



FI MU

**Faculty of Informatics
Masaryk University Brno**

Distributed Memory LTL Model Checking Based on Breadth First Search

by

**Jiří Barnat
Luboš Brim
Jakub Chaloupka**

FI MU Report Series

FIMU-RS-2004-07

Copyright © 2004, FI MU

September 2004

**Copyright © 2004, 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:**

`http://www.fi.muni.cz/veda/reports/`

Further information can be obtained by contacting:

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

Distributed Memory LTL Model Checking Based on Breadth First Search *

Jiří Barnat

Faculty of Informatics, MU Brno
Botanická 68a, 602 00 Brno,
Czech Republic
barnat@fi.muni.cz

Luboš Brim

Faculty of Informatics, MU Brno
Botanická 68a, 602 00 Brno,
Czech Republic
brim@fi.muni.cz

Jakub Chaloupka

Faculty of Informatics, MU Brno
Botanická 68a, 602 00 Brno,
Czech Republic
xchalou1@fi.muni.cz

September 13, 2004

Abstract

We propose a parallel distributed memory on-the-fly algorithm for enumerative LTL model checking. The algorithm is designed for network of workstations communicating via MPI. The detection of cycles (faulty runs) effectively employs the so called back-level edges. In particular, a parallel level synchronized breadth first search of the graph is performed to discover all back-level edges and for each level the back-level edges are checked in parallel by a nested search procedure to confirm or refute the presence of a cycle. Several optimizations of the basic algorithm are presented and advantages and drawbacks of their application to distributed LTL model-checking are discussed. Experimental evaluation of the algorithm is presented.

*Research supported by the Grant Agency of Czech Republic grant No. 201/03/0509

1 Introduction

With the increase in complexity of computer systems, it becomes important to develop formal methods for ensuring their quality. Various techniques for automated and semi-automated analysis and verification of computer systems are routinely used in software engineering practice. In particular, model checking has become a very practical technique due to its push-button character.

Model checking is a fully automated formal technique for the verification of concurrent software. It is based on representation of the system being analyzed as a (finite state) transition system (Kripke model), while system requirements are typically expressed as properties in temporal logics. Deciding whether a system satisfies a given property amounts to checking if the corresponding temporal formula is true in the Kripke model.

Although model checking has been applied fairly successfully to verification of several real life systems, its applicability to a wider class of practical systems has been hampered by the state explosion problem (i.e. the enormous increase in the size of the state space).

The use of distributed memory and/or parallel processing to combat the state explosion problem gained interest in recent years. For large industrial models, the state space does not completely fit into the main memory of a computer and hence model checking algorithm becomes very slow as soon as the memory is exhausted and system starts swapping. A possible approach to dealing with these practical limitations is to increase the computational power (especially random access memory) by building a powerful parallel computer as a network (cluster) of workstations. Individual workstations communicate through message passing interface. From outside a cluster appears as a single parallel computer with high computing power and huge amount of memory.

In this paper we present a novel distributed memory approach to explicit-state (enumerative) model checking for linear temporal logic (LTL). LTL is a major logic used in formal verification known for very efficient sequential solution based on automata [13] and successful implementation within several verification tools.

The best sequential algorithm for the LTL model checking problem is the Nested DFS. Unfortunately, it employs the crucial DFS postorder to perform the cycle detection which is not maintained any more if the algorithm is adopted to the distributed memory parallel environment. Since all the solutions suggested up to now build on the

Nested DFS, they all have to cope with this problem. While one algorithmic solution significantly restricts the partition function used for dividing the Kripke model among participating workstations (let alone the token based solution that prevent parallelism at all), the other one maintains complex additional data structure.

We propose a completely different algorithmic solution that avoids the problematic DFS postorder but employs so called back-level edges to perform the cycle detection. In short, a back-level edge is any transition in the graph that does not increase the breadth first search distance from the initial state. Note that any cycle in the graph has to contain at least one back-level edge. Obviously, the breadth first search distances can be computed by a breadth first search (BFS) which can be, contrary to the DFS, reasonably parallelized. The idea of the new algorithm is to perform a distributed memory level-synchronized breadth first search of the graph to discover all the back-level edges and to check each back-level edge for being a part of a cycle.

The rest of the paper is organized as follows. Section 2 gives a brief recapitulation of the LTL model-checking problem including the explanation of so called state explosion problem and partial order reduction technique. Section 3 introduces the main idea of the parallel algorithm. Focus is on the distributed discovering of back-level edges. Section 4 presents a technique for detection of cycles. Section 5 elaborates various optimizations of the cycle detection algorithm. Section 6 describes how to modify the algorithm to detect accepting cycles. Section 7 adds counterexample generation capability to the algorithm. Section 8 combines the algorithm with the partial order reduction technique. Section 11 focuses on the complexity issues. Section 9 summarizes case-studies and experiments we have conducted and finally, in Section 10 we relate our approach to the other work on distributed LTL model-checking, give some conclusions and outline future work.

2 LTL Model Checking

Formulas of a linear temporal logic are made of atomic propositions and boolean and temporal operators. The basic commonly used LTL temporal operators are G – globally, F – eventually, and U – until. For example, the formula $GF(x > 1)$ means that at every moment (G) during a system execution there is a future moment (F) in the execution such that $(x > 1)$ holds. Each LTL formula thus defines a set of executions. A set of executions can be equivalently represented as a Büchi automaton. A *Büchi automaton* is a

finite automaton accepting infinite words by passing through an *accepting state* infinitely many times. Therefore, a language (of infinite words) accepted by a Büchi automaton is non-empty (the automaton accepts at least one infinite word) if and only if there is at least one cycle containing an accepting state reachable from the initial state in its underlying graph.

In the automata-theoretic approach to LTL model-checking [13] the set of all system executions is represented as a Büchi automaton as well. A system meets a property if and only if all possible executions of the system satisfy the property given by an LTL formula. The decision procedure is as follows. First, the verified formula is negated and a Büchi automaton representing all invalid (property-breaking) executions is built. It is called a *negative claim automaton*. Then a new Büchi automaton corresponding to the synchronous product of the Büchi automaton representing the system behavior and the negative claim automaton is constructed. Obviously, the language of the new *product automaton* is empty if and only if the system does not contain an invalid execution. Hence, the LTL model-checking problem is reduced to the emptiness problem for Büchi automata.

A straightforward approach to solving the emptiness problem is to decompose the underlying graph into strongly connected components (SCCs) which can be done in time linear in the size of the graph using Tarjan's algorithm [12]. However, constructing SCCs is not memory efficient since the states in the components must be stored explicitly during the procedure. Courcoubetis et al. [7] have proposed an elegant way to avoid the explicit computation of SCCs. The idea is to use a *nested depth first search* – *Nested DFS* to find accepting states that are reachable from themselves (to compute *accepting cycles*). The first search (*primary*) is used to search for reachable accepting states while the second one (*nested*), initiated for each discovered accepting state, tries to detect reachability of the accepting state from itself proving thus existence of an accepting cycle. Each individual nested search need not visit states already visited by previous nested searches resulting thus in linear time complexity, however, to ensure the correctness of the algorithm the nested searches have to be performed in the DFS postorder. This makes the algorithm “inherently sequential”, hence unsuitable for usage in parallel and distributed environment.

Finally, we briefly recall one of the other successful techniques for fighting the state space explosion problem. It is called partial order reduction [5]. This technique involves verification of representative executions only instead of the whole system be-

havior. These representatives are chosen somehow automatically during the traversal of the graph by exploring only a subset of all possible successors of a state. Moreover, to ensure the correctness of the reduction at least one state on every cycle has to be fully expanded.

Since LTL model checking can be reduced to the detection of (accepting) cycles in the graph, we sometimes use graph notation. In particular, we do not distinguish between states and vertices and between edges and transitions. We use the notions as it suits us.

3 Back-level edge concept

Before presenting the distributed memory algorithm for back-level edge detection we give the formal definition of the back-level edge concept and introduce some notation.

Definition 3.1. *Let $\mathcal{G} = (V, E)$ be a graph with the source vertex s and let $u \in V$ be its vertex. By $d(u)$ we denote the distance of the vertex u from the source vertex s . Let $\pi = s=s_0, s_1, s_2, \dots, s_{n-1}, s_n=u$ be the shortest path from s to u , then $d(u) = n$. The set of vertices with the same distance is called level, the set $\{u \in V \mid d(u) = k\}$ is denoted by $\text{level}(k)$.*

Definition 3.2. *Let $\mathcal{G} = (V, E)$ be a graph with the source vertex s . An edge $(u, v) \in E$ is called a back-level edge if and only if $d(u) \geq d(v)$. The vertex u is called start vertex of the back-level edge, v is called destination vertex.*

The relation between cycles in a graph and the back-level edges is stated in the following lemmas.

Lemma 3.3. *Any cycle in the graph \mathcal{G} contains at least one back-level edge.*

Proof: Let $\pi = v_0, \dots, v_n, n \geq 0$ be a cycle in the graph \mathcal{G} such that it does not contain a back-level edge. As the length of the path is at least one we have $d(v_0) < d(v_n)$. But $v_0 = v_n$ (the path is a cycle), hence $d(v_0) = d(v_n)$ which is a contradiction. ■

Lemma 3.4. *For each cycle $\pi = v_0, \dots, v_n, n \geq 0$ there exists $j : 0 \leq j \leq n$ such that $d(v_j) \geq d(v_i)$ for all $i : 0 \leq i \leq n$. Furthermore, any edge in the cycle π emanating from the state v_j is a back-level edge. Denote $d(v_j)$ by $\text{maxdepth}(\pi)$.*

Proof: A cycle is a finite path, hence the existence of the maximal value $\text{maxdepth}(\pi)$ for vertices in the cycle is obvious. Consider an edge (v, u) of the cycle π such that $d(v) = \text{maxdepth}(\pi)$. Suppose (v, u) is not a back-level edge. Then $d(v) < d(u)$ and

$\text{maxdepth}(\pi) < d(u)$. This is in contradiction with the maximality of $\text{maxdepth}(\pi)$. ■

Note that a cycle can contain several back-level edges and also more than one back-level edge of “maximal depth”.

In a sequential case the back-level edges can be easily identified by a slightly modified breadth first search of the graph. Breadth-first search is a simple algorithm that for a given graph $\mathcal{G} = (V, E)$ and a *source* vertex $s \in V$ systematically explores the edges of \mathcal{G} to discover every vertex reachable from s . It expands the *frontier* between discovered and undiscovered vertices uniformly across the breadth of the graph (hence the name). An important well known property of a breadth first search is given in the following lemma.

Lemma 3.5. *Let $\mathcal{G} = (V, E)$ be a graph with the initial state s and let $u \in V$ be its state. If a breadth first search algorithm expands a state u whose distance is $d(u)$ then all the states v such that $d(v) < d(u)$ have already been expanded by the algorithm.*

The modification of the breadth first search we employ to detect back-level edges is actually known as a *single source shortest path* algorithm [6]. The SSSP algorithm is able to compute the distance as defined above for all the reachable states in such a way that the distance of a state is computed and assigned to the state before the state is expanded and edges emanating from it explored. This is important since we intend to build an algorithm that works on-the-fly. To distinguish states that have already been expanded (and so the distance have been assigned to them) from the other ones the algorithm maintains the set of visited states.

Having computed the distances for already explored states we can easily check each edge for being a back-level edge immediately during its exploration. In particular, the three following cases are possible (suppose that the explored edge is (u, v))

- $d(v)$ is not computed yet — (u, v) is not a back-level edge,
- $d(v)$ is already computed but $d(v) > d(u)$ — (u, v) is not a back-level edge,
- $d(v)$ is already computed and $d(v) \leq d(u)$ — (u, v) is a back-level edge.

The pseudo-code of the sequential algorithm for detection of back-level edges is given in Figure 1.


```

1 proc BL-EDGE-DETECTION
2    $u := F_{init}()$ 
3    $d(u) := 0$ 
4   Queue :=  $\emptyset$ 
5   enqueue(Queue, u)
6   Visited = {u}
7   while (Queue  $\neq \emptyset$ ) do
8      $u := dequeue(queue)$ 
9     foreach  $v \in F_{succs}(u)$  do
10      if  $v \notin Visited$ 
11      then  $d(v) = d(u) + 1$ 
12          enqueue(Queue, v)
13          Visited := Visited  $\cup \{v\}$ 
14      else if  $d(v) \leq d(u)$ 
15          then Print (“Back-level edge (u,v)”)
16      fi
17    fi
18  od
19 od
20 end

```

Figure 1: Sequential algorithm for back-level edge detection

If we want to perform the algorithm BL-EDGE-DETECTION in a distributed memory environment, several additional facts have to be taken into account. The graph is distributed among workstations in the standard way. Thus each workstation explores only states that it owns, i.e. those successors of a currently explored state that are local are enqueued to the local queue and those successors that are remote are sent to the queues of relevant workstations. The main problem of such a distributed memory algorithm is that the “breadth first search frontier” can get split, i.e. there may be a state that is expanded before all the states with lower distance, which obviously breaks the property given in Lemma 3.5. This is due to different speeds of participating workstations and delays caused by the network communication. As a result some back-level edges might remain undetected and vice versa. We can illustrate the case on the graph in Figure 2.

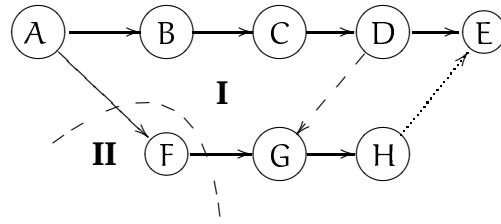


Figure 2: Wrongly detected Back-Level Edges

Suppose that the graph is partitioned between workstations I and II and that A is the initial state of the graph. Due to the network communication delay we can be almost sure that the state G will be visited by the distributed memory breadth first search algorithm through the path A, B, C, D, G (performed entirely on the workstation I) before it is encountered through the path A, F, G whose part is computed on the workstation II. Hence, the distance assigned to the state G will be greater than the distance assigned to the state F and the back-level edge (F, G) will not be revealed. Moreover, the distance assigned to the state H will be greater than the distance assigned to the state E which means that the edge (H, E) will be erroneously reported as a back-level edge.

Several solutions are at hand. The common one exploits the fact that the algorithm is able to detect a state that is assigned an invalid distance value. In particular, whenever the difference between the distance of the source state of an edge and the distance of the destination state of the edge is greater than one then the distance associated to the destination state is invalid. Obviously, it can be improved to the value of the distance of the source state plus one. To compute the distances correctly it suffices to initiate a new exploration at the source state of the edge and let it propagate the new improved distances to all relevant successors. However, this requires reexploration of some states. To show this let us consider again the situation given in Figure 2. The distance associated to the state F is 1 but the distance associated to the state G is 4 when the edge (F, G) is explored. It must be the case that the distance associated to the state G is invalid. Therefore, it is fixed to keep the value of 2 and reexploration of states below G is initiated. In our case, it fixes only the distance associated to the state H .

