1 \in R \) in \( T \) and \( s'_1B_{s_1} \). By point 2
   of Def. 6 \( h(s'_1) = h(s_1) \). Thus, by Def. 5 there exists in \( h(T) \) a transition
   \( r_0 \xrightarrow{a_0} r_1 \), where \( r_1 = h(s_1) \). Using analogous arguments we can show by
   induction on \( i \) that for each transition \( s_i \xrightarrow{a_i} s_{i+1} \in R \) from \( p \) there exists a
   corresponding transition \( r_i \xrightarrow{a_i} r_{i+1} \in R_h \) from \( q, 1 \leq i \leq n-1 \), which proves
   the existence of \( q \).
2. By point 1 of Def. 6 \( h(s_0) = r_0 \) implies \( r_0B_{s_0} \). Since \( r_0 \xrightarrow{a_0} r_1 \), there exists
   \( s_1 \in S \) such that \( s_0 \xrightarrow{a_2} s_1 \) and \( r_1B_{s_1} \). Applying point 2 of Def. 6 we obtain
   \( h(s_1) = h(r_1) \). Thus, we need to show that \( h(r_1) = r_1 \). By point 1
   of Def. 6 we have that \( r_1Bh(r_1) \). Now we use the fact that \( r_1 \) is a unique
   representative in \( h(T) \) of its equivalence class (under \( B \)). More precisely, from
   the definition of \( h(T) \) (Def. 5) it follows that there exists a state \( s \) in \( T \) such
   that \( h(s) = r_1 \). By point 1 of Def. 6 \( sB_{r_1} \). Using again point 2 of Def. 6 one
   obtains \( h(s) = h(r_1) \), and consequently, \( h(r_1) = r_1 \). Continuing in this way,
   we construct the path \( q \) by finding for each transition \( r_i \xrightarrow{a_i} r_{i+1} \in R_h \) from
   \( p \), a corresponding transition \( s_i \xrightarrow{a_i} s_{i+1} \in R \), \( 1 \leq i \leq n-1 \).

\[ \square \]

**Lemma 5.** There exists a reachable acceptance cycle in the LTS \( T \) iff there
exists a reachable acceptance cycle in \( h(T) \).
Proof. Let \( c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_0 \) be a reachable acceptance cycle in \( T \) and let us suppose without loss of generality that \( s_0 \) is an acceptance state. Because \( c \) is reachable, also any state on \( c \), and therefore \( s_0 \), is reachable in \( T \). Thus, there exists in \( T \) a path \( p = \hat{s} \xrightarrow{b_0} q_1 \xrightarrow{b_1} \ldots q_{n-1} \xrightarrow{b_{n-1}} s_0 \). By point 1 of Lemma 4 there exists a path from \( h(\hat{s}) \) to \( h(s_0) \), and therefore \( h(s_0) \) is reachable in \( h(T) \).

Similarly, point 1 of Lemma 4 implies that \( c' = h(s_0) \xrightarrow{a_0} h(s_1) \xrightarrow{a_1} \ldots h(s_{n-1}) \xrightarrow{a_{n-1}} h(s_0) \) is a cycle in \( h(T) \). As by Def. 5 \( h(s_0) \) is an acceptance state, it follows that \( c' \) is an acceptance cycle in \( h(T) \).

For the reverse direction, let \( c = r_0 \xrightarrow{a_0} r_1 \xrightarrow{a_1} \ldots r_{n-1} \xrightarrow{a_{n-1}} r_n \), where \( r_0 = r_n \), be a reachable acceptance cycle in \( h(T) \). Like in the previous cases, without loss of generality let us assume that \( r_0 \) is an acceptance state.

By point 2 of Lemma 4 \( c \) can be mapped into a path \( p = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_0 \) in \( T \), such that \( r_i = h(s_i) \), \( 1 \leq i \leq n \). If \( s_0 \neq s_n \), i.e., the path \( p \) is not a cycle, we repeat the mapping of \( c \), starting this time from \( s_n \). We continue this unfolding of \( c \) in \( T \) until we close a cycle \( c' \) in \( T \). Obviously, this will eventually happen, because the number of times we repeat the mapping of \( c \) is limited by the number of states in the equivalence class of \( s_0 \) (which is finite because there are only finitely many states in an LTS). Moreover, \( c' \) contains \( s_0 \) or some other state from its equivalence class, therefore, an acceptance state.

It remains to prove that \( c' \) is reachable in \( T \). By point 2 of Lemma 4 the above described construction of \( c' \) can be started in any state \( s_0 \), such that \( r_0 = h(s_0) \). We show that at least one such a state (and therefore \( c' \) which is constructed starting from it) is reachable in \( T \). Because \( c \) is reachable, there exists a path in \( h(T) \) from the initial state \( h(\hat{s}) \) to \( r_0 \). By point 2 of Lemma 4 there exists a corresponding path in \( T \) from \( \hat{s} \) to some state \( s_0 \) such that \( r_0 = h(s_0) \). Thus, \( s_0 \) is reachable in \( T \). \( \square \)

We generate and search the reduced abstract state space for acceptance cycles with a straightforward modification of the standard (postorder) NDFS from Figure 4, which we call reduced NDFS (RNDFS). To this end the searched state space is made to comply with Def. 5. More precisely, RNDFS is obtained from the standard NDFS algorithm by replacing all the occurrences (except in lines 4 and 14) of the newly generated state \( s' \) with its representative \( h(s') \). The RNDFS algorithm is given in Fig. 7.

As the RNDFS algorithm is in fact NDFS applied to the reduced state space, its correctness follows directly from the correctness of the NDFS algorithm (Theorem 1) and the bidirectional cycle correspondence Lemma 5. Thus, we have the following claim:

**Theorem 3.** Given an LTS \( T \) and a selection function \( h \), the nested depth first search algorithm with bisimulation preserving reduction (RNDFS) in Figure 7, when called on (the initial state) \( h(\hat{s}) \), reports a cycle if and only if there is a reachable acceptance cycle in \( T \).

An important advantage of the standard NDFS algorithm is that the erroneous execution can be recreated by dumping the contents of the stack (\textit{Stack1}}
proc dfs1(s)
    add s to Stack1
    add {s,0} to States
    for each transition (s,a,s') do
        add {s,0},a,{h(s'),0} to Transitions
        if {h(s'),0} not in States then dfs1(h(s')) fi
    od
    if accepting(s) then seed := {s,1}; dfs2(h(s0,1)) fi
    delete s from Stack1
end

proc dfs2(s) /* the nested search */
    add s to Stack2
    add {s,1} to States
    for each transition (s,a,s') do
        add {{s,1},a,{h(s'),1}} to Transitions
        if {h(s'),1} == seed then report cycle
        else if {h(s'),1} not in States then dfs2(h(s')) fi
    delete s from Stack2
od

Fig. 7. Nested depth first search algorithm with bisimulation preserving reduction (RNDFS).

and Stack2. Unfortunately, this is no longer true with RNDFS. The reason is that in RNDFS dfs1 and dfs2 are not called with the newly generated states as arguments, but instead from their representatives (lines 6 and 16). Consequently, the stacks might contain a sequence of states which corresponds to an execution sequence that do not exist in the original LTS $T$. (This effect of introducing transitions between representative states in $h(T)$ is obvious from Def. 5.) In order to solve this problem, we modify the RNDFS algorithm such that dfs1 and dfs2 are called on the newly generated states and also the original states are saved on the stack instead of their representatives. However, in order to still benefit from the reduction only the representatives are saved in States, Transitions and the seed. The modified RNDFS (MRNDFS) is given in Fig. 8.

Notice that in fact the MRNDFS algorithm explores part of the original state space $T$, i.e., part of the original execution sequences, while building the abstract state space. As argued above, this is because always the original state is unfolded in the line

for each transition (s,a,s') do

and also the procedures dfs1 and dfs2 are called with the original state as argument, i.e. the original state is stored in Stack1 or Stack2. The exploration of the original state space is pruned whenever an equivalent state has already been explored.
Next, we show that the algorithms MRNDFS and RNDFS are equivalent in the sense that they produce (part of) the same LTS $h(T)$.

Lemma 6. Given an LTS $T$ and a selection function $h$, for every execution $E$ of the RNDFS algorithm on $h(T)$, started in $h(\hat{s})$, there exists an execution $E'$ of the MRNDFS algorithm on $T$, started in $\hat{s}$, such that the parts of $h(T)$ which are saved in the variables $States$ and $Transitions$ in both algorithms are the same.

Proof. The proof is similar to the proof of Lemma 1. We construct an execution $E'$ of the MRNDFS algorithm applied to $T$ while tracing the execution $E$ of RNDFS applied to $h(T)$. Using the same denotations as in the proof of Lemma 1, we define analogous invariants, i.e., we show that at each point the following holds:

1. $\text{length}(Stack_E^i) = \text{length}(Stack_{E'}^i), i = 1, 2, \text{ and } Stack_E^i(j) = h(Stack_{E'}^i(j)), i = 1, 2, 0 \leq j \leq \text{length}(Stack_E^i) - 1$, and

2. $States_E = States_{E'}$ and $Transitions_E = Transitions_{E'}$.

