Distributed Explicit Fair Cycle Detection

Ivana Čerňa 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 dis-
tribution of explicitly presented data structure and the features of symbolic algo-


rithm 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 decid-
ing emptiness of \( \omega \)-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 cy-


cle, 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 autom-

on 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 iden-
tification 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 [33,24]. Fair cycle detection is used to check the CTL
formula \( \text{EG} f \) under the full (generalised) fairness constraints [13]. Hence, the core pro-


cur 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 [20]

* Supported by GA ČR grant no. 201/00/1023
and SMV [27] 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 ([31, 26, 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 [32]. 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] (NESTED DFS) optimizes the memory requirements and is able to detect cycles on-the-fly. This makes NESTED DFS 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 [29]. 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, 28]). 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 [31, 22, 23, 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.

2
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 emphasise 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 [25].

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 $\omega$-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 state set $U$, the graph $G(U) = (U, E \cap (U \times U))$ is the graph induced by $U$. 

3
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 $\text{init\_state}$ 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 [28]. 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 $\mu$-calculus formulae. Our algorithm makes use of the functions $\text{Reachability}(S) = \mu Z. (S \cup \text{image}(Z))$ and $\text{Elimination}(S) = \nu Z. (S \cap \text{image}(Z))$. The set $\text{image}(Z)$ contains all successors of states from $Z$ in a graph $G$. The function $\text{Reachability}(S)$ computes the set of all states that are reachable from the set $S$. The function $\text{Elimination}(S)$ 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 $\text{Elimination}(S)$ 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:

```plaintext
proc OWCT(Y(G, F, init\_state))
  S := \text{Reachability}(\text{init\_state});
  old := \emptyset;
  while (S \neq old) do
    old := S;
    S := \text{Reachability}(S \cap F);
    S := \text{Elimination}(S);
  od
  return (S \neq \emptyset);
end
```

The assignment $S := \text{Reachability}(S \cap F)$ 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 $S := \text{Elimination}(S)$ 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 $\text{Reachability}(S)$ function is straightforward (see the procedures $\text{RESET}$ and $\text{REACHABILITY}$ respectively). The emulation of $\text{Elimination}(S)$ is more involved: concurrently with the emulation of $\text{Reachability}(S)$ 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 $\text{ELIMINATION}$. 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, 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.
- $\text{queue}$ is an initially empty queue.

\begin{verbatim}
proc DETECT-CYCLE(G, F, init_state)
  put init_state into queue;
  inS[init_state] := true;
  \text{REACHABILITY};
  while (Ssize \neq oldSsize \land Ssize > 0) do
    \text{RESET};
    \text{REACHABILITY};
    \text{ELIMINATION};
  od
  return(Ssize > 0);
end
\end{verbatim}

\(^1\) 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.
1 proc RESET
2 old Ssize := Ssize;
3 Ssize := 0;
4 foreach q ∈ V do
5 inS[q] := inS[q] ∧ q ∈ F;
6 p[q] := 0;
7 if inS[q] then Ssize := Ssize + 1;
8 put q in queue;
9 ụ
10 od
11 end

1 proc REACHABILITY
2 while queue ≠ ∅ do
3 remove q from queue;
4 foreach (q, r) ∈ E do
5 if ¬inS[r] then inS[q] := true;
6 Ssize := Ssize + 1;
7 put r in queue;
8 if p[r] = 0 then remove r from L;
9 p[r] := p[r] + 1;
10 ụ
11 od
12 end

1 proc ELIMINATION
2 while L ≠ ∅ do
3 remove q from L;
4 inS[q] := false;
5 Ssize := Ssize − 1;
6 foreach (q, r) ∈ E do
7 p[r] := p[r] − 1;
8 if p[r] = 0 then put r to L;
9 ụ
10 od
11 end

Theorem 1 (Correctness), DETECT-CYCLE terminates and returns true if and only if G contains a fair cycle reachable from the initial 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 Appendix.

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).

```plaintext
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
    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
        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, 31, 4] we assume the MIMD architecture of a network of workstations, which communicate via mes-
sage 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.

The procedures \textsc{Reset}, \textsc{Reachability}, and \textsc{Elimination} can be easily trans-
formed 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 work-
station sends an appropriate message to the \textit{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 (\textsc{Re-
set}, \textsc{Reachability}, or \textsc{Elimination}). As soon as a workstation completes the pro-
cedure it sends a message to the \textit{Manager} and becomes idle. When all workstations are
idle and there are no pending messages the \textit{Manager} synchronises all workstations and
the computation continues.

The need of synchronisation after each procedure is the reason why we have cho-
sen the \textit{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 than for example the well-known Emerson-Lei algorithm\footnote{We note
that some other algorithms studied by [14] perform even less external iterations. These
algorithms make use of the \textit{preimage} computation (i.e. computation of predecessors),
which is usually not available in the explicit model checking}. 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 \textsc{Detect-
Cycle} algorithm. It can be found in the full version of the paper [10].

The distributed counterpart of the procedure \textsc{Extract-Cycle} comes by in a similar
way as for \textsc{Detect-Cycle}. The important point is that only one \textsc{Nested BFS} 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 \textit{Manager}. The \textit{Manager} puts the incoming candidates into a queue and successively
starts \textsc{Nested BFS} 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
variants called \textit{weak} and \textit{generalised} Büchi automata and \textit{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 \textsc{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 [30] 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-DETekt-CYCLE(G, F, init_state)
    S := Reachability(init_state);
    S := Elimination(S \cap F);
    return (S \neq \emptyset);
end
```