If we use this solution, we can be sure that the correct distance value will be eventually assigned to all the states, but unfortunately the set of back-level edges as generated by such an algorithm is quite useless. To get a meaningful set of back-level edges we

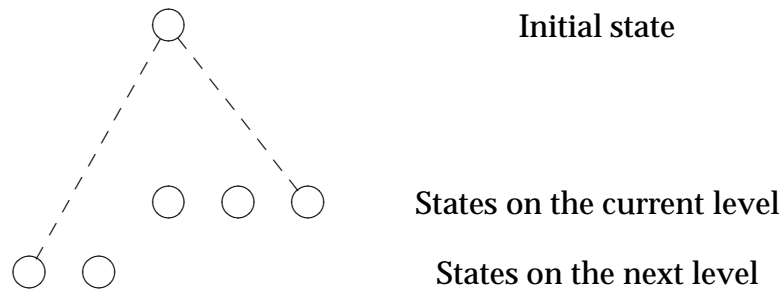


Figure 3: Breadth first search levels

can postpone searching for back-level edges until the graph is completely explored and distances are fixed. However, this is not a good solution because it requires two explorations of the graph and prevents on-the-fly detection of back-level edges (and so cycles). Hence, for our distributed memory algorithm we chose a different solution that requires some synchronization but does not suffer from the mentioned disadvantages.

The chosen solution builds on preventing the breadth first search frontier from getting split. In particular, every workstation in the distributed memory algorithm organizes states from its part of the frontier into two queues, the current level queue (CLQ) and the next level queue (NLQ). See Figure 3 for graphical illustration of the levels. Each workstation processes states from the current level queue only. Note that parallel processing of states from current level queues of different workstations does not break the property given in Lemma 3.5. When a workstation generate a new state, the state is inserted in the next level queue. Naturally, the remote states are not stored in local next level queue but they are sent to the owning workstations at first and stored to the next level queue there. Workstations participating the distributed memory computation synchronize (by calling `SYNCHRONIZE()`) as soon as all their current level queues are empty. After the synchronization each workstation moves all the states from the next level queue to the current level queue (hence the next level queue is cleared) and continues with the exploration of states from the current level queue.

Note that it is difficult in the distributed computation to compute the distance for a newly generated state if its relevant predecessor is not local to the same workstation. In such a case the workstation cannot access the distance value of the predecessor. However, if we perform level synchronized breadth first search it is easier to count the number of synchronizations to obtain the distance of a state than to compute it from the

```

1 proc DISTRIBUTED-MEMORY-BL-EDGE-DETECTION(WorkstationId)
2   CLQ =  $\emptyset$ ; NLQ =  $\emptyset$ ; Visited =  $\emptyset$ ; Level = 0
3   initstate :=  $F_{init}()$ ; finished := false
4   if (WorkstationId = Owner(initstate))
5     then enqueue(NLQ, (-, initstate))
6   fi
7   while ( $\neg$ finished) do
8     swap(NLQ, CLQ)
9     while (CLQ  $\neq$   $\emptyset$ ) do
10      (p, v) := dequeue(CLQ)
11      if (v  $\notin$  Visited)
12        then Visited := Visited  $\cup$  {v}
13        d(v) := Level;
14        foreach t  $\in$   $F_{succs}(v)$  do
15          if (Owner(t)  $\neq$  WorkstationId)
16            then SendTo(Owner(t), enqueue(NLQ, (v, t)))
17            else enqueue(NLQ, (v, t))
18          fi
19        od
20        else if (d(v) < Level)
21          then Print("Back-level edge (p,v)")
22        fi
23      fi
24    od
25    Synchronize(finished := (all NLQ =  $\emptyset$ ))
26    Level = Level + 1
27  od
28 end

```

Figure 4: Distributed memory algorithm for back-level edge detection

distance of its predecessor. The distributed memory algorithm uses the variable `Level` to count the synchronizations and to set the distances.

Similarly, it is difficult to report a back-level edge if the distributed memory computation is considered. A back-level edge is detected and reported when its destination state is reached. Alas, the source state of the edge may be remote to the workstation that detected the back-level edge and thus locally inaccessible. To solve this problem we have to modify slightly the contents of the current level and next level queues. While in the sequential algorithm the objects enqueued to the queue representing the frontier are single states, in the distributed memory algorithm the objects enqueued to both current level and next level queues are the explored edges. Each edge is actually a pair of states that contains the state to be expanded and its predecessor. Naturally, this also requires to enlarge remote state network messages to bear not only the state to be expanded but also the corresponding predecessor. The pseudo-code of the distributed memory back-level edge detection algorithm is given in Figure 4.

Finally, we note that the distributed computation is terminated if no workstation has a state in its next level queue after the synchronization. This can be detected by exchanging several additional messages among the workstations when the workstations are synchronized. However, we do not give these in the pseudo-code but suppose the procedure `Synchronize()` to perform this check and to set the variable `finished` properly.

Lemma 3.6. *Let $G = (V, E)$ be a graph with an initial state s . The algorithm given in Figure 4 if executed on the graph computes the correct distances $d(u)$ for all the reachable states $u \in V$.*

Proof: Let us denote by $\hat{d}(u)$ the real distance of a state u from the initial state and by $d(u)$ the distance as computed by the algorithm. Note that once the algorithm assigns to a state u some value $d(u)$, then $d(u)$ is never changed. This is because u gets into the set `Visited` which prevents u from being reexpanded and thus $d(u)$ from being changed. Also note that the variable `Level` changes its value monotonically.

In the following we use the mathematical induction with respect to the real distance of a state from the initial state to show that each state u is assigned the correct distances $d(u)$. As the base of the induction let us consider the initial state s . We can easily conclude from the pseudo-code that $d(s) = 0$ when the algorithm finishes and so that $d(s) = \hat{d}(s)$. Now let us suppose that for all states u with $\hat{d}(u) \leq n$ we have $d(u) = \hat{d}(u)$ and let us consider a state v such that $\hat{d}(v) = n + 1$.

At first we will show that the state v cannot be expanded before the variable `Level` reaches the value of $n + 1$. If it expanded before then there would be an immediate predecessor t of the state v such that $d(t) < (n + 1) - 1 = n = \hat{d}(t)$ which is in contradiction to the induction assumption. Hence, we have that v is at first expanded when `Level` reaches $n + 1$. Now we will show that $d(v) = n + 1$.

Since $\hat{d}(v) = n + 1$ we know that there is at least one immediate predecessors p of the state v that have $\hat{d}(p) = n$. From the induction assumption we know that $d(p) = n$ and thus we can conclude that there was at least one edge (p, v) inserted in the next level queue on the workstation owning v when the variable `Level` equaled to n . Since we know that the state v could not be expanded earlier we can conclude that it will be expanded after the next synchronization, i.e. when the variable `Level` equals to $n + 1$. From which we have that $d(v) = n + 1$. ■

Lemma 3.7. *Let G be a graph with an initial state. If the algorithm given in Figure 4 is executed on the graph then it reports exactly all the back-level edges of the graph.*

Proof: We can conclude from the pseudo-code of the algorithm and from Lemma 3.6 that when an edge (p, v) is extracted from the current level queue then $d(p) = \text{Level} - 1$. From the pseudo-code we can also see that a back-level edge is reported only if $d(v) < \text{Level}$. Together we have that if the edge is reported then $d(p) > d(v) - 1$ which satisfy the definition of the back-level edge.

It remains to show that if (p, v) is a back-level edge then it is reported by the algorithm. At first we need to show that the state v has already been expanded when the edge is dequeued from the current level queue. But this is obvious, because in the other case the state v would be assigned distance $d(v) = d(p) + 1$, hence $d(v) > d(p)$ which is due to Lemma 3.6 in the contradiction to the assumption that (p, v) is a back-level edge.

When the edge (p, v) is extracted from the current level queue we know that $d(p) = \text{Level} - 1$. Since we have that $d(v) \leq d(p)$, we also have that $d(v) < \text{Level}$. The state v has already been explored before the edge (p, v) is dequeued and thus it is present in the set `Visited`. These facts are sufficient to conclude from the pseudo-code that the edge (p, v) will be reported by the algorithm. ■

Lemma 3.8. *Let G be a finite graph with an initial state. If the algorithm given in Figure 4 is executed on the graph then it finishes eventually.*

Proof: We can see from the pseudo-code of the algorithm that it stores all the states that has already been expanded to the set $Visited$. Once a state is inserted to this set it is never expanded again. Hence, from the finiteness of the graph we can be sure that there are only finitely many states enqueued to a queue NLQ, thus there are only finitely many states that are dequeued from a queue CLQ from which we can conclude that the inner **while** cycle finishes everytime. Since the contents of the queues NLQ and CLQ is swapped always after the queue CLQ is emptied and a state may be inserted in a queue NLQ only finitely many times it is obvious that there is an iteration of the outer **while** cycle after which all the local queues NLQ empty. This is detected by the `Synchronized()` procedure and the outer **while** cycle is terminated. Hence, the algorithm finishes eventually. ■

4 Cycle detection

To complete the distributed memory algorithm we have to describe how we employ the back-level edges for cycle detection. In general, a sufficient technique to decide about the presence of a cycle in a graph is to check each state of the graph for its *self-reachability*, i.e. the reachability of the state from itself. This can be done by performing as many search procedures as there are the states in the graph (each procedure checking one state). Obviously, such an approach is too expensive to be used for cycle detection in practice. However, the idea of testing self-reachability is not bad. (It is used by the standard Nested DFS algorithm as well.) The main problem of this approach is that there have to be too many search procedures executed in order to detect a cycle. In the algorithm suggested in this section we build on the approach of self-reachability tests but we fairly limit the number of the search procedures that are needed to reveal a cycle in the graph.

Actually, to reveal a cycle it is enough to perform self-reachability test for only one state on a cycle. We showed in Lemma 3.3 that every cycle contains at least one back-level edge, hence if the self-reachability test is performed for all the states that are source states of at least one back-level edge, then at least one state from a cycle is checked and so there is no cycle that could be missed.

In Lemma 3.4 we showed another interesting fact we use for cycle detection. The Lemma says that any cycle contains such a back-level edge that all the states on the cycle have the distance less or equal to the distance of the source state of the back-level

edge. Hence, when testing a source state of some back-level edge for self-reachability, we can restrict the search to explore only the states that have distances less or equal to the distance of the source state of the back-level edge. If the tested source state is on a cycle but it is not in the maximal depth of the cycle then the cycle is not detected. However, in such a case the cycle is detected by the search procedure that checks for self-reachability the source state of a back-level edge that lies in the maximal depth of the cycle.

Now we can describe the basic version of the *distributed memory BFS based cycle detection algorithm*. The algorithm performs alternately two phases. The task of the first phase (henceforth called *primary*) is to reveal all back-level edges while the task of the second phase (henceforth called *nested*) is to test each discovered back-level edge for being a part of a cycle. The primary phase employs the level synchronized breadth first search of the graph as it was described in the previous subsection. As soon as the primary search synchronizes after expanding all states on a current level the primary phase of the algorithm is interrupted and the nested phase begins. In particular, self-reachability testing procedures (henceforth *nested procedures*) are initiated for source states of all the back-level edges that were discovered during the exploration of the current level. Let us call the back-level edges discovered during the exploration of the current level *current back-level edges*. Note that the source states of current back-level edges are on the level that precedes the current level, hence their distance from the initial state of the graph is $\text{Level} - 1$. The goal of each nested procedure is to hit the source state of a back-level edge from which it was initiated (so called *target*). If at least one nested procedure succeeds then the presence of a cycle is ensured and the algorithm is terminated. Otherwise, the nested phase of the algorithm finishes without discovering a cycle and the primary search procedure continues with the exploration of the next level. Since there are many nested procedures performed concurrently, the target of each nested procedure cannot be maintained in a single variable (as in the case of the Nested DFS algorithm) but has to be propagated by the nested procedures. Note that contrary to the standard breadth first search states are not marked as visited by nested procedures and thus may be reexpanded many times.

The pseudo-code of the BFS based cycle detection algorithm is given in Figure 5. The algorithm consists of two procedures, both of them are performed on all participating workstations. The procedure `BFS-BASED-CYCLE-DETECTION` corresponds to the primary phase of the algorithm, hence it performs the back-level edge detection. Obvi-


```

1 proc BFS-BASED-CYCLE-DETECTION(WorkstationId)
2   CLQ := NLQ := BLQ := Visited :=  $\emptyset$ ; Level := 0
3   initstate :=  $F_{init}()$ ; finished := false
4   if (WorkstationId = Owner(initstate))
5     then enqueue(NLQ, ( $-$ , initstate))
6   fi
7   while ( $\neg$ finished) do
8     swap(NLQ, CLQ)
9     while (CLQ  $\neq \emptyset$ ) do
10      (p, v) := dequeue(CLQ)
11      if (v  $\notin$  Visited)
12        then Visited := Visited  $\cup$  {v}; d(v) := Level
13        foreach t  $\in F_{succs}(v)$  do
14          if (Owner(t)  $\neq$  WorkstationId)
15            then SendTo(Owner(t), enqueue(NLQ, (v, t)))
16            else enqueue(NLQ, (v, t))
17          fi
18        od
19        else if (d(v) < Level)
20          then enqueue(BLQ, (p, v))
21        fi
22      fi
23    od
24    Synchronize(finished := ( $\text{all } NLQ = \emptyset$ ))
25    CHECK-BL-EDGES(WorkstationId)
26    Level = Level + 1
27  od
28 end

```

```

1 proc CHECK-BL-EDGES(WorkstationId)
2   while ( $\neg$ Synchronize()  $\vee$  BLQ  $\neq$   $\emptyset$ ) do
3     if (BLQ  $\neq$   $\emptyset$ )
4       then (target, q) := dequeue(BLQ)
5         if d(q) < Level
6           then if (q = target) then Report("Cycle-Detected")
7             fi
8           foreach t  $\in$  Fsuccs(q) do
9             if (Owner(t)  $\neq$  WorkstationId)
10              then SendTo(Owner(t), enqueue(BLQ, (target, t)))
11              else enqueue(BLQ, (target, t))
12            fi
13          od
14        fi
15      fi
16    od
17 end

```

Figure 5: BFS based cycle detection algorithm

ously, it is quite similar to the distributed memory back-level edge detection algorithm given in Figure 4. The only difference is that instead of reporting the back-level edges the procedure stores them in a local queue BLQ. Everytime a level of the graph is completely explored the workstations synchronize. Before proceeding to the next level all the local procedures CHECK-BL-EDGES are initiated.

The procedures CHECK-BL-EDGES perform the nested phase of the algorithm, i.e. they implement the nested searches for all the back-level edge source states revealed. In general, each individual nested procedure needs to know two states to be able to perform its task. These are the state to be expanded (q) and the state to be hit ($target$). When a back-level edge that commences a new nested procedure is dequeued from the queue BLQ the source state of the edge is considered to be the target of the nested procedure while the destination state of the edge is the first state to be expanded.

