A Nested Depth First Search Algorithm for Model Checking with Symmetry Reduction

Dragan Bošnački

Dept. of Math. and Computer Science, Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
phone: +31 40 247 5159, fax: +31 40 246 8508, dragan@win.tue.nl

Abstract. We present an algorithm for the verification of properties of distributed systems, represented as Büchi automata, which exploits symmetry reduction. The algorithm is developed in the more general context of bisimulation preserving reductions along the lines of Emerson, Jha and Peled. Our algorithm is a modification of the nested depth first search (NDFS) algorithm by Courcoubetis, Yannakakis, Vardi and Wolper. As such, it has the standard advantages (memory and time efficiency) that NDFS shows over the state space exploration algorithms based on maximal strongly connected components in the state space graph. In addition, a nice feature of the presented algorithm is that it works also with multiple (non-canonical) representatives for the symmetry equivalence classes. Also, instead of an abstract counter-example, our algorithm is capable of reproducing a counter-example which exists in the original unreduced state space, which is an important feature for debugging.

Keywords: Model checking, state space reduction techniques, symmetry reduction, nested depth first search algorithm, multiple representatives.

1 Introduction

Over the last decades, the complexity of computer systems has been increasing rapidly, with a tendency towards distribution and concurrency. The correct functioning of these complex systems is becoming an ever larger problem. Many verification and proof techniques have been invented to solve this problem. An important class of techniques are those based on a fully automatic, exhaustive traversal of the state space of a concurrent system. Well-known representatives are the various model-checking techniques.

An infamous problem complicating an exhaustive traversal of the state space of a concurrent system is the state explosion, caused by the arbitrary interleaving of independent actions of the various components of the system. Several techniques have been developed to cope with this problem. Among those techniques symmetry reductions [15,8,4] are one of the most successful. They exploit the inherent symmetry of the model which is present in many systems like: mutual exclusion algorithms, cache coherence protocols, bus communication protocols, etc. After observing that the symmetry in the description of the model results in a symmetric state space, the key idea is to partition the state space into equivalence classes of (symmetric) states. Then, the state space exploration can be performed in the usually smaller quotient state space that consists only of (representatives of the) equivalence classes.

The problem of finding canonical, i.e., unique, representatives of equivalence classes is also known as the *orbit problem.* The orbit problem is equivalent to the graph isomorphism problem[4], for which no polynomial algorithm is known. As a result, often with symmetry reduction the verification time can become critical. On the other hand, finding multiple (non-canonical) representatives usually boils down to sorting algorithms [8,3]. An obvious drawback of the multiple representatives is that they provide less state space reduction compared to the canonical representatives. However, often in practice it turns out that, with an acceptable increasing of the state space, the verification time can be improved significantly by using multiple representatives [15,3].

In this paper we present an algorithm that exploits symmetry reduction for model checking. We describe the algorithm in the more general setting of bisimulation preserving reductions, along the lines of Emerson, Jha and Peled [10]. As symmetry reduction is just a special case of a bisimulation preserving reduction, all results are valid for the former too.

We assume that in general we are checking liveness properties specified as Büchi automata [16].[1] 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 special states called *acceptance* states) in the graph representing the product of the system (model) state space with the automaton representing the (complement of the) property (c.f. [6,5]). Our algorithm is a modification of the nested depth first search (NDFS) algorithm by Courcoubetis, Yannakakis, Vardi and Wolper [6] which reports an acceptance cycle if and only if there is one in the state space graph.

We present two versions of the algorithm both based on NDFS. The first version is a straightforward adaptation to NDFS of the algorithms used in the literature for state space exploration with symmetry reduction (e.g. [15,10]). We show that in general the algorithm provides a counter example for the checked property which is in the abstract state space and as such contains transitions which are not in the original state space. Therefore, we present the second version of the NDFS algorithm which is in fact exploring (part of) the original state space. As a consequence the found erroneous executions are always "realistic" in the sense that they exist in the original state space too.

Unlike the other algorithms for explicit state model checking which are known in the literature, our algorithm does not require unique (canonical) representatives of the bisimulation (symmetry) equivalence classes. As discussed above this can lead to shorter verification times.

[1] This means that the algorithm can be used for temporal logics which can be translated into Büchi automata, like, for instance, linear-time temporal logic (LTL) [7].