The algorithm `WEAK-DETekt-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 `NESTED-DFS` 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* $F$ 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.
**Procedure** GENERALIZED-DETETCYCLE($G, \mathcal{F}, \text{init}_\text{state}$)

$S := \text{Reachability}($init\_state$)$;
old := $\emptyset$

while ($S \neq \text{old}$) do
old := $S$
foreach $F_i \in \mathcal{F}$ do
$S := \text{Reachability}(S \cap F_i)$; \textbf{end}
$S := \text{Elimination}(S)$; \textbf{end}
return ($S \neq \emptyset$);

**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 [24]).

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

**Procedure** STREETT-DETECTCYCLE($G, \mathcal{F}, \text{init}_\text{state}$)

$S := \text{Reachability}($init\_state$)$;
old := $\emptyset$

while ($S \neq \text{old}$) do
old := $S$
foreach $(P, Q) \in \mathcal{F}$ do
$S := (S - P) \cup \text{Reachability}(S \cap Q)$; \textbf{end}
$S := \text{Elimination}(S)$; \textbf{end}
return ($S \neq \emptyset$);

For the proof of correctness see [24]. 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.

**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.
<table>
<thead>
<tr>
<th>System Size</th>
<th>Algorithm</th>
<th>Time (s)</th>
<th>External Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peterson 176</td>
<td>NESTEDDFS</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>ABF</td>
<td>NESTEDDFS</td>
<td>0.22</td>
<td></td>
</tr>
<tr>
<td>Ring1 172 032</td>
<td>NESTEDDFS</td>
<td>17.13</td>
<td></td>
</tr>
<tr>
<td>Elevator2 368 925</td>
<td>NESTEDDFS</td>
<td>35.10</td>
<td></td>
</tr>
<tr>
<td>Philosophers 608 185</td>
<td>NESTEDDFS</td>
<td>72.68</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DETECT-CYCLE</td>
<td>52.04</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. Sequential experiments for graphs without fair cycles.

<table>
<thead>
<tr>
<th>System Size</th>
<th>Algorithm</th>
<th>Time (s)</th>
<th>Extract time (s)</th>
<th>External Iterations</th>
<th>Fair cycle</th>
<th>Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutex 232</td>
<td>NESTEDDFS</td>
<td>0.01</td>
<td>0.01</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Ring3 589 542</td>
<td>NESTEDDFS</td>
<td>2.70</td>
<td>1.70</td>
<td>2</td>
<td>28</td>
<td>23</td>
</tr>
<tr>
<td>Elevator1 693 548</td>
<td>NESTEDDFS</td>
<td>7.28</td>
<td>1.80</td>
<td>8</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>Ring2 1.027 394</td>
<td>NESTEDDFS</td>
<td>12.82</td>
<td>11.31</td>
<td>40</td>
<td>52</td>
<td>14</td>
</tr>
</tbody>
</table>

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.

![Graphs](image.png)

Fig. 1. Comparison of distributed NESTEDDFS and DETECT-CYCLE.
We have compared our algorithm to the distributed version of \textsc{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 \textsc{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 \textsc{NestedDFS}.

The current implementation of \textsc{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.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{Scalability}
\caption{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.}
\end{figure}

We have also implemented the distributed \textsc{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]).
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 savings 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\cite{1,8}.

There are several alternatives to One Way Catch Them Young in the literature, for excellent reviews see\cite{28,14}. 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


Appendix

Correctness and Complexity of DETECT-CYCLE

In what follows we denote $S$ the set of states $q$ such that $\text{in} S[q] = \text{true}$ and particularly $S_i^l$ the set of states $q$ such that $\text{in} S[q] = \text{true}$ 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 $\text{in} S[q] = \text{true}$. REACHABILITY performs the standard breadth-first search and empties the queue.

**Lemma 2.** The invariant $q \in L \Rightarrow (q \in S \land p[q] = 0)$ 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 $\text{Size}$ is the size of the set $S$.

