



FI MU

**Faculty of Informatics
Masaryk University**

Distributed Explicit Fair Cycle Detection: Set Based Approach

by

**Ivana Černá
Radek Pelánek**

Distributed Explicit Fair Cycle Detection (Set Based Approach)

Ivana Černá and Radek Pelánek *

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic
{cerna, xpelanek}@fi.muni.cz

Abstract. The fair cycle detection problem is at the heart of both LTL and fair CTL model checking. This paper presents a new distributed scalable algorithm for explicit fair cycle detection. Our method combines the simplicity of the distribution of explicitly presented data structure and the features of symbolic algorithm allowing for an efficient parallelisation. If a fair cycle (i.e. counterexample) is detected, then the algorithm produces a cycle, which is in general shorter than that produced by depth-first search based algorithms. Experimental results confirm that our approach outperforms that based on a direct implementation of the best sequential algorithm.

1 Introduction

The fair cycle detection problem is at the heart of many problems, namely in deciding emptiness of ω -automata like generalised Büchi and Streett automata, and in model checking of specifications written in linear and branching temporal logics like LTL and fair CTL.

A generalised Büchi automaton [10] is provided together with several sets of accepting states. A run of such an automaton is accepting if it contains at least one state from every accepting set infinitely often. Accordingly, the language of the automaton is nonempty if and only if the graph corresponding to the automaton contains a reachable *fair cycle*, that is a cycle containing at least one state from every accepting set, or equivalently a reachable *fair strongly connected component*, that is a nontrivial strongly connected component (SCC) that intersects each accepting set. The acceptance condition for Streett automata [34] is more involved and consists of pairs of state sets. The language of the automaton is nonempty if and only if the automaton graph contains a cycle such that for every pair of sets whenever the cycle intersects the first set of the pair then it intersects also the second set. The nonemptiness check for Streett automata can thus be also based on identification of the fair SCCs of the automaton graph. Other types of automata for which the nonemptiness check is based on identification of fair cycles are listed in [15].

The LTL model checking problem and the LTL model checking with strong fairness (compassion) reduce to language emptiness checking of generalised Büchi automata and Streett automata respectively [36, 26]. Fair cycle detection is used to check the CTL formula $\mathbf{EG}f$ under the full (generalised) fairness constraints [15]. Hence, the

* Supported by GA ČR grant no. 201/00/1023

core procedure in many model checking algorithms is the fair cycle detection. These algorithms are in common use in explicit and symbolic LTL model checkers such as SPIN [22] and SMV [29] respectively, in fair-CTL model checkers such as SMV, VIS [7], and COSPAN [19].

Despite the developments in recent years, the main drawbacks of model checking tools are their high space requirements that still limit their applicability. Distributed model checking tackles with the space explosion problem by exploiting the amount of resources provided by parallel environment. Powerful parallel computers can be build of Networks Of Workstations (NOW). Thanks to various message passing interfaces (e.g., PVM, MPI) a NOW appears from the outside as a single parallel computer with a huge amount of memory.

Reports by several independent groups ([33, 28, 17, 4, 3]) have confirmed the usefulness of distributed algorithms for the state-space generation and reachability analysis. Methods for distributing LTL and CTL model checking have been presented in [1, 2, 8] and [6] respectively. However, until today not much effort has been taken to consider distributed algorithms for fair cycle detection. In our search for an effective distributed algorithm let us first discuss diverse sequential algorithms for fair cycle detection.

In *explicit algorithms* the states of a graph are represented individually. The decomposition of the graph into SCC can be solved in linear time by the Tarjan algorithm [35]. With the use of this decomposition it is easy to determine fair components and hence our problem has linear time complexity. Moreover, the nested depth-first search algorithm [23] (NESTEDDFS) optimises the memory requirements and is able to detect cycles *on-the-fly*. This makes NESTEDDFS the optimal sequential algorithm.

The explicit representation allows for a direct distribution of the state space. States of the graph are distributed over particular computers in NOW and are processed in parallel. When necessary, messages about individual states are passed to the neighbour computers. However, the depth-first search crucially depends on the order in which vertices are visited and the problem of depth-first search order is P-complete [31]. Therefore it is considered to be inherently sequential and we cannot hope for its good parallelisation (unless NC equals P).

Symbolic algorithms represent sets of states via their characteristic function, typically with binary decision diagrams (BDDs) [9, 13], and operate on entire sets rather than on individual states. This makes the depth-first approach inapplicable and symbolic algorithms typically rely on the breadth-first search (for surveys see [16, 30]). Unfortunately, the time complexity of symbolic algorithms is not linear; the algorithms contain a doubly-nested fixpoint operator, hence require time quadratic in the size of the graph in the worst case. The main advantage of symbolic algorithms over their explicit counterpart is the fact that BDDs provide a more compact representation of the state space capturing some of the regularity in the space and allow to verify systems with extremely large number of states, many orders of magnitude larger than could be handled by the explicit algorithms [11]. Nevertheless, there are applications where explicit model checkers outperform the others, for examples see [33, 24, 25, 14]

Thank to the fact that symbolic algorithms search the graph in a manner where the order in which vertices are visited is not crucial, these algorithms are directly parallelizable. On the other hand, the distribution of the BDD data structure is rather complicated. A parallel reachability BDD-based algorithm in [20] partitions the set of

states into slices owned by particular processes. However, the state space has to be dynamically repartitioned to achieve the memory balance and the method requires passing large BDDs between processes, both for sending non-owned states to their owners and for balancing. This causes a significant overhead.

Bearing all the reported arguments in mind we have tried to set down a parallel algorithm for fair cycle detection combining advantages of both explicit and symbolic approach. Our algorithm is in its nature explicit as the states are represented individually. The state space is well distributable and the parallel computation needs to communicate only information about individual states. The way how the algorithm computes resembles that of symbolic algorithms and thus allows for a good parallelisation of the computation alone.

Since our algorithm is based on symbolic ones, its worst-case complexity is $O(n \cdot h)$ where h is the height of the SCC quotient graph. Previous experiments ([16]) clearly show that this height is in practice very small and thus the algorithm is nearly linear. This observation has been confirmed also by our experiments.

The proposed algorithm is not on-the-fly and the whole state space has to be generated. For this reason the algorithm is meant not to replace but to complement the depth-first search based algorithms used in LTL model checking. The depth-first search based algorithms are of help before spacing out the available memory. On the other hand, our algorithm performs better in cases when the whole state space has to be searched. This distinction has been confirmed also by our initial performance evaluation using several protocols. Our algorithm outperforms that based on a direct implementation of the best sequential algorithm in a distributed environment especially in cases, when a fair cycle is not detected.

In model checking applications, the existence of a fair cycle indicates a failure of the property. In such a case, it is essential that the user is given a fair cycle as a *counterexample*, typically presented in the form of a finite stem followed by a cycle. The counterexample should be as short as possible, to facilitate debugging. Finding the shortest counterexample, however, is NP-complete [21]. The great advantage of our approach is that thanks to the breadth-first search character of the computation the computed fair cycle (counterexample) is very short in comparison with those computed by a depth-first search based algorithm.

Last but not least, we would like to emphasize that the algorithm is compatible with other state-space saving techniques used in LTL model checking. Namely, the algorithm can be applied together with static partial order reduction [27].