Everytime a pair is dequeued from the queue the procedure CHECK-BL-EDGES proceeds as follows. At first it checks whether q (i.e. the state to be expanded) is not below the depth of the $target$. Note that in the distributed memory environment $d(target)$ may be generally inaccessible, hence the value of variable $Level$ is used instead in the test (recall that $d(target) = Level - 1$). If the test succeeds the state q is checked for being the target itself. If it is so the presence of a cycle is reported, otherwise the state q is expanded and its successors (combined with the target to make the proper pairs) are inserted in the appropriate queues in accordance with the partition function. Note that all the procedures CHECK-BL-EDGES terminate synchronized after all the pairs enqueued in all local queues BLQ are processed.

In the following lemmas we prove the correctness of the algorithm given in Figure 5. In particular we show that the algorithm reports cycle if and only if there is a cycle in the examined graph and that the algorithm terminates. Note that if we speak about a procedure BFS-BASED-CYCLE-DETECTION or CHECK-BL-EDGES we actually mean all the local procedures performed on all participating workstations. Further we assume for the needs of the proofs that the procedure Synchronize correctly implements a distributed termination detection algorithm.

Lemma 4.1. *If there is a cycle in the examined graph then the algorithm finishes by reporting the presence of a cycle.*

Proof: The key observation we employ in the following is that all the queues BLQ are “FI-FO” structure. Thus if a pair is inserted in the queue and the algorithm is not terminated then the pair is also dequeued from the queue (processing of each pair is finite).

Suppose there is a cycle in the graph. Let us consider only the cycles that have the minimal $\text{maxdepth}(\pi)$ among all the cycles in the graph. Let k be the distance, i.e. $k = \text{maxdepth}(\pi)$. From Lemma 3.4 we can conclude that each such a cycle contains at least one back-level edge that emanates from the level_k . Furthermore, from Lemma 3.7 and the pseudo-code of the algorithm we can conclude that all these back-level edges are enqueued in appropriate queue BLQ when Level equals to $k + 1$ and the procedures BFS-BASED-CYCLE-DETECTION synchronize.

Let $\pi = p_0, \dots, p_n$ be one of the cycles, hence $p_0 = p_n$ and p_0 is the source state of the corresponding maximal depth back-level edge. We will show that if the algorithm is not terminated then each pair (p_0, p_i) for $1 \leq i \leq n$ is inserted in a queue BLQ on some workstation at least once. To show this we employ the mathematical induction. As for the base case we have to show that (p_0, p_1) is inserted in the queue BLQ on workstation owning the state p_1 , but this is done by the procedure BFS-BASED-CYCLE-DETECTION because (p_0, p_1) is the back-level edge. So let us suppose that all the pairs (p_0, p_i) for $1 \leq i \leq j < n$ were inserted in a queue BLQ at least once on at least one workstation. We will show that then also the pair (p_0, p_{j+1}) had to be inserted in the queue on workstation owning p_{j+1} . However, this is clear. The pair (p_0, p_j) had to be dequeued from the queue on the workstation owning p_j and because $d(p_j) \leq k < \text{Level}$, the state p_j had to be expanded. We can see from the pseudo-code of the procedure CHECK-BL-EDGES that all the pairs for immediate successors of the state p_j are inserted in an appropriate queues BLQ, hence namely the pair (p_0, p_{j+1}) . This completes the induction.

The algorithm does not finish until all pairs from queues BLQ are processed or a Report function is called. If the report function is called then we are done because the presence of a cycle is reported. Suppose the algorithm does not report the presence of a cycle (i.e. the algorithm finishes without reporting the presence of a cycle or the algorithm cycles forever). But from the arguments above it follows that the pair $(p_0, p_n) = (p_0, p_0)$ is dequeued and processed by the procedure. In such a situation, we have that $d(p_0) = k$, $k < \text{Level}$, and $p_0 = p_0$ from which we can conclude (following the pseudo-code) that the Report function is called. Hence we get a contradiction and the lemma is proved. ■

Lemma 4.2. *If the algorithm reports the presence of a cycle then there is at least one cycle in the examined graph.*

Proof: It is clear from the pseudo-code that if the algorithm finished and reported the presence of a cycle then one of the procedures CHECK-BL-EDGES had to dequeue a pair (p, p) from the corresponding queue BLQ. A pair (p, p) could be inserted in the queue for two different reasons. If it was inserted in the queue by the procedure BFS-BASED-CYCLE-DETECTION then (p, p) is a back-level edge in the examined graph. In such a case the presence of a cycle in the graph is proved.

Let us now suppose that (p, p) is not a back-level edge. This means that the pair was inserted in the queue by a procedure CHECK-BL-EDGES when it processed an immediate predecessor of the state p . Let us denote this predecessor by p_1 . However, this means that there was also a pair (p, p_1) that was dequeued by some procedure CHECK-BL-EDGES from its queue BLQ. Again we can distinguish two cases: (p, p_1) is a back-level edge or it was inserted in the queue by a procedure CHECK-BL-EDGES when it processed an immediate predecessor p_2 of the state p_1 , i.e. when it processed the pair (p, p_2) . We can continue in this manner and gradually generate immediate predecessors p_3, p_4, \dots . However, all the procedures CHECK-BL-EDGES could have enqueued only finitely many pairs in the queues BLQ since the beginning of the computation, hence we have to reach such an immediate predecessor p_n that was enqueued in a queue as a pair (p, p_n) by a procedure BFS-BASED-CYCLE-DETECTION. Now it remains to realize that the constructed path $p, p_n, p_{n-1}, \dots, p_2, p_1, p$ supports the presence of a cycle in the examined graph. ■

Lemma 4.3. *If the examined graph is finite then the algorithm finishes eventually.*

Proof: If there is a cycle in the examined graph then the algorithm finishes according to the Lemma 4.1. It remains to show that the algorithm terminates also if there is no cycle in the graph.

Let us suppose that any procedure CHECK-BL-EDGES initiated from any procedure BFS-BASED-CYCLE-DETECTION finishes eventually. In such a case we can conclude that the algorithm finishes eventually from Lemma 3.8. Thus we only need to prove that if there is no cycle in the graph then any procedure CHECK-BL-EDGES finishes.

Since the procedure CHECK-BL-EDGES is called from the procedure BFS-BASED-CYCLE-DETECTION we know that there are only finitely many pairs enqueued in the queue BLQ when it is initiated. It is obvious that processing of a single pair finishes (each state has only finitely many immediate successors), hence we only need to show that a single pair is inserted in a queue BLQ only finitely many times. If we analyze the

pseudo-code of the procedure we can see that a pair (u, v) can be inserted in a queue BLQ only if there is a path from u to v . Moreover, if a pair (u, v) is inserted to the queue due to a path from u to v it is never inserted to the queue due to the same path again. Thus a pair (u, v) can be inserted in a queue BLQ at most as many times as there are paths from u to v . Since we know that there are no cycles in the graph, we are done because in such a case there are only finitely many paths between any two states of the graph, hence there are only finitely many pairs inserted in the queue BLQ. Obviously, these are all processed eventually which means that the procedure finishes. ■

5 Improving the algorithm

When checking a state u for its self-reachability the corresponding nested procedure revisits every state v that is reachable from the state u but is not below the state u as many times as there are different paths leading from u to v . Obviously, in such a case some states may be revisited many times by the nested procedure. Obviously, this revisiting makes the algorithm quite slow. Therefore, we suggest a modification of the procedure CHECK-BL-EDGES that decreases the revisiting factor fairly.

At first we would like to eliminate the revisits of states made by a single nested procedure. This is possible in the standard search approaches like breadth first search or depth first search where the states that has already been expanded can be marked as visited. So when a newly generated state should be expanded it is at first checked whether it is marked as visited. If it is the case then the state has been expanded previously and so it need not be expanded again. This approach is used for example for the nested procedure of the standard Nested DFS algorithm where it suffices to enrich each state with a single bit to distinguish whether it has been expanded in the nested procedure or not. However, in our algorithm there are many nested procedures performed in parallel each having a different goal. Hence, if we wanted to use this approach, we would have to maintain for each state as many differentiating bits as there are running nested procedures. Obviously, this would be quite memory demanding since the number of running nested procedures cannot be reasonably bounded. As the memory is the primary resource that is extended by employing the distributed memory environment, we use slightly different technique to reduce the revisiting factor.

The technique we suggest has reasonable additional space requirements. However, contrary to the approach described above it does not prevent revisiting completely. Its

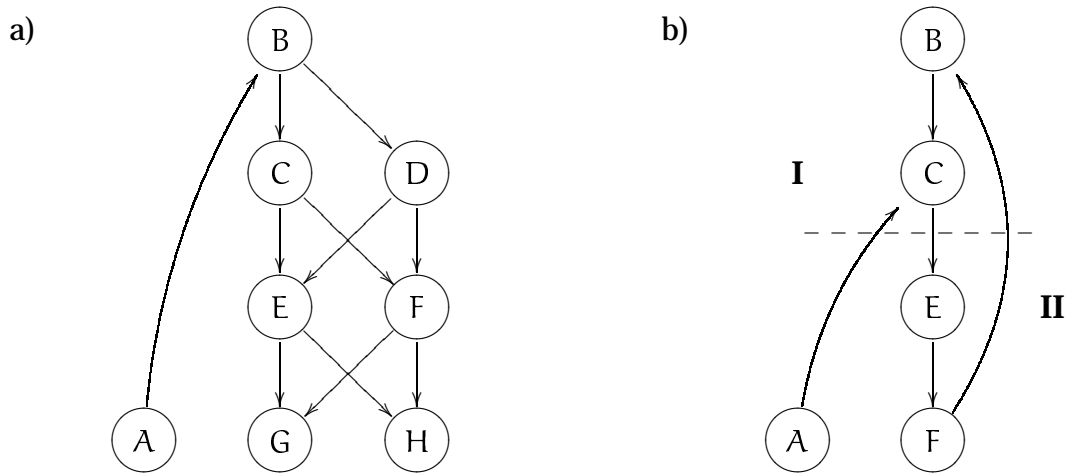


Figure 6: Repeated state expansions

basic idea is to replace the bundle of the distinguishing bits associated to each state by a single nested procedure identifier. Note that each nested procedures can be easily identified by its target. In the identifier stored at the state we cannot obviously remember all the nested procedures that has expanded a state but only one. In particular, we store the identifier of the last nested procedure that expanded the state. So if a nested procedure reaches a state such that the identifier stored at the state is the same as the identifier of the procedure, then the procedure omits the expansion of the state. Hence, a nested procedure is allowed to expand a state it has expanded previously only if the state was expanded by another nested procedure in the meantime.

To illustrate the power of the presented optimization let us consider the graph a) in Figure 6. It can be easily seen that the nested procedure initiated for the back-level edge (A, B) will visit and expand the state G four times (along all the paths from B to G) if the optimization is not considered. However, if it is considered then the state G will be visited twice (from the states E and F) and expanded only once (when being visited for the first time).

It is clear that the presented optimization is too weak to prevent revisits of such a states that are alternately expanded by two or more nested procedures. In general, we can say that the less nested procedures are performed in parallel the more powerfull the technique is. Nevertheless, we would like also to reduce the state revisits that are made by different nested procedures. To this end we suggest another optimization.

The idea of the optimization is quite simple: instead of storing the identifier of the last nested procedure we store the identifier of the greatest nested procedure that has

ever expanded the state. However, to be able to recognize the greatest nested procedure identifier we need at first to define their *ordering*. We define it as follows (note that we use square brackets to distinguish a nested procedure identifier from a single state):

$$[A] > [B] \iff d(A) > d(B) \vee (d(A) = d(B) \wedge A > B)$$

We can see that to order the identifiers of nested procedures that originate on different levels we exploit the distances from the initial state as computed during the primary phase of the algorithm. However, to order the identifiers of nested procedure originating on the same level we need to have an ordering on the states. Since there are no restriction on this ordering, it can be defined easily. Technically, to order the states we use the state corresponding bit vectors.

Having the ordering a nested procedure is allowed to expand (and thus proceed through) a state either if the identifier of the nested procedure is greater than the identifier stored at the state or the state has not been expanded by a nested procedure at all (the identifier stored at the state is undefined). Such an approach reduces the revisits of states significantly, however, it breaks the cycle detection capability of the algorithm as the nested procedure that should reveal a cycle may be stopped before it reaches its target. This situation is illustrated on the graph b) in Figure 6. Let us suppose an ordering of state in which $A > F$. If the nested procedure $[A]$ (the one corresponding to the back-level edge (A, C) and thus having A as its target) reaches the state C before the nested procedure $[F]$ then the nested procedure $[F]$ is stopped at the the state C because the identifier stored at the state C is $[A]$ and $[A] > [F]$. The nested procedure $[A]$ continues through the states E, F, B at which identifiers $[\perp], [F], [F]$, respectively, are stored. And stops at the state C at which identifier $[A]$ is stored (obviously $[A] \not> [A]$). Hence, the cycle F, B, C, E, F remains undetected.

To address this problem we further change the identifier of each nested procedure to be a couple $[T, n]$, where T is the target of the procedure and n is the number of *current* back-level edges the procedure has passed through. Note that the identification of a nested procedure is dynamically modified as the procedure proceeds and passes current back-level edges. Since we changed the identifiers of the nested procedures, we need also to redefine the ordering. We do it as follows:

$$[A, x] > [B, y] \iff [A] > [B] \vee ([A] = [B] \wedge x > y).$$

Now the algorithm can detect a cycle in *two* different ways. A cycle can be detected either by a nested procedure that hits its target or by a nested procedure whose number

of passed current back-level edges exceeds the total number of current back-level edges. Note that the latter case is possible only if there is a back-level edge that is a part of a cycle.

To exemplify this optimization let us consider again the graph b) in Figure 6 and an ordering of states such that $A > F$. There are two back-level edges emanating from different source states and so there are two nested procedures initiated, namely $[A, 0]$ and $[F, 0]$. When they reach the state C they have both passed one current back-level edge, hence they are identified as $[A, 1]$ and $[F, 1]$. If the nested procedure $[A, 1]$ reaches the state C before the procedure $[F, 1]$, the $[F, 1]$ procedure is stopped at the state C. In such a case the procedure $[A, 1]$ continues in the exploration of the graph. When it visits the state C for the second time, its identification is $[A, 2]$, which is obviously greater than $[A, 1]$, and so the procedure is not stopped but continues through the state C again. It goes through the states E and F and increases its number of passed current back-level edges when it reaches the state B. At that moment the number of passed current back-level edges by the procedure $[A, 3]$ (originally initiated as $[A, 0]$ at the state A) exceeds the total number of current back-level edges (which is two) and the cycle is detected. Note that the cycle was detected by a nested procedure that was initiated at a state that is not a part of the cycle.