Initially, the invariants hold. This is because the execution $E$ begins by adding both $(h(\hat{s}), 0)$ to $States$ and $Stack$. The execution $E'$ in the very beginning also adds $h(\hat{s})$ to its $States$ after pushing $\hat{s}$ in its $Stack$. We show that the lockstep execution of $E$ and $E'$ preserves the invariants. Let $s_E$ and $s_{E'}$ be the states
NDFS for Symmetry Reduction

which are currently visited by \( E \) and \( E' \), respectively. The states are the top elements of the corresponding \( \text{Stack} \) variables, both in \( E \) and \( E' \). Thus, it follows by the first invariant that \( s_E = h(s_E') \). This implies by Def. 6 that there exists a bisimulation \( \mathcal{B} \) such that \( s_E \mathcal{B} s_{E'} \). Thus, there exists a transition which can be taken in the MRNDFS (i.e. \( E' \)), and which corresponds to the transition which is taken in the RNDFS (i.e. \( E \)). When the RNDFS algorithm chooses successor states in line 4, this can be done in any order. We assume that the transitions in the \texttt{for all} statements are chosen such that the following holds: if \( s \xrightarrow{a} s_1 \) is selected by MRNDFS before \( s \xrightarrow{a} s_2 \), the transition \( h(s) \xrightarrow{a} h(s_1) \) is selected by RNDFS before the transition \( h(s) \xrightarrow{a} h(s_2) \). The existence of the last two transitions is implied by the bisimilarity between \( s \) and \( h(s) \). Because the representatives of each equivalence class are unique, the same transition is added to \( \text{Transitions} \) in both algorithms. Note that in general MRNDFS can explore a different number of transitions in its \texttt{for all} statement than RNDFS in the same statement. This is because, in general, two bisimilar states do not have the same number of outgoing transitions labeled with the same action and leading to corresponding bisimilar states.\(^2\) However, this is not a problem because the extra transitions do not cause additional states and transitions to be saved in \( \text{Transitions and States} \). Thus, they can be executed in the \texttt{for all} iteration, for instance, after all the other transitions are explored, without any effect to \( \text{Stack, States and Transitions} \). The obtained successor states are bisimilar too and consequently \( \texttt{dfs1} \) and \( \texttt{dfs2} \) are called in the RNDFS iff they are called in MRNDFS. Moreover the argument states are bisimilar which preserves the elementwise bisimilarity of the \( \text{Stacks} \) and implies that the same states are added to \( \text{States} \) when the recursive call is entered. \( \square \)

As by Lemma 6 the MRNDFS algorithm boils down to the RNDFS algorithm (which is applied to the abstract state space \( h(T) \)), its correctness is a corollary of Theorem 3:

**Theorem 4.** Given an LTS \( T \), the modified nested depth first search algorithm with bisimulation preserving reduction (MRNDFS) in Figure 8, when called on \( \hat{s} \), reports a cycle if and only if there is a reachable acceptance cycle in \( T \).

### 4.2 Bisimulation Preserving Reduction with Weak Fairness

Unlike for the ordinary acceptance cycles, there does not exist a correspondence between the fair acceptance cycles in \( T \) and \( h(T) \). This can be seen on the following example \(^3\):

**Example 1.** Consider the LTS in Fig. 9. State \( s_0 = \hat{s} \) is the initial state and \( F = \{ s_1, s_4 \} \). For simplicity, the edges of \( T \) are labeled only with the transition \( \text{Pids} \). Obviously, the acceptance cycle \( s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_2 \rightarrow s_1 \) in \( T \) is fair, while

\(^2\) However, for symmetry reduction, that we introduce later, this number is always the same, because of the specific way the selection function \( h \) is chosen.

\(^3\) We owe this example to Dennis Dams
the reduced LTS $h(T)$, where $h(s_0) = s_0$, $h(s_1) = s_1$, $h(s_2) = s_2$, $h(s_3) = s_2$, $h(s_4) = s_1$, $h(s_5) = s_5$, and $h(s_6) = s_5$, given in Fig. 10, does not contain a fair acceptance cycle.

From the discussion above it is obvious that the straightforward approach of applying the (weak) fairness algorithm from Section 3 to $h(T)$ is not going to work.

Another straightforward way to model check under weak fairness utilizing the bisimulation preserving reduction is to incorporate the weak fairness requirement into the formula (automaton) that expresses the property. In that way the RNDFS algorithm can be used in its original form from Fig. 7. However, due to the additional conditions imposed on $h$ by the property component
of the states of $T$, this approach can be quite ineffective. For instance, it is known that for the case when the bisimulation is symmetry reduction (that we consider in a moment), such an approach does not give any reduction [11].

However, one can find satisfactory solutions for special cases of bisimulation preserving reductions. In the next section we present one such algorithm for symmetry reduction.

### 4.3 Symmetry Reduction with Weak Fairness

The algorithm for combining symmetry reduction with weak fairness that we present below is based on the theory developed in [11]. Here we do the necessary adjustments in order to integrate it with the NDFS cycle detection algorithm from Section 2.2.

Given a LTS $T = (S, R, L, A, \hat{s}, F)$ let $\text{Perm}(I)$ and $\text{Perm}(S)$ be the groups of permutations of the sets $I = \{1, \ldots, N\}$ of pids and $S$ of states, respectively. Both $\text{Perm}(I)$ and $\text{Perm}(S)$ are groups under the functional composition $\circ$ defined as: For any two permutations $\pi_1, \pi_2$, $\pi_1 \circ \pi_2 \overset{def}{=} \pi_1(\pi_2(x))$. We assume with $e$ the identity permutation and $\pi^{-1}$ is the inverse of $\pi$. Further, we assume that each permutation $\pi \in \text{Perm}(I)$ can be lifted into the permutation $\pi^*$ on the state set $S$. This assumption is quite natural regarding the way symmetry reduction is handled in practice (see for instance [19, 10, 3]). Formally, we require that there exists a mapping $(\cdot)^* : \text{Perm}(I) \rightarrow \text{Perm}(S)$ which maps each $\pi \in \text{Perm}(I)$ into $\pi^* \in \text{Perm}(S)$.

**Definition 7.** Given a LTS $T = (S, R, L, A, \hat{s}, F)$ and a mapping $(\cdot)^* : \text{Perm}(I) \rightarrow \text{Perm}(S)$, a subgroup $G$ of $\text{Perm}(I)$ is called a symmetry group of $T$ iff for all $\pi \in G$

- $s \xrightarrow{a} s' \in R$ iff $\pi^*(s) \xrightarrow{a} \pi^*(s') \in R$ and $\text{Pid}(\pi^*(s)) \xrightarrow{a} \pi^*(s') = \pi(\text{Pid}(s) \xrightarrow{a} s')$.
- For all $s \in S \pi^*(s) = \hat{s}$ iff $s = \hat{s}$
- For all $s \in S \text{L}(s) = \text{L}(\pi^*(s))$.
- $s \in F$ 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 $\Theta_G = \{(s_1, s_2) \mid s_1$ and $s_2$ are in the same orbit\}. From the definition of symmetry it is trivial to show that $\Theta_G$ is a bisimulation on $T$. Thus, it follows that each function $h : S \rightarrow S$ which has the following two properties

- $(s, h(s)) \in \Theta$.
- If $(s, s') \in \Theta$, then $h(s) = h(s')$

is a bisimulation preserving selection function on $T$. In what follows we assume that $h$ denotes a bisimulation selection function which satisfies the aforementioned two properties.

---

4 Roughly speaking, each atomic proposition in the property formula (Büchi automaton) should be invariant under $h$, in order for $\text{L}(s) = \text{L}(h(s))$ to hold.
4.4 Annotated LTS

As has been discussed above, the abstract LTS obtained according to Def. 5 cannot be used to detect fair acceptance cycles in the original LTS. The obvious reason for that is that in general a transition in $h(T)$ represents several transitions from $T$, which has as a consequence that the Pids of the original transitions are lost. For instance, in Fig. 10 transition $s_2 \rightarrow s_3$ and $s_3 \rightarrow s_2$ of $h(T)$ represents the transitions $s_2 \xrightarrow{1} s_3$ and $s_3 \xrightarrow{2} s_2$ of $T$ given in Fig. 9. In order to solve this problem, we define along the lines of [11] a less compressed version of the reduced LTS from Def. 5. Using some additional information about the transitions it is possible to recover the paths in the original LTS $T$. More precisely, it is possible by passing along an acceptance cycle in $h(T)$ to recover a corresponding fair acceptance cycle in $T$.

To this end first notice that for any two states $s, s' \in S$ belonging to the same orbit there can exist several permutations $\pi \in G$ such that $\pi^*(s) = s'$.

Given a state $s$ and its (orbit) representative $h(s)$, from the permutations $\pi$ such that $\pi^*(s) = h(s)$ we choose according to some criterion a (unique) canonical permutation $\pi_s$. For brevity, in the sequel, given a state $s$ and a permutation $\pi$, we write $\pi(s)$ instead of $\pi^*(s)$. One can say that $\pi_s$ is the encoding permutation which encodes state $s$ from $T$ into its representative $h(s)$ in $h_G(T)$, while $\pi_s^{-1}$ is the decoding permutation which recovers $s$ when applied to $h(s)$. In the new version of the abstract LTS we annotate the transitions with the decoding canonical permutation which we use later in order to regenerate paths of the original LTS $T$.

