Distributed Explicit Fair Cycle Detection

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 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 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 [13].

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 [29, 23]. Fair cycle detection is used to check the CTL formula $\mathbf{EG}f$ under the full (generalised) fairness constraints [13]. Hence, the 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

* Supported by GA ČR grant no. 201/03/0509

SPIN [20] and SMV [26] respectively, in fair-CTL model checkers such as SMV, VIS [7], and COSPAN [17].

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 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 ([30, 25, 15, 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 [31]. 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 [21] (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 [28]. 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], 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 [14, 27]). 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. Nevertheless, there are applications where explicit model checkers outperform the others, for examples see [30, 22, 12]

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 [18] 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 ([14]) 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 [19]. 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 emphasis 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 [24].

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. The proof of its correctness and analysis of its complexity can be found in Appendix. 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.

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 $(s_1, \ldots, s_k) \in V^+$ 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 set of states 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 (with respect to set inclusion) 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 a 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 [14] and [27]. As the base for our algorithm we have chosen the *One Way Catch Them Young* algorithm [14]. The reasons for this choice are discussed in Section 4.

Symbolic algorithms are conveniently described with the help of μ-calculus formulae. Our algorithm makes use of the functions $\mathit{Reachability}(S) = \mu Z.(S \cup \mathit{image}(Z))$ and $\mathit{Elimination}(S) = \nu Z.(S \cap \mathit{image}(Z))$. The set $\mathit{image}(Z)$ contains all successors of states from Z in a graph G. The function $\mathit{Reachability}(S)$ computes the set of all states that are reachable from the set S. The function $\mathit{Elimination}(S)$ 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 $\mathit{Elimination}(S)$ 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 := ∅;
  while (S ≠ old) do
        old := S;
        S := Reachability(S ∩ F);
        S := Elimination(S);
  od
  return (S ≠ ∅);
end
```

The assignment $S := \mathit{Reachability}(S \cap F)$ 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 $S := \mathit{Elimination}(S)$ 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 OWCTY is $O(n^2)$ steps[1] 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 [14].

3 Sequential algorithm

In this section we present a new sequential algorithm for fair cycle detection problem, prove it correctness, and analyse its complexity. The distributed version of the algorithm is discussed 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 the 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:

- *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 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.

```
proc DETECT-CYCLE(G, F, init_state)
  put init_state into queue;
  inS[init_state] := true;
  REACHABILITY;
  while (Ssize ≠ oldSsize ∧ Ssize > 0) do
        RESET;
        REACHABILITY;
        ELIMINATION;
  od
  return(Ssize > 0);
end
```

[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.

```
proc Reset
  oldSsize := Ssize;
  Ssize := 0;
  foreach q ∈ V do
    inS[q] := inS[q] ∧ q ∈ F;
    p[q] := 0;
    if inS[q] then Ssize := Ssize + 1;
                   put q in queue;
                   put q in L; fi
  od
end
```

```
proc Reachability
  while queue ≠ ∅ do
        remove q from queue;
        foreach (q, r) ∈ E do
          if (¬inS[r]) then inS[q] := true;
                            Ssize := Ssize + 1;
                            put r in queue; fi
          if p[r] = 0 then remove r from L; fi
          p[r] := p[r] + 1;
        od
  od
end
```

```
proc Elimination
  while L ≠ ∅ do
        remove q from L;
        inS[q] := false;
        Ssize := Ssize − 1;
        foreach (q, r) ∈ E do
          p[r] := p[r] − 1;
          if p[r] = 0 then put r to L fi
        od
  od
end
```

Theorem 1 (Correctness). Detect-Cycle *terminates and returns true if and only if G contains a fair cycle reachable from the init_state.*

Theorem 2 (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.*

Proofs of both theorems are in full version of the paper [10].

3.2 Extraction of a Fair Cycle

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 the extraction 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 revealed fair cycles (i.e. counterexamples) tend to be much shorter than those generated by depth-first based algorithms (see Section 6).