Plan of the work Section 2 reviews basic notions and explains the basics of symbolic fair cycle detection algorithms. In Section 3 a new sequential explicit fair cycle detection algorithm is presented together with the proof of its correctness and the analysis of its complexity. The distributed version of the algorithm is described in Section 4. Modifications of the algorithm allowing for a fair cycle detection for generalised Büchi and Streett automata and a simplification for weak ω -automata are presented in Section 5. Section 6 presents experimental results on real examples and compares the performance of our algorithm to a distributed implementation of the best sequential algorithm. Section 7 concludes.

2 Fair Cycle Detection Problem

A directed graph is a pair $G = (V, E)$, where V is a finite set of *states* and $E \subseteq V \times V$ is a set of *edges*. A *path* from $s_1 \in V$ to $s_k \in V$ is a sequence $(s_1, \dots, s_k) \in V^+$ such that $(s_i, s_{i+1}) \in E$ for $1 \leq i < k$. A *cycle* is a path from a state s to itself. We say that a state r (a cycle c) is *reachable* from a state s if there exists a path from s to r (to a state r on the cycle c). Moreover, every state is reachable from itself. Given a state set U , the graph $G(U) = (U, E \cap (U \times U))$ is the graph *induced* by U .

A *strongly connected component* (SCC) of G is a maximal set of states $C \subseteq V$ such that for each $u, v \in C$, the state v is reachable from u and vice versa. The *quotient graph* of G is a graph (W, H) , such that W is the set of the SCCs of G and $(C_1, C_2) \in H$ if and only if $C_1 \neq C_2$ and there exist $r \in C_1, s \in C_2$ such that $(r, s) \in E$. The *height* of the graph G is the length of the longest path in the quotient graph of G (note that the quotient graph is acyclic).

A strongly connected component C is a *trivial* component if $G(C)$ has no edges and *initial* if it is the source of the quotient graph. Let $F \subseteq V$ be a set of *fair states*. An SCC C is a *fair component* if it is nontrivial and $C \cap F \neq \emptyset$. A cycle is *fair* if it contains a fair state.

The *fair cycle detection problem* is to decide, for a given graph G with a distinguished initial state *init_state* and a set of fair states F , whether G contains a fair cycle reachable from the initial state. In the positive case a fair cycle should be provided.

Our goal is to bring in an algorithm for the fair cycle detection problem that is not based on a depth-first search and thus enables effective distribution. Here we take an inspiration in symbolic algorithms for cycle detection, namely in SCC hull algorithms. These algorithms compute the set of states that contains all fair components. Algorithms maintain the approximation of the set and successively remove unfair components until they reach a fixpoint. Different strategies of removal of unfair components lead to different algorithms. An overview, taxonomy, and comparison of symbolic algorithms can be found in independent reports by Fisler et al. [16] and Ravi et al. [30]. As the base for our algorithm we have chosen the *One Way Catch Them Young* algorithm [16]. The reasons for this choice are discussed at the beginning of Section 4.1.

Symbolic algorithms are conveniently described with the help of μ -calculus formulae. Our algorithm makes use of the following two functions:

$$Reachability(S) = \mu Z.(S \cup image(Z))$$

$$Elimination(S) = \nu Z.(S \cap image(Z))$$

The set $image(Z)$ contains all successors of states from Z in the graph G . The function $Reachability(S)$ computes the set of all states that are reachable from the set S . The function $Elimination(S)$ computes the set of all states q for which either q lies on a cycle in S or q is reachable from a cycle in S along a path that lies in S . The computation of $Elimination(S)$ is performed by successive removal of states that do not have predecessors in S . With the help of these functions the algorithm *One Way Catch Them Young* can be formulated as follows:

```

proc OWCTY( $G, F, init\_state$ )
   $S := Reachability(init\_state)$ ;
   $old := \emptyset$ ;
  while ( $S \neq old$ ) do
     $old := S$ ;
     $S := Reachability(S \cap F)$ ;
     $S := Elimination(S)$ ;
  od
  return ( $S \neq \emptyset$ );
end

```

The assignment $S := Reachability(S \cap F)$ removes from the set S all initial components of $G(S)$, which do not contain any fair state (in fact only SCCs reachable from a fair component are left in S). The assignment $S := Elimination(S)$ removes from the set S all initial trivial components (besides others). Thus each iteration of the **while** cycle (so called *external iteration*) removes initial unfair components of $G(S)$ until the fixpoint is reached.

The worst-case complexity of the algorithm is $O(n^2)$ steps¹ or more precisely $O(h \cdot n)$, where n is the number of states of the graph and h is the height of G . However, numerous experiments show that the number of external iterations tends to be very low and hence the number of steps is practically linear [16].

3 Sequential algorithm

In this section we present a new sequential explicit algorithm that computes a hull, that is, a set of states that contains all fair components in a way which resembles the set based algorithm *One Way Catch Them Young*. In the second part an algorithm enumerating a fair cycle is introduced. The correctness of both algorithms is proved, and their complexity is analysed. The distributed version of the algorithm is given in the next section.

3.1 Detection of a Fair Cycle

The explicit algorithm DETECT-CYCLE emulates the behaviour of the OWCTY algorithm. The set S is represented explicitly. For each state q the information whether q is in the set S is stored in boolean array inS . The emulation of the intersection operation and the $Reachability(S)$ function is straightforward (see the procedures RESET and REACHABILITY respectively). The emulation of $Elimination(S)$ is more involved: concurrently with the emulation of $Reachability(S)$ we count for each state q the number of its predecessors belonging to the set S (array p). On top of that we keep the list L of vertices, which have no predecessors in S , that is, those for which $p[q] = 0$. These vertices are eliminated from S in the procedure ELIMINATION. Data structures used by the algorithm and their initial settings are:

¹ The complexity of symbolic algorithms is usually measured in number of steps (*image computations*), since the real complexity depends on the conciseness of the BDD representation.

Sequential algorithm for fair cycle detection

```
1 proc DETECT-CYCLE( $G, F, init\_state$ )
2   put  $init\_state$  into  $queue$ ;
3    $inS[init\_state] := true$ ;
4   REACHABILITY;
5   while ( $Ssize \neq oldSsize \wedge Ssize > 0$ ) do
6     RESET;
7     REACHABILITY;
8     ELIMINATION;
9   od
10  return( $Ssize > 0$ );
11 end
```

```
1 proc RESET
2    $oldSsize := Ssize$ ;
3    $Ssize := 0$ ;
4   foreach  $q \in V$  do
5      $inS[q] := inS[q] \wedge q \in F$ ;
6      $p[q] := 0$ ;
7     if  $inS[q]$  then  $Ssize := Ssize + 1$ ;
8       put  $q$  in  $queue$ ;
9       put  $q$  in  $L$ ; fi
10  od
11 end
```

```
1 proc REACHABILITY
2   while  $queue \neq \emptyset$  do
3     remove  $q$  from  $queue$ ;
4     foreach  $(q, r) \in E$  do
5       if ( $\neg inS[r]$ ) then  $inS[q] := true$ ;
6          $Ssize := Ssize + 1$ ;
7         put  $r$  in  $queue$ ; fi
8       if  $p[r] = 0$  then remove  $r$  from  $L$ ; fi
9        $p[r] := p[r] + 1$ ;
10    od
11  od
12 end
```

```
1 proc ELIMINATION
2   while  $L \neq \emptyset$  do
3     remove  $q$  from  $L$ ;
4      $inS[q] := false$ ;
5      $Ssize := Ssize - 1$ ;
6     foreach  $(q, r) \in E$  do
7        $p[r] := p[r] - 1$ ;
8       if  $p[r] = 0$  then put  $r$  to  $L$  fi
9     od
10  od
11 end
```

- inS is a boolean array and is set to *false* for each state.
- p is an integer array and is set to 0 for each state.
- L is a list of states, initially empty. L is implemented as doubly linked list, hence all necessary operations (insertion, deletion, and removal of a state) can be performed in constant time.
- $Ssize$ and $oldSsize$ are number variables initially set to 1 and 0 respectively.
- $queue$ is an initially empty queue.