**Definition 8.** Given a LTS $T = (S, R, L, A, s, F)$ with a symmetry group $G$ and a selection function $h : S \to S$ we define the corresponding annotated quotient LTS $h_G(T)$ to be $(S_h, R_h, L_h, A_h, h(\hat{s}), F_h)$, where

- $S_h = h(S)$,
- $A_h = A \times G$,
- $r_1 \xrightarrow{a, \pi_s^{-1}} r_2 \in R_h \subseteq S_h \times A_h \times S_h$ iff $r_1 \xrightarrow{a} s \in R$ and $h(s) = r_2$, with $\text{Pid}(r_1 \xrightarrow{a, \pi_s^{-1}} r_2) = \text{Pid}(r_1 \xrightarrow{a} s)$. (Intuitively, the transition $r_1 \xrightarrow{a} s \in R$ is represented by $r_1 \xrightarrow{a, \pi_s^{-1}} r_2 \in R_h$.)
- for all $r \in S_h, L_h(r) = L(r)$, and
- $F_h = h(F)$.

**Example 2.** The annotated quotient LTS for the LTS $T$ in Fig. 9 is given in Fig. 11. (Recall that we label the transitions with their pids instead of actions.) There are only two processes in the system. The set $G = \{e, f\}$, where $e$ is the

---

5 For instance, consider the following situation. Let processes $i$, $j$ and $k$ be instances of the same program text and let all these processes have access only to local variables. Further, let in some state $s$ the program counters and the local variables of these processes have the same value. This means that except for the pids, the parts of the processes in the state description are the same. Obviously, the permutation which only swaps processes $i$ and $j$ will produce the same state as the permutation which only swaps $i$ and $k$. 

---
identity permutation, and $f$ is the “flip” permutation which maps 1 to 2 and vice versa. (Notice that $e^{-1} = e$ and $f^{-1} = f$.)

Using the permutations we are able to “unwind” the paths from $h_G(T)$ transition by transition into paths in $T$. Given a path $p = r_0 \xrightarrow{a_0,\pi_1} r_1 \xrightarrow{a_1,\pi_2} \ldots r_{n-1} \xrightarrow{a_{n-1},\pi_n} r_n$ we define the “cumulative” permutation along $p$ up to the state $r_i$.

\[ \Pi_{p,i} = \begin{cases} 
  \pi_1 \circ \pi_2 \circ \ldots \circ \pi_i, & 1 \leq i \leq n \\
  e, & i = 0 
\end{cases} \]

The following lemma from [11] establishes a more precise correspondence between the paths in $T$ and $h_G(T)$:

**Lemma 7 ([11]).**

1. If $p = r_0 \xrightarrow{a_0,\pi_1} r_1 \xrightarrow{a_1,\pi_2} \ldots r_{n-1} \xrightarrow{a_{n-1},\pi_n} r_n$ is a path in $h_G(T)$, then the path obtained by unwinding $p$,

\[ \text{unwind}(p) \overset{def}{=} r_0 \xrightarrow{a_0} \Pi_{p,1}(r_1) \xrightarrow{a_1} \ldots \Pi_{p,n-1}(r_{n-1}) \xrightarrow{a_{n-1}} \Pi_{p,n}(r_n) \]

where $\text{Pid}(\Pi_{p,i-1}(r_{i-1}) \xrightarrow{a_{i-1}} \Pi_{p,i}(r_i)) = \Pi_{p,i-1}(\text{Pid}(r_{i-1} \xrightarrow{a_{i-1},\pi_i} r_i))$, $1 \leq i \leq n$, is a path in $T$. (Notice that $r_0 = \Pi_{p,0}(r_0)$.)

2. If $p = r_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n$ is a path in $T$ starting at a representative state $r_0$, then there exists a path in $h_G(T)$

\[ q = r_0 \xrightarrow{a_0,\pi_1} r_1 \xrightarrow{a_1,\pi_2} \ldots r_{n-1} \xrightarrow{a_{n-1},\pi_n} r_n \]

such that $p = \text{unwind}(q)$.

Intuitively, an edge permutation $\pi_i$ can be seen as a “relative” decoding which applied to the representative state $r_i$ recovers the original state $s_i$, under the assumption that the processes are not permuted in the state $r_{i-1}$. In contrast, $\Pi_{p,i}$
is an “absolute” decoding which restores $s_i$ taking into account the cumulative effect of all permutations from the start state $r_0$.

Having the permutation on the edges, one can recover fair acceptance cycles in $T$:

**Example 3.** The acceptance cycle $s_1 \xrightarrow{1_2} s_2 \xrightarrow{1_3} s_2 \xrightarrow{1_2} s_2 \xrightarrow{1_6} s_1$, which is not fair in $h_G(T)$ (Fig. 11), is unwound into the fair acceptance cycle $s_1 \xrightarrow{1_2} s_2 \xrightarrow{1_3} s_3 \xrightarrow{2_1} s_2 \xrightarrow{1_2} s_1$ in $T$ (Fig. 9).

Given a cycle $c$ and some integer $n \geq 1$, then $c^n$ denotes the cycle obtained by concatenating $n$ copies of $c$. Given a path $p$ and a cycle $c$ we say that $p$ is along $c$ iff there exists an integer $n \geq 1$, and paths (possibly with length 0) $q, q'$ such that $c^n = qpq'$. If $p$ is a cycle, then we say that $p$ is a subcycle of $c$.

In order to establish a correspondence with the fair cycles in $h_G(T)$ called *subtly fair cycles* [11]. We would like to consider as fair those cycles in $h_G(T)$ that can be unwound into fair cycles in $T$ as implied by the following definition:

**Definition 9.** Cycle $c$ in $h_G(T)$ is subtly (weakly) fair iff for each state $r$ in $c$ and for each $i \in I$ there exists a path $p = r_0 \xrightarrow{a_0,\pi_1} r_1 \ldots \xrightarrow{a_{n-1},\pi_n} r_n \xrightarrow{a_n,\pi_{n+1}} r_{n+1}$ along $c$, such that

- $r = r_0$,
- process $j$, where $j = \Pi_{p,n}^{-1}(i)$, is disabled or executed in $r_n$ (i.e., $\text{Pid}(r_n) a_{n+1},\pi_{n+1}$ $r_{n+1} = j$).

In order to grasp the intuition behind the above definition first notice that, because the pids are scrambled in $h_G(T)$, the process with pid $i$ in $r_0$ appears “disguised” in each state $r_k$ ($0 \leq k \leq n + 1$) of $p$ as process with pid $l$, where $l = \Pi_{p,k}^{-1}(i)$. In other words, process $l$ is the encoding in $r_k$ of process $i$. Thus, if we take the pids in $r_0$ as a reference point, when we detect that process $j = \Pi_{p,n}^{-1}(i)$ is executed or disabled in $r_n$, this actually means that process $i$ (which we are tracing along $p$) is the one which is executed or disabled.

Another useful observation is that $j = \Pi_{p,n}^{-1}(i)$ is the same as $\Pi_{p,n}(j) = i$, i.e., the decoding of pid $j$ is $i$. By Lemma 7, the path $p$ in $h_G(T)$ from the definition above mimics a path $p'$ in the original LTS $T$, such that both $p$ and $p'$ begin in the same state $r_0$. Thus, it is really process $i$ which is executed or disabled in $p'$, but in the annotated LTS the same process occurs encoded as process $j$ (which is executed or disabled in the corresponding abstract path $p$).

**Note 1.** The requirement from Def. 9 is satisfied for each state in $c$ iff it is satisfied for some state in $c$. Suppose that the requirement is satisfied for some state $r$, and let $p' = r'_0 \xrightarrow{a'_0,\pi'_1} r'_1 \ldots \xrightarrow{a'_{k-1},\pi'_k} r'_k$ be a path along $c$ from some arbitrary state $r'$ to $r = r'_k$. Let $m = \Pi_{p',k}^{-1}(i)$, for a given $i \in I$. As $r$ satisfies the requirement of the definition, there is some path $p = r_0 \xrightarrow{a_0,\pi_1} r_1 \ldots \xrightarrow{a_{n-1},\pi_n} r_n \xrightarrow{a_n,\pi_{n+1}} r_{n+1}$ along $c$, where $r_0 = r$, such that $j = \Pi_{p,n}^{-1}(m)$.
is disabled or executed in \( r_n \). Obviously the path obtained by concatenating \( p' \)
and \( p \) satisfies the requirements of the definition for process \( i \) with regard to
\( r' \) because process \( j = \Pi_{p,n}^{-1}(\Pi_{p',k}^{-1}(i)) = (\Pi_{p',k} \circ \Pi_{p,n})^{-1}(i) = \Pi_{p,p,k+n}^{-1}(i) \)
is disabled or executed in \( r_n \).

The next two Lemmata follow directly from Def. 9 and Lemma 7:

**Lemma 8.** If a given cycle \( c = r_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} r_0 \in T \) is fair and
\( r_0 = h(r_0) \), i.e., \( r_0 \) is a representative state, then there exists in \( h_G(T) \) a subtly fair cycle \( c' \), such that \( c = \text{unwind}(c') \).