Finally, we suggest seemingly strange modification of the procedure CHECK-BL-EDGES. In particular, we suggest not to use the “FI-FO” structure BLQ for the nested procedures, but to use a different structure BBLQ into which items from the queue BLQ are copied in a way described below. This modification forces the local processing of a nested procedure to search the graph in a depth first manner, however the remote states are standardly enqueued to the remote queues. In the case there is a cycle in the examined graph this optimization should help a nested procedure to discover a cycle faster than if it used strictly breadth first search. Note that this modification cannot be applied directly to the procedure CHECK-BL-EDGES without the optimizations suggested above because it may result in a non-terminating procedure.

The pseudo-code of the improved version of the procedure CHECK-BL-EDGES bears the name CHECK-BL-EDGES2 and is given in Figure 7. Contrary to the original procedure the new version requires in addition to the workstation identification (*WorkstationId*) also the number of current back-level edges (*nmbrbl*). The new procedure proceeds as follows. At first it dequeues all the pairs from the queue BLQ, change them into quadruples and enqueue them into the queue BBLQ. Each quadru-

```

1 proc CHECK-BL-EDGES2 (WorkstationId, nمبرbl)
2   BBLQ :=  $\emptyset$ ;
3   while (BLQ  $\neq \emptyset$ ) do
4     (target, q) := dequeue(BLQ)
5     enqueue(BBLQ, (q, Level-1, target, 0))
6   od
7   while ( $\neg$ Synchronized()  $\vee$  BBLQ  $\neq \emptyset$ ) do
8     if (BBLQ  $\neq \emptyset$ )
9       then (q, prelevel, target, bl) := dequeue(BBLQ);
10      if d(q) < Level
11        then if (q = target) then Report("Cycle-Detected") fi
12        if (prelevel = Level-1)
13          then bl := bl + 1
14          if (bl > nمبرbl)
15            then Report("Cycle-Detected")
16          fi
17        fi
18        if ((Level-1 > q.level)  $\vee$  (Level-1 = q.level  $\wedge$ 
19          (target > q.target  $\vee$  (target = q.target  $\wedge$  bl > q.bl))))
20          then q.target := target
21          q.bl := bl; q.level := Level-1
22          foreach t  $\in$   $F_{\text{succs}}(q)$  do
23            if (Owner(t)  $\neq$  WorkstationId)
24              then SendTo(Owner(t),
25                enqueue(BBLQ, (t, d(q), target, bl)))
26              else push(BBLQ, (t, d(q), target, bl))
27            fi
28          od
29        fi
30      fi
31    fi
32  od
33 end

```

Figure 7: Improved procedure CHECK-BL-EDGES

ple consists of a state to be explored (q), the distance of the immediate predecessor of the state to be explored ($prelevel$), the target of the nested procedure ($target$) and the number of passed current back-level edges (bl). The variable $prelevel$ is used to detect whether the nested procedure passes through a current back-level edge. The meaning of other variables should be already clear. When the queue BBLQ is ready, the procedure CHECK-BL-EDGES2 proceeds similarly to the procedure CHECK-BL-EDGES. It dequeues a quadruple from the queue BBLQ and proceeds the state to be expanded only if it is not below the target. If q equals to the target a cycle can be reported (the first way of cycle detection). Then the variable $prelevel$ is checked whether it keeps the value of $Level - 1$. If it is so then it means that the edge due to which the currently processed quadruple was inserted in the queue is a current back-level edge. (We can conclude this from the fact that the source state of the edge has the same distance from the initial state as the target and the fact that the states below the target are not processed.) In such a case the back-level edge counter is increased and then it is checked whether it overflows the total number of current back-level edges on the examined level. If it does, a cycle is reported (the second way of cycle detection). In the other case it is checked whether the identification of the nested procedure is great enough to allow the procedure to expand the state to be expanded. Note that $q.target$, $q.bl$ are the values of $target$ and bl of the identifier stored at the state q . The value $q.level$ is used to remember the distance of the target of last nested procedure that expanded the state. Initially, the value of $q.level$ equals to -1 . Hence, if a state is visited for the first time its expansion is guaranteed. If the nested procedure is allowed to expand the state, the values of $q.target$, $q.bl$, and $q.level$ are properly updated, and the immediate successors of the state q are generated. Local successors are pushed to the front of the queue BBLQ, while the remote ones are enqueued to the corresponding queues on remote workstations. Note that the distance of the state q ($d(q)$) is used in the produced quadruples in order to become the value of the variable $prelevel$ when a quadruple is dequeued.

It remains to say how to modify the BFS-BASED-CYCLE-DETECTION algorithm given in Figure 5 to employ the new procedure CHECK-BL-EDGES2. The modification is quite easy. Actually it suffice to call the procedure CHECK-BL-EDGES2 instead of the procedure CHECK-BL-EDGES. Nevertheless, to be able to do so we need to know the number of current back-level edges ($nmbrrbl$). We can get the number by counting up the sizes of all queues BLQ. To do so we have to can insert a new call of the procedure `Synchronize()` just before the place where the procedure CHECK-BL-EDGES2 is called.

We can then employ this call of the procedure to get the value of nmbrbl in a similar way it is used to count the sizes of the queues NLQ in order to detect the end of exploration of the graph.

Now we state and prove lemmas giving the correctness of the *improved* BFS-BASED-CYCLE-DETECTION algorithm as given in Figures 5 and 7. At first we prove an auxiliary Lemma.

Lemma 5.1. *Let us assume that the procedures CHECK-BL-EDGES2 were initiated after the procedures BFS-BASED-CYCLE-DETECTION synchronized on such a level that there is no cycle in the graph such that the distances of all states on the cycle are less than $\text{Level} - 1$. Further, let $q_0 = (q, \text{prelevel}, \text{target}, \text{bl})$ be a quadruple that is dequeued from the queue BBLQ by the procedure CHECK-BL-EDGES2 running on a participating workstation before the procedures CHECK-BL-EDGES2 synchronize and the next level of the graph is explored. Let q_1, q_2, \dots, q_n be the contents of the queue BBLQ immediately after a quadruple q_0 was dequeued. Then either the algorithm finishes by reporting the presence of a cycle before the quadruple q_1 is dequeued from the queue BBLQ or the procedure CHECK-BL-EDGES2 dequeues the quadruple q_1 eventually.*

Proof: It is obvious from the pseudo-code that the algorithm cannot terminate reporting no presence of a cycle without dequeuing all the quadruples from the queue BBLQ on the workstation. Hence, we only need to show that the procedure CHECK-BL-EDGES2 cannot cycle forever without dequeuing the quadruple q_1 . Let us suppose the contrary, i.e. the procedure cycles forever without dequeuing the quadruple q_1 from the queue BBLQ.

We know that processing of each quadruple is finite (this is because any state has only finitely many immediate successors). However, when a quadruple q is completely processed and the next quadruple is going to be dequeued from the queue BBLQ it need not be the quadruple that was just behind the quadruple q when the quadruple q was dequeued. This is because some quadruples may be pushed to the front of the queue when local successors of the state from the quadruple q were generated.

Let us denote by R the set of all quadruples that occur in the part of the queue BBLQ that precedes the quadruple q_1 during the execution of the algorithm. Note that there are only finitely many different quadruples that may occur in R . This is because we have only finitely many states in the graph (this limits q and target). The graph has only finitely many levels (this limits prelevel) and no presence of a cycle is reported (hence $\text{bl} < \text{nmbrbl}$).

Since we know that processing of any quadruple is finite, the algorithm must dequeue quadruples from the queue BBLQ continuously. We have only finitely many different quadruples, hence there is at least one quadruple that is dequeued from the queue BBLQ infinitely many times. Let I be the set of all quadruples that are inserted to and dequeued from the queue BBLQ infinitely many times. Obviously, $I \subseteq R$.

Let $p = (q', \text{prelevel}', \text{target}', \text{bl}')$ be a quadruple from I . Let us call descendants of a quadruple p all the quadruples that are generated from the quadruple p or from descendants of the quadruple p . We show that any quadruple that is (not necessarily immediate) descendant of the quadruple p differs from p . This is because a descendant of the quadruple p has to contain a state that is reachable from the state q' . So to be the same as p it must be reached along a path from q' to q' . Such a path is a cycle and thus according to our assumptions it contains at least one state that has the distance from the initial state equal to $\text{Level} - 1$. However, when the procedure CHECK-BL-EDGES2 processes a quadruple containing successors of this state, it obviously increases the value of bl . Note that the value of bl is never decreased, hence any descendant of this quadruple must differ from the quadruple p .

Now we can define a partial ordering on I . If $a, b \in I$ and b is a (not necessarily immediate) descendant of a , then $a < b$. Obviously, it cannot be that $b < a$ at the same time, otherwise a would be its own descendant which is not possible as we showed. Hence, we can choose a minimal member in I . Let us denote this minimal member by r . We know that r was inserted in R infinitely many times, but only as descendant of finitely many quadruples that were processed only finitely many times. Obviously, this is not possible and hence we get the needed contradiction. So if the algorithm is not terminated by reporting the presence of a cycle, the procedure CHECK-BL-EDGES2 dequeues the quadruple q_1 eventually. ■

Lemma 5.2. *If there is a cycle in the examined graph then the algorithm will finish by reporting the presence of a cycle eventually.*

Proof: Let us consider all the cycles π_1, \dots, π_m such that they have the minimal maximal depth of a state on the cycle among all the cycles in the graph. Let k be the maximal distance of a state in such a cycle, i.e. $k = \text{maxdepth}(\pi_i)$ for some $1 \leq i \leq m$. From Lemma 3.4 we can conclude that each cycle contains at least one back-level edge that emanates from the level_k . Furthermore, from Lemma 3.7 and the pseudo-code of the algorithm we can conclude that all these back-level edges are enqueued in appro-

appropriate queue BLQ when Level equals to $k + 1$ and the procedures BFS-BASED-CYCLE-DETECTION synchronize. Let n_{mbrbl} be the number of back-level edges enqueued in all queues BLQ after the synchronization. Now let us suppose that the algorithm did not finish or finished without reporting the presence of a cycle.

Let $\pi_x = p_0, \dots, p_n$ be one of the cycles with minimal maximal depth of a state on the cycle and let $p_0 = p_n$ and p_0 be the source state of the corresponding maximal depth back-level edge. Furthermore, let us consider the greatest values $target$ and bl (i.e. the greatest nested procedure identification) stored at a state q of the cycle π_x during the execution of the procedure CHECK-BL-EDGES2 when $Level = k + 1$. Let us denote this greatest nested procedure identification by $[T, b]$. Note that if the procedure finishes then the maximum obviously exists, if it cycles forever then from the pseudocode we can see that the variable bl cannot overflow the value of n_{mbrbl} , otherwise a cycle is reported and the algorithm terminated. Hence, the value of bl is limited by n_{mbrbl} . Since there are only finitely many states in the examined graph the value of $target$ can be limited too and so the maximum exists.

Since (p_0, p_1) is a back-level edge that is enqueued to the queue BLQ on the workstation owning the state p_1 when the procedure CHECK-BL-EDGES2 is initiated on the workstation we can easily conclude that there is a corresponding quadruple $(p_1, Level-1, p_0, 0)$ enqueued to the queue BBLQ. By repeated application of Lemma 5.1 we know that either the algorithm terminated by reporting the presence of a cycle (which is not possible according to our assumptions) or the quadruple is dequeued eventually. In the latter case, we can be sure that the nested procedure identification greater or equal to $[p_0, 0]$ is stored to at least one state on the cycle π_x . Hence, we can suppose in the following that $T \geq p_0$.

At first let us suppose that $T = p_0$. In such a case there is a greatest m such that $1 \leq m \leq n$ and $[T, b]$ is stored at the state p_m . This means that the state p_m was expanded when $target = p_0$ and $bl = b$. If $m = n$ then $p_m = p_n = p_0 = target$. Hence, when the corresponding quadruple p_m was dequeued the algorithm must have been obviously terminated and cycle reported. However, this is in contradiction with our assumption and thus $m < n$. Since p_m was expanded, obviously all immediate successors of the state p_m had to be generated and corresponding quadruples inserted in appropriate queues. Again by repeated application of Lemma 5.1 we can be sure that either the algorithm terminated by reporting the presence of a cycle (which is not possible) or the

quadruple having the state p_{m+1} was dequeued eventually. Suppose now the situation when the quadruple containing the successor p_{m+1} was dequeued.

Since $d(p_{m+1}) \leq d(p_0) < \text{Level}$ we can be sure that the nested procedure was tested whether it is allowed to expand the state p_{m+1} . Obviously, the test had to succeed because $[T, b]$ is the maximal nested procedure identification stored at a state in the cycle. This means that the state p_{m+1} was expanded and so $[T, b]$ was stored at the state p_{m+1} . Which is obviously in contradiction to the assumption that m is the maximal index of a state on the cycle at which $[T, b]$ is stored. Therefore, the case that $T = p_0$ is not possible.

Now let us suppose that $T > p_0$. Let us denote by S the set of all the states in the cycle π_x that have the same distance from the initial state of the graph as the state p_0 , $S = \{p_{j_0}, \dots, p_{j_r}\}$. Further, let us consider a state $r \in S$ such that $[T, b]$ is stored at the state. At first we show that there is at least one such state in the set S . Let us choose a state p_m on the cycle π_x such that $[T, b]$ is stored at the state. If $p_m \in S$, we put $r = p_m$, otherwise there is a such a part of the cycle p_m, \dots, p_{m+h} that $p_m, \dots, p_{m+h-1} \notin S$ and $p_{m+h} \in S$. It must be the case that $[T, b]$ is stored at all the states p_m, \dots, p_{m+h} , otherwise there is a state p_z for $m \leq z < m+h$ in the sequence whose immediate successor have the stored nested procedure identification less than $[T, b]$. However, when the state p_z was expanded by the nested procedure, all the quadruples for its immediate successors were enqueued to the appropriate queues BBLQ. According to Lemma 5.1 either the algorithm finished and reported the presence of a cycle or the quadruple $(p_{z+1}, d(p_z), T, b)$ was dequeued eventually. Since the nested procedure identification stored at the state p_{z+1} has to be less than $[T, b]$ the state p_{z+1} is expanded by the nested procedure and so $[T, b]$ is stored at the state p_{z+1} which is contradiction to choice of the state p_z . Thus we know that all the states p_m, \dots, p_{m+h} have $[T, b]$ stored at them and so we can put $r = p_{m+h}$, thus $r \in S$.

Since $[T, b]$ is stored at the state r , we know that r was expanded by the nested procedure $[T, b]$ and so a quadruple $(t, d(r), T, b)$ was inserted in some queue BBLQ for the immediate successor t on the cycle π_x . By Lemma 5.1 we know that either the algorithm finished and reported the presence of a cycle, or the quadruple was dequeued eventually. In the latter case the variable prelevel was assigned the value $\text{Level} - 1$ and the variable bl the value b . We can see from the pseudo-code that in such a case the variable bl was increased before the state t was checked for being allowed to be expanded. In such a case either the algorithm was terminated by reporting the presence of a cycle (if $\text{bl} > \text{nmbrbl}$) or the test for being allowed to expand the state t succeeded ($[T, b]$ was

maximal). When the state t was expanded the identification $[T, b + 1]$ had to be stored at the state t . However, this is in contradiction to the maximality of $[T, b]$ and thus it is not possible.