Correctness

In what follows we denote S the set of states q such that $inS[q] = true$ and particularly S_i^j the set of states q such that $inS[q] = true$ just before the i -th execution of the line l in DETECT-CYCLE ($l = 6, 7, 8$). Arguments are presented in the manner, which allows their transfer to the distributed algorithm.

Lemma 1. *At the end of REACHABILITY the set S is the set of states that are reachable from states which were in the queue at the beginning of the procedure.*

Proof: Whenever the procedure REACHABILITY is called, the *queue* contains exactly all the states for which $inS[q] = true$. REACHABILITY performs the standard breadth-first search and empties the *queue*.

Lemma 2. *The invariant $q \in L \Rightarrow (q \in S \wedge p[q] = 0)$ holds true during the whole computation of DETECT-CYCLE.*

Proof: Only states r with $p[r] = 0$ are put in L in ELIMINATION and RESET. To show $L \subseteq S$ we notice that $queue = S = L$ at the end of RESET and S is the set of states reachable from L at the end of REACHABILITY (Lemma 1). Only states reachable from L are put to L in ELIMINATION but those states are already in S .

Lemma 3. *Immediately after executing RESET, REACHABILITY and ELIMINATION respectively, the value of $Ssize$ is the size of the set S .*

Proof: Whenever a new state q is added to S in REACHABILITY the variable $Ssize$ is changed accordingly. In RESET only those states which are kept in S are counted. Correctness for ELIMINATION follows from the inclusion $L \subseteq S$ (Lemma 2).

Lemma 4. *Immediately after executing REACHABILITY and ELIMINATION respectively, the value of $p[q]$ is the number of those direct predecessors of the state q , which belong to S .*

Proof: Whenever a state r is attained in REACHABILITY the value $p[r]$ is updated. Whenever a state is deleted from S in ELIMINATION all its direct successors are visited and their respective values are updated.

On the other side, the value of $p[r]$ is changed only when some of its direct predecessors is added to/removed from *queue* (Lemma 2).

Lemma 5. *During one execution of the procedure REACHABILITY each state is inserted to and deleted from the queue at most once. During one execution of the procedure ELIMINATION each state is removed from L at most once.*

Proof: No state is removed from S in REACHABILITY. Moreover, $q \in queue \Rightarrow q \in S$ and the state q is added to $queue$ only if $q \notin S$.

The assertion for ELIMINATION follows from Lemma 2 and the fact that states are removed simultaneously both from L and S .

Lemma 6.

- $S_7^i = S_6^i \cap F$.
- S_8^i is the set of states reachable from the set S_7^i .
- S_6^{i+1} is the set of all states q for which either q lies on a cycle in $G(S_8^i)$ or q is reachable in $G(S_8^i)$ from a cycle in $G(S_8^i)$.

Proof: The first equality follows directly from the code of the procedure RESET.

The second fact is a direct consequence of Lemma 1, because the content of $queue$ at the beginning of REACHABILITY is S_7^i .

By Lemma 4, value $p[q]$ is the number of direct predecessors of q in $G(S_8^i)$ and only states with none predecessors are removed from S in ELIMINATION. Therefore all states with the required property are in S_6^{i+1} . On the other hand, all predecessors of the state q not satisfying the condition will eventually be removed (this can be formalised by induction on the length of the longest chain of predecessor of a given state), hence eventually $p[q]$ is set to 0, the state q is put in L and removed from the set S afterwards.

Lemma 7. $S_6^{i+1} \subseteq S_6^i$

Proof:

The assertion can be proved by induction on i . For the base case $i = 1$ we argue that S_6^1 is the set of all states reachable from $init_state$ and all the states put in S in REACHABILITY (line 7) are reachable from $init_state$ and thus $S_6^2 \subseteq S_6^1$.

For the general case we suppose $S_6^{i+1} \subseteq S_6^i$. Then we can reason with the use of Lemma 6 as follows: $(S_6^{i+1} \subseteq S_6^i) \Rightarrow (S_6^{i+1} \cap F \subseteq S_6^i \cap F) \Rightarrow (S_7^{i+1} \subseteq S_7^i) \Rightarrow$ each state reachable from S_7^{i+1} is reachable from S_7^i as well $\Rightarrow (S_8^{i+1} \subseteq S_8^i) \Rightarrow$ each state that lies on (or is reachable from) a cycle in S_8^{i+1} lies on (or is reachable from) a cycle in S_8^i as well $\Rightarrow (S_6^{i+2} \subseteq S_6^{i+1})$.

Theorem 1 (Termination). *The DETECT-CYCLE algorithm terminates.*

Proof: The termination of REACHABILITY and ELIMINATION follows from Lemma 5. The termination of RESET is straightforward.

By Lemma 7, $S_6^{i+1} \subseteq S_6^i$ which together with Lemma 3 ensures that the condition on line 5 eventually becomes false and DETECT-CYCLE terminates as well.

Theorem 2 (Completeness). *If G contains a fair cycle reachable from the $init_state$ then DETECT-CYCLE returns true.*

Proof: Let C be a fair cycle in G and q a fair state that lies on the cycle C . We prove by induction on i that $q \in S_6^i$. For the base case $i = 1$ we argue that S_6^1 is the set of states reachable from $init_state$ and thus $q \in S_6^1$.

Now let $q \in S_6^i$. By Lemma 6, $q \in S_7^i$. The state q as well as all the states reachable from q belong to S_8^i . Namely, the whole cycle C belongs to S_8^i and by Lemma 6 cycle C belongs also to the set S_6^{i+1} .

Hence after executing the **while** loop the state q belongs to S , therefore $Ssize > 0$ (Lemma 3) and DETECT-CYCLE returns true.

Theorem 3 (Soundness). *If DETECT-CYCLE returns true, then G contains a fair cycle reachable from the $init_state$.*

Proof: Let us suppose that DETECT-CYCLE terminates after k iterations of the **while** cycle. Since the algorithm returns *true*, $Ssize > 0$, $S_6^k = S_6^{k-1}$ and S_6^k is nonempty (Lemma 3 and 7).

Let us consider the decomposition of S_6^k into SCCs. Let C be the initial component. We demonstrate that C is fair (that is, C contains a fair state and is nontrivial). This implies the assertion of the theorem.