**Proof.** Let \( c = r_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} r_0 \) be a fair cycle in \( T \) beginning at the representative state \( r_0 \). By point 2 of Lemma 7 there exist a corresponding cycle in \( h_G(T) \)
\[
c' = r_0 \xrightarrow{a_0,\pi_1} r_1 \xrightarrow{a_1,\pi_2} \ldots r_{n-1} \xrightarrow{a_{n-1},\pi_n} r_0
\]
such that \( c = \text{unwind}(c') \). Because \( c \) is fair, for any \( i \in I \) there exist some
\( 0 \leq k < n \) such that process \( i \) is disabled or executed in the state \( s_k \) in \( c \). If process \( i \) is executed, then the definition of \( \text{unwind} \) (Lemma 7, item 1) implies that
process \( j = \Pi_{c,k}^{-1}(s_k) \) is executed in \( r_k = \Pi_{c,k}^{-1}(r_k) \). If process \( i \) is disabled in \( s_k \), then from the definition of symmetry (Def. 7) it follows that also process
\( j = \Pi_{c,k}^{-1}(i) \) is disabled in \( r_k \). Thus, we can conclude that that for the state \( r_0 \) in \( c' \)
for each \( i \in I \) there exists a path \( p \) along \( c' \) which satisfies the requirements of the
definition of subtly fair cycle (Def. 9). By Note 1 it follows that the requirements of the Def. 9 are also satisfied for all states of \( c' \).(\( \square \))

**Lemma 9.** If cycle \( c \) in \( h_G(T) \) is subtly fair, then there exists \( l \geq 1 \) such that the cycle \( \text{unwind}(c') \) is fair in \( T \).

**Proof.** Consider an arbitrary process \( i \in I \). It is clear that \( l \) can be chosen such that the path \( p \) from Def. 9 is a prefix of \( c' \). From Def. 9 we have that process
\( j = \Pi_{p,n}^{-1}(s_n) \) is disabled or executed in the state \( r_n \) in \( p \). If \( j \) is executed in \( r_n \), then the definition of \( \text{unwind} \) implies that \( i = \Pi_{p,n}(j) \) is executed in the state \( s_n = \Pi_{p,n}(r_n) \) from \( \text{unwind}(c') \). If \( j \) is disabled, then by the definition of symmetry, \( i = \Pi_{p,n}(j) \) is disabled in \( s_n \).(\( \square \))

Lemma 9 and 8 imply the following theorem:

**Theorem 5.** There exists in \( T \) a reachable fair acceptance cycle iff there exists in \( h_G(T) \) a reachable subtly fair acceptance cycle.

**Proof.** Let \( c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_0 \) be a reachable fair acceptance

cycle in \( T \), such that \( s_0 \) is not necessarily a representative state. Consider the
cycle obtained by applying the canonical permutation \( \pi_{s_0} \) to each transition of
\( c \): \( \pi_{s_0}(c) = \pi_{s_0}(s_0) \xrightarrow{a_0} \pi_{s_0}(s_1) \xrightarrow{a_1} \ldots \pi_{s_0}(s_{n-1}) \xrightarrow{a_{n-1}} \pi_{s_0}(s_0) \), with \( \text{Pid}(\pi_{s_0}(s_k) \xrightarrow{a_k} \pi_{s_0}(s_{k+1})) = \pi_{s_0}(\text{Pid}(s_k \xrightarrow{a_k} s_{k+1}))(1 \leq k < n) \). The cycle \( \pi_{s_0}(c) \) is in \( T \) by the
definition of symmetry (Def. 7). By the fairness of \( c \) for each \( i \in I \) the process
\( \pi_{s_0}(i) \) is enabled or executed in \( \pi_{s_0}(c) \), which implies that \( \pi_{s_0}(c) \) is also fair. As
\[\pi_{s_0}(s_0) = h(s_0)\] is a representative state the existence of a corresponding subtly fair cycle \(c'\) in \(h_G(T)\) (such that \(\pi_{s_0}(c) = \text{unwind}(c')\)) follows from Lemma 8. Obviously, both \(\pi_{s_0}(c)\) and \(c'\) contain acceptance states, thus \(c'\) is an acceptance cycle. Because \(c\) is reachable there exists a path in \(T\) from the initial state \(s\) to \(s_0\). By the definition of symmetry (Def. 7), \(s = h(\hat{s})\). Thus, the second part of Lemma 7 implies that there exists a corresponding path in \(h_G(T)\) from the initial state \(h(\hat{s})\) to \(h(s_0)\), i.e., that \(c'\) is reachable.

For the reverse direction, let \(c\) be an acceptance cycle in \(h_G(T)\). The existence of the corresponding cycle \(c'\) in \(T\) is given by Lemma 9. From the definition of the annotated LTS it easy to see that \(c'\) must contain acceptance states. In order to prove the reachability of \(c'\) in \(T\) consider the representative state \(r\) in which we begin the unwinding of \(c\) according to Lemma 9. As \(c\) is reachable there exists a path \(p\) in \(h_G(T)\) from \(h(\hat{s})\) to \(r\). Because \(h(\hat{s}) = \hat{s}\), by the first part of Lemma 7 there exists the path \(\text{unwind}(p)\) in \(T\) from \(\hat{s}\) to some state \(s\) such that \(r = \pi(s)\) for some \(\pi \in G\). By the definition of symmetry, \(\pi_s(p)\), defined analogously as \(\pi_{s_0}(c)\) above, is a path in \(T\) from \(\pi_s(\hat{s}) = \hat{s}\) to \(\pi_s(s) = r\). Thus, \(r\), and therefore \(c'\), is reachable in \(T\).

\[\square\]

### 4.5 Threaded LTS

Thanks to the preserved permutations we are able to establish a correspondence between the fair cycles. However, extracting the fair acceptance cycles directly from \(h_G(T)\) can still be difficult. This is mainly because during the state space search in \(h_G(T)\) we might have to pass several times through the same (representative) state. In order to get the intuition behind the problem consider the path \(p\) from Def. 9 which we construct for detecting some subtly fair cycle \(c\). Recall that because the pids are scrambled in \(h_G(T)\), process \(i\) (from the state \(r_0\)) which we are tracing along the path \(p\), in the state \(r_k\) \((0 \leq k \leq n + 1)\) appears “disguised” as process \(l = \Pi_{p,i}^{-1}(i)\). As in general \(\Pi_{k_1,p}^{-1}(i) \neq \Pi_{k_2,p}^{-1}(i)\) \((0 \leq k_1 < k_2 \leq n + 1)\), it can happen that two states \(r_{k_1}\) and \(r_{k_2}\) of \(p\) are the same, but process \(i\) has different encodings in each of them. Obviously, the different occurrences in \(p\) of the same state \(r\) from the cycle \(c\) are distinguished by the pid \(l\) of the process which is the encoding in \(r\) of process \(i\). In order to overcome this problem we define another, less compressed, version of \(h_G(T)\).

The new kind of LTS is a straightforward adaptation of the notion of threaded graph from [11]. The new threaded LTS physically implements the annotating permutations. To this end, the states are extended with a component which takes values from the set \(I\). In this way one can distinguish between the above mentioned different appearances of the same state along some path. In the new LTS there exists a transition between two states \((r_1, i), (r_2, j)\) if there is a transition between \(r_1\) and \(r_2\) in \(h_G(T)\) annotated with \(\pi\) and \(j = \pi^{-1}(i)\). As \(\pi^{-1}\) is the “relative” encoding permutation, this makes it easier to keep track of a particular process \(i\) in the reduced state space. Because the original states from \(T\) are scrambled in \(h_G(T)\), process \(i\) in \(r_1\) appears in \(r_2\) encoded as process \(j\). The second components of the states in \(h_G(T)\) reflect this transformation.
Definition 10. The threaded LTS associated with \( h_G(T) \) is the LTS \( h_G^*(T) = (S_h^*, R_h^*, L_h^*, A_h^*, \hat{s}^*, F_h^*) \), where

\( - S_h^* = (S_h \times I) \cup \{ \hat{s}^* \} \),
\( - A_h^* = A \cup \{ \tau \} \) (Note that, unlike \( A_h \), the actions of \( A_h^* \) do not contain permutations from \( G \) as a second component),
\( - \cdot \) for all \( i \in I \), \( \hat{s}^* \overset{a}{\rightarrow} (h(\hat{s}), i) \in R_h^* \subseteq S_h^* \times A_h^* \times S_h^* \) and \( \operatorname{Pid}(\hat{s}^* \overset{a}{\rightarrow} (h(\hat{s}), i)) = 0 \),
\( - (r_1, i) \overset{a}{\rightarrow} (r_2, j) \in R_h^* \subseteq S_h^* \times A_h^* \times S_h^* \) iff \( r_1 \overset{a, \pi}{\rightarrow} r_2 \in R_h \), \( j = \pi^{-1}(i) \), and \( \operatorname{Pid}((r_1, i) \overset{a}{\rightarrow} (r_2, j)) = \operatorname{Pid}(r_1 \overset{a, \pi}{\rightarrow} r_2) \) (Note that the transition pids are preserved),
\( - \) for all \( (r, i) \in S_h^* \), \( L_h^*((r, i)) = L_h(r) \), and \( L_h^*(\hat{s}^*) = L_h(h(\hat{s})) \).
\( - F_h^* = \{ (r, i) \mid r \in F_h \} \)

Note 2. The initial state \( \hat{s}^* \) is only used to represent that the search in \( h_G^*(T) \) can start nondeterministically in any copy \( (h(\hat{s}), i) \) of the initial state of \( h_G(T) \), and it does not play any role in the sequel. There are only transitions from but not to the initial state \( \hat{s}^* \). Also the pids of the transitions from the initial state could have been assigned arbitrarily.