So in all cases we get a contradiction with our assumptions. Hence, if there is a cycle in the examined graph, then the algorithm terminates by reporting the presence of a cycle. ■

Lemma 5.3. *If the algorithm reports the presence of a cycle then there is at least one cycle in the examined graph.*

Proof: Suppose that the quadruple dequeued in the iteration in which the presence of a cycle was reported is $(q, \text{prelevel}, \text{target}, \text{bl})$. At first we prove that if such a quadruple is dequeued from a queue BBLQ then there is a path from the state target to the state q in the examined graph. Such a quadruple can be inserted in the queue either due to the fact that (target, q) is a current back-level edge that was dequeued from the queue BLQ or due to the fact that there is an immediate predecessor p_1 of the state q that was expanded by a nested procedure. However, in the latter case there must be a quadruple with the state p_1 being its state to be expanded. Obviously, either (target, p_1) is a current back-level edge or there is an immediate predecessor p_2 of the state p_1 that was expanded previously. We can continue in this manner and gradually generate immediate predecessors p_3, p_4, \dots . However, all the procedures CHECK-BL-EDGES2 could have inserted only finitely many quadruples in the queues BBLQ since the beginning of the computation to the moment of dequeuing the quadruple $(q, \text{prelevel}, \text{target}, \text{bl})$, hence we have to reach such an immediate predecessor p_n that was enqueued in some queue due to a current back-level edge (target, p_n) . It is clear that the sequence $\pi = \text{target} = p_{n+1}, p_n, \dots, p_2, p_1, p_0 = q$ is a path from the state target to the state q .

We can see from the pseudo-code that there are two reasons for which the algorithm may report the presence of a cycle. In the case that $q = \text{target}$ the cycle is supported directly by the path π . In the other case ($\text{bl} = \text{nمبرbl} + 1$) we only need to realize that there are $\text{nمبرbl} + 1$ states in the path π that have the same distance from the initial state as the state target . Obviously, all the edges in the path π emanating from these states have to be current back-level edges (no state with greater distance is allowed to be processed). Since there are only nمبرbl current back-level edges there are at least two states p_j, p_k in the path π from which the same back-level edge emanates. Since

$p_j = p_k$, the path from the state p_j to the state p_k supports the presence of a cycle in the examined graph. ■

Lemma 5.4. *If the examined graph is finite then the algorithm terminates.*

Proof: The proof is basically the same as the proof of Lemma 3.8. So, if there is a cycle in the examined graph then according Lemma 5.2 the algorithm terminates by reporting the presence of a cycle. Thus we only show that the algorithm terminates if there is no cycle in the graph. Let us suppose that every procedure CHECK-BL-EDGES2 initiated from any procedure BFS-BASED-CYCLE-DETECTION finishes eventually. In such a case we can conclude that the algorithm finishes eventually from Lemma 3.8. Hence, we only need to prove that if there is no cycle in the graph then any procedure CHECK-BL-EDGES2 finishes eventually. We prove this by showing that each quadruple can be inserted in a queue BBLQ only finitely many times. Nevertheless, it can be easily followed from the pseudo-code that a quadruple $(q, \text{prelevel}, \text{target}, \text{bl})$ can be inserted in a queue BBLQ only if there is a path from the state target to the state q . Moreover, if a quadruple is inserted in the queue due to a path it is never inserted in the queue again due to the same path. Since we suppose that the graph contains no cycle, there are only finitely many paths (each being finite) between any two states of the graph. This gives us needed bound on the number of quadruple insertions in the queue. ■

6 Accepting cycle detection

Unless we are under special circumstances, we cannot directly use the algorithm BFS-BASED-CYCLE-DETECTION to detect the language emptiness of a Büchi automaton as needed in order to do the LTL model checking. This is because the algorithm cannot distinguish between accepting and non-accepting cycles. Nevertheless, if we exploit the verified property to decompose the product automaton graph into components, we can use the algorithm to perform a limited accepting cycle detection.

To be more precise the graph can be decomposed into components of the three following types: non-accepting, fully accepting, and partially accepting. A non-accepting component contains no accepting states, a fully accepting component contains accepting states only, and a partially accepting component contains both. If we want to detect accepting cycles only, we need not search for cycles in the non-accepting components

and may use the normal cycle detection in the fully accepting components. (Note that any cycle in a fully accepting component must be an accepting cycle.) Since the types of components are completely determined by the verified property, which is quite small in practice, we are able to precompute the decomposition of the product automaton graph easily in advance. Naturally, we can use the algorithm `BFS-BASED-CYCLE-DETECTION` for LTL model checking problem as it is if the examined product automaton graph is made of non-accepting and fully accepting components only.

Moreover, if the algorithm `BFS-BASED-CYCLE-DETECTION` is used to reveal accepting cycles in a product automaton graph, it can be further optimized. In particular, we need not search for back-level edges and test back-level edges in non-accepting components. Obviously, a back-level edge whose source state or destination states is in a non-accepting component cannot be a part of an accepting cycle.

Although in [4, 8] the authors argue that 95% of practically verified properties produce Büchi automata without partially accepting components, we found the capability of accepting cycle detection quite useful. Hence in the following we suggest an extension of the algorithm `BFS-BASED-CYCLE-DETECTION` that makes the algorithm capable of accepting cycle detection and thus makes it applicable to the full LTL model checking problem.

The basic idea behind detection of accepting cycles in partially accepting components is to prevent the algorithm from detecting a non-accepting cycle. For this purpose each nested procedure maintains an additional (*accepting*) bit to indicate that it has passed through an accepting state since its last pass through a current back-level edge. In particular, this accepting bit is set to true whenever the procedure reaches an accepting state and is set to false whenever the procedure passes a current back-level edge. The bit is set to false initially.

There are two ways the algorithm detects a cycle: by hitting the target of the procedure or by exceeding the number of current back-level edges. To prevent a nested procedure from exceeding the number of current back-level edges on a non-accepting cycle we modify the nested procedure so that it counts a current back-level edge only if the accepting bit is set to true. As regards the first way (hitting the target), we modify the nested procedure so that it reports a cycle only if it reaches the target state and the bit is set to true.

Note that on an accepting cycle there must be such a back-level edge that if any nested procedure repeatedly walks around the cycle, then it always reaches the source

state of the edge with the accepting bit set to true. However, the source state may be reachable from itself also along a path that does not contain any accepting bit. Hence, it may happen that the nested procedure reaches and stores its identification to the state along the non-accepting path first and so when the nested procedure reaches the state along the accepting path it is not allowed to expand the state. In such a case the accepting cycle may remain undetected. To prevent this we again extend a nested procedure identification. In addition to the target of the procedure and the counter of passed current back-level edges the new identifier contains also the accepting bit. If we define that $\text{true} > \text{false}$ we can modify the requested ordering on nested procedure identification obviously, i.e.

$$[A, x, b] > [B, y, c] \iff [A, x] > [B, y] \vee ([A, x] = [B, y] \wedge b > c)$$

We demonstrate the behavior of the modified nested procedure on the graphs a) and b) in Figure 8. In both cases the nested procedure for the back-level edge (A, C) arrives at the state C as a procedure that is identified by $[A, 0, \text{false}]$. This is because the accepting bit is initially set to false and the state A is not an accepting state. Thus the procedure does not increase its counter of passed back-level edges when it passes the edge (A, C) . Similarly, the nested procedure for the back-level edge (F, B) arrives at the state B as the procedure $[F, 0, \text{false}]$.

Let us first assume that either $A < F$ or the procedure $[F, 0, \text{false}]$ arrives at the state C before the procedure $[A, 0, \text{false}]$. In such a case the procedure $[F, 0, \text{false}]$ continues through the states C and E and hits its target (the state F). While in the graph b) the procedure reaches its target with the accepting bit set to true, in the graph a) it reaches its target with the bit set to false. Obviously, this can distinguish between accepting and non-accepting cycles.

Now let us assume that $A > F$ and the $[A, 0, \text{false}]$ arrives at the state C before the procedure $[F, 0, \text{false}]$ does. In such a case the procedure $[F, 0, \text{false}]$ is stopped when it arrives at the state C , while the procedure $[A, 0, \text{false}]$ continues in the search. In the case of the graph a) the procedure $[A, 0, \text{false}]$ passes the current back-level edge (F, B) without increasing its counter of passed current back-level edges because its accepting bit remains set to false. This means that the procedure does not change its identification and so it is stopped when it arrives at the state C for the second time. In the case of the graph b) the procedure $[A, 0, \text{false}]$ sets its accepting bit to true and becomes $[A, 0, \text{true}]$ when it passes the accepting state E which allows the procedure to increase

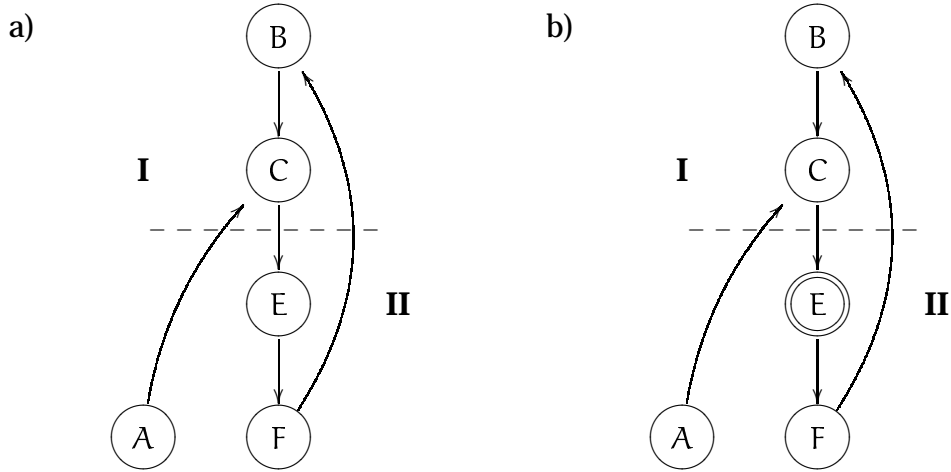


Figure 8: Accepting cycle detection

its counter of passed back-level edges when it passes the back-level edge (F, B) . Note that the accepting bit is reset to false when the counter is increased. The procedure then arrives at the state C for the second time being identified as $[A, 1, \text{false}]$. This means that the procedure is not stopped but it continues in the search. At the state E it sets its accepting bit to true changing its identifier to $[A, 1, \text{true}]$ and passes the back level-edge (F, B) changing its identifier to $[A, 2, \text{false}]$. Then it goes through the state C for the third time. At the state E it sets the accepting bit to true again $([A, 2, \text{true}])$ and after passing the back-level edge (F, B) it exceeds the number of current back-level edges. Hence, the existence of an accepting cycle is correctly detected.

Note that if the algorithm is modified to detect accepting cycles only, it may happen that there are non-accepting cycles that are completely above the currently tested level. However, these non-accepting cycles do not influence the cycle detection at all because they have neither accepting states nor current back-level edges. Finally, note there may be nested procedures that are initiated for a source state of a back-level edge which is a part of an accepting cycle but that cannot detect the accepting cycle by the first way of cycle detection (i.e. by hitting the target).

The procedure `CHECK-BL-EDGES` that is modified in order to detect accepting cycles only bears the name `CHECK-BL-EDGES3` and its pseudo-code is given in Figure 9. Naturally, the new variable `abit` correspond to the accepting bit.

Since the proof of the correctness of the algorithm `BFS-BASED-CYCLE-DETECTION` that employs the procedure `CHECK-BL-EDGES3` is quite similar to the proof of the corec-

```

1 proc CHECK-BL-EDGES3 (WorkstationId, nمبرbl)
3   BBLQ :=  $\emptyset$ ;
4   while (BLQ  $\neq \emptyset$ ) do
5     (target, q) := dequeue(BLQ)
6     enqueue(BBLQ, (q, Level-1, target, 0, false))
7   od
8   while ( $\neg$ Synchronize()  $\vee$  BBLQ  $\neq \emptyset$ ) do
9     if (BBLQ  $\neq \emptyset$ )
10      then (q, prelevel, target, bl, abit) := dequeue(BBLQ);
11      if d(q) < Level
12        then if (IsAccepting(q)) then abit := true fi
13        if (q = target  $\wedge$  abit = true)
14          then Report("Accepting Cycle Detected") fi
15        if (prelevel = Level-1  $\wedge$  abit = true)
16          then bl := bl + 1; abit := false
17          if (bl > nمبرbl)
18            then Report("Accepting Cycle Detected") fi
19          fi
20        if ((Level-1 > q.level)  $\vee$  (Level-1 = q.level  $\wedge$ 
21          (target > q.target  $\vee$  (target = q.target  $\wedge$  bl > q.bl)  $\vee$ 
22          (target = q.target  $\wedge$  bl = q.bl  $\wedge$  abit > q.abit)))
23          then q.target := target; q.bl := bl
24          q.level := Level-1; q.abit := abit
25          foreach t  $\in$   $F_{\text{succs}}(q)$  do
26            if (Owner(t)  $\neq$  WorkstationId)
27              then SendTo(Owner(t), enqueue(BBLQ,
28                (t, d(q), target, bl, abit)))
29              else push(BBLQ, (t, d(q), target, bl, abit))
30            fi
31          od
32        fi
33      fi fi od
34 end

```

Figure 9: Improved procedure CHECK-BL-EDGES with accepting cycle detection

ctness of the algorithm employing the procedure CHECK-BL-EDGES2, we state the necessary Lemmas and sketch their proofs only.

Lemma 6.1. *Let us assume that the procedures CHECK-BL-EDGES3 were initiated after the procedures BFS-BASED-CYCLE-DETECTION synchronized on such a level that there is no accepting cycle in the graph such that the distances of all states on the cycle are less than $\text{Level} - 1$. Further, let $q_0 = (q, \text{prelevel}, \text{target}, \text{bl}, \text{abit})$ be a quintuple that is dequeued from the queue BBLQ by the procedure CHECK-BL-EDGES3 running on a participating workstation before the procedures CHECK-BL-EDGES3 synchronize and the next level of the graph is explored. Let q_1, q_2, \dots, q_n be the contents of the queue BBLQ immediately after the quintuple q_0 was dequeued. Then either the algorithm finishes by reporting the presence of an accepting cycle before the quintuple q_1 is dequeued from the queue BBLQ or the procedure CHECK-BL-EDGES3 dequeues the quintuple q_1 eventually.*

Proof: We can see from the pseudo-code that the algorithm cannot terminate and report no presence of an accepting cycle without dequeuing all the quintuples from the queue BBLQ. Hence, we only need to show that the procedure CHECK-BL-EDGES3 cannot cycle forever without dequeuing the quintuple q_1 . Let us suppose the contrary in the following.