Let us suppose that $C \cap F = \emptyset$. The set S_6^{k-1} contains only states reachable from $S_6^{k-1} \cap F = S_6^k \cap F$ and because C is initial no state from C is in S_6^{k-1} . Consequently C is not contained in S_6^k (Lemma 6), a contradiction.

If the component C were trivial, it would be removed from the set $S_6^k = S_6^{k-1}$ by the procedure ELIMINATION due to Lemma 6.

Theorem 4 (Complexity). *The worst-case complexity of the algorithm DETECT-CYCLE is $O(h \cdot (n + m))$, where n is the number of states in G , m is the number of edges in G , and h is the height of G .*

Proof: The complexity of the procedure RESET is $O(n)$. Both REACHABILITY and ELIMINATION procedures have complexity $O(m)$ (Lemma 5). Thus it remains to show that the **while** loop in DETECT-CYCLE can iterate at most h times.

For a graph H , let us denote by h_u the length of the longest path in the quotient graph of H starting in an initial unfair component (the unfair height of H). By induction on i we prove that the unfair height of $G(S_i^6)$ is at most $h - i + 1$. The assertion clearly holds for $i = 1$ as $h_u \leq h$. For the induction step we note that by Lemma 6 in the i -th iteration of the **while** cycle all initial unfair components of S_i^6 are removed from S_i^6 . This claim together with the observations that all SCCs of S_{i+1}^6 are also SCCs in S_i^6 and the quotient graph of S_{i+1}^6 is a subgraph of the quotient graph of S_i^6 guarantee that the **while** loop in DETECT-CYCLE iterates at most h times.

3.2 Extraction of a Fair Cycle

In model checking applications a fair cycle corresponds to a counterexample (a trace of a verified system which does not satisfy a given specification). The knowledge of a counterexample helps developers to tune the system. Accordingly, the shortest counterexamples are searched for.

In this section we present an algorithm, which complements DETECT-CYCLE and for graphs with fair cycles returns a particular fair cycle. The algorithm for extraction of a fair cycle makes use of values stored in the boolean array inS computed by DETECT-CYCLE. The set S (represented via inS) initially contains all fair cycles.

The procedure EXTRACT-CYCLE searches the graph G from the initial state for a fair state s from the set S . A nested search is initialised from s and an existence of a cycle from s to s is checked. In the nested search only the graph $G(S)$ induced by S is searched. Moreover, every state, which has been completely searched by a nested search without discovering a cycle can be safely removed from S . This ensures that each state is visited in nested searches only once and the algorithm has linear complexity.

In both searches the graph is traversed in a breadth-first manner. Nevertheless, the order in which states are visited is not important and this allows for an effective distribution of the computation. The discovered cycle is output with the help of *parent* values.

The great advantage of our approach is that due to the fact that the graph is searched in a breadth-first fashion the counterexamples tend to be much shorter than those generated by depth-first based algorithms (see Section 6).

Sequential algorithm for the extraction of a fair cycle

```
proc EXTRACT-CYCLE( $G, F, init\_state, inS$ )
  put  $init\_state$  into  $queue$ ;
  while cycle not found do
    remove  $s$  from  $queue$ ;
    if  $inS[s] \wedge s \in F$  then NESTEDBFS( $s$ ); fi
    foreach  $(s, r) \in E$  do
      if  $parent[r] = nil$  then  $parent[r] := s$ ;
      put  $r$  in  $queue$ ; fi
    od
  od
  while  $s \neq init\_state$  do output  $s$ ;  $s := parent[s]$ ; od
end
```

```
proc NESTEDBFS( $s$ )
  put  $s$  into  $queue2$ ;
  while cycle not found and  $queue2$  not empty do
    remove  $q$  from  $queue2$ ;
    foreach  $(q, r) \in E$  do
      if  $inS[r] \wedge parent2[r] = nil$  then  $parent2[r] = q$ ;
      put  $r$  in  $queue2$  fi
      if  $r = s$  then cycle found;
       $r := parent2[r]$ ;
      while  $r \neq s$  do output  $r$ ;  $r := parent2[r]$ ; od
    fi
  od
   $inS[q] := false$ ;
od
end
```

Lemma 8 (Soundness). *The sequence of states output by EXTRACT-CYCLE forms (in the reverse order) a cycle containing a fair state followed by a path from the fair state to the initial state.*

Proof: Each state s visited in the **while** cycle of EXTRACT-CYCLE is reachable from the *init_state* and similarly each state r visited in NESTEDBFS(s) is reachable from s . Since NESTEDBFS is initialised only from fair states, the lemma follows.

Lemma 9 (Completeness). *The algorithm EXTRACT-CYCLE finds a fair cycle.*

Proof: Let C be an initial component of the quotient graph of $G(S)$, where S is the set computed by DETECT-CYCLE($G, F, init_state$). In NESTEDBFS only the induced

graph $G(S)$ is searched and thus no state from C can be reached (and removed from S) by NESTEDBFS initialised in a state outside C . By the proof of Theorem 3, the component C is also fair. For that reason it must be the case that either a fair cycle is found somewhere outside C or EXTRACT-CYCLE reaches a fair state s in C and consequently NESTEDBFS(s) discovers a cycle from s to s .

Lemma 10 (Complexity). *The complexity of EXTRACT-CYCLE is $O(n + m)$.*

Proof: The EXTRACT-CYCLE procedure visits each state only once. NESTEDBFS visits only states in S and once a state is completely searched by NESTEDBFS it is removed from S . Hence, NESTEDBFS visits each state at most once too.

4 Distributed Algorithm

Similar to other works devoted to the distributed model checking [6, 3, 8, 33, 4] we assume the MIMD architecture of a network of workstations, which communicate via message passing (no global information is directly accessible). All workstations execute the same program. One workstation is distinguished as a *Manager* and is responsible for the initialisation of the computation, detection of the termination, and output of results.

The set of states of the graph to be searched for fair cycles is partitioned into disjoint subsets. The partition is determined by the function *Owner*, which assigns every state q to a workstation i . Each workstation is responsible for the graph induced by the owned subset of states. The way how states are partitioned among workstations is very important as it has a direct impact on the communication complexity and thus on the runtime of the algorithm. We do not discuss it here because it is itself quite a difficult problem, which moreover depends on a particular application.

4.1 Detection of a Fair Cycle

The procedures RESET, REACHABILITY, and ELIMINATION can be easily transformed into distributed ones. Each workstation performs the computation on its part of the graph. Whenever a state s belonging to a different workstation is reached, the workstation sends an appropriate message to the *Owner*(s). All workstations periodically read incoming messages and perform required commands (*Serve messages*).

Computations on particular workstations can be performed in parallel. However, some synchronisation is unavoidable. All workstations perform the same procedure (RESET, REACHABILITY, ELIMINATION, or COUNT-SIZE). As soon as a workstation completes the procedure it sends a message to the *Manager* and becomes idle. When all workstations are idle and there are no pending messages the *Manager* synchronises all workstations and the computation continues.

The need of synchronisation after each procedure is the reason why we have chosen the *One Way Catch Them Young* algorithm as a base for our explicit algorithm. The analysis and experiments by Fisler et al. [16] indicates that this algorithm performs less external iterations than for example the well-known Emerson-Lei algorithm². The number of external iteration determines the number of necessary synchronisations.