In general, our algorithm features all advantages that NDFS has over the algorithms for state space exploration which are based on maximal strongly connected components (MSCC) [1]. Probably the most important one among these advantages is that, unlike the existing algorithms for symmetry reduction, our algorithm is compatible with the memory efficient approximative verification techniques like bit-state hashing [12] and hash-compact [18]. 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. Finally, the NDFS algorithms are in principle easier to implement than the MSCC based ones.

Related work other than [10]. This paper can be seen as a follow up of [3]. However, in [3] only safety properties are considered and the emphasis is on heuristics for coping with the orbit problem. In this paper we "reuse" the heuristics from [3] and focus on the algorithm for generating the reduced state space for liveness properties.

The usefulness of multiple representatives is discussed in several papers (e.g. [15,4]). However, these papers deal either only with safety properties or they are in the context of symbolic model checking. To the best of our knowledge there is no paper which tackles this issue in the area of explicit-state model checking for liveness properties.

In a recent paper [14] a nested depth first search algorithm was implicitly assumed, but in a context of a specific reduction which exploits so called heap symmetries. Also, [14] assumes only canonical representatives.

2 Preliminaries

In the sequel we adopt the automata theoretic approach to model checking [17]. In particular, we assume that the properties that are checked are given as Büchi automata. As it was mentioned in the introduction, we work with the graph obtained as a product of the state space graph representing the system (more precisely: the system model) with the automaton (graph) representing the negation of the property. The algorithms to obtain the product state space are quite standard (c.f. [6,5]). In the sequel we assume that the product state space is given. (An on-the-fly integration of the algorithms for obtaining the product state space with the algorithms presented in this paper should be trivial (c.f. [5]).)

We represent the state space of the system which is checked as a *labeled transition system* formally defined as follows:

Definition 1. *Let Π 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 $(s, a, s') \in R$),*

- $L : S \to 2^{\Pi}$ *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 paper.

An action a is *enabled* in a state $s \in S$ iff $s \xrightarrow{a} s' \in R$ for some $s' \in S$. 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 $i \geq 0$. A state s is *reachable* iff there exists a finite execution sequence that starts at $\hat{s}$ and ends in s. 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. $s_0 = s_n$. 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* $\equiv \subseteq S_1 \times S_2$ *is called a* bisimulation *between* T_1 *and* T_2 *iff the following conditions hold:*

- $\hat{s}_1 \equiv \hat{s}_2$*;*
- *If* $s \equiv s'$*, 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* $s_1 \equiv s_2$*;*
  - *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* $s_1 \equiv s_2$*;*

We say that T_1 *and* T_2 *are* bisimilar *iff there exists a bisimulation between* T_1 *and* T_2*.*

2.1 Standard Nested Depth-First Search Algorithm

The algorithms presented in this paper are based on the algorithm of [6] for memory efficient verification of liveness (temporal) properties, called nested depth-first search (NDFS) algorithm. In the rest of this section we give a brief overview of the NDFS algorithm of [6].

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.

Note that in in an implementation of the algorithm in Fig. 1, as well as in all other algorithms in this paper, we need to save only the states of the LTS – saving transitions is not needed.

```
proc dfs1(s)
   add s to Stack
   add s to States
   for each transition (s,a,s')  do
     add {s,a,s'} to Transitions
     if s' not in States then dfs1(s') fi
   od
   delete s from Stack
end
```

Fig. 1. Basic DFS algorithm.

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.

```
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").

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.

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.

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 [6]. 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. 3.

```
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. 3. Nested depth first search (NDFS) algorithm.

The following claim (Theorem 1 from [6]) establishes the correctness of the algorithm

Theorem 1 ([6]). *Given an LTS T, the* NDFS algorithm *in Fig. 3, when called on $\hat{s}$, 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 [13]. The states $(s, 0)$ and $(s, 1)$ can be stored together as (s, b_1, b_2), where b_1 (respectively b_2) is a bit which is set to 1 iff $(s, 0)$ (resp. $(s, 1)$) has been already visited during the first (resp. second) DFS. Thus, NDFS requires virtually the same memory as if the verification was done with only one copy of the state space.

3 Bisimulation Preserving Reduction

In this section we recall some definitions and results from [10]. Our starting point is the idea to perform the model checking in 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. The abstract state space consists of (not necessarily unique) representatives of these classes, chosen by a function h, with transitions between them as defined below.