**Proof:** Whenever a new state $q$ is added to $S$ in REACHABILITY the variable $\text{Size}$ 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, $q \in \text{queue} \Rightarrow q \in S$ 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.**
- $S_0^1 = S_0^0 \cap F$.
- $S_0^i$ is the set of states reachable from the set $S_0^i$.
- $S_0^{i+1}$ is the set of all states $q$ for which either $q$ lies on a cycle in $G(S_0^i)$ or $q$ is reachable in $G(S_0^i)$ from a cycle in $G(S_0^i)$.
Proof: The first equality follows directly from the code of the procedure \textsc{reset}.

The second fact is a direct consequence of Lemma 1, because the content of \textit{queue} at the beginning of \textsc{reachability} is $S^i_0$.

By Lemma 4, value $p[q]$ is the number of direct predecessors of $q$ in $G(S^i_0)$ and only states with none predecessors are removed from $S$ in \textsc{elimination}. Therefore all states with the required property are in $S^{k+1}_0$. 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. $S^{i+1}_0 \subseteq S^i_0$

Proof: The assertion can be proved by induction on $i$. For the base case $i = 1$ we argue that $S^i_0$ is the set of all states reachable from \textit{init.state} and all the states put in $S$ in \textsc{reachability} (line 7) are reachable from \textit{init.state} and thus $S^i_0 \subseteq S^i_0$.

For the general case we suppose $S^{i+1}_0 \subseteq S^i_0$. Then we can reason with the use of Lemma 6 as follows: $S^{i+1}_0 \subseteq S^i_0 \Rightarrow (S^{i+1}_0 \cap F \subseteq S^i_0 \cap F) \Rightarrow (S^{i+1}_0 \subseteq S^i_0 \cap F) \Rightarrow$ each state reachable from $S^{i+1}_0$ is reachable from $S^i_0$ as well $\Rightarrow$ $(S^{i+1}_0 \subseteq S^i_0 \cap F) \Rightarrow$ each state that lies on (or is reachable from) a cycle in $S^{i+1}_0$ lies on (or is reachable from) a cycle in $S^i_0$ as well $\Rightarrow$ $(S^{i+2}_0 \subseteq S^{i+1}_0)$.

Theorem 5 (Termination). The \textsc{detect-cycle} algorithm terminates.

Proof: The termination of \textsc{reachability} and \textsc{elimination} follows from Lemma 5.

The termination of \textsc{reset} is straightforward.

By Lemma 7, $S^{i+1}_0 \subseteq S^i_0$ which together with Lemma 3 ensures that the condition on line 5 eventually becomes false and \textsc{detect-cycle} terminates as well.

Theorem 6 (Completeness). If $G$ contains a fair cycle reachable from the \textit{init.state} then \textsc{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 $q \in S^i_0$. For the base case $i = 1$ we argue that $S^1_0$ is the set of states reachable from \textit{init.state} and thus $q \in S^1_0$.

Now let $q \in S^i_0$. By Lemma 6, $q \in S^i_0$. The state $q$ as well as all the states reachable from $q$ belong to $S^i_0$. Namely, the whole cycle $C$ belongs to $S^i_0$ and by Lemma 6 cycle $C$ belongs also to the set $S^{i+1}$.

Hence after executing the while loop the state $q$ belongs to $S$, therefore $S$ size > 0 (Lemma 3) and \textsc{detect-cycle} returns true.

Theorem 7 (Soundness). If \textsc{detect-cycle} returns true, then $G$ contains a fair cycle reachable from the \textit{init.state}.

Proof: Let us suppose that \textsc{detect-cycle} terminates after $k$ iterations of the while cycle. Since the algorithm returns true, $S^k_0 = S^{k-1}_0$ and $S^k_0$ is nonempty (Lemma 3 and 7).

Let us consider the decomposition of $S^k_0$ 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 $S^{k-1}_0$ contains only states reachable from $S^{k-1}_0 \cap F = S^{k-1}_0 \cap F$ and because $C$ is initial no state from $C$ is in $S^{k-1}_0$. Consequently $C$ is not contained in $S^k_0$ (Lemma 6), a contradiction.

If the component $C$ were trivial, it would be removed from the set $S^k_0 = S^{k-1}_0$ by the procedure \textsc{elimination} due to Lemma 6.
**Theorem 8 (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 $O(n)$. **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 $G(S_i^0)$ 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 $S_i^0$ are removed from $S_i^0$. This claim together with the observations that all SCCs of $S_{i+1}^0$ are also SCCs in $S_i^0$ and the quotient graph of $S_{i+1}^0$ is a subgraph of the quotient graph of $S_i^0$ guarantee that the **while** loop in **Detect-Cycle** iterates at most $h$ times.

**Correctness and Complexity of ** **Extract-Cycle**

**Theorem 9 (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.

**Theorem 10 (Completeness),** The **Extract-Cycle** procedure 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**($G, F, \text{init\_state}$). 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 7, 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 $s$.

**Theorem 11 (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.