We know that processing of each quintuple is finite. However, when a quintuple q is processed some quintuples may be pushed to the front of the queue when local successors of the state from the quintuple q are generated. Let us denote by R the set of all quintuples that occur in the part of the queue BBLQ that precedes the quadruple q_1 during the execution of the algorithm. Note that there are only finitely many different quintuples that may occur in R . The algorithm dequeues quintuples from the queue BBLQ continuously and therefore, there is at least one quintuple that is dequeued from the queue infinitely many times. Let $I \subseteq R$ be the set of all quintuples that are inserted to and dequeued from the queue infinitely many times.

Let $p = (q', \text{prelevel}', \text{target}', \text{bl}', \text{abit}')$ be a quintuple from I that is dequeued from the queue BBLQ. The quintuple p may be generated from itself and inserted in the queue BBLQ as its own (not necessarily immediate) descendant only finitely many times (there are finitely many acyclic paths from q' to q'). If the quintuple was inserted in the queue infinitely many times then the state q' would be expanded by the nested procedure that bears the identification $[\text{target}', \text{bl}', \text{abit}']$ infinitely many times as well. Obviously, this is not possible due to the nested procedure identification stored at the state. Hence, we can define a partial ordering on quadruples in I as follows. If $a, b \in I$

and b is inserted in in the queue BBLQ as a (not necessarily immediate) descendant of a infinitely many times, then $a < b$. Obviously, it cannot be that $b < a$ at the same time. Hence, we can choose a minimal member r in I . We know that r was inserted in the queue BBLQ infinitely many times, but only as descendant of finitely many quintuples that were processed only finitely many times. Obviously, this is not possible and is the needed contradiction. So, if the algorithm is not terminated by reporting the presence of an accepting cycle, the procedure CHECK-BL-EDGES3 dequeues the quintuple q_1 eventually. ■

Lemma 6.2. *If there is an accepting cycle in the examined graph then the algorithm will finish by reporting the presence of an accepting cycle eventually.*

Proof: Let us consider the level from which emanate the maximal depth back-level edges of shallowest accepting cycles and let us suppose that the procedures CHECK-BL-EDGES3 were initiated after the synchronization of the procedure BFS-BASED-CYCLE-DETECTION on that level. Further, let us suppose that the algorithm did not finish or finished without reporting the presence of an accepting cycle. Note that in the latter case all quintuples inserted in queues BBLQ must have been dequeued.

Let $[T, b, a]$ be the maximal nested procedure identification stored at a state of one of the shallowest accepting cycles. Let the cycle be $p_0, \dots, p_{n-1}, p_n = p_0$. Note that the maximal value exists even in the case of infinite run of the algorithm (bl is limited by corresponding $nmbrbl$, otherwise an accepting cycle is reported). Note also that T is greater or equal to any source state of a back-level edge on the cycle.

According to our assumptions and Lemma 6.1 we know that any quintuple that is inserted in a queue BBLQ by the nested procedure $[T, b, a]$ is eventually dequeued. Naturally, this means that all the states on the cycle are expanded by at least one nested procedure. Nevertheless, this also means that all the states on the cycle were expanded at least once by the nested procedure $[T, b, a]$ and so the nested procedure identification $[T, b, a]$ is stored at each state on the cycle. From which we can conclude that a equals to $true$ ($[T, b, a]$ is the maximal value and an accepting state on the cycle was processed). Now it remains to consider the situation when a destination state of some maximal depth back-level edge on the cycle was expanded by the procedure $[T, b, a]$. Since the state is a destination state of a maximal depth back-level edge, we have $prelevel = Level - 1$. Thus the identification of the nested procedure must have been changed to $[T, b + 1, false]$ before that state was expanded. $[T, b + 1, false]$ is obviously greater

than $[T, b, a]$ and so it is in contradiction to the maximality of $[T, b, a]$. So, if there is an accepting cycle in the examined graph, then the algorithm terminates by reporting the presence of an accepting cycle. ■

Lemma 6.3. *If the algorithm reports the presence of an accepting cycle then there is at least one accepting cycle in the examined graph.*

Proof: Suppose that the quintuple dequeued in the iteration in which the presence of a cycle was reported is $(q, \text{prelevel}, \text{target}, \text{bl}, \text{abit})$. Obviously, if such a quintuple is dequeued from a queue BBLQ then there is a corresponding path from the state target to the state q in the examined graph. Let $\pi = \text{target} = p_{n+1}, p_n, \dots, p_2, p_1, p_0 = q$ be the path.

There are two reasons for which the algorithm may report the presence of an accepting cycle. In the case that $q = \text{target}$ the accepting cycle is supported directly by the path π (note that π has to contain an accepting state as $\text{abit} = \text{true}$). In the other case ($\text{bl} = \text{nمبرbl} + 1$) we have that there are $\text{nمبرbl} + 1$ current back-level edges in the path π such that there is at least one accepting state between any of these back-level edges. Since there are only nمبرbl current back-level edges on the currently explored level there are at least two states p_j, p_k in the path π from which the same back-level edge emanates. Since $p_j = p_k$, the path from the state p_j to the state p_k supports the presence of an accepting cycle in the examined graph. ■

Lemma 6.4. *If the examined graph is finite then the algorithm terminates.*

Proof: If there is a cycle in the examined graph then according Lemma 6.2 the algorithm terminates by reporting the presence of an accepting cycle. We only show that the algorithm terminates if there is no accepting cycle in the graph. Let us suppose that any procedure CHECK-BL-EDGES3 initiated from any procedure BFS-BASED-CYCLE-DETECTION finishes eventually. In such a case we can conclude that the algorithm finishes eventually from Lemma 3.8. Thus it remains to prove that if there is no accepting cycle in the graph then all procedures CHECK-BL-EDGES3 finish eventually.

We prove this by showing that each quintuple can be inserted in a queue BBLQ only finitely many times. Obviously, a quintuple $(q, \text{prelevel}, \text{target}, \text{bl}, \text{abit})$ can be inserted in a queue BBLQ only if there is a path from the state target to the state q . Moreover, if a quintuple is inserted in the queue due to a path, it is never inserted in the

queue due to the same path again. To complete the proof we show that if a quintuple is inserted to a queue BBLQ due to a path $\text{target} = r_0, \dots, r_m, q$ then r_0, \dots, r_m is an acyclic path. Suppose it is not. Then we have that there are i, j such that $0 \leq i < j \leq m$ and $r_i = r_j$. Consider the situation when the nested procedure originating at the state target visits the state r_i . If the nested procedure is allowed to expand the state r_i and it visits later on the state r_j due to a path r_i, r_{i+1}, \dots, r_j then its nested procedure identification is less or equal to the nested procedure identification stored at the state. This is because the path r_i, r_{i+1}, \dots, r_j is a non-accepting cycle and we can see from the pseudo-code of the procedure CHECK-BL-EDGES3 that the identification of a nested procedure cannot increase unless the nested procedure passes through an accepting state. Hence, the nested procedure is not allowed to expand the state r_j and no successors in the path (including the state q) are inserted in the queue BBLQ. Hence a quintuple can be inserted in a queue BBLQ only due to a acyclic path from the target of the nested procedure to the explored state. Obviously, there are only finitely many acyclic paths (each being finite) between any two states of the graph. This gives us needed bound on the number of quintuple insertions in the queue. ■

7 Counterexample Generation

Model checking algorithms should be able to provide the user with a counterexample in the case the verified property is violated. In general, the computed counterexamples can be quite long which might make it difficult to locate an error. Thus computing the shortest possible counterexample greatly facilitates the debugging process. In this subsection we present a technique to generate short counterexamples using the algorithm BFS-BASED-CYCLE-DETECTION.

A counterexample consists of two parts: an *accepting cycle* and a *path* that reaches it from the initial state of the graph. In the standard Nested DFS algorithm the counterexample is simply generated by following the depth first search stacks: the DFS stack of the nested search is used to reconstruct the accepting cycle while the DFS stack of the primary search gives a path to it. However, in our algorithm we do not have any DFS stacks, hence a different approach has to be considered. Recall that our algorithm uses, similarly to the Nested DFS algorithm, two search phases.

The idea is quite simple. We use additional data structure for the primary phase of the algorithm to keep the shortest path from a state to the initial state of the graph and

an additional data structure for the secondary phase of the algorithm to record a trace of the path followed by nested procedures. The structures are nothing more than the standard *parent graphs*.

More precisely, during the primary phase of the algorithm we store to each state v that has not been visited yet the value of its immediate predecessor p in the primary search (called primary parent). Note that it is assigned only once during the whole computation. During the secondary phase of the algorithm we store to each state its immediate predecessor in the nested search (called secondary parent). The secondary parent is assigned every time a nested procedure is allowed to pass through the state. In both cases, the parents stored at states induce edges in the corresponding parent graph. Now we show how one can traverse the primary and secondary parent graphs to reconstruct the counterexample.

The accepting cycle part of a counterexample is generated using the secondary parent graph. Once the existence of an accepting cycle in the graph is detected, one could be tempted to follow the DFS parents back to and through the cycle. However, due to the fact that several nested procedures can be run in parallel, a state can be expanded several times by different nested procedure each time possibly changing its secondary parent. As a result, the secondary parent graph may not contain a cycle even though the presence of an accepting cycle is reported. A possible situation is exemplified in Figure 10. Suppose the *first* nested procedure starts from the state F and the secondary parent graph is build as the procedure continues (see Figure 10.a). Suppose that after the nested procedure passes through the state C the *second* nested procedure which has been started from the state A visits the state C before the first nested procedure actually closes the cycle. Further suppose that $A > F$. When exploring the state C by the second procedure, the secondary parent for C is reset to point to the state A , hence the possible cycle in the secondary parent graph is interrupted (see Figure 10.b).

To solve this problem we assign to the nested procedure that detected the cycle a new (highest) identification and we re-execute it independently. In our case given in Figure 10 we start the nested procedure from the state F once more with identification $[F', 0, false]$ where $F' > F$ and $F' > A$. The second nested procedure originating from the state A is not performed at all.

During counterexample generation the manager workstation is used as a “collector” that all the workstations participating in the generation send information to. Note that states forming the counterexample are spread across network according to the partition

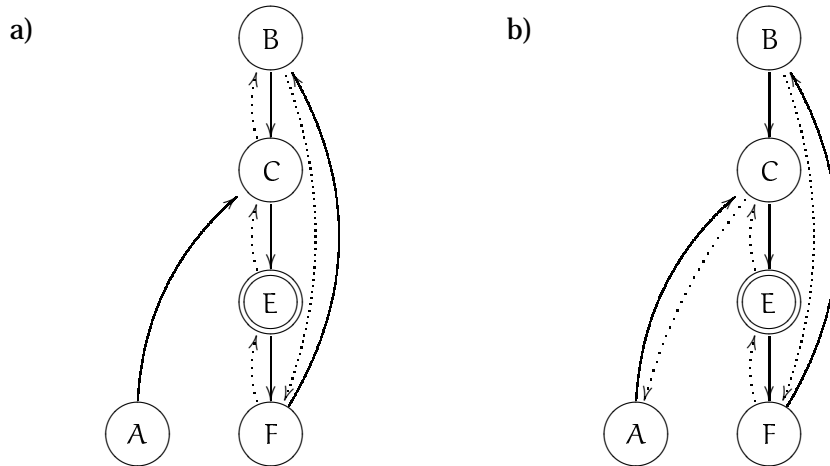


Figure 10: Interrupting a cycle in the secondary parent graph

function. The counterexample is generated in two steps. In the first one the secondary parent graph is traversed starting at the state where the cycle was detected. When traversing the secondary parent graph the visited states are marked in order to discover the cycle. Once an already marked state is visited, the cycle can be reconstructed from the vertices that have been sent and saved on the manager workstation. Moreover, the manager is able to determine the state v with the smallest distance from the initial state on the cycle. In the second step of the generation the primary parents are traversed from the state v back to the initial state of the examined graph. After finishing the second step, the whole counterexample can be put together using the information stored on the manager workstation.

A significant positive feature of the algorithm is related to the length of counterexamples it provides. Since the algorithm is primarily based on breadth first search exploration, the counterexamples tend to be short. See the chapter on experiments for more details.

8 Partial Order Reduction

In this subsection we suggest how to combine the partial order reduction with the distributed memory algorithm `BFS-BASED-CYCLE-DETECTION`.

The primary task of the algorithm is to check whether the examined graph contains (accepting) cycles. However, to accomplish the task the algorithm has to solve the state space generation problem as well, i.e. it has to compute the explicit representation of

the verified graph from the given implicit one. Recall that if the partial order reduction technique is not considered, then the graph is implicitly given by the functions F_{init} and F_{succs} . If the reduction is involved then the graph is given by the functions F_{init} and F_{ample} .

We can provide the algorithm with the function F_{ample} only if we are able to check the ample conditions **C0**, **C1**, and **C2** during the generation of the graph. However, checking these conditions can be done in the same way as it is done in the standard sequential state space generation case. Naturally, if the function F_{succs} is replaced by the function F_{ample} the algorithm **BFS-BASED-CYCLE-DETECTION** has to guarantee that the condition **C3** is satisfied as well, otherwise the reduction is not correct.

Our aim is to develop a counterpart of the condition **C3-dfs**. The condition **C3-dfs** is used instead the condition **C3** in depth first search based generation of the state space. It exploits the depth first search stack to ensure that at least one state is fully expanded on each cycle in the product automaton graph. Unfortunately, we do not have search stack in the distributed memory breadth first based algorithm so we cannot use this condition, but have to suggest a different one. However, if we realize the fact that a necessary condition for closing a cycle in breadth first search is that the state closing the cycle has to be the destination state of a back-level edge, we are actually done. The newly suggested replacement of the condition **C3** used for the breath first search based generation of the state space is the following.