```
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] ∧ s ∈ F then NESTEDBFS(s); fi
        foreach (s, r) ∈ E do
          if parent[r] = nil then parent[r] := s;
                                    put r in queue; fi
        od
  od
  while s ≠ 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) ∈ E do
          if inS[r] ∧ parent2[r] = nil then parent2[r] = q;
                                             put r in queue2 fi
          if r = s then cycle found;
                        r := parent2[r];
                        while r ≠ s do output r; r := parent2[r]; od
          fi
        od
        inS[q] := false;
  od
end
```

Theorem 3 (Correctness). *The* EXTRACT-CYCLE *procedure finds a fair cycle. 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.*

Theorem 4 (Complexity). *The complexity of* EXTRACT-CYCLE *is* $O(n + m)$.

4 Distributed Algorithm

Similar to other works devoted to the distributed model checking [6, 3, 8, 30, 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 (see e.g. [2]).

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.

Computations on particular workstations can be performed in parallel. However, some synchronisation is unavoidable. All workstations perform the same procedure (Re- set, Reachability, or Elimination). 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 at al. [14] indicates that this algorithm performs less external iterations then for example the well-known Emerson-Lei algorithm[2]. The number of external iteration determines the number of necessary synchronisations.

Due to space limitations we do not display the pseudo-code of the distributed Detect- Cycle algorithm. It can be found in the full version of the paper [10].

The distributed counterpart of the procedure Extract-Cycle comes by in a similar way as for Detect-Cycle. The important point is that only one NestedBFS can be performed at a time. 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.

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

[2] We note that some other algorithms studied by [14] 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

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.

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 [11] 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 [29] 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 ∩ F);
  return (S ≠ ∅);
end
```

The algorithm WEAK-DETECT-CYCLE has several advantages. At first, its complexity is $O(n + m)$. This 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.

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.

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 [16]. 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 the 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

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 [23]).

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

proc STREETT-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 [23]. 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 algorithms. 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.

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`).

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 having 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 [14] 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) NestedDFS 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)	External Iterations
`Peterson`	NestedDFS	0.02	
376	Detect-Cycle	0.06	18
`ABP`	NestedDFS	0.22	
7 286	Detect-Cycle	0.41	1
`Ring1`	NestedDFS	17.13	
172 032	Detect-Cycle	7.61	1
`Elevator2`	NestedDFS	35.10	
368 925	Detect-Cycle	55.76	30
`Philosophers`	NestedDFS	72.68	
608 185	Detect-Cycle	52.04	1

Table 1. Sequential experiments for graphs without fair cycles.

System Size	Algorithm	Time (s)	Extract time (s)	External Iterations	Fair cycle Prefix	 Loop
`Mutex`	NestedDFS	0.01			76	3
232	Detect-Cycle	0.02	0.01	2	2	2
`Ring3`	NestedDFS	2.70			14420	3
389 542	Detect-Cycle	29.07	1.17	2	28	23
`Elevator1`	NestedDFS	7.28			304	76
683 548	Detect-Cycle	99.43	1.80	8	20	22
`Ring2`	NestedDFS	12.82			2754	363
1 027 394	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.

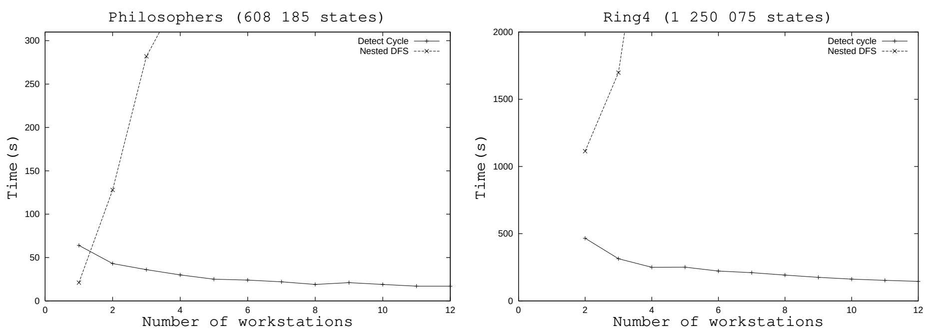

Fig. 1. Comparison of distributed NestedDFS and Detect-Cycle.