Definition 3. *Given a function $h : S \to S$ on LTS $T = (S, R, L, A, \hat{s}, F)$, we define the corresponding* abstract LTS *$h(T)$ to be $(S_h, R_h, L_h, A, h(\hat{s}), F_h)$, where*

- *$S_h = h(S)$, the set of representatives,*
- *$r_1 \xrightarrow{a} r_2 \in R_h$ iff there exists $s \in S$ such that $r_1 \xrightarrow{a} s \in R$ and $h(s) = r_2$.*
- *for all $r \in S_h, L_h(r) = L(r)$.*
- *$F_h = h(F)$.*

In order to preserve the properties of interest that hold in T also in $h(T)$ we need to impose some additional constraints on the function h. In particular, we require that the equivalence class induced by the partitioning is a bisimulation.

Definition 4. *For a given LTS T, a function $h : S \to S$ is a* selection function *iff there exists a bisimulation $\equiv \subseteq S \times S$ on T (i.e., between T and T) such that for all $s \in S$*

- *$s \equiv h(s)$, and*
- *$h(s) = \hat{s}$ iff $s = \hat{s}$.*

Intuitively, the function h picks one or more representatives for each equivalence class of S induced by $\equiv$. It should be emphasized that in the definition of selection function in [10] h is required to satisfy the additional property

$$s \equiv s' \text{ implies that } h(s) = h(s')$$

which in the sequel we call *canonicity requirement.* Obviously, with this requirement each equivalence class has a canonical (unique) representative. In what follows we assume that h is a selection function without the canonicity requirement, which implies multiple representatives per equivalence class.

We say that h *preserves* the bisimulation relation $\equiv$. The following result is implied by the definitions given above:

Lemma 1 ([10]). *Given an LTS T and a selection function h, T and $h(T)$ are bisimilar.*

Lemma 1 is actually Lemma 8 from [10]. The proof of that lemma in [10] is valid in our setting too because it does not use the canonicity requirement for the representatives. The following lemma, is implied directly by Lemma 1. It gives the path correspondence between the original and the abstract LTSs.

Lemma 2. *Let T be an LTS with a selection function h, and let $h(T)$ be the corresponding abstract LTS. If $\equiv$ is a bisimulation which is preserved by h, then the following path correspondence holds. For any $s_0, r_0 \in S$ such that $s_0 \equiv r_0$, there exists a finite (resp. infinite) path $p = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ...$ in T iff there exists a finite (resp. infinite) path $q = r_0 \xrightarrow{a_0} r_1 \xrightarrow{a_1} ...$ in $h(T)$ and $s_i \equiv r_i$ for all i.*

A direct consequence of Lemma 1 is the following result:

Lemma 3. *There exists a reachable acceptance cycle in the LTS T iff there exists a reachable acceptance cycle in $h(T)$.*

Proof. Let $c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ...s_{n-1} \xrightarrow{a_{n-1}} s_0$ 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 $p = \hat{s} \xrightarrow{b_0} q_1 \xrightarrow{b_1} ...q_{n-1} \xrightarrow{b_{n-1}} s_0$. Let $\equiv$ be the bisimulation that corresponds to h. Then by the definition of selection function $\hat{s} \equiv h(\hat{s})$. Thus, by Lemma 2 there exists a path from $h(\hat{s})$ to some state $r_0 \equiv s_0$. So, r_0 is a reachable acceptance state in $h(T)$. By Lemma 2 to the cycle c in T there corresponds in $h(T)$ a path $p^0 = r_0 \xrightarrow{a_0} r_1 \xrightarrow{a_1} \ldots r_{n-1} \xrightarrow{a_{n-1}} r_0^1$ which is not necessarily a cycle. As $s_0 \equiv r_0^1$, we can unfold p again according to Lemma 2, but this time beginning at r_0^1, which will produce the path $p^1 = r_0^1 \xrightarrow{a_0} r_1^1 \xrightarrow{a_1} \ldots r_{n-1}^1 \xrightarrow{a_{n-1}} r_0^2$. In an analogous way we can repeat this unfolding arbitrary many times which will give us a sequence of paths $p^0, p^1, p^2, \ldots$ with the corresponding end states $r_0^1, r_0^2, r_0^3 \ldots$. As the equivalence class of the (acceptance) states $r_0^0 = r_0, r_0^1, r_0^2, ...$ is finite, for some $i, j \geq 0$ we will have that $r_0^i = r_0^j$. Obviously, the concatenation of the paths p^i to p^{j-1} is an acceptance cycle in $h(T)$.