Example 4. The threaded graph \( h_G^*(T) \) corresponding to \( h_G(T) \) from Fig. 11 is given in Fig. 12. Notice that \( h_G^*(T) \) is not symmetric in its pid labeling as

![Threaded graph example](image)

the original LTS \( T \) is. This is because the pids of the transitions from \( h_G(T) \) are preserved in \( h_G^*(T) \). In other words, the pids of the image transitions in \( h_G^*(T) \) are not affected by the annotating \( h_G(T) \). On the other hand, the second components of the states in \( h_G^*(T) \) do depend on the annotating permutations,
which causes the asymmetry. (Although in this particular case, for \( N = 2 \), the threaded LTS has greater size than the original LTS \( T \), in practice, for greater values of \( N \), the LTS \( h_T^N(T) \) is much smaller than \( T \). We revisit this issue later when we discuss the complexity of the algorithm which combines symmetry with fairness.)

Given a process \( i \) and a path \( p \) in \( h_T(T) \), we define in \( h_T^N(T) \) a corresponding “thread” \( q \), which is actually the path \( p \) whose states are extended such that their second components reflect the encodings of process \( \text{"thread"} \) \( q \). More precisely, if \( p \) begins in the state \( r_0 \), then \( q \) begins in \((r_0, i)\). In each state \((r_k, i_k)\) of \( q \) the second component \( i_k \) is the pid of the process which is the encoding in \( r_k \) of the process \( i \) from \( r_0 \).

The following lemma, analogous to Lemma 3.7 from [11], follows directly from Def. 10. It establishes the correspondence between the paths in \( h_T(T) \) and their threads in \( h_T^N(T) \):

**Lemma 10.** 1. If \( p = r_0 \xrightarrow{a_0, \pi_1} r_1 \xrightarrow{a_1, \pi_2} \ldots r_{n-1} \xrightarrow{a_{n-1}, \pi_n} r_n \) is a path in \( h_T(T) \) and \( i \in I \), then the path

\[
\text{thread}(p, i) \overset{\text{def}}{=} (r_0, i) \xrightarrow{\text{Pid}(r_0, \Pi_{p,1}^{-1}(\pi_1))} (r_1, I^{-1}(\pi_1, \Pi_{p,2}^{-1}(\pi_1))) \xrightarrow{\text{...}} (r_{n-1}, \Pi_{p,n-1}^{-1}(\pi_{n-1})) \xrightarrow{\text{...}} (r_n, \Pi_{p,n}^{-1}(\pi_n))
\]

with \( \text{Pid}(r_k, \Pi_{p,k}^{-1}(\pi_k)) = \text{Pid}(r_k) \xrightarrow{a_{k+1}} \Pi_{p,k+1}(\pi_k) \) for each \( k \) with respect to process \( i \).

2. If \( p = (r_0, i_0) \xrightarrow{a_0, \pi_1} (r_1, i_1) \xrightarrow{a_1} \ldots (r_{n-1}, i_{n-1}) \xrightarrow{a_{n-1}, \pi_n} (r_n, i_n) \) is a path in \( h_T^N(T) \), then there exists a corresponding path in \( h_T(T) \)

\[
q = r_0 \xrightarrow{a_0, \pi_1} r_1 \xrightarrow{a_1, \pi_2} \ldots r_{n-1} \xrightarrow{a_{n-1}, \pi_n} r_n
\]

such that \( p = \text{thread}(q, i_0) \).

**Proof.** 1. Let \( p \) and \( i \) be as defined in the lemma. We prove the existence of \( \text{thread}(p, i) \) by induction on the length of \( p \). The base case, when \( p \) consists of only one transition, i.e., \( n = 1 \), follows directly by Def. 10, by observing that \( \Pi_{p,1}^{-1}(\pi_1) = \pi_1^{-1} \). Suppose that there exists a thread with respect to \( i \) for each \( n = k \) and let \( p = r_0 \xrightarrow{a_0, \pi_1} r_1 \xrightarrow{a_1, \pi_2} \ldots r_k \xrightarrow{a_k, \pi_{k+1}} r_{k+1} \) be a path in \( h_T(T) \). By the induction hypothesis to the path \( p' = r_0 \xrightarrow{a_0, \pi_1} r_1 \xrightarrow{a_1, \pi_2} \ldots r_{k-1} \xrightarrow{a_{k-1}, \pi_k} r_k \) there corresponds the thread \( \text{thread}(p', i) \) in \( h_T^N(T) \) which terminates in the state \((r_k, \Pi_{p,k}^{-1}(\pi_k))\). By Def. 10 to the transition \( \xrightarrow{a_k, \pi_{k+1}} r_{k+1} \) in \( h_T(T) \) there corresponds in \( h_T^N(T) \) the transition \((r_k, \Pi_{p,k}^{-1}(\pi_k)) \xrightarrow{a_k} (r_{k+1}, \pi_{k+1}) \). It is easy to check that \( \pi_{k+1}^{-1}(\Pi_{p,k}^{-1}(\pi_k)) = \Pi_{p,k+1}^{-1}(\pi_{k+1}) \), which implies that indeed \( \text{thread}(p', i) \) can be extended into \( \text{thread}(p, i) \).

2. Using very similar arguments as for point 1 above, we show by induction on the length of \( p \) that for each state \((r_k, i_k)\) \((0 \leq k \leq n)\) in \( p \) there exists a path \( q \) in \( h_T(T) \) such that \( i_k = \Pi_{q,k}^{-1}(i_k) \). For \( n = 1 \) Def. 10 implies immediately that to \((r_0, i_0) \xrightarrow{a_0} (r_1, i_1)\) there corresponds the transition \( r_0 \xrightarrow{a_0} r_1 \) in \( h_T(T) \).
such that \( i_1 = \pi_1^{-1}(i_0) = \Pi_{q,1}^{-1}(i_0) \). Assume that the claim holds for \( n = k \). The induction hypothesis implies that there exists a path \( q' \) in \( h_G(T) \) that terminates in the state \( r_k \). By Def. 10 to the transition \((r_k, i_k) \overset{a_k}{\rightarrow} (r_{k+1}, i_{k+1})\) there corresponds the transition \( r_k \overset{\pi_1}{\rightarrow} r_{k+1} \) in \( h_G(T) \) such that \( i_{k+1} = \pi_{k+1}^{-1}(i_k) \). Thus, with this last transition \( q' \) can be extended into the path \( q \).

The induction hypothesis implies \( i_k = \Pi_{q,k}^{-1}(i_0) \). Thus, taking into account that \( \pi_{k+1}(\Pi_{q,k}^{-1}(i_0)) = \Pi_{q,k+1}^{-1}(i_0) \), it is obvious that \( q \) is the wanted path.

\[ \square \]

Intuitively, given a path \( p \) in \( h_G(T) \) and a pid \( i \), along \( \text{thread}(p, i) \) one can follow the evolution of \( i \) under the cumulative permutation starting at the initial state \( r_0 \) of the path \( p \). Notice that \( \Pi_{p,j}^{-1}(i) \) is exactly the encoding of the process \( i \) in the state \( r_j \) on \( p \).

The next step is to establish a correspondence between the subtly fair acceptance cycles from \( h_G(T) \) and the acceptance cycles in \( h_G^*(T) \). We first describe informally the basic idea. Each subtly fair cycle in \( h_G(T) \) contains \( N \) subcycles \( c_i \) (\( i \in I \)) such that (1) \( c_i \) corresponds to process \( i \) and (2) \( c_i \) is subtly fair with respect to process \( i \) in the sense that there exists a state \( r \) on \( c_i \) such that the process which is encoding of process \( i \) in \( r \) is executed or disabled in \( r \). Now, showing that the cycle \( c \) is subtly fair boils down to showing for each process \( i \in I \) that its corresponding subcycle \( c_i \) is subtly fair with respect to it. In order to be sure that in our check we cover all the processes from \( I \), i.e., that no two subcycles correspond to the same process, we require that the subcycles share a common (acceptance) state \( r' \). (One can consider this state as a reference point from which we start the paths \( p \) from the definition Def. 9.) As each subcycle \( c_i \) has a corresponding image (thread) \( c^*_i \) in \( h_G(T) \), we check the subtle fairness of \( c_i \) by checking that \( c^*_i \) is fair in the sense which is defined later.

Thus, from the discussion above it follows that we need new kinds of fairness. We begin with the following definition of a cycle which is fair with respect to one particular process:

**Definition 11.** Cycle \( c \) in \( h_G(T) \) is subtly (weakly) fair with respect to process \( i, i \in I \), iff there exists a state \( r \) of \( c \) and a path along \( c \) starting at \( r = r_0 \), \( p = r_0 \overset{a_0 \pi_1}{\rightarrow} r_1 \ldots \overset{a_n \pi_n}{\rightarrow} r_n \overset{a_{n+1} \pi_{n+1}}{\rightarrow} r_{n+1} \), such that process \( j = \Pi_{p,n}^{-1}(i) \) is disabled in \( r_n \) or it is executed in \( c \) in \( r_n \), i.e., \( \text{Pid}(r_n) = j \).