² We note that some other algorithms studied by [16] perform even less external iterations. These algorithms make use of the *preimage* computation (i.e. computation of predecessors), which is usually not available in the explicit model checking.


```

proc COUNT-SIZE
  if Manager
    then sum up local_Ssize from all workstations;
      if global_Ssize = old_global_Ssize then send(all, stop);
      else send(all, continue); fi
    else send(Manager, local_Ssize);
      Wait_for_message;
    fi
  end

```

```

proc SYNCHRONIZATION
  if Manager
    then if all processes are idle and there are no pending messages
      then send(all, finished)
      else Wait_for_message;
    fi
    else send(Manager, I am idle)
      Wait_for_message;
      if message  $\neq$  finished then send(Manager, I am not idle) fi
    fi
  end

```

```

proc VISIT-STATE(r)
  if  $\neg$ inS[r] then inS[r] := true;
    local_Ssize := local_Ssize + 1;
    put r in queue; fi
  if p[r] = 0 then remove r from L; fi
  p[r] := p[r] + 1;
end

```

```

proc ELIMINATE-STATE(r)
  p[r] := p[r] - 1;
  if p[r] = 0 then put r to L; fi
end

```

The correctness proof is an analog to that for the sequential algorithm. In fact, under a proper modification all lemmas and theorems from Section 3 hold for the distributed algorithm as well.

The number of iterations of the **while** cycle in DETECT-CYCLE is bounded above by the height of the quotient of G . The complexity of all procedures is linear with respect to the size of the owned part of the graph.

4.2 Extraction of a Fair Cycle

The distributed counterpart of the procedure EXTRACT-CYCLE comes by in a similar way as for DETECT-CYCLE. The basic traversal is executed in parallel. Whenever a workstation finds a suitable candidate s for the nested traversal (that is, $s \in S \cap F$) it sends it to the *Manager*. The *Manager* puts the incoming candidates into a queue

and successively starts NESTEDBFS from them. The important point is that only one NESTEDBFS can be performed at a time.

The termination detection of the NESTEDBFS in a case that it failed to find a cycle is done in the same way as the detection of the end of procedures in distributed DETECT-CYCLE.

Distributed algorithm for the extraction of a fair cycle

proc EXTRACT-CYCLE

```

if init_state is local then put init_state into queue; fi
while not (finished or cycle_found) do
  Serve_messages;
  if queue  $\neq \emptyset$ 
    then remove s from queue;
    if  $inS[s] \wedge s \in F$  then send(Manager, NESTEDBFS candidate= s); fi
    foreach  $(s, r) \in E$  do
      if r is local then VISIT1(r, s);
      else send(Owner(r), VISIT1(r, s)); fi
    od
  fi
od
end

```

proc NESTEDBFS(*s*)

```

  put s into queue;
  while not (finished or cycle_found) do
    Serve_messages;
    if queue  $\neq \emptyset$ 
      then remove q from queue;
      foreach  $(q, r) \in E$  do
        if r is local then VISIT2(r, q, s);
        else send(Owner(r), VISIT2(r, q, s)); fi
      od
      inS[q] := false;
      else SYNCHRONIZATION;
    fi
  od
end

```

proc VISIT1(*r*, *s*)

```

  if parent[r] = nil then parent[r] := s;
  put r in queue; fi
end

```

proc VISIT2(*r*, *q*, *s*)

```

  if  $inS[r] \wedge parent2[r] = nil$  then parent2[r] = q;
  put r in queue2 fi
  if r = s then send(all, cycle_found);
  PRINT-CYCLE(r, s) fi
end

```

```

proc PRINT-CYCLE(at, start)
  flag := cycle;
  while not finished do
    if at is local
      then output at;
      if at = start then send(all, (flag := way)); fi
      if at = init_state then send(all, finished); fi
      if flag = cycle then at := parent2[at]; else at := parent[at]; fi
      if at is not local then send(Owner(at), continue from at) fi
    else Wait_for_message;
  fi
od
end

```

The correctness of the distributed EXTRACT-CYCLE algorithm again follows from that for the sequential one.

5 Modifications

In LTL model checking one often encounters not only Büchi automata for which the non-emptiness problem directly corresponds to a detection of fair cycles, but also their variants called *weak* and *generalised* Büchi automata and *Streett* automata. For these automata the non-emptiness problem corresponds to a slightly different version of the fair cycle detection problem. The advantage of the DETECT-CYCLE algorithm is that it can be easily modified in order to solve these problems.

In this section we provide pseudocodes of set based algorithms for the modified problems. The necessary modifications in both sequential and distributed explicit algorithms straightforwardly reflect changes of the set based algorithm and we do not state them.

5.1 Weak Graphs

We say that a graph G with a set F of fair states is *weak* if and only if each component C in SCC decomposition of G is either fully contained in F ($C \subseteq F$) or is disjoint with F ($C \cap F = \emptyset$).

Our study of hierarchy of temporal properties [12] suggests that in many cases the resulting graph is weak. Thus it is useful to develop specialised algorithms for these graphs. Actually, Bloem, Ravi, and Somenzi [5] have already performed experiments with specialised symbolic algorithms and state-of-the-art algorithms for generation of automaton for an LTL formula [32] include heuristics generating automaton as “weak” as possible.

From the definition of weak graphs it follows that the set F is a union of some SCCs. Thus a fair component exists if and only if some *nontrivial* component is contained in F . These observations lead to the following algorithm:

```

proc WEAK-DETECT-CYCLE( $G$ ,  $F$ , init_state)
   $S$  := Reachability(init_state);
   $S$  := Elimination( $S \cap F$ );
  return ( $S \neq \emptyset$ );
end

```

The algorithm WEAK-DETECT-CYCLE has several advantages. At first, its complexity is $O(n+m)$ which is asymptotically better than the complexity of DETECT-CYCLE and is the same as the complexity of the NESTEDDFS algorithm. At second, in the distributed environment, the specialised algorithm needs to synchronise only two times. And finally, this algorithm is easier to implement and provide better possibilities for heuristics and optimisations (especially in the distributed environment) than the depth-search based ones.

Thus one can use the specialised algorithm profitably whenever it is possible. The natural question is how expensive is to find out whether a graph is weak. In model checking applications the graph to be searched for fair cycles is a product of a system description (that is a graph without fair states) and a rather small graph expressing a desired property of the system. The weakness of the graph is determined by the property graph and hence it suffices to put the small graph to the weakness test.

5.2 Generalised Fair Condition

Generalised fair condition \mathcal{F} is a set $\{F_i\}$ of fair sets. A cycle is *fair in respect to a generalised fair condition* $\{F_i\}$ if and only if for each fair set F_i there exists a state q on the cycle such that $q \in F_i$.

In model checking applications, algorithms translating an LTL formula into an automaton usually end up with generalised fair conditions [18]. One can transform (and model checker tools usually do so) the generalised condition into the ordinary one through a “counter construction”. But the transformation increases the number of states, which is highly undesirable. Therefore it is more favourable to test directly the generalised condition.