Using symmetric arguments one can also prove the other direction. □

4 NDFS Algorithms for Bisimulation Preserving Reduction

We generate and search the reduced abstract state space for acceptance cycles with a straightforward modification of the standard (postorder) NDFS from Figure 3, which we call reduced NDFS (RNDFS). To this end the searched state space is made to comply with Def. 3. 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. 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,{h(s'),0}} to Transitions
6       if {h(s'),0} not in States then dfs1(h(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,{h(s'),1}} to Transitions
16      if {h(s'),1} == seed then report cycle
17      else if {h(s'),1} not in States then dfs2(h(s')) fi
18    delete s from Stack2
19    od
20 end
```

Fig. 4. Nested depth first search algorithm with bisimulation preserving reduction (RNDFS).

In order to show the correctness of the RNDFS algorithm we first show that it is equivalent to the NDFS algorithm applied on the reduced state space. In fact the generation of $h(T)$ from T according to Def. 3 and the NDFS on $h(T)$ are done simultaneously in RNDFS.

Lemma 4. *Given an LTS T and a selection function h, the nested depth first search algorithm (NDFS) in Figure 3 when called on $h(\hat{s})$ and applied on $h(T)$ reports a cycle iff the nested depth first search algorithm with bisimulation preserving reduction (RNDFS) in Figure 4, when called on the initial state $h(\hat{s}) = \hat{s}$ and applied on T reports a cycle.*

Proof. We will show that for every execution E of the NDFS algorithm on $h(T)$, started in $h(\hat{s})$, there exists an execution E' of the RNDFS 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. Additionally, if E reports a cycle then also E' reports a cycle.

We will construct an execution E' of the RNDFS algorithm applied to T while tracing the execution E of NDFS applied to $h(T)$. We denote the j-th element from the bottom of the $Stack_i$, $i = 1, 2$, with $Stack_i(j)$, i.e., $Stack_i(0)$ is the bottom element. The function $length(Stack_i)$ returns the number of elements in $Stack_i$. Thus $Stack_i(length(Stack_i) - 1)$ is the top element. The superscript of a variable denotes the execution, i.e., algorithm, it belongs to.

We will show by induction of the execution length that at each point the following invariants hold:

1. $length(Stack_i^E) = length(Stack_i^{E'})$,$i = 1, 2$, and $Stack_i^E(j) = Stack_i^{E'}(j))$,$i = 1, 2$, $0 \leq j \leq length(Stack_i^{E'} - 1)$;
2. $States^E = States^{E'}$;
3. $Transitions^E = Transitions^{E'}$;
4. $seed^E = seed^{E'}$.

Basic Step: Initially, the invariants hold because both E and E' begin by adding $h(\hat{s})$ to *States* and *Stack*. Now, we advance E and show how it can be mimicked by E', while preserving the invariants.

Induction Step: Suppose that in E a new state is generated in line 4 of the NDFS (Fig. 3) and the corresponding transition (s, a, s') is added to $Transitions^E$. By the induction hypothesis (IH) the invariants hold and consequently $s^E = s^{E'} = s$. According to Def. 3 there exists a state s_1 in T such that $s \xrightarrow{a} s_1$ and $h(s_1) = (s')^E$. Thus we take in line 4 of RNDFS $(s')^{E'} = s_1$. The relation $h((s')^{E'}) = (s')^E$ has the following consequences. First, exactly the same transition is saved in $Transition^{E'}$ as in $Transition^E$. Further, taking into account the IH, more particularly invariant 2, we conclude that if in line 6 (17) `dfs1` (`dfs2`) is called in E, then it is also called in E'. Moreover in both executions `dfs1` are called with the same state as argument, which further preserves the invariants through the statements in line 2 and 3 (12 and 13). Finally, in line 16, as by IH (invariant 4) *seed* has the same value in both executions, if a cycle is reported in E it will be also reported in E' (which is actually our main goal to prove).

Further, as $s^E = s^{E'}$ (IH, invariant 1), in line 8 if the acceptance state test is passed in E, then it is obviously true in E' too. This means that the lock-step execution of E and E' can be continued by assigning the same value (IH, invariant 1) to `seed` in both of them, and thus preserving invariant 4, in particular. Also `dfs2` is called in both executions and with the same argument, which will further preserve the invariants, as discussed above.

It is also not difficult to see that the `delete` statements in lines 9 and 19 will preserve the invariants, invariant 1 in particular.

Using completely symmetric argument one can show how the executions of RNDFS on T can be mimicked by the executions of NDFS on $h(T)$ which completes the other direction of the theorem. □

Theorem 2. *Given an LTS T and a selection function h, the nested depth first search algorithm with bisimulation preserving reduction (RNDFS) in Figure 4, when called on (the initial state) $h(\hat{s}) = \hat{s}$, reports a cycle if and only if there exists a reachable acceptance cycle in T.*

Proof. From Theorem 1, we have that when applied on $h(T)$, NDFS is guaranteed to report a cycle iff there exists a reachable acceptance cycle in $h(T)$. And by Lemma 3 there exists such a cycle in $h(T)$ iff there exists a reachable acceptance cycle in T, which proves the theorem. □