C3-bfs *If a state s is not fully expanded, then $ample(s)$ does not contain an action that if taken, produces a back-level edge.*

The condition requires each source state of a back-level edge to be fully expanded. Recall that a back-level edge is detected when its destination state is explored. If the source state of a detected back-level edge is local, we can easily enqueue the source state of the edge to the queue CLQ as a state to be fully re-expanded. However, it might happen that the source state and destination state of a back-level edge are owned by different workstations. In such a case a message requesting full expansion of the source state of the back-level edge has to be sent to the workstation owning the source state. Obviously, this makes the distributed memory back-level edge detection algorithm quite complicated. In particular, each local queue CLQ no more keeps only the states to be explored but it keeps also the states to be fully re-expanded. Since the states to be fully re-expanded cannot be precomputed in the same way as the states to be ex-

```

1 proc DISTRIBUTED-MEMORY-BL-EDGE-DETECTION-POR(WorkstationId)
2    $CLQ = \emptyset$ ;  $NLQ = \emptyset$ ;  $Visited = \emptyset$ ;  $Level = 0$ 
3    $initstate := F_{init}()$ ;  $finished := false$ 
4   if (WorkstationId = Owner( $initstate$ ))
5     then enqueue( $NLQ$ , ( $-$ ,  $initstate$ , ample)) fi
6   while ( $\neg finished$ ) do
7     swap( $NLQ$ ,  $CLQ$ )
8     while ( $\neg Synchronize(finished := (all\ NLQ = \emptyset))$ ) do
9       if ( $CLQ \neq \emptyset$ )
10        then ( $p, v, w := dequeue(CLQ)$ )
11          if ( $v \notin Visited$ )
12            then  $Visited := Visited \cup \{v\}$ ;  $v.fully := false$ ;  $d(v) := Level$ 
13              foreach  $t \in F_{ample}(v)$  do
14                if (Owner( $t$ )  $\neq$  WorkstationId)
15                  then SendTo(Owner( $t$ ), enqueue( $NLQ$ , ( $v, t$ )))
16                  else enqueue( $NLQ$ , ( $v, t$ ))
17                fi od
18              else if ( $d(v) < Level$ )
19                then Print("Back-level edge ( $p, v$ )")
20                if (Owner( $p$ )  $\neq$  WorkstationId)
21                  then SendTo(Owner( $p$ ),
22                    enqueue( $CLQ$ , ( $-$ ,  $p$ , fully)))
23                  else enqueue( $CLQ$ , ( $-$ ,  $p$ , fully))
24                fi fi fi
25              if ( $w = fully$ )
26                then foreach  $t \in (F_{succs}(v) \setminus F_{ample}(v))$  do
27                  if (Owner( $t$ )  $\neq$  WorkstationId)
28                    then SendTo(Owner( $t$ ), enqueue( $CLQ$ , ( $v, t$ , ample)))
29                    else enqueue( $CLQ$ , ( $v, t$ , ample))
30                  fi od
31                 $v.fully := true$  fi fi od
32             $Level = Level + 1$ 
33   od end

```

Figure 11: Distributed memory algorithm for back-level edge detection with POR

plored (i.e. when exploring the previous level), the queues CLQ have to be replenished with these states during the processing of states from the queues.

This requires a few modifications of the algorithm. Firstly, states stored at the queues CLQ, NLQ are enriched with a flag whether they are the states to be explored or the states to be fully re-expanded. If a state dequeued from the queue CLQ is a state to be fully re-expanded only those successors of the state are generated that were not generated when the state was expanded for the first time. Secondly, the processing of a queue CLQ cannot finish when the queue is empty since some states to be fully re-expanded may still arrive from other workstations. Thus the distributed termination detection mechanism has to be employed to detect that the processing of the states from all the queues CLQ has already finished and next level can be processed.

Finally, each explored state is associated a bit flag `fully` whether it was or was not fully re-expanded during the state space generation. This flag is obviously useless for the back-level edge detection, however it is needed if the procedure is meant to be used in the algorithm `BFS-BASED-CYCLE-DETECTION`. In such a case the algorithm has to ensure that the same successors of a state are generated during the primary and secondary phase of the algorithm, otherwise the reduction in the size of the product automaton is lost. Thus when expanding a state in the secondary phase of the algorithm the function F_{ample} is used if the flag is not set to true. The function F_{succs} is used otherwise.

Just for clarity the pseudo-code of the procedure `DISTRIBUTED-MEMORY-BL-EDGE-DETECTION-POR` that can be used to detect back-level edges in the reduced product automaton graph is given in Figure 11.

The condition **C3-bfs** ensures that there is at least one fully expanded state on each cycle in the reduced product automaton graph. Nevertheless, since there may be many back-level edges that are not a part of a cycle, there may be also many fully expanded states that do not belong to any cycle. That is why we cannot expect the reduction to be as efficient as in the case of the standard condition **C3-dfs**. For more details on this see the experimental results given in the next chapter.

9 Experimental evaluation

The experimental version of the algorithm has been implemented in C/C++ using `mpich` implementation of the standard network-support library *MPI*. Contrary to our previous experimental work on parallel algorithms we have decided not to embed the algorithm

Problem	Model (M)	Verified property (φ)	$M \stackrel{?}{\models} \varphi$
elev1	Elevator	$G(r1 \implies (F(p1 \wedge o)))$	yes
elev2	Elevator	$G(r1 \implies (\neg p1 U(p1 U(p1 \wedge o))))$	no
elev3	Elevator	$G(r0 \implies (\neg p0 U(p0 U(\neg p0 U(p0 U(p0 \wedge o))))))$	yes
elev4	Elevator	$G(r1 \implies (\neg p1 U(p1 U(\neg p1 U(p1 U(p1 \wedge o))))))$	yes
elev5	Elevator	$G(r2 \implies (\neg p2 U(p2 U(\neg p2 U(p2 U(p2 \wedge o))))))$	yes
elev6	Elevator	elev3 \wedge elev4	yes
elev7	Elevator	$FG(p1)$	no
lead1	Leader election	$F(\text{elected})$	yes
lead2	Leader election	$FG(\text{oneleader})$	yes
lead3	Leader election	noleader U oneleader	yes
lead4	Leader election	$F(\text{moreleaders})$	no
pet1	Peterson	$G(p0cs \implies F(\neg p0cs))$	yes
pet2	Peterson	$G((\neg p0cs) \implies F(p0cs))$	no
pet3	Peterson	$GF(\text{someoneincs})$	yes
pet-E0	Peterson-error	state space generation	—
pet-E1	Peterson-error	$G(p0cs \implies F(\neg p0cs))$	yes
pet-E2	Peterson-error	$G((\neg p0cs) \implies F(p0cs))$	no
pet-E3	Peterson-error	$GF(\text{someoneincs})$	no
phi1	Philosophers	$GF(\text{eat0})$	no
phi2	Philosophers	$G(\text{one0} \implies F(\text{eat0}))$	no
phi3	Philosophers	$GF(\text{someoneeats})$	no
phi-L1	Philosophers-left	$GF(\text{eat0})$	no
phi-L2	Philosophers-left	$G(\text{one0} \implies F(\text{eat0}))$	no
phi-L3	Philosophers-left	$GF(\text{someoneeats})$	yes
rte1	RT Ethernet	$G(r0 \implies (\neg e U (e U (\neg e \wedge (rt0 R \neg e))))))$	yes
rte2	RT Ethernet	$G(w0 \implies (\neg e U (e U (\neg e \wedge (rt0 R \neg e))))))$	no
rte3	RT Ethernet	$G(r0 \implies (rt0 R \neg e))$	no
rte4	RT Ethernet	$GF(\text{nrt0})$	yes
rte5	RT Ethernet	$GF(\text{rt0})$	no

Table 1: Verified model checking problems

into an existing tool, e.g. SPIN. The main reason was that the SPIN has been built basically for sequential enumerative LTL model-checking and it has proven to be quite difficult to be modified. Therefore, we implemented the algorithm directly as a prototype (not using all the very efficient implementation techniques and optimizations encountered in SPIN). After all, the primary reason for the implementation was to be able to make a preliminary experimental evaluation of our new approach to the distributed on-the-fly back-level edge based cycle detection.

In the distributed implementation built on the MPI library each workstation has its own unique number that is used for its identification. The workstation with the lowest identification number (the manager) performs all the management related to the distributed computation, in particular the manager starts the synchronization protocol (during which the workstations synchronize and exchange all the necessary information) and the distributed termination detection protocol.

The partition function used for slicing the state space was chosen to partition the graph evenly without optimizations either for perfect load balancing or for minimal communication complexity. However, the load was reasonably well balanced in all performed experiments and the communication overhead caused by “cross-edges”(successive vertices placed on different workstations) has not influenced the comparative value of the experiments. For more efficient communication between workstations we did not send separate messages, but sent them in packets of pre-specified size. The optimal size of a packet depends on the network connection and the underlying communication structure. In our case we have achieved the best results for packets of size around 100 single messages.

For our set of experiments we chose several standard software protocols. All systems are parametrized: the *Philosophers* problem by the number of philosophers, the *Elevator* by the number of floors served, the *Peterson*, *Leader election*, and the *Real Time Ethernet* by the number of participating processes. The LTL formulas specifying the checked properties are given in Table 1 together with the answer to the model-checking problem.

9.1 Back-Level Edge Distribution

We start by a few experiments to measure the frequency and distribution of the back-level edges in product automaton graphs. In particular, we measured the maximal breadth first search level that was reached during the verification, the total number of revealed back-level edges, and the number of back-level edges that were tested for

Problem ($M \not\models \varphi$)	Maximal Level	All Back-Level Edges	Explored Back-Level Edges
elev2(10)	50	296469	118510
elev7(10)	24	18291	18291
lead4(5)	91	40	40
pet2(4)	31	115071	38173
pet-E3(4)	7	72	24
pet-E3(5)	7	120	40
phi2(10)	5	572	66
phi-L2(10)	5	581	65
rte2(9)	206	146101	23136
rte5(9)	75	74	23

Problem ($M \models \varphi$)	Maximal Level	All Back-Level Edges	Explored Back-Level Edges
elev1(10)	64	302094	101647
elev3(10)	64	381735	175392
elev4(10)	64	372604	161953
elev6(10)	64	3373453	655523
lead3(5)	119	0	0
pet-E1(4)	23	6357	0
pet1(4)	55	121920	313
phi-L3(10)	29	213733	44287
rte1(9)	446	169095	0
rte4(9)	446	497755	164469

Figure 12: Number of Back-Level Edges and Levels

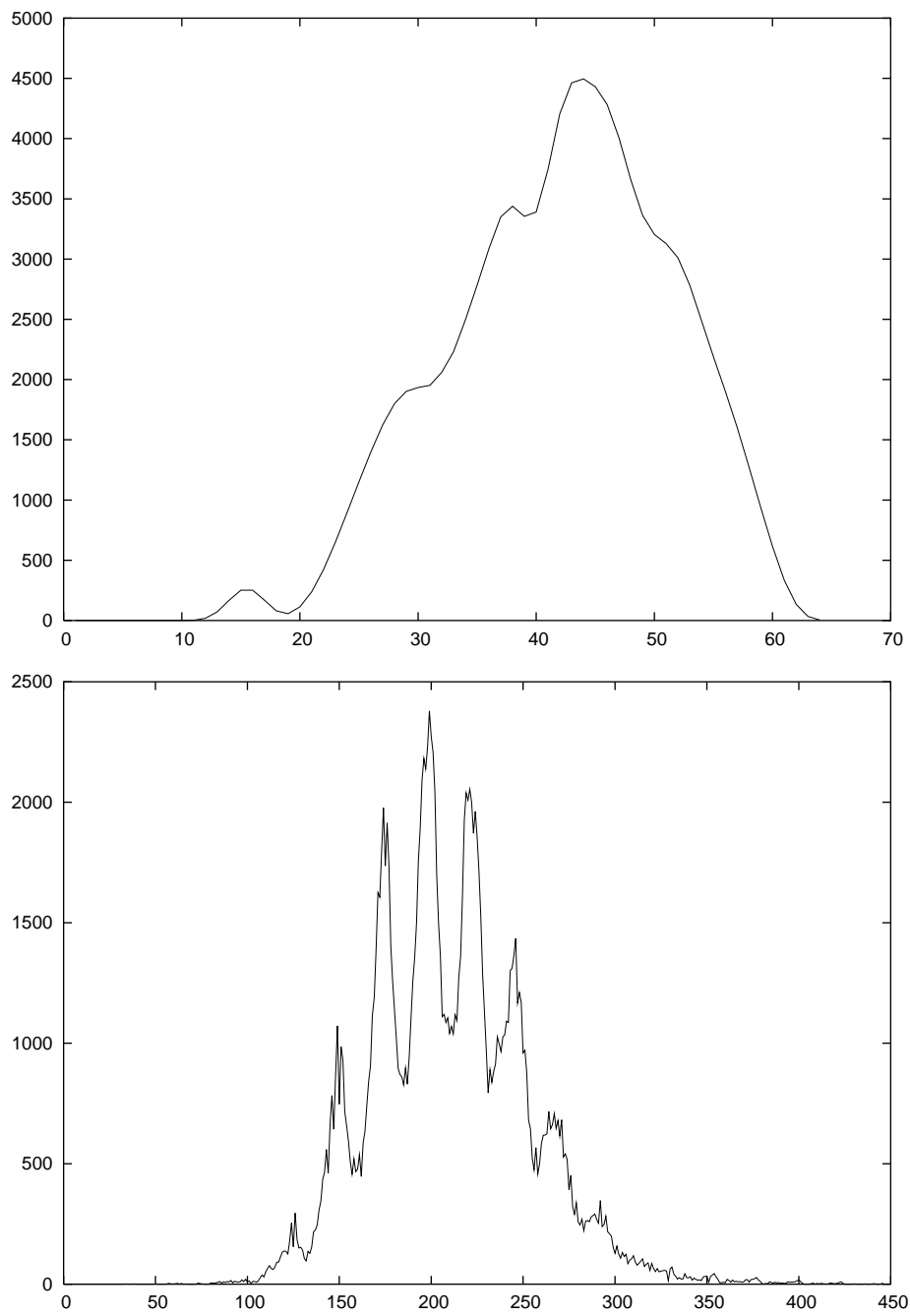


Figure 13: Explored Back-Level Edge Distribution for **elev1(10)** and **rte4(9)**

being the part of an accepting cycle (hence the back-level edges occurring in the fully and partially accepting components of the graph). All the values we measured are given in tables in Figure 12. Note that for the cases where the property is not satisfied the maximal level reached during the verification corresponds to the level where the accepting cycle was discovered (as the rest of the graph was not explored).

We were also interested in the distribution of the back-level edges in product automaton graphs. We found out that the distributions are quite similar for most cases and thus we depict only two of them in Figure 13.

There are two noteworthy points that could be drawn from these experiments. Firstly, if there is an accepting cycle in the product automaton graph, then it seems that the cycle is quite often discovered by examining just a few back-level edges found in early stage of the computation (very often on the first level containing back-level edges). However, there are models and verified formulas that do not stick to this observation at all as demonstrated, e.g., by the elev2 and elev7 model checking problems. Secondly, there are product automaton graphs that contain so few back-level edges (possibly none) that the suggested distributed memory algorithm can perform the formal verification of an LTL formula as efficiently as it would perform a distributed state space generation. We consider this fact to be quite interesting.

9.2 Scalability

In this subsection we measure how the implemented algorithm scales. In particular, we are interested in the runtimes and memory consumption made on a workstation if various numbers of workstations are involved in the computation. Selected experimental results concerning the scalability are given in Figure 15. Time values are reported in seconds while memory values in megabytes. The corresponding graphs are given in Figure 14. It can be seen that the more workstations participate the computation the less time is needed to perform the verification. However, if the verified model is too small, the runtimes increase with the number of used workstations. This is because the time needed to initialize the network layer dominates the time needed to perform the verification. The same holds for memory, i.e. the more workstations participate the computation the less are the memory requirements. An exception situations are those where the memory load caused by the table of visited states is too small to dominate the memory overhead caused by the network layer.