The modification of the DETECT-CYCLE algorithm for generalised condition is rather simple. It suffices to guarantee that states in S are reachable from all fair sets.

```

proc GENERALIZED-DETECT-CYCLE( $G, \mathcal{F}, init\_state$ )
   $S := Reachability(init\_state);$ 
   $old := \emptyset;$ 
  while ( $S \neq old$ ) do
     $old := S;$ 
    foreach  $F_i \in \mathcal{F}$  do
       $S := Reachability(S \cap F_i);$ 
    od
     $S := Elimination(S);$ 
  od
  return ( $S \neq \emptyset$ );
end

```

5.3 Streett Fair Condition

Streett fair condition \mathcal{F} is a set of tuples $\{(P_i, Q_i)\}$. A cycle C is *fair in respect to a Streett fair condition* if and only if for each tuple (P_i, Q_i) it holds $C \cap P_i \neq \emptyset \Rightarrow C \cap Q_i \neq \emptyset$.

Streett fair condition is used to express strong fairness (compassion), that is, intuitively “if there is an infinite number of requests then there is an infinite number of responses”. Strong fairness can be expressed in LTL and thus it is possible to use the algorithm for (generalised) Büchi fair condition in order to check properties of system

with strong fairness requirements. However, this approach leads to the blowup of the size of formula automaton and thus it is more efficient to check the strong fairness directly (see [26]).

The set based algorithm for the Street fair condition can be formulated as follows:

```

proc STREET-DETECT-CYCLE( $G, \mathcal{F}, init\_state$ )
   $S := Reachability(init\_state)$ ;
   $old := \emptyset$ ;
  while ( $S \neq old$ ) do
     $old := S$ ;
    foreach ( $(P_i, Q_i) \in \mathcal{F}$ ) do
       $S := (S - P_i) \cup Reachability(S \cap Q_i)$ ;
    od
     $S := Elimination(S)$ ;
  od
  return ( $S \neq \emptyset$ );
end

```

For the proof of correctness see [26]. Corresponding modification of the explicit algorithm is more technically involved though rather straightforward.

The important fact is that other algorithms like NESTEDDFS or algorithm presented in [8] cannot cope with generalised and Streett condition in such a simple way (in fact the distributed algorithm from [8] cannot be directly modified to cope with generalised and Streett fair cycles).

6 Experiments

We performed series of experiments in order to test the practical usefulness of the proposed algorithm. In this section we mention representative results and discuss conclusions we have drawn from the experiments.

The implementation has been done in C++ and the experiments have been performed on a cluster of twelve 700 MHz Pentium PC Linux workstations with 384 Mbytes of RAM each interconnected with a fast 100Mbps Ethernet and using Message Passing Interface (MPI) library. Reported runtimes are averaged over several executions.

6.1 Examples

Graphs for experiments were generated from a protocol and an LTL formula in advance and programs have been provided with an explicit representation of a graph. This approach simplifies the implementation. However, as discussed later it has an unpleasant impact on the scalability of the distributed algorithm.

For graphs generation a simple model-checking tool has been used allowing us to generate graphs with approximately one million states. The algorithm was tested on several classical model checking examples:

- Absence of a starvation for a simple mutual exclusion protocol and for the Peterson protocol (Mutex, Peterson).

- Safety property for the alternation bit protocol (ABP).
- Reply properties (with fairness) for a model of an elevator (Elevator1, Elevator2).
- Safety and liveness properties for a token ring (Ring1, Ring2, Ring3, Ring4).
- Liveness property for the dining philosophers problem (Philosophers).

6.2 General Observations

At first, we have compared the sequential version of our algorithm with the sequentially optimal NESTEDDFS algorithm. We remind that from the theoretical point of view our algorithm is asymptotically worse. Table 1 summarises experiments with graphs without fair cycles and Table 2 covers experiments with graphs with fair cycles.

System Size	Algorithm	Time (s)	External Iterations
Peterson 376	NESTEDDFS	0.02	
	DETECT-CYCLE	0.06	18
ABP 7 286	NESTEDDFS	0.22	
	DETECT-CYCLE	0.41	1
Ring1 172 032	NESTEDDFS	17.13	
	DETECT-CYCLE	7.61	1
Elevator2 368 925	NESTEDDFS	35.10	
	DETECT-CYCLE	55.76	30
Philosophers 608 185	NESTEDDFS	72.68	
	DETECT-CYCLE	52.04	1

Table 1. Sequential experiments for graphs without fair cycles.

The following conclusions can be drawn from the experiments:

- The number of external iterations of DETECT-CYCLE is very small (less than 40) even for large graphs. This observation is supported by experiments in [16] with the symbolic implementation of the set-based algorithm. They obtained similar results for hardware circuits problems.
- The complexity of DETECT-CYCLE is in practice nearly linear.
- The runtime of our algorithm is comparable to NESTEDDFS for correct specifications (graphs without fair cycles).
- In the case of an erroneous specification (graphs with fair cycles) the NESTEDDFS algorithm is significantly faster because it is able to detect cycles “on-the-fly” without traversing the whole graph.
- On the other hand, the counterexamples generated by DETECT-CYCLE are significantly shorter because of the breadth-first nature of the algorithm. This is practically very important feature as counterexamples consisting of several thousands of states (as those generated by NESTEDDFS) are quite useless.

- The last observation compares the runtime of the first phase (cycle detection) to the second phase (cycle extraction) of our algorithm. Evidently the time needed for the second phase is significantly shorter than that for the first phase. Thus potential optimisations, heuristics, etc. of the algorithms should be directed at the first phase.

System Size	Algorithm	Time (s)	Extract time (s)	External Iterations	Fair cycle Prefix	Fair cycle Loop
Mutex 232	NESTEDDFS	0.01			76	3
	DETECT-CYCLE	0.02	0.01	2	2	2
Ring3 389 542	NESTEDDFS	2.70			14420	3
	DETECT-CYCLE	29.07	1.17	2	28	23
Elevator1 683 548	NESTEDDFS	7.28			304	76
	DETECT-CYCLE	99.43	1.80	8	20	22
Ring2 1 027 394	NESTEDDFS	12.82			2754	363
	DETECT-CYCLE	305.51	11.31	40	52	14

Table 2. Sequential experiments for graphs with a fair cycle. The column Time gives the overall time, Extract time is the time needed for the extraction of the cycle.

6.3 Distributed Tests

We note that experiments concerning the distributed version are only preliminary since the current implementation is straightforward and is far from being optimal. For example, it suffers from problems with load-balancing. The only optimisation that we have used is the reduction of communication by packing several messages into one.

We have compared our algorithm to the distributed version of NESTEDDFS where only one processor, namely the one owning the actual state in the depth-first search, is executing the search at a time. The network is in fact running the sequential algorithm with extended memory. The runtime of NESTEDDFS increases with the number of workstations thanks to the additional communication. On the other hand, our algorithm can take advantage of more workstations since it exploits parallelism. Hence in the distributed environment our algorithm convincingly outperforms NESTEDDFS.