An important feature of the standard NDFS algorithm is that if an acceptance cycle is detected, the erroneous execution can be recreated by dumping the contents of the stack stacks (`Stack1` 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, in general 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 $h(T)$ which do not exist in T is obvious from Def. 3.)

A straightforward way to solve this problem is to define that given a subclass C of the bisimulation equivalence class the representative $h(s)$ is always the first state from C which is generated by the RNDFS algorithm.[2] (Notice that the correctness of RNDFS is independent of the choice of the selection function h.) This choice of a representative will imply that if a new state s' is generated such that $h(s')$ is not already in the `States`, then $h(s') = s'$. As a consequence `dfs1` and `dfs2` will be always called with the original state s' as an argument. In this way the stack always contains (a sequence of states that belong to) an execution which is also present in the original LTS T.

More formally, for our purpose we can always construct a selection function for RNDFS in the following way. Suppose we are given a selection function h on T. Then it is easy to see that h_1, defined as

$$h_1(s) = \begin{cases} s \ , & \text{if there is no state } s' \in \mathit{States} \text{ such that } h(s') = h(s); \\ s' \ , & \text{where } s' \in \mathit{States} \text{ and } h(s') = h(s), \ \text{otherwise} \end{cases}$$

is also a selection function on T and therefore it can be used in RNDFS. In other words, for each subclass of a bisimulation class (defined by the partitioning induced by h), the state of this subclass which is first generated by the RNDFS is used as a unique representative of the subclass. In this way RNDFS also generates (in the stacks) part of the original LTS T.

However, such a choice has a drawback regarding the efficiency. This is because in order to compute $h_1(s')$ one has to check if there is already a state

[2] a similar idea is also used in [8].

s belonging to the subclass of s' and which is already in `States`. As the computing of $h_1(s')$ is performed for each generated s', i.e., for each call of the `dfs` procedures, obviously it is critical for the overall performance of the algorithm.

One "naive" way to check if $h_1(s')$ is in `States` is to first compute $s'' = h(s')$. Then for each state s in `States` to check if $s'' = h(s)$. If this is true, then according to the definition of h_1 the representative of the subclass exists and $h_1(s') = s$. Obviously, it is very inefficient to recompute $h(s)$ for each s in `States` each time a check for a new s' is performed. One can avoid this overhead by saving $h(h_1(s'))$ in `States` instead of $h_1(s')$. Notice that if $h_1(s')$ has to be saved, then $h(h_1(s')) = h(s')$, This is because in such a case s' is the first state of its subclass which is generated by the search and therefore $h_1(s') = s'$. Intuitively, we use the selection function h to compute a unique, i.e. canonical, representative of each subclass. (Notice that these subclass representatives can be found efficiently in polynomial time.) If two states s_1, s_2 have the same canonical representative, i.e., if $h(s_1) = h(s_2)$, then they belong to the same subclass.

The above discussion brings us to the modified version of RNDFS, MRNDFS, given in Fig. 5.

If in RNDFS with selection function h_1 `dfs1` (`dfs2`) is called only if the representative $h_1(s')$ is not in `Stack` and consequently $h_1(s') = s'$. Therefore, in MRNDFS `dfs1` and `dfs2` are called with the newly generated states s' as arguments and consequently, the original states are saved on the stack instead of their representatives. However, in order to still benefit from the reduction only the (canonical forms of the) representatives $h(s')$ are saved in `States`, `Transitions` and the `seed`.

Next, we show that the algorithms MRNDFS and RNDFS are equivalent in the sense that they both report a cycle iff there is a reachable acceptance cycle in in the original LTS T. We first show that the parts of the original T which are generated by RNDFS and MRNDFS (i.e., the states kept in `Stack`) are the same, while they differ in the representatives that they save in `States`. However, there exists a representative of a given subclass in `States` of RNDFS iff a representative of the same subclass is also saved in `States` of MRNDFS. More precisely, we have the following:

Lemma 5. *Given an LTS T and selection functions h and h_1 as defined above, the nested depth first search algorithm with bisimulation preserving reduction (RNDFS) in Figure 4 when called on $\hat{s}$ and applied on T with selection function h_1 reports a cycle iff the modified nested depth first search algorithm with bisimulation preserving reduction (MRNDFS) in Figure 5, when called on the initial state $\hat{s}$ and applied on T with selection function h reports a cycle.*

Proof. The proof is very similar to the proof of Lemma 4. We construct an execution E' of the MRNDFS algorithm while tracing the execution E of RNDFS, when both algorithms are applied to T. Using the same denotations as in the proof of Lemma 4, we define analogous invariants, i.e., we show that at each point of the lock-step execution of the algorithms the following holds:

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