The following obvious result is essentially an alternative definition of a fair acceptance cycle in \( h_G(T) \):

**Lemma 11.** Cycle \( c \) in \( h_G(T) \) is subtly fair iff for each process \( i, i \in I \) there exists a subcycle of \( c, c_i \), such that \( c_i \) is subtly fair with respect to process \( i \).

**Proof.** Let \( c \) be a subtly fair cycle and let \( r \) be an arbitrary state in \( r \). By Def. 9 there exists a path \( p \) along \( c \), which begins in \( r \), as defined in Def. 9. Consider an arbitrary process \( i \). From the definition of subtly fair cycle with respect to a given process (Def. 11) it is clear that one can use the same state \( r \) and path \( p \)
in order to satisfy Def. 11 for \( c \). Thus, \( c \) is also subtly fair with respect to process \( i \). Since \( c \) is subcycle of itself by taking \( c_i = c \) we obtain the the desired subcycle for each \( i \in I \).

For the reverse direction, given a cycle \( c \) in \( h_G(T) \), let us assume that there exist subcycles \( c_i (i \in I) \) of \( c \) such that \( c_i \) is subtly fair with respect to process \( i \). Consider the cycle \( c_i \) for an arbitrary \( i \). By the definition of a subtly fair cycle with respect to a single process (Def. 11) there exists a state \( r \) of \( c \) and a path \( p \) along \( c \) which have the properties required in that definition. As \( c_i \) is a part of \( c \) the path \( p \) is also along \( c \). Taking into account Note 1, if such a path \( p \) exists from the state \( r \) on \( c \), then there is a path with the same properties for all states in \( c \). Therefore, Def. 9 is satisfied for process \( i \). As \( i \) was arbitrary chosen, the same argument can be repeated for all \( i \in I \), which implies that \( c \) is a subtly fair cycle.

\( \square \)

**Note 3.** Obviously cycle \( c \) in \( h_G(T) \) is subtly fair iff it is subtly fair with respect to each process \( i \in I \). Also, given a state \( s \) on \( c \) one can always choose the subcycles \( c_i \) such that \( s \) belongs to all of them.

In a similar way like for \( h_G(T) \), we need to adapt the definition of fairness to \( h_G^*(T) \):

**Definition 12.** Cycle \( c \) in \( h_G^*(T) \) is plainly fair with respect to process \( i \) iff there exist in \( c \) a state \((r, i)\) and a state \((r', j)\) in which process \( j \) is disabled or it is executed in \( c \).

The intuition is that the cycle \( c \) corresponds to the cycle \( c' \) in \( h_G(T) \) obtained by omitting the second components of the states in \( c \). In terms of Def. 9, we can construct the path \( p \) along the cycle \( c \) (i.e., \( c' \)), with \( r_0 = r \) (i.e, \((r, i)\)) and \( r_n = r' \), (i.e., \((r', j)\)). Obviously, \( j = \Pi_{p,n}^{-1}(i) \), as required by Def. 9.

It is easy to derive from Lemma 10 the following correspondence between cycles of \( h_G(T) \) and \( h_G^*(T) \) which are fair with respect to a particular process:

**Lemma 12.** For all \( i \in I \), cycle \( c \) in \( h_G(T) \) is a subtly fair with respect to process \( i \) iff there exists in \( h_G^*(T) \) a cycle which is plainly fair with respect to process \( i \).

**Proof.** Let \( c = r_0 \xrightarrow{a_0,\pi_1} r_1 \xrightarrow{a_1,\pi_2} \ldots r_{n-1} \xrightarrow{a_{n-1},\pi_n} r_0 \) be a subtly fair with respect to process \( i \) cycle in \( h_G(T) \). The path in \( h_G^*(T) \)

\[
\text{thread}(c,i) = (r_0,i) \xrightarrow{\pi} (r_1,\Pi_{p,1}^{-1}(i)) \xrightarrow{\pi} \ldots (r_{n-1},\Pi_{p,n-1}^{-1}(i)) \xrightarrow{\pi} (r_n,\Pi_{p,n}^{-1}(i))
\]

need not be a cycle because in general \( \Pi_{p,n}^{-1}(i) \neq i \). However, for every permutation \( \pi \in G \) there exists \( l \geq 1 \) such that \( \pi^l = e \). Consequently, for some \( l \) it holds \( (\Pi_{p,n}^{-1}(i))^l(i) = i \). This implies that \( \text{thread}(c',i) \) (which begins at \((r_0,i)\)) is a cycle in \( h_G^*(T) \). Because \( c \) is subtly fair with respect to process \( i \), there exists some \( k \) such that process \( j = \Pi_{c',k}^{-1}(i) \) will be executed or disabled in some state \( r_k \) in \( c' \). If process \( j \) is executed in \( r_k \), then by item 1 of the path correspondence
Lemma 10 it is clear that also process $j$ is executed in the state $(r_k, j)$ of the thread of $c'$. By the definition of the threaded LTS (Def. 10) process $j$ is disabled in $r_k$ iff it is disabled in $(r_k, j)$. Thus, the thread of $c'$ is plainly fair with respect to process $i$.

Conversely, let us assume that $c = (r_0, i_0)^{a_0} (r_1, i_1)^{a_1} \ldots (r_{n-1}, i_{n-1})^{a_{n-1}} (r_0, i_0)$, for an arbitrary $i = i_0$, is a plainly fair cycle in $h^*_G(T)$. By Lemma 10 there exist a cycle $c'$ such that $c = \text{thread}(c', i)$. Because $c$ is plainly fair with respect to $i$ there exists $k$ such that process $i_k$ is executed or disabled in $(r_k, i_k)$. By repeating the reasoning from the proof of the path correspondence Lemma 10 one can show that $i_k = \Pi_{i'}^{-1} (i_0)$, and consequently $i_k$ will play the role of $j$ from the definition of subtly fair cycle (Def. 9). By the definition of thread it follows that if process $i_k$ is executed in $(r_k, i_k)$, then $i_k$ is executed in the state $r_k$ of $c'$. By the definition of threaded LTS we have that process $i_k$ is disabled in $r_k$ iff it is disabled in $(r_k, i_k)$. Thus, $c'$ is subtly fair with respect to process $i$. \qed

Example 5. Consider the cycle $c = s_1 - s_2 - s_2 - s_2 - s_1$ of $h_G(T)$ in Fig. 11. Its subcycle $c_1 = s_1 - s_2 - s_1$ is fair with respect to process 1, while $c$ itself is subtly fair with respect to process 2. In the threaded LTS $h^*_G(T)$ (Fig. 12) the cycle $(s_1, 1) - (s_2, 1) - (s_1, 1)$, which is plainly fair with respect to process 1, and the cycle $(s_1, 2) - (s_2, 2) - (s_2, 2) - (s_1, 2)$, which is plainly fair with respect to process 2, correspond to $c_1$ and $c$, respectively.

Finally, combining the previous results one can establish the following theorem which is the basis of our algorithm. It states the cycle correspondence between the original LTS $T$ and the corresponding threaded LTS $h^*_G(T)$.

**Theorem 6.** There exists a reachable fair acceptance cycle in $T$ iff for some reachable acceptance state $s$ in $T$ there exists in $h^*_G(T)$ for each $i \in I$ an acceptance cycle $c_i$ which is plainly fair with respect to process $i$ and contains the (acceptance) state $(h(s), i)$.

**Proof.** Assume that there exist a reachable fair acceptance cycle in $T$. By Theorem 5 there exists a reachable subtly fair acceptance cycle $c$ in $h_G(T)$. By Lemma 11 this is equivalent with the existence of $h_G(T)$ of $N$ subcycles of this acceptance cycle, each corresponding to a particular process. (Note that in this direction we did not need explicitly the acceptance state $s$.) By Lemma 12 to each subcycle $c_i$, which is subtly fair with respect to process $i$ ($i \in I$), there corresponds a thread in $h^*_G(T)$ which is plainly fair with respect to $i$.

For the reverse direction assume that there exists in $h^*_G(T)$ a set of $N$ cycles $c_i$, such that $c_i$ is plainly fair with respect to process $i$ ($i \in I$) and each of them contains the acceptance state $(h(s), i)$. According to Lemma 12 for each $c_i$ there exists a subtly fair cycle $c'_i$ in $h_G(T)$. From Lemma 10 it follows that $h(s)$ is in $c'_i$. Now Lemma 11 implies that $c'_i$ can be combined into one subtly fair acceptance cycle in $h_G(T)$. As $s$ is reachable in $T$, it follows by Lemma 7 (point 2) that $h(s)$ is also reachable in $h_G(T)$, i.e., $c'$ is reachable in $h_G(T)$. Finally, Theorem 5 implies the existence of a reachable fair acceptance cycle in $T$. \qed
4.6 An Algorithm for Symmetry Reduction with Weak Fairness

From the previous theorem it is clear that we need an algorithm for finding for each process $i \in I$ an acceptance cycle in $h^*_G(T)$ which is plainly fair with respect to it. This can be done by modifying the cycle check in the NDFS algorithm. Thus, starting in each acceptance state $s$, instead of checking for one (plainly fair) acceptance cycle, we need to check for $N$ acceptance cycles passing through $s$.