The current implementation of DETECT-CYCLE algorithm is not optimised and does not scale ideally. We identify two main reasons. The first one is the straightforwardness of our implementation. The second, more involved reason, is based on fact that in our experiments we use pre-generated graphs, which however are not too large in comparison to the memory capacity of the NOW. Consequently the local computations are very fast and the slow communication has high impact on the overall runtime. We infer, in a similar way as [6], that if the algorithm computed the graph on-the-fly from the specification language then the communication and synchronisation would have smaller impact on the runtime and the algorithm would achieve better speedup. To support this explanation we have measured besides the real time taken

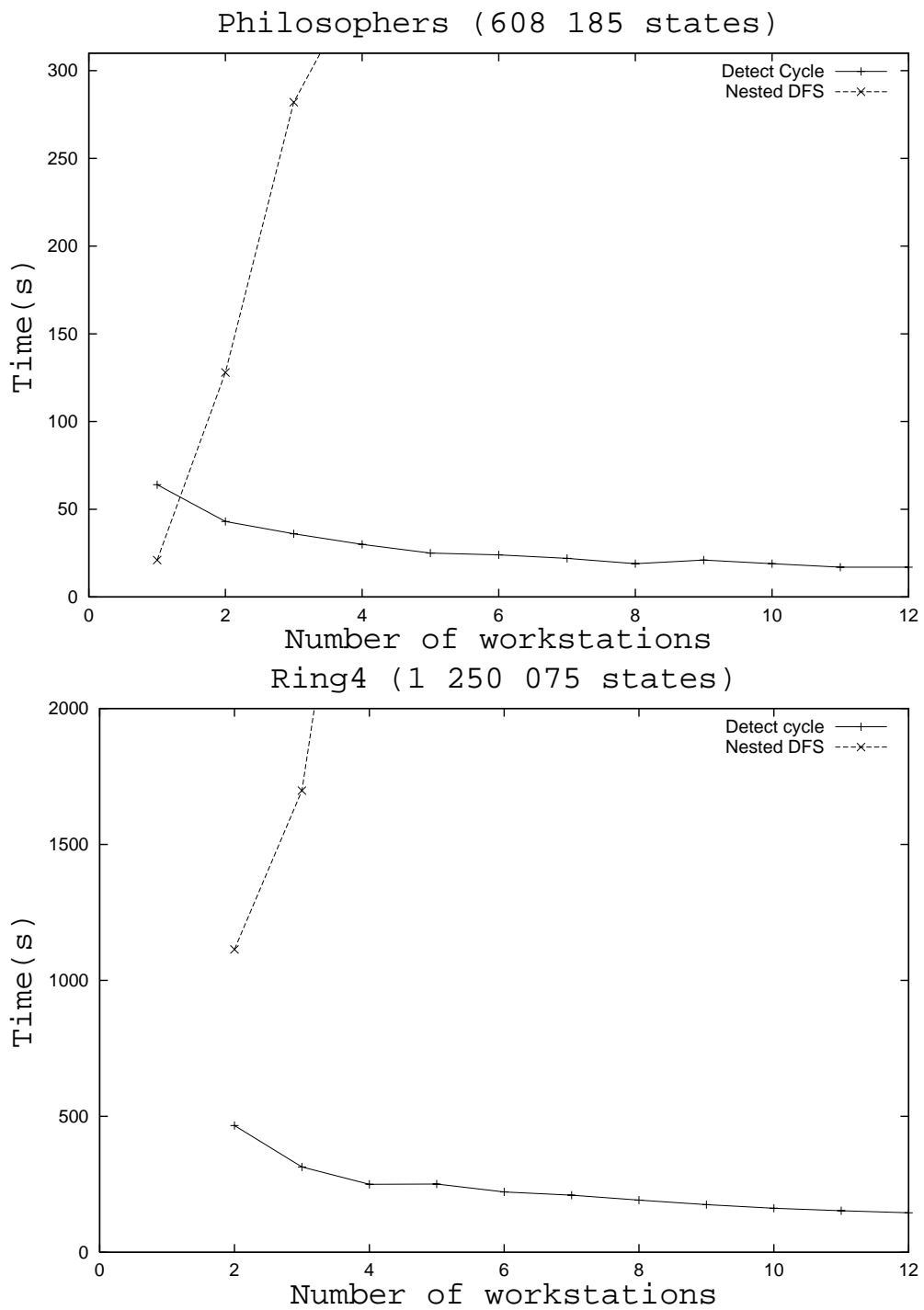


Fig. 1. Comparison of distributed NESTEDDFS and DETECT-CYCLE. The system Ring4 cannot be handled by one computer.

by the computation also the CPU time consumed by particular workstations. Fig. 2 resumes the results. The numbers indicate that the time taken by a local computation (CPU time) really scales well.

We have also implemented the distributed WEAK-DETECT-CYCLE algorithm and performed a comparison of the general and the specialised algorithm on weak graphs. Fig. 3 indicates that the use of specialised algorithm can yield a considerable improvement.

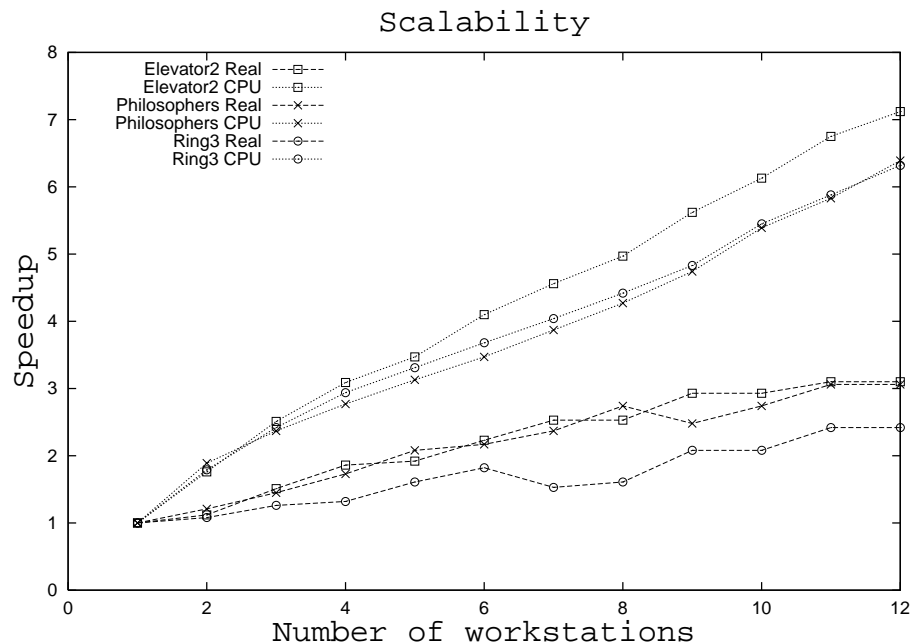


Fig. 2. Dependency of the runtime on the number of workstations. Figure shows the difference between real time taken by the program and the average CPU time used by a workstation.

7 Conclusions & Future Work

In this paper, we presented a new *distributed* algorithm for fair cycle detection problem. The demand for such an algorithm becomes visible especially referring to automata-based LTL model checking. This verification method suffers from the state explosion. Distributed model checking allows to cope with the state explosion by reason of allocation of the state space to several workstations in a network.

Our distributed algorithm comes out from a set-based algorithm, which searches the state space in a breadth-first search manner, which makes a distribution possible. On the other hand, the state space is represented explicitly and thus can be partitioned very naturally. The algorithm is compatible with other state space saving methods, namely with static partial order reduction. It aims not to replace but to complement the classical nested depth-first search algorithm used in explicit LTL model checkers as it demonstrates its efficiency especially in cases when the searched space does not contain any fair cycle.