```
1  proc dfs1(s)
2     add s to Stack1
3     add {h(s),0} to States
4     for each transition (s,a,s')  do
5       add {{h(s),0},a,{h(s'),0}} to Transitions
6       if {h(s'),0} not in States then dfs1(s') fi
7     od
8     if accepting(s) then seed := {h(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 {h(s),1} to States
14    for each transition (s,a,s')  do
15      add {{h(s),1},a,{h(s'),1}} to Transitions
16      if {h(s'),1} == seed then report cycle
17      else if {h(s'),1} not in States then dfs2(s') fi
18    od
19    delete s from Stack2
20 end
```

Fig. 5. Modified nested depth first search algorithm with bisimulation preserving reduction (MRNDFS).

2. a state s is in $States^E$ iff there exists a unique state s' in $States^{E'}$ such that $h(s) = s'$.
3. a transition $s_1 \xrightarrow{a} s_2$ is in $Transitions^E$ iff there exists a unique transition $s'_1 \xrightarrow{a} s'_2$ in $Transitions^{E'}$, such that $h(s_1) = s'_1$ and $h(s_2) = s'_2$.
4. $h(seed^E) = seed^{E'}$.
5. In RNDFS (E) the variable s^E always contains a state which is its own representative, i.e., $h_1(s^E) = s^E$.

Baisc Step: Initially, the invariants hold. As both algorithms are called with $\hat{s}$ as argument, this implies that both executions E and E' begin by pushing $\hat{s}$ in *Stack*. As $States^E$ is empty, $h_1(\hat{s}) = \hat{s}$, which means that execution E saves $(\hat{s}, 0)$ in $States^E$, while execution E' saves $(h(\hat{s}), 0)$ to $States^{E'}$. In the beginning $Transitions$ are empty and *seed* is undefined in both executions, so the invariants hold vacuously.

Induction Step: We show that the lockstep execution of E and E' preserves the invariants. Let s_E and $s_{E'}$ be the states which are currently visited by E and E', respectively. The states are the top elements of the corresponding *Stack* variables, both in E and E'. Thus, it follows by the induction hypothesis (IH) (invariant 1) that $s_E = s_{E'} = s$. So, each transition which is taken in lines 4 and 14 of RNDFS can be mimicked by (the same lines in) MRNDFS. The obtained successor states s' are also the same. Depending on $h_1(s')$, we have to consider two cases.

1. If the representative $h_1(s')$ is not in $States^E$, then $h_1(s') = s'$. By IH (invariant 2) we also have that $h(s')$ is not in $States^{E'}$.
2. If the representative of s', $s_2 = h_1(s')$, is already in $States^E$, we conclude from the definition of h_1 that $h(s') = h(s_2)$. Thus by IH (invariant 2) $h(s')$ is already in $States^{E'}$.

In lines 5 and 15, RNDFS saves $s \xrightarrow{a} h_1(s')$ in $Transitions^E$, while MRNDFS saves $h(s) \xrightarrow{a} h(s')$ as a corresponding transition in $Transitions^{E'}$. Recall that s is the same in both E and E'. If $h_1(s')$ is not in $States^E$, $h_1(s') = s'$ (case 1 above) and invariant 3 is obviously preserved. Also, if the representative $s_2 = h_1(s')$ is already in $States^E$, it is easy to see from the above discussion (case 2) that $h(s') = h(s_2)$ is the destination state of the transition which is saved in E'. Therefore invariant 3 is preserved in this case too.

Using the same arguments about $h_1(s')$ one can show that `dfs1` in line 6 is called in MRNDFS, if it is called in RNDFS. Further, `dfs1` in RNDFS is always called with s' as parameter (case 1), i.e., exactly with the same parameter as in MRNDFS. As a consequence the same state is pushed in the *Stack* in line 2 in both algorithms. Thus, invariant 1 is preserved. As the state does not exist in *States* it will be added to the latter. Again from the definition of h_1 this means that $h(s)$ is added to $States^{E'}$ and s to $States^E$, which preserves invariant 2. In an analogous way we argue the preservation of the invariants by the call of `dfs2` in line 17.