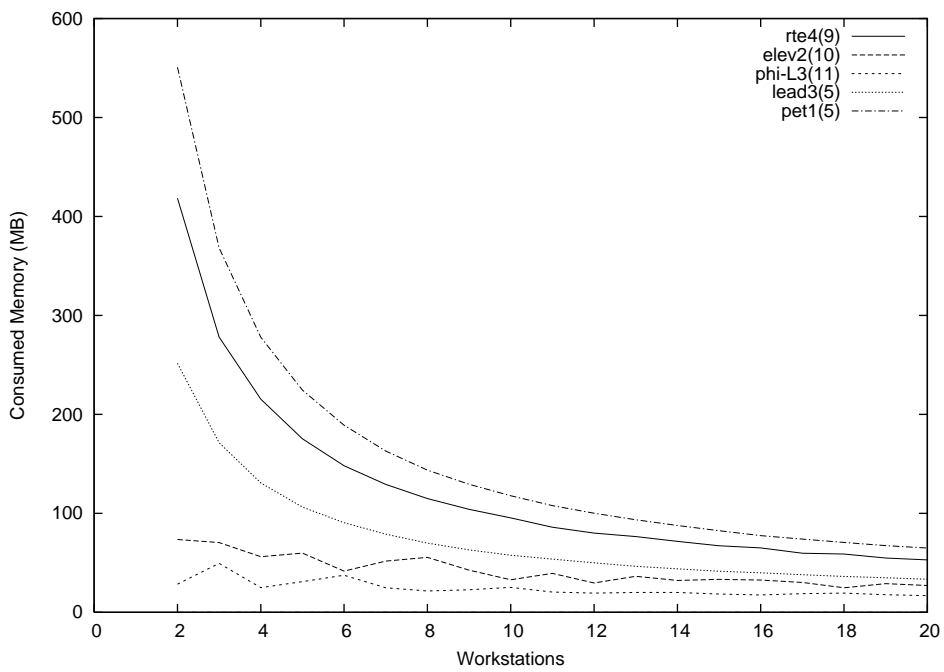
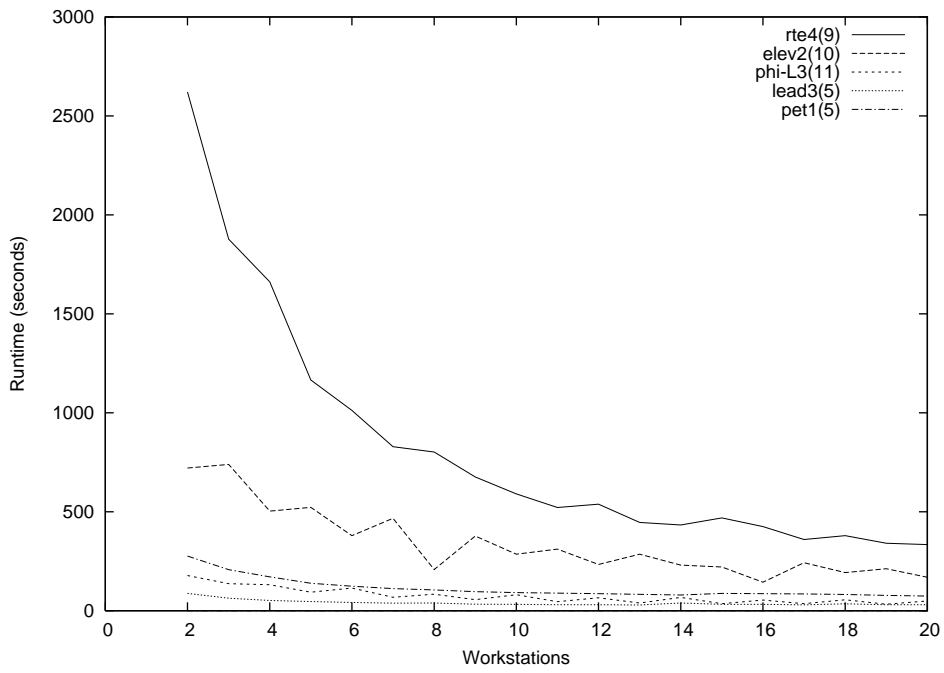


Figure 14: Back-Level Edge based cycle detection scalability graphs

Problem	5 Workstations		10 Workstations		15 Workstations		20 Workstations	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
elev1(10)	752	42.3	433	25.3	325	26.0	223	22.1
elev2(10)	522	59.9	285	32.9	220	33.2	169	27.1
lead3(5)	46	106.3	32	57.7	33	41.5	30	33.5
rte4(9)	1166	175.3	590	95.4	469	67.2	334	52.9
pet-E3(4)	2	7.9	4	8.0	7	8.2	8	8.5
phi-L3(11)	95	30.6	81	25.3	36	18.5	49	16.8

Figure 15: Back-Level Edge based cycle detection scalability

9.3 Counterexample Generation

The table in Figure 16 shows the lengths of the generated counterexamples for selected model checking problems. The column “Counterexample Length” gives the lengths of the counterexamples as produced by the new algorithm while the column “Counterexample Length (NDFS)” gives the lengths of counterexamples as produced by the Nested DFS algorithm. It can be seen that our algorithm produces significantly shorter counterexamples than the standard Nested DFS does. On the other hand, the time needed by the Nested DFS algorithm to discover and generate a counterexample is shorter in general. The difference is obviously caused by the fact that our algorithm explores many more states in comparison to the Nested DFS algorithm before it discovers an accepting cycle. Time in seconds needed to generate the counterexamples is given in the table as well. Note the difference in runtime needed by the Nested DFS algorithm to generate the counterexample for the problem phi1(15). The longer reported value give the runtime needed by the algorithm when the counterexample generation procedure had not been optimized.

9.4 C3-BFS Partial Order Reduction

Other experiments were performed to evaluate the partial order reduction based on the condition **C3-bfs**. See the table in Figure 17. We measured the number of states in the full graph (the column “no POR”) and the number of states in the reduced graph (the column “POR”). In addition, we measured the number of states in the graph that was reduced by the standard partial order reduction as given in [5] (the column “POR-DFS”).

Problem	Counterexample Length	Counterexample Length (NDFS)	Time	Time (NDFS)
elev2(9)	85	323	136	1
elev2(10)	65	356	527	1
lead4(4)	74	74	3	1
lead4(5)	92	92	69	1
pet2(3)	39	63	1	1
pet2(4)	85	128	7	1
pet3(4)	12	215	1	1
pet3(5)	12	625	1	1
phi1(12)	5	4116	1	1
phi1(13)	5	18579	1	1
phi1(14)	5	18579	1	2
phi1(15)	5	120730	1	14 (635)
rte2(9)	207	17478	54	4
rte2(10)	244	60520	76	16

Figure 16: Counterexample lengths and generation times

This value was measured using the standard sequential DFS algorithm (“—” means the algorithm exceeds the available memory). Note that a reduction was achieved only for the Leader election and the Peterson models which we believe is due to the immature implementation of the ample set selection in DiVinE. While the reduction based on **C3-bfs** condition was the same as the standard partial order reduction in the case of Leader election model, it was worse in the case of Peterson model. Hence, we can conclude that there are models for which the **C3-bfs** partial order reduction works and there are also models for which it works only marginally or even does not work at all.

Finally, we measured the impact of the suggested partial order reduction on the counterexample generation. The table in Figure 18 shows counterexample lengths (“CE Length”), time needed to generate them (“Time”), and the deepest levels that were reached during the verification (“Level”). It can be seen that if the partial order reduction is employed then the shallowest accepting cycles may get slightly deeper. This may increase the number of states that are explored by the algorithm before an accepting cycle is discovered as well as the time needed to discover and generate the counterex-

Problem	Number of states no POR	Number of states POR	Number of states POR-DFS
elev2(10)	1113062	1113062	1113062
lead1(4)	110537	60681	60681
lead1(5)	3629011	1487891	1487891
pet1(3)	2429	2335	1815
pet1(4)	132104	130965	103963
pet1(5)	9516142	9478643	—
phi2(12)	847653	847653	847653
phi2(13)	2468548	2468548	—
rte1(10)	5759277	5759277	5759277

Figure 17: **C3-BFS** Partial Order Reduction

Problem	CE Length	CE Length	Runtime	Runtime	Level	Level
	No POR	POR	No POR	POR	No POR	POR
lead2(4)	74	74	3	3	73	73
lead2(5)	92	92	69	61	91	91
pet2(3)	39	38	1	1	20	21
pet2(4)	85	89	7	394	31	33
pet3(4)	12	14	1	1	7	9
pet3(5)	12	15	1	1	7	10

Figure 18: Partial order reduction and counterexample generation

ample (see the row **pet2(4)**). Nevertheless, the impact of the partial order reduction on the lengths of counterexamples is generally minimal.

10 Conclusion

We propose a parallel LTL model checking algorithm for alleviating the state explosion problem by distribution of the state space. The algorithm is based on a combination of the breadth-first-like state space generation with a back-level edge controlled depth-first-like search for accepting cycles. The aim of this paper was to describe the idea of the algorithm in detail including counterexample generation and partial order reduction

utilization. Experimental results obtained with a prototype implementation were also given.

In the LTL model checking applications, the existence of an accepting cycle indicates a failure of the property. In such a case, it is essential that the user is given an accepting cycle as a counterexample. The counterexample should be as short as possible, to facilitate debugging. The advantage of our approach is that thanks to the breadth-first search character of the computation the generated counterexample is much shorter in comparison with those computed by a depth-first search based algorithms. On the other hand, on extremely large systems a depth-first search approach may discover an error while a breadth-first search may fail to do so (even if ran in a distributed environment). The reason is that depth-first search usually need not explore so many states to find the error.

A very positive feature of the algorithm is its behaviour on graphs with a small number of back-level edges. In such cases the run of our algorithm for the full LTL model checking practically equals to the reachability analysis. There were several models confirming this feature. On the other hand, a drawback shows up when the graph contains many back-level edges. Moreover, if the graph does not contain an accepting cycle, it must be fully explored, and frequent revisiting of states causes the time of the computation to be much longer than the time of a simple reachability analysis.

Another feature of our algorithm is that it works on-the-fly. The on-the-fly approach to model checking has proven its superiority over global approach in many case studies.

As for the partial order reduction our algorithm works unexpectedly well. The achieved reduction was practically the same as for the depth first search based reduction. It is partly caused by the poor implementation of the reduction technique and we expect our algorithm to work a bit worse in comparison with depth first like algorithms when the implementation gets optimized.

There are several known approaches to distribution and/or parallelization of the LTL model-checking problem. A distributed implementation of the SPIN [10] LTL model-checker restricted to model-checking safety LTL properties (those properties that do not involve cycle detection and hence can be reduced to the *reachability* problem) was described in [11]. We extended the work of [11] to the full distributed LTL model-checking in SPIN in [1]. The idea behind was to employ additional data structures that allow to ensure the global DFS postorder on at least the most critical states. The disadvantage of this algorithm is that it performs only one nested search at a time. In [2], the

problem of LTL model-checking is reduced to detecting negative cycles in a weighted directed graph. Recently, in [3] another algorithm for distributed enumerative LTL model-checking has been proposed. The algorithm implements the enumerative version of the symbolic “One-Way-Catch-Them-Young ” algorithm [9]. The algorithm shows in many situations a linear behavior, however it is not on-the-fly, hence the whole state space has to be generated. The algorithm [3] shows better performance than our algorithm on some systems without faulty runs, while our algorithm is superior in finding bugs. For this reason the two algorithms could be meant not to replace but to complement each other.

11 Complexity

The primary phase of the algorithm actually performs breadth first search of the graph. Thus the time complexity of each locally performed procedure BFS-BASED-CYCLE-DETECTION is $\mathcal{O}(m + n)$, where m is the number of transitions and n is the number of states in the verified product automaton graph. If we employ the partial order reduction then each state of the graph may be expanded twice. However, this does not increase the asymptotic complexity.

It is the procedure CHECK-BL-EDGES that mostly adds to the overall complexity of the algorithm. For a given source state of a back-level edge the procedure traverses the graph along all paths originating at the state. This is in $(\mathcal{O}(n!))$ In the worst case there can be n source states of back-level edges, hence the overall complexity of the second phase of the algorithm is $\mathcal{O}(n \times (n!))$. When improving the behaviour of the algorithm by the procedures CHECK-BL-EDGES2 or CHECK-BL-EDGES3 the theoretical complexity of the algorithm improves as well. If these procedures are considered each nested procedure may expand a single state up to n or $2n$ times, respectively. Hence, the overall complexity of the procedures gets in $\mathcal{O}(n^2 \times (m + n))$.

Since the time complexity of the secondary phase of the algorithm dominates the complexity of the primary phase, the time complexity of the algorithm is in $\mathcal{O}(n^2 \times (m + n))$.

As for the space complexity we have to sum up the space needed to store all the necessary information at a single state. In the worst case we store at a single state its depth, the depth of last nested procedure that passed through the state, the last nested procedure identification (target state, number of passed bl, accepting bit), the primary

and secondary phase parents, and a bit flag to keep the information about the full expansion of the state. It may seem that the space needed to store these values is quite large, however it is not so in practice. The crucial fact is that the additional space needed is constant for a single state. Thus it may be easily compensated by involving more workstations in the computation.

Finally, we should estimate the number of messages the algorithm needs to exchange during its execution. If we do not count messages exchanged when a counterexample is reconstructed and messages exchanged due to distributed memory termination detections (within which the numbers of current back-level edges and states in the queues NLQ are collected), we can state that there are at most as many messages as there are expansions of a state, thus $\mathcal{O}(n^2 \times (m + n))$.

References

- [1] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'2001)*, volume 2057 of LNCS, pages 200–216, Toronto, Canada, 2001. Springer-Verlag.
- [2] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *Proceedings of Foundations of Software Technology and Theoretical Computer Science (FST-TCS'01)*, volume 2245 of LNCS, pages 96–107. Springer Verlag, 2001.
- [3] I. Černá and R. Pelánek. Distributed explicit fair cycle detection. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop*, volume 2648 of LNCS, pages 49–73. Springer-Verlag, 2003.
- [4] I. Černá and R. Pelánek. Relating hierarchy of temporal properties to model checking. In *Mathematical Foundations of Computer Science (MFCS)*, volume 2747 of LNCS, pages 318–327. Springer, 2003.
- [5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

- [6] T. H. Cormen, Ch. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT, 1990.
- [7] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1:275–288, 1992.
- [8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proc. Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, 1998.
- [9] R. Hojati, H. Touati, R. P. Kurshan, and R. K. Brayton. Efficient omega-regular language containment. In G. von Bochmann and D. K. Probst, editors, *Computer Aided Verification: Proc. of the Fourth International Workshop CAV'92*, pages 396–409. Springer, Berlin, Heidelberg, 1993.
- [10] G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
- [11] Flavio Lerda and Riccardo Sisto. Distributed-memory Model Checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of LNCS. Springer-Verlag, 1999.
- [12] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM journal on computing*, pages 146–160, Januar 1972.
- [13] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16–18 June 1986*, pages 332–344. IEEE Computer Society Press, Washington, DC, 1986.