Finding a Cycle which is Plainly Fair with respect to one Process. We first present an algorithm which can be used for checking for a cycle which is plainly fair with respect to one particular process. Such a cycle check boils down to a relatively simple graph algorithm, given in Fig. 13, which is a variant of (preorder) nested depth first search. Started at the root node (in our case: an acceptance state) $r$, the algorithm finds a cycle which passes through $r$ and which contains at least one special node (in our case: a state in which process $i$ is disabled or a state which is generated via a transition by process $i$).

```plaintext
/* seed == r, where r is the root node */
1 proc dfs2(s)
2   add s to Stack1
3   add {s,0} to Nodes
4   for each transition (s,a,s') do
5     add {{s,0},a,{s',0}} to Edges
6     if special(s') then
7       if {s',1} not in Nodes then dfs3(s') fi
8       else
9         if {s',0} not in Nodes then dfs2(s') fi
10     fi
11   od
12   delete s from Stack1
13 end

14 proc dfs3(s)
15   add s to Stack2
16   add {s,1} to Nodes
17   for each transition (s,a,s') do
18     add {{s,1},a,{s',1}} to Edges
19     if s' == seed then report cycle
20     else if {s',1} not in Nodes then dfs3(s',1) fi
21   od
22   delete s from Stack2
23 end
```

Fig. 13. Finding a cycle passing through the initial state and a special node.
The procedures \texttt{dfs2} and \texttt{dfs3} work in separate state spaces. The cycle check is started by calling \texttt{dfs2} at the root node \( r \). If a special node \( s \) is encountered, then \texttt{dfs3} is called. The search continues in the second state space until (the copy of) \( r \) is matched (which means that the desired cycle is detected), or all states reachable from \( s \) are generated, without detecting a cycle, in which case the control is returned back to \texttt{dfs2}. Unlike in the standard NDFS algorithm, in the algorithm in Fig. 13 \texttt{seed} is a constant equal to \( r \). The following theorem states formally the correctness of the algorithm:

**Lemma 13.** Given a graph \( G \), a node \( r \) of \( G \), and a set of special nodes \( S \) (which is a subset of the nodes of \( G \)), the algorithm in Fig. 13, when called on \( r \), reports a cycle iff there exists a cycle in \( G \) which contains \( r \) and at least one special node \( s \in S \).

**Proof.** It is clear that if the algorithm reports a cycle, then it has found a path back to the root \( r \). As the cycle is reported only by \texttt{dfs3}, and as \texttt{dfs3} is called only when a special node is generated, it follows that the detected cycle contains at least one node \( s \in S \).

For the opposite direction, let us suppose that there exists a cycle which contains \( r \) and some state \( s \in S \). Let \( Q \subseteq S \) be the set of special nodes which are in some cycle passing through \( r \). First notice that for each node which is generated in \texttt{dfs2} all its successors are explored either by calling \texttt{dfs2} or \texttt{dfs3} (lines 6-10). This and the correctness of the DFS algorithm, guarantees that, unless the algorithm is stopped by reporting a cycle, each state reachable from \( r \) will be generated at least once (by \texttt{dfs2} or \texttt{dfs3}, or by both). Therefore, also all states in \( Q \) will be generated. Moreover, at least one node from \( Q \) will be generated by \texttt{dfs2}. Suppose that this is not true, i.e., suppose that the nodes in \( Q \) are only generated by \texttt{dfs3}. For each such a node \( s \in Q \) there exists a path back to \( r \). On the other hand, \texttt{dfs3} is called on some special node \( s' \in S \) generated by \texttt{dfs2}. As there is a path from \( r \) to \( s' \) and from \( s' \) to \( s \), obviously \( s' \) also must be in \( Q \), which is a contradiction.

Now, let \( s \) be the node in \( Q \) which is the first (in preorder) generated by \texttt{dfs2}. We show that a cycle through \( s \) will be reported. Suppose, this is not true, i.e., that \texttt{dfs3} is wrongly truncated on some node \( s' \) from which there exists a path back to \( r \). This is only possible if \( s' \) has been generated by some previous cycle check from another state \( s'' \in S \). This implies that there exists a cycle passing through \( s'', s', \) and \( r \), i.e., \( s'' \in Q \). As \( s'' \) must have been generated by \texttt{dfs2} before \( s \), this is a contradiction. \hfill \( \Box \)

**A NDFS Algorithm for Symmetry Reduction with Weak Fairness.**

Now that we have the cycle check algorithm we are ready to proceed with the algorithm for symmetry reduction under weak fairness (RWF), given in Fig. 14. In analogy with the weak fairness (WF) algorithm, we do the search for the plainly fair cycles sequentially, starting from process 1. If a cycle for process \( i \) is found, then we continue by starting the search for a cycle corresponding to
process $i + 1$. In accord with Theorem 6, the algorithm reports a fair acceptance cycle in $T$ if it finds for each $i$ a corresponding cycle $c_i$.

The algorithm is a version of the standard NDFS algorithm – a kind of double nested depth first search. The first DFS is performed by calling $\text{dfs1}$ in the initial state of $h_G(T)$. The cycle check is implemented by a straightforward adaptation of the algorithm from Fig. 13. The procedures $\text{dfs2}$ and $\text{dfs3}$ in Fig. 13 correspond to $\text{dfs2}$ and $\text{dfs3}$, respectively in RWF. Nodes and Edges are renamed into States and Transitions, respectively. Like in the weak fairness algorithm, as the process identity plays a crucial role, the for statement from line 4 of the algorithm in Fig. 6 is expanded into two nested for iterations (lines 14 and 20 in the RWF algorithm). The function special from line 6 in Fig. 6 is refined into a check if the process is disabled or executed (lines 17 and 21, respectively). Function $\text{perm}(s', C1)$ returns the value $\pi_{s'}(C1) = (\pi^{-1})_{s'}(C1)$ (in accord with Def. 10). The cycle check is started in postorder for each acceptance state by calling $\text{dfs2}$.

The cycle check for a particular process $i$ is launched by calling $\text{dfs2}$ in the acceptance state $(s, i)$. Procedure $\text{dfs2}$ works in a copy of $h_G^*(T)$ corresponding to process $i$. Analogously to the WF algorithm, each state is extended with a counter that denotes the process to which the copy corresponds. If a state is generated in which process $i$ is disabled or executed (lines 17 and 21, respectively), then $\text{dfs3}$ is called. The latter operates in a copy of $h_G(T)$ (instead of $h_G^*(T)$) corresponding to process $i$, which is also denoted by a counter added to the state. This small optimization is trivially justified by the following result, which is proved along the same lines as Lemma 1 from [14]:

**Lemma 14.** Let there exist a path from $(s, j)$ to $(s', j')$ in $h_G^*(T)$. Then there exists a path in $h_G^*(T)$ from $(s', j')$ to $(s, j)$ iff there exists a path from $s'$ to $s$ in $h_G(T)$.

**Proof.** The second part of the path correspondence lemma Lemma 10 implies that if there exists a path from $(s', j')$ to $(s, j)$ in $h_G^*(T)$, then there is also a path from $s'$ to $s$ in $h_G(T)$.

For the opposite direction, first notice that existence of a path from $(s, j)$ to $(s', j')$ in $h_G^*(T)$ implies by the path correspondence Lemma 10 (point 2) that there is a path from $s$ to $s'$ in $h_G(T)$ as well. Combined with the assumption that there exists a path from $s'$ to $s$, this gives us a cycle $c$ in $h_G(T)$ from $s$ back to itself via $s'$. In the same way as in the proof of the cycle correspondence Lemma 12 one can show that there exists $l \geq 1$ such that the thread$(c', j)$ (which begins in $(s, j)$) is a cycle in $h_G(T)$. This cycle has as a prefix the thread of the path from $s$ to $s'$, which means that it contains both $(s, j)$ and $(s', j')$. Thus, there exists a path from $(s', j')$ to $(s, j)$. \hfill $\Box$

The search continues in $h_G(T)$ until (a copy of) the acceptance state $s$ is matched (which means that the desired cycle is detected), or all states reachable from $s$ are generated, without detecting a cycle. (In the latter case the control is returned back to $\text{dfs2}$.) If a cycle plainly fair with respect to process $i$ is found the algorithm reports a success only if $i = N$ (line 39), otherwise the cycle check
continues with the next process by calling dfs2 with process $i + 1$ and state space copy $i + 1$ as arguments (line 40).

The correctness of the RWF algorithm is given by the following claim:

**Theorem 7.** Given a LTS $T$ the algorithm for reduction under weak fairness (RWF) in Fig. 14 when called in $h(\hat{s})$ reports a cycle iff there exists a reachable fair acceptance cycle in $T$.

**Proof.** Suppose that the algorithm reports a cycle. From the correctness of the general DFS algorithm (c.f. [1]) and the structure of the RWF algorithm, it is clear that in that case the algorithm has found in each pair of copies of $h_G$ and $h^*_G$, i.e., for each $i \in I$, a plainly fair cycle $c_i$ which contains $(r, i)$, where $r$ is some acceptance state in $h_G(T)$. By Theorem 6 this implies the existence of a fair acceptance cycle in $T$.