In a straightforward way one can prove that the invariants are preserved also by the rest of the algorithm. Here we omit the further details. □

The following claim states the correctness of MRNDFS:

Theorem 3. *Given an LTS T, the modified nested depth first search algorithm with bisimulation preserving reduction (MRNDFS) in Figure 5, when called on $\hat{s}$, reports a cycle if and only if there exists a reachable acceptance cycle in T.*

Proof. By Lemma 5 MRNDFS reports a cycle iff RNDFS reports a cycle. By Theorem 2 RNDFS reports a cycle iff there exists a reachable acceptance cycle in T, which proves the claim. □

4.1 Experimental Support

A prototype implementation of the MRNDFS algorithm is used in SymmSpin [3], an extension of the model checker Spin [12] with symmetry reductions. We tried it on the case studies from [3] with encouraging results. Namely, the obtained reductions were usually of several orders of magnitude, in fact, very similar with the results for the same examples for safety properties, reported in [3]. Due to space constraints we give only the results for two examples. Table 1 contains the results for a typical bounded response property for the Data Base Manager example, while Table 2 gives the results for Peterson's mutual exclusion protocol for a property of the same class. The symmetry reduction in both examples was performed using a selection function h which corresponds to the "pc-sorted" heuristic from [3] and which uses multiple representatives.

Table 1. Results for the Data Base Manager example.

number of processes	6		7		8		9		10	
symmetry reduction	no	yes	no	yes	no	yes	no	yes	no	yes
number of states	2924	65	10215	113	35002	179	118109	307	362048	487
time [min:sec]	0.3	< 0.1	1.7	< 0.1	7.0	0.1	29.3	0.1	1:45.8	0.2

Table 2. Results for Peterson's mutual exclusion protocol.

number of processes	2		3		4	
symmetry reduction	no	yes	no	yes	no	yes
number of states	263	137	11318	2396	out of mem.	46804
time [min:sec]	< 0.1	< 0.1	1.0	0.4	–	9.0

5 Conclusion

In this paper we presented an algorithm for model checking properties defined as Büchi automata while employing bisimulation preserving reductions [10] with multiple representatives. The algorithm is based on the nested depth first search (NDFS) algorithm of [6]. As such the algorithm features all the advantages that the NDFS approach has over analogous algorithms based on the search for maximal strongly connected components (MSCC). We presented two versions of the algorithm. The first version, which was a straightforward extension to NDFS of the algorithm of [10] lacked the property to always reproduce a "realistic" counterexample execution. As this property is an important feature of model checking, we also presented a modification of the algorithm which was capable of showing a counterexample which always exists in the original state space. The theoretical results were supported by experiments performed with SymmSpin [3].

From practical point of view, probably the main direction for future work will be to integrate the algorithm presented in this paper with fairness. Unfortunately, the algorithms that can be found in the literature for symmetry reduction with fairness (e.g. [9,11]) are not compatible with the NDFS concept. This is because they are developed for state space search algorithms based on the MSCC approach. It can be shown though that the weak process fairness algorithm which is implemented in Spin (a version of Choueka's flag algorithm) is compatible with our algorithm in one direction – false positives can be generated, but if a fair acceptance cycle is reported by MRNDFS with fairness, then a fair acceptance cycle also exist in the original state space. An obvious challenge is to modify the algorithm such that it holds for the other direction too.