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 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. Experiments indicate that the specialised algorithm can yield a considerable improvement (see the full version [10]).

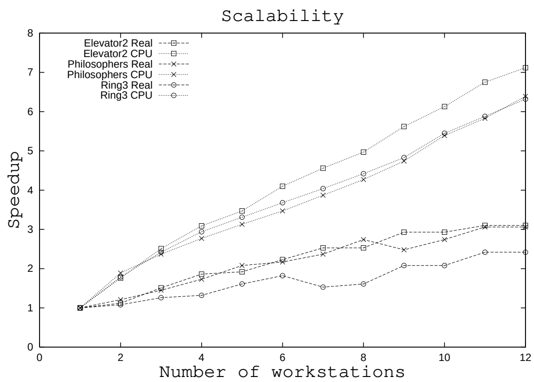

Fig. 2. Dependency of the runtime on the number of workstations. Figure shows the difference between real time taken by the program and the average CPU time used by a workstation.

7 Conclusions & Future Work

In this paper, we present 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.

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 [27, 14]. 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. Stříbrná. Distributed LTL Model-Checking in SPIN. In *Proc. SPIN Workshop*, 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. University of Southampton, UK, 2002.
3. G. Behrmann. A performance study of distributed timed automata reachability analysis. In *Proc. PDMC*, volume 68 of *ENTCS*. Elsevier, 2002.
4. S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Proc. FMCAD*, 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. CAV*, 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. TACAS*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.
7. R. K. Brayton et al. VIS: a system for verification and synthesis. In *Proc. FMCAD*, 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. FST TCS*, 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. I. Černá and R. Pelánek. Distributed explicit fair cycle detection. Technical Report FIMU-RS-2002-09, Faculty of Informatics, Masaryk University, 2002. www.fi.muni.cz/informatics/reports/.
11. I. Černá and R. Pelánek. Relating the hierarchy of temporal properties to model checking. Submitted, 2002.
12. C. Eisner and D. Peled. Comparing symbolic and explicit model checking of a software system. In *Proc. SPIN Workshop*, volume 2318 of *LNCS*, page 230 239. Springer, 2002.
13. E. A. Emerson and C.-L. Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8:275 – 306, 1987.
14. K. Fisler, R. Fraer, G. Kamhi Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Proc. TACAS*, volume 2031 of *LNCS*, pages 420–434. Springer, 2001.
15. H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proc. SPIN Workshop*, volume 2057 of *LNCS*, pages 215+. Springer, 2001.
16. 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.
17. R. H. Hardin, A. Harel, and R. P. Kurshan. COSPAN. In *Proc. CAV*, volume 1102 of *LNCS*, pages 423 – 427. Springer, 1996.
18. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proc. CAV*, volume 1855 of *LNCS*, pages 20–35. Springer, 2000.
19. R. Hojati, R. K. Brayton, and R. P. Kurshan. BDD-based debugging using language containment and fair CTL. In *Proc. CAV*, volume 697 of *LNCS*, pages 41 – 58. Springer, 1993.
20. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
21. G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. SPIN Workshop*, pages 23–32. American Mathematical Society, 1996.
22. 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.
23. Y. Kesten, A Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. ICALP*, volume 1443 of *LNCS*, pages 1–16. Springer, 1998.
24. R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenign1. Static partial order reduction. In *Proc. TACAS*, volume 1384 of *LNCS*, pages 345 – 357. Springer, 1998.
25. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. SPIN Workshop*, volume 1680 of *LNCS*, Berlin, Germany, 1999. Springer.
26. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1994.
27. K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *Proc. FMCAD*, volume 1954 of *LNCS*, pages 143–160. Springer, 2000.
28. J.H. Reif. Depth-first search is inherrently sequential. *Information Processing Letters*, 20(5):229–234, 1985.
29. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Proc. CAV*, volume 1855 of *LNCS*, pages 248–263. Springer, 2000.
30. U. Stern and D.L. Dill. Parallelizing the Murφ verifier. In *Proc. CAV*, volume 1254 of *LNCS*, pages 256–267. Springer, 1997.
31. R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on computing*, pages 146–160, 1972.