Assume that there exists a fair acceptance cycle in $T$. Theorem 5 guarantees that there exists a corresponding subtly fair acceptance cycle in $h_G(T)$, while Theorem 6 implies the existence of the desired cycles $c_i$ in $h_G(T)$ for all $i \in I$. We show that one set of such cycles will be detected by the RWF algorithm. Let $r$ be the acceptance state which is the first in postorder in $h_G(T)$ (i.e., the first one deleted from Stack1 in the RWF algorithm) such that there exists a subtly fair acceptance cycle $c$ in $h_G(T)$ which contains it. We show that when the RWF algorithm starts the cycle set check from $r$, in each pair of state space copies corresponding to process $i$ none of the states from the plainly fair cycle $c_i$ (corresponding to $c$) is already entered in the state space. As a consequence, the algorithm will detect $c_i$ by generating these states. The proof is by contradiction and it is similar to the proof [7, 5] of the standard NDFS algorithm. Assume that the claim is not true, i.e., that the cycle check for some process $i$ is wrongly truncated because a state $(r'', C_1, C_2)$ or $(r'', C_2)$ has been found which already was in States. Lemma 13 ensures that such a state could not be generated by the current cycle check. Thus the problematic state must have been generated by some previous cycle check, i.e., a previous call of dfs2 or dfs3, from some acceptance state $r'$ in $h_G(T)$. From the path correspondence results one can see that this is true if and only if there exists a path $p_1$ from $r'$ to $r''$ in $h_G(T)$. On the other hand, as there is a path $p_2$ between $r''$ and $r$ (which is a part of the cycle $c$), this also means that there exists a path from $r'$ to $r$ (the concatenation of $p_1$ and $p_2$). Combining this with the fact that $r'$ is deleted from Stack1 before $r$, we conclude from the properties of the DFS that $r$ must have been generated by dfs1 before $r'$. (This means that $r$ has been on Stack1 when $r''$ was visited by the nested search from $r'$.) In graph-theoretic terminology we say that $r$ is an ancestor of $r'$. This implies that there exists a path $p_3$ between $r$ and $r'$ (see, for instance, Lemma 1 of [7]) (This path was contained in Stack1 at the moment when $r''$ has been visited from $r'$.) Because $c$ is subtly fair, the concatenation of the paths $p_1$, $c$ (starting at $r''$ via $r$ and back to $r''$), $p_2$ and $p_3$ is also a subtly fair acceptance cycle containing $r'$ (see Fig. 15). This is in contradiction with our choice of $r$ to be the first acceptance state removed from Stack1 that is in a subtly fair acceptance cycle. \qed
Fig. 14. Reduction under weak fairness (RWF) algorithm.
4.7 Complexity of the RWF Algorithm

We compare the complexity of the RWF algorithm with the complexity of the algorithms of Emerson and Sistla (ES95) [11] and Gyuris and Sistla (GS97) [14]. As we already mentioned, 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 mainly refer to the former for comparison. The GS97 algorithm is on-the-fly and it is based on the algorithm for finding all maximal strongly connected components from [1].

It was already mentioned that in our algorithms we need to store only states, while transitions are used only in the proofs. Therefore in the sequel we will consider the number of states as the size, denoted as $|T|$, of a given LTS $T$. In our space complexity calculations we assume that the memory needed for the DFS stack is much smaller than the memory which is used for the States. In practice, this is often a reasonable assumption.

A straightforward analysis of the RWF algorithm shows that we need $N$ copies of the threaded LTS $h_G^*(T)$ (in the dfs2 part of the cycle check for each of the $N$ processes) and $N+1$ copies of the annotated reduction $h_G(T)$ – one copy for the first depth first search with dfs1 and $N$ copies for the dfs3 part of the cycle check. From Def. 10 one can see that the size of $h_G^*(T)$ is $O(N \cdot |h_G(T)|)$. This gives us $O((N^2 + N + 1) \cdot |h_G(T)|)$, i.e., $O(N^2 \cdot |h_G(T)|)$ for the space complexity of RWF. As $h_G(T)$ is usually much smaller (by a factor close to $N!$) than the original LTS $T$ the gain in reduction with the RWF algorithm is obvious.

Moreover, in practice the real memory requirements are much smaller, because there is an efficient way of storing the states of the copies of $h_G^*(T)$ and $h_G(T)$. (Recall that we do not have to store transitions.) The storage technique is due to Gerard Holzmann and it is used in the implementation of the weak fairness algorithm used by the model checker Spin. The technique is a generalization of the trick with the two bits described in Section 2.2 for the original NDFS algo-
Algorithm. It uses the fact that the copies of each state $s$ in $h_G(T)$, $(s, 0), \ldots, (s, N)$ differ only in their counter components. This means that we can represent all $N+1$ copies by keeping the description that corresponds to $s$ plus an additional array $flag$ of $N$ bits to differentiate the copies of $s$. The bit $flag[C]$ is set iff the state $(s, C)$ is in the state space. In this way, instead of $(N + 1) \cdot |s|$ bits, where $|s|$ is the size of the description of $s$, we need only $|s| + N + 1$ bits for all copies of $s$. Because in practice $|s| \gg N + 1$, the memory complexity is virtually reduced to $O(|h(T)|)$. With a similar reasoning we can show that we need $|s| + N^2$ bits to represent all copies of a given state in $h_G^*(T)$, instead of $N^2 \cdot |s|$ with the straightforward approach.

GS97 uses several extra data structures of which two integers are essential. For computing of the maximal strongly connected components, the GS97 algorithm has to keep two special unique numbers for each state. Therefore $2 \cdot \log|h_G(T)|$ extra bits are needed in the state space description. The above described efficient storage technique is not used, but even if it had been used, in general $2N\log|h_G(T)|$ extra bits would have been needed for the two unique numbers for each of the $N$ copies of the state. Thus, even with this minimal assumed overhead, for systems where $2 \cdot \log|h_G(T)| > N$ our algorithm will have shorter description of the state vector. For instance, for $\log|h_G(T)| = 20$, i.e. around $10^6$ states even with 39 processes the overhead in the RWF algorithm will be smaller. In practice most of the time we have a much smaller number of processes, while state spaces of $10^6$ are often encountered.

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. [19, 3]). 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.

The time complexity of the algorithm is also dictated by the sizes of $h_G(T)$ and $h_G^*(T)$, except that we have to take into account also the transitions. From Def. 10 it is clear that each transition $r_1 \overset{a, \pi}{\rightarrow} r_2$ from $h_G(T)$ induces at most $N$ transitions in $h_G^*(T)$ (one transition from each copy $(r_1, i)$). Thus, also with the transitions included in the size of the state space $|h_G^*(T)|$ is $O(N \cdot |h_G(T)|)$. By repeating the reasoning for the space complexity we obtain that the time complexity of the RWF algorithm is $O(N^2 \cdot |h_G(T)|)$. (Recall that we assume that the canonical representatives are computed in constant time.) This is the same complexity as for the GS97 algorithm.

As the GS97 algorithm requires finding maximal strongly connected components, RWF has all the advantages that NDFS has over the maximal strongly connected components. Probably the most important among those is that, unlike GS97, RWF is compatible with the approximative verification techniques like bit-state hashing [15] or hash-compact [24].
Another advantage of the RWF algorithm is more efficient error detection. Intuitively, it is much easier (faster and within less memory) to find an acceptance cycle than to identify a whole maximal strongly connected component which contains an acceptance state. Also with RWF it is easier to reconstruct an execution which leads to the error.

The annotating permutations in the RWF algorithm are generated as a byproduct of the algorithm for finding representatives. For instance, with virtually no time penalty the algorithm from [3] can be extended to also produce the permutation $\pi_s$ for a given state $s$. In GS97 the permutations (as well as several other extra data structures) are saved as a part of the state description. Although it is mentioned in [14] that keeping the permutations is not necessary, because they can be always recalculated, it is unclear if this can be done without a significant time overhead.

5 Conclusion and Future Work

We presented an efficient algorithm for model checking under weak fairness using reduction based on symmetry and proved its correctness. To this end we first discussed an algorithm for model checking under weak fairness. As the second intermediate step we gave an algorithm for model checking with bisimulation preserving reduction without fairness. The correctness of these two algorithms was also proved. Finally, we presented an extension of the theory developed by Emerson and Sistla [11] in order to fit the concept of the nested depth first search.

An important future task is to prove compatibility of our algorithm with other state space reduction techniques. In that way we can use the combination to obtain better state space reduction. The compatibility of partial order with the bisimulation preserving reduction can be shown along the lines of [12]. It will be more challenging to reconcile symmetry reduction and partial order reduction under weak fairness.

Recently we have been extending the model checker Spin with symmetry reduction with encouraging results [3]. As part of this extension, we intend to try both the bisimulation preserving reduction algorithm (MRNDFS) and the reduction under weak fairness (RWF) algorithm in practice.

It is straightforward to show that the algorithm for bisimulation preserving reduction is compatible with the discrete-time extension of Spin from [2]. We conjecture that this is the case also with the RWF algorithm. The RWF algorithm might also be of interest for dense-time systems regarding the problem of solving non-zenoness from [4] that relies on weak fairness.

Acknowledgments. The author would like to thank Dennis Dams for his detailed proofreading, the numerous examples, and the invaluable suggestions, Jos Baeten for the proofreading and the idea for a more elegant proof of Lemma 4, as well as Leszek Holenderski, Gerard Holzmann, and Doron Peled for their helpful comments regarding the questions treated in this chapter.
References

16. G. J. Holzmann, Personal communication.