It should not be difficult to prove that the (M)RNDFS algorithm is compatible with partial order reduction (POR). Our optimism is based on [10] and [2] where it is shown that POR is compatible with symmetry with canonical and multiple representatives, respectively, as well as on [13] where NDFS was adapted to POR. As the two reduction techniques are orthogonal, their combination can be very effective, as it was confirmed experimentally [3].

Acknowledgments. The author is grateful to Dennis Dams for the numerous discussions over the issues treated in the paper and the valuable feedback, and to Gerard Holzmann for his remark that lead to the MRNDFS algorithm.

References

1. A.V. Aho, J.E. Hopcroft, J.D. Ulmann, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.
2. D. Bošnački, *Partial Order and Symmetry Reductions for Discrete Time*, to appear in Proc. of RT-TOOLS '02, Copenhagen, Denmark, 2002.
3. D. Bošnački, D. Dams, L. Holenderski, *Symmetric Spin*, 7th Int. SPIN Workshop on Model Checking of Software SPIN 2000, pp. 1–19, LNCS 1885, Springer, 2000.
4. E.M. Clarke, R. Enders, T. Filkorn, S. Jha, Exploiting Symmetry in Temporal Logic Model Checking, *Formal Methods in System Design*, Vol. 19, 77–104, 1996.
5. E.M. Clarke, Jr., O. Grumberg, D.A. Peled, *Model Checking*, The MIT Press, 2000.
6. C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, *Memory Efficient Algorithms for the Verification of Temporal Properties*, Formal Methods in System Design I, pp. 275-288, 1992.
7. E.A. Emerson, *Temporal and Modal Logic*, in J. van Leeuwen (ed.), Formal Models and Semantics, pp. 995–1072, Elsevier, 1990.
8. E.A. Emerson, A.P. Sistla, *Symmetry and model checking*, Proc. of CAV'93 (Computer Aided Verification), LNCS 697, pp. 463–478, Springer, 1993.
9. E.A. Emerson, A.P. Sistla, *Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic Approach*, ACM Transactions on Programming Languages and Systems, Vol. 19, 4, pp. 617–638, ACM Press, 1997.
10. E.A. Emerson, S. Jha, D. Peled, *Combining partial order and symmetry reductions*, in Ed Brinksma (ed.), Proc. of TACAS'97 (Tools and Algorithms for the Construction and Analysis of Systems), LNCS 1217, pp. 19–34, Springer, 1997.
11. V. Gyuris, A.P. Sistla, *On-the fly model checking under fairness that exploits symmetry*, in O. Grumberg (ed.), Proc. of CAV'97 (Computer Aided Verification), LNCS 1254, pp. 232–243, Springer, 1997.
12. G.J. Holzmann, *Design and Validation of Communication Protocols*, Prentice Hall, 1991. Also: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.
13. G. Holzmann, D. Peled, M. Yannakakis, *On Nested Depth First Search*, Proc. of the 2nd Spin Workshop, Rutgers University, New Jersey, USA, 1996.
14. R. Iosif, *Symmetry Reduction Criteria for Software Model Checking*, Model Checking Software, Proc. of SPIN 2002, LNCS 2318, pp. 22–41, Springer, 2002.
15. C.N. Ip, D.L. Dill, Better verification through symmetry. *Formal Methods in System Design*, Vol. 9, pp. 41–75, 1996.
16. W. Thomas, *Automata on Infinite Objects*, in J. van Leeuwen (ed.), Formal Models and Semantics, pp. 995–1072 Elsevier, 1990.
17. M. Vardi, P. Wolper, *An automata-theoretic approach to automatic program verification.* In Proc. of the 1st Symposium on Logic in Computer Science LICS '86, pp. 322–331, 1986.
18. P. Wolper, D. Leroy, *Reliable Hashing without Collision Detection*, Proc. of CAV'93 (Computer Aided Verification), LNCS 697, pp. 59–70, Springer, 1993.