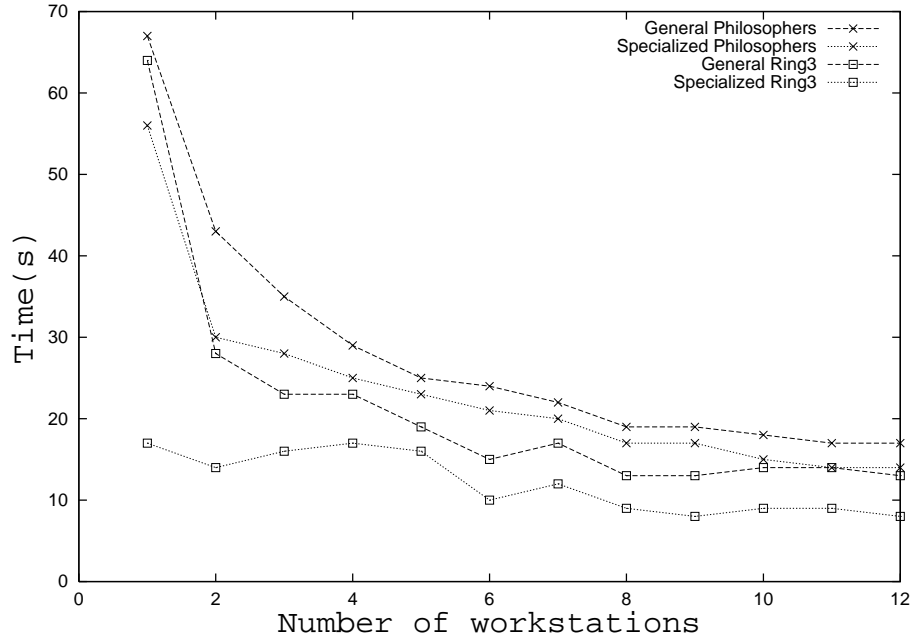


Fig. 3. DETECT-CYCLE and WEAK-DETECT-CYCLE on weak graphs.

We have implemented our approach within an experimental platform. We found out that the complexity of our algorithm is nearly linear. The runtime of the sequential DETECT-CYCLE algorithm is comparable to that of NESTEDDFS on correct specifications. For an erroneous specifications counterexamples generated by our algorithm tend to be significantly shorter. The distributed DETECT-CYCLE algorithm is noteworthy faster than the distributed implementation of NESTEDDFS for all types of graphs. In the future we plan to implement our approach to an existing tool and to compare its efficiency with other distributed LTL model checking algorithms ([1, 8]).

There are several alternatives to *One Way Catch Them Young* in the literature, for excellent reviews see [30, 16]. The natural question thus is whether similar distributed algorithms for fair cycle detection as the one we have proposed can be build upon other symbolic algorithms for cycle detection.

References

1. J. Barnat, L. Brim, and J. Štříbrná. Distributed LTL Model-Checking in SPIN. In *Proc. SPIN Workshop on Model Checking of Software*, volume 2057 of LNCS, pages 200 – 216. Springer, 2001.
2. J. Barnat, L. Brim, and I. Černá. Property driven distribution of Nested DFS. In *Proc. Workshop on Verification and Computational Logic*, number DSSE-TR-2002-5 in DSSE Technical Report, pages 1 – 10. Dept. of Electronics and Computer Science, University of Southampton, UK, 2002.
3. G. Behrmann. A performance study of distributed timed automata reachability analysis. In *Proc. Workshop on Parallel and Distributed Model Checking*, volume 68 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

4. S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Proc. Formal Methods in Computer-Aided Design*, volume 1954 of LNCS, pages 390–404, 2000.
5. R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In *Proc. Computer Aided Verification*, volume 1633 of LNCS, pages 222–235. Springer, 1999.
6. B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free μ -calculus. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of LNCS, pages 543–558. Springer, 2001.
7. R. K. Brayton et al. VIS: a system for verification and synthesis. In *Proc. Formal Methods in Computer Aided Design*, volume 1166 of LNCS, pages 248 – 256. Springer, 1996.
8. L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of LNCS, pages 96–107. Springer, 2001.
9. R.E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, volume C-35(8), pages 677 – 691, 1986.
10. J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. International Congress on Logic, Methodology and Philosophy Science*, pages 1 – 11. Stanford university Press, 1960.
11. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
12. I. Černá and R. Pelánek. Relating the hierarchy of temporal properties to model checking. Submitted, 2002.
13. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.
14. C. Eisner and D. Peled. Comparing symbolic and explicit model checking of a software system. In *Proc. SPIN Workshop on Model Checking of Software*, volume 2318 of LNCS, page 230–239. Springer, 2002.
15. E. A. Emerson and C.-L. Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8:275 – 306, 1987.
16. K. Fisler, R. Fraer, G. Kamhi Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Proc. Tools and Algorithms for Construction and Analysis of Systems*, volume 2031 of LNCS, pages 420–434. Springer, 2001.
17. H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proc. SPIN Workshop on Model Checking of Software*, volume 2057 of LNCS, pages 215+. Springer, 2001.
18. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.
19. R. H. Hardin, A. Harel, and R. P. Kurshan. COSPAN. In *Proc Conference on Computer Aided Verification*, volume 1102 of LNCS, pages 423 – 427. Springer, 1996.
20. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proc. Conference on Computer Aided Verification*, volume 1855 of LNCS, pages 20–35. Springer, 2000.
21. R. Hojati, R. K. Brayton, and R. P. Kurshan. BDD-based debugging using language containment and fair CTL. In *Proc. Conference on Computer Aided Verification*, volume 697 of LNCS, pages 41 – 58. Springer, 1993.
22. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
23. G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. SPIN Workshop on Model Checking of Software*, pages 23–32. American Mathematical Society, 1996.
24. A. J. Hu. *Techniques for efficient formal verification using binary decision diagrams*. PhD thesis, Stanford University, 1995.

25. A. J. Hu, G. York, and D. L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *Proc. Design automation Conference*, pages 276 – 282, 1994.
26. Y. Kesten, A Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. Automata, Languages and Programming*, volume 1443 of *LNCS*, pages 1–16. Springer, 1998.
27. R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenign1. Static partial order reduction. In *Tools and Algorithms for Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 345 – 357. Springer.
28. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. SPIN Workshop on Model Checking of Software*, volume 1680 of *LNCS*, Berlin, Germany, 1999. Springer.
29. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1994.
30. K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *Proc. Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, pages 143–160. Springer, 2000.
31. J.H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.
32. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Proc. Computer Aided Verification*, volume 1855 of *LNCS*, pages 248–263. Springer, 2000.
33. U. Stern and D.L. Dill. Parallelizing the Mur φ verifier. In *Proc. Computer Aided Verification*, volume 1254 of *LNCS*, pages 256–267. Springer, 1997.
34. J. R. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54(1 - 2):121 – 141, 1982.
35. R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on computing*, pages 146–160, 1972.
36. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *LNCS*, pages 238 – 266. Springer, 1996.

**Copyright © 2002, Faculty of Informatics, Masaryk University.
All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

**Publications in the FI MU Report Series are in general accessible
via WWW and anonymous FTP:**

`http://www.fi.muni.cz/informatics/reports/
ftp ftp.fi.muni.cz (cd pub/reports)`

Copies may be also obtained by contacting:

**